Introduction

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Programming languages

- A language is a medium for communication

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions
- Originally, directly connected to architecture
  - Memory locations store values, registers allow arithmetic
  - Load a value from memory location M into register R
  - Add the contents of register R_1 and R_2 and store the result back in R_1
  - Write the value in R_1 to memory location M'

Programming languages

- A language is a medium for communication
- Programming languages communicate computational instructions
- Originally, directly connected to architecture
  - Memory locations store values, registers allow arithmetic
  - Load a value from memory location M into register R
  - Add the contents of register R_1 and R_2 and store the result back in R_1
  - Write the value in R_1 to memory location M'
- Tedious and error-prone

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries
- Express such ideas in the programming language
  - Translate "high level" programming language to "low level" machine language
  - Compilers, interpreters

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions / procedures, recursion
  - Aggregate data structures — arrays, lists, dictionaries
- Express such ideas in the programming language
  - Translate "high level" programming language to "low level" machine language
  - Compilers, interpreters
- Trade off expressiveness for efficiency
  - Less control over how code is mapped to the architecture
  - But fewer errors due to mismatch between intent and implementation

Styles of programming

- Imperative vs declarative

Styles of programming

- Imperative vs declarative
- Imperative
  - How to compute
  - Step by step instructions on what is to be done

Styles of programming

- Imperative vs declarative
- Imperative
  - How to compute
  - Step by step instructions on what is to be done
- Declarative
  - What the computation should produce
  - Often exploit inductive structure, express in terms of smaller computations
  - Typically avoid using intermediate variables
  - Combination of small transformations — functional programming

Imperative vs Declarative Programming, by example

- Add values in a list

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

- Describe the desired output by induction
  - Base case: Empty list has sum `0`
  - Inductive step: Add first element to the sum of the rest of the list

Imperative vs Declarative Programming, by example

- Add values in a list
- Imperative (in Python)

```
def sumlist(l):
  mysum = 0
  for x in l:
    mysum = mysum + x
  return(mysum)
```

- Intermediate values `mysum`, `x`
- Explicit iteration to examine each element of the list

- Declarative (in Python)

```
def sumlist(l):
  if l == []:
    return(0)
  else:
    return(l[0] + sumlist(l[1:]))
```

- Describe the desired output by induction
  - Base case: Empty list has sum `0`
  - Inductive step: Add first element to the sum of the rest of the list
- No intermediate variables

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`
- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

Imperative vs Declarative Programming, by example, . . .

- Sum of squares of even numbers upto `n`

- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Imperative vs Declarative Programming, by example, ...

- Sum of squares of even numbers upto `n`
- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Can code functionally in an imperative language!

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
            filter(even,
                   range(n+1)))))
```

Imperative vs Declarative Programming, by example, ...

- Sum of squares of even numbers upto `n`
- Imperative (in Python)

```
def sumsquareeven(n):
  mysum = 0
  for x in range(n+1):
    if x%2 == 0:
      mysum = mysum + x*x
  return(mysum)
```

- Can code functionally in an imperative language!
- Helps identify natural units of (reusable) code

- Declarative (in Python)

```
def even(x):
  return(x%2 == 0)

def square(x):
  return(x*x)

def sumsquareeven(n):
  return(
    sum(map(square,
          filter(even,
                 range(n+1)))))
```

Names, types, values

- Internally, everything is stored a sequence of bits

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data
- We impose a notion of type to create some discipline
  - Intepret bit strings as "high level" concepts
  - Nature and range of allowed values
  - Operations that are permitted on these values

Names, types, values

- Internally, everything is stored a sequence of bits
- No difference between data and instructions, let alone numbers, characters, booleans
  - For a compiler or interpreter, our code is its data
- We impose a notion of type to create some discipline
  - Intepret bit strings as "high level" concepts
  - Nature and range of allowed values
  - Operations that are permitted on these values
- Strict type-checking helps catch bugs early
  - Incorrect expression evaluation — like dimension mismatch in science
  - Incorrect assignment — expression value does not match variable type

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap

Abstract datatypes, object-oriented programming

- Collections are important
  - Arrays, lists, dictionaries
- Abstract data types
  - Structured collection with fixed interface
  - Stack is a sequence, but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queue using sorted or unsorted lists, or using a heap
- Object-oriented programming
  - Focus on data types
  - Functions are invoked through the object rather than passing data to the functions
  - In Python, `mylist.sort()` vs `sorted(mylist)`

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there

What this course is about

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, ...
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there
- ... and why new ones are still being created

Types

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

The role of types

- Interpreting data stored in binary in a consistent manner
  - View sequence of bits as integers, floats, characters, ...
  - Nature and range of allowed values
  - Operations that are permitted on these values

The role of types

- Interpreting data stored in binary in a consistent manner
  - View sequence of bits as integers, floats, characters, ...
  - Nature and range of allowed values
  - Operations that are permitted on these values
- Naming concepts and structuring our computation
  - Especially at a higher level
  - `Point` vs `(Float,Float)`
  - Banking application: accounts of different types, customers ...

The role of types

- Interpreting data stored in binary in a consistent manner
  - View sequence of bits as integers, floats, characters, ...
  - Nature and range of allowed values
  - Operations that are permitted on these values
- Naming concepts and structuring our computation
  - Especially at a higher level
  - `Point` vs `(Float,Float)`
  - Banking application: accounts of different types, customers ...
- Catching bugs early
  - Incorrect expression evaluation — like dimension mismatch in science
  - Incorrect assignment — expression value does not match variable type

Dynamic vs static typing

- Every variable we use has a type
- How is the type of a variable determined?

Dynamic vs static typing

- Every variable we use has a type
- How is the type of a variable determined?
- Python determines the type based on the current value
  - Dynamic typing — names derive type from current value
  - `x = 10` — `x` is of type `int`
  - `x = 7.5` — now `x` is of type `float`
  - An uninitialized name as no type

Dynamic vs static typing

- Every variable we use has a type
- How is the type of a variable determined?
- Python determines the type based on the current value
  - Dynamic typing — names derive type from current value
  - `x = 10` — `x` is of type `int`
  - `x = 7.5` — now `x` is of type `float`
  - An uninitialized name as no type
- Static typing — associate a type in advance with a name
  - Need to declare names and their types in advance value
  - `int x`, `float a`, ...
  - Cannot assign an incompatible value — `x = 7.5` is no longer legal

Dynamic vs static typing

- "Isn't it convenient that we don't have to declare variables in advance in Python?"
- Yes, but . . .

Dynamic vs static typing

- "Isn't it convenient that we don't have to declare variables in advance in Python?"
- Yes, but . . .
- Difficult to catch errors, such as typos

```
def factors(n):
  factorlist = []
  for i in range(1,n+1):
    if n%i == 0:
       factorlst = factorlist + [i]
  return(factorlist)
```

Dynamic vs static typing

- "Isn't it convenient that we don't have to declare variables in advance in Python?"
- Yes, but ...
- Difficult to catch errors, such as typos

```
def factors(n):
  factorlist = []
  for i in range(1,n+1):
    if n%i == 0:
        factorlst = factorlist + [i]   # Typo!
  return(factorlist)
```

- Empty user defined objects
  - Linked list is a sequence of objects of type `Node`
  - Convenient to represent empty linked list by `None`
  - Without declaring type of `l`, Python cannot associate a type after `l = None`

Types for organizing concepts

- Even simple type "synonyms" can help clarify code
  - 2D point is a pair `(float,float)`, 3D point is triple `(float,float,float)`
  - Create new type names `point2d` and `point3d`
  - These are synonyms for `(float,float)` and `(float,float,float)`
  - Makes intent more transparent when writing, reading and maintaining code

Types for organizing concepts

- Even simple type "synonyms" can help clarify code
  - 2D point is a pair `(float,float)`, 3D point is triple `(float,float,float)`
  - Create new type names `point2d` and `point3d`
  - These are synonyms for `(float,float)` and `(float,float,float)`
  - Makes intent more transparent when writing, reading and maintaining code
- More elaborate types — abstract datatypes and object-oriented programming
  - Consider a banking application
  - Data and operations related to accounts, customers, deposits, withdrawals, transfers
  - Denote accounts and customers as separate types
  - Deposits, withdrawals, transfers can be applied to accounts, not customers
  - Updating personal details applies to customers, not accounts

Static analysis

- Identify errors as early as possible — saves cost, effort

Static analysis

- Identify errors as early as possible — saves cost, effort
- In general, compilers cannot check that a program will work correctly
  - Halting problem — Alan Turing

Static analysis

- Identify errors as early as possible — saves cost, effort
- In general, compilers cannot check that a program will work correctly
  - Halting problem — Alan Turing
- With variable delarations, compilers can detect type errors at compile-time — static analysis
  - Dynamic typing would catch these errors only when the code runs
  - Executing code also slows down due to simultaneous monitoring for type correctness

Static analysis

- Identify errors as early as possible — saves cost, effort
- In general, compilers cannot check that a program will work correctly
  - Halting problem — Alan Turing
- With variable delarations, compilers can detect type errors at compile-time — static analysis
  - Dynamic typing would catch these errors only when the code runs
  - Executing code also slows down due to simultaneous monitoring for type correctness
- Compilers can also perform optimizations based on static analysis
  - Reorder statements to optimize reads and writes
  - Store previously computed expressions to re-use later

Summary

- Types have many uses
  - Making sense of arbitrary bit sequences in memory
  - Organizing concepts in our code in a meaningful way
  - Helping compilers catch bugs early, optimize compiled code
- Some languages also support automatic type inference
  - Deduce the types of variable statically, based on the context in which they are used
  - `x = 7` followed by `y = x + 15` implies `y` must be `int`
  - If the inferred type is consistent across the program, all is well

Memory Management

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list
- Scope of a variable
  - When the variable is available for use

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list

- Scope of a variable
  - When the variable is available for use
  - In the following code, the `x` in `f()` is not in scope within call to `g()`

```
def f(l):
  ...
  for x in l:
    y = y + g(x)
  ...
```

```
def g(m):
  ...
  for x in range(m):
    ...
```

Keeping track of variables

- Variables store intermediate values during computation
  - Typically these are local to a function
  - Can also refer to global variables outside the function
  - Dynamically created data, like nodes in a list

- Scope of a variable
  - When the variable is available for use
  - In the following code, the x in f() is not in scope within call to g()

```
def f(l):
  ...
  for x in l:
    y = y + g(x)
  ...
```

```
def g(m):
  ...
  for x in range(m):
    ...
```

- Lifetime of a variable
  - How long the storage remains allocated
  - Above, lifetime of x in f() is till f() exits
  - "Hole in scope" — variable is alive but not in scope

Memory stack

- Each function needs storage for local variables

Memory

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called

Memory

Storage for `factorial(3)`	
`n`	3
`factorial(n-1)`	

- Call `factorial(3)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits

Memory

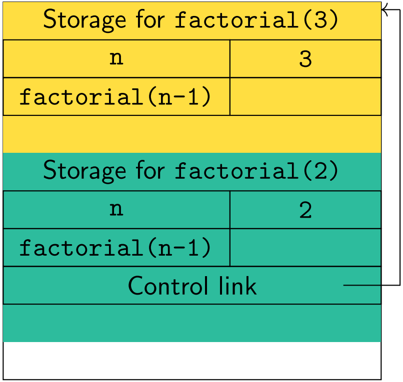
Storage for `factorial(3)`	
`n`	3
`factorial(n-1)`	
Storage for `factorial(2)`	
`n`	2
`factorial(n-1)`	

- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record

Memory

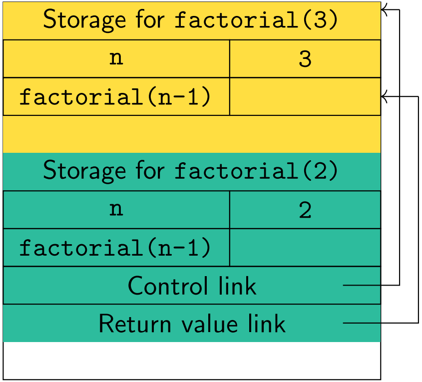


- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result

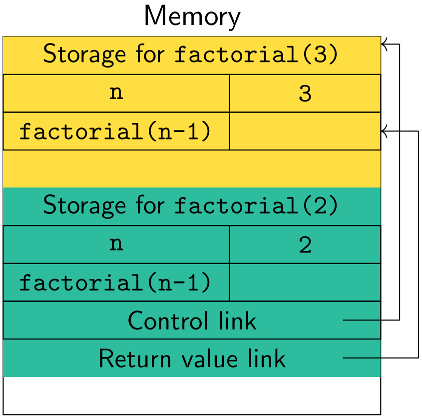
Memory



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

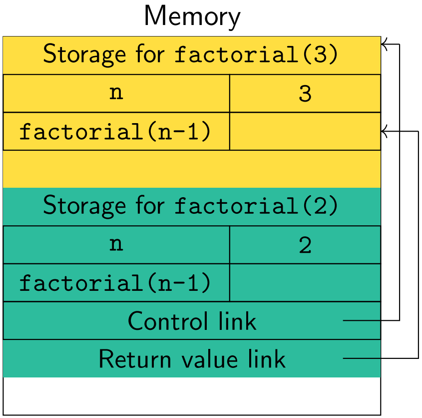
- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result
- Scope of a variable
  - Variable in activation record at top of stack
  - Access global variables by following control links



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Memory stack

- Each function needs storage for local variables
- Create activation record when function is called
- Activation records are stacked
  - Popped when function exits
  - Control link points to start of previous record
  - Return value link tells where to store result
- Scope of a variable
  - Variable in activation record at top of stack
  - Access global variables by following control links
- Lifetime of a variable
  - Storage allocated is still on the stack



- Call `factorial(3)`
- `factorial(3)` calls `factorial(2)`

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):            x = 7
    ...                myl = [8,9,10]
    ...                f(x,myl)
```

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7
    ...              myl = [8,9,10]
    ...              f(x,myl)
```

- Parameters are part of the activation record of the function
    - Values are populated on function call

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7              a = x
    ...              myl = [8,9,10]     l = myl
    ...              f(x,myl)           ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):        x = 7                a = x
   ...             myl = [8,9,10]       l = myl
   ...             f(x,myl)             ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7                a = x
    ...              myl = [8,9,10]       l = myl
    ...              f(x,myl)             ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters
  - Call by value — copy the value
    - Updating the value inside the function has no side-effect

Passing arguments to functions

- When a function is called, arguments are substituted for formal parameters

```
def f(a,l):          x = 7               a = x
    ...              myl = [8,9,10]      l = myl
    ...              f(x,myl)            ... code for f() ...
```

- Parameters are part of the activation record of the function
  - Values are populated on function call
  - Like having implicit assignment statements at the start of the function
- Two ways to initialize the parameters
  - Call by value — copy the value
    - Updating the value inside the function has no side-effect
  - Call by reference — parameter points to same location as argument
    - Can have side-effects
    - Be careful: can update the contents, but cannot change the reference itself

Heap

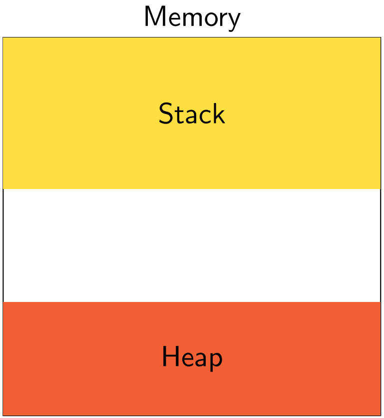
- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record

Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!

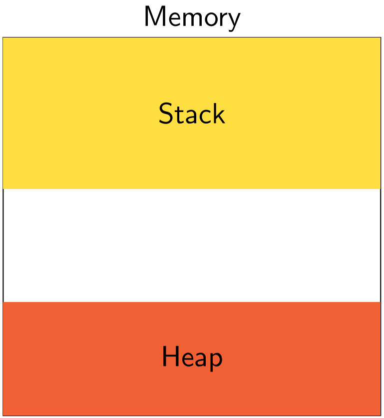
Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!
  - Conceptually, allocate heap storage from "opposite" end with respect to stack



Heap

- Function that inserts a value in a linked list
  - Storage for new node allocated inside function
  - Node should persist after function exits
  - Cannot be allocated within activation record
- Separate storage for persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as the heap data structure!
  - Conceptually, allocate heap storage from "opposite" end with respect to stack
- Heap storage outlives activation record
  - Access through some variable that is in scope



Managing heap storage

- On the stack, variables are deallocated when a function exits

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we "return" unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we "return" unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C
  - Error-prone — memory leaks, invalid assignments

Managing heap storage

- On the stack, variables are deallocated when a function exits
- How do we "return" unused storage on the heap?
  - After deleting a node in a linked list, deleted node i now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C
  - Error-prone — memory leaks, invalid assignments
- Automatic garbage collection (Java, Python, . . .)
  - Run-time environment checks and cleans up dead storage — e.g., mark-and-sweep
    - Mark all storage that is reachable from program variables
    - Return all unmarked memory cells to free space
  - Convenience for programmer vs performance penalty

Summary

- Variables have scope and lifetime
  - Scope — whether the variable is available in the program
  - Lifetime — whether the storage is still allocated
- Activation records for functions are maintained as a stack
  - Control link points to previous activation record
  - Return value link tells where to store result
- Heap is used to store dynamically allocated data
  - Outlives activation record of function that created the storage
  - Need to be careful about deallocating heap storage
  - Explicit deallocation vs automatic garbage collection

Abstraction and modularity

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Stepwise refinement

- Begin with a high level description of the task

```
begin
  print first thousand prime numbers
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people
- Program refinement — focus on code, not much change in data structures

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances
- Refine `PrintStatement()` to include `PrintTransactions()`
  - Now we need to record transactions for each account
  - Data representation also changes
  - Cascading impact on other functions that operate on accounts

Modular software development

- Use refinement to divide the solution into components

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour
- Main challenge: suitable language to write specifications
  - Balance abstraction and detail, should not be another programming language!
  - Cannot algorithmically check that specification is met (halting problem!)

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements
- Object-oriented programming
  - Organize ADTs in a hierarchy
  - Implicit reuse of implementations — subtyping, inheritance

Summary

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks
  - Bottom up: combine simple building blocks
- Modular description of components
  - Interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently, preserving interface and specification
- PL support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming

Object-oriented programming

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — e.g., a counter
  - An entire filesystem or database could be a single object

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — e.g., a counter
  - An entire filesystem or database could be a single object
- Distinguishing features of object-oriented programming
  - Abstraction
  - Subtyping
  - Dynamic lookup
  - Inheritance

History of object-oriented programming

- Objects first introduced in Simula —
  simulation language, 1960s

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue
- Challenges
  - Queue must be well-typed, yet hold all types of events

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

History of object-oriented programming

- Objects first introduced in Simula — simulation language, 1960s
- Event-based simulation follows a basic pattern
  - Maintain a queue of events to be simulated
  - Simulate the event at the head of the queue
  - Add all events it spawns to the queue
- Challenges
  - Queue must be well-typed, yet hold all types of events
  - Use a generic simulation operation across different types of events
    - Avoid elaborate checking of cases

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object
- Data-centric view of programming
  - Focus on what data we need to maintain and manipulate

Abstraction

- Objects are similar to abstract datatypes
  - Public interface
  - Private implementation
  - Changing the implementation should not affect interactions with the object
- Data-centric view of programming
  - Focus on what data we need to maintain and manipulate
- Recall that stepwise refinement could affect both code and data
  - Tying methods to data makes this easier to coordinate
  - Refining data representation naturally tied to updating methods that operate on the data

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?
- Arrange types in a hierarchy
  - A subtype is a specialization of a type
  - If `A` is a subtype of `B`, wherever an object of type `B` is needed, an object of type `A` can be used
    - Every object of type `A` is also an object of type `B`
    - Think subset — if $X \subseteq Y$, every $x \in X$ is also in Y

Subtyping

- Recall the Simula event queue
  - A well-typed queue holds values of a fixed type
  - In practice, the queue holds different types of objects
  - How can this be reconciled?
- Arrange types in a hierarchy
  - A subtype is a specialization of a type
  - If `A` is a subtype of `B`, wherever an object of type `B` is needed, an object of type `A` can be used
    - Every object of type `A` is also an object of type `B`
    - Think subset — if $X \subseteq Y$, every $x \in X$ is also in Y
- If `f()` is a method in `B` and `A` is a subtype of `B`, every object of `A` also supports `f()`
  - Implementation of `f()` can be different in `A`

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself
- Different from overloading
  - Operation `+` is addition for `int` and `float`
  - Internal implementation is different, but choice is determined by static type

Dynamic lookup

- Whether a method can be invoked on an object is a static property — type-checking
- How the method acts is a dynamic property of how the object is implemented
  - In the simulation queue, all events support a simulate method
  - The action triggered by the method depends on the type of event
  - In a graphics application, different types of objects to be rendered
  - Invoke using the same operation, each object "knows" how to render itself
- Different from overloading
  - Operation `+` is addition for `int` and `float`
  - Internal implementation is different, but choice is determined by static type
- Dynamic lookup
  - A variable `v` of type `B` can refer to an object of subtype `A`
  - Static type of `v` is `B`, but method implementation depends on run-time type `A`

Inheritance

- Re-use of implementations

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)
- Usually one hierarchy of types to capture both subtyping and inheritance
  - `A` can inherit from `B` iff `A` is a subtype of `B`

Inheritance

- Re-use of implementations
- Example: different types of employees
  - `Employee` objects store basic personal data, date of joining
  - `Manager` objects can add functionality
    - Retain basic data of `Employee` objects
    - Additional fields and functions: date of promotion, seniority (in current role)
- Usually one hierarchy of types to capture both subtyping and inheritance
  - `A` can inherit from `B` iff `A` is a subtype of `B`
- Philosophically, however the two are different
  - Subtyping is a relationship of interfaces
  - Inheritance is a relationship of implementations

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation
- But `Stack` and `Queue` are not subtypes of `Deque`
  - If `v` of type `Deque` points an object of type `Stack`, cannot invoke `insert-rear()`, `delete-rear()`
  - Similarly, no `insert-front()`, `delete-rear()` in `Queue`

Subtyping vs inheritance

- A deque is a double-ended queue
  - Supports `insert-front()`, `delete-front()`, `insert-rear()` and `delete-rear()`
- We can implement a stack or a queue using a deque
  - Stack: use only `insert-front()`, `delete-front()`,
  - Queue: use only `insert-rear()`, `delete-front()`,
- `Stack` and `Queue` inherit from `Deque` — reuse implementation
- But `Stack` and `Queue` are not subtypes of `Deque`
  - If `v` of type `Deque` points an object of type `Stack`, cannot invoke `insert-rear()`, `delete-rear()`
  - Similarly, no `insert-front()`, `delete-rear()` in `Queue`
- Interfaces of `Stack` and `Queue` are not compatible with `Deque`
  - In fact, `Deque` is a subtype of both `Stack` and `Queue`

Summary

- Objects are like abstract datatypes
- Uniform way of encapsulating different combinations of data and functionality
- Distinguishing features of object-oriented programming
  - Abstraction
    - Public interface, private implementation, like ADTs
  - Subtyping
    - Hierarchy of types, compatibility of interfaces
  - Dynamic lookup
    - Choice of method implementation is determined at run-time
  - Inheritance
    - Reuse of implementations

Classes and objects

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, . . .
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template
  - Each object maintains a separate copy of local data
  - Invoke methods on objects — send a message to the object

Example: 2D points

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`

Example: 2D points

- A point has coordinates (x, y)
  - Each point object stores its own internal values `x` and `y` — instance variables
  - For a point `p`, the local values are `p.x` and `p.y`
  - `self` is a special name referring to the current object — `self.x`, `self.y`
- When we create an object, we need to set it up
  - Implicitly call a constructor function with a fixed name
  - In Python, constructor is called `__init__()`
  - Parameters are used to set up internal values
  - In Python, the first parameter is always `self`

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy
```

Adding methods to a class

- Translation: shift a point by $(\Delta x, \Delta y)$
  - $(x, y) \mapsto (x + \Delta x, y + \Delta y)$
  - Update instance variables
- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$
  - Does not update instance variables
  - state of object is unchanged

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b

  def translate(self,dx,dy):
    self.x += dx
    self.y += dy

  def odistance(self):
    import math
    d = math.sqrt(self.x*self.x +
                  self.y*self.y)
    return(d)
```

Changing the internal implementation

- Polar coordinates: (r,θ), not (x,y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r

```
import math
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)

  def odistance(self):
    return(self.r)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r \cos \theta$, $y = r \sin \theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Changing the internal implementation

- Polar coordinates: (r, θ), not (x, y)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = \tan^{-1}(y/x)$
- Distance from origin is just r
- Translation
  - Convert (r, θ) to (x, y)
  - $x = r \cos \theta$, $y = r \sin \theta$
  - Recompute r, θ from $(x + \Delta x, y + \Delta y)$
- Interface has not changed
  - User need not be aware whether representation is (x, y) or (r, θ)

```
def translate(self,dx,dy):
  x = self.r*math.cos(self.theta)
  y = self.r*math.sin(self.theta)
  x += dx
  y += dy
  self.r = math.sqrt(x*x + y*y)
  if x == 0:
    self.theta = math.pi/2
  else:
    self.theta = math.atan(y/x)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Abstraction

- User should not know whether `Point` uses `(x,y)` or `(r,theta)`
  - Interface remains identical
  - Even constructor is the same
- Python allows direct access to instance variables from outside the class

```
p = Point(5,7)
p.x = 4  # Point is now (4,7)
```

  - Breaks the abstraction
  - Changing the internal implementation of `Point` can have impact on other code
- Rely on programmer discipline

```
class Point:
  def __init__(self,a=0,b=0):
    self.x = a
    self.y = b
```

```
class Point:
  def __init__(self,a=0,b=0):
    self.r = math.sqrt(a*a + b*b)
    if a == 0:
      self.theta = math.pi/2
    else:
      self.theta = math.atan(b/a)
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance

- Define `Square` to be a subtype of `Rectangle`
  - Different constructor
  - Same instance variables
- The following is legal

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))

class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance ...

- Can change the instance variable in `Square`
  - `self.side`

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance . . .

- Can change the instance variable in `Square`
  - `self.side`
- The following gives a run-time error

```
s = Square(5)
a = s.area()
p = s.perimeter()
```

  - `Square` inherits definitions of `area()` and `perimeter()` from `Rectangle`
  - But `s.width` and `s.height` have not been defined!
  - Subtype is not forced to be an extension of the parent type

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.side = s
```

Subtyping and inheritance ...

- Subclass and parent class are usually developed separately

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.width = w
    self.height = h

  def area(self):
    return(self.width*self.height)

  def perimeter(self):
    return(2*(self.width+self.height))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Subclass and parent class are usually developed separately
- Implementor of `Rectangle` changes the instance variables
- The following gives a run-time error

  ```
  s = Square(5)
  a = s.area()
  p = s.perimeter()
  ```

  - `Square` constructor sets `s.width` and `s.height`
  - But the instance variable names have changed!
  - Why should `Square` be affected by this?

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` does (and should) not know the names of the private instance variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance . . .

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` does (and should) not know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Subtyping and inheritance ...

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Instance variables `wd` and `ht` of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` does (and should) not know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables
- Static type checking catches errors early

```
class Rectangle:
  def __init__(self,w=0,h=0):
    self.wd = w
    self.ht = h

  def area(self):
    return(self.wd*self.ht)

  def perimeter(self):
    return(2*(self.wd+self.ht))


class Square(Rectangle):
  def __init__(self,s=0):
    self.width = s
    self.height = s
```

Summary

- A class is a template describing the instance variables and methods for an abstract datatype
- An object is a concrete instance of a class
- We should separate the public interface from the private implementation
- Hierarchy of classes to implement subtyping and inheritance
- A language like Python has no mechanism to enforce privacy etc
  - Can illegally manipulate private instance variables
  - Can introduce inconsistencies between subtype and parent type
- Use strong declarations to enforce privacy, types
  - Do not rely on programmer discipline
  - Catch bugs early through type checking

Programming Concepts Using Java

Week 1 Revision

W01:L01: Introduction

- Explore concepts in programming languages
  - Object-oriented programming
  - Exception handling, concurrency, event-driven programming, . . .
- Use Java as the illustrative language
  - Imperative, object-oriented
  - Incorporates almost all features of interest
- Discuss design decisions where relevant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there
- . . . and why new ones are still being created

W01:L02: Types

- Types have many uses
  - Making sense of arbitrary bit sequences in memory
  - Organizing concepts in our code in a meaningful way
  - Helping compilers catch bugs early, optimize compiled code
- Some languages also support automatic type inference
  - Deduce the types of a variable statically, based on the context in which they are used
  - `x = 7` followed by `y = x + 15` implies `y` must be `int`
  - If the inferred type is consistent across the program, all is well

W01:L03: Memory Management

- Variables have **scope** and **lifetime**
  - Scope — whether the variable is available in the program
  - Lifetime — whether the storage is still allocated
- Activation records for functions are maintained as a stack
  - Control link points to previous activation record
  - Return value link tells where to store result
- Two ways to initialize parameters
  - Call by value
  - Call by reference
- Heap is used to store dynamically allocated data
  - Outlives activation record of function that created the storage
  - Need to be careful about deallocating heap storage
  - Explicit deallocation vs automatic garbage collection

W01:L04: Abstraction and Modularity

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks
  - Bottom up: combine simple building blocks
- Modular description of components
  - Interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently, preserving interface and specification
- PL support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming

W01:L05: OOPS

- Objects are like abstract datatypes
- Uniform way of encapsulating different combinations of data and functionality
- Distinguishing features of object-oriented programming
  - Abstraction
    - Public interface, private implementation, like ADTs
  - Subtyping
    - Hierarchy of types, compatibility of interfaces
  - Dynamic lookup
    - Choice of method implementation is determined at run-time
  - Inheritance
    - Reuse of implementations

W01:L06: Classes

- A class is a template describing the instance variables and methods for an abstract datatype
- An object is a concrete instance of a class
- We should separate the public interface from the private implementation
- Hierarchy of classes to implement subtyping and inheritance
- A language like Python has no mechanism to enforce privacy etc
  - Can illegally manipulate private instance variables
  - Can introduce inconsistencies between subtype and parent type
- Use strong declarations to enforce privacy, types
  - Do not rely on programmer discipline
  - Catch bugs early through type checking

A first taste of Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 2

Getting started

**The C Programming Language,
Brian W Kernighan, Dennis M Ritchie**

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

`hello, world`

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

Getting started

The C Programming Language,
Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

`hello, world`

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

Getting started

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

`hello, world`

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

Getting started

**The C Programming Language,
Brian W Kernighan, Dennis M Ritchie**

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words
`hello, world`

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

- . . . and Java

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated?

- Let's unpack the syntax

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility
- How does the program start?
  - Fix a function name that will be called by default
  - From C, the convention is to call this function `main()`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated ...

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated . . .

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`
- Visibility
  - Function has be available to run from outside the class
  - Modifier `public`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated . . .

- Availability
  - Functions defined inside classes are attached to objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated . . .

- The actual operation
  - System is a public class

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated ...

- The actual operation
  - System is a public class
  - out is a stream object defined in System
    - Like a file handle
    - Note that out must also be static

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated ...

- The actual operation
  - System is a public class
  - out is a stream object defined in System
    - Like a file handle
    - Note that out must also be static
  - println() is a method associated with streams
    - Prints argument with a newline, like Python print()

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

- Punctuation `{`, `}`, `;` to delimit blocks, statements
  - Unlike layout and indentation in Python

Compiling and running Java code

- A Java program is a collection of classes

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

- Java programs are usually interpreted on Java Virtual Machine (JVM)
  - JVM provides a uniform execution environment across operating systems
  - Semantics of Java is defined in terms of JVM, OS-independent
  - "Write once, run anywhere"

Compiling and running Java code

- javac compiles into JVM bytecode
  - javac helloworld.java creates bytecode file helloworld.class

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

- Note:
  - `javac` requires file extension `.java`
  - `java` should not be provided file extension `.class`
  - `javac` automatically follows dependencies and compiles all classes required
    - Sufficient to trigger compilation for class containing `main()`

Summary

- The syntax of Java is comparatively heavy
- Many modifiers: unavoidable overhead of object-oriented design
  - Visibility: `public` vs `private`
  - Availability: all functions live inside objects, need to allow `static` definitions
  - Will see more modifiers as we go along
- Functions and variable types have to be declared in advance
- Java compiles into code for a virtual machine
  - JVM ensures uniform semantics across operating systems
  - Code is guaranteed to be portable

Basic datatypes in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 2

Scalar types

- In an object-oriented language, all data should be encapsulated as objects

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

- 2-byte `char` for Unicode

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

- Boolean constants are true, false

```
boolean b1, b2;

b1 = false;
b2 = true;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

- Modifier `final` indicates a constant

```
final float pi = 3.1415927f;

pi = 22/7;  // Flagged as error;
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

  ```
  float f = 22/7;    // Value is 3.0
  ```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n

- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;    // Same as a = a+1
b--;    // Same as b = b-1
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+, -, *, /, %`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n

- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;    // Same as a = a+1
b--;    // Same as b = b-1
```

- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7;     // Same as a = a+7
b *= 12;    // Same as b = b*12
```

Strings

- `String` is a built in class

```
String s,t;
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

```
s[3] = 'p';
s[4] = '!';
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

    ```
    s[3] = 'p';
    s[4] = '!';
    ```

- Instead, invoke method `substring` in class `String`
  - `s = s.substring(0,3) + "p!";`

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

    ```
    s[3] = 'p';
    s[4] = '!';
    ```

- Instead, invoke method `substring` in class `String`
  - `s = s.substring(0,3) + "p!";`
- If we change a `String`, we get a new object
  - After the update, `s` points to a new `String`
  - Java does automatic garbage collection

Arrays

- Arrays are also objects

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`

Arrays

- Arrays are also objects
- Typical declaration

  ```
  int[] a;
  a = new int[100];
  ```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`

- Size of the array can vary
- Array constants: `{v1, v2, v3}`
- For example

  ```
  int[] a;
  int n;

  n = 10;
  a = new int[n];

  n = 20;
  a = new int[n];

  a = {2, 3, 5, 7, 11};
  ```

Summary

- Java allows scalar types, which are not objects
  - `int`, `long`, `short`, `byte`, `float`, `double`, `char`, `boolean`
- Declarations can include initializations
- Strings and arrays are objects
- Numerous versions of Java: we will use Java 11
- Extensive online documentation — look up in case of doubt
  https://docs.oracle.com/en/java/javase/11/docs/api/index.html

Control flow in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 2

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`
- Multiway branching – `switch`

Conditional execution

- `if (c) {...} else {...}`
  - `else` is optional
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- No `elif`, à la Python
  - Indentation is not forced
  - Just align `else if`
  - Nested `if` is a single statement, no separate braces required
- No surprises
- Aside: no `def` for function definition

```
public class MyClass {

  ...

  public static int sign(int v) {
    if (v < 0) {
      return(-1);
    } else if (v > 0) {
      return(1);
    } else {
      return(0);
    }
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;

    while (n > 0){
      sum += n;
      n--;
    }

    return(sum);
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration
  - Useful for interactive user input

```
do {
  read input;
} while (input-condition);
```

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }
```

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update
- Intended use is
  `for(i = 0; i < n; i++){...}`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }
```

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update
- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is

  ```
  for(i = 0; i < n; i++){...}
  ```

- Completely equivalent to

  ```
  i = 0;
  while (i < n) {
    i++;
  }
  ```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is

```
for(i = 0; i < n; i++){...}
```

- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

- However, not good style to write `for` instead of `while`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

- However, not good style to write `for` instead of `while`
- Can define loop variable within loop
  - The scope of `i` is local to the loop
  - An instance of more general local scoping allowed in Java

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for (int i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

  ```
  for x in l:
    do something with x
  ```

- Again `for`, different syntax

  ```
  for (type x : a)
    do something with x;
  }
  ```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for (int v : a){
      sum += v;
    }

    return(sum);
  }
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

- Again `for`, different syntax

```
for (type x : a)
  do something with x;
}
```

- It appears that loop variable must be declared in local scope for this version of `for`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for (int v : a){
      sum += v;
    }

    return(sum);
  }
```

Multiway branching

- `switch` selects between different options

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions
- Aside: here return type is `void`
  - Non-`void` return type requires an appropriate `return` value

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Summary

- Program layout: semi-colons, braces
- Conditional execution: `if`, `else`
- Conditional loops: `while`, `do-while`
- Iteration: two kinds of `for`
  - Local declaration of loop variable
- Multiway branching: `switch`
  - `break` to avoid falling through

Defining classes and objects in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 2

Classes and objects

- A class is a template for an encapsulated type
- An object is an instance of a class
- How do we create objects?
- How are objects initialized?

Defining a class

- Definition block using `class`, with class name
  - Modifier `public` to indicate visibility
  - Java allows `public` to be omitted
  - Default visibility is public to `package`
  - Packages are administrative units of code
  - All classes defined in same directory form part of same package

```
public class Date {

  private int day, month, year;

  ...

}
```

Defining a class

- Definition block using `class`, with class name
  - Modifier `public` to indicate visibility
  - Java allows `public` to be omitted
  - Default visibility is public to `package`
  - Packages are administrative units of code
  - All classes defined in same directory form part of same package
- Instance variables
  - Each concrete object of type `Date` will have local copies of `date`, `month`, `year`
  - These are marked `private`
  - Can also have `public` instance variables, but breaks encapsulation

```
public class Date {

  private int day, month, year;

  ...

}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the instance variables?

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the instance variables?
- Can add methods to update values
  - `this` is a reference to current object

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

```
public class Date {
  private int day, month, year;

  public void setDate(int d, int m,
                      int y){
    this.day = d;
    this.month = m;
    this.year = y;
  }
}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

```
public class Date {
  private int day, month, year;

  public void setDate(int d, int m,
                      int y){
    day = d;
    month = m;
    year = y;
  }
}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous
- What if we want to check the values?
  - Methods to read and report values

```
public class Date {
  ...

  public int getDay(){
    return(day);
  }

  public int getMonth(){
    return(month);
  }

  public int getYear(){
    return(year);
  }

}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous
- What if we want to check the values?
  - Methods to read and report values
- Accessor and Mutator methods

```
public class Date {
  ...

  public int getDay(){
    return(day);
  }

  public int getMonth(){
    return(month);
  }

  public int getYear(){
    return(year);
  }

}
```

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - `d = new Date(13,8,2015);`

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }
}
```

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - `d = new Date(13,8,2015);`
- Constructors with different signatures
  - `d = new Date(13,8);` sets `year` to `2021`
  - Java allows function overloading — same name, different signatures
    - Python: default (optional) arguments, no overloading

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    day = d;
    month = m;
    year = 2021;
  }
}
```

Constructors . . .

- A later constructor can call an earlier one using `this`

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    this(d,m,2021);
  }
}
```

Constructors . . .

- A later constructor can call an earlier one using `this`
- If no constructor is defined, Java provides a default constructor with empty arguments
  - `new Date()` would implicitly invoke this
  - Sets instance variables to sensible defaults
  - For instance, `int` variables set to `0`
  - Only valid if *no* constructor is defined
  - Otherwise need an explicit constructor without arguments

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    this(d,m,2021);
  }
}
```

Copy constructors

- Create a new object from an existing one

```
public class Date {
  private int day, month, year;

  public Date(Date d){
    this.day = d.day;
    this.month = d.month;
    this.year = d.year;
  }
}
```

Copy constructors

- Create a new object from an existing one
- Copy constructor takes an object of the same type as argument
  - Copies the instance variables
  - Use object name to disambiguate which instance variables we are talking about
  - Note that private instance variables of argument are visible

```
public class Date {
  private int day, month, year;

  public Date(Date d){
    this.day = d.day;
    this.month = d.month;
    this.year = d.year;
  }
}


public void UseDate() {
  Date d1,d2;
  d1 = new Date(12,4,1954);
  d2 = new.Date(d1);
}
```

Copy constructors

- Create a new object from an existing one
- Copy constructor takes an object of the same type as argument
  - Copies the instance variables
  - Use object name to disambiguate which instance variables we are talking about
  - Note that private instance variables of argument are visible
- Shallow copy vs deep copy
  - Want new object to be disjoint from old one
  - If instance variable are objects, we may end up aliasing rather than copying
  - Discuss later — cloning objects

```
public class Date {
  private int day, month, year;

  public Date(Date d){
    this.day = d.day;
    this.month = d.month;
    this.year = d.year;
  }
}


public void UseDate() {
  Date d1,d2;
  d1 = new Date(12,4,1954);
  d2 = new.Date(d1);
}
```

Summary

- A class defines a type
- Typically, instance variables are private, available through accessor and mutator methods
- We declare variables using the class name as type
- Use `new` to create an object
- Constructor is called implicitly to set up an object
  - Multiple constructors — overloading
  - Reuse — one constructor can call another
  - Default constructor, if none is defined
  - Copy constructor — make a copy of an existing object

Basic input and output in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 2

Interacting with a Java program

- We have seen how to print data
  - `System.out.println("hello, world");`
- How do we read data

Reading input

- Simplest to use is the `Console` class
  - Functionality similar to Python `input()`

Reading input

- Simplest to use is the Console class
  - Functionality similar to Python input()
- Defined within System
  - Two methods, readLine and readPassword
  - readPassword does not echo characters on the screen
  - readLine returns a string (like Python input())
  - readPassword returns an array of char — for security reasons

```
Console cons = System.console();
String username =
    cons.readLine("User name: ");
char[] passwd =
    cons.readPassword("Password: ");
```

Reading input

- Simplest to use is the `Console` class
  - Functionality similar to Python `input()`
- Defined within `System`
  - Two methods, `readLine` and `readPassword`
  - `readPassword` does not echo characters on the screen
  - `readLine` returns a string (like Python `input()`)
  - `readPassword` returns an array of `char` — for security reasons

```
Console cons = System.console();
String username =
    cons.readLine("User name: ");
char[] passwd =
    cons.readPassword("Password: ");
```

- More general `Scanner` class
  - Allows more granular reading of input
  - Read a full line, or read an integer, ...

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
int age = in.nextInt();
....
```

Generating output

- `System.out.println(arg)` prints `arg` and goes to a new line
  - Implicitly converts argument to a string

Generating output

- `System.out.println(arg)` prints `arg` and goes to a new line
  - Implicitly converts argument to a string
- `System.out.print(arg)` is similar, but does not advance to a new line

Generating output

- `System.out.println(arg)` prints `arg` and goes to a new line
  - Implicitly converts argument to a string
- `System.out.print(arg)` is similar, but does not advance to a new line
- `System.out.printf(arg)` generates formatted output
  - Same conventions as `printf` in C
  - Read the documentation

Programming Concepts Using Java

Week 2 Revision

Getting started

- Java program to print hello, world

```
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("hello, world);
    }
}
```

- A Java program is a collection of classes
- All code in Java lives within a class
- Modifier public specifies visibility
- The signature of main( )
    - Input parameter is an array of strings; command line arguments
    - No output, so return type is void
- Write once, run anywhere

Scalar types

- Java has eight primitive scalar types
  - int, long, short, byte
  - float, double
  - char
  - boolean
- We declare variables before we use them

```
int x, y;
x = 5;
y = 10;
```

- Characters are written with single-quotes (only)

```
char c = 'x';
```

- Boolean constants are true, false

```
boolean b1, b2;
b1 = false;
b2 = true;
```

Scalar types

- Initialize at time of declaration

```
flat pi =  3.1415927f;
```

- Modifier final indicates a constant

```
final float pi = 3.1415927f;
```

Operators

- Arithmetic operators are the usual ones

  `+, -, *, /, %`

- No separate integer division operator //
- When both arguments are integer, / is integer division
- No exponentiation operater, use Math.pow()
- Math.pow(a,n) returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++; // Same as a = a+1
b--; // Same as b = b-1
```

- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7; // Same as a = a+7
```

Strings

- String is a built-in class
- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- + is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
// "Hello world"
```

- Strings are not arrays of characters
- Instead use s.charAt(0), s.substring(0,3)

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

- Or int a[] instead of int[] a
- a.length gives size of a
- Array indices run from 0 to a.length-1

Control flow

- Conditional execution

```
if (condition) { ... } else { ... }
```

- Conditional loops

```
while (condition) { ... }
do { ... } while (condition)
```

- Iteration - Two kinds of for
- Multiway branching – switch

Classes and objects

- A class is a template for an encapsulated type
- An object is an instance of a class

```
public class Date {
    private int day, month, year;
    public Date(int d, int m, int y){
        day = d;
        month = m;
        year = y;
    }
    public int getDay(){
        return(day);
    }
}
```

- Instance variables - Each concrete object of type Date will have local copies of date, month, year

Creating and initializing objects

- new creates a new object
- How do we set the instance variables?
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - d = new Date(13,8,2015);
- Constructor overloading - same name, different signatures
- A constructor can call another one using this
- If no constructor is defined, Java provides a default constructor with empty arguments
  - new Date() would implicitly invoke this
  - Sets instance variables to sensible defaults
  - For instance, int variables set to 0
  - Only valid if no constructor is defined
  - Otherwise need an explicit constructor without arguments

Copy constructors

- Create a new object from an existing one

```
public class Date {
    private int day, month, year;
    public Date(int d, int m, int y){
        day = d; month = m; year = y;
    }
    public Date(Date d){
        this.day = d.day; this.month = d.month; this.year = d.year;
    }
}
public class UseDate() {
    public static void main(String[] args){
        Date d1,d2;
        d1 = new Date(12,4,1954); d2 = new.Date(d1);
    }
}
```

Basic input and output in java

- Reading input
  - Use Console class
  - Use Scanner class

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
int age = in.nextInt();
```

The philosophy of OO programming

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Algorithms + Data Structures = Programs

- Title of Niklaus Wirth's introduction to Pascal

Algorithms + Data Structures = Programs

- Title of Niklaus Wirth's introduction to Pascal
- Traditionally, algorithms come first

Algorithms + Data Structures = Programs

- Title of Niklaus Wirth's introduction to Pascal
- Traditionally, algorithms come first
- Structured programming
  - Design a set of procedures for specific tasks
  - Combine them to build complex systems

Algorithms + Data Structures = Programs

- Title of Niklaus Wirth's introduction to Pascal
- Traditionally, algorithms come first
- Structured programming
  - Design a set of procedures for specific tasks
  - Combine them to build complex systems
- Data representation comes later
  - Design data structures to suit procedural manipulations

Object Oriented design

- Reverse the focus

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems
- Example: simple web browser
    - 2000 procedures manipulating global data

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems
- Example: simple web browser
  - 2000 procedures manipulating global data
  - ...vs 100 classes, each with about 20 methods

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems
- Example: simple web browser
  - 2000 procedures manipulating global data
  - ...vs 100 classes, each with about 20 methods
  - Much easier to grasp the design

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems
- Example: simple web browser
  - 2000 procedures manipulating global data
  - ...vs 100 classes, each with about 20 methods
  - Much easier to grasp the design
  - Debugging: an object is in an incorrect state

Object Oriented design

- Reverse the focus
- First identify the data we want to maintain and manipulate
- Then identify algorithms to operate on the data
- Claim: works better for large systems
- Example: simple web browser
  - 2000 procedures manipulating global data
  - ...vs 100 classes, each with about 20 methods
  - Much easier to grasp the design
  - Debugging: an object is in an incorrect state
  - Search among 20 methods rather than 2000 procedures

Object Oriented design: Example

- An order processing system typically involves
  - Items
  - Orders
  - Shipping addresses
  - Payments
  - Accounts

Object Oriented design: Example

- An order processing system typically involves
  - Items
  - Orders
  - Shipping addresses
  - Payments
  - Accounts
- What happens to these objects?
  - Items are added to orders
  - Orders are shipped, cancelled
  - Payments are accepted, rejected

Object Oriented design: Example

- An order processing system typically involves
  - Items
  - Orders
  - Shipping addresses
  - Payments
  - Accounts
- What happens to these objects?
  - Items are added to orders
  - Orders are shipped, cancelled
  - Payments are accepted, rejected
- Nouns signify objects, verbs denote methods that operate on objects
  - Associate with each order, a method to add an item

Designing objects

- Behaviour — what methods do we need to operate on objects?

Designing objects

- Behaviour — what methods do we need to operate on objects?
- State — how does the object react when methods are invoked?
  - State is the information in the instance variables
  - Encapsulation — should not change unless a method operates on it

Designing objects

- Behaviour — what methods do we need to operate on objects?
- State — how does the object react when methods are invoked?
  - State is the information in the instance variables
  - Encapsulation — should not change unless a method operates on it
- Identity — distinguish between different objects of the same class
  - State may be the same — two orders may contain the same item

Designing objects

- Behaviour — what methods do we need to operate on objects?
- State — how does the object react when methods are invoked?
  - State is the information in the instance variables
  - Encapsulation — should not change unless a method operates on it
- Identity — distinguish between different objects of the same class
  - State may be the same — two orders may contain the same item
- These features interact
  - State will typically affect behaviour
  - Cannot add an item to an order that has been shipped
  - Cannot ship an empty order

Relationship between classes

- Dependence
  - `Order` needs `Account` to check credit status
  - `Item` does not depend on `Account`
  - Robust design minimizes dependencies, or coupling between classes

Relationship between classes

- Dependence
  - `Order` needs `Account` to check credit status
  - `Item` does not depend on `Account`
  - Robust design minimizes dependencies, or coupling between classes
- Aggregation
  - `Order` contains `Item` objects

Relationship between classes

- Dependence
  - `Order` needs `Account` to check credit status
  - `Item` does not depend on `Account`
  - Robust design minimizes dependencies, or coupling between classes
- Aggregation
  - `Order` contains `Item` objects
- Inheritance
  - One object is a specialized versions of another
  - `ExpressOrder` inherits from `Order`
  - Extra methods to compute shipping charges, priority handling

Summary

- An object-oriented approach can help organize code in large projects
- This course is not about software engineering
- Nevertheless, useful to know the motivation underlying OO programming to understand design choices in a programming language like Java

Subclasses and inheritance

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

A Java class

- An `Employee` class

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

A Java class

- An `Employee` class
- Two private instance variables

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

A Java class

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

A Java class

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object
- Accessor and mutator methods to set instance variables

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

A Java class

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object
- Accessor and mutator methods to set instance variables
- A public method to compute bonus

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.
- `Manager` is a subclass of `Employee`
  - Think of subset

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`
- Use parent class's constructor using `super`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`
- Use parent class's constructor using `super`
- A constructor for `Manager`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

```
public class Manager extends Employee{
  ..
  public Manager(String n, double s, String sn){
     super(n,s);   /* super calls
                      Employee constructor */
     secretary = sn;
  }
}
```

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```
- But the following will not work

  ```
  Manager m = new Employee(...)
  ```

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  `Employee e = new Manager(...)`
- But the following will not work

  `Manager m = new Employee(...)`

- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  `Employee e = new Manager(...)`
- But the following will not work

  `Manager m = new Employee(...)`

- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?
- One can now presumably write

  `Employee[] e = new Manager(...)[100]`

Summary

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclass can add more instance variables and methods
  - Can also override methods — later
- Subclasses cannot see private components of parent class
- Use `super` to access constructor of parent class

Dynamic dispatch and polymorphism

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclasses cannot see private components of parent class
- Subclass can add more instance variables and methods

```
public class Employee{
  private String name;
  private double salary;

  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }
  public String getName(){ ... }
  public double getSalary(){ ... }

  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}

public class Manager extends Employee{
  private String secretary;
  public boolean setSecretary(name s){ ... }
  public String getSecretary(){ ... }
}
```

Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclasses cannot see private components of parent class
- Subclass can add more instance variables and methods
- Can also override methods

```
public class Employee{
  private String name;
  private double salary;

  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }
  public String getName(){ ... }
  public double getSalary(){ ... }

  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}

public class Manager extends Employee{
  private String secretary;
  public boolean setSecretary(name s){ ... }
  public String getSecretary(){ ... }
}
```

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
  return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

Dynamic dispatch

- Manager can redefine bonus()

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class bonus() via super
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
    return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`

- Dynamic dispatch (dynamic binding, late method binding, . . .) turns out to be more useful
  - Default in Java, optional in languages like C++ (`virtual` function)

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Recall the event simulation loop that motivated Simula to introduce objects

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Recall the event simulation loop that motivated Simula to introduce objects
- Also referred to as runtime polymorphism or inheritance polymorphism

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays
- Made possible by overloaded methods defined in class `Arrays`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Type casting

- Consider the following assignment

  ```
  Employee e = new Manager(...)
  ```

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`

- Cast fails (error at run time) if `e` is not a `Manager`

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`

- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`

- Cast fails (error at run time) if `e` is not a `Manager`

- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"

- Can also use type casting for basic types

```
double d = 29.98;
int nd = (int) d;
```

Summary

- A subclass can override a method from a parent class
- Dynamic dispatch ensures that the most appropriate method is called, based on the run-time identity of the object
- Run-time/inheritance polymorphism, different from overloading
  - We will later see another type of polymorphism, structural polymorphism
  - For instance, use the same sorting function for array of any datatype that supports a comparison operation
  - Java uses the term *generics* for this
- Use type-casting (and reflection) overcome static type restrictions

The Java class hierarchy

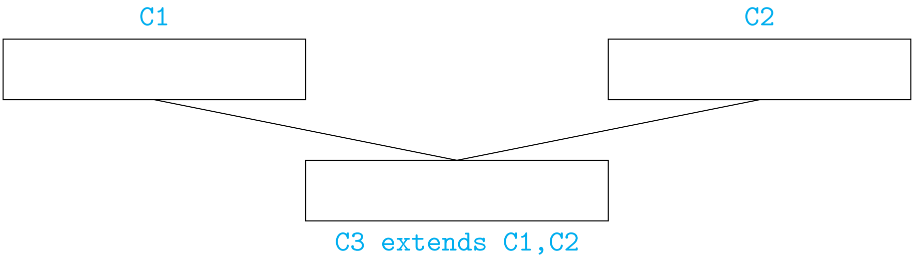
Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

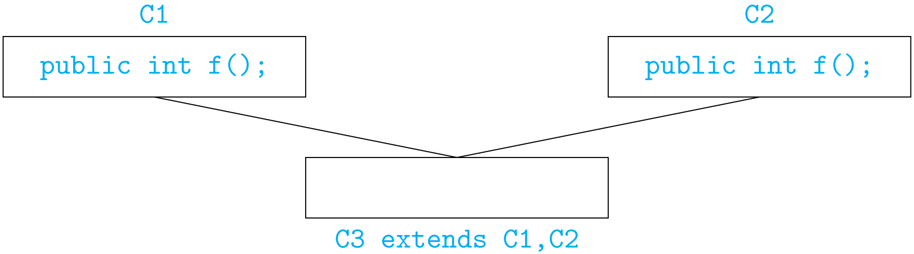
Week 3

Multiple inheritance



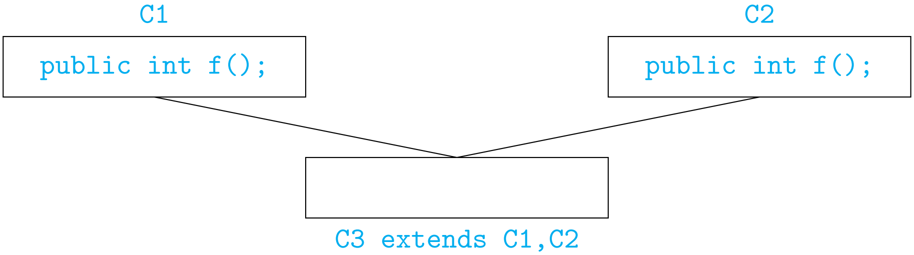
- Can a subclass extend multiple parent classes?

Multiple inheritance



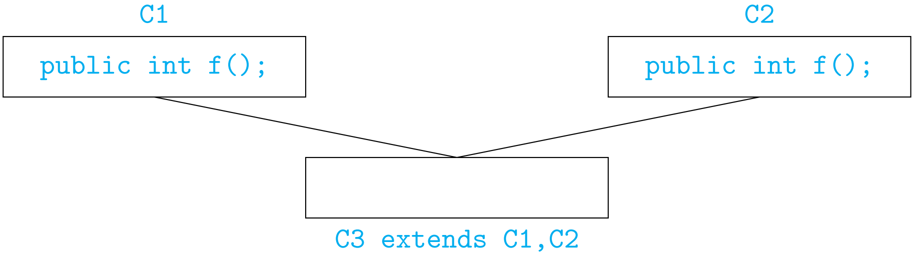
- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?

Multiple inheritance



- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?
- Java does not allow multiple inheritance

Multiple inheritance



- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?
- Java does not allow multiple inheritance
- C++ allows this if `C1` and `C2` have no conflict

Java class hierarchy

- No multiple inheritance — tree-like

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to pointer equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to pointer equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to pointer equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`
- To print `o`, use `System.out.println(o+"");`
  - Implicitly invokes `o.toString()`

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that `==` is pointer equality, by default

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that `==` is pointer equality, by default
- If a class overrides `equals()`, dynamic dispatch will use the redefined function instead of `Object.equals()` for `objarr[i] == o`

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

- Unfortunately,
  `boolean equals(Date d)`
  does not override
  `boolean equals(Object o)`!

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

- Unfortunately,
  `boolean equals(Date d)`
  does not override
  `boolean equals(Object o)`!

- Should write, instead

```
public boolean equals(Object d){
  if (d instanceof Date){
    Date myd = (Date) d;
    return ((this.day == myd.day) &&
            (this.month == myd.month) &&
            (this.year == myd.year));
  }
  return(false);
}
```

  - Note the run-time type check and the cast

Overriding functions

- Overriding looks for "closest" match

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

- `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

- `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`
- Use `boolean equals(Employee e)`

Summary

- Java does not allow multiple inheritance
  - A subclass can extend only one parent class
- The Java class hierarchy forms a tree
- The root of the hierarchy is a built-in class called `Object`
  - `Object` defines default functions like `equals()` and `toString()`
  - These are implicitly inherited by any class that we write
- When we override functions, we should be careful to check the signature

Subtyping vs inheritance

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal
- Inheritance
  - Subtype can reuse code of the main type
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`
  - `Manager.bonus()` uses `Employee.bonus()`

Subtyping vs inheritance

- Recall the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping vs inheritance

- Recall the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?

Subtyping vs inheritance

- Recall the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types

Subtyping vs inheritance

- Recall the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types
- Inheritance
  - Can suppress two functions in a `deque` and use it as a `queue` or `stack`
  - Both `queue` and `stack` inherit from `deque`

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two

Java modifiers

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 3

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods
- Let's look at some examples of situations where different combinations make sense

`public` vs `private`

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`

`public` vs `private`

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`
- Can `private` methods make sense?

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`
- Can `private` methods make sense?
- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- `push()` needs to check if stack has space

```
public class Stack {
  ...
  public void push (int i){
    if (tos < size){
      values[tos] = i;
      tos = tos+1;
    }else{
      // Deal with stack overflow
    }
    ...
  }
  ...
}
```

private methods

- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- `push()` needs to check if stack has space
- Deal gracefully with stack overflow
  - `private` methods invoked from within `push()` to check if stack is full and expand storage

```
public class Stack {
  ...
  public void push (int i){
    if (stack_full()){
      extend_stack();
    }
    ... // Usual push operations
  }
  ...
  private boolean stack_full(){
    return(tos == size);
  }

  private void extend_stack(){
    /* Allocate additional space,
       reset size etc */
  }
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of `day` and `month`

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- `Date` class
  - Private instance variables `day`, `month`, `year`
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of `day` and `month`
- Instead, allow only combined update

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

static components

- Use static for components that exist without creating objects
  - Library functions, main(), ...
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- lastorderid is private static field

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- `lastorderid` is private static field
- Common to all objects in the class

static components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`
- These `static` components are also `public`
- Do `private static` components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- `lastorderid` is private static field
- Common to all objects in the class
- Be careful about concurrent updates!

`final` components

- `final` denotes that a value cannot be updated

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class
- A `final` method cannot be overridden

Summary

- `private` and `public` are natural artefacts of encapsulation
  - Usually, instance variables are `private` and methods are `public`
  - However, `private` methods also make sense
- Modifiers `static` and `final` are orthogonal to `public`/`private`
- Use `private static` instance variables to maintain bookkeeping information across objects in a class
  - Global serial number, count number of objects created, profile method invocations, ...
- Usually `final` is used with instance variables to denote constants
- Also makes sense for methods
  - A `final` method cannot be overridden by a subclass
- Can also have `private` classes, later

Programming Concepts Using Java

Week 3 Revision

W03:L01: The philosophy of OO programming

- Structured programming
  - The algorithms come first
    - Design a set of procedures for specific tasks
    - Combine them to build complex systems
  - Data representation comes later
    - Design data structures to suit procedural manipulations
- Object Oriented design
  - First identify the data we want to maintain and manipulate
  - Then identify algorithms to operate on the data
- **Designing objects**
  - **Behaviour** – what methods do we need to operate on objects?
  - **State** – how does the object react when methods are invoked?
    - State is the information in the instance variables
    - **Encapsulation** – should not change unless a method operates on it

W03:L01: The philosophy of OO programming (Cont.)

- Relationship between classes
  - Dependence
    - Order needs Account to check credit status
    - Item does not depend on Account
    - Robust design minimizes dependencies, or coupling between classes
  - Aggregation
    - Order contains Item objects
  - Inheritance
    - One object is a specialized versions of another
    - ExpressOrder inherits from Order
    - Extra methods to compute shipping charges, priority handling

W03:L02: Subclasses and inheritance

- A subclass extends a parent class
- Subclass inherits instance variables and methods from the parent class
- Subclass can add more instance variables and methods
  - Can also override methods
- Subclasses cannot see private components of parent class
- Use super to access constructor of parent class
- Manager objects inherit other fields and methods from Employee
- Every Manager has a name, salary and methods to access and manipulate these.

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- Manager can redefine bonus()

```
double bonus(float percent){
  return 1.5*super.bonus(percent);
}
```

  - Uses parent class bonus() via super
  - Overrides definition in parent class
- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke e.setSecretary()?
  - e is declared to be an Employee
  - Static typechecking – e can only refer to methods in Employee

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- What about e.bonus(p)? Which bonus() do we use?
  - Static: Use Employee.bonus()
  - Dynamic: Use Manager.bonus()
- **Dynamic dispatch** (dynamic binding, late method binding, . . .) turns out to be more useful
- **Polymorphism**
  - Every Employee in emparray "knows" how to calculate its bonus correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);
emparray[0] = e;
emparray[1] = m;
for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

- Signature of a function is its name and the list of argument types
- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - Employee.bonus()
  - Manager.bonus()
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
    // sorts contents of darr
Arrays.sort(iarr);
    // sorts contents of iarr
class Arrays{
    ...
    public static void sort(double[] a){..}
        // sorts arrays of double[]
    public static void sort(int[] a){..}
        // sorts arrays of int[]
    ...
}
```

W03:L03: Dynamic dispatch and polymorphism (Cont.)

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`
- e.setSecretary() does not work
  - Static type-checking disallows this
- Type casting — convert e to Manager

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if e is not a Manager
- Can test if e is a Manager

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

W03:L04: The Java class hierarchy

- Java does not allow multiple inheritance
  - A subclass can extend only one parent class
- The Java class hierarchy forms a tree
- The root of the hierarchy is a built-in class called Object
  - Object defines default functions like equals() and toString()
  - These are implicitly inherited by any class that we write
- When we override functions, we should be careful to check the signature
- Useful methods defined in Object

```
public boolean equals(Object o) // defaults to pointer equality
public String toString()        // converts the values of the
                                // instance variables to String
```

- For Java objects x and y, x == y invokes x.equals(y)
- To print o, use System.out.println(o+"");
  - Implicitly invokes o.toString()

W03:L05: Subtyping vs inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If B is a subtype of A, wherever we require an object of type A, we can use an object of type B
  - Employee e = new Manager(...); is legal
  - Compatibility of interfaces
- Inheritance
  - Subtype can reuse code of the main type
  - B inherits from A if some functions for B are written in terms of functions of A
  - Manager.bonus() uses Employee.bonus()
  - Reuse of implementations
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two

W03:L06: Java modifiers

- private and public are natural artefacts of encapsulation
  - Usually, instance variables are private and methods are public
  - However, private methods also make sense
- Modifiers static and final are orthogonal to public/private
- Use private static instance variables to maintain bookkeeping information across objects in a class
  - Global serial number, count number of objects created, profile method invocations, . . .
- Usually final is used with instance variables to denote constants
- A final method cannot be overridden by a subclass
- A final class cannot be inherited
- Can also have private classes

Abstract classes and interfaces

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

Grouping together classes

- Sometimes we collect together classes under a common heading

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  `public double perimeter()`

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```

- Could define a function in `Shape` that returns an absurd value

  ```
  public double perimeter() { return(-1.0); }
  ```

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```
- Could define a function in `Shape` that returns an absurd value

  ```
  public double perimeter() { return(-1.0); }
  ```
- Rely on the subclass to redefine this function

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```
- Could define a function in `Shape` that returns an absurd value

  ```
  public double perimeter() { return(-1.0); }
  ```
- Rely on the subclass to redefine this function
- What if this doesn't happen?
  - Should not depend on programmer discipline

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

```
public abstract double perimeter();
```

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions
- `Shape` must itself be declared to be `abstract`

  ```
  public abstract class Shape{
    ...
    public abstract double perimeter();
    ...
  }
  ```

Abstract classes . . .

- Can still declare variables whose type is an abstract class

Abstract classes ...

- Can still declare variables whose type is an abstract class

```
Shape shapearr[] = new Shape[3];
int sizearr[] = new int[3];

shapearr[0] = new Circle(...);
shapearr[1] = new Square(...);
shapearr[2] = new Rectangle(...);

for (i = 0; i < 2; i++){
  sizearr[i] = shapearr[i].perimeter();
     // each shapearr[i] calls the appropriate method
  ...
}
```

Generic functions

- Use abstract classes to specify generic properties

```
public abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Generic functions

- Use abstract classes to specify generic properties

```
public abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

- Now we can sort any array of objects that extend `Comparable`

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use a[i].cmp(a[j])
  }
}
```

Generic functions ...

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

Generic functions . . .

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

- To use this definition of `quicksort`, we write

```
public class Myclass extends Comparable{
  private double size;   // quantity used for comparison

  public int cmp(Comparable s){
    if (s instanceof Myclass){
      // compare this.size and ((Myclass) s).size
      // Note the cast to access s.size
    }
  }
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
      ...
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
     ...
}
```

- Can extend only one class, but can implement multiple interfaces

Summary

- We can use the class hierarchy to group together related classes
- An abstract method in a parent class forces each subclass to implement it in a sensible manner
- Any class with an abtract method is itself abstract
    - Cannot create objects corresponding to an abstract class
    - However, we can define variables whose type is an abstract class
- Abstract classes can also describe capabilities, allowing for generic functions
- An interface is an abstract class with no concrete components
    - A class to extend only one parent class, but it can implement any number of interfaces

Interfaces

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

Interfaces

- An interface is a purely abstract class
  - All methods are abstract
- A class implements an interface
  - Provide concrete code for each abstract function
- Classes can implement multiple interfaces
  - Abstract functions, so no contradictory inheritance
- Interfaces describe relevant aspects of a class
  - Abstract functions describe a specific "slice" of capabilities
  - Another class only needs to know about these capabilities

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

Exposing limited capabilities

- Generic quicksort for any datatype that supports comparisons
- Express this capability by making the argument type Comparable[]
  - Only information that quicksort needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by Comparable objects through an interface

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by `Comparable` objects through an interface
- However, we cannot express the intended behaviour of `cmp` explicitly

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Adding methods to interfaces

- Java interfaces extended to allow functions to be added

Adding methods to interfaces

- Java interfaces extended to allow functions to be added
- Static functions
  - Cannot access instance variables
  - Invoke directly or using interface name: `Comparable.cmpdoc()`

```
public interface Comparable{
  public static String cmpdoc(){
    String s;
    s = "Return -1 if this < s, ";
    s = s + "0 if this == s, ";
    s = s + "+1 if this >  s.";
    return(s);
}
```

Adding methods to interfaces

- Java interfaces extended to allow functions to be added
- Static functions
  - Cannot access instance variables
  - Invoke directly or using interface name: `Comparable.cmpdoc()`
- Default functions
  - Provide a default implementation for some functions
  - Class can override these
  - Invoke like normal method, using object name: `a[i].cmp(a[j])`

```
public interface Comparable{
  public static String cmpdoc(){
    String s;
    s = "Return -1 if this < s, ";
    s = s + "0 if this == s, ";
    s = s + "+1 if this >  s.";
    return(s);
}
```

```
public interface Comparable{
  public default int cmp(Comparable s) {
    return(0);
  }
}
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods

```
public interface Person{
  public default String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  implements Person, Designation {...}
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods
- Subclass must provide a fresh implementation

```
public interface Person{
  public default String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  implements Person, Designation {
  ...

  public String getName(){
    ...
  }
}
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods
- Subclass must provide a fresh implementation
- Conflict could be between a class and an interface
  - `Employee` inherits from class `Person` and implements `Designation`
  - Method inherited from the class "wins"
  - Motivated by reverse compatibility

```
public class Person{
  public String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  extends Person implements Designation {
  ...
}
```

Summary

- Interfaces express abstract capabilities
  - Capabilities are expressed in terms of methods that must be present
  - Cannot specify the intended behaviour of these functions
- Java later allowed concrete functions to be added to interfaces
  - Static functions — cannot access instance variables
  - Default functions — may be overridden
- Reintroduces conflicts in multiple inheritance
  - Subclass must resolve the conflict by providing a fresh implementation
  - Special "class wins" rule for conflict between superclass and interface
- Pitfalls of extending a language and maintaining compatibility

Private classes

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

Nested objects

- An instance variable can be a user defined type
  - `Employee` uses `Date`

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

  ...

}

public class Date {
  private int day, month year;

  ...
}
```

Nested objects

- An instance variable can be a user defined type
  - `Employee` uses `Date`
- `Date` is a public class, also available to other classes

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

   ...

}

public class Date {
  private int day, month year;

   ...
}
```

Nested objects

- An instance variable can be a user defined type
  - `Employee` uses `Date`
- `Date` is a public class, also available to other classes
- When could a private class make sense?

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

  ...

}

public class Date {
  private int day, month year;

  ...
}
```

Nested objects

- LinkedList is built using Node

```
public class Node {
  public Object data;
  public Node next;
  ...
}

public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return(returnval);
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList

```
public class Node {
  public Object data;
  public Node next;
  ...
}

public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return(returnval);
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList
- Instead, make Node a private class
  - Nested within LinkedList
  - Also called an inner class

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){
    ...
  }

  private class Node {
    public Object data;
    public Node next;
    ...
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList
- Instead, make Node a private class
  - Nested within LinkedList
  - Also called an inner class
- Objects of private class can see private components of enclosing class

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){
    ...
  }

  private class Node {
    public Object data;
    public Node next;
    ...
  }
}
```

Summary

- An object can have nested objects as instance variables
- In some situations, the structure of these nested objects need not be exposed
- Private classes allow an additional degree of data encapsulation

Summary

- An object can have nested objects as instance variables
- In some situations, the structure of these nested objects need not be exposed
- Private classes allow an additional degree of data encapsulation
- Combine private classes with interfaces to provide controlled access to the state of an object

Controlled interaction with objects

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access
- Update date as a whole, rather than individual components

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access
- Update date as a whole, rather than individual components
- Does this provide sufficient control?

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

Querying a database

- Object stores train reservation information
  - Can query availability for a given train, date

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying
- Need to connect the query to the logged in status of the user

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying
- Need to connect the query to the logged in status of the user
- "Interaction with state"

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public QueryObject login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }

  private class QueryObject {
    public int getStatus(int trainno, Date d) {
      // Return number of seats available
      // on train number trainno on date d
      ...
    }
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public QueryObject login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }

  private class QueryObject {
    public int getStatus(int trainno, Date d) {
      // Return number of seats available
      // on train number trainno on date d
      ...
    }
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?
- Use an interface!
  - Interface describes the capability of the object returned on login

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
      ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
     ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries
- Limit the number of queries per login?

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
      ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries
- Limit the number of queries per login?
- Maintain a counter
  - Add instance variables to object returned on login
  - Query object can remember the state of the interaction

```
public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    private int numqueries;
    private static int QLIM;

    public int getStatus(int trainno, Date d){
      if (numqueries < QLIM){
        // respond, increment numqueries
      }
    }
  }
}
```

Summary

- Can provide controlled access to an object
- Combine private classes with interfaces
- External interaction is through an object of the private class
- Capabilities of this object are known through a public interface
- Object can maintain instance variables to track the state of the interaction

Callbacks

Madhavan Mukund

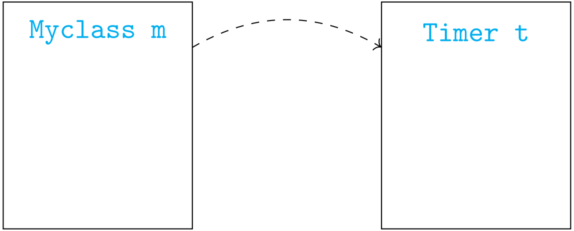
https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

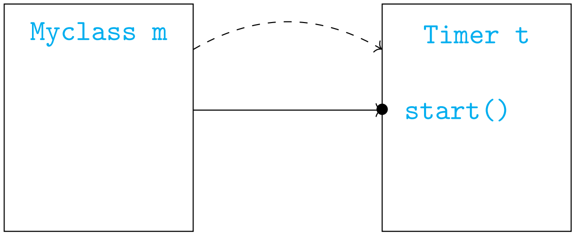
Implementing a call-back facility

- `Myclass m` creates a `Timer t`



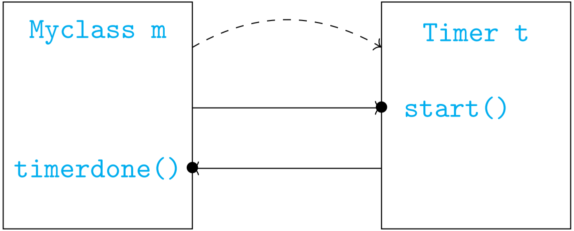
Implementing a call-back facility

- `Myclass m` creates a `Timer t`
- Start `t` to run in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!



Implementing a call-back facility

- `Myclass m` creates a `Timer t`
- Start `t` to run in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!
- `Timer t` notifies `Myclass m` when the time limit expires
  - Assume `Myclass m` has a function `timerdone()`



Implementing callbacks

- Code for `Myclass`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel
- `Timer` specific to `Myclass`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel
- `Timer` specific to `Myclass`
- Create a generic `Timer`?

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

A generic timer

- Use Java class hierarchy

A generic timer

- Use Java class hierarchy
- Parameter of `Timer` constructor of type `Object`
  - Compatible with all caller types

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    ((Myclass) owner).timerdone();
    // I'm done
  }
}
```

A generic timer

- Use Java class hierarchy
- Parameter of `Timer` constructor of type `Object`
  - Compatible with all caller types
- Need to cast `owner` back to `Myclass`

```
public class Myclass{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    ((Myclass) owner).timerdone();
    // I'm done
  }
}
```

Use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

Use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

- Modify `Myclass` to implement `Timerowner`

```
public class Myclass
   implements Timerowner{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

Use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

- Modify `Myclass` to implement `Timerowner`
- Modify `Timer` so that `owner` is compatible with `Timerowner`

```
public class Myclass
   implements Timerowner{

  public void f(){
    ..
    Timer t =
      new Timer(this);
      // this object
      // created t
    ...
    t.start(); // Start t
    ...
  }

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel
  private Timerowner owner;

  public Timer(Timerowner o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Summary

- Callbacks are useful when we spawn a class in parallel
- Spawned object notifies the owner when it is done
- Can also notify some other object when done
  - `owner` in `Timer` need not be the object that created the `Timer`
- Interfaces allow this callback to be generic
  - `owner` has to have the capability to be notified

Iterators

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 4

Linear list

- A generic linear list of objects

Linear list

- A generic linear list of objects
- Internal implementation may vary

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation

```
public class Linearlist {
  // Array implementation
  private int limit = 100;
  private Object[] data = new Object[limit];
  private int size;  // Current size

  public Linearlist(){ size = 0; }

  public void append(Object o){
    data[size] = o;
    size++;
    ...
  }
  ...
}
```

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation
- A linked list implementation

```
public class Linearlist {
  private Node head;
  private int size;

  public Linearlist(){ size = 0; }

  public void append(Object o){
    Node m;

    for (m = head; m.next != null; m = m.next){}
    Node n = new Node(o);
    m.next = n;

    size++;
  }
  ...
  private class Node (...}
}
```

Iteration

- Want a loop to run through all values in a linear list

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access . . .

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access . . .
- . . . and we don't know which implementation is in use!

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iterators

- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

Iterators

- Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

- Encapsulate this functionality in an interface called `Iterator`

```
public interface Iterator{
  public abstract boolean has_next();
  public abstract Object get_next();
}
```

Iterators

- How do we implement `Iterator` in `Linearlist`?

Iterators

- How do we implement `Iterator` in `Linearlist`?
- Need a “pointer” to remember position of the iterator

Iterators

- How do we implement `Iterator` in `Linearlist`?
- Need a “pointer” to remember position of the iterator
- How do we handle nested loops?

```
for (i = 0; i < data.length; i++){
  for (j = 0; j < data.length; j++){
      ... // do something with data[i] and data[j]
  }
}
```

Iterators

- Solution: Create an `Iterator` object and export it!

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{

  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }

 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return(it);
  }
 }
```

Iterators

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{

  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }

 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return(it);
  }
 }
```

- Definition of `Iter` depends on linear list

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
...
```

Iterators

- Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator();

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
...
```

- For nested loops, acquire multiple iterators!

```
Linearlist l = new Linearlist();
...
Object oi,oj;
Iterator i,j;

i = l.get_iterator();
while (i.has_next()){
  oi = i.get_next();
  j = l.get_iterator();
  while (j.has_next()){
    oj = j.get_next();
    ... // do something with oi, oj
  }
}
...
```

Summary

- Iterators are another example of interaction with state
  - Each iterator needs to remember its position in the list
- Export an object with a prespecified interface to handle the interaction
- The new Java `for` over lists implicitly constructs and uses an iterator

```
for (type x : a)
  do something with x;
}
```

Programming Concepts Using Java

Week 4 Revision

Abstract classes

- Sometimes we collect together classes under a common heading
- Classes Swiggy, Zomato and UberEat are all food order
- Create a class FoodOrder so that Swiggy, Zomato and UberEat extend FoodOrder
- We want to force every FoodOrder class to define a function

  `public void order() {}`
- Now we should force every class to define the public void order();
- Provide an abstract definition in FoodOrder
- public abstract void order();

Interfaces

- An interface is a purely abstract class
- All methods are abstract by default
- All data members are final by default
- If any class implement an interface, it should provide concrete code for each abstract method
- Classes can implement multiple interfaces
- Java interfaces extended to allow static and default methods from JDK 1.8 onwards
- If two interfaces has same default/static methods then its implemented class must provide a fresh implementation
- If any class wants to extend another class and an interface then it should inherit the class and implements interface

private classes

- An instance variable can be a user defined type

```
public class BookMyshow{
     String user;
     int tickets;
     Payment payement;
}
public class Payment{
    int cardno;
    int cvv;
}
```

- Payment is a public class, also available to other classes
- Payment class has sensitive information, so there is a security concern.

private classes

- We cannot declare Payment class as private outside the BookMyshow class
- You can declare Payment class as private inside the BookMyshow class

```
public class BookMyshow{
     String user;
     int tickets;
     Payment payement;
     private class Payment{
        int cardno;
        int cvv;
     }
}
```

- Now Payment class is a private member of the BookMyshow class
- Now Payment class only available to the BookMyshow class

Interaction with State(Manipulating objects)

- Consider the class student below.
- Student class is encapsulated by private variables.

```
public class Student{
     private String rollno;
     private String name;
     private int age;
     //3 mutator methods
     //3 Accessor methods
}
```

- Consider Student class has student1,student2.....student60 objects
- Update date as a whole, rather than individual components

Interaction with State(Manipulating objects)


```
public class Student{
    private String rollno;
    private String name;
    private int age;
    public void setStudent(String rollno,String name,int age){
    }
}
```

- Now public void setStudent(String rollno, String name, int age) update the Student object as a whole.

Java Call back methods.

- what is call back method?

```
interface Notification{
void notification();//should be overridden in WorkingDay and Weekend
}
class WorkingDay implements Notification{
}
class Weekend implements Notification{
}
class Timer{//Timer will decide which call back function should be call
}
public class User {
    public static void main(String[] args) {
        Timer timer=new Timer();
        timer.start(new Date());
    }
}
```

Iterators

- what is Iterator?
- You can loop through any data structure using an Iterator.

```
public interface Iterator{
public abstract boolean has_next();
public abstract Object get_next();
}
```

Polymorphism revisited

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities
  - Reverse an array/list
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list

Polymorphism

- In object-oriented programming, polymorphism usually refers to the effect of dynamic dispatch
  - `S` is a subclass of `T`
  - `S` overrides a method `f()` defined in `T`
  - Variable `v` of type `T` is assigned to an object of type `S`
  - `v.f()` uses the definition of `f()` from `S` rather than `T`
- Every object "knows" what it needs to do
- More generally, polymorphism refers to behaviour that depends only a specific capabilities — structural polymorphism
  - Reverse an array/list (should work for any type)
  - Search for an element in an array/list (need equality check)
  - Sort an array/list (need to compare values)

Structural polymorphism

- Use the Java class hierarchy to simulate this

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`

```
public void reverse (Object[] objarr){
  Object tempobj;
  int len = objarr.length;
  for (i = 0; i < n/2; i++){
    tempobj = objarr[i];
    objarr[i] = objarr[(n-1)-i];
    objarr[(n-1)-i] = tempobj;
  }
}
```

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
    - `==` translates to `Object.equals()`

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length; i++){
    if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Structural polymorphism

- Use the Java class hierarchy to simulate this
- Polymorphic `reverse`
- Polymorphic `find`
  - `==` translates to `Object.equals()`
- Polymorphic `sort`
  - Use interfaces to capture capabilities

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}

public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

Type consistency

- Polymorphic function to copy an array

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}


Date[] datearr = new Date[10];
Employee[] emparr = new Employee[10];

arraycopy(datearr,emparr); // Run-time error
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}


public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(etktarr,tktarr); // Allowed
```

Type consistency

- Polymorphic function to copy an array
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time
- More generally source array can be a subtype of the target array
- But the converse is illegal

```
public static void arraycopy (Object[] src,
                              Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}


public class Ticket {...}
public class ETicket extends Ticket{...}

Ticket[] tktarr = new Ticket[10];
ETicket[] etktarr = new ETicket[10];

arraycopy(tktarr,etktarr); // Illegal
```

Polymorphic data structures

- Arrays, lists, . . . should allow arbitrary elements

Polymorphic data structures

- Arrays, lists, ... should allow arbitrary elements
- A polymorphic list stores values of type `Object`

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

  public void insert(Object newdata){...}

  private class Node {
    private Object data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Arrays, lists, ... should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval;
    ...
    return(returnval);
  }

  public void insert(Object newdata){...}

  private class Node {
    private Object data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Arrays, lists, ... should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){...}

  private class Node {...}
}
```

```
LinkedList list = new LinkedList();
Ticket t1,t2;

t1 = new Ticket();
list.insert(t1);
t2 = (Ticket)(list.head());
// head() returns an Object
```

Polymorphic data structures

- Arrays, lists, ... should allow arbitrary elements
- A polymorphic list stores values of type `Object`
- Two problems
  - Type information is lost, need casts
  - List need not be homogenous!

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){...}

  private class Node {...}
}
```

```
LinkedList list = new LinkedList();
Ticket t = new Ticket();
Date d = new Date();
list.insert(t);
list.insert(d);
...
```

Generic programming in Java

- Java added generic programming to address these issues
- Classes and functions can have type parameters
  - `class LinearList<T>` holds values of type `T`
  - `public T head(){...}` returns a value of same type `T` as enclosing class
- Can describe subclass relationships between type variables
  - `public static <S extends T,T> void arraycopy (S[] src, T[] tgt){...}`

Generic programming in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Structural polymorphism

- Functions that depends only a specific capabilities
  - Reverse an array/list — should work for any type
  - Search for an element in an array/list — need equality check
  - Sort an array/list — need to compare values
- May need to impose constraints on types of arguments
  - Copying an array needs source type to extend target type
- Polymorphic data structures
  - Hold values of an arbitrary type
  - Homogenous
  - Should not have to cast return values

Java Generics

- Use type variables

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` . . . "

```
public <T> void reverse (T[] objarr){
  T tempobj;
  int len = objarr.length;
  for (i = 0; i < n/2; i++){
    tempobj = objarr[i];
    objarr[i] = objarr[(n-1)-i];
    objarr[(n-1)-i] = tempobj;
  }
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error

```
public <T> int find (T[] objarr, T o){
  int i;
  for (i = 0; i < objarr.length; i++){
    if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - “For every type `T` . . . ”
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical

```
public static <T> void arraycopy (T[] src,
                                  T[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - "For every type `T` ..."
- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error
- Polymorphic `arraycopy`
  - Source and target types must be identical
- A more generous `arraycopy`
  - Source and target types may be different
  - Source type must extend target type

```
public static <S extends T,T>
              void arraycopy (S[] src,
                              T[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length);
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

Polymorphic data structures

- A polymorphic list

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter T applies to the class as a whole
- Internally, the T in Node is the same T

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- A polymorphic list
- The type parameter `T` applies to the class as a whole
- Internally, the `T` in `Node` is the same `T`
- Also the return value of `head()` and the argument of `insert()`
- Instantiate generic classes using concrete type

```
public class LinkedList<T>{
    ...
}

LinkedList<Ticket> ticketlist =
          new LinkedList<Ticket>();
LinkedList<Date> datelist =
          new LinkedList<Date>();

Ticket t = new Ticket();
Date d = new Date();

ticketlist.insert(t);
datelist.insert(d);
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

- `T` in the argument of `insert()` is a new `T`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

```
public <T> void
           insert(T newdata){...}
```

- `T` in the argument of `insert()` is a new `T`
- Quantifier `<T>` masks the type parameter `T` of `LinkedList`

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Polymorphic data structures

- Be careful not to accidentally hide a type variable

  ```
  public <T> void
             insert(T newdata){...}
  ```

- T in the argument of insert() is a new T
- Quantifier <T> masks the type parameter T of LinkedList
- Contrast with

  ```
  public <T> static void
     arraycopy (T[] src, T[] tgt){...}
  ```

```
public class LinkedList<T>{
  private int size;
  private Node first;

  public T head(){
    T returnval;
    ...
    return(returnval);
  }

  public <T> void insert(T newdata){...}

  private class Node {
    private T data;
    private Node next;
    ...
  }
}
```

Summary

- Generics introduce structural polymorphism into Java through type variables
- Classes and functions can have type parameters
  - `class LinearList<T>` holds values of an arbitrary type `T`
  - `public T head(){...}` returns a value of same type `T` used when creating the list
- Can describe subclass relationships between type variables
  - `public static <S extends T,T> void arraycopy (S[] src, T[] tgt){...}`
- Be careful not to accidentally hide type variables

  `public <T> void insert(T newdata){...}` inside `class LinearList<T>`

  vs

  `public <T> static void arraycopy (T[] src, T[] tgt){...}`

Java generics and subtyping

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```

- A type error at run time!

Extending subtyping in contexts

- If `S` is compatible with `T`, `S[]` is compatible with `T[]`

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;
  // OK. ETicket[] is a subtype of Ticket[]
```

- But ...

```
...
ticketarr[5] = new Ticket();
// Not OK.  ticketarr[5] refers to an ETicket!
```

- A type error at run time!
- Java array typing is covariant
  - If `S extends T` then `S[] extends T[]`

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generics and subtypes

- Generic classes are not covariant
  - `LinkedList<String>` is not compatible with `LinkedList<Object>`
- The following will not work to print out an arbitrary `LinkedList`

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<Object> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- How can we get around this limitation?

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- `<T>` is a type quantifier: *For every type* `T`, ...

Generic methods

- As we have seen, we can make the method generic by introducing a type variable

```
public class LinkedList<T>{...}

public static <T> void printlist(LinkedList<T> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
```

- `<T>` is a type quantifier: *For every type* `T`, ...
- Note that `T` is not actually used inside the function
  - We use `Object o` as a generic variable to cycle through the list

Wildcards

- Instead, use ? as a wildcard type variable

```java
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

Wildcards

- Instead, use `?` as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- `?` stands for an arbitrary unknown type

Wildcards

- Instead, use ? as a wildcard type variable

```
public class LinkedList<T>{...}

public static void printlist(LinkedList<?> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    System.out.println(o);
  }
}
```

- ? stands for an arbitrary unknown type
- Avoids unnecessary type variable quantification when the type variable is not needed elsewhere

Wildcards

- Can define variables of a wildcard type

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

Wildcards

- Can define variables of a wildcard type

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

- But need to be careful about assigning values

```
public class LinkedList<T>{...}

LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

Wildcards

- Can define variables of a wildcard type

```
public class LinkedList<T>{...}

LinkedList<?> l;
```

- But need to be careful about assigning values

```
public class LinkedList<T>{...}

LinkedList<?> l = new LinkedList<String>();
l.add(new Object()); // Compile time error
```

- Compiler cannot guarantee the types match

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`

Bounded wildcards

- Suppose `Circle`, `Square` and `Rectangle` all extend `Shape`
- `Shape` has a method `draw()`
- All subclasses override `draw()`

Bounded wildcards

- Suppose Circle, Square and Rectangle all extend Shape
- Shape has a method draw()
- All subclasses override draw()
- Want a function to draw all elements in a list of Shape compatible objects

```
public static void drawAll(LinkedList<? extends Shape> l){
  Object o;
  Iterator i = l.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    o.draw();
  }
}
```

Bounded wildcards

- Copying a LinkedList, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

Bounded wildcards

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T,T>
       void listcopy (LinkedList<?> src,
                      LinkedList<T> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

- Can reverse the constraint, using `super`

```
public static <T,? super T>
       void listcopy (LinkedList<T> src,
                      LinkedList<?> tgt){
  Object o;
  Iterator i = srt.get_iterator();
  while (i.has_next()){
    o = i.get_next();
    trt.add(o);
  }
}
```

Summary

- Java generics are not covariant, unlike arrays
- Cannot substitute `Object` for `T` to get most general type
- Instead, use type quantification `<T>` or wild card type variable `?`
- Wild card can be used wherever the type `T` is not required within the function
  - When `T` is not needed for return type, or to declare local variables
- Wild cards can be bounded
  - `LinkedList<? extends T>`
  - `LinkedList<? super T>`

Reflection

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Reflection

Wikipedia

Reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour.

Reflection

Wikipedia

Reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour.

- Two components involved in reflection

Reflection

Wikipedia

Reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour.

- Two components involved in reflection
  - Introspection

    A program can observe, and therefore reason about its own state.

Reflection

Wikipedia

Reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behaviour.

- Two components involved in reflection
  - Introspection

    A program can observe, and therefore reason about its own state.
  - Intercession

    A program can modify its execution state or alter its own interpretation or meaning.

Reflection in Java

- Simple example of introspection

```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

Reflection in Java

- Simple example of introspection

```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

- What if we don't know the type that we want to check in advance?

Reflection in Java

- Simple example of introspection

```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

- What if we don't know the type that we want to check in advance?
- Suppose we want to write a function to check if two different objects are both instances of the same class?

```
public static boolean classequal(Object o1, Object o2){
  ...
  // return true iff o1 and o2 point to objects of same type
  ...
}
```

Reflection in Java ...

```
public static boolean classequal(Object o1, Object o2){...}
```

Reflection in Java . . .

```
public static boolean classequal(Object o1, Object o2){...}
```

- Can't use `instanceof`
  - Will have to check across all defined classes
  - This is not even a fixed set!

Reflection in Java ...

```
public static boolean classequal(Object o1, Object o2){...}
```

- Can't use `instanceof`
  - Will have to check across all defined classes
  - This is not even a fixed set!
- Can't use generic type variables
  - The following code is syntactically disallowed

    ```
    if (o1 instance of T) { ...}
    ```

Introspection in Java

- Can extract the class of an object using `getClass()`

Introspection in Java

- Can extract the class of an object using `getClass()`
- Import package `java.lang.reflect`

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

Introspection in Java

- Can extract the class of an object using `getClass()`
- Import package `java.lang.reflect`

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

- What does `getClass()` return?

Introspection in Java

- Can extract the class of an object using `getClass()`
- Import package `java.lang.reflect`

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

- What does `getClass()` return?
- An object of type `Class` that encodes class information

The class `Class`

- A version of `classequal` the explicitly uses this fact

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

The class `Class`

- A version of `classequal` the explicitly uses this fact

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

- For each currently loaded class `C`, Java creates an object of type `Class` with information about `C`

The class Class

- A version of classequal the explicitly uses this fact

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

- For each currently loaded class C, Java creates an object of type Class with information about C
- Encoding execution state as data — reification
    - Representing an abstract idea in a concrete form

Using the Class object

- Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
  // Create a new object of same type as obj
...
```

Using the Class object

- Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
  // Create a new object of same type as obj
...
```

- Can also get hold of the class object using the name of the class

```
...
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
...
```

Using the Class object

- Can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
  // Create a new object of same type as obj
...
```

- Can also get hold of the class object using the name of the class

```
...
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
...
```

- . . . , or, more compactly

```
...
Object o = Class.forName("Manager").newInstance();
...
```

The class `Class` ...

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`

The class `Class` ...

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- Constructors, methods and fields themselves have structure
  - Constructors: arguments
  - Methods : arguments and return type
  - All three: modifiers `static`, `private` etc

The class `Class` ...

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- Constructors, methods and fields themselves have structure
  - Constructors: arguments
  - Methods : arguments and return type
  - All three: modifiers `static`, `private` etc
- Additional classes `Constructor`, `Method`, `Field`

The class `Class` ...

- From the `Class` object for class `C`, we can extract details about constructors, methods and fields of `C`
- Constructors, methods and fields themselves have structure
  - Constructors: arguments
  - Methods : arguments and return type
  - All three: modifiers `static`, `private` etc
- Additional classes `Constructor`, `Method`, `Field`
- Use `getConstructors()`, `getMethods()` and `getFields()` to obtain constructors, methods and fields of `C` in an array.

The class `Class` ...

- Extracting information about constructors, methods and fields

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
...
```

The class `Class` ...

- Extracting information about constructors, methods and fields

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
...
```

- `Constructor`, `Method`, `Field` in turn have functions to get further details

The class Class ...

- Example: Get the list of parameters for each constructor

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
  Class params[] = constructors[i].getParameterTypes();
  ..
}
```

The class Class ...

- Example: Get the list of parameters for each constructor

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
  Class params[] = constructors[i].getParameterTypes();
  ..
}
```

- Each parameter list is a list of types
  - Return value is an array of type `Class[]`

The class `Class` ...

- We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }
  // construct an array  of arguments
methods[3].invoke(obj,args);
  // invoke methods[3] on obj with arguments args
...
```

The class `Class` ...

- We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }
  // construct an array  of arguments
methods[3].invoke(obj,args);
  // invoke methods[3] on obj with arguments args
...

Field[] fields = c.getFields();
Object o =  fields[2].get(obj);
   // get the value of fields[2] from obj
...
fields[3].set(obj,value);
  // set the value of fields[3] in obj to value
...
```

Reflection and security

- Can we extract information about private methods, fields, ...?

Reflection and security

- Can we extract information about private methods, fields, ...?
- `getConstructors()`, ...only return publicly defined values

Reflection and security

- Can we extract information about private methods, fields, ...?
- `getConstructors()`, ... only return publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`

Reflection and security

- Can we extract information about private methods, fields, ...?
- `getConstructors()`, ...only return publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`
- Should this be allowed to all programs?

Reflection and security

- Can we extract information about private methods, fields, ...?
- `getConstructors()`, ...only return publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`
- Should this be allowed to all programs?
- Security issue!

Reflection and security

- Can we extract information about private methods, fields, . . . ?
- `getConstructors()`, . . . only return publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`
- Should this be allowed to all programs?
- Security issue!
- Access to private components may be restricted through external security policies

Using reflection

- `BlueJ`, a programming environment to learn Java

Using reflection

- `BlueJ`, a programming environment to learn Java
- Can define and compile Java classes

Using reflection

- BlueJ, a programming environment to learn Java
- Can define and compile Java classes
- For compiled code, create object, invoke methods, examine state

Using reflection

- BlueJ, a programming environment to learn Java
- Can define and compile Java classes
- For compiled code, create object, invoke methods, examine state
- Uses reflective capabilities of Java — BlueJ need not internally maintain "debugging" information about each class

Using reflection

- BlueJ, a programming environment to learn Java
- Can define and compile Java classes
- For compiled code, create object, invoke methods, examine state
- Uses reflective capabilities of Java — BlueJ need not internally maintain "debugging" information about each class
- See http://www.bluej.org

Limitations of Java reflection

- Cannot create or modify classes at run time
  - The following is not possible
    ```
    Class c = new Class(....);
    ```
  - An environment like `BlueJ` must invoke Java compiler before you can use a new class

Limitations of Java reflection

- Cannot create or modify classes at run time
  - The following is not possible

    ```
    Class c = new Class(....);
    ```

  - An environment like `BlueJ` must invoke Java compiler before you can use a new class
- Contrast with Python
  - `class XYZ:` can be executed at runtime in Python

Limitations of Java reflection

- Cannot create or modify classes at run time
  - The following is not possible
    ```
    Class c = new Class(....);
    ```
  - An environment like `BlueJ` must invoke Java compiler before you can use a new class
- Contrast with Python
  - `class XYZ:` can be executed at runtime in Python
- Other OO languages like Smalltalk allow redefining methods at run time

Java generics at run time

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 5

Erasure of generic information

- Type erasure — Java does not keep record all versions of `LinkedList<T>` as separate types
  - Cannot write

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```

Erasure of generic information

- Type erasure — Java does not keep record all versions of `LinkedList<T>` as separate types
  - Cannot write

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```
- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`

Erasure of generic information

- Type erasure — Java does not keep record all versions of `LinkedList<T>` as separate types
  - Cannot write

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```

- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Or, the upper bound, if one is available
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`

Erasure of generic information

- Type erasure — Java does not keep record all versions of `LinkedList<T>` as separate types
  - Cannot write

    ```
    if (s instanceof LinkedList<String>){ ... }
    ```

- At run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Or, the upper bound, if one is available
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`
- Since no information about `T` is preserved, cannot use `T` in expressions like

  ```
  if (o instanceof T) {...}
  ```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

- As a consequence the following overloading is illegal

```
public class Example {
    public void printlist(LinkedList<String> strList) { }
    public void printlist(LinkedList<Date> dateList) { }
}
```

Erasure and overloading

- Type erasure means the comparison in following code fragment returns `True`

```
o1 = new LinkedList<Employee>();
o2 = new LinkedList<Date>();

if (o1.getClass() == o2.getClass){
   // True, so this block is executed
}
```

- As a consequence the following overloading is illegal

```
public class Example {
    public void printlist(LinkedList<String> strList) { }
    public void printlist(LinkedList<Date> dateList) { }
}
```

- Both functions have the same signature after type erasure

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;           // OK
newarray = new T[100]; // Cannot create!
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;            // OK
newarray = new T[100]; // Cannot create!
```

- An ugly workaround ...

```
T[] newarray;
newarray = (T[]) new Object[100];
```

Arrays and generics

- Recall the covariance problem for arrays
  - If `S extends T` then `S[] extends T[]`
- Can lead to run time type errors

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;  // OK. ETicket[] is a subtype of Ticket[]
...
ticketarr[5] = new Ticket();  // Not OK.  ticketarr[5] refers to an ETicket!
```

- To avoid similar problems, can declare a generic array, but cannot instantiate it

```
T[] newarray;           // OK
newarray = new T[100]; // Cannot create!
```

- An ugly workaround . . . generates a compiler warning but works!

```
T[] newarray;
newarray = (T[]) new Object[100];
```

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .
- Wrapper class for each basic type:

Basic type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`

Basic type	Wrapper Class
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Wrapper classes

- Type erasure — at run time, all type variables are promoted to `Object`
  - `LinkedList<T>` becomes `LinkedList<Object>`
- Basic types `int`, `float`, . . . are not compatible with `Object`
- Cannot use basic type in place of a generic type variable `T`
  - Cannot instantiate `LinkedList<T>` as `LinkedList<int>`, `LinkedList<double>`, . . .
- Wrapper class for each basic type:

Basic type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`

Basic type	Wrapper Class
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

- All wrapper classes other than `Boolean`, `Character` extend the class `Number`

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...

- Autoboxing — implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

Wrapper classes

- Converting from basic type to wrapper class and back

```
int x = 5;
Integer myx = Integer(x);
int y = myx.intValue();
```

- Similarly, `byteValue()`, `doubleValue()`, ...
- Autoboxing — implicit conversion between base types and wrapper types

```
int x = 5;
Integer myx = x;
int y = myx;
```

- Use wrapper types in generic data structures

Summary

- Java generics come with some restrictions
- Information about type variables is erased at runtime
  - `LinkedList<T>` becomes `LinkedList<Object>`
  - `LinkedList<? extends Shape>` becomes `LinkedList<Shape>`
- Limits the use reflection on generic types — cannot write
  - `if (o instanceof LinkedList<String>) {...}`
  - `if (o instanceof T) {...}`
- Cannot overload function signatures using instantiation of generic types
- Cannot instantiate arrays of generic type
- Need to box built-in types using wrapper types

The benefits of indirection

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Abstract data types

- Separate public interface from private implementation

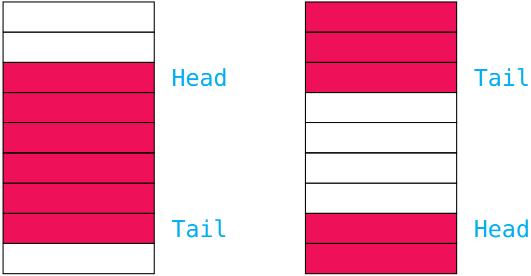
Abstract data types

- Separate public interface from private implementation
- For instance, a (generic) queue

```
public class Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

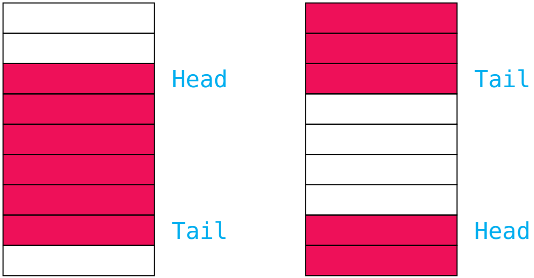
Abstract data types

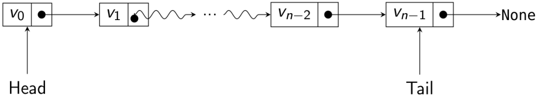
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array



Abstract data types

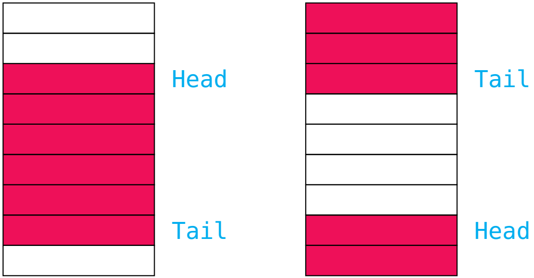
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list

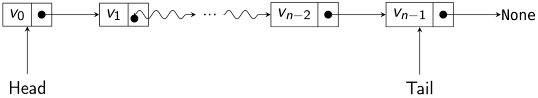




Abstract data types

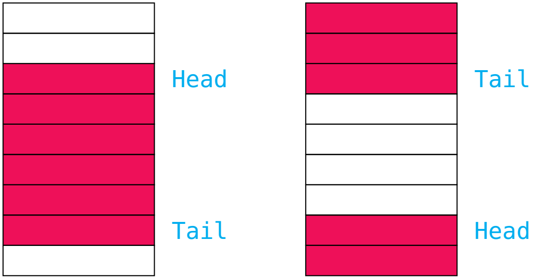
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one

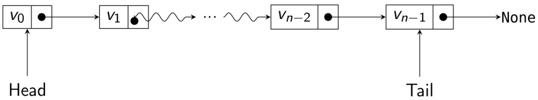




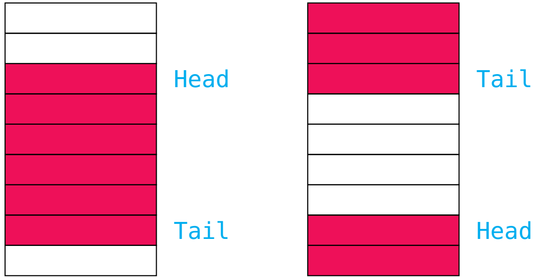
Abstract data types

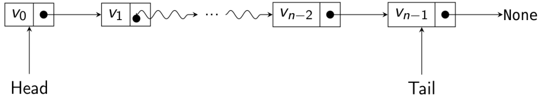
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one
- Public interface is unchanged





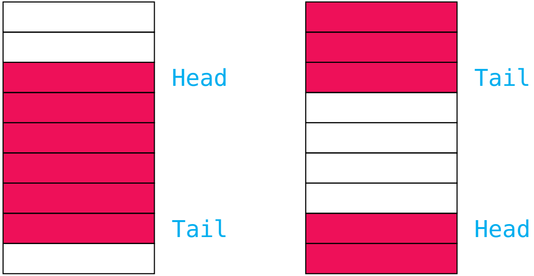
- Is the user indifferent to choice of implementation?

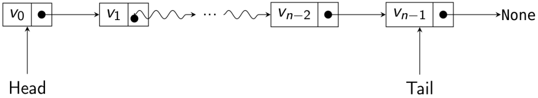




Abstract data types . . .

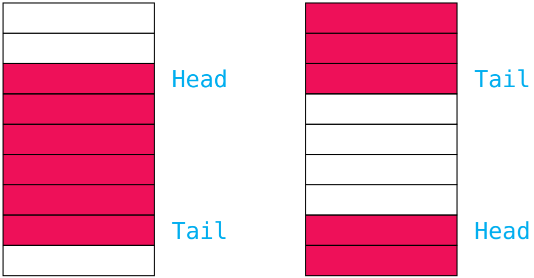
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects

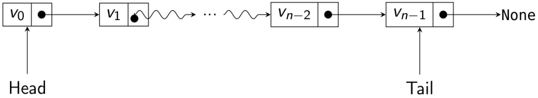




Abstract data types . . .

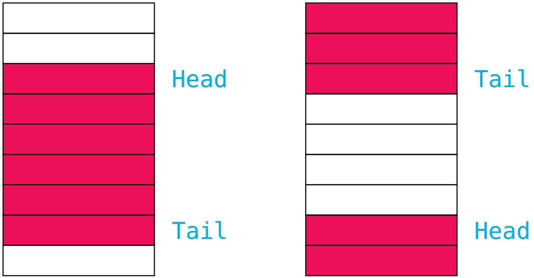
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation

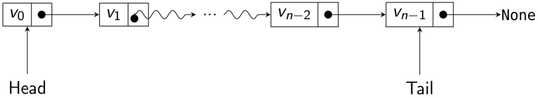




Abstract data types . . .

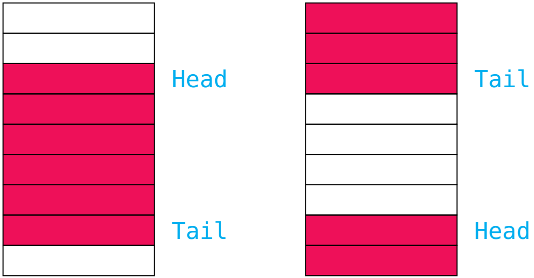
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size

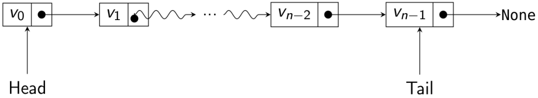




Abstract data types ...

- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size
- Offer user a choice of implementation?





Multiple impementations

- Create two separate implementations

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}

public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}

public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}


public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`
- And also every function header, auxiliary variable, ... associated with it

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- Benefit of indirection — to use a different implementation for `dateq`, only need to update the instantiation

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Summary

- Use interfaces to flexibly choose between multiple concrete implementations

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
    - Organization provides senior staff with an office car
    - Concrete: each official has an assigned car — what if it breaks down?
    - Indirection: a pool of office cars, use any that is available

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

"Fundamental theorem of software engineering"

All problems in computer science can be solved by another level of indirection.

Butler Lampson, Turing Award 1992

Collections

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates
- Instead, organize these data structures by functionality

Built-in data types

- Most programming languages provide built-in collective data types
  - Arrays, lists, dictionaries, ...
- Java originally had many such pre-defined classes
  - `Vector`, `Stack`, `Hashtable`, `Bitset`, ...
- Choose the one you need
- ...but changing a choice requires multiple updates
- Instead, organize these data structures by functionality
- Create a hierarchy of abstract interfaces and concrete implementations
  - Provide a level of indirection

The `Collection` interface

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The `Collection` interface

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries
- `add()` — add to the collection

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}
```

The Collection interface

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries
- `add()` — add to the collection
- `iterator()` — get an object that implements `Iterator` interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

The Collection interface

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- `add()` — add to the collection
- `iterator()` — get an object that implements `Iterator` interface
- Use iterator to loop through the elements

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  ...
}

public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

  ```
  if (element.equals(obj))
  ```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Using iterators

- Use iterator to loop through the elements
- Java later added "for each" loop
  - Implicitly creates an iterator and runs through it
- Generic functions to operate on collections
- How does this line work?

  ```
  if (element.equals(obj))
  ```
- Later!

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

```
Collection<String> cstr = new ...;
for (String element : cstr){
  // do something with element
}
```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
  return false;
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}

Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // Delete element if it has some property
  if (property(element)) {
     iter.remove();
  }
}
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.remove(); // Error
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();
...
iter.remove();
iter.next();
iter.remove();
```

Removing elements

- Iterator also has a `remove()` method
  - Which element does it remove?
- The element that was last accessed using `next()`
- To remove consecutive elements, must interleave a `next()`
- To remove the first element, need to access it first

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

```
Collection<String> cstr = new ...;
Iterator<String> iter = cstr.iterator();

// Remove first element in cstr
iter.next();
iter.remove();
```

The `Collection` interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

The Collection interface — the full story

- How does this line work?

  `if (element.equals(obj))`

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}
```

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The `Collection` interface — the full story

- How does this line work?

  ```
  if (element.equals(obj))
  ```

- Actually, `Collection` defines a much larger set of abstract methods
  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - A different `remove()` from the one in `Iterator`
- To implement the `Collection` interface, need to implement all these methods!

```
public static <E> boolean
       contains(Collection<E> c, Object obj) {
  for (E element : c)
    if (element.equals(obj))
      return true;
    return false;
}

public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement Collection, need to implement all these methods!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The `AbsractCollection` class

- To implement `Collection`, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The `AbsractCollection` class

- To implement `Collection`, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!

```
public interface Collection<E>{
  boolean add(E element);
  Iterator<E> iterator();
  int size() boolean isEmpty();
  boolean contains(Object obj);
  boolean containsAll(Collection<?> c);
  boolean equals(Object other);
  boolean addAll(Collection<? extends E> from);
  boolean remove(Object obj);
  boolean removeAll(Collection<?> c);
  ...
}
```

The AbsractCollection class

- To implement `Collection`, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, `AbstractCollection` abstract class implements `Collection`

```
public abstract class AbstractCollection<E>
                      implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

The AbsractCollection class

- To implement `Collection`, need to implement all these methods!
- "Correct" solution — provide default implementations in the interface
- Added to Java interfaces later!
- Instead, `AbstractCollection` abstract class implements `Collection`
- Concrete classes now extend `AbstractCollection`
  - Need to define `iterator()` based on internal representation
  - Can choose to override `contains()`, ...

```
public abstract class AbstractCollection<E>
                         implements Collection<E> {
  ...
  public abstract Iterator<E> iterator();

  public boolean contains(Object obj) {
    for (E element : this)
      if (element.equals(obj))
        return true;
    return false;
  }
  ...
}
```

Summary

- The `Collection` interface captures abstract properties of collections
  - Add an element, create an iterator, ...
- Can use for each loop to avoid explicit iterator
- Write generic functions that operate on collections
- `Collection` defines many additional abstract functions, tedious if we have to implement each of them
- `AbstractCollection` provides default implementations to many functions required by `Collection`
- Concrete implementations of collections extend `AbstractCollection`

Concrete Collections

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Built-in data types

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries

Built-in data types

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?

Built-in data types

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries
- Collections can be further organized based on additional properties
  - Are the elements ordered?
  - Are duplicates allowed?
  - Are there constraints on how elements are added, removed?
- In the spirit of indirection, these are captured by interfaces that extend `Collection`
  - Interface `List` for ordered collections
  - Interface `Set` for collections without duplicates
  - Interface `Queue` for ordered collections with constraints on addition and deletion

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access

```
public interface List<E>
                 extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);
}
```

The `List` interface

- An ordered collection can be accessed in two ways
  - Through an iterator
  - By position — random access
- Additional functions for random access
- `ListIterator` extends `Iterator`
  - `void add(E element)` to insert an element before the current index
  - `void previous()` to go to previous element
  - `boolean hasPrevious()` checks that it is legal to go backwards

```java
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The `List` interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index `i`
  - In a linked list, must start at the beginning and traverse `i` links

```
public interface List<E>
                extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

The `List` interface and random access

- Random access is not equally efficient for all ordered collections
  - In an array, can compute location of element at index `i`
  - In a linked list, must start at the beginning and traverse `i` links
- Tagging interface `RandomAccess`
  - Tells us whether a `List` supports random access or not
  - Can choose algorithmic strategy based on this

```
public interface List<E>
             extends Collection<E>{
  void add(int index, E element);
  void remove(int index);
  E get(int index);
  E set(int index, E element);

  ListIterator<E> listIterator();
}
```

```
if (c instanceof RandomAccess) {
  // use random access algorithm
} else {
  // use sequential access algorithm
}
```

The `AbstractList` interface

- Recall that `AbstractCollection` is the "usable" version of `Collection`

The `AbstractList` interface

- Recall that `AbstractCollection` is the “usable” version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations

The `AbstractList` interface

- Recall that `AbstractCollection` is the "usable" version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further subclass to distinguish lists without random access

The `AbstractList` interface

- Recall that `AbstractCollection` is the “usable” version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further subclass to distinguish lists without random access
- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Internally, the usual flexible linked list
  - Efficient to add and remove elements at arbitrary positions

The AbstractList interface

- Recall that `AbstractCollection` is the "usable" version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further subclass to distinguish lists without random access
- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Internally, the usual flexible linked list
  - Efficient to add and remove elements at arbitrary positions
- Concrete generic class `ArrayList<E>` extends `AbstractList`
  - Flexible size array, supports random access

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available
  - This loop will execute a fresh scan from start to element `i` in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available
  - This loop will execute a fresh scan from start to element `i` in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of `add()` available
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before current position of the iterator

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available
  - This loop will execute a fresh scan from start to element `i` in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of `add()` available
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before current position of the iterator
- In `Collection`, `add()` returns `boolean` — did the add update the collection?
  - `add()` may not update a set, always works for lists

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Not random access
  - But random access methods of `AbstractList` are still available
  - This loop will execute a fresh scan from start to element `i` in each iteration!

```
LinkedList<String> list = new ...;

for (int i = 0; i < list.size(); i++)
  // do something with list.get(i);
```

- Two versions of `add()` available
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before current position of the iterator
- In `Collection`, `add()` returns `boolean` — did the add update the collection?
  - `add()` may not update a set, always works for lists
- `add()` in `ListIterator` returns `void`

The Set interface

- A set is a collection without duplicates

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function
- Or arrange values in a two dimensional structure
  - Balanced search tree

The Set interface

- A set is a collection without duplicates
- `Set` interface is identical to `Collection`, but behaviour is more constrained
  - `add()` should have no effect, and return `false`, if the element already exists
  - `equals()` should return `true` if contents match after disregarding order
- Two interfaces, same signature?
- Use `Set` to constrain values to satisfy additional constraints
- `Set` implementations typically designed to allow efficient membership tests
- Ordered collections loop through a sequence to find an element
- Instead, map the value to its position
  - Hash function
- Or arrange values in a two dimensional structure
  - Balanced search tree
- As usual, concrete set implementations extend `AbstractSet`, which extends `AbstractCollection`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order

Concrete sets

- `HashSet` implements a hash table
  - Underlying storage is an array
  - Map value `v` to a position `h(v)`
  - If `h(v)` is unoccupied, store `v` at that position
  - Otherwise, collision — different strategies to handle this case
- Checking membership is fast — check if `v` is at position `h(v)`
- Unordered, but supports `iterator()`
- Scan elements in unspecified order
- Visit each element exactly once

- `TreeSet` uses a tree representation
  - Values are ordered
  - Maintains a sorted collection
- Iterator will visit elements in sorted order
- Insertion is more complex than a hash table
  - Time $O(\log n)$ if the set has n elements

The `Queue` interface

- Ordered, remove front, insert rear

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error

- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible

- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface `Deque`, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface `Deque`, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

- Interface `PriorityQueue`
  - `remove()` returns highest priority item

The Queue interface

- Ordered, remove front, insert rear
- `Queue` interface supports the following

```
boolean add(E element);
E remove();
```

  - If queue full, `add()` flags an error
  - If queue empty, `remove()` flags an error
- Gentler versions of `add()`, `remove()`

```
boolean offer(E element);
E poll();
```

  - Return `false` or `null`, respectively, if not possible
- Inspect the head, no update

```
E element(); // Throws exception
E peek();    // Returns null
```

- Interface `Deque`, double ended queue

```
boolean addFirst(E element);
boolean addLast(E element);
boolean offerFirst(E element);
boolean offerLast(E element);
E pollFirst();
E pollLast();
E getFirst();
E getLast();
E peekFirst();
E peekLast();
```

- Interface `PriorityQueue`
  - `remove()` returns highest priority item
- Concrete implementations
  - `LinkedList` — implements `Queue`
  - `ArrayDeque` — circular array `Deque`

Summary

- Different types of `Collection` are specified by subinterfaces
  - `List`, `Set`, `Queue`
- `List` allows random access, more functional `ListIterator`
- `Set` constrains collection to not have duplicates
- `Queue` supports restricted add and remove methods
- Each interface has corresponding version under `AbstractCollection`
- Concrete implementations extend `AbstractList`, `AbstractSet` and `AbstractQueue`

Maps

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Maps

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries

Maps

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, . . .
  - But not key-value structures like dictionaries
- Key-value structures come under the `Map` interface
  - Two type parameters
  - `K` is the type for keys
  - `V` is the type for values
  - `get(k)` fetches value for key `k`
  - `put(k,v)` updates value for key `k`

```
public interface Map<K,V>{
  V get(Object key);
  V put(K key, V Value);

  boolean containsKey(Object key);
  boolean containsValue(Object value);
  ...
}
```

Maps

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries
- Key-value structures come under the `Map` interface
  - Two type parameters
  - `K` is the type for keys
  - `V` is the type for values
  - `get(k)` fetches value for key `k`
  - `put(k,v)` updates value for key `k`

```
public interface Map<K,V>{
  V get(Object key);
  V put(K key, V Value);

  boolean containsKey(Object key);
  boolean containsValue(Object value);
  ...
}
```

- As expected, keys form a set
  - Only one entry per key-value
  - Assigning a fresh value to existing key overwrite the old value
  - `put(k,v)` returns the previous value associated with `k`, or `null`

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry
- `Map` has the following default method

  `V getOrDefault(Object key, V defaultValue)`

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry
- `Map` has the following default method

  ```
  V getOrDefault(Object key, V defaultValue)
  ```
- For instance

  ```
  Map<String, Integer> scores = ...;
  int score = scores.getOrDefault(bat,0);
  ```

  sets `score` to `0` if key `bat` is not present

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry
- `Map` has the following default method

```
V getOrDefault(Object key, V defaultValue)
```

- For instance

```
Map<String, Integer> scores = ...;
int score = scores.getOrDefault(bat,0);
```

  sets `score` to `0` if key `bat` is not present

- Now, we can update the entry for key `bat` as follows

```
scores.put(bat,
   scores.getOrDefault(bat,0)+newscore);
   // Add newscore to value of bat
```

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry
- `Map` has the following default method

  ```
  V getOrDefault(Object key, V defaultValue)
  ```
- For instance

  ```
  Map<String, Integer> scores = ...;
  int score = scores.getOrDefault(bat,0);
  ```

  sets `score` to `0` if key `bat` is not present
- Now, we can update the entry for key `bat` as follows

  ```
  scores.put(bat,
     scores.getOrDefault(bat,0)+newscore);
     // Add newscore to value of bat
  ```
- Alternatively, use `putIfAbsent()` to initialize a missing key

  ```
  scores.putIfAbsent(bat,0);
  scores.put(bat,scores.get(bat)+newscore);
  ```

Updating a map

- Key-value stores are useful to accumulate quantities
  - Frequencies of words in a text
  - Total runs in a tournament
- The initialization problem
  - Update the value if the key exists
  - Otherwise, create a new entry
- `Map` has the following default method

  ```
  V getOrDefault(Object key, V defaultValue)
  ```
- For instance

  ```
  Map<String, Integer> scores = ...;
  int score = scores.getOrDefault(bat,0);
  ```

  sets `score` to `0` if key `bat` is not present
- Now, we can update the entry for key `bat` as follows

  ```
  scores.put(bat,
     scores.getOrDefault(bat,0)+newscore);
     // Add newscore to value of bat
  ```
- Alternatively, use `putIfAbsent()` to initialize a missing key

  ```
  scores.putIfAbsent(bat,0);
  scores.put(bat,scores.get(bat)+newscore);
  ```
- Or use `merge()`

  ```
  scores.merge(bat,newscore,Integer::sum);
  ```
  - Initialize to `newscore` if no key `bat`
  - Otherwise, combine current value with `newscore` using `Integer::sum`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

- Can now iterate through a `Map`

```
Set<String> keys = strmap.keySet();
for (String key : keys) {
  do something with key
}
```

Extracting keys and values

- Methods to extract keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- Keys form a `Set` while values form an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views

- Can now iterate through a `Map`

```
Set<String> keys = strmap.keySet();
for (String key : keys) {
  do something with key
}
```

- Use `entrySet()` to operate on key and associated value without looking up map again

```
for (Map.Entry<String, Employee> entry :
                              staff.entrySet()){
  String k = entry.getKey();
  Employee v = entry.getValue();
  do something with k, v
}
```

Concrete implementations of Map

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

`LinkedHashMap`

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both `keySet()` and `value()` enumerate in order of insertion

Concrete implementations of Map

HashMap

- Similar to HashSet
- Use a hash table to store keys and values
- No fixed order over keys returned by keySet()

TreeMap

- Similar to TreeSet
- Use a balanced search tree to store keys and values
- Iterator over keySet() will process keys in sorted order

LinkedHashMap

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both keySet() and value() enumerate in order of insertion
- Can also use access order
  - Each get() or put() moves key-value pair to end of list
  - Process entries in least recently used order — scheduling, caching

Concrete implementations of `Map`

`HashMap`

- Similar to `HashSet`
- Use a hash table to store keys and values
- No fixed order over keys returned by `keySet()`

`TreeMap`

- Similar to `TreeSet`
- Use a balanced search tree to store keys and values
- Iterator over `keySet()` will process keys in sorted order

`LinkedHashMap`

- Remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- Iterators over both `keySet()` and `value()` enumerate in order of insertion
- Can also use access order
  - Each `get()` or `put()` moves key-value pair to end of list
  - Process entries in least recently used order — scheduling, caching
- Similarly, `LinkedHashSet`

Summary

- The `Map` interface captures properties of key-value stores
  - `get()`, `put()`, `containsKey()`, `containsValue()`, ...
- Parameterized by two type variables, `K` for keys and `V` for values
- Keys form a set
- Different ways to update a key entry, depending on whether the key already exists
  - `getOrDefault()`, `putIfAbsent()`, `merge()`
- Extract keys as a `Set`, values as a `Collection`, key-value pairs as a `Set`
  - `keySet()`, `values()`, `entrySet()`
- Use these "views" to iterate over all key-value pairs in the map
- Concrete implementations: `HashMap`, `TreeMap`, `LinkedHashMap`

Dealing with errors

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 7

When things go wrong

- Our code could encounter many types of errors
  - *User input* — enter invalid filenames or URLs
  - *Device errors* — printer jam, network connection drops
  - *Resource limitations* — disk full
  - *Code errors* — invalid array index, key not present in hash table, refer to a variable that is `null`, divide by zero, . . .

When things go wrong

- Our code could encounter many types of errors
  - *User input* — enter invalid filenames or URLs
  - *Device errors* — printer jam, network connection drops
  - *Resource limitations* — disk full
  - *Code errors* — invalid array index, key not present in hash table, refer to a variable that is `null`, divide by zero, ...
- Signalling errors
  - Return an invalid value: -1 at end of file, `null`
  - What if there is no obvious invalid value?

Exception handling

- Code that generates error raises or throws an exception

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash
- ... or passes the exception back up the calling chain

Exception handling

- Code that generates error raises or throws an exception
- Notify the type of error
  - Information about the nature of the exception
  - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
  - Extract information about the error from the exception object
  - Graceful interruption rather than program crash
- ... or passes the exception back up the calling chain
- Declare if a method can throw an exception
  - Compiler can check whether calling code has made a provision to handle the exception

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, ...

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, ...
- Checked exceptions
  - Typically user-defined, code assumptions violated
    - In a list of orders, quantities should be positive integers

Summary

- Exception handling — gracefully recover from errors that occur when running code
- Throw an exception — generate an object encapsulating information about the error
- Catch an exception — decode the nature of the error and take corrective action
- Java organizes exceptions in a hierarchy, by type
  - `Error` — internal errors within JVM, "not the programmer's fault"
  - `RunTimeException` — coding errors, could have been avoided by runtime checks in code
  - Checked exceptions — user-defined, violations of assumptions made by code
    - To contrast, `Error` and `RunTimeException` are called unchecked exceptions

Exceptions in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 7

Java's classification of errors

- All exceptions descend from class `Throwable`
  - Two branches, `Error` and `Exception`
- `Error` — relatively rare, "not the programmer's fault"
  - Internal errors, resource limitations within Java runtime
  - No realistic corrective action possible, notify caller and terminate gracefully
- `Exception` — two sub branches
  - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
  - Array index out of bounds, invalid hash key, ...
- Checked exceptions
  - Typically user-defined, code assumptions violated
    - In a list of orders, quantities should be positive integers

Catching and handling exceptions

- `try`–`catch`
  - Enclose code that may generate exception in a `try` block
  - Exception handler in `catch` block
  - Similar to Python

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- `try`–`catch`
  - Enclose code that may generate exception in a `try` block
  - Exception handler in `catch` block
  - Similar to Python
- If `try` encounters an exception, rest of the code in the block is skipped

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- try–catch
  - Enclose code that may generate exception in a try block
  - Exception handler in catch block
  - Similar to Python
- If try encounters an exception, rest of the code in the block is skipped
- If exception matches the type in catch, handler code executes

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- `try`–`catch`
  - Enclose code that may generate exception in a `try` block
  - Exception handler in `catch` block
  - Similar to Python
- If `try` encounters an exception, rest of the code in the block is skipped
- If exception matches the type in `catch`, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- `try`–`catch`
  - Enclose code that may generate exception in a `try` block
  - Exception handler in `catch` block
  - Similar to Python
- If `try` encounters an exception, rest of the code in the block is skipped
- If exception matches the type in `catch`, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code
- Top level uncaught exception — program crash

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  examine e and handle it
  ...
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hiearachy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hiearachy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`
- Catch blocks are tried in sequence
  - Match exception type against each one in turn

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Catching and handling exceptions

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hiearachy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`
- Catch blocks are tried in sequence
  - Match exception type against each one in turn
- Order `catch` blocks by argument type, more specific to less specific
  - `IOException` would intercept `FileNotFoundException`

```
try {
  code that might throw exceptions
}
catch (FileNotFoundException e) {
  handle missing files
}
catch (UnknownHostException e) {
  handle unknown hosts
}
catch (IOException e) {
  handle all other I/O issues
}
```

Generating exceptions

- When does a function generate an exception?

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, ...

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, ...
- Code calls another function that generates an exception

Generating exceptions

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds, invalid hash key, ...
- Code calls another function that generates an exception
- Your code detects an error and generates an exception
  - `throw` a checked exception

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - "Signals that EOF has been reached unexpectedly during input"

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - "Signals that EOF has been reached unexpectedly during input"
- Create an object of exception type and `throw` it

```
throw new EOFException();
```

Notifying checked exceptions

- Example: you write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - "Signals that EOF has been reached unexpectedly during input"
- Create an object of exception type and `throw` it

```
throw new EOFException();
```

- Can also pass a diagnostic message when constructing exception object

```
String errormsg = "Content-Length:" + contentlen + ", Received: " + rcvdlen;
throw new EOFException(errormsg);
```

Throwing exceptions . . .

- How does caller know that `readData()` generates `EOFException`?

Throwing exceptions . . .

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions ...

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header
- Can throw multiple types of exceptions

```
String readFile(String filename)
    throws FileNotFoundException,
           EOFException { ... }
```

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions ...

- How does caller know that `readData()` generates `EOFException`?
- Declare exceptions thrown in header
- Can throw multiple types of exceptions

```
String readFile(String filename)
    throws FileNotFoundException,
           EOFException { ... }
```

- Can throw any subtype of declared exception type

```
String readFile(String filename)
    throws IOException { ... }
```

  - Can throw `FileNotFoundException`, `EOFException`, both subclasses of `IOException`

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions ...

- Method declares the exceptions it throws

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions ...

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception
- Need not advertise unchecked exceptions
  - `Error`, `RunTimeException`

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Throwing exceptions . . .

- Method declares the exceptions it throws
- If you call such a method, you must handle it
- ... or pass it on; your method should advertise that it throws the same exception
- Need not advertise unchecked exceptions
  - `Error`, `RunTimeException`
- Should not normally generate `RunTimeException`
  - Fix the error or report suitable checked exception

```
String readData(Scanner in)
    throws EOFException {
  ...
  while (...) {
    if (!in.hasNext()) {
      // EOF encountered
      if (n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
    ...
  }
  return(s);
}
```

Customized exceptions

- Don't want negative numbers in a `LinearList`

Customized exceptions

- Don't want negative numbers in a `LinearList`
- Define a new class extending `Exception`

```
public class NegativeException extends Exception{

  private int error_value;
   // Negative value that generated exception

  public NegativeException(String message, int i){
    super(message);  // Appeal to superclass
    error_value = i; // constructor to set message
  }

  public int report_error_value(){
    return error_value;
  }
}
```

Customized exceptions

- Don't want negative numbers in a `LinearList`
- Define a new class extending `Exception`
- Throw this from `LinearList`
  - Note that `add` advertises the fact that it throws a `NegativeException`

```
public class NegativeException extends Exception{
   ...
}

public class LinearList{
  ...
  public add(int i) throws NegativeException{
    ...
    if (i < 0){
      throw new NegativeException("Negative input",i);
    }
    ...
  }
}
```

More on catching exceptions

- Can extract information about the exception

```
try {
  ...
  call a function that may
    throw an exception
  ..
}
catch (ExceptionType e){
  ...
  String errormsg = e.getMessage();
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`
- Add information when you chain exceptions

```
try {
  ...
  access database
  ..
}
catch (SQLException e){
  ...
  String errormsg =
     "database error" + e.getMessage();
  ServletException newe =
     new ServletException(errormsg);
  newe.initCause(e);
  throw newe;
  ...
}
```

More on catching exceptions

- Can extract information about the exception
- Chaining exceptions
  - Process and throw a new exception from `catch`
- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`, `initCause()`
- Add information when you chain exceptions
- Retrieve information when you catch exception

```
try {
   ...
}
catch (ServletException e){
   ...
   Throwable original = e.getCause();
   ...
}
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`

```
try{
     ...
   }

catch (ExceptionType1 e){...}

catch (ExceptionType2 e){...}

finally{
     ...
  // Always executed, whether try
  // terminates normally or
  // exceptionally. Use for clean up.
}
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`
- Different scenarios

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, ...)
- Add a block labelled `finally`
- Different scenarios
  - No error — 1,2,5,6

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, ...)
- Add a block labelled `finally`
- Different scenarios
  - No error — 1,2,5,6
  - `IOException` in `try`, no exception in `catch` — 1,3,4,5,6

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, . . .)
- Add a block labelled `finally`
- Different scenarios
  - No error — 1,2,5,6
  - `IOException` in `try`, no exception in `catch` — 1,3,4,5,6
  - `IOException` in `try`, chained exception in `catch` — 1,3,5

```
FileInputStream in =
  new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e) {
  // 3
  show error message
  // 4
}
finally {
  // 5
  in.close();
}
// 6
```

Summary

- Use `try`-`catch` to safely call functions that may generate errors
- Can `throw` an exception — usually checked exception
- Must advertise checked exceptions that are thrown in function header
  - Java compiler enforces that code that calls such a function handles the exception or passes it on
- Can inspect exceptions and chain them with information about original source
- Use `finally` to clean up resources that may be left open when code is interrupted by an exception

Packages

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 7

Packages

- Java has an organizational unit called `package`

Packages

- Java has an organizational unit called `package`
- Can use `import` to use packages directly

```
import java.math.BigDecimal
```

Packages

- Java has an organizational unit called `package`
- Can use `import` to use packages directly

```
import java.math.BigDecimal
```

- All classes in `.../java/math`

```
import java.math.*
```

Packages

- Java has an organizational unit called `package`
- Can use `import` to use packages directly

  `import java.math.BigDecimal`
- All classes in `.../java/math`

  `import java.math.*`
- Note that `*` is not recursive. Cannot write

  `import java.*`

Creating and naming packages

- Can create our own hierarchy of packages

Creating and naming packages

- Can create our own hierarchy of packages
- Naming convention is similar to Internet domain name, but in reverse
  - Internet domain: `onlinedegree.iitm.ac.in`
  - Package name: `in.ac.iitm.onlinedegree`

Creating and naming packages

- Can create our own hierarchy of packages
- Naming convention is similar to Internet domain name, but in reverse
  - Internet domain: `onlinedegree.iitm.ac.in`
  - Package name: `in.ac.iitm.onlinedegree`
- Add a package header to include a class in a package

```
package in.ac.iitm.onlinedegree;

public class Employee { ... }
```

Creating and naming packages

- Can create our own hierarchy of packages
- Naming convention is similar to Internet domain name, but in reverse
  - Internet domain: `onlinedegree.iitm.ac.in`
  - Package name: `in.ac.iitm.onlinedegree`
- Add a package header to include a class in a package

```
package in.ac.iitm.onlinedegree;

public class Employee { ... }
```

- By default, all classes in a directory belong to same anonymous package

More about visibility

- We have seen modifiers `public` and `private`

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy
  - `protected` means visible within subtree, so all subclasses

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy
  - `protected` means visible within subtree, so all subclasses
  - Normally, a subclass cannot expand visibility of a function

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility with respect to inheritance hierarchy
  - `protected` means visible within subtree, so all subclasses
  - Normally, a subclass cannot expand visibility of a function
  - However, `protected` can be made `public`

Assertions

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 7

Documenting and checking assumptions

- Functions may have constraints on the parameters

```
public static double myfn(double x){
  // Assume x >= 0
  ...
}
```

Documenting and checking assumptions

- Functions may have constraints on the parameters
- We could check the condition and throw an exception

```
public static double myfn(double x)
  throws IllegalArgumentException {
  // Assume x >= 0
  if (x < 0){
    throw new
          IllegalArgumentException("x < 0");
  }
}
```

Documenting and checking assumptions

- Functions may have constraints on the parameters
- We could check the condition and throw an exception
- What if `myfn` is only used internally by our own code
  - Flag errors during development, debugging
  - But diagnostic code should not trigger at run time
  - Performance, and other considerations

```
public static double myfn(double x)
  throws IllegalArgumentException {
  // Assume x >= 0
  if (x < 0){
    throw new
          IllegalArgumentException("x < 0");
  }
}
```

Documenting and checking assumptions

- Functions may have constraints on the parameters
- We could check the condition and throw an exception
- What if `myfn` is only used internally by our own code
  - Flag errors during development, debugging
  - But diagnostic code should not trigger at run time
  - Performance, and other considerations
- Instead, "assert" the property you assume to hold

```
public static double myfn(double x){
  assert x >= 0;
}
```

Assertions

- If assertion fails, code throws `AssertionError`

```
public static double myfn(double x){
  assert x >= 0;
}
```

Assertions

- If assertion fails, code throws `AssertionError`
- This should not be caught
  - Abort and print diagnostic information (stack trace)

```
public static double myfn(double x){
  assert x >= 0;
}
```

Assertions

- If assertion fails, code throws `AssertionError`
- This should not be caught
  - Abort and print diagnostic information (stack trace)
- Can provide additional information to be printed with diagnostic message

```
public static double myfn(double x){
  assert x >= 0 : x;
}
```

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

```
java -enableassertions MyCode
```

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

```
java -enableassertions MyCode
```

- Can use `-ea` as abbreviation for `-enableassertions`

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

  `java -ea:Myclass MyCode`

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

  `java -ea:Myclass MyCode`
- . . . or a package

  `java -ea:in.ac.iitm.onlinedegree MyCode`

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

  `java -ea:Myclass MyCode`
- . . . or a package

  `java -ea:in.ac.iitm.onlinedegree MyCode`
- Similarly, disable assertions globally or selectively

```
java -disableassertions MyCode
java -da:MyClass MyCode
```

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

  `java -ea:Myclass MyCode`
- . . . or a package

  `java -ea:in.ac.iitm.onlinedegree MyCode`
- Similarly, disable assertions globally or selectively

```
java -disableassertions MyCode
java -da:MyClass MyCode
```

- Can combine the two

```
java -ea in.ac.iitm.onlinedegree
     -da:MyClass MyCode
```

Enabling and disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require recompilation
- Use the following flag to run with assertions enabled

```
java -enableassertions MyCode
```

- Can use `-ea` as abbreviation for `-enableassertions`
- Can selectively turn on assertions for a class

```
java -ea:Myclass MyCode
```

- . . . or a package

```
java -ea:in.ac.iitm.onlinedegree MyCode
```

- Similarly, disable assertions globally or selectively

```
java -disableassertions MyCode
java -da:MyClass MyCode
```

- Can combine the two

```
java -ea in.ac.iitm.onlinedegree
     -da:MyClass MyCode
```

- Separate switch to enable assertions for system classes

```
java -enablesystemassertions MyCode
java -esa MyCode
```

Summary

- Assertion checks are supposed to flag fatal, unrecoverable errors
  - Do not `catch` them!
- If you need to flag the error and take corrective action, use exceptions instead
- Turned on only during development and testing
  - Not checked at run time after deployment

Logging

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 7

Diagnostic messages

- Typical to generate messages within code for diagnosis

Diagnostic messages

- Typical to generate messages within code for diagnosis
- Naive approach is to use print statements
  - Need to add / subtract as we go along
  - Enable and disable explicitly

Diagnostic messages

- Typical to generate messages within code for diagnosis
- Naive approach is to use print statements
  - Need to add / subtract as we go along
  - Enable and disable explicitly
- Instead log diagnostic messages separately
  - Logs are arranged hierarchically — choose the level of logging needed
  - Can be displayed in different formats
  - Logs can be processed by other code — handlers
    - Can filter out uninteresting entries
  - Logging controlled by a configuration file

Logging

- Simplest: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

Logging

- Simplest: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- This prints the following

```
January 10, 2022 10:12:15 PM LoggingImageViewer myFunction
INFO: Edit->Copy menu item selected
```

Logging

- Simplest: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- This prints the following

```
January 10, 2022 10:12:15 PM LoggingImageViewer myFunction
INFO: Edit->Copy menu item selected
```

- Suppress logging by executing the following code

```
Logger.getGlobal().setLevel(Level.OFF);
```

Logging

- Simplest: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- This prints the following

```
January 10, 2022 10:12:15 PM LoggingImageViewer myFunction
INFO: Edit->Copy menu item selected
```

- Suppress logging by executing the following code

```
Logger.getGlobal().setLevel(Level.OFF);
```

- Create a custom logger

```
private static final Logger myLogger =
   Logger.getLogger("in.ac.iitm.onlinedegree");
```

Logging

- Simplest: call `info()` method of global logger:

```
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- This prints the following

```
January 10, 2022 10:12:15 PM LoggingImageViewer myFunction
INFO: Edit->Copy menu item selected
```

- Suppress logging by executing the following code

```
Logger.getGlobal().setLevel(Level.OFF);
```

- Create a custom logger

```
private static final Logger myLogger =
   Logger.getLogger("in.ac.iitm.onlinedegree");
```

  - Logger names are hierarchical, like package names
  - Setting a property for `in.ac.iitm` automatically sets it for `in.ac.iitm.onlinedegree`

Logging levels

- Seven logging levels
  - `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`

Logging levels

- Seven logging levels
  - `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
- By default, first three levels are logged

Logging levels

- Seven logging levels
  - `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
- By default, first three levels are logged
- Can set a different level

```
logger.setLevel(Level.FINE);
```

Logging levels

- Seven logging levels
  - `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
- By default, first three levels are logged
- Can set a different level

```
logger.setLevel(Level.FINE);
```

- Turn on all levels, or turn off all logging

```
logger.setLevel(Level.ALL);
logger.setLevel(Level.OFF);
```

Logging levels

- Seven logging levels
  - `SEVERE`, `WARNING`, `INFO`, `CONFIG`, `FINE`, `FINER`, `FINEST`
- By default, first three levels are logged
- Can set a different level

```
logger.setLevel(Level.FINE);
```

- Turn on all levels, or turn off all logging

```
logger.setLevel(Level.ALL);
logger.setLevel(Level.OFF);
```

- Can also change logging properties through a configuration file
  - Look up the documentation

Summary

- Logging gives us more flexibility and control over tracking diagnostic messages than simple print statements
- Can define a hierarchy of loggers
- Seven levels of messages — control which levels are printed
- Control logging from within code or through external configuration file

Cloning

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 8

Copying an object

- Normal assignment creates two references to the same object
  - Updates via either name update the object

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1;
e2.setname("Eknath");  // e1 also updated
```

Copying an object

- Normal assignment creates two references to the same object
  - Updates via either name update the object
- What if we want two separate but identical objects?
  - e2 should be initialized to a disjoint copy of e1

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1;
e2.setname("Eknath");  // e1 also updated
```

Copying an object

- Normal assignment creates two references to the same object
  - Updates via either name update the object
- What if we want two separate but identical objects?
  - e2 should be initialized to a disjoint copy of e1
- How does one make a faithful copy?

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1;
e2.setname("Eknath");   // e1 also updated
```

The `clone()` method

- `Object` defines a method `clone()`

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}
```

The `clone()` method

- `Object` defines a method `clone()`
- `e1.clone()` returns a bitwise copy of `e1`

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

The `clone()` method

- `Object` defines a method `clone()`
- `e1.clone()` returns a bitwise copy of `e1`
- Why a bitwise copy?
  - `Object` does not have access to private instance variables
  - Cannot build up a fresh copy of `e1` from scratch

```java
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

The `clone()` method

- `Object` defines a method `clone()`
- `e1.clone()` returns a bitwise copy of `e1`
- Why a bitwise copy?
  - `Object` does not have access to private instance variables
  - Cannot build up a fresh copy of `e1` from scratch
- What could go wrong with a bitwise copy?

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String n, double s){
    name = n;
    salary = s;
  }

  public void setname(String n){
    name = n;
  }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

Shallow copy

- What if we add an instance variable `Date` to `Employee`?
  - Assume `update()` updates the components of a `Date` object

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){
    name = n;
  }

  public void setbday(int dd, int mm, int yy){
    birthday.update(dd,mm,yy);
  }
}
```

Shallow copy

- What if we add an instance variable `Date` to `Employee`?
  - Assume `update()` updates the components of a `Date` object
- Bitwise copy made by `e1.clone()` copies the reference to the embedded `Date`
  - `e2.birthday` and `e1.birthday` refer to the same object
  - `e2.setbday()` affects `e1.birthday`

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){
    name = n;
  }

  public void setbday(int dd, int mm, int yy){
    birthday.update(dd,mm,yy);
  }
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 name not updated
e2.setbday(16,4,1997); // e1 bday updated!
```

Shallow copy

- What if we add an instance variable `Date` to `Employee`?
  - Assume `update()` updates the components of a `Date` object
- Bitwise copy made by `e1.clone()` copies the reference to the embedded `Date`
  - `e2.birthday` and `e1.birthday` refer to the same object
  - `e2.setbday()` affects `e1.birthday`
- Bitwise copy is a shallow copy
  - Nested mutable references are copied verbatim

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){
    name = n;
  }

  public void setbday(int dd, int mm, int yy){
    birthday.update(dd,mm,yy);
  }
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 name not updated
e2.setbday(16,4,1997); // e1 bday updated!
```

Shallow copy

- What if we add an instance variable `Date` to `Employee`?
    - Assume `update()` updates the components of a `Date` object
- Bitwise copy made by `e1.clone()` copies the reference to the embedded `Date`
    - `e2.birthday` and `e1.birthday` refer to the same object
    - `e2.setbday()` affects `e1.birthday`
- Bitwise copy is a shallow copy
    - Nested mutable references are copied verbatim

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){
    name = n;
  }

  public void setbday(int dd, int mm, int yy){
    birthday.update(dd,mm,yy);
  }
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 name not updated
e2.setbday(16,4,1997); // e1 bday updated!
```

Deep copy

- Deep copy recursively clones nested objects

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}
}
```

Deep copy

- Deep copy recursively clones nested objects
- Override the shallow `clone()` from `Object`

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){
    Employee newemp =
          (Employee) super.clone()
    Date newbday = birthday.clone();
    newemp.birthday = newbday;
    return newmp;
  }
}
```

Deep copy

- Deep copy recursively clones nested objects
- Override the shallow `clone()` from `Object`
- `Object.clone()` returns an `Object`
  - Cast `super.clone()`

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){
    Employee newemp =
          (Employee) super.clone()
    Date newbday = birthday.clone();
    newemp.birthday = newbday;
    return newmp;
  }
}
```

Deep copy

- Deep copy recursively clones nested objects
- Override the shallow `clone()` from `Object`
- `Object.clone()` returns an `Object`
  - Cast `super.clone()`
- `Employee.clone()` returns an `Employee`
  - Allowed to change the return type

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){
    Employee newemp =
          (Employee) super.clone()
    Date newbday = birthday.clone();
    newemp.birthday = newbday;
    return newmp;
  }
}
```

Deep copy . . .

- What if `Manager` extends `Employee`?

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}
```

Deep copy . . .

- What if Manager extends Employee?
- New instance variable promodate

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}

public class Manager extends Employee {
  private Date promodate;
  ...
}
```

Deep copy . . .

- What if `Manager` extends `Employee`?
- New instance variable `promodate`
- `Manager` inherits deep copy `clone()` from `Employee`

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}

public class Manager extends Employee {
  private Date promodate;
  ...
}
```

Deep copy . . .

- What if `Manager` extends `Employee`?
- New instance variable `promodate`
- `Manager` inherits deep copy `clone()` from `Employee`
- However `Employee.clone()` does not know that it has to deep copy `promodate`!

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}

public class Manager extends Employee {
  private Date promodate;
  ...
}
```

Deep copy . . .

- What if `Manager` extends `Employee`?
- New instance variable `promodate`
- `Manager` inherits deep copy `clone()` from `Employee`
- However `Employee.clone()` does not know that it has to deep copy `promodate`!
- Cloning is subtle, so Java puts in some restrictions

```
public class Employee {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}

public class Manager extends Employee {
  private Date promodate;
  ...
}
```

Restrictions on clone()

- To allow `clone()` to be used, a class has to implement `Cloneable` interface
  - Marker interface

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

Restrictions on `clone()`

- To allow `clone()` to be used, a class has to implement `Cloneable` interface
  - Marker interface
- `clone()` in `Object` is `protected`
  - Only `Employee` objects can `clone()`

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}
}
...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e2 = e1.clone();
e2.setname("Eknath");  // e1 not updated
```

Restrictions on clone()

- To allow `clone()` to be used, a class has to implement `Cloneable` interface
  - Marker interface
- `clone()` in `Object` is `protected`
  - Only `Employee` objects can `clone()`
- Redefine `clone()` as `public` to allow other classes to clone `Employee`
  - Expanding visibility from `protected` to `public` is allowed

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone(){...}
}
```

Restrictions on `clone()`

- To allow `clone()` to be used, a class has to implement `Cloneable` interface
  - Marker interface
- `clone()` in `Object` is `protected`
  - Only `Employee` objects can `clone()`
- Redefine `clone()` as `public` to allow other classes to clone `Employee`
  - Expanding visibility from `protected` to `public` is allowed
- `Object.clone()` throws `CloneNotSupportedException`
  - Catch or report this exception
  - Call `clone()` in `try` block

```
public class Employee implements Cloneable {
  private String name;
  private double salary;
  private Date birthday;
  ...
  public void setname(String n){...}

  public void setbday(...){...}

  public Employee clone()
    throws CloneNotSupportedException {...}
}
```

Summary

- Making a faithful copy of an object is a tricky problem
- Java provides a `clone()` function in `Object` that does shallow copy

Summary

- Making a faithful copy of an object is a tricky problem
- Java provides a `clone()` function in `Object` that does shallow copy
- However, shallow copy aliases nested objects

Summary

- Making a faithful copy of an object is a tricky problem
- Java provides a `clone()` function in `Object` that does shallow copy
- However, shallow copy aliases nested objects
- Deep copy solves the problem, but inheritance can create complications

Summary

- Making a faithful copy of an object is a tricky problem
- Java provides a `clone()` function in `Object` that does shallow copy
- However, shallow copy aliases nested objects
- Deep copy solves the problem, but inheritance can create complications
- To force programmers to consciously think about these subtleties, Java puts in some checks to using `clone()`

Summary

- Making a faithful copy of an object is a tricky problem
- Java provides a `clone()` function in `Object` that does shallow copy
- However, shallow copy aliases nested objects
- Deep copy solves the problem, but inheritance can create complications
- To force programmers to consciously think about these subtleties, Java puts in some checks to using `clone()`
- Must implement marker interface `Cloneable` to allow `clone()`
- `clone()` is `protected` by default. override as `public` if needed
- `clone()` in `Object` throws `CloneNotSupportedException`, which must be taken into account when overriding

Type inference

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 8

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference
- Derive type information from context. For instance, `s` should be `String`

```
s = "Hello, " + "world";
```

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;  // Allowed by subtyping
```

Type declarations vs type inference

- Java insists that all variables are declared in advance, with type information
- The compiler can then check whether the program is well-typed
- An alternative approach is to do type inference
- Derive type information from context. For instance, `s` should be `String`

  ```
  s = "Hello, " + "world";
  ```

- Propagate type information: now `t` is also `String`

  ```
  t = s + 5;
  ```

```
public class Employee {...}

public class Manager extends Employee {...}

Employee e;

Manager m;

...

m = new Manager(...);
e = m;   // Allowed by subtyping
```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```
  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```

  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```

- More ambitious?

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    `s = "Hello, " + "world";`

  - Propagate type information based on already inferred types

    `t = s + 5;`

- More ambitious?
  - If `x.bonus()` is legal, `x` must be `Manager` rather than `Employee`

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume code is well-typed, derive most general types
  - Use information from constants to determine type

    ```
    s = "Hello, " + "world";
    ```
  - Propagate type information based on already inferred types

    ```
    t = s + 5;
    ```
- More ambitious?
  - If `x.bonus()` is legal, `x` must be `Manager` rather than `Employee`
- Keep track of and validate type obligations

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}


...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types
- Typing judgements should ideally be made at compile-time, not at run-time
  - Static analysis of code

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference

- Assume program is type-safe, derive most general types compatible with code
  - Use information from constants to determine type
  - Propagate type information based on already inferred types
- Typing judgements should ideally be made at compile-time, not at run-time
  - Static analysis of code
- Balance flexibility with algorithmic tractability

```
public class Employee {...}

public class Manager extends Employee {
   ...
   public double bonus (...) {...}
}

...

public static f(Employee x){
  ...
  double d = x.bonus(...);
    // x must be a Manager?
  ...
}
```

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value

```
var b = false;   // boolean

var s = "Hello, world";   // String
```

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value
- Be careful about format for numeric constants

```
var b = false;  // boolean

var s = "Hello, world";  // String

var d = 2.0;  // double

var f = 3.141f; // float
```

Type inference in Java

- Java allows limited type inference
  - Only for local variables in functions
  - Not for instance variables of a class
- Use generic `var` to declare variables
  - Must be initialized when declared
  - Type is inferred from initial value
- Be careful about format for numeric constants
- For classes, infer most constrained type
  - `e` is inferred to be `Manager`
  - `Manager` extends `Employee`
  - If `e` should be `Employee`, declare explicitly

```
var b = false;  // boolean

var s = "Hello, world";  // String

var d = 2.0;  // double

var f = 3.141f; // float

var e = new Manager(...);  // Manager
```

Summary

- Automatic type inference can avoid redundancy in declarations

```
Manager m = new Manager(...);
```

- Assuming the program is type-safe, derive most general types compatible with the code
  - Compiler can infer type from expressions used to assign values
  - Inferred type information can be propagated
- Challenge is to do this statically, at compile-time
- Java allows limited type inference
  - Only local variables that are initialized when they are declared

Higher order functions

Madhavan Mukund

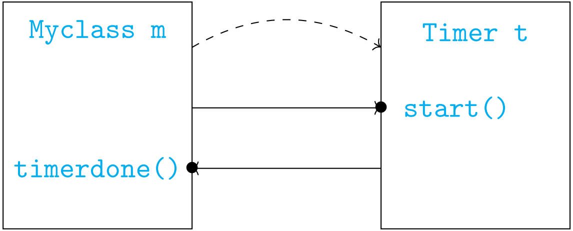
https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 8

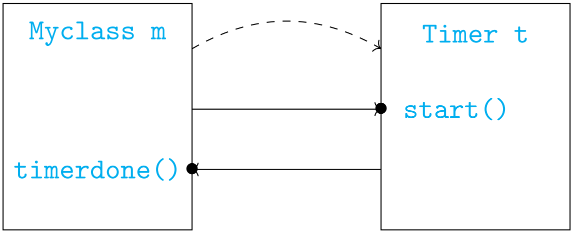
Passing functions

- Recall callbacks
  - `Myclass m` creates a `Timer t`
  - `t` starts running in parallel
  - `t` notifies `m` when the time limit expires



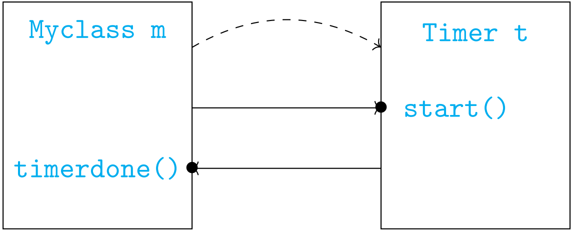
Passing functions

- Recall callbacks
  - `Myclass m` creates a `Timer t`
  - `t` starts running in parallel
  - `t` notifies `m` when the time limit expires
- `m` needs to pass `timerdone()` to `t`



Passing functions

- Recall callbacks
  - `Myclass m` creates a `Timer t`
  - `t` starts running in parallel
  - `t` notifies `m` when the time limit expires
- `m` needs to pass `timerdone()` to `t`
- Achieved this through an interface




```
public interface Timerowner{
  public abstract void timerdone();
}

public class Myclass
       extends Timerowner{
  ...
}
```

```
public class Timer implements Runnable{
  private Timerowner owner;
  ...
  public void start(){
    ...
    owner.timerdone();
  }
}
```

Passing functions

- Customize `Arrays.sort`

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function
- Implement `Comparator`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}


public class StringCompare
  implements Comparator<String>{

  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
}
```

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function
- Implement `Comparator`
- Pass to `Arrays.sort`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public class StringCompare
  implements Comparator<String>{

  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
}
```

```
String[] strarr = new ...;
Arrays.sort(strarr,StringCompare);
```

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public interface Timerowner{
  public abstract void timerdone();
}
```

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- How can we directly pass the required function?

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public interface Timerowner{
  public abstract void timerdone();
}
```

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- How can we directly pass the required function?
- In Python, function names are similar to variable names
  - Define a function
  - Pass it as an argument to another function
  - `map` is a higher order function

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

```
public interface Timerowner{
  public abstract void timerdone();
}
```

```
def square(x):
  return(x*x)

l = list(map(square,range(100)))
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, ...

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, ...
- Substitute wherever a functional interface is specified

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (String s1, String s2) ->
               s1.length() - s2.length());
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, ...
- Substitute wherever a functional interface is specified
- Limited type inference is also possible
  - Java infers `s1` and `s2` are `String`

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (String s1, String s2) ->
               s1.length() - s2.length());
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (s1, s2) ->
               s1.length() - s2.length());
```

Lambda expressions

- More complicated function body can be defined as a block

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Lambda expressions

- More complicated function body can be defined as a block
- Note that the function is anonymous only for the caller

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Lambda expressions

- More complicated function body can be defined as a block
- Note that the function is anonymous only for the caller
- The function that receives the lambda expression still needs to use a functional interface for the parameter type

```
public static <T> void
   Arrays.sort(T[] a, Comparator<T> c)}
```

  - Inside `Arrays.sort()`, refer to the function by the name `compare()` defined in the `Comparator` interface

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a `Map` object
  - Here `sum` is a static method in `Integer`

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a `Map` object
  - Here `sum` is a static method in `Integer`

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

- Here is the corresponding expression, assuming type inference

```
(i,j) -> Integer::sum(i,j)
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a `Map` object
  - Here `sum` is a static method in `Integer`

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

- Here is the corresponding expression, assuming type inference

```
(i,j) -> Integer::sum(i,j)
```

- Expression should call a function, and nothing else — this expression cannot be replaced by a method reference

```
(i,j) -> Integer::sum(i,j) > 0
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

- `object::InstanceMethod`
  - Method reference is `o::f`
  - Arguments are passed to `o.f`

```
(x1,x2,..,xk) -> o.f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

`(x1,x2,..,xk) -> f(x1,x2,...,xk)`

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

`(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)`

- `object::InstanceMethod`
  - Method reference is `o::f`
  - Arguments are passed to `o.f`

`(x1,x2,..,xk) -> o.f(x1,x2,...,xk)`

- Can also pass references to constructors

Summary

- Many languages support higher-order functions
  - Passing a function as an argument to another function
- In object-oriented programming, this is achieved using interfaces
  - Encapsulate the function to be passed as an object
- Java allows functions to be passed directly in place of functional interfaces
  - Interface consists of a single function
- Lambda expressions describe anonymous functions
  - Cannot pass lambda expressions in general
  - Only when the argument is a functional interface
- Can pass a method reference if the lambda expression consists of a single function call

Streams

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 8

Operating on collections

- We usually use an iterator to process a collection
  - Suppose we have split a text file as a list of words
  - We want to count the number of long words in the list

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

Operating on collections

- We usually use an iterator to process a collection
  - Suppose we have split a text file as a list of words
  - We want to count the number of long words in the list
- An iterator generates all elements from a collection as a sequence

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

Operating on collections

- We usually use an iterator to process a collection
  - Suppose we have split a text file as a list of words
  - We want to count the number of long words in the list
- An iterator generates all elements from a collection as a sequence
- Alternative approach
  - Generate a stream of values from a collection
  - Operations transform input streams to output streams
  - Terminate with a result

```
List<String> words = ....;
long count = 0;
for (String w : words) {
  if (w.length() > 10) {
    count++;
  }
}
```

```
long count = words.stream()
            .filter(w -> w.length() > 10)
            .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how
- Processing can be parallelized
  - `filter()` and `count()` in parallel

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

```
long count = words.parallelStream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Why streams?

- Stream processing is declarative
  - Recall, declarative vs imperative
  - Focus on what to compute, rather than how
- Processing can be parallelized
  - `filter()` and `count()` in parallel
- Lazy evaluation is possible
  - Suppose we want first 10 long words
  - Stop generating the stream once we find 10 such words
  - Need not generate the entire stream in advance
  - Can even work, in principle, with infinite streams!

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

```
long count = words.parallelStream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Working with streams

- Create a stream

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}

long count = words.parallelStream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
}

long count = words.parallelStream()
                  .filter(w -> w.length() > 10)
                  .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result

```
long count = words.stream()
             .filter(w -> w.length() > 10)
             .count();
}
```

```
long count = words.parallelStream()
             .filter(w -> w.length() > 10)
             .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result
- A stream does not store its elements
  - Elements stored in an underlying collection
  - Or generated by a function, on demand

```
long count = words.stream()
               .filter(w -> w.length() > 10)
               .count();
}
```

```
long count = words.parallelStream()
               .filter(w -> w.length() > 10)
               .count();
}
```

Working with streams

- Create a stream
- Pass through intermediate operations that transform streams
- Apply a terminal operation to get a result
- A stream does not store its elements
  - Elements stored in an underlying collection
  - Or generated by a function, on demand
- Stream operations are non-destructive
  - Input stream is untouched

```
long count = words.stream()
             .filter(w -> w.length() > 10)
             .count();
}
```

```
long count = words.parallelStream()
             .filter(w -> w.length() > 10)
             .count();
}
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface
- Use static method `Stream.of()` for arrays

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
  - Provide a function that produces values on demand, with no argument

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
  - Provide a function that produces values on demand, with no argument
- `Stream.iterate()` — a stream of dependent values
  - Initial value, function to generate the next value from the previous one

```java
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);

Stream<Integer> integers =
  Stream.iterate(0, n -> n+1)
```

Creating streams

- Apply `stream()` to a collection
  - Part of `Collections` interface
- Use static method `Stream.of()` for arrays
- Static method `Stream.generate()` generates a stream from a function
  - Provide a function that produces values on demand, with no argument
- `Stream.iterate()` — a stream of dependent values
  - Initial value, function to generate the next value from the previous one
  - Terminate using a predicate

```
List<String> wordlist = ...;
Stream<String> wordstream = wordlist.stream();

String[] wordarr = ...;
Stream<String> wordstream = Stream.of(wordarr);

Stream<String> echos =
  Stream.generate(() -> "Echo");

Stream<Double> randomds =
  Stream.generate(Math::random);

Stream<Integer> integers =
  Stream.iterate(0, n -> n+1)

Stream<Integer> integers =
  Stream.iterate(0, n -> n < 100, n -> n+1)
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words
- `map()` applies a function to each element in the stream.
  - Extract the first letter of each long word

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words
- `map()` applies a function to each element in the stream.
  - Extract the first letter of each long word
- What if `map()` function generates a list?
  - Suppose we have `explode(s)` that returns the list of letters in `s`
  - `map()` produces stream with nested lists

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> explode(s));
```

Processing streams

- `filter()` to select elements
  - Takes a predicate as argument
  - Filter out the long words
- `map()` applies a function to each element in the stream.
  - Extract the first letter of each long word
- What if `map()` function generates a list?
  - Suppose we have `explode(s)` that returns the list of letters in `s`
  - `map()` produces stream with nested lists
- `flatMap()` flattens (collapses) nested list into a single stream

```
List<String> wordlist = ...;
Stream<String> longwords =
   wordlist.stream()
   .filter(w -> w.length() > 10);
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .map(s -> s.substring(0,1));
```

```
List<String> wordlist = ...;
Stream<String> startlongwords =
   wordlist.stream()
   .filter(w -> w.length() > 10)
   .flatMap(s -> explode(s));
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);

Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5
- Start after element matches a criterion — `dropWhile()`
  - Start after number larger than 0.05

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);
```

```
Stream<Double> randomds =
  Stream.generate(Math::random)
        .dropWhile(n -> n <= 0.05);
```

Stream transformations

- Make a stream finite — `limit(n)`
  - Generate 100 random numbers
- Skip `n` elements — `skip(n)`
  - Discard first 10 random numbers
- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5
- Start after element matches a criterion — `dropWhile()`
  - Start after number larger than 0.05
- Can also combine streams, extract distinct elements, sort, ...

```
Stream<Double> randomds =
  Stream.generate(Math::random).limit(100);

Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);

Stream<Double> randomds =
  Stream.generate(Math::random)
        .takeWhile(n -> n >= 0.5);

Stream<Double> randomds =
  Stream.generate(Math::random)
        .dropWhile(n -> n <= 0.05);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(10)
        .max(Double::compareTo);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
    Return value is optional type — later

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Reducing a stream to a result

- Number of elements — `count()`
  - Count random numbers larger than 0.1
- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty? Return value is optional type — later
- First element — `findFirst()`
  - First random number above 0.999
  - Again, deal with empty stream
- And more ...

```
long countrand =
  Stream.generate(Math::random)
        .limit(100).
        .filter(n -> n > 0.1)
        .count();

Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> firstrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n > 0.999)
        .findFirst();
```

Streams

- We can view a collection as a stream of elements
- Process the stream rather than use an iterator
- Declarative way of computing over collections — popular in functional programming
- Create a stream, transform it, reduce it to a result
- Can create a stream from any collection, or generate from a function
- Stream transformations are non-destructive: filter, map, limit to a finite number, skip elements, ...
- Various functions to reduce to a result — deal with empty streams

Optional Types

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 9

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
- `max()` of empty stream is undefined
  - Return value could be `Double` or `null`

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
- `max()` of empty stream is undefined
  - Return value could be `Double` or `null`
- `Optional<T>` object
  - Wrapper
  - May contain an object of type `T`
    - Value is present
  - Or no object

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Handling missing optional values

- Use `orElse()` to pass a default value

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand = maxrand.orElse(-1.0);
```

Handling missing optional values

- Use `orElse()` to pass a default value
- Use `orElseGet()` to call a function to generate replacement for a missing value

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand = maxrand.orElseGet(
     () -> SomeFunctionToGenerateDouble
   );
```

Handling missing optional values

- Use `orElse()` to pass a default value
- Use `orElseGet()` to call a function to generate replacement for a missing value
- Use `orElseThrow()` to generate an exception when a missing value is encountered

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand =
   maxrand.orElseThrow(
     IllegalStateException::new
   );
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored

```
optionalValue.ifPresent(v -> Process v);
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresent(v -> results.add(v));
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present
  - As usual, pass the function in different forms

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresent(results::add);
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present
  - As usual, pass the function in different forms
- Specify an alternative action if the value is not present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresentOrElse(
   v -> results.add(v),
   () -> System.out.println("No max")
);
```

Creating an optional value

- Creating an optional value
  - `Optional.of(v)` creates value `v`
  - `Optional.empty` creates empty optional

```
public static Optional<Double>
   inverse(Double x){

   if (x == 0) {
     return Optional.empty();
   }else{
     return Optional.of(1 / x);
   }
}
```

Creating an optional value

- Creating an optional value
  - `Optional.of(v)` creates value `v`
  - `Optional.empty` creates empty optional
- Use `ofNullable()` to transform `null` automatically into an empty optional
  - Useful when working with functions that return object of type `T` or `null`, rather than `Optional<T>`

```
public static Optional<Double>
   inverse(Double x) {

   return Optional.ofNullable(1 / x);
}
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> maxrandsqr =
  maxrand.map(v -> v*v);
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output
- Another example

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.map(results::add);
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output
- Another example
- Supply an alternative for a missing value
  - If value is present, it is passed as is
  - If value is empty, value generated by `or()` is passed

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> fixrand =
   maxrand.or(() -> Optional.of(-1.0));
```

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`
- Instead, use `flatMap`
  - `s.f().flatMap(T::g)`
  - If `s.f()` is present, apply `g()`
  - Otherwise return empty `Optional<U>`

```
Optional<U> result = s.f().flatMap(T::g);
```

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`
- Instead, use `flatMap`
  - `s.f().flatMap(T::g)`
  - If `s.f()` is present, apply `g()`
  - Otherwise return empty `Optional<U>`
- For example, pass output of earlier safe `inverse()` to safe `squareRoot()`

```
public static Optional<Double>
    inverse(Double x) {
    if (x == 0) {
      return Optional.empty();
    }else{
      return Optional.of(1 / x);
    }
}
public static Optional<Double>
    squareRoot(Double x){
    if (x < 0) {
      return Optional.empty();
    }else{
      return Optional.of(Math.sqrt(x));
    }
}

Optional<Double> result =
  inverse(x).flatMap(MyClass::squareRoot);
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username

```
Optional<User> lookup(String id) {...}
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`

```
Optional<User> lookup(String id) {...}
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`
- Pass through a `flatMap`

```
Stream<String> ids = ...;
Stream<User> users = ids.map(Users::lookup)
    .flatMap(Optional::stream);
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`
- Pass through a `flatMap`
- What if `lookup` was implemented without using `Optional`?
  - `oldLookup` returns `User` or `null`
  - Use `ofNullable` to regenerate `Optional<User>`

```
Stream<String> ids = ...;
Stream<User> users = ids.flatMap(
    id -> Stream.ofNullable(
              Users.oldLookup(id)
          )
    );
```

Summary

- `Optional<T>` is a clean way to encapsulate a value that may be absent
- Different ways to process values of type `Optional<T>`
  - Replace the missing value by a default
  - Ignore missing values
- Can create values of type `Optional<T>` where outcome may be undefined
- Can write functions that transform optional values to optional values
- `flatMap` allows us to cascade functions with optional types
  - Use `flatMap` to regenerate a stream from optional values

Collecting results from streams

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 9

Collecting values from a stream

- Convert collections into sequences of values — streams

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?
- `Stream` defines a standard iterator, use to loop through values in a stream

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?
- `Stream` defines a standard iterator, use to loop through values in a stream
- Alternatively, use `forEach` with a suitable function

```
mystream.forEach(System.out::println);
```

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?
- `Stream` defines a standard iterator, use to loop through values in a stream
- Alternatively, use `forEach` with a suitable function
- Can convert a stream into an array using `toArray()`
  - Creates an array of `Object` by default

```
mystream.forEach(System.out::println);
```

```
Object[] result = mystream.toArray();
```

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?
- `Stream` defines a standard iterator, use to loop through values in a stream
- Alternatively, use `forEach` with a suitable function
- Can convert a stream into an array using `toArray()`
  - Creates an array of `Object` by default
- Pass array constructor to get a more specific array type

```java
mystream.forEach(System.out::println);
```

```java
Object[] result = mystream.toArray();
```

```java
String[] result =
   mystream.toArray(String[]::new);
   // mystream.toArray() has type Object[]
```

Storing a stream as a collection

- What if we want to convert the stream back into a collection?

Storing a stream as a collection

- What if we want to convert the stream back into a collection?
- Use `collect()`
  - Pass appropriate factory method from `Collectors`
  - Static method that directly calls a constructor

Storing a stream as a collection

- What if we want to convert the stream back into a collection?
- Use `collect()`
  - Pass appropriate factory method from `Collectors`
  - Static method that directly calls a constructor
- Create a list from a stream

```
List<String> result =
   mystream.collect(Collectors.toList());
```

Storing a stream as a collection

- What if we want to convert the stream back into a collection?
- Use `collect()`
  - Pass appropriate factory method from `Collectors`
  - Static method that directly calls a constructor
- Create a list from a stream
- . . . or a set

```
List<String> result =
   mystream.collect(Collectors.toList());
```

```
Set<String> result =
   mystream.collect(Collectors.toSet());
```

Storing a stream as a collection

- What if we want to convert the stream back into a collection?
- Use `collect()`
  - Pass appropriate factory method from `Collectors`
  - Static method that directly calls a constructor
- Create a list from a stream
- . . . or a set
- To create a concrete collection, provide a constructor

```
List<String> result =
   mystream.collect(Collectors.toList());
```

```
Set<String> result =
   mystream.collect(Collectors.toSet());
```

```
TreeSet<String> result =
   stream.collect(
     Collectors.toCollection(
       TreeSet::new
     )
   );
```

Stream summaries

- We saw how to reduce a stream to a single result value — `count()`, `max()`, ...
  - In general, need a stream of numbers

Stream summaries

- We saw how to reduce a stream to a single result value — `count()`, `max()`, ...
  - In general, need a stream of numbers
- `Collectors` has methods to aggregate summaries in a single object
  - `summarizingInt` works for a stream of integers
  - Pass function to convert given stream to numbers — here `String::length`
  - Returns `IntSummaryStatistics` that stores count, max, min, sum, average

```
IntSummaryStatistics summary =
  mystream.collect(
      Collectors.summarizingInt(
        String::length)
      );
```

Stream summaries

- We saw how to reduce a stream to a single result value — `count()`, `max()`, ...
  - In general, need a stream of numbers
- `Collectors` has methods to aggregate summaries in a single object
  - `summarizingInt` works for a stream of integers
  - Pass function to convert given stream to numbers — here `String::length`
  - Returns `IntSummaryStatistics` that stores count, max, min, sum, average

```
IntSummaryStatistics summary =
  mystream.collect(
     Collectors.summarizingInt(
       String::length)
     );

double averageWordLength = summary.getAverage()
double maxWordLength = summary.getMax();
```

- Methods to access relevant statistics
  - `getCount()`,`getMax()`, `getMin()`, `getSum()`, `getAverage()`,

Stream summaries

- We saw how to reduce a stream to a single result value — `count()`, `max()`, ...
  - In general, need a stream of numbers
- `Collectors` has methods to aggregate summaries in a single object
  - `summarizingInt` works for a stream of integers
  - Pass function to convert given stream to numbers — here `String::length`
  - Returns `IntSummaryStatistics` that stores count, max, min, sum, average

```
IntSummaryStatistics summary =
  mystream.collect(
      Collectors.summarizingInt(
        String::length)
      );

double averageWordLength = summary.getAverage()
double maxWordLength = summary.getMax();
```

- Methods to access relevant statistics
  - `getCount()`,`getMax()`, `getMin()`, `getSum()`, `getAverage()`,
- Similarly, `summarizingLong()` and `summarizingDouble()` return `LongSummaryStatistics` and `DoubleSummaryStatistics`

Converting a stream to a map

- Convert a stream of `Person` to a map
  - For `Person p`, `p.getID()` is key and `p.getName()` is value

```
Stream<Person> people = ...;

Map<Integer, String> idToName =
   people.collect(
     Collectors.toMap(
       Person::getId,
       Person::getName
     )
   );
```

Converting a stream to a map

- Convert a stream of `Person` to a map
  - For `Person p`, `p.getID()` is key and `p.getName()` is value
- To store entire object as value, use `Function.identity()`

```
Stream<Person> people = ...;

Map<Integer, Person> idToPerson =
   people.collect(
     Collectors.toMap(
       Person::getId,
       Function.identity()
     )
   );
```

Converting a stream to a map

- Convert a stream of `Person` to a map
  - For `Person p`, `p.getID()` is key and `p.getName()` is value
- To store entire object as value, use `Function.identity()`
- What happens if we use name for key and id for value?

```
Stream<Person> people = ...;

Map<String, Integer> nameToID =
   people.collect(
     Collectors.toMap(
       Person::getName,
       Person::getId
     )
   );
```

Converting a stream to a map

- Convert a stream of `Person` to a map
  - For `Person p`, `p.getID()` is key and `p.getName()` is value
- To store entire object as value, use `Function.identity()`
- What happens if we use name for key and id for value?
  - Likely to have duplicate keys — `IllegalStateException`

```
Stream<Person> people = ...;

Map<String, Integer> nameToID =
   people.collect(
     Collectors.toMap(
       Person::getName,
       Person::getId
     )
   );
```

Converting a stream to a map

- Convert a stream of `Person` to a map
  - For `Person p`, `p.getID()` is key and `p.getName()` is value
- To store entire object as value, use `Function.identity()`
- What happens if we use name for key and id for value?
  - Likely to have duplicate keys — `IllegalStateException`
- Provide a function to fix such problems

```
Stream<Person> people = ...;

Map<String, Integer> nameToID =
   people.collect(
     Collectors.toMap(
       Person::getName,
       Person::getId,
       (existingValue, newValue) ->
          existingValue
     )
   );
```

Grouping and partitioning values

- Instead of discarding values with duplicate keys, group them

Grouping and partitioning values

- Instead of discarding values with duplicate keys, group them
- Collect all ids with the same name in a list

```
Stream<Person> people = ...;

Map<String, List<Person>> nameTopersons =
   people.collect(
     Collectors.groupingBy(
       Person::getName
     )
   );
```

Grouping and partitioning values

- Instead of discarding values with duplicate keys, group them
- Collect all ids with the same name in a list
- Instead, may want to partition the stream using a predicate

```
Stream<Person> people = ...;

Map<String, List<Person>> nameTopersons =
   people.collect(
     Collectors.groupingBy(
       Person::getName
     )
   );
```

Grouping and partitioning values

- Instead of discarding values with duplicate keys, group them
- Collect all ids with the same name in a list
- Instead, may want to partition the stream using a predicate
- Partition names into those that start with `A` and the rest
    - Key values of resulting map are `true` and `false`

```
Stream<Person> people = ...;

Map<Boolean, List<Person>> aAndOtherPersons =
   people.collect(
     Collectors.partitioningBy(
       p -> p.getName().substr(0,1).equals("A")
     )
   );

List<Person> startingLetterA =
  aAndOtherPersons.get(true);
```

Summary

- We converted collections into sequences and processed them as streams
- After transformations, we may want to process a stream as a collection
- Use iterators, `forEach()` to process a stream element by element
- Use `toArray()` to convert to an array
- Factory methods in `Collector` allow us to convert a stream back into a collection of our choice
- Can convert an arbitrary stream into a stream of numbers and collect summary statistics
- Can convert a stream into a map
- Can group values by a key, or partition by a predicate

Input/output streams

Madhavan Mukund

`https://www.cmi.ac.in/~madhavan`

Programming Concepts using Java

Week 9

Input and output streams

- Input: read a sequence of bytes from some source
  - A file, an internet connection, memory . . .
- Output: write a sequence of bytes to some source
  - A file, an internet connection, memory . . .

Input and output streams

- Input: read a sequence of bytes from some source
  - A file, an internet connection, memory ...
- Output: write a sequence of bytes to some source
  - A file, an internet connection, memory ...
- Java refers to these as input and output streams
  - Not the same as stream objects in class `Stream`

Input and output streams

- Input: read a sequence of bytes from some source
  - A file, an internet connection, memory ...
- Output: write a sequence of bytes to some source
  - A file, an internet connection, memory ...
- Java refers to these as input and output streams
  - Not the same as stream objects in class `Stream`
- Input and output values could be of different types
  - Ultimately, input and output are raw uninterpreted bytes of data
  - Interpret as text — different Unicode encodings
  - Or as binary data — integers, floats, doubles, ...

Input and output streams

- Input: read a sequence of bytes from some source
  - A file, an internet connection, memory ...
- Output: write a sequence of bytes to some source
  - A file, an internet connection, memory ...
- Java refers to these as input and output streams
  - Not the same as stream objects in class `Stream`
- Input and output values could be of different types
  - Ultimately, input and output are raw uninterpreted bytes of data
  - Interpret as text — different Unicode encodings
  - Or as binary data — integers, floats, doubles, ...
- Use a pipeline of input/output stream transformers
  - Read raw bytes from a file, pass to a stream that reads text
  - Generate binary data, pass to a stream that writes raw bytes to a file

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`

```
abstract int read();
int read(byte[] b);
byte[] readAllBytes();
// ... and more
```

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Check availability before reading

```
abstract int read();
int read(byte[] b);
byte[] readAllBytes();
// ... and more
```

```
InputStream in = ....
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
   var data = new byte[bytesAvailable];
   in.read(data);
}
```

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Check availability before reading
- Write bytes to output

```
abstract void write(int b);
void write(byte[] b);
// ... and more

OutputStream out = ...
byte[] values = ...;
out.write(values);
```

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Check availability before reading
- Write bytes to output
- Close a stream when done — release resources

```
abstract void write(int b);
void write(byte[] b);
// ... and more

OutputStream out = ...
byte[] values = ...;
out.write(values);


in.close();
```

Reading and writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Check availability before reading
- Write bytes to output
- Close a stream when done — release resources
- Flush an output stream — output is buffered

```
abstract void write(int b);
void write(byte[] b);
// ... and more
```

```
OutputStream out = ...
byte[] values = ...;
out.write(values);
```

```
in.close();
```

```
out.flush();
```

Connecting a stream to an external source

- Input and output streams ultimately connect to external resources
  - A file, an internet connection, memory ...
  - We limit ourselves to files

Connecting a stream to an external source

- Input and output streams ultimately connect to external resources
  - A file, an internet connection, memory …
  - We limit ourselves to files
- Create an input stream attached to a file

```
var in = new FileInputStream("input.class");
```

Connecting a stream to an external source

- Input and output streams ultimately connect to external resources
  - A file, an internet connection, memory ...
  - We limit ourselves to files
- Create an input stream attached to a file
- Create an output stream attached to a file

```
var in = new FileInputStream("input.class");

var out = new FileOutputStream("output.bin");
```

Connecting a stream to an external source

- Input and output streams ultimately connect to external resources
  - A file, an internet connection, memory ...
  - We limit ourselves to files
- Create an input stream attached to a file
- Create an output stream attached to a file
- Overwrite or append?
  - Pass a boolean second argument to the constructor

```
var in = new FileInputStream("input.class");

var out = new FileOutputStream("output.bin");

var out = new
   FileOutputStream("newoutput.bin",false);
   // Overwrite

var out = new
   FileOutputStream("sameoutput.bin",true);
   // Append
```

Reading and writing text

- Recall `Scanner` class
    - Can apply to any input stream

```
var fin = new FileInputStream("input.txt");
var scin = new Scanner(fin);

var scin = new Scanner(
      new FileInputStream("input.txt")
    );
```

Reading and writing text

- Recall `Scanner` class
  - Can apply to any input stream
- Many read methods

```
var fin = new FileInputStream("input.txt");
var scin = new Scanner(fin);

var scin = new Scanner(
      new FileInputStream("input.txt")
    );


String s = scin.nextLine(); // One line
String w = scin.next(); // One word
int i = scin.nextInt(); // Read an int
boolean b = scin.hasNext(); // Any more words?
```

Reading and writing text

- Recall `Scanner` class
  - Can apply to any input stream
- Many read methods
- To write text, use `PrintWriter` class
  - Apply to any output stream

```
var fout = new FileOutputStream("output.txt");
var pout = new PrintWriter(fout);

pout var = new PrintWriter(
      new FileOutputStream("output.txt");
    );
```

Reading and writing text

- Recall `Scanner` class
  - Can apply to any input stream
- Many read methods
- To write text, use `PrintWriter` class
  - Apply to any output stream
- Use `println()`, `print()` to write txt

```
var fout = new FileOutputStream("output.txt");
var pout = new PrintWriter(fout);

pout var = new PrintWriter(
      new FileOutputStream("output.txt");
    );
```

```
String msg = "Hello, world";
pout.println(msg);
```

Reading and writing text

- Recall `Scanner` class
  - Can apply to any input stream
- Many read methods
- To write text, use `PrintWriter` class
  - Apply to any output stream
- Use `println()`, `print()` to write txt
- Example: Copy input text file to output text file

```
var in = new Scanner(...);
var out = new PrintWriter(...);

while (in.hasNext()){
    String line = in.nextLine();
    out.println(line);
}
```

Reading and writing text

- Recall `Scanner` class
  - Can apply to any input stream
- Many read methods
- To write text, use `PrintWriter` class
  - Apply to any output stream
- Use `println()`, `print()` to write txt
- Example: Copy input text file to output text file
- Beware: input/output methods generate many different kinds of exceptions
  - Need to wrap code with `try` blocks

```
var in = new Scanner(...);
var out = new PrintWriter(...);

while (in.hasNext()){
    String line = in.nextLine();
    out.println(line);
}
```

Reading and writing binary data

- To read binary data, use `DataInputStream` class
  - Can apply to any input stream

```
var fin = new FileInputStream("input.class");
var din = new DataInputStream(fin);

var din = new DataInputStream(
      new FileInputStream("input.class")
    );
```

Reading and writing binary data

- To read binary data, use `DataInputStream` class
  - Can apply to any input stream
- Many read methods

```
var fin = new FileInputStream("input.class");
var din = new DataInputStream(fin);

var din = new DataInputStream(
      new FileInputStream("input.class")
    );
```

```
readInt, readShort, readLong
readFloat, readDouble,
readChar, readUTF
readBoolean
```

Reading and writing binary data

- To read binary data, use `DataInputStream` class
  - Can apply to any input stream
- Many read methods
- To write binary data, use `DataOutputStream` class
  - Apply to any output stream

```
var fout = new FileOutputStream("output.bin");
var dout = new DataOutputStream(fout);

var dout = new DataOutputStream(
      new FileOutputStream("output.bin")
    );
```

Reading and writing binary data

- To read binary data, use `DataInputStream` class
  - Can apply to any input stream
- Many read methods
- To write binary data, use `DataOutputStream` class
  - Apply to any output stream
- Many write methods

```
var fout = new FileOutputStream("output.bin");
var dout = new DataOutputStream(fout);

var dout = new DataOutputStream(
      new FileOutputStream("output.bin")
    );
```

```
writeInt, writeShort, writeLong
writeFloat, writeDouble
writeChar, writeUTF
writeBoolean
writeChars
writeByte
```

Reading and writing binary data

- To read binary data, use `DataInputStream` class
  - Can apply to any input stream
- Many read methods
- To write binary data, use `DataOutputStream` class
  - Apply to any output stream
- Many write methods
- Example: Copy input binary file to output binary file
  - Again, be careful to catch exceptions

```
var in = new DataInputStream(...);
var out = new DataOutputStream(...);

int bytesAvailable = in.available();
while (bytesAvailable > 0){
    var data = new byte[bytesAvailable];
    in.read(data);
    out.write(data);
    bytesAvailable = in.available();
}
```

Other features

- Buffering an input stream
  - Reads blocks of data
  - More efficient

```
var din = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("grades.dat")
   )
);
```

Other features

- Buffering an input stream
  - Reads blocks of data
  - More efficient
- Speculative reads
  - Examine the first element
  - Return to stream if necessary

```
var din = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("grades.dat")
   )
);

var pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("grades.dat")));

int b = pbin.read();

if (b != '<') pbin.unread(b);
```

Other features

- Buffering an input stream
  - Reads blocks of data
  - More efficient
- Speculative reads
  - Examine the first element
  - Return to stream if necessary
- Streams are specialized
  - `PushBackStream` can only `read()` and `unread()`
  - Feed to a `DataInputStream` to read meaningful data

```
var pbin = new PushbackInputStream(
      new BufferedInputStream(
         new FileInputStream("grades.dat")));

var din = new DataInputStream(pbin);
```

Other features

- Buffering an input stream
  - Reads blocks of data
  - More efficient
- Speculative reads
  - Examine the first element
  - Return to stream if necessary
- Streams are specialized
  - `PushBackStream` can only `read()` and `unread()`
  - Feed to a `DataInputStream` to read meaningful data

```
var pbin = new PushbackInputStream(
      new BufferedInputStream(
         new FileInputStream("grades.dat")));

var din = new DataInputStream(pbin);
```

- Java has a whole zoo of streams for different tasks
  - Random access files, zipped data, ...

Other features

- Buffering an input stream
  - Reads blocks of data
  - More efficient
- Speculative reads
  - Examine the first element
  - Return to stream if necessary
- Streams are specialized
  - `PushBackStream` can only `read()` and `unread()`
  - Feed to a `DataInputStream` to read meaningful data

```
var pbin = new PushbackInputStream(
      new BufferedInputStream(
         new FileInputStream("grades.dat")));

var din = new DataInputStream(pbin);
```

- Java has a whole zoo of streams for different tasks
  - Random access files, zipped data, ...
- Chain together streams in a pipeline
  - Read binary data from a zipped file

    `FileInputStream` $\rightarrow$
    `ZipInputStream` $\rightarrow$
    `DataInputStream`

Summary

- Java's approach to input/output is to separate out concerns
- Chain together different types of input/output streams
  - Connect an external source as input or output
  - Read and write raw bytes
  - Interpret raw bytes as text
  - Interpret raw bytes as data
  - Buffering, speculative read, random access files, zipped data, ...
- Chaining together streams appears tedious, but adds flexibility

Serialization

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 9

Reading and writing objects

- We can read and write binary data
  - `DataInputStream`, `DataOutputStream`

Reading and writing objects

- We can read and write binary data
  - `DataInputStream`, `DataOutputStream`
- Read and write low level units
  - Bytes, integers, floats, characters, . . .

Reading and writing objects

- We can read and write binary data
  - `DataInputStream`, `DataOutputStream`
- Read and write low level units
  - Bytes, integers, floats, characters, ...
- Can we export and import objects directly?

Reading and writing objects

- We can read and write binary data
  - `DataInputStream`, `DataOutputStream`
- Read and write low level units
  - Bytes, integers, floats, characters, . . .
- Can we export and import objects directly?
- Why would we want to do this?
  - Backup objects onto disk, with state
  - Restore objects from disk
  - Send objects across a network

Reading and writing objects

- We can read and write binary data
  - `DataInputStream`, `DataOutputStream`
- Read and write low level units
  - Bytes, integers, floats, characters, ...
- Can we export and import objects directly?
- Why would we want to do this?
  - Backup objects onto disk, with state
  - Restore objects from disk
  - Send objects across a network
- Serialization and deserialization

Reading and writing objects . . .

- To write objects, Java has another output stream type, `ObjectOutputStream`

```
var out = new ObjectOutputStream(
    new FileOutputStream("employee.dat"));
```

Reading and writing objects ...

- To write objects, Java has another output stream type, `ObjectOutputStream`
- Use `writeObject()` to write out an object

```
var out = new ObjectOutputStream(
    new FileOutputStream("employee.dat"));

var emp = new Employee(...);
var boss = new Manager(...);
out.writeObject(emp);
out.writeObject(boss);
```

Reading and writing objects . . .

- To write objects, Java has another output stream type, `ObjectOutputStream`
- Use `writeObject()` to write out an object
- To read back objects, use `ObjectInputStream`

```
var out = new ObjectOutputStream(
    new FileOutputStream("employee.dat"));

var emp = new Employee(...);
var boss = new Manager(...);
out.writeObject(emp);
out.writeObject(boss);

var in = new ObjectInputStream(
    new FileInputStream("employee.dat"));
```

Reading and writing objects . . .

- To write objects, Java has another output stream type, `ObjectOutputStream`
- Use `writeObject()` to write out an object
- To read back objects, use `ObjectInputStream`
- Retrieve objects in the same order they were written, using `readObject()`

```
var out = new ObjectOutputStream(
    new FileOutputStream("employee.dat"));

var emp = new Employee(...);
var boss = new Manager(...);
out.writeObject(emp);
out.writeObject(boss);

var in = new ObjectInputStream(
    new FileInputStream("employee.dat"));

var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

Reading and writing objects . . .

- To write objects, Java has another output stream type, `ObjectOutputStream`
- Use `writeObject()` to write out an object
- To read back objects, use `ObjectInputStream`
- Retrieve objects in the same order they were written, using `readObject()`
- Class has to allow serialization — implement marker interface `Serializable`

```
var out = new ObjectOutputStream(
    new FileOutputStream("employee.dat"));
```

```
var emp = new Employee(...);
var boss = new Manager(...);
out.writeObject(emp);
out.writeObject(boss);
```

```
var in = new ObjectInputStream(
    new FileInputStream("employee.dat"));
```

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

```
public class Employee
       implements Serializable {...}
```

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents
- `ObjectInputStream` "reconstructs" the object, effectively calls a constructor

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents
- `ObjectInputStream` "reconstructs" the object, effectively calls a constructor
- What happens when many objects share the same object as an instance variable?

```
class Manager extends Employee {
   private Employee secretary;
   ....
}
```

  - Two managers have the same secretary
  - How do we avoid duplicating objects when serializing?

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents
- `ObjectInputStream` "reconstructs" the object, effectively calls a constructor
- What happens when many objects share the same object as an instance variable?

```
class Manager extends Employee {
   private Employee secretary;
   ....
}
```

  - Two managers have the same secretary
  - How do we avoid duplicating objects when serializing?
- Each object is assigned a serial number
  - When first encountered, save the data to output stream
  - If saved previously, record serial number
  - Reverse the process when reading

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents
- `ObjectInputStream` "reconstructs" the object, effectively calls a constructor
- What happens when many objects share the same object as an instance variable?

```
class Manager extends Employee {
   private Employee secretary;
   ....
}
```

  - Two managers have the same secretary
  - How do we avoid duplicating objects when serializing?
- Each object is assigned a serial number — hence serialization
  - When first encountered, save the data to output stream
  - If saved previously, record serial number
  - Reverse the process when reading

Customizing serialization

- Some objects should not be serialized
  — values of file handles, ...

Customizing serialization

- Some objects should not be serialized — values of file handles, ...
- Mark such fields as `transient`

```
public class LabeledPoint
      implements Serializable{
   private String label;
   private transient Point2D.Double point;
   ...
}
```

Customizing serialization

- Some objects should not be serialized — values of file handles, ...
- Mark such fields as `transient`
- Can override `writeObject()`
  - `defaultWriteObject()` writes out the object with all non-transient fields
  - Then explicitly write relevant details of transient fields

```
private void
      writeObject(ObjectOutputStream out)
      throws IOException{
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}
```

Customizing serialization

- Some objects should not be serialized — values of file handles, . . .
- Mark such fields as `transient`
- Can override `writeObject()`
  - `defaultWriteObject()` writes out the object with all non-transient fields
  - Then explicitly write relevant details of transient fields
- . . . and `readObject()`
  - `defaultReadObject()` reconstructs object with all non-transient fields
  - Then explicitly reconstruct transient fields

```
private void
      writeObject(ObjectOutputStream out)
      throws IOException{
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}

private void
      readObject(ObjectInputStream in)
      throws IOException {
   in.defaultReadObject();
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

Handle with care!

- Serialization is a good option to share data within an application

Handle with care!

- Serialization is a good option to share data within an application
- Over time, older serialized objects may be incompatible with newer versions
    - Some mechanisms for version control, but still some pitfalls possible

Handle with care!

- Serialization is a good option to share data within an application
- Over time, older serialized objects may be incompatible with newer versions
  - Some mechanisms for version control, but still some pitfalls possible
- Deserialization implicitly invokes a constructor
  - Running code from an external source
  - Always a security risk

Summary

- Serialization allows us to export and import objects, with state
  - Backup objects onto disk, with state
  - Restore objects from disk
  - Send objects across a network
- Use `ObjectOutputStream` and `ObjectInputStream` to write and read objects
- Serial numbers are used to ensure only a single copy of each shared object is archived
- Mark fields that should not be serialized as `transient`
  - Customize `writeObject()` and `readObject()`
- Serialization carries risks
  - Version control of objects
  - Running unknown code

Concurrency: Threads and Processes

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via "shared memory"
  - Context switches are easier

Concurrent programming

- Multiprocessing
  - Single processor executes several computations "in parallel"
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via "shared memory"
  - Context switches are easier
- Henceforth, we use process and thread interchangeably

Shared variables

- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true

Shared variables

- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for race conditions
  - Shared variables must be updated consistently

Creating threads in Java

- Have a class extend `Thread`

```
public class Parallel extends Thread{
  private int id;

  public Parallel(int i){ id = i; }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel

```
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);        // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread

```
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);          // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    for (int i = 0; i < 5; i++){
      p[i] = new Parallel(i);
      p[i].start();  // Start p[i].run()
    }                // in concurrent thread
  }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!

```java
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);          // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();  // Start p[i].run()
    }                 // in concurrent thread
  }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException` — later

```java
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);        // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();  // Start p[i].run()
    }                 // in concurrent thread
  }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException` — later

Typical output

```
My id is 0
My id is 3
My id is 2
My id is 1
My id is 4
My id is 0
My id is 2
My id is 3
My id is 4
My id is 1
My id is 0
My id is 3
My id is 1
My id is 2
My id is 4
My id is 0
...
```

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`

```
public class Parallel implements Runnable{
  // only the line above has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}
```

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`
- To use `Runnable` class, explicitly create a `Thread` and `start()` it

```
public class Parallel implements Runnable{
  // only the line above has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       t[i] = new Thread(p[i]);
             // Make a thread t[i] from p[i]
       t[i].start();  // Start off p[i].run()
                      // Note: t[i].start(),
    }                 //   not p[i].start()
  }
}
```

Summary

- Common to have logically parallel actions with a single application
  - Download from one webpage while browsing another
- Threads are lightweight processes with shared variables that can run in parallel
- Use `Thread` class or `Runnable` interface to create parallel threads in Java

Race conditions

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Threads and shared variables

- Threads are lightweight processes with shared variables that can run in parallel
- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially false
  - Clicking `Stop` sets it to true
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for race conditions
  - Shared variables must be updated consistently

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`: `transfer()` and `audit()`

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Maintaining data consistency

- `double accounts[100]` describes 100 bank accounts
- Two functions that operate on `accounts`: `transfer()` and `audit()`
- What are the possibilities when we execute the following?

```
Thread 1                     Thread 2
...                          ...
status =                     System.out.
  transfer(500.00,7,8);       print(audit());
...                          ...
```

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Maintaining data consistency . . .

- What are the possibilities when we execute the following?

```
Thread 1                    Thread 2
...                         ...
status =                    System.out.
  transfer(500.00,7,8);      print(audit());
...                         ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Maintaining data consistency . . .

- What are the possibilities when we execute the following?

```
Thread 1                      Thread 2
...                           ...
status =                      System.out.
  transfer(500.00,7,8);        print(audit());
...                           ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Maintaining data consistency . . .

- What are the possibilities when we execute the following?

```
Thread 1                  Thread 2
...                       ...
status =                  System.out.
  transfer(500.00,7,8);    print(audit());
...                       ...
```

- `audit()` can report an overall total that is 500 more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
  - Can even report an error if `transfer` happens atomically

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Atomicity of updates

- Two threads increment a shared variable `n`

Thread 1	Thread 2
`...`	`...`
`m = n;`	`k = n;`
`m++;`	`k++;`
`n = m;`	`n = k;`
`...`	`...`

Atomicity of updates

- Two threads increment a shared variable `n`

Thread 1	Thread 2
`...`	`...`
`m = n;`	`k = n;`
`m++;`	`k++;`
`n = m;`	`n = k;`
`...`	`...`

- Expect `n` to increase by 2 ...

Atomicity of updates

- Two threads increment a shared variable `n`

```
Thread 1          Thread 2
...               ...
m = n;            k = n;
m++;              k++;
n = m;            n = k;
...               ...
```

- Expect `n` to increase by 2 . . .
- . . . but, time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`
- Mutually exclusive access to critical regions of code

```
boolean transfer (double amount,
                  int source,
                  int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

double audit(){
  // total balance across all accounts
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Summary

- Concurrent update of a shared variable can lead to data inconsistencny
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

Mutual Exclusion

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Mutual exclusion

- Concurrent update of a shared variable can lead to data inconsistenccy
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

Mutual exclusion for two processes

- First attempt

Thread 1
```
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

Thread 2
```
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update

Mutual exclusion for two processes

- First attempt

Thread 1

```
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

Thread 2

```
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed ...

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is guaranteed . . .
- . . . but one thread is locked out permanently if other thread shuts down
  - Starvation!

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

Mutual exclusion for two processes . . .

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed . . .

Mutual exclusion for two processes ...

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is guaranteed ...
- ... but if both threads try simultaneously, they block each other
  - Deadlock!

Peterson's algorithm

Thread 1

```
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

Thread 2

```
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches

Peterson's algorithm

Thread 1

```
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

Thread 2

```
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches
- If both try simultaneously, `turn` decides who goes through

Peterson's algorithm

Thread 1

```
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

Thread 2

```
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches
- If both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For n process mutual exclusion other solutions exist
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clever solutions for different situations
- Need to argue correctness in each case
- Instead, provide higher level support in programming language for synchronization

Summary

- We can construct protocols that guarantee mutual exclusion to critical sections
  - Watch out for starvation and deadlock
- These protocols cleverly use regular variables
  - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization

Test and Set

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step

Test and set

- The fundamental issue preventing consistent concurrent updates of shared varuables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive

Semaphores

- Programming language support for mutual exclusion

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch passeren, to pass
  - `V(s)` — from Dutch vrygeven, to release

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch passeren, to pass
  - `V(s)` — from Dutch vrygeven, to release
- `P(S)` atomically executes the following

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch passeren, to pass
  - `V(s)` — from Dutch vrygeven, to release

- `P(S)` atomically executes the following

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

- `V(S)` atomically executes the following

```
if (there are threads waiting
    for S to become positive)
  wake one of them up;
    //choice is nondeterministic
else
  increment S;
```

Using semaphores

- Mutual exclusion using semaphores

Thread 1

```
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

Thread 2

```
...
P(S);
// Enter critical section
   ...
// Leave critical section
V(S);
...
```

Using semaphores

- Mutual exclusion using semaphores

```
Thread 1                        Thread 2
...                             ...
P(S);                           P(S);
// Enter critical section       // Enter critical section
   ...                             ...
// Leave critical section       // Leave critical section
V(S);                           V(S);
...                             ...
```

- Semaphores guarantee
  - Mutual exclusion
  - Freedom from starvation
  - Freedom from deadlock

Problems with semaphores

- Too low level

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each `P(S)` has a matching `V(S)`

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region that it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each `P(S)` has a matching `V(S)`
- Can even execute `V(S)` without having done `P(S)`

Summary

- Test-and-set is at the heart of most race conditions
- Need a high level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors

Monitors

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 10

Atomic test-and-set

- Test-and-set is at the heart of most race conditions
- Need a high level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors

Monitors

- Attach synchronization control to the data that is being protected

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors

- Attach synchronization control to the data that is being protected
- Monitors — Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive
- If `Thread 1` is executing `transfer` and `Thread 2` invokes `audit`, it must wait

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive
- If `Thread 1` is executing `transfer` and `Thread 2` invokes `audit`, it must wait
- Implicit queue associated with each monitor
  - Contains all processes waiting for access
  - In practice, this may be just a set, not a queue

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`
- If these calls are reordered and `accounts[j] < 400` initially, this will fail

Making monitors more flexible

- Our definition of monitors may be too restrictive

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

- This should always succeed if `accounts[i] > 500`
- If these calls are reordered and `accounts[j] < 400` initially, this will fail
- A possible fix — let an account wait for pending inflows

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait

Monitors — wait()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

- All other processes are blocked out while this process waits!
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait
- Dual operation to notify and wake up suspended processes

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

Monitors — `notify()`

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?

Monitors — `notify()`

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction
- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor

Monitors — notify()

```
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction
- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor
- Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
  - At wake-up, the state was fine, but it has changed again due to some other concurrent action

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g., not enough inflow into the account to allow transfer
- A thread can be again interleaved between notification and running
  - At wake-up, the state was fine, but it has changed again due to some other concurrent action
- `wait()` should be in a `while`, not in an `if`

```
boolean transfer (double amount, int source, int target){
  while (accounts[source] < amount){  wait();  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue

Condition variables

- After `transfer`, `notify()` is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue
- Monitor can have condition variables to describe internal queues

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Summary

- Concurrent programming with atomic test-and-set primitives is error prone
- Monitors are like abstract datatypes for concurrent programming
  - Encapsulate data and methods to manipulate data
  - Methods are implicitly atomic, regulate concurrent access
  - Each object has an implicit external queue of processes waiting to execute a method
- `wait()` and `notify()` allow more flexible operation
- Can have multiple internal queues controlled by condition variables

Monitors in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

Monitors

- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish
- Implicit queue associated with each monitor
  - Contains all processes waiting for access

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads
- Notify change — `q[target].notify()`

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads
- Notify change — `q[target].notify()`
- Signal and exit — notifying process immediately exits the monitor
- Signal and wait — notifying process swaps roles with notified process
- Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Monitors in Java

- Monitors incorporated within existing class definitions

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically
- Each object has a lock
  - To execute a `synchronized` method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically
- Each object has a lock
  - To execute a synchronized method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time
- Wait for lock in external queue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- `wait()` and `notify()` to suspend and resume

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- `wait()` and `notify()` to suspend and resume
- Wait — single internal queue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- `wait()` and `notify()` to suspend and resume
- Wait — single internal queue
- Notify
  - `notify()` signals one (arbitrary) waiting process
  - `notifyAll()` signals all waiting processes
  - Java uses signal and continue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`
- Each object has its own internal queue

```
Object o = new Object();

public int f(){
  ..
  synchronized(o){
      ...
      o.wait();   // Wait in queue attached to "o"
      ...
  }
}

public double g(){
  ..
  synchronized(o){
      ...
      o.notifyAll();   // Wake up queue attached to
      ...
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized

```
public double h(){
  synchronized(this){
    ...
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized
- "Anonymous" `wait()`, `notify()`, `notifyAll()` abbreviate `this.wait()`, `this.notify()`, `this.notifyAll()`

```
public double h(){
  synchronized(this){
    ...
  }
}
```

Object locks . . .

- Actually, `wait()` can be “interrupted” by an `InterruptedException`

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`
- Likewise, use `o.wait()`, `o.notify()`, `o.notifyAll()` only in block synchronized on `o`

Reentrant locks

- Separate `ReentrantLock` class

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`
- Always `unlock()` in `finally` — avoid abort while holding lock

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`
- Always `unlock()` in `finally` — avoid abort while holding lock
- Why reentrant?
  - Thread holding lock can reacquire it
  - `transfer()` may call `getBalance()` that also locks `bankLock`
  - Hold count increases with `lock()`, decreases with `unlock()`
  - Lock is available if hold count is 0

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Summary

- Every object in Java implicitly has a lock
- Methods tagged `synchronized` are executed atomically
  - Implicitly acquire and release the object's lock
- Associated condition variable, single internal queue
  - `wait()`, `notify()`, `notifyAll()`
- Can synchronize an arbitrary block of code using an object
  - `sycnchronized(o) { ... }`
  - `o.wait()`, `o.notify()`, `o.notifyAll()`
- Reentrant locks work like semaphores

Threads in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException` — later

```
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);        // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();  // Start p[i].run()
    }                 // in concurrent thread
  }
}
```

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException` — later

Typical output

```
My id is 0
My id is 3
My id is 2
My id is 1
My id is 4
My id is 0
My id is 2
My id is 3
My id is 4
My id is 1
My id is 0
My id is 3
My id is 1
My id is 2
My id is 4
My id is 0
...
```

Java threads ...

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`
- To use `Runnable` class, explicitly create a `Thread` and `start()` it

```
public class Parallel implements Runnable{
  // only the line above has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       t[i] = new Thread(p[i]);
             // Make a thread t[i] from p[i]
       t[i].start();  // Start off p[i].run()
                      // Note: t[i].start(),
    }                 //   not p[i].start()
  }
}
```

Life cycle of a Java thread

A thread can be in six states

Life cycle of a Java thread

A thread can be in six states

- New: Created but not `start()`ed.

Life cycle of a Java thread

A thread can be in six states

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing

Life cycle of a Java thread

A thread can be in six states

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing
- Not available to run
  - Blocked — waiting for a lock, unblocked when lock is granted
  - Waiting — suspended by `wait()`, unblocked by `notify()` or `notfifyAll()`
  - Timed wait — within `sleep(..)`, released when sleep timer expires

Life cycle of a Java thread

A thread can be in six states

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing
- Not available to run
  - Blocked — waiting for a lock, unblocked when lock is granted
  - Waiting — suspended by `wait()`, unblocked by `notify()` or `notfifyAll()`
  - Timed wait — within `sleep(..)`, released when sleep timer expires
- Dead: thread terminates.

Life cycle of a Java thread

A thread can be in six states — thread status via `t.getState()`

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing
- Not available to run
  - Blocked — waiting for a lock, unblocked when lock is granted
  - Waiting — suspended by `wait()`, unblocked by `notify()` or `notfifyAll()`
  - Timed wait — within `sleep(..)`, released when sleep timer expires
- Dead: thread terminates.

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`
- No exception raised if thread is running!
  - `interrupt()` sets a status flag
  - `interrupted()` checks interrupt status and clears the flag
- Detecting an interrupt while running or waiting

```
public void run(){
  try{
    j = 0;
    while(!interrupted() && j < 100){
      System.out.println("My id is "+id);
      sleep(1000);    // Sleep for 1000 ms
      j++;
    }
  }
  catch(InterruptedException e){}
}
```

More about threads . . .

- Check a thread's interrupt status
  - Use `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag

More about threads . . .

- Check a thread's interrupt status
  - Use `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- Can give up running status
  - `yield()` gives up active state to another thread
  - Static method in `Thread`

More about threads . . .

- Check a thread's interrupt status
  - Use `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- Can give up running status
  - `yield()` gives up active state to another thread
  - Static method in `Thread`
  - Normally, scheduling of threads is handled by OS — preemptive
  - Some mobile platforms use cooperative scheduling — thread loses control only if it yields

More about threads ...

- Check a thread's interrupt status
  - Use `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- Can give up running status
  - `yield()` gives up active state to another thread
  - Static method in `Thread`
  - Normally, scheduling of threads is handled by OS — preemptive
  - Some mobile platforms use cooperative scheduling — thread loses control only if it yields
- Waiting for other threads
  - `t.join()` waits for `t` to terminate

Summary

- To run in parallel, need to extend `Thread` or implement `Runnable`
  - When implmenting `Runnable`, first create a `Thread` from `Runnable` object
- `t.start()` invokes method `run()` in parallel
- Threads can become inactive for different reasons
  - Block waiting for a lock
  - Wait in internal queue for a condition to be notified
  - Wait for a sleep timer to elapse
- Threads can be interrupted
  - Be careful to check both `interrupted` status and handle `InterruptException`
- Can yield control, or wait for another thread to terminate

Concurrent Programming: An Example

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  - Cars on the bridge going in the same direction $\Rightarrow$ can cross
  - No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  - Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  - Cars on the bridge going in the same direction $\Rightarrow$ can cross
  - No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  - Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  - Cars on the bridge going in the same direction $\Rightarrow$ can cross
  - No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  - Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.
- When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more.

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)
- `Bridge` has a public method `public void cross(int id, boolean d, int s)`

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)
- `Bridge` has a public method `public void cross(int id, boolean d, int s)`
  - `id` is identity of car

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)
- `Bridge` has a public method `public void cross(int id, boolean d, int s)`
  - `id` is identity of car
  - `d` indicates direction
    - `true` is North
    - `false` is South

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)
- `Bridge` has a public method `public void cross(int id, boolean d, int s)`
  - `id` is identity of car
  - `d` indicates direction
    - `true` is North
    - `false` is South
  - `s` indicates time taken to cross (milliseconds)

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change
    `Car 10 going South stuck at Fri Feb 25 12:42:13 IST 2022`

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change
    `Car 10 going South stuck at Fri Feb 25 12:42:13 IST 2022`
  - The direction changes
    `Car 10 switches bridge direction to South at Fri Feb 25 12:42:13 IST 2022`

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change
    `Car 10 going South stuck at Fri Feb 25 12:42:13 IST 2022`
  - The direction changes
    `Car 10 switches bridge direction to South at Fri Feb 25 12:42:13 IST 2022`
  - A car enters the bridge
    `Car 10 going South enters bridge at Fri Feb 25 12:42:13 IST 2022`

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change
    `Car 10 going South stuck at Fri Feb 25 12:42:13 IST 2022`
  - The direction changes
    `Car 10 switches bridge direction to South at Fri Feb 25 12:42:13 IST 2022`
  - A car enters the bridge
    `Car 10 going South enters bridge at Fri Feb 25 12:42:13 IST 2022`
  - A car leaves the bridge
    `Car 10 leaves at Fri Feb 25 12:42:14 IST 2022`

Analysis

- The "data" that is shared is the `Bridge`

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — `int bcount`
  - Current direction of bridge — `boolean direction`

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — `int bcount`
  - Current direction of bridge — `boolean direction`
- The method `public void cross(int id, boolean d, int s)` changes the state of the bridge
  - Concurrent execution of `cross` can cause problems . . .

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — `int bcount`
  - Current direction of bridge — `boolean direction`
- The method `public void cross(int id, boolean d, int s)` changes the state of the bridge
  - Concurrent execution of `cross` can cause problems ...
- ...but making `cross` a synchronized method is too restrictive
  - Only one car on the bridge at a time
  - Problem description explicitly disallows such a solution

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
- Make `enter` and `leave` synchronized

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
- Make `enter` and `leave` synchronized
- `travel` is just a means to let time elapse — use `sleep`

Analysis . . .

Code for `cross`

```
public void cross(int id, boolean d, int s){

    // Get onto the bridge (if you can!)
    enter(id,d);

    // Takes time to cross the bridge
    try{
        Thread.sleep(s);
    }
    catch(InterruptedException e){}

    // Get off the bridge
    leave(id);
}
```

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter
- Otherwise, `wait()` for the state of the bridge to change

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter
- Otherwise, `wait()` for the state of the bridge to change
- In each case, print a diagnostic message

Code for enter

```
private synchronized void enter(int id, boolean d){
    Date date;

    // While there are cars going in the wrong direction
    while (d != direction && bcount > 0){

        date = new Date();
        System.out.println("Car "+id+" going "+direction_name(d)+" stuck at "+date);

        // Wait for our turn
        try{
            wait();
        }
        catch (InterruptedException e){}
    }

    ...
```

Code for enter

```
private synchronized void enter(int id, boolean d){
    ...
    while (d != direction && bcount > 0){ ... wait() ...}
    ...

    if (d != direction){  // Switch direction, if needed
        direction = d;
        date = new Date();
        System.out.println("Car "+id+" switches bridge direction
           to "+direction_name(direction)+" at "+date);
    }

    bcount++;  // Register our presence on the bridge

    date = new Date();
    System.out.println("Car "+id+" going "+direction_name(d)+" enters bridge at "+date);
}
```

Code for `leave`

Leaving the bridge is much simpler

Code for `leave`

Leaving the bridge is much simpler

- Decrement the car count

Code for `leave`

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars

Code for `leave`

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars . . . provided car count is zero

Code for `leave`

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars . . . provided car count is zero

```
private synchronized void leave(int id){
    Date date = new Date();
    System.out.println("Car "+id+" leaves at "+date);

    // "Check out"
    bcount--;

    // If everyone on the bridge has checked out, notify the
    // cars waiting on the opposite side
    if (bcount == 0){
        notifyAll();
    }
}
```

Summary

- Concurrent programming can be tricky
- Need to synchronize access to shared resources
- . . . while allowing concurrency
- This bridge crossing example is a prototype for a number of real world requirements

Thread safe collections

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance
- Can we implement collections to allow such concurrent updates in a safe manner — make them thread safe?

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment accounts[i], neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]`, neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability
- Contrast with serializability in databases, where transactions (sequences of updates) appear atomic

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates
- Sequences of updates (transfer from one account to another) still need to be manually synchronized to work properly

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.
- Bank account example
  - Transfer threads insert transfer instructions into shared queue
  - Update thread processes instructions from the queue, modifies bank accounts
  - Only the update thread modifies the data structure
  - No synchronization necessary

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.
- Bank account example
  - Transfer threads insert transfer instructions into shared queue
  - Update thread processes instructions from the queue, modifies bank accounts
  - Only the update thread modifies the data structure
  - No synchronization necessary
- How does a consumer thread know when to check the queue?

Blocking queues

- Blocking queues block when ...
  - ...you try to add an element when the queue is full
  - ...you try to remove an element when the queue is empty

Blocking queues

- Blocking queues block when ...
  - ... you try to add an element when the queue is full
  - ... you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available

Blocking queues

- Blocking queues block when ...
  - ... you try to add an element when the queue is full
  - ... you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
  - Producers write intermediate results into the queue
  - Consumers retrieve these results and make further updates

Blocking queues

- Blocking queues block when . . .
  - . . . you try to add an element when the queue is full
  - . . . you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
  - Producers write intermediate results into the queue
  - Consumers retrieve these results and make further updates
- Blocking automatically balances the workload
  - Producers wait if consumers are slow and the queue fills up
  - Consumers wait if producers are slow to provide items to process

Summary

- When updating collections, locking the entire data structure for individual updates is wasteful
- Sufficient to protect access within a local portion of the structure
  - Ensure that two updates do not overlap
  - Region to protect depends on the type of collection
  - Implement using lower level locks of suitable granularity
- Java provides built-in thread safe collections
- One of these is a blocking queue
  - Use a blocking queue to coordinate producers and consumers
  - Ensure safe access to a shared data structure without explicit synchronization

Graphical interfaces and event-driven programming

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 12

GUIs and events

- How do we design graphical user interfaces?

GUIs and events

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen

GUIs and events

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen
- Keystrokes, mouse clicks have to be sent to appropriate window

GUIs and events

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen
- Keystrokes, mouse clicks have to be sent to appropriate window
- In parallel to main activity, record and respond to these `events`
  - Web browser renders current page
  - Clicking on a link loads a different page

Keeping track of events

- Remember coordinates and extent of each window

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
    - Check which windows are positioned at (x, y)
    - Check if one of them is "active"
    - Inform that window about mouse click

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone
- Programming language support for higher level events

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone
- Programming language support for higher level events
  - Run time support for language maps low level events to high level events

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone
- Programming language support for higher level events
  - Run time support for language maps low level events to high level events
  - OS reports low level events: mouse clicked at (x, y), key `'a'` pressed

Keeping track of events

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone
- Programming language support for higher level events
  - Run time support for language maps low level events to high level events
  - OS reports low level events: mouse clicked at (x, y), key `'a'` pressed
  - Program sees high level events: Button was clicked, box was ticked ...

Better PL support for events

- Programmer directly defines components such as windows, buttons, ... that "generate" high level events

Better PL support for events

- Programmer directly defines components such as windows, buttons, ... that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application

Better PL support for events

- Programmer directly defines components such as windows, buttons, ... that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application
- Programming language has mechanisms for
  - Describing what types of events a component can generate
  - Setting up an association between components and listeners

Better PL support for events

- Programmer directly defines components such as windows, buttons, ... that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application
- Programming language has mechanisms for
  - Describing what types of events a component can generate
  - Setting up an association between components and listeners
- Different events invoke different functions
  - Window frame has `Maximize`, `Iconify`, `Close` buttons
- Language "sorts" out events and automatically calls the correct function in the listener

An example

- A `Button` with one event, press button

An example

- A `Button` with one event, press button
- Pressing the button invokes the function `buttonpush(..)` in a listener

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when
              // a button is pushed
  }
}
```

An example

- A `Button` with one event, press button
- Pressing the button invokes the function `buttonpush(..)` in a listener
- We have set up an association between `Button b` and a listener `ButtonListener m`

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when
              // a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify
                     // m when pushed
```

An example

- A `Button` with one event, press button
- Pressing the button invokes the function `buttonpush(..)` in a listener
- We have set up an association between `Button b` and a listener `ButtonListener m`
- Nothing more needs to be done!

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...        // what to do when
                // a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify
                     // m when pushed
```

An example ...

- Communicating each button push to the listener is done automatically by the run-time system

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when
              // a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify
                     // m when pushed
```

An example . . .

- Communicating each button push to the listener is done automatically by the run-time system
- Information about the button push event is passed as an object to the listener

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when
              // a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify
                     // m when pushed
```

An example . . .

- Communicating each button push to the listener is done automatically by the run-time system
- Information about the button push event is passed as an object to the listener
- `buttonpush(...)` has arguments
  - Listener can decipher source of event, for instance

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when
              // a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);    // Tell b to notify
                      // m when pushed
```

Timer

- Recall `Timer` example

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `Timerowner` when it is done, via a function `timerdone()`

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `Timerowner` when it is done, via a function `timerdone()`
- Abstractly, timer duration elapsing is an event, and `Timerowner` is notified when the event occurs
  - In the timer, the notification is done explicitly, manually
  - In the button example, the notification is handled internally, automatically

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `Timerowner` when it is done, via a function `timerdone()`
- Abstractly, timer duration elapsing is an event, and `Timerowner` is notified when the event occurs
  - In the timer, the notification is done explicitly, manually
  - In the button example, the notification is handled internally, automatically
- In our example, `Myclass m` was itself the `Timerowner` to be notified

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `Timerowner` when it is done, via a function `timerdone()`
- Abstractly, timer duration elapsing is an event, and `Timerowner` is notified when the event occurs
  - In the timer, the notification is done explicitly, manually
  - In the button example, the notification is handled internally, automatically
- In our example, `Myclass m` was itself the `Timerowner` to be notified
- In principle, `Timer t` could be passed a reference to any object that implements `Timerowner` interface

Summary

- Event driven programming is a natural way of dealing with graphical user interface interactions
- User interacts with object through mouse clicks etc
- These are automatically translated into events and passed to listeners
- Listeners implement methods that react appropriately to different types of events

The Swing toolkit

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 12

Event driven programming in Java

- `Swing` toolkit to define high-level components

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listeners
    - `Exit browser` reported to all windows currently open

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listeners
    - `Exit browser` reported to all windows currently open
- Must explicitly set up association between component and listener

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listeners
    - `Exit browser` reported to all windows currently open
- Must explicitly set up association between component and listener
- Events are “lost” if nobody is listening!

A button that paints its background red

- `JButton` is Swing class for buttons

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push
  - Invokes `actionPerformed(...)` in listener

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push
  - Invokes `actionPerformed(...)` in listener
- Button push is an `ActionEvent`

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push
  - Invokes `actionPerformed(...)` in listener
- Button push is an `ActionEvent`

```
public class MyButtons{
  private JButton b;
  public MyButtons(ActionListener a){
     b = new JButton("MyButton");
       // Set the label on the button
     b.addActionListener(a);
       // Associate an listener
  }
}
```

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push
  - Invokes `actionPerformed(...)` in listener
- Button push is an `ActionEvent`

```
public class MyButtons{
  private JButton b;
  public MyButtons(ActionListener a){
     b = new JButton("MyButton");
       // Set the label on the button
     b.addActionListener(a);
       // Associate an listener
  }
}
public class MyListener implements ActionListener
  public void actionPerformed(ActionEvent e){...}
    // What to do when a button is pressed
}

public class XYZ{
  MyListener l = new MyListener();
    // ActionListener l
  MyButtons m = new MyButtons(l);
    // Button m, reports to l
}
```

Embedding the button in a panel

- To actually display the button, we have to do more

Embedding the button in a panel

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`

Embedding the button in a panel

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`
  - First import required Java packages

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Embedding the button in a panel

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`
  - First import required Java packages
  - The panel will also serve as the event listener

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel
     implements ActionListener{

    ...

}
```

Embedding the button in a panel

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`
  - First import required Java packages
  - The panel will also serve as the event listener
  - Create the button, make the panel a listener and add the button to the panel

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel
     implements ActionListener{

  private JButton redButton;

  public ButtonPanel(){
    redButton = new JButton("Red");
    redButton.addActionListener(this);
    add(redButton);
  }

  ...

}
```

Embedding the button in a panel

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`
  - First import required Java packages
  - The panel will also serve as the event listener
  - Create the button, make the panel a listener and add the button to the panel
- Listener sets the panel background to red when the button is clicked

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel
     implements ActionListener{
  private JButton redButton;

  public ButtonPanel(){
    redButton = new JButton("Red");
    redButton.addActionListener(this);
    add(redButton);
  }

  public void actionPerformed(ActionEvent evt){
    Color color = Color.red;
    setBackground(color);
    repaint();
  }
}
```

Embedding the panel in a frame

- Embed the panel in a frame — `JFrame`

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }

  // Implement WindowListener

  ..
}
```

Embedding the panel in a frame

- Embed the panel in a frame — `JFrame`
- Corresponding listener class is `WindowListener`

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }

  // Implement WindowListener

  ..
}
```

Embedding the panel in a frame

- Embed the panel in a frame — `JFrame`
- Corresponding listener class is `WindowListener`
- `JFrame` generates seven different types of events
  - Each of the seven events automatically calls a different function in `WindowListener`

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }
      ...
  }

  // Seven methods required for
  // implementing WindowListener
  // Six out of seven are stubs

  ...
}
```

Embedding the panel in a frame

- Embed the panel in a frame — `JFrame`
- Corresponding listener class is `WindowListener`
- `JFrame` generates seven different types of events
  - Each of the seven events automatically calls a different function in `WindowListener`
- Need to implement `windowClosing` event to terminate the window
- Other six types of events can be ignored

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }
      ...
  }

  // Six of seven methods required for
  // implementing WindowListener are stubs
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowActivated(WindowEvent e){}
  public void windowClosed(WindowEvent e){}
  public void windowDeactivated(WindowEvent e){}
  public void windowDeiconified(WindowEvent e){}
  public void windowIconified(WindowEvent e){}
  public void windowOpened(WindowEvent e){}
}
```

Embedding the panel in a frame

- One more complication

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }
      ...
  }

  // Six of seven methods required for
  // implementing WindowListener are stubs
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowActivated(WindowEvent e){}
  public void windowClosed(WindowEvent e){}
  public void windowDeactivated(WindowEvent e){}
  public void windowDeiconified(WindowEvent e){}
  public void windowIconified(WindowEvent e){}
  public void windowOpened(WindowEvent e){}
}
```

Embedding the panel in a frame

- One more complication
- `JFrame` is "complex", many layers

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  public ButtonFrame(){ ... }
      ...
  }

  // Six of seven methods required for
  // implementing WindowListener are stubs
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowActivated(WindowEvent e){}
  public void windowClosed(WindowEvent e){}
  public void windowDeactivated(WindowEvent e){}
  public void windowDeiconified(WindowEvent e){}
  public void windowIconified(WindowEvent e){}
  public void windowOpened(WindowEvent e){}
}
```

Embedding the panel in a frame

- One more complication
- `JFrame` is "complex", many layers
- Items to be displayed have to be added to `ContentPane`

```
public class ButtonFrame extends JFrame
       implements WindowListener {

  Private Container contentPane;

  public ButtonFrame(){
    setTitle("ButtonTest");
    setSize(300, 200);

    // ButtonFrame listens to itself
    addWindowListener(this);

    // ButtonPanel is added to the contentPane
    contentPane = this.getContentPane();
    contentPane.add(new ButtonPanel());
  }

  // Six of seven methods required for
  // implementing WindowListener are stubs
}
```

Finally, a main function

- Create a `JFrame` and make it visible

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest{
  public static void main(String[] args) {
    EventQueue.invokeLater(
      () -> {
          JFrame frame = new ButtonFrame();
          frame.setVisible(true);
      }
    );
  }
}
```

Finally, a main function

- Create a JFrame and make it visible
- EventQueue.invokeLater() puts the Swing object in a separate event despatch thread

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest{
  public static void main(String[] args) {
    EventQueue.invokeLater(
      () -> {
           JFrame frame = new ButtonFrame();
           frame.setVisible(true);
      }
    );
  }
}
```

Finally, a main function

- Create a JFrame and make it visible
- EventQueue.invokeLater() puts the Swing object in a separate event despatch thread
- Ensures that GUI processing does not interfere with other computation

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest{
  public static void main(String[] args) {
    EventQueue.invokeLater(
      () -> {
          JFrame frame = new ButtonFrame();
          frame.setVisible(true);
      }
    );
  }
}
```

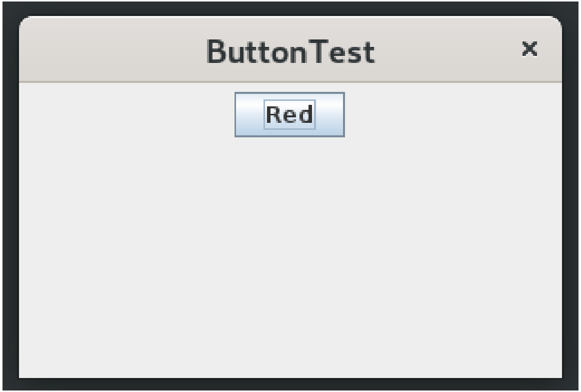
Finally, a main function

- Create a JFrame and make it visible
- EventQueue.invokeLater() puts the Swing object in a separate event despatch thread
- Ensures that GUI processing does not interfere with other computation
- GUI does not get blocked, avoid subtle synchronization bugs

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest{
  public static void main(String[] args) {
    EventQueue.invokeLater(
      () -> {
          JFrame frame = new ButtonFrame();
          frame.setVisible(true);
      }
    );
  }
}
```

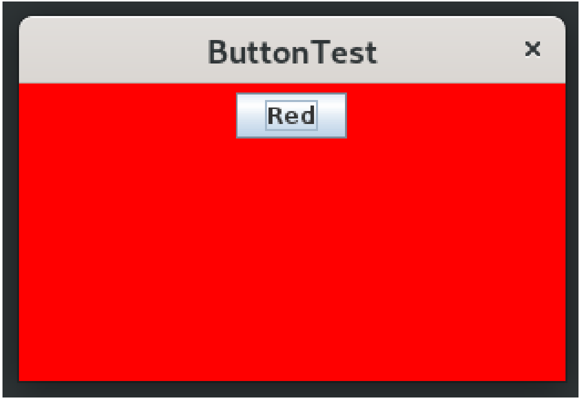
Finally, a main function

- Create a `JFrame` and make it visible
- `EventQueue.invokeLater()` puts the Swing object in a separate event despatch thread
- Ensures that GUI processing does not interfere with other computation
- GUI does not get blocked, avoid subtle synchronization bugs
- Output — before the button is clicked



Finally, a main function

- Create a `JFrame` and make it visible
- `EventQueue.invokeLater()` puts the Swing object in a separate event despatch thread
- Ensures that GUI processing does not interfere with other computation
- GUI does not get blocked, avoid subtle synchronization bugs
- Output — before the button is clicked
- . . . and after



Summary

- The Swing toolkit has different types of objects
- Each object generates its own type of event
- Create an appropriate event handler and link it to the object
- The unit that Swing displays is a frame
- Individual objects have to be embedded in panels which are then added to a frame

More Swing examples

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 12

Connecting multiple events to a listener

- One listener can listen to multiple objects

Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue

```
public class ButtonPanel extends JPanel
                    implements ActionListener{
  // Panel has three buttons
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");
    ...
  }

  public void actionPerformed(ActionEvent evt){
    ...
  }
}
```

Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons

```
public class ButtonPanel extends JPanel
                          implements ActionListener{
  // Panel has three buttons
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");
    // ButtonPanel listens to all three buttons
    yellowButton.addActionListener(this);
    blueButton.addActionListener(this);
    redButton.addActionListener(this);
    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

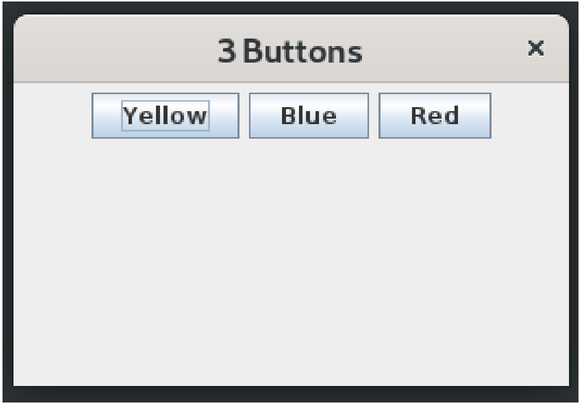
Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event
    - Keep the existing colour if the source is not one of these three buttons

```
public class ButtonPanel extends JPanel
                   implements ActionListener{
  ...
  public void actionPerformed(ActionEvent evt){
    // Find the source of the event
    Object source = evt.getSource();
    // Get current background colour
    Color color = getBackground();

    if (source == yellowButton)
      color = Color.yellow;
    else if (source == blueButton)
      color = Color.blue;
    else if (source == redButton)
      color = Color.red;

    setBackground(color);
    repaint();
  }
}
```

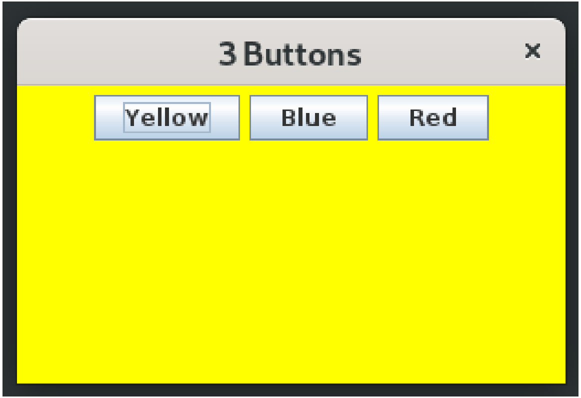
Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event
  - Keep the existing colour if the source is not one of these three buttons
- Output — before any button is clicked



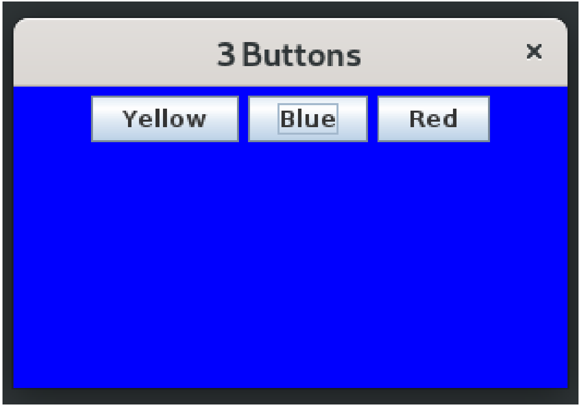
Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event
  - Keep the existing colour if the source is not one of these three buttons
- Output — before any button is clicked . . . and after each is clicked



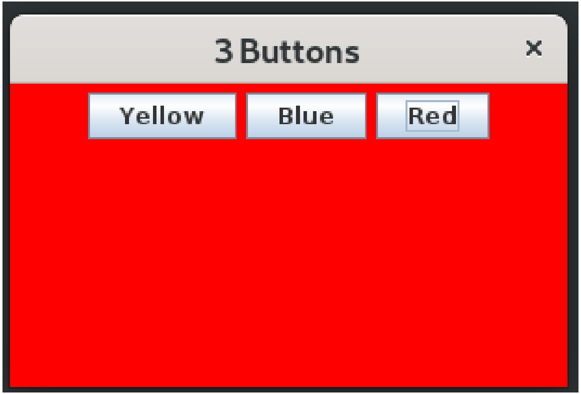
Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event
  - Keep the existing colour if the source is not one of these three buttons
- Output — before any button is clicked . . . and after each is clicked



Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event
  - Keep the existing colour if the source is not one of these three buttons
- Output — before any button is clicked . . . and after each is clicked



Multicasting: multiple listeners for an event

- Two panels, each with three buttons, `Red`, `Blue`, `Yellow`

```
import ...
public class ButtonPanel extends JPanel
                   implements ActionListener{
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    ...

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

Multicasting: multiple listeners for an event

- Two panels, each with three buttons, `Red`, `Blue`, `Yellow`
- Clicking a button in either panel changes background colour in both panels

```
import ...
public class ButtonPanel extends JPanel
                   implements ActionListener{
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    ...

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

Multicasting: multiple listeners for an event

- Two panels, each with three buttons, `Red`, `Blue`, `Yellow`
- Clicking a button in either panel changes background colour in both panels
- Both panels must listen to all six buttons
  - However, each panel has references only for its local buttons
  - How do we determine the source of an event from a remote button?

```
import ...
public class ButtonPanel extends JPanel
                         implements ActionListener{
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    ...

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...

```
import ...
public class ButtonPanel extends JPanel
                   implements ActionListener{
  private JButton yellowButton, blueButton,
                  redButton;

  public ButtonPanel(){
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    yellowButton.setActionCommand("YELLOW");
    blueButton.setActionCommand("BLUE");
    redButton.setActionCommand("RED");

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
```

Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`

```
public class ButtonPanel extends JPanel
                   implements ActionListener{
  ...
  public void actionPerformed(ActionEvent evt){
    Color color = getBackground();
    String cmd = evt.getActionCommand();

    if (cmd.equals("YELLOW"))
      color = Color.yellow;
    else if (cmd.equals("BLUE"))
      color = Color.blue;
    else if (cmd.equals("RED"))
      color = Color.red;

    setBackground(color);
    repaint();
  }
  ...
}
```

Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel

```
public class ButtonPanel extends JPanel
                         implements ActionListener{
  ...

  public void addListener(ActionListener o){
    // Add a commmon listener for all
    // buttons in this panel
    yellowButton.addActionListener(o);
    blueButton.addActionListener(o);
    redButton.addActionListener(o);
  }
}
```

Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, . . .
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame

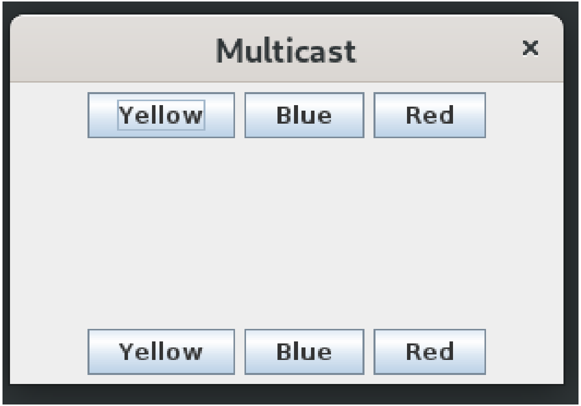
```
public class ButtonFrame extends JFrame
                  implements WindowListener{
  private Container contentPane;
  private ButtonPanel b1, b2;

  public ButtonFrame(){
    ..
    b1 = new ButtonPanel();   // Two panels
    b2 = new ButtonPanel();

    // Each panel listens to both sets of buttons
    b1.addListener(b1);  b1.addListener(b2);
    b2.addListener(b1);  b2.addListener(b2);

    contentPane = this.getContentPane();
    // Set layout to separate out panels in frame
    contentPane.setLayout(new BorderLayout());
    contentPane.add(b1,"North");
    contentPane.add(b2,"South");
  }
```

Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame
- How it works

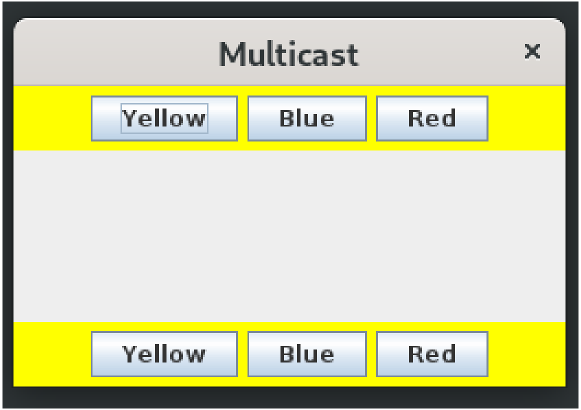
Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame
- How it works



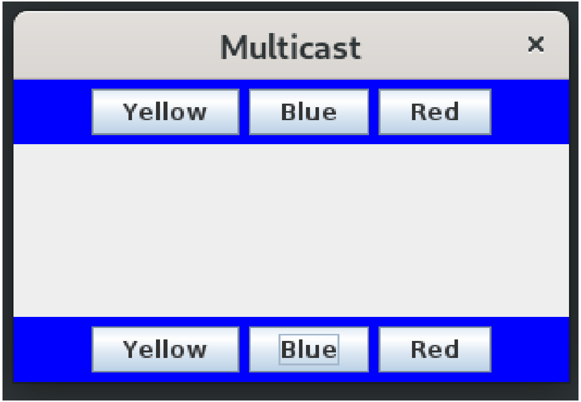
Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame
- How it works



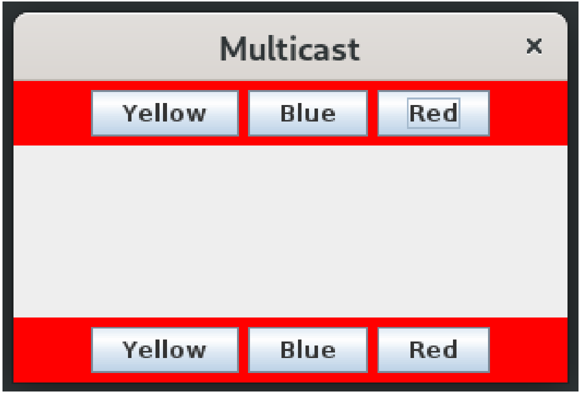
Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame
- How it works



Multicasting: multiple listeners for an event

- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons, ...
- Choose colour according to `ActionCommand`
- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel
- Add both panels to the same frame
- How it works



Other elements – checkboxes

- `JCheckbox`: a box that can be ticked

Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green

```
import ...
public class CheckBoxPanel extends JPanel
                    implements ActionListener{
  private JCheckBox redBox;
  private JCheckBox blueBox;

  public CheckBoxPanel(){
    redBox = new JCheckBox("Red");
    blueBox = new JCheckBox("Blue");

    ...
  }
}
```

Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again ActionListener

```
import ...
public class CheckBoxPanel extends JPanel
                   implements ActionListener{
  private JCheckBox redBox;
  private JCheckBox blueBox;

  public CheckBoxPanel(){
    redBox = new JCheckBox("Red");
    blueBox = new JCheckBox("Blue");

    redBox.addActionListener(this);
    blueBox.addActionListener(this);

    ...
  }
}
```

Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again ActionListener
- Checkbox state: selected or not

```
import ...
public class CheckBoxPanel extends JPanel
                      implements ActionListener{
  private JCheckBox redBox;
  private JCheckBox blueBox;

  public CheckBoxPanel(){
    redBox = new JCheckBox("Red");
    blueBox = new JCheckBox("Blue");

    redBox.addActionListener(this);
    blueBox.addActionListener(this);

    redBox.setSelected(false);
    blueBox.setSelected(false);

    add(redBox);
    add(blueBox);
    ...
  }
```

Other elements – checkboxes

- `JCheckbox`: a box that can be ticked
- A panel with two checkboxes, `Red` and `Blue`
  - Only `Red` ticked, background red
  - Only `Blue` ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again `ActionListener`
- Checkbox state: selected or not
- `isSelected()` returns current state

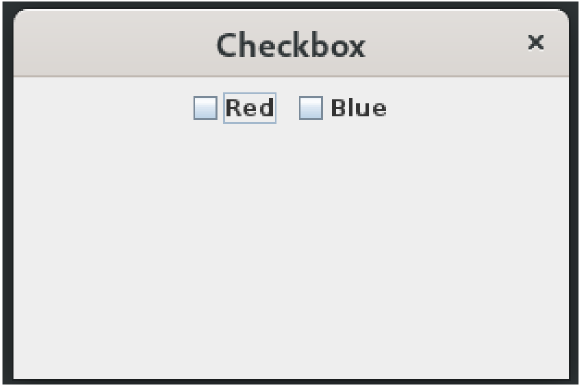
```
public class CheckBoxPanel extends JPanel
                     implements ActionListener{
  ...

  public void actionPerformed(ActionEvent evt){

    Color color = getBackground();

    if (blueBox.isSelected())
      color = Color.blue;
    if (redBox.isSelected())
      color = Color.red;
    if (blueBox.isSelected() &&
        redBox.isSelected())
      color = Color.green;

    setBackground(color);
    repaint();
  }
}
```

Other elements – checkboxes

- `JCheckbox`: a box that can be ticked
- A panel with two checkboxes, `Red` and `Blue`
  - Only `Red` ticked, background red
  - Only `Blue` ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again `ActionListener`
- Checkbox state: selected or not
- `isSelected()` returns current state
- Rest similar to basic button example

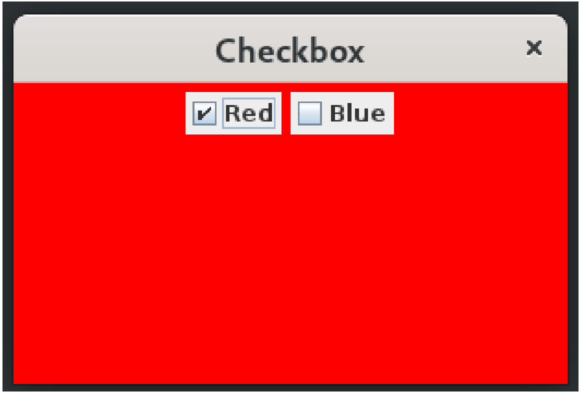
Other elements – checkboxes

- `JCheckbox`: a box that can be ticked
- A panel with two checkboxes, `Red` and `Blue`
  - Only `Red` ticked, background red
  - Only `Blue` ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again `ActionListener`
- Checkbox state: selected or not
- `isSelected()` returns current state
- Rest similar to basic button example



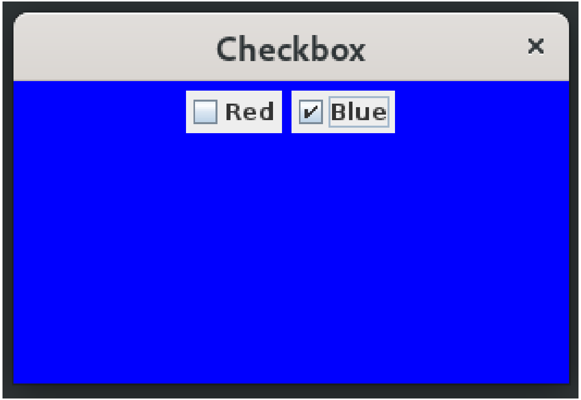
Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again ActionListener
- Checkbox state: selected or not
- isSelected() returns current state
- Rest similar to basic button example



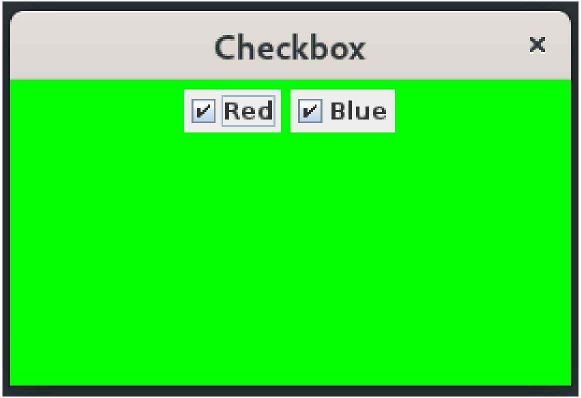
Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again ActionListener
- Checkbox state: selected or not
- isSelected() returns current state
- Rest similar to basic button example



Other elements – checkboxes

- JCheckbox: a box that can be ticked
- A panel with two checkboxes, Red and Blue
  - Only Red ticked, background red
  - Only Blue ticked, background blue
  - Both ticked, background green
- Only one action — click the box
  - Listener is again ActionListener
- Checkbox state: selected or not
- isSelected() returns current state
- Rest similar to basic button example



Summary

- Swing components such as buttons, checkboxes generate high level events
- Each event is automatically sent to a listener
  - Listener capability is described using an interface
  - Event is sent as an object — listener can query the event to obtain details such as event source, action label, ... and react accordingly
- Association between event generators and listeners is flexible
  - One listener can listen to multiple objects
  - One component can inform multiple listeners
- Must explicitly set up association between component and listener
  - Events are "lost" if nobody is listening!
- Swing objects are the most aesthetically pleasing, but useful to understand how GUI programming works across other languages