Introduction

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	1
Lecture URL	https://youtu.be/3a1FXBR6QXY
Notion URL	https://21f1003586.notion.site/Introduction-95ec26f2dc934c96a69762cad8582900
Week #	1

Programming Languages

- A language is a medium for communication
- Programming langauges 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 the 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 process

Abstraction

- Abstractions used in computational thinking
  - Assigning values to named variables
  - Conditional execution
  - Iteration
  - Functions/Procedures, recursion
  - Aggregate data structures — arrays, list, 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
- Imperative
  - How to compute
  - Step-by-step instrutions on what is to be done
- Declarative
  - What the computation should produce
  - Often exploit inductive structures, 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** (in Python)

```
def sum_list(l):
  sum = 0
  for x in l:
    sum += x
  return sum
```

- Intermediate values `sum, x`
- Explicit iteration to examine each element in the list
- **Declarative** (in Python)

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

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

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

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

- We can code functionally in an imperative language
- Helps us identify natural units of (reusable) code
- **Declarative** (in Python)

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

def square(x):
  return x * x

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

Names, types and values

- Internally, everything is stored as 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
  - Interpret 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 data types
  - Structured collection with fixed interface
  - Stack, for example, is a sequence but only allows `push` and `pop`
  - Separate implementation from interface
    - Priority queue allows `insert` and `delete-max`
    - Can implement a priority queues 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, `my_list.sort()` vs `sorted(my_list)`

What is yet to come ...

- 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 the features
- Discuss design decisions where relavant
  - Every language makes some compromises
- Understand and appreciate why there is a zoo of programming languages out there, *lol*
- And why new ones are still being created

Types

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	2
Lecture URL	https://youtu.be/0Gl9ygUk4K8
Notion URL	https://21f1003586.notion.site/Types-e3dd9b780c9d410086c44c63bffb6356
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
- Naming concepts and structuring our computation
  - Especially at a higher level
  - `Point` vs `(Float, Float)`
  - Banking applications → accounts of different types, customers, ...
- Catching bugs early
  - Incorrect expression evaluation
  - Incorrect assignment

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*** → derive type from the current value
  - `x = 10` — `x` is of type `int`
  - `x = 7.5` — now `x` is of type `float`

    - An uninitialized name has no type
- ***Static typing*** → associate a type in advance with a name
    - Need to declare names and their types in advance
    - `int x, float a, ...`
    - Cannot assign an incompatible value — `x = 7.5` is illegal

- It is 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 here!
  return factorlist
```

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

Types of 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 the 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 to customers
    - Updating personal details applies to customers, not accounts

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 declarations, compilers can detect type errors at compile time - ***static analysis***
    - Dynamic typing would catch these errors only when the code runs
    - Executing code also shows down due to simultaneous monitoring for type correctness
- Compilers can also perform optimizations based on static analysis
    - Re-order 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, everything will go fine

Memory Management

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	3
Lecture URL	https://youtu.be/b4nsGWXNm2c
Notion URL	https://21f1003586.notion.site/Memory-Management-bf48fedf690e4df5991b9b7e9b9e1480
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
- Scope of a variable
  - When the variable is available for use
  - In the following code, the `x` in `f()` is not in scope withing call to `g()`

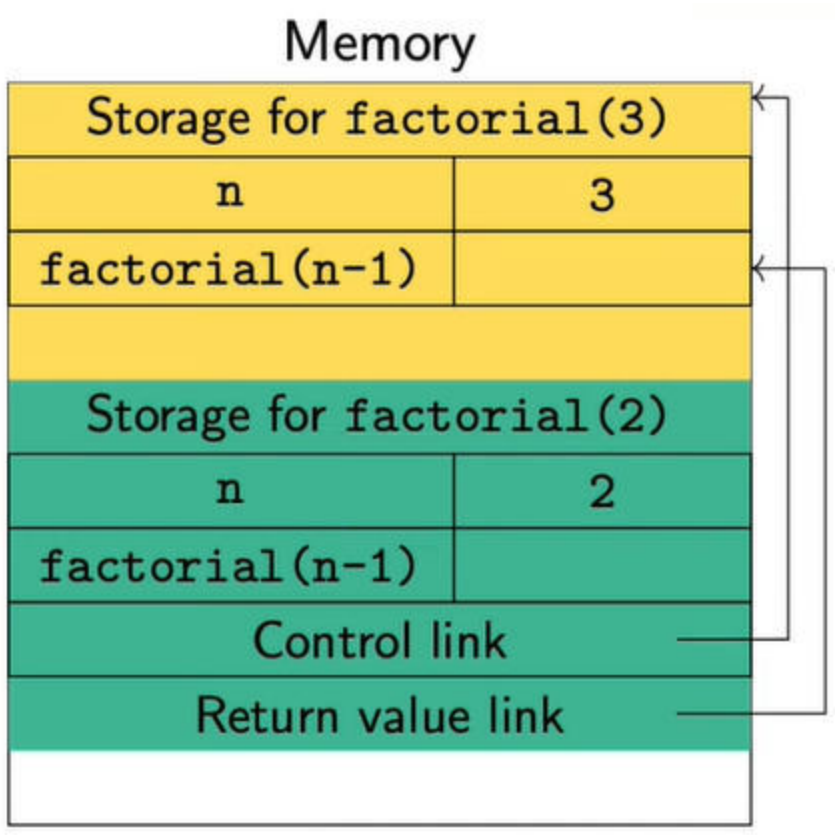
```
def f(l):
    ...
    for x in l:
      y = y + g(x)
    ...
```

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

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

Memory stack

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



Call `factorial(3)`

`factorial(3)` calls `factorial(2)`

- 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

Passing arguments to a function

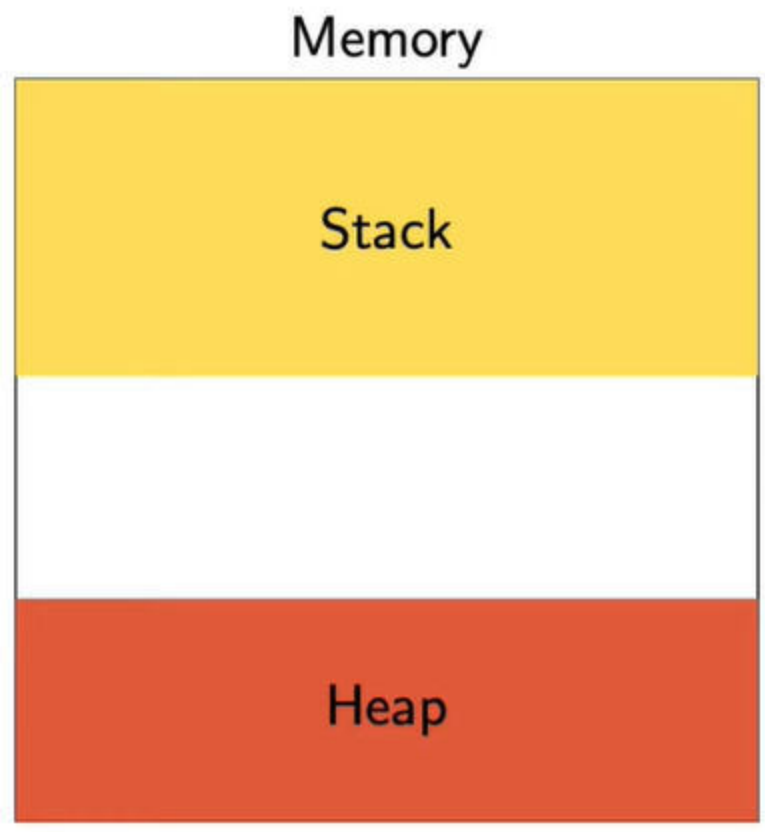
- When a function is called, arguments are substituted for formal parameters

```
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 not side effect
  - ***Call by reference*** → parameter points to same location as the argument
    - Can have side-effects
    - It 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
- We need a separate storage for the persistent data
  - Dynamically allocated vs statically declared
  - Usually called the heap
    - Not the same as heap data structure
  - Conceptually, allocate heap storage from “opposite” end with respect to the stack



- Heap store outlives activation record
  - Access through some variable that is in scope

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 is now dead storage, unreachable
- Manual memory management
  - Programmer explicitly requests and returns heap storage
    - `p = malloc(...)` and `free(p)` in C language
  - Error-prone — memory leaks, invalid assignments
- Automatic garbage collection (Java, Python, ...)
  - Run-time environment checks and cleans up dead storage
    - Mark all storage that is reachable from program variables
    - Return all unmarked memory cells to free space
  - Convenience for programmer vs peformance 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 results
- 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

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	4
Lecture URL	https://youtu.be/8ciDI5cUhS
Notion URL	https://21f1003586.notion.site/Abstraction-and-Modularity-d39c0b22b59a4238b7173fef6074ebdc
Week #	1

Stepwise Refinement

- Begin with a high level description of the task
- Refine the tasks 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()`
- How do we represent each amount?
  - Only need the current balance
  - Overall, an array of balance

- 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
- 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 the 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, re-use 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 the 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 tasks 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
- Programming Language support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming

Object-Oriented Programming

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	5
Lecture URL	https://youtu.be/NmYcNMPUlzY
Notion URL	https://21f1003586.notion.site/Object-Oriented-Programming-3bb75c2bcc92484d96ff97ef71bffed9
Week #	1

Objects

- An object is like an abstract datatype
  - Hidden data with set of public operations
  - All interactions through operations — messages, methods, member-functions ...
- Uniform way of encapsulating different combinations of data and functionality
  - An object can hold single integer — eg. a counter
  - An entire filesystem or database coule 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, $1960s$
- 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-types, yet hold all types of events
    - Use a generic simulation operation across different types of events
        - Avoid elaborate checking of cases

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?
- 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
- 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 runtime type `A`

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()`
- 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 runtime
    - Inheritance
        - Reuse of implementations

Classes and Objects

Type	📒 Lecture
Date	@December 27, 2021
Lecture #	6
Lecture URL	https://youtu.be/TJC0WhS6FNo
Notion URL	https://21f1003586.notion.site/Classes-and-Objects-a6d38f8b11b44c269cb11bb0eb209107
Week #	1

Programming with Objects

- Objects are like abstract datatypes
  - Hidden data with set of public operations
  - All interactions through operations — methods ...
- ***Class***
  - Template for a data type
  - How a data is stored
  - How public functions manipulate the data
- ***Object***
  - Concrete instance of the above mentioned template
  - Each object maintains its separate copy of local data
  - Invoke methods on objects — Equivalent to "***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` — these are called 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 = y
```

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

```
# This will go inside the Point class
def translate(self, dx, dy):
  self.x += dx
  self.y += dy
```

- Distance from the origin
  - $d = \sqrt{x^2 + y^2}$
  - Does not update instance variables
  - state of object is unchanged

```
# This will go inside the Point class
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)
  - $r = \sqrt{x^2 + y^2}$
  - $\theta = tan^{-1}(y/x)$
- 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
```

- Translation
  - Convert (r, θ) to (x, y)
  - $x = rcos\theta, y = rsin\theta$
  - 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

- Users of our code should not know whether `Point` uses `(x, y)` or `(r, theta)`
  - Interface remains indentical, or should remain 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)`
  - This defeats the purpose of abstraction
  - Changing the internal implementation of `Point` can have impact on other code
    - Usually called breaking changes

```
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
```

- The following code is legal

  `s = Square(5)`

  `a = s.area()`

  `p = s.perimeter()`
  - `Square` inherits definitions from `area()` and `perimeter()` from `Rectangle`

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
```

- The following code gives a runtime 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

---

- Subclasses 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
```

- The following code gives a runtime 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

- 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

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 class 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 subtypes and parent types
- Use strong declarations to enfore privacy, types
  - Do not rely on programmer disciplines
  - Catch bugs early through type checking

A First Taste of Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	1
Lecture URL	https://youtu.be/ULpvhw2hmSg
Notion URL	https://21f1003586.notion.site/A-First-Taste-of-Java-c8303394228e4344969df9d13f8c4bd0
Week #	2

Getting Started

- In Python

```
print('hello, world')
```

- In C ...

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

- And then there is Java ...

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

Why so complicated?

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

- All code in Java lies 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()`
- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameters is an array of strings
    - Command line arguments
  - No output, hence the `main` function has the return type `void`
- Visibility
  - Function has to be available to run from outside the class
  - Modifier `public`
- Availability
  - Functions defined inside classes are attached to the objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects, *belongs to the class*
- 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()`
    - Adds a newline character `\n` at the end of the statement
- Punctuation `{`, `}`, `;` to delimit the blocks, statements
  - Unlike layout and indentation in Python

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`
- Java programs are usually interpreted on ***Java Virtual Machine (JVM)***
  - JVM provides a uniform execution environment across OSes
  - Semantics of Java is defined in terms of JVM, OS-independent
  - It came with a slogan “Write once, run anywhere”
- `javac` compiles into JVM ***bytecode***
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`
- Note:
  - `javac` requires file extension `.java`
  - `java` should not be provided file extension `.class`

    - `javac` automatically follows dependencies and compiles all the 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` and `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

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	2
Lecture URL	https://youtu.be/IwTdVsLxz04
Notion URL	https://21f1003586.notion.site/Basic-Datatypes-in-Java-3cfdcb7953c14d4890ccc104b8264b26
Week #	2

Scalar Types

- In an object-oriented langauge, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has 8 primitive datatypes
  - `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;
float y;
char c;
boolean b1;
```

- The assignment statement works as usual

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

- Characters are written with single-quotes (only)
  - Double quotes mark string

```
char c, d;
c = ‘x’;
d = ‘\u03C0’;  // Greek pi, unicode
```

- Boolean constants are `true, false`

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

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
float y;
```

  - Use this judiciously to retain readability
- Initialize at the time of declaration

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

- Modifier `final` marks as constant

```
final float pi = 3.1415927f;
pi = 22/7;  // Flagged as error
```

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 operator, use `Math.pow()`
- `Math.pow(a, n)` returns a^n
- Special operators for incrementin 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;
  ```

- 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"
  ```

- String are not arrays of characters
  - We cannot write the following

    ```
    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
  - Strings are immutable
  - After the update, `s` points to a new `String`
  - Java does automatic garbage collection

Arrays

- Arrays are also objects
- Typical declaration

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

  or

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

- `a.length` gives the 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

Control Flow in Java

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	3
Lecture URL	https://youtu.be/0RHDEourhdo
Notion URL	https://21f1003586.notion.site/Control-Flow-in-Java-3a68f0e8e0fb49a9babfea919410dc68
Week #	2

Control flow

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

- `do { ... } while (c);`
    - Condition is checked at the end of the loop
    - At one iteration, even if the condition is false
    - 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 language
- `for (init; cond; upd) { ... }`
    - `init` is initialization
    - `cond` is terminating condition
    - `upd` is loop update

- Intended use is

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

- Completely equivalent to

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

- However, not a 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

Sample code

```
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);
  }
}
```

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`

Sample code

```
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
- The default behaviour of `switch` is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch

    - `break` available for loops as well
- Options have to be constants
    - Cannot use conditional expressions
- Aside: here return type is `void`
    - Non-void return type requires an appropriate return value

Sample code

```
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

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	4
Lecture URL	https://youtu.be/XB-PcJJpKXg
Notion URL	https://21f1003586.notion.site/Defining-Classes-and-Objects-in-Java-5064c2d02789467897f97abe6e60c846
Week #	2

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 the same directory form part of the same package
- Instance variable
  - Each concrete object of type `Date` will have local copies of `date, month, year`
  - These are marked `private`
  - Can also have `public` instance variable, but breaks encapsulation

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

Creating Object

- Declare type using class name
- `new` creates a new object

    - How do we set the instance variables?
- We can add methods to update values
    - `this` is a a reference to current object

```
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;
  }
}

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

    - We 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()`
- 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` keyword
- 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, we 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
- Copy constructor takes an object of the same type as an 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 the old one
  - If instance variable are objects, we may end up aliasing rather than copying

```
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
  - Re-use — 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

Type	📒 Lecture
Date	@January 3, 2022
Lecture #	5
Lecture URL	https://youtu.be/zvtvQcUhFxo
Notion URL	https://21f1003586.notion.site/Basic-Input-and-Output-in-Java-9629c4bcf2de4676bf073f0997d4f87d
Week #	2

Interacting with a Java program

- We have seen how to print data
  - `System.out.println(”Hello, world”);`
- But how do we read data?

Reading input

- Simplest to use is the `Console` class
  - Functionality quite 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`

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

- A more general `Scanner` class
  - Allows more granular reading of the 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
- `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 language

The philosophy of OO programming

Type	📒 Lecture
Date	@January 8, 2022
Lecture #	1
Lecture URL	https://youtu.be/PF6bAB-d9hI
Notion URL	https://21f1003586.notion.site/The-philosophy-of-OO-programming-43af67bf48ea4fd99b7b3a7a488fb8f3
Week #	3

Algorithms + Data Structures = Programs

- Title of Niklaus Wirth’s introduction to Pascal
- Traditionally, algorithms come first
- Data representation comes later

Object Oriented design

- Reverses the traditional 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 maintaining 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
- What happens to these objects?
    - Items are *added* to orders
    - Orders are *shipped, cancelled*
    - Payments are *accepted, rejected*
- Nouns signify objects, verbs denote the 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?
- State — how does the object react when methods are invoked?
    - State is the info 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
- Aggregation
    - `Order` contains `Item` objects
- Inheritance
    - One object is a specialized version 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

Subclasses and Inheritance

Type	📒 Lecture
Date	@January 8, 2022
Lecture #	2
Lecture URL	https://youtu.be/6tDGUnETvv4
Notion URL	https://21f1003586.notion.site/Subclasses-and-Inheritance-25f005c285f04a58927eed6577ddb8f2
Week #	3

A typical Java class

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

  /* Constructors ... */

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

  // Accessor methods
  public String getName() { ... }
  public String getSalary() { ... }

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

What do we have here

- 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

Subclasses

- Managers are special types of employees with extra features

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

- `Manager` object 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`
- `Manager` objects do not automatically have access to private data of the 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`?
  - 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);
  }
}

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

Inheritance

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

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

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

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 the constructor of parent class

Dynamic dispatch and polymorphism

Type	📒 Lecture
Date	@January 8, 2022
Lecture #	3
Lecture URL	https://youtu.be/zN-6WryuloU
Notion URL	https://21f1003586.notion.site/Dynamic-dispatch-and-polymorphism-0d56d1e9d22a468d9f2fd3744afd0b25
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 the 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 {
  private String secretary;
  public boolean setSecretary(String 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
- 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

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

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

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

- 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

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);  // Sort the contents of darr
Arrays.sort(iarr);  // Sort the contents of iarr


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

- **Overloading:** multiple methods, same name, different signatures (different parameters), choice is static
- **Overriding:** multiple methods, same name, same signature, choice is static

    - `Employee.bonus()`
    - `Manager.bonus()`
- **Dynamic dispatch:** multiple methods, same signature, choice made at run-time

Type casting

- Consider the following assignment

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

  `((Manager) e).setSecretary(s)`
- Cast fails (errors at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

  `if(e instanceof Manager) { ((Manager) e).setSecretary(s); }`
- We can also use type casting for basic data types

  `double d = 29.98;`

  `int nd = (int) d;`

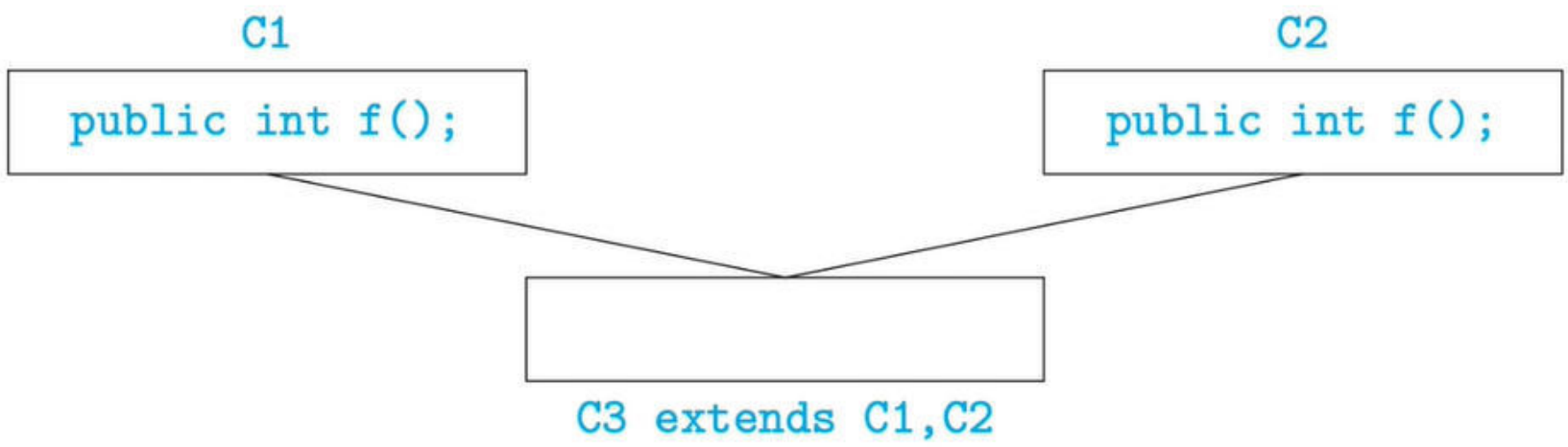
Summary

- A sub-class 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 functions for array of any datatype that supports a comparison operation
    - Java uses the term *generics*
- Use type-casting (and reflection) overcome static type restrictions

The Java class hierarchy

Type	Lecture
Date	@January 8, 2022
Lecture #	4
Lecture URL	https://youtu.be/WRN3QWICaag
Notion URL	https://21f1003586.notion.site/The-Java-class-hierarchy-42f67eff075e438a81fe1dc7c31725fa
Week #	3

Multiple inheritance



- Can a sub-class 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 only if `C1` and `C2` have no conflict

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 value 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()`
- 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`
- 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 this 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;
}
```

- 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 sub-class can only extend 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

Type	📒 Lecture
Date	@January 8, 2022
Lecture #	5
Lecture URL	https://youtu.be/CYY7IT-YHVA
Notion URL	https://21f1003586.notion.site/Subtyping-vs-inheritance-74043d6113e04d2faeab33d3706f824a
Week #	3

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 re-use 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
  - `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`

Java modifier

Type	📒 Lecture
Date	@January 8, 2022
Lecture #	6
Lecture URL	https://youtu.be/IO-K87_QXGs
Notion URL	https://21f1003586.notion.site/Java-modifier-fab02a734e3b4208b6c190821c5e9a79
Week #	3

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`
- We also use `private` methods sometimes to do some work that shouldn’t be directly accessible to public
- Example: a `Stack` class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values;
  private int size;
  private int top;

  public void push(int i) {
    ...
```

```
  }

  public int pop() {
    ...
  }

  public boolean isEmpty() {
    return top == 0;
  }
}
```

- `push()` needs to check if the stack has space

```
public class Stack {
  ...
  public void push(int i) {
    if(top < size) {
      values[top] = i;
      top += 1
    } else {
      // Deal with stack overflow
    }
    ...
  }
  ...
}
```

- 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 operation
  }
  ...
  private boolean stack_full() {
    return top == size;
  }
  private void extend_stack() {
    /* Allocate additional space, reset size etc. */
  }
}
```

Accessor and Mutator

- 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 int getDay() { ... }
  public int getMonth() { ... }
  public int getYear() { ... }

  public void setDay(int d) { ... }
  public void setMonth(int m) { ... }
  public void setYear(int y) { ... }
}
```

- Instead, only allow combined update

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

  public int getDay() { ... }
  public int getMonth() { ... }
  public int getYear() { ... }

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combinations
  }
}
```

`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`
- We do use `private static` sometimes
- 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
- Concurrent updates need some care

`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 re-define functions at run-time

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
- A `final` method cannot be overridden by a subclass
- We can also have `private` classes

Abstract Classes and Interfaces

Type	📒 Lecture
Date	@January 14, 2022
Lecture #	1
Lecture URL	https://youtu.be/RiGkT9NDof4
Notion URL	https://21f1003586.notion.site/Abstract-Classes-and-Interfaces-e462336f5d97499d8c8b1d6d09ea045d
Week #	4

Abstract classes

- Provide an abstract definition of the method

  `public abstract double perimeter();`

- Forces sub-classes to provide a concrete implementation
- Cannot create objects from a class that has abstract functions
- Also, the class containing an abstract function must be declared abstract

```
public abstract class Shape {
  ...
  public abstract double perimeter();
  ...
}
```

- We 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(int i = 0; i < 3; ++i) {
  sizearr[i] = shapearr[i].perimeter()
  // Here, each shapearr[i] calls the appropriate method
  ...
}
```

Generic Functions

- We can use abstract classes to specify generic properties

```
public abstract class Comparable {
  public abstract int cmp(Comparable s);
  // return -1 if this < s
  // return 0 if this == s
  // return 1 if this > s
}
```

- Now we can sort any array of objects that extend `Comparable`

```
public class SortFunctions {
  public static void quicksort(Comparable[] a) {
    ...
    // Code for quicksort goes here
    // Except that to compare a[i] and a[j], we use a[i].cmp(a[j])
  }
}
```

- To use this

```
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
      // We have to cast in order to access s.size
    }
  }
}
```

Multiple inheritance

- 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) { ... }
  ...
}
```

We 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 abstract 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 can extend only one parent class, but it can implement any number of interfaces

Interfaces

Type	📒 Lecture
Date	@January 14, 2022
Lecture #	2
Lecture URL	https://youtu.be/adkU2TMbJfk
Notion URL	https://21f1003586.notion.site/Interfaces-7255fac9db6d4cb98f4961bb2b66f699
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
- 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) {
    ...
```

```
    // Code for quicksort goes here, 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;
  // Return 0 if this == s;
  // Return +1 if this > 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 += "0 if this == 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
- 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 the conflict 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

Type	📒 Lecture
Date	@January 14, 2022
Lecture #	3
Lecture URL	https://youtu.be/6eZ9mNX-GQ8
Notion URL	https://21f1003586.notion.site/Private-classes-d69d60fc19a24745a7d6c9bbf687d9d1
Week #	4

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;
  ...
}
```

- `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;
  }
}
```

- 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 class Node {
    private Object data;
    private Node next;
    ...
  }

  private int size;
  private Node first;

  public Object head() { ... }

  public void insert(Object newData) { ... }
}
```

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

Type	📒 Lecture
Date	@January 14, 2022
Lecture #	4
Lecture URL	https://youtu.be/YefHD5_AGiw
Notion URL	https://21f1003586.notion.site/Controlled-interaction-with-Objects-0aa4802d22c749e3b156b477a8f9e078
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
- Can ensure data integrity by regulating access

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

  public int getDay() { ... }
  public int getMonth() { ... }
  public int getYear() { ... }

  public void setDay(int d) { ... }
  public void setMonth(int m) { ... }
  public void setYear(int y) { ... }
}
```

- Update data as a whole, rather than individial components

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

  public int getDay() { ... }
  public int getMonth() { ... }
  public int getYear() { ... }
```

```
  public void setDate(int d, int m, int y) {
    ...
    // Validate the d-m-y combination
  }
}
```

Querying a database

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require the user to login 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 the number of seats available
    // on train number trainno on date d
    ...
  }
}
```

- Need to connect the query to the logged in status of the user
- Use objects
  - On login, user receives an object that can make a query
  - Object is created from private class that can lookup `railwayDB`

```
public class RailwayBooking {
  private BookingDB railwayDB;

  private class QueryObject {
    public int getStatus(int trainno, Date d) {
      // Return the number of seats available
      // on train trainnoon date d
      ...
    }
  }

  public QueryObject login(String u, String p) {
    QueryObject qobj;
    if(validLogin(u, p)) {
      qobj = new QueryObject();
      return qobj;
    }
  }
}
```

- How does the user know the capabilities of the private class `QueryObject`?
- Use an interface
  - Interface describes the capabilities 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(validLogin(u, p)) {
      qobj = new QueryObject();
      return qobj;
    }
  }

  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d) {
      ...
```

```
    }
  }
}
```

- 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(validLogin(u, p)) {
      qobj = new QueryObject();
      return qobj;
    }
  }

  private class QueryObject implements QIF {
    private int numQueries;
    private final int QLIM;

    public int getStatus(int trainno, Date d) {
      if(numQueries < QLIM) {
        // Respond, incrememement numQueries
        ...
      }
      ...
    }
  }
}
```

Summary

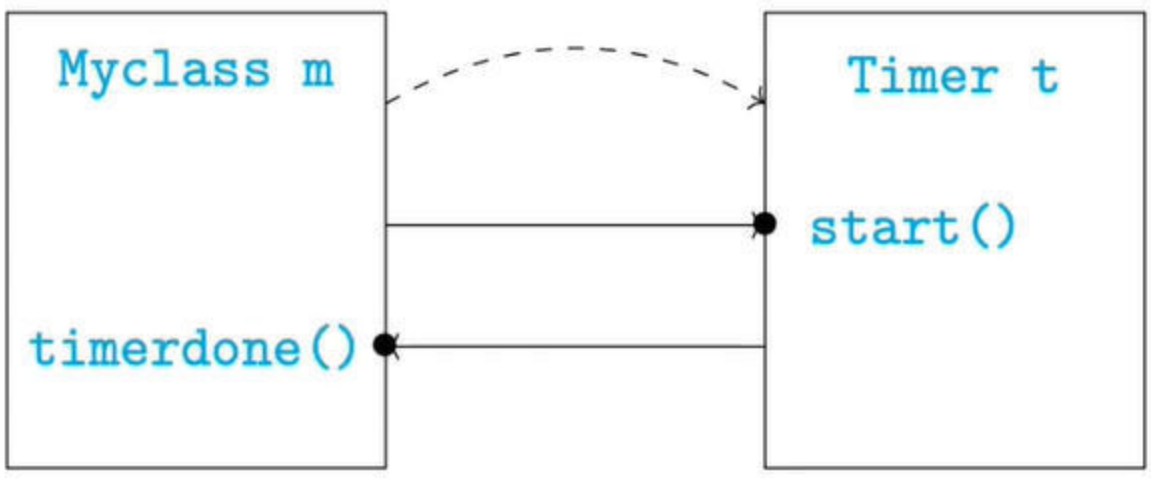
- Can provide controlled access to an object
- Combine private classes with interfaces
- External interaction is through an object of the private class
- Capabilities of this object are know through a public interface
- Object can maintain instance variables to track the state of the interaction

Callbacks

Type	📒 Lecture
Date	@January 14, 2022
Lecture #	5
Lecture URL	https://youtu.be/CKjGnZCvlng
Notion URL	https://21f1003586.notion.site/Callbacks-3ed65be72c694d3d8d7b9d78bb505138
Week #	4

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`
- `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` is specific to `MyClass`
  - We need 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;  // Creator
  }

  public void start() {
    ...
    owner.timerDone();
  }
}
```

A generic timer

- A 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;  // Creator
  }

  public void start() {
    ...
    ((MyClass) owner).timerDone();
  }
}
```

Use interfaces

- Define an interface for callback
- Modify `MyClass` to implement `TimerOwner`
- Modify `Timer` so that owner is compatible with `TimerOwner`

```
public interface TimerOwner {
  public abstract void timerDone();
}

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;  // Creator
  }

  public void start() {
    ...
    owner.timerDone();
  }
}
```

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

Type	Lecture
Date	@January 14, 2022
Lecture #	6
Lecture URL	https://youtu.be/BG_Btui0K1o
Notion URL	https://21f1003586.notion.site/Iterators-36904f8c2b464d8a87c7dcf96aed5d11
Week #	4

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;

  public LinearList() { size = 0; }

  public void append(Object o) {
    data[size++] = o;
    ...
  }
  ...
}
```

- 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 the values in a linear list
- If the list is an array with public access, we could write this

```
int i;
for(i = 0; i < data.length; ++i) {
  ... // do something with data[i]
}
```

- For a linked list with public access, we could write this

```
Node m;
for(m = head; m !+ null; m = m.next) {
  ... // do something with m.data
}
```

- But we do not have public access
- And we do not know which implementation is in use either

Iterator

- 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();
}
```

- How do we implement `Iterator` in `LinearList`?
- Need a pointer to remember position of the iterator
- How do we handle nested loops?

```
for(int i = 0; i < data.length; ++i) {
  for(int j = 0; i < data.length; ++j) {
    ... // do something with data[i] and data[j]
  }
}
```

- Solution → Create an `Iterator` object and export it

```
public class LinearList {
  private class Iter implements Iterator {
    private Node position;
    public Iter() { ... }
    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 the linear list

- 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 pre-specified 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;
}
```

Polymorphism

Type	📒 Lecture
Date	@January 22, 2022
Lecture #	1
Lecture URL	https://youtu.be/xvzxGV3pdMM
Notion URL	https://21f1003586.notion.site/Polymorphism-7f72e8aba0ab4f45ab160c5c3f6ad48a
Week #	5

Polymorphism

- In object-oriented programming, polymorphism (*inheritance*) usually refers to the effect of dynamic dispatch
  - `S` is a sub-class 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

- Uses the Java class hierarchy to simulate this
- Polymorphic `reverse`

```
public void reverse(Object[] arr) {
  int n = arr.length;

  for(int i = 0, i < n/2; ++i) {
    Object temp = arr[i];
    arr[i] = arr[(n - 1) - i];
    arr[(n - 1) - i] = temp;
```

```
  }
}
```

- Polymorphic `find`
  - `==` translates to `Object.equals()`

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

- 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
- Need to ensure that target array is type compatible with source array
  - Type errors should be flagged at compile time

```
public void arraycopy(Object[] src, Object[] target) {
  int limit = Math.min(src.length, target.length);
  for(int i = 0; i < limit; ++i) {
    target[i] = src[i];
  }
}

Date[] datearr = new Date[10];
Employee[] emparr = new Employee[10];

arraycopy(datearr, emparr); // Runtime error
```

- More generally source array can be a subtype of the target array
  - But the converse is illegal

```
public void arraycopy(Object[] src, Object[] target) {
  int limit = Math.min(src.length, target.length);
  for(int i = 0; i < limit; ++i) {
    target[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
// But arraycopy(tktarr, etktarr) is illegal
```

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;
    ...
  }
}
```

- 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
```

  - 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 the same type `T` as enclosing class
- Can describe subclass relationships between type variables
  - `public <S extends T,T> void arraycopy(S[] src, T[] target) { ... }`

Generics programming in Java

Type	📒 Lecture
Date	@January 23, 2022
Lecture #	2
Lecture URL	https://youtu.be/wqqOMQb0ft0
Notion URL	https://21f1003586.notion.site/Generics-programming-in-Java-c7282fcce5ca4faa844fcd0b12da63d5
Week #	5

Java Generics

- Use type variables
- Polymorphic `reverse` in Java
  - Type quantifier before return type
  - “For every type `T` ...”

```
public <T> void reverse(T[] arr) {
  T temp;
  int n = arr.length;

  for(int i = 0; i < n / 2; ++i) {
    temp = arr[i];
    arr[i] = arr[(n - 1) - i];
    arr[(n - 1) - i] = temp;
  }
}
```

- Polymorphic `find` in Java
  - Searching for a value of incompatible type is now a compile-time error

```
public <T> int find(T[] arr, T o) {
  for(int i = 0; i < arr.length; ++i) {
    if(arr[i] == o) return i;
  }

  return -1;
}
```

- Polymorphic `arraycopy`

- Source and target types must be identical

```
public <T> static void arraycopy(T[] src, T[] target) {
  int i, limit;
  limit = Math.min(src.length, target.length);

  for(i = 0; i < limit; ++i) {
    target[i] = src[i];
  }
}
```

- A more generous `arraycopy`
  - Source and target types may be different
  - Source type must extend target type

```
public <S extends T, T> static void arraycopy(T[] src, T[] target) {
  int i, limit;
  limit = Math.min(src.length, target.length);

  for(i = 0; i < limit; ++i) {
    target[i] = src[i];
  }
}
```

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;
    ...
  }
}
```

- 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);
```

- Be careful not to accidentally hide a type variable

  `public <T> void insert(T newDate) { ... }`
- `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[] target) { ... }`

```
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 <S extends T, T> void arraycopy (S[] src, T[] target) { ... }`
- Be careful not to accidentally hide type variables

  `public <T> void insert(T newData) { ... }` inside `class LinkedList<T>`

  vs

  `public <T> static void arraycopy (T[] src, T[] target) { ... }`

Java Generics and Subtyping

Type	📒 Lecture
Date	@January 23, 2022
Lecture #	3
Lecture URL	https://youtu.be/Dd9rAJo5H9s
Notion URL	https://21f1003586.notion.site/Java-Generics-and-Subtyping-d795e88a149c4cb887aa1a25dc9a7c3c
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[]
```

- But

```
...
ticketarr[5] = new Ticket();
// Not OK because 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>`
- 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);
  }
}
```

- `<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

```
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
- We can define variables of a wildcard type

  `public class LinkedList<T> { ... }`

  `LinkedList<?> l;`

- But, we 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`
- `Shape` has a method `draw()`
- All sub-classes override `draw()`
- We want a function to draw all elements in a list of `Shape` compatible objects

```
public static void drawAll(LinkedList<? extends Shape>) {
  Object o;
```

```
  Iterator i = l.get_iterator();

  while(i.has_next()) {
    o = i.get_next();
    o.draw();
  }
}
```

- Copying a `LinkedList`, using a wildcard

```
public static <? extends T, T> void listcopy(LinkedList<?> src, LinkedList<T> tgt) {
  Object o;
  Iterator i = src.get_iterator();

  while(i.has_next()) {
    o = i.get_next();
    tgt.add(o);
  }
}
```

- Can reverse the constraint, using `super`

```
public static <T, ? super T> void listcopy(LinkedList<T> src, LinkedList<?> tgt) {
  Object o;
  Iterator i = src.get_iterator();

  while(i.has_next()) {
    o = i.get_next();
    tgt.add(o);
  }
}
```

Summary

- Java generics are not covariant, unlike arrays
- Cannot substitute `Object` for `T` to get the 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

Type	Lecture
Date	@January 24, 2022
Lecture #	4
Lecture URL	https://youtu.be/bPpWwY6YLyg
Notion URL	https://21f1003586.notion.site/Reflection-ec355c74651049079fb11673cd032bbc
Week #	5

Reflection

According to Wikipedia

Reflective programming or ***reflection*** is the ability of a process to examine, introspect, and modify its own structure

- 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) {
  ...
}
```

- 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 the same type
  ...
}
```

- Can’t use `instanceof`
  - We 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 instanceof T) { ... }`

Introspection in Java

- We 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` that explicitly uses this face

```
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

- We can create new instances of a class at runtime

```
...
Class c = obj.getClass();
Object o = c.newInstance();
// Create a new object of the 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`
- 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
- 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

- 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[]`
- 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 argument 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, ...?
- `getConstructors()`, ... only returns publicly defined values
- Separate functions to also include private components
  - `getDeclaredConstructors()`
  - `getDeclaredMethods()`
  - `getDeclaredFields()`
- Should this be allowed to all programs as this is a security issue

- Access to private components may be restricted through external security policies

Using reflection

- `BlueJ`, a programming environment to learn Java
- It 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

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

Type	📒 Lecture
Date	@January 24, 2022
Lecture #	5
Lecture URL	https://youtu.be/V7l0pVIIaEU
Notion URL	https://21f1003586.notion.site/Java-Generics-at-run-time-ab5e5c54b02c4f8cbaae7fddcae58f2e
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>) { ... }`
- 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 the 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[]`
- 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 that generates 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>`
- Basic types `int, float, ...` are not compatible with `Object`
- We cannot use basic type in place of a generic type variable `T`
  - We 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`

- 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 their 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

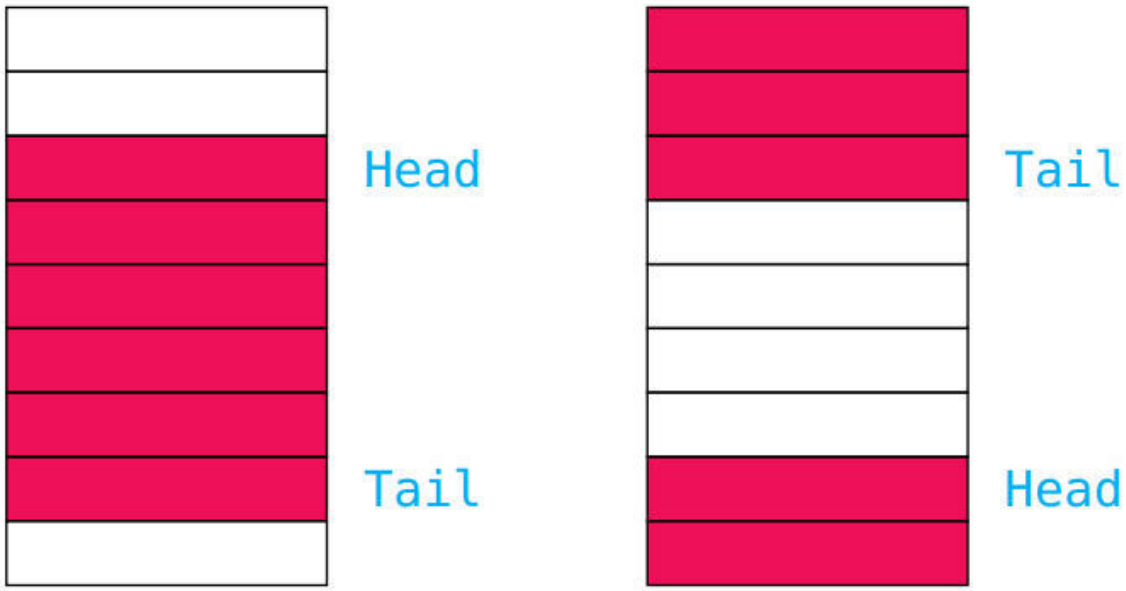
Type	📒 Lecture
Date	@February 4, 2022
Lecture #	1
Lecture URL	https://youtu.be/LY7TRCI3z24
Notion URL	https://21f1003586.notion.site/The-benefits-of-Indirection-866f9be0ef8046bb922186b620d02b8d
Week #	6

Abstract data types

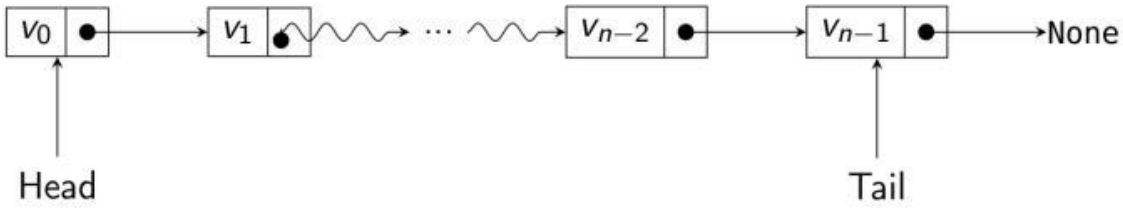
- It separates the public interface from the private implementation
- For example, a genetic Queue

```
public class Queue<E> {
  public void add(E element) { ... }
  public E remove() { ... }
  public int size() { ... }
  ...
}
```

- Concrete implementation could be a circular array



- or a Linked List



- The person who implemented the class `Queue` can choose either one
- The public interface remains un-changed

- But the `interface` does not capture other aspects, like ...
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked List is better — circular array has a bounded size

Multiple implementations

- 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() { ... }
  ...
}
```

- The user gets to decide ...

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq = new CircularArrayQueue<Date>();
stringq = new LinkedListQueue<String>();
```

- But we'd have to change the declaration of `dateq` if we later felt the need for a flexible size `dateq`
    - Also, we'd have to make changes to every function header, auxiliary variables, ... etc. associated with it

Adding indirection

- Instead of whatever we did earlier, create a `Queue` interface
    - The concrete implementations then 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 CircularArrayQueue<E> implements Queue<E> {
  public void add(E element) { ... }
  public E remove() { ... }
  public int size() { ... }
  ...
}
```

- Use the `interface` to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq = new CircularArrayQueue<Date>();
stringq = new LinkedListQueue<String>();
```

- The benefits of ***indirection***
  - To use a different implementation for `dateq`
  - Only need to update the instantiation

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? Re-imburse the 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*

Collection

Type	📒 Lecture
Date	@February 4, 2022
Lecture #	2
Lecture URL	https://youtu.be/-EwtNZTsOVw
Notion URL	https://21f1003586.notion.site/Collection-e4cdb6c129cc46c6ad9bc49269b06c72
Week #	6

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`
    - `...`
- We'd often choose the one we need, but changing a choice usually required multiple updates
- Instead, we'd 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
- `add()` — add to the collection
- `iterator()` — get an object that implements `Iterator` interface
- Use the 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 the 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 the element
}
```

- Java later added the “for each” loop
  - It implicitly creates an iterator and runs through it

```
Collection<String> cstr = new ...;
for(String element : cstr) {
  // do something with element
}
```

- Generic functions to operate on collections

```
public static <E> boolean contains(Collection<E> c, Object obj) {
  for(E element : c) {
    if(element.equals(obj)) {
      return true;
    }
  }
  return false;
}
```

- *Explained later: How does the following line work*
  - `if(element.equals(obj))`

Removing elements

- Iterator also has a `remove()` method

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
  ...
}
```

- It removes the element that was last accessed by `next()`
  - When the iterator starts, the next points to the first element
    - This means, the iterator is standing over, basically, nothing
  - When we call the `next()` for the first time ...
    - The iterator passes the first element and stands right before the second element pointing to it
    - Now, if we call the `remove()` method, it will delete the first element as it was the last accessed element by `next()`
    - Now, we CANNOT call `remove()` again immediately after the previous deletion, because the iterator has not passed any new element and is still pointing to the 2nd element
      - This will cause an error

```
Collection<String> cstr = new  ...;
Iterator<String>iter = cstr.iterator();
while(iter.hasNext()) {
  String element = iter.next();
  // Delete the element if it has some property
  if(property(element)) {
    element.remove();
  }
}
```

- To remove consecutive elements, we must interleave a `next()`

```
// The following code will cause an error, described above
iter.remove();
iter.remove();
```

```
// The following code is fine
iter.remove();
```

```
iter.next();
iter.remove();
```

- To remove the first element, we need to access it first

```
// To remove the first element in cstr
iter.next();
iter.remove();
```

The `Collection` interface — contd.

- How does the following 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;
}
```

- Actually, `Collection` defines a much larger set of abstract 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);
}
```

  - `addAll(from)` adds elements from a compatible collection
  - `removeAll(c)` removes elements present in `c`
  - This `remove()` is quite different from the one present in `Iterator`
- So, in order to implement the `Collection` interface, we need to implement all these methods

The `AbstractCollection` class

- To implement Collection, we need to implement all the methods required (mentioned above)
- So a correct solution — provide default implementations in the interface itself
- So, `AbstractCollection` is a abstract class which 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;
  }
  ...
}
```

- So, the concrete class now extends the `AbstractCollection`
  - We need to define `iterator()` based on the internal representation
  - We can choose to override `contains()`

Summary

- The `Collection` interface captures abstract properties of collections
  - Add an element, create an iterator, ...
- We 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 one of them
- `AbstractCollection` provides default implementations to many functions required by `Collection`
- Concrete implementations of collections extend `AbstractCollection`

Concrete Collections

Type	📒 Lecture
Date	@February 5, 2022
Lecture #	3
Lecture URL	https://youtu.be/XPzVy2DyO6E
Notion URL	https://21f1003586.notion.site/Concrete-Collections-949ea83681704e34a60baed60ec6dc1f
Week #	6

Built-in data types

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not the key-value data structures like dictionaries
- Collections can be further organized based on additional properties like ...
  - Are the elements ordered?
  - Are duplicates allowed

    - Are there constraints on how elements are added, removed?
- These are captured by interfaces that extend `Collection`
    - Interface `List` for ordered collections
    - Interface `Set` for collection 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
- 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 the previous element
    - `boolean hasPrevious()` checks whether it is legal to go backwards

```
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 the ordered collections
    - In an array, we can compute the location of element at index `i`
    - In a linked list, we must start at the beginning and traverse `i` links
- Tagging interface `RandomAccess`
    - It tells us whether a `List` supports random access or not
    - We 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

- `AbstractCollection` is a usable version of `Collection`
- Correspondingly, `AbstractList` extends `AbstractCollection`
  - Inherits default implementations
- `AbstractSequentialList` extends `AbstractList`
  - A further sub-class to distinguish lists without random access
- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - Internally, the usual flexible linked list
  - It is efficient to add and remove elements at arbitrary positions
- Concrete generic class `ArrayList<E>` extends `AbstractList`
  - It has flexible size array
  - It supports random access

Using concrete list classes

- Concrete generic class `LinkedList<E>` extends `AbstractSequentialList`
  - No random access
  - But random access methods of `AbstractList` are still available
  - The following loop with execute a fresh scan from the 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()` are available
  - `add()` from `Collection` appends to the end of the list
  - `add()` from `ListIterator` inserts a value before the current position of the iterator
- In `Collection`, `add()` returns boolean — which represents “did the `add` update the collection?”
  - `add()` may not update a set, always work for lists
- `add()` is `ListIterator` returns `void`

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
- Use `Set` to constraint values to satisfy additional constraints
- `Set` implementations are typically designed to allow efficient membership tests
- Ordered collections loop throught a sequence to find an element
- Instead, map the value to its position
  - Hash function
- Or arrange values in a 2D 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
- Checking membership is fast — check if `v` is at position `h(v)`
- It is unordered, but supports `iterator()`
    - Scan the 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
- `Queue` interface supports the following
    - `boolean add(E element);`
    - `E remove();`
        - If the queue is 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 sub-interfaces
    - `List`, `Set`, `Queue`
- `List` allows random access, more functional `ListIterator`
- `Set` constraints 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

Type	📒 Lecture
Date	@February 5, 2022
Lecture #	4
Lecture URL	https://youtu.be/fWXf5aI3Cr0
Notion URL	https://21f1003586.notion.site/Maps-6569131529294cf48862b4d54d044f0c
Week #	6

Maps

- The `Collection` interface abstracts properties of grouped data
  - Arrays, lists, sets, ...
  - But not key-value structures like dictionaries

```
public interface Map<K,V> {
  V get(Object key);
  V put(K key, V value);

  boolean containsKey(Object key);
  boolean containsKey(Object value);
```

```
  ...
}
```

- 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 the value for the key `k`
  - `put(k,v)` updates the value for the key `k`
- The keys form a set
  - Only one entry per key-set
  - Assigning a fresh value to existing key will 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
- 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 the key `bat` is not present
- Now, we can update the entry for the key `bat` as follows

  `scores.put(bat, scores.getOrDefault(bat, 0) + newscore);`
- 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 there is no such key `bat`
  - Otherwise, combine the current value with `newscore` using `Integer::sum`

Extracting keys and values

- Methods to extract the keys and values

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K,V>> entrySet();
```

- Keys form a `Set` while values from an arbitrary `Collection`
- Key-value pairs form a set over a special type `Map.Entry`
- Java calls these views
- We can now iterate over a `Map`

```
Set<String> keys = strmap.keySet();
for(String key : keys) {
  // do something with the key
}
```

- Use `entrySet()` to operate on the 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 implementation of Map

`HashMap`

- It works similar to `HashSet`
- It uses a hash table to store the keys and values

- There is no fixed order over which keys returned by `keySet()`

`TreeMap`

- It is similar to `TreeSet`
- It uses a balanced search tree to store the keys and values
- An iterator over `keySet()` will process the keys in a sorted order

`LinkedHashMap`

- It remembers the order in which keys were inserted
- Hash table entries are also connected as a (doubly) linked list
- The iterators over both `keySet()` and `value()` enumerate in the order of insertion
- We can also use access order
  - Each `get()` and `put()` moves key-value pair to the end of the list
  - Process the entries in least recently used order — scheduling, caching
- There is a similar `LinkedHashSet`

Summary

- The `Map` interface captures properties of key-value stores
  - `get(), put(), containsKey(), containsValue(), ...`
- Parameterized by two types 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()`
- We use these “views” to iterate over all the key-value pairs in the map
- Concrete implementations → `HashMap, TreeMap, LinkedHashMap`

Dealing with Errors

Type	📒 Lecture
Date	@February 9, 2022
Lecture #	1
Lecture URL	https://youtu.be/5t0UPIdDYo4
Notion URL	https://21f1003586.notion.site/Dealing-with-Errors-8fc0f8b9d3a34990b6bce6edae5a4495
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 the hash table
        - Refer to a variable that is `null`
        - Divide by zero
        - `...`
- Signalling errors
    - Return an invalid value
        - `-1` at the end of the file
        - `null`
    - What if there is no obvious invalid value?

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 the information about the error from the exception object
    - Graceful interruption rather than a program crash
- Or pass 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`
- `Error` — relatively rare, “not the programmer’s fault”
    - Internal errors, resource limitations within Java runtime
    - No realistic corrective action possible, notify the 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 mode by code
        - To contrast, `Error` and `RunTimeException` are called unchecked exceptions

Exceptions in Java

Type	📒 Lecture
Date	@February 9, 2022
Lecture #	2
Lecture URL	https://youtu.be/hsVEjkYKjpg
Notion URL	https://21f1003586.notion.site/Exceptions-in-Java-ef362d53fecf4772b79aa3aa15da872f
Week #	7

Catching and Handling Exceptions

- `try-catch`
  - Enclose the code that may generate exception in a `try` block
  - Exception handler in a `catch` block
  - It is a similar to python

```
try {
  ...
```

```
  call a function that may throw an exception
  ...
} catch(ExceptionType e) {
  ...
  examine e and handle it
  ...
}
```

- 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

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hierarchy
  - `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 the missing files
  ...
} catch(UnknownHostException e) {
  ...
  handle unknown hosts
  ...
} catch(IOException e) {
  ...
  handle all other I/O issues
  ...
}
```

Generating Exception

- 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

Notify 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”
- Created 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 the caller know that `readDate()` 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;
}
```

- Can throw multiple types of exceptions

```
String readFile(String filename) throw FileNotFoundException, EOFException {
  ...
}
```

- Can throw any subtype of declared exception type

  `String readFile(String filename) throw IOException { ... }`

  - Can throw `FileNotFoundException`, `EOFException`, both subclasses of `IOException`

- Method declares the excpetions 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

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 the super class
```

```
    error_value = i; // Constructor to set message
  }

  public int report_error_value() {
    return error_value;
  }
}
```

- Throw this from `LinearList`
  - Note that `add` advertises the fact that it throws a `NegativeException`

```
public class NegativeException extends Exception {
  ...
}

public class LinearList {
  ...
  public void add(int i) throws NegativeException {
    ...
    if(i < 0) {
      throw new NegativeException("Negative input", i);
    }
    ...
  }
}
```

More on catching exceptions

- We can extract information about the exception

```
try {
  ...
  call a function that may throw an exception
  ...
} catch(ExceptionType e) {
  ...
  String errormsg = e.getMessage();
  ...
}
```

- Chaining Exceptions
  - Process and throw a new exception from `catch`

```
try {
  ...
  access database
  ...
```

```
} catch(SQLException e) {
  String errormsg = "database error" + r.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`
  - `initCause()`

```
try {
  ...
  access database
  ...
} catch(SQLException e) {
  String errormsg = "database error" + r.getMessage();
  ServletException newe = new ServletException(errormsg);
  newe.initCause(e);
  throw newe;
  ...
}
```

- 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
- 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 it for cleanup
  ...
}
```

- 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

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	3
Lecture URL	https://youtu.be/U_rxCLyJHgw
Notion URL	https://21f1003586.notion.site/Packages-f5284e2ccd894903b8d3b0cf07dae6c0
Week #	7

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, we cannot write ...

  `import java.*`

Creating and naming packages

- We 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 the same anonymous 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
- Can also restrict visibility w.r.t. inheritance hierarchy
  - `protected` means visible within the sub-tree, so all sub-classes
  - Normally, a sub-class cannot expand visibility of a function
  - However, `protected` can be made `public`

Assertions

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	4
Lecture URL	https://youtu.be/hKl3tCQdfTI
Notion URL	https://21f1003586.notion.site/Assertions-ef6b535c2bba4aaf8ff671f548df705a
Week #	7

Documenting and checking assumptions

- Functions may have constraints on the parameters

```
public static double myfn(double x) {
  // Assume x >= 0
  ...
}
```

- 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");
  }
}
```

- What if `myfn` is only used internally by our own code
  - Flag errors during development, debugging
  - But diagnostic code should not trigger at runtime
  - Performance, and other considerations
- Instead, “assert” the property you assume to hold

```
public static double myfn(double x) {
  asset x >= 0;
}
```

Assertions

- If assertion fails, the code throws `AssertionError`
- This should not be caught
  - Abort and print the diagnostic information (stack trace)
- We can also provide additional information to be printed with the diagnostic message

```
public static double myfn(double x) {
  asset x >= 0 : x;
}
```

Enabling and Disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require re-compilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- We can use `-ea` as abbreviation for `-enableassertions`

- We 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
  ```

- We can combine the two

  ```
  java -ea in.ac.iitm.onlinedegree -da:MyClass MyCode
  ```

- Separate switch to enable assertions for system class

  ```
  java -enablesystemassertions MyCode
  ```

  ```
  java -esa MyCode
  ```

Summary

- Assertion checks are supposed to flag fatal unrecoverable errors
  - We don't `catch` them
- If we need to flag the error and take corrective actions, we instead use exceptions
- It is turned on only during development and testing
  - It is not checked at runtime after deployment

Logging

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	5
Lecture URL	https://youtu.be/fkGfOzVC8zI
Notion URL	https://21f1003586.notion.site/Logging-dd33ff3f3c5f4a27af0ca313258e1cd2
Week #	7

Diagnostic messages

- It is rather typical to generate messages within code for diagnosis
- Naive approach is to use the print statements
  - The 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 config file

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

  ```
  Logging.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
- By default, first three levels are logged
- Can set a different level

  ```
  logger.setLevel(Level.FINE);
  ```

- Turn on all the levels, or turn off all logging

  ```
  logger.setLevel(Level.ALL);
  ```

  ```
  logger.setLevel(Level.OFF);
  ```

- Can also change logging properties through a config file

Summary

- Logging gives us more flexibility and control over tracking diagnostic messages than simple print statments
- We can define a hierarchy of loggers
- Seven level of messages — control which levels are printed
- Control logging from within code or through enxternal config file

Cloning

Type	📒 Lecture
Date	@February 13, 2022
Lecture #	1
Lecture URL	https://youtu.be/aJxPvy1poaU
Notion URL	https://21f1003586.notion.site/Cloning-9242921d53284a55a78a90c339dd8230
Week #	8

Copying an object

- Normal assignment creates two references of the same object
  - Basically, two different variables point to the same object in memory
  - Updating via any name will update the same object
- But, what if we want two separate but identical objects?

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

```
    public Employee(String name, String salary) {
        this.name = name;
        this.salary = salary;
    }

    public void setName(String name) {
        this.name = name;
    }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e1 = e1;
e2.setName("Eknath");   // This updates e1 as well
```

  - `e2` should be initialized to a disjoint copy of `e1`

- How does one make a faithful copy?

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 the private instance variables
  - So, it 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 name, String salary) {
        this.name = name;
        this.salary = salary;
    }

    public void setName(String name) {
        this.name = name;
    }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e1 = e1.clone();
e2.setName("Eknath");   // This does NOT update e1
```

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 name) {
        this.name = name;
    }

    public void setbday(int dd, int mm, int yy) {
        birthday.update(dd,mm,yy);
    }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e1 = e1.clone();
e2.setName("Eknath");   // This does NOT update e1.name
e2.setbday(16,4,1996);  // This CHANGES e1.bday
```

- 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 `e2.birthday`
- Bitwise copy is a ***shallow copy***
  - Nested mutable reference are copied verbatim

Deep copy

- Deep copy recursively clones the nested objects

```
public class Employee {
    private String name;
    private double salary;
    private Date birthday;
    ...
    public void setName(String name) { ... }

    public void setbday(...) { ... }

    public Employee clone() {
        Employee newEmp = (Employee) super.clone();
        Date newBday = birthday.clone();
```

```
        newEmp.birthday = newBday;
        return newEmp;
    }
}
```

- 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

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 name) { ... }

    public void setbday(...) { ... }

    public Employee clone() { ... }
}

public class Manager extends Employee {
    private Date promodate;
    ...
}
```

- Manager inherits deep copy `clone()` from `Employee`
- However, `Employee.clone()` does know that it has to deep copy `promodate`
- Cloning is subtle, so Java puts in some restrictions

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;

    public void setName(String name) { ... }

    public void setbday(...) { ... }
}

...
Employee e1 = new Employee("Dhruv", 21500.0);
Employee e1 = e1.clone();
e2.setName("Eknath");   // This does NOT update e1
```

- `clone()` in `Object` is `protected`
  - Only `Employee` objects can `clone()`
- Re-define `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 name) { ... }

    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
  - However, shallo copy aliases nested objects
- Deep copy solves this 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` is needed
- `clone()` in `Object` throws `CloneNotSupportedException`, which must be taken into account when overriding

Type Inference

Type	📒 Lecture
Date	@February 13, 2022
Lecture #	2
Lecture URL	https://youtu.be/o1cQO3iB2Ok
Notion URL	https://21f1003586.notion.site/Type-Inference-8c7d6d6f90c543498b68c1486ebb73d3
Week #	8

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 sub-typing
```

- An alternative approach is to do type inference
- Derive type information from context
  - For instance, `s` should be a `String`

    `s = "Hello, " + "world";`
- Propagate type information: now `t` is also `String`

  `t = s + 5;`

Type inference

- Assume the 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;`

```
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?
  ...
}
```

- If `x.bonus()` is legal, `x` must be `Manager` rather than `Employee`
- Keep track of and validate type obligations

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

Type inference in Java

- Java allows limited type inference
    - Only for local variables in a function
    - 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 = 3.14f; // float
var e = new Manager(...); // Manager
```

Summary

- Automatic type inference can avoid redundancy in declarations

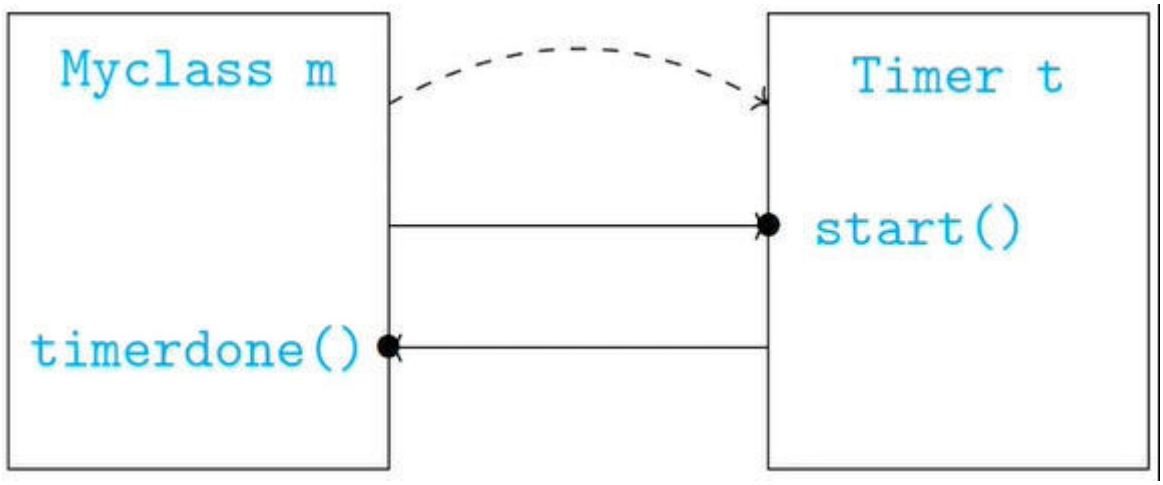
  `Manager m = new Manager(...);`
- Assuming the program is type-safe, derive most general types compatible with the code
    - Compiler can infer type from expressions used to assign values
    - Inferred type information can be propagated

- 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

Type	📒 Lecture
Date	@February 13, 2022
Lecture #	3
Lecture URL	https://youtu.be/4nrbgx0e4RA
Notion URL	https://21f1003586.notion.site/Higher-Order-Functions-ee9b794297e943719bb1f9f0fca7e2f9
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
- `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`
- `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 Comparable<String> {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
}

String[] strarr = new ...;
Arrays.sort(srtarr, StringCompare);
```

Functional interfaces

- Interfaces that define a single functions are called functional interfaces
  - `Comparator, TimerOwner`
- Is there a way to pass the required functions?

```
public interface Comparator<T> {
  public abstract int compare(T o1, T o2);
}

public interface Timerowner {
  public abstract void timerdone();
}
```

- 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

```
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
- 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());
```

- 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;
}
```

Passed 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's 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)
```

- `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

Type	📒 Lecture
Date	@February 13, 2022
Lecture #	4
Lecture URL	https://youtu.be/xs94mCRwqVQ
Notion URL	https://21f1003586.notion.site/Streams-da5ae640124a45ca9892e8584a5e1cdd
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
- 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
  - Focus on what to compute, rather than how to
- 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
- 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

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 a 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 arguments
    - Filter out the long words

```
List<String> wordlist = ...;
Stream<String> longwords = wordlist.stream().filter(w -> w.length() > 10);
```

- `map()` applies a function to each element in the stream
  - Extract the first letter of each long word

```
List<String> wordlist = ...;
Stream<String> startlongwords =
  wordlist.stream()
  .filter(w -> w.length() > 10)
  .map(s -> s.substring(0,1));
```

- 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> startlongwords =
  wordlist.stream()
  .filter(w -> w.length() > 10)
  .map(s -> explode(s));
```

- flatMap() flattens (collapses) nested list into a single stream

```
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);
```

- Skip `n` elements — `skip(n)`
  - Discard the first 10 random numbers

```
Stream<Double> randomds =
  Stream.generate(Math::random).skip(10);
```

- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5

```
Stream<Double> randomds =
  Stream.generate(Math::random)
  .takeWhile(n -> n >= 0.5);
```


- Stop when element matches a criterion — `takeWhile()`
  - Stop with number smaller than 0.5

```
Stream<Double> randomds =
  Stream.generate(Math::random)
  .takeWhile(n -> n >= 0.5);
```


- Start after elements matches a criterion — `dropWhile()`
  - Start after number larger than 0.05

```
Stream<Double> randomds =
  Stream.generate(Math::random)
  .dropWhile(n -> n <= 0.05);
```

- Can also combine streams, extract distinct elements, sort, ...

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 option type — later

- First element — `findFirst()`
  - First random number above 0.999
  - Again, deal with empty stream

```
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