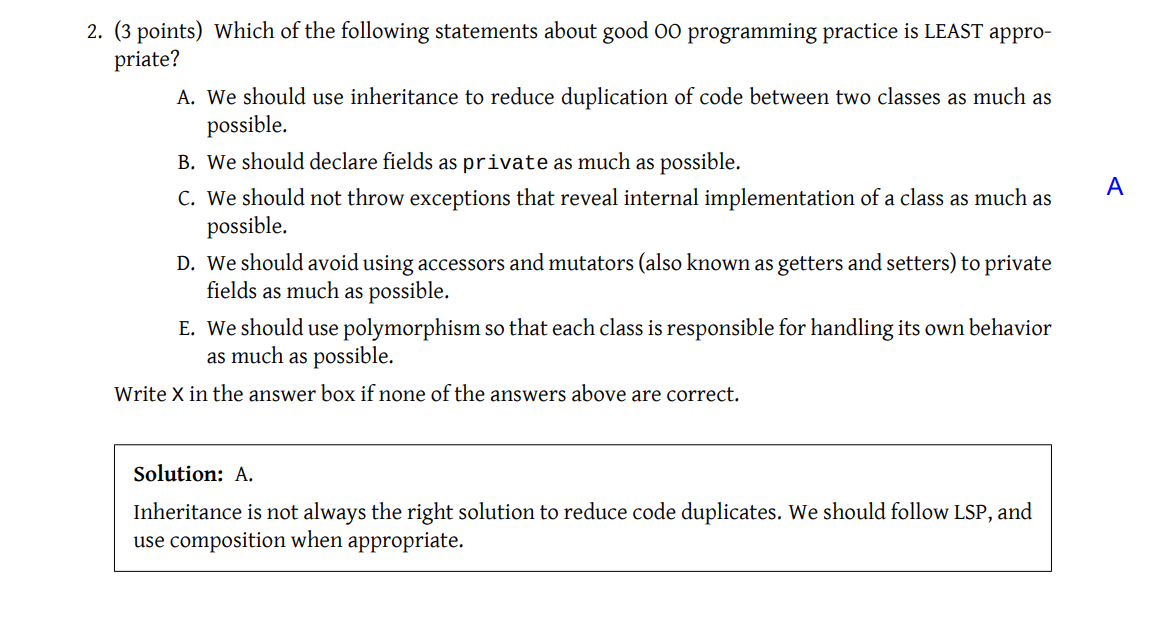
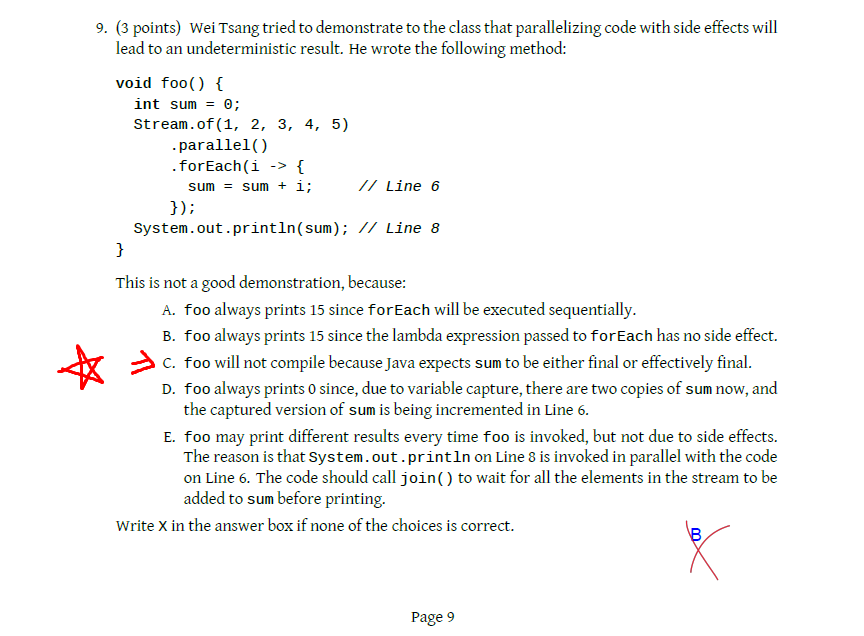
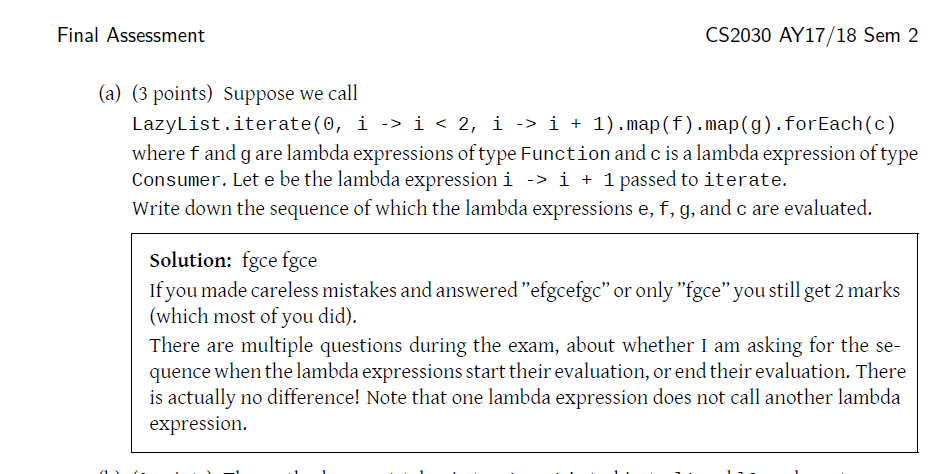
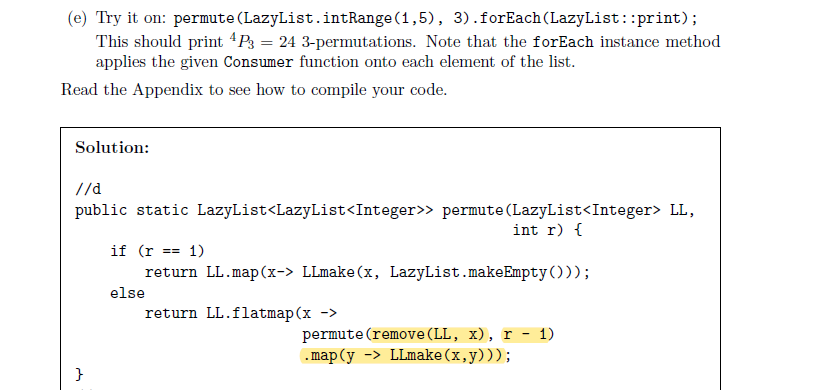
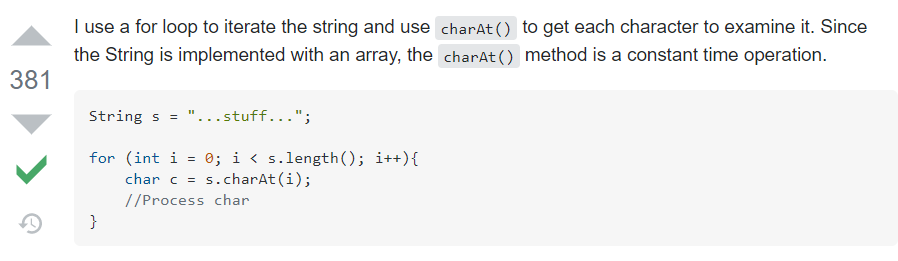
# Trickier Concepts (For Revision)

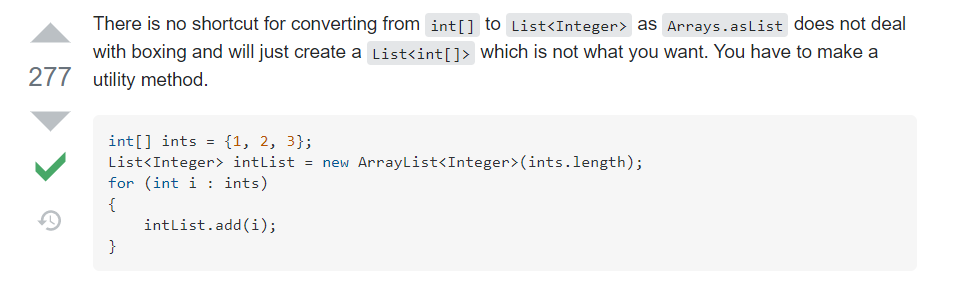


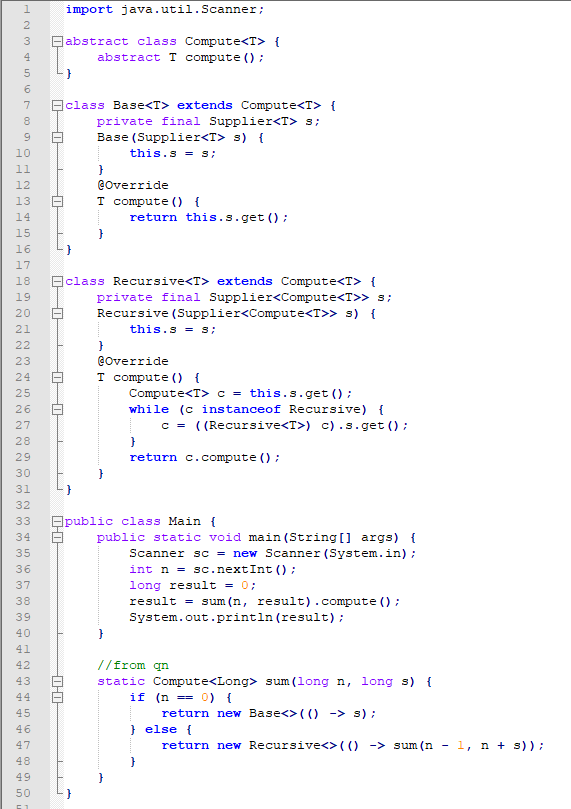
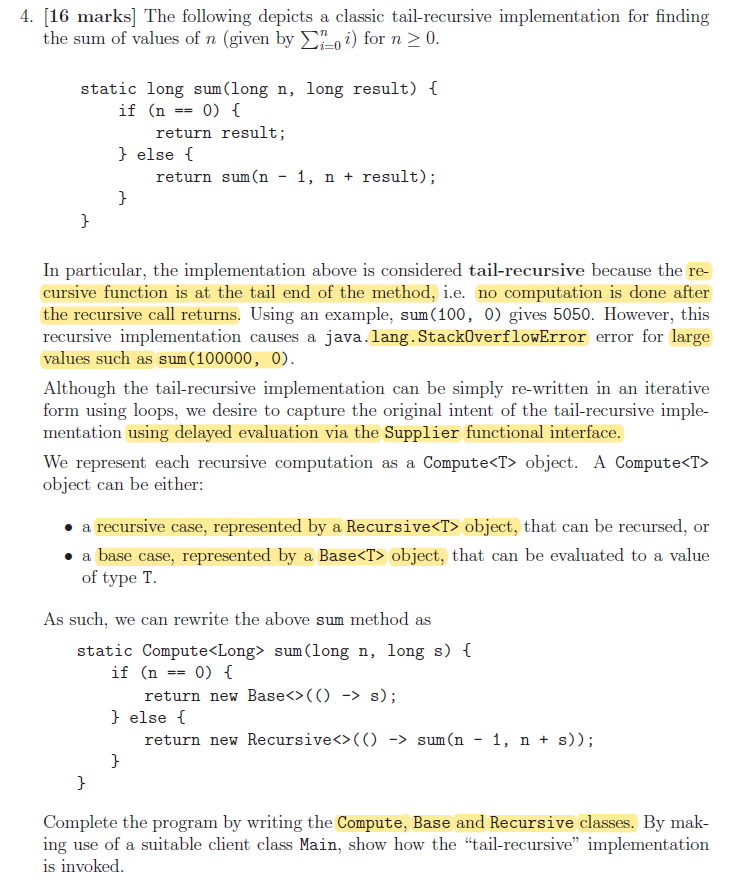


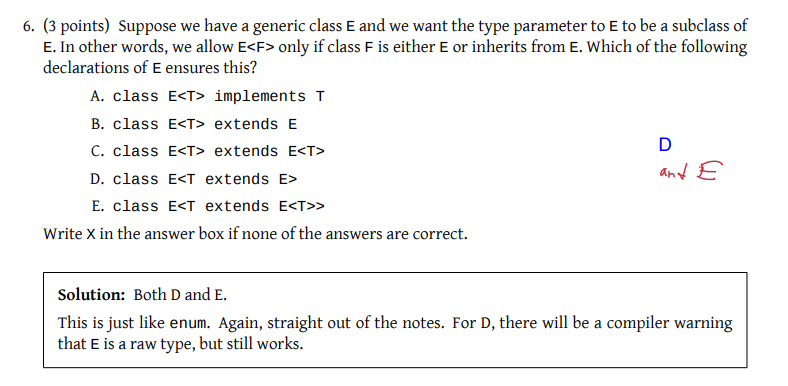


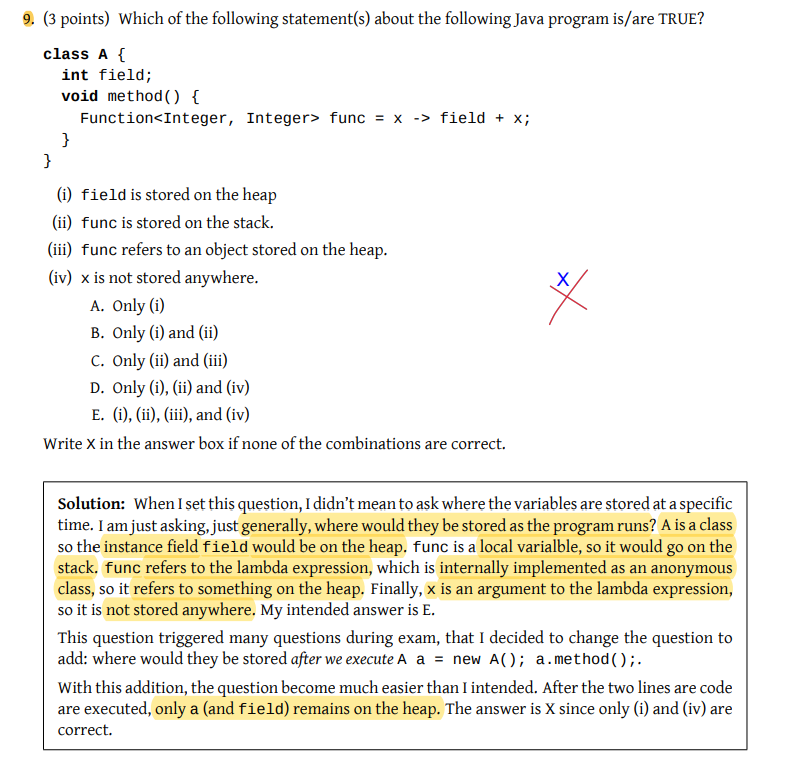


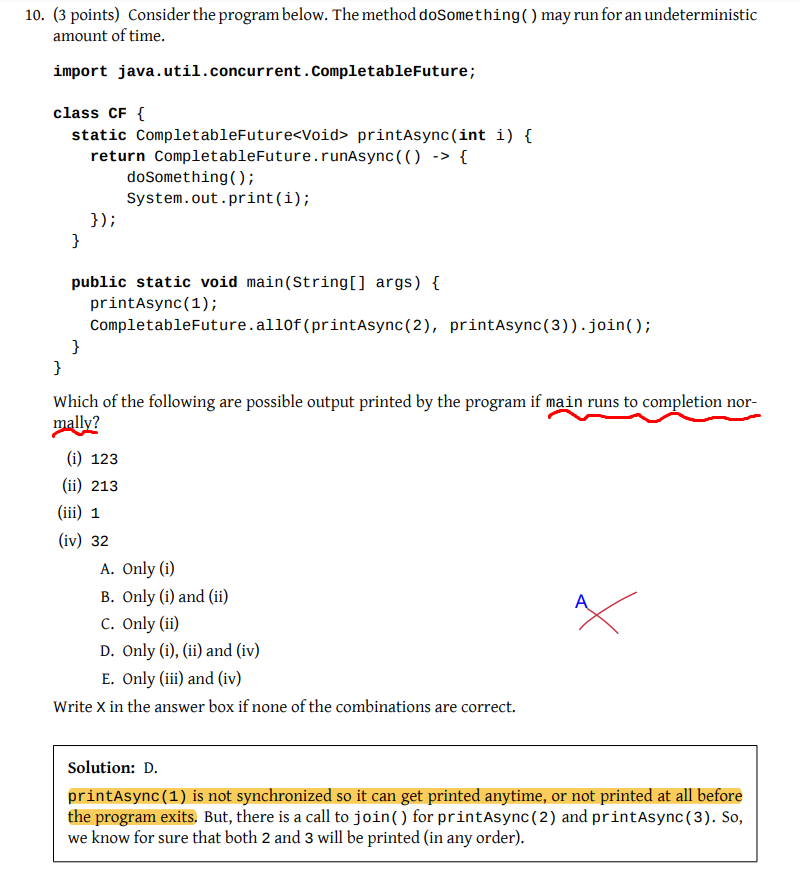


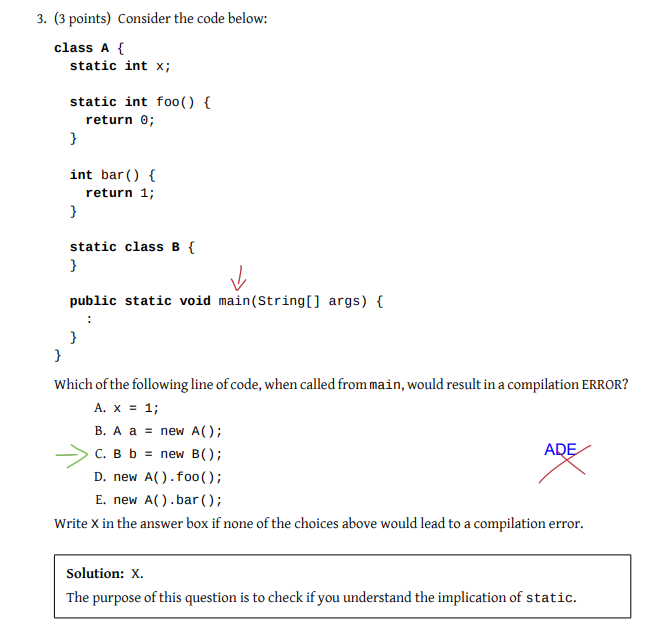
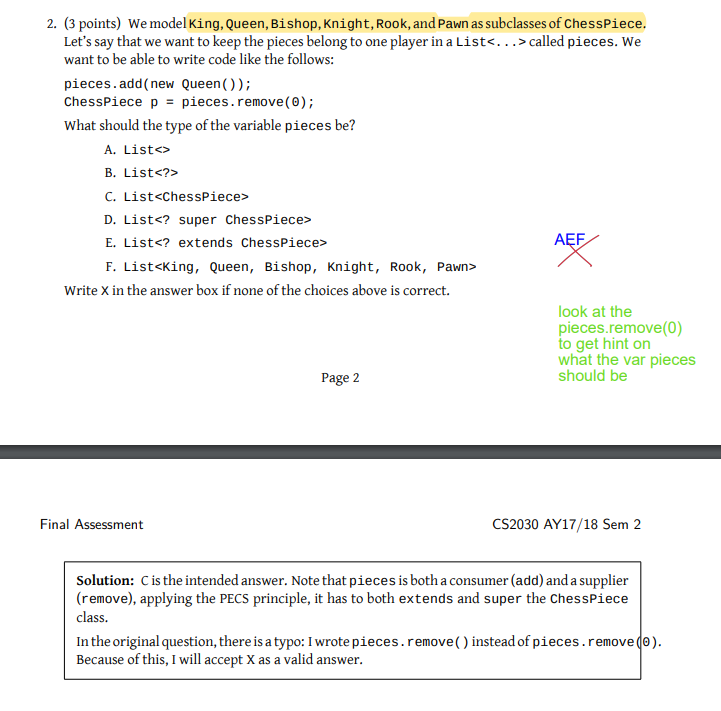


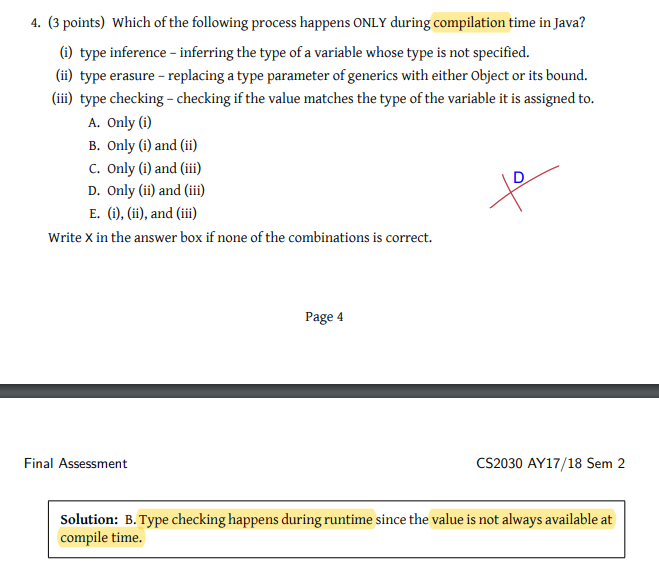


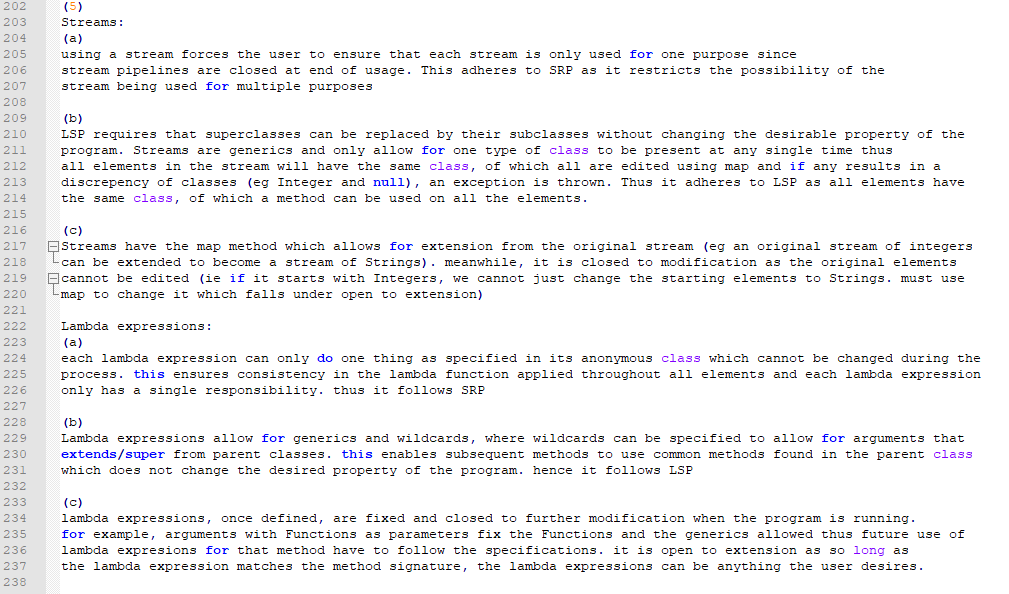






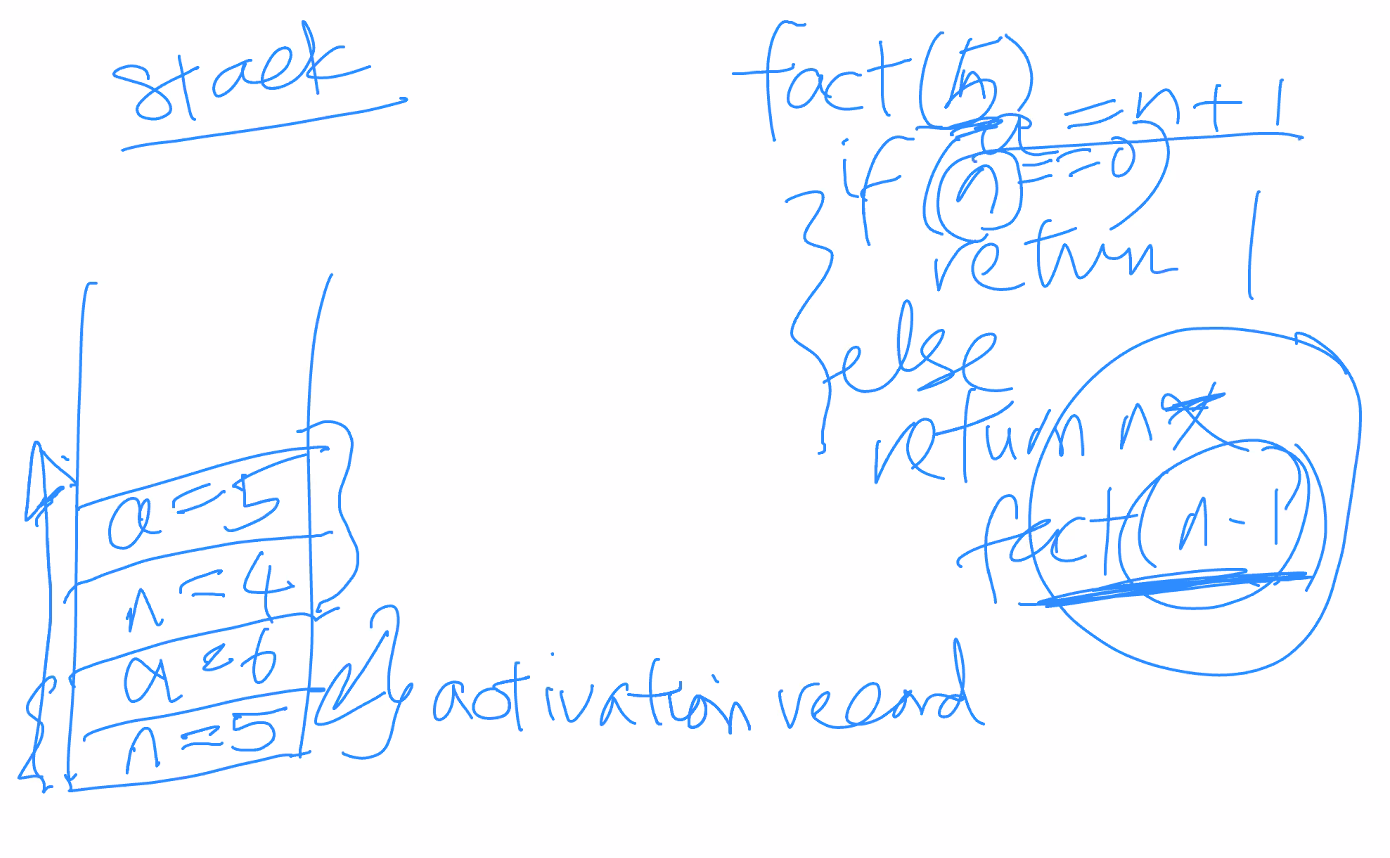
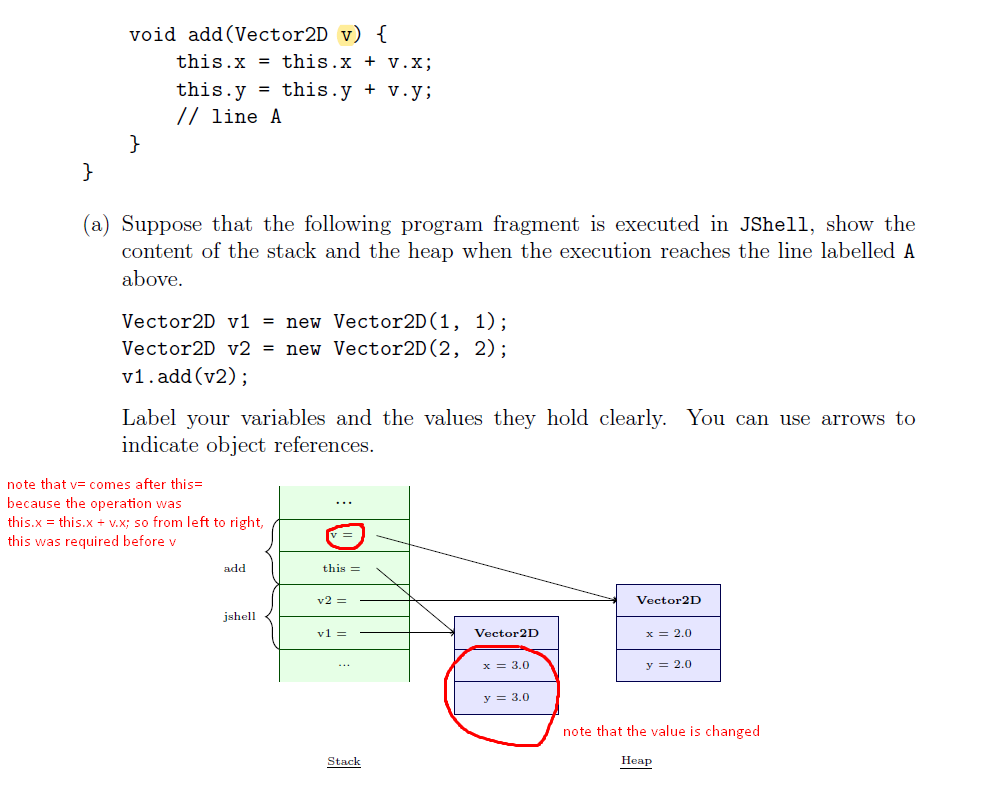






# Java Memory Model

## Stack

* Local variables stored here (eg int 5 inside some function)
* LIFO storing activation records of method/function calls’s arguments
* Indicate with arrow whether is it top to bot or bot to top for activation record/stack frame (same thing)
* 
* 
* When the function ends, the function is cleared
* Recursive calls get arguments pushed onto the stack too in LIFO manner
* Return value is NOT called to the stack. Just returned to the caller. Then pops the activation records in recurive manner
* Primitive types are stored on stack
* ALL VARIABLES are on the stack. It’s data type is dependent. Eg

Student s = new Student();

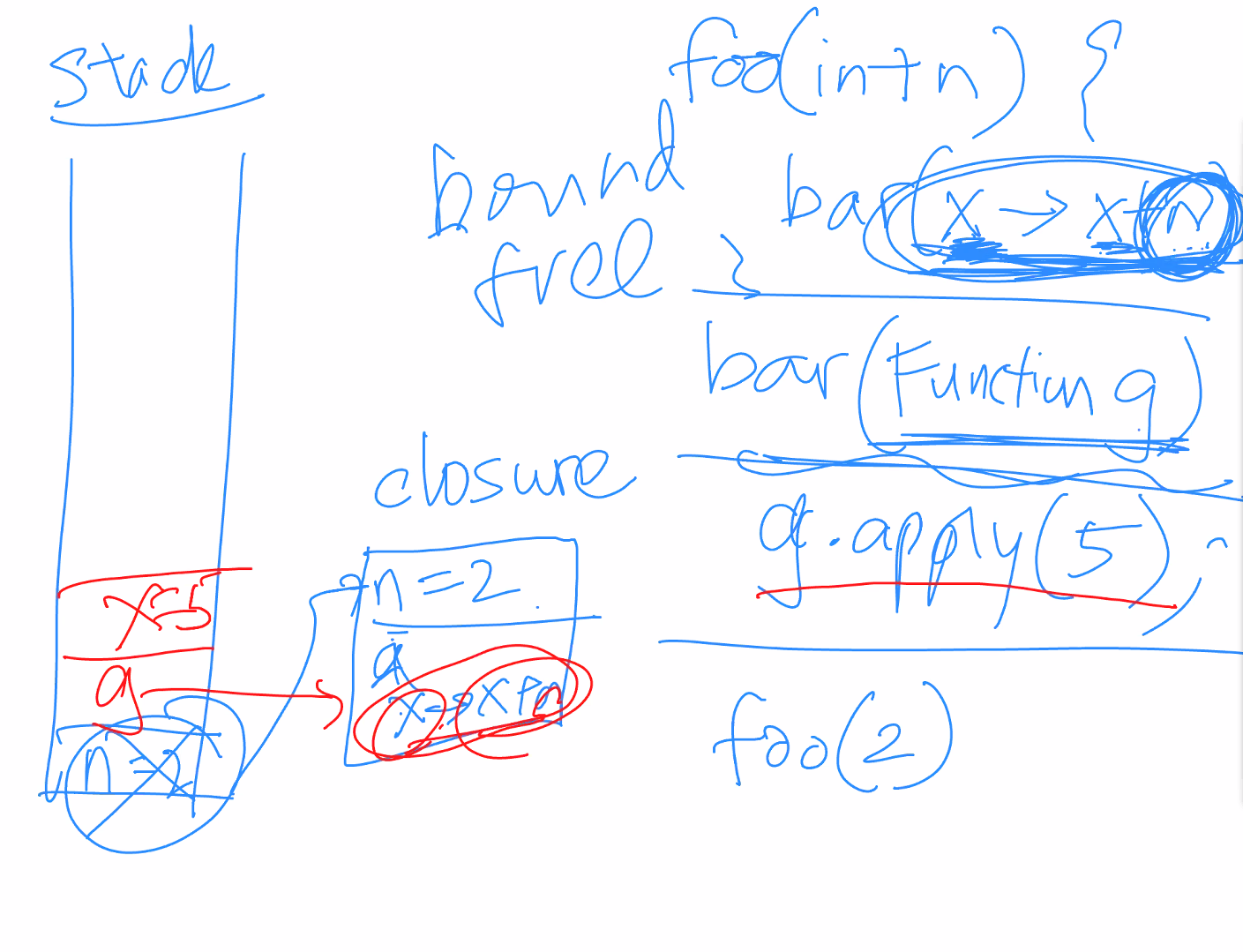
s is stored on the stack, new Student() is stored on the heap

* Method ends => stack is cleared. Heap may be cleared or may not, depending on whether there are more references to the object.

## Heap

* Storing java objects when new XXX is invoked
* Reference types are stored here
* Garbage collection (ie objects that have no more references to it). It is actually a thread going around to remove these objects.

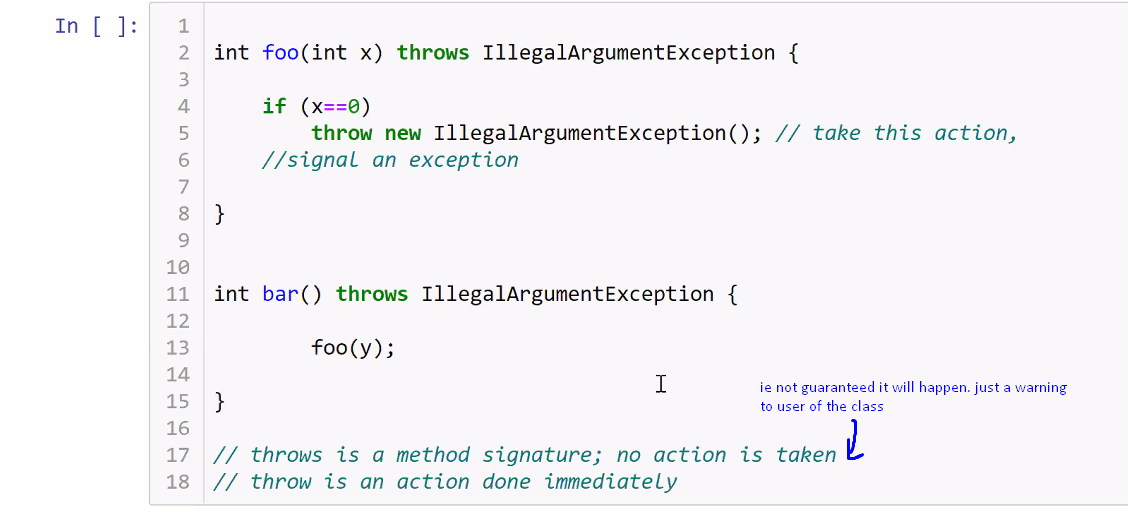
**Lambda closure (in the heap)**

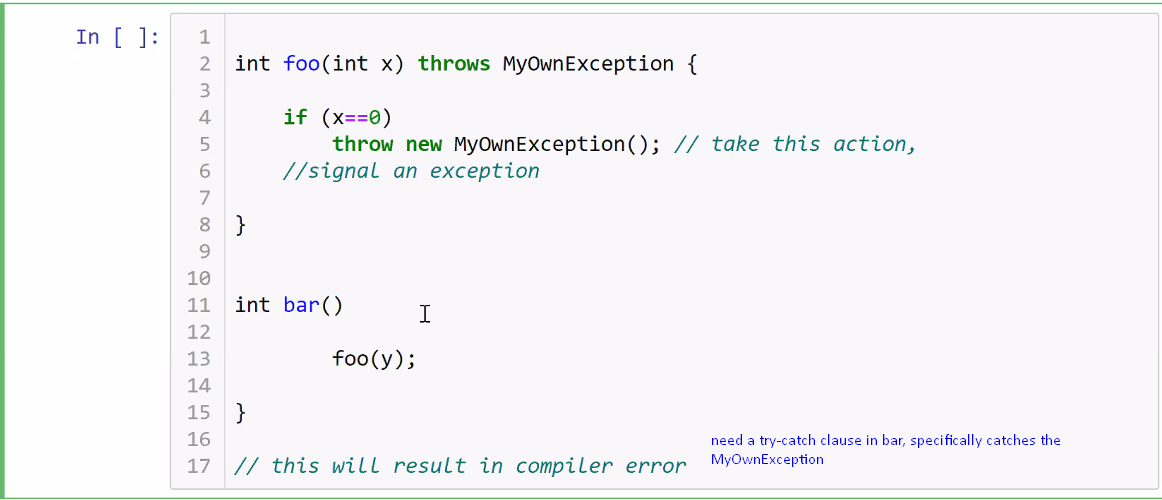
* Body of lambda is trapped here
* Free variable => variable that does not appear in the param list eg x-> x+ n; n is the free variable, x is the bound variable
* 
* Free variables will be will stay in lambda closure. Ie when foo ends, the n in the stack is removed but the value of n is captured inside the closure.

## Non-Heap

* Storing loaded classes and other meta data
* Static fields are stored here

# OOP





But runtime exception is not checked at compiletime so sometimes it works even w/o the explicit throws because they are runtime exceptions.

## OOP Principles (and some others)

**Abstract Class**

To enable polymorphism by ensuring future method calls refer to the abstract class and ensuring type-safety.

* not desirable to inherit properties from different parents
* but still appropriate to inherit functionality as specified by the methods from different parents

**Interface**

Class can only inherit from 1 Parent class (ie extends) but can implement multiple interfaces (ie implements). Eg Circle extends Shape implements Calculate, Summary, etc

* interfaces cannot be instantiated, are implicitly public
* class implementing interface must have all the overridden methods required by the interface (a lot like abstract classes).
* Interface can inherit from multiple interfaces (ie interface PrintableShape extends Printable, Shape{}). BUT consider if it violates SRP

### 01: Abstraction

* Abstraction barrier sits between the client and the implementer
* The Implementor defines the data/functional abstractions using lower-level data and processes.
* The Client only needs to access the higher-level data-type and methods and should not be exposed to the lower-level implementation
* Abstraction barrier is NOT broken if an object of type T accesses another object of type T’s instance fields (since they are both of type T).

### 02: Encapsulation

Develop a mental model that is correct, consistent, and complete

* Packaging: Think about what the method requires and put it in the respective class. Eg distanceBetween that calculates the dist between points should be placed inside the Point class rather than the Circle class. contains method checks if a point is inside a Circle, so it should be inside Circle class rather than Point class.
* Information hiding: don’t allow for getter and setter methods.

### 03: Inheritance (is-a)

Is-a relationship ie Child extends Parent.

* super keyword. super(…) accesses the parent’s constructor ie sends the child constructor’s argument inputs to the parent. super.radius accesses the parent’s radius field, super.contains() uses the parent’s contains() method.

### 04: Polymorphism

Use of Overriding methods (ie @Override) so that child method does something different as compared to the Parent.

* Type-casting to ensure that there is no compile time error.

### Tell-Don’t-Ask

Client should tell the object what to do, not ask the object for data and the client do it.

Ie Client: foo(2).calculate(); rather than Client.calculate(foo(2).getValue());

### Non-Cyclic Dependency

* Modules should not have arrows pointing in a circular loop
* To prevent cyclic dependency, use abstraction (Open-closed principle)

### Immutability

* private static **final**; private final int; etc…
* return new objects, not the same one as before

### Composition (has-a)

Has-a relationship. Ie Circle has a Point, not Circle is a Point.

* Difference from inheritance is that the classes are distinct. While they may share common methods such as calculate or getPoint etc, they are not the same so it should not be a is-a relationship. Eg getPoint in Circle class could refer to get the centre of the Circle but getPoint in Point class just returns the coordinates of the Point.

## SOLID Principles

### Single Responsibility Principle (SRP)

An object should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the object

* One method for each responsibility
* Ie do not have a getArea() method that returns both the area and the perimeter (the perimeter value should be returned in another method getPerimeter)

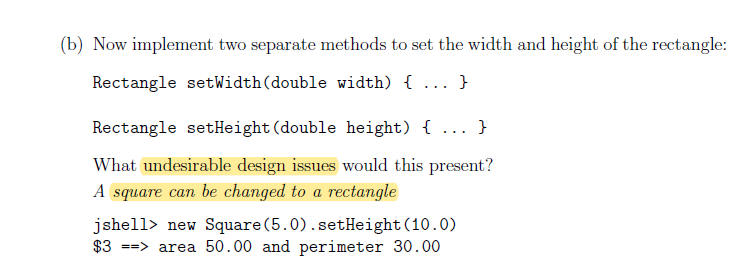
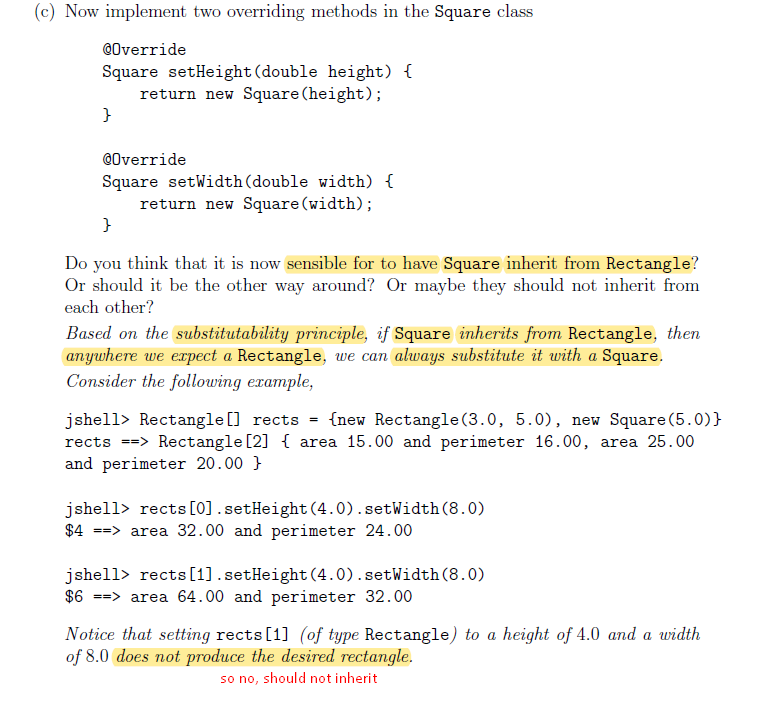
### Open-closed Principle

Software entities should be open to extension but closed to modification

* Program by Interface, not by Implementation (at least try)
* Use abstractions for dependencies (ie abstract classes or interfaces) instead of concrete classes
* Functionality is to be added by creating new classes that implements the interface or extends from the abstract class

### Liskov’s Substitutability Principle

If S is a subclass of T, then an object of type T can be replaced by that of type S without changing the desirable property of the program. Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

* ie replacing UnitCircle with Circle should work as expected which means UC.scaleBy(x) will just return a UC but Circle.scaleBy(x) must return a Circle with the params scaled by a factor of x. to solve this, use an overridden method in UnitCircle.
* return type of the overriding method cannot be more general than that of the overridden one.
* 
* 

### Interface Segregation Principle

Implementing classes need not implement unwanted methods. Many client-specific interfaces are better than one general-purpose interface.

* clients should not be forced to depend upon interface members they do not use

### Dependency Inversion Principle

Make software components explicitly declare their dependencies or collaborators through their APIs, instead of acquiring them by themselves. One should depend upon abstractions, not concretions.

* DIP is neither dependency injection (DI) nor inversion of control (IoC), although they all work great together
* Software components should not be tightly coupled to each other which hard to reuse, replace, mock and test, and results in rigid designs

## Early & Late Binding

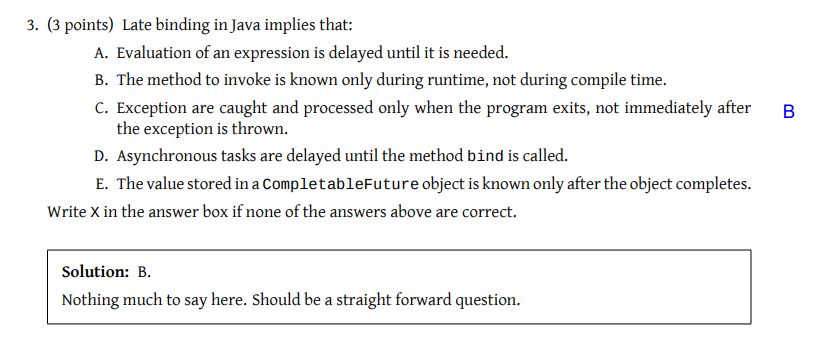
**Static (Early) Binding vs Dynamic (Late) Binding**

Static (early) binding is basically to use instanceof to check the type of object it is – whether is it the child or the parent etc. occurs during compile time.

* Method overloading uses static binding

Dynamic (late) binding is to treat all of the objects as the parent so only at runtime does the compiler determine which method, eg toString(), to use.

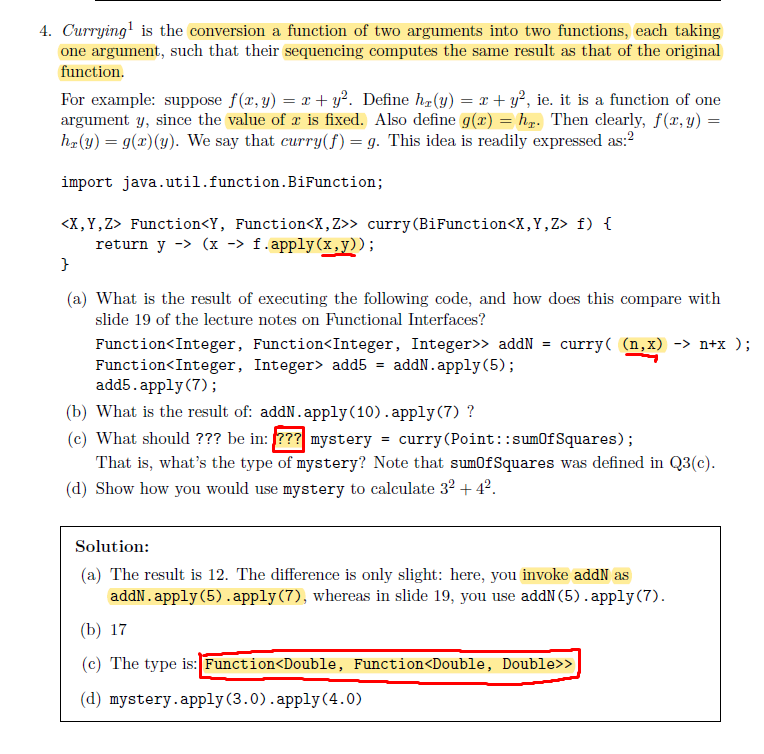
* Polymorphism + late binding allows for extensible implementations (think abstract classes) since it does not require the original code to be modified, assuming the original code references the abstract class



# Generics

“allows a type or method to operate on objects of various types while providing compile-time type safety”

Use abstraction principle to avoid defining similar classes with duplicate code. Eg Sandbox<Integer> and Sandbox<String> use the same Sandbox methods.



## Generic Typing

* Only need to replace generic typing with concrete types eg String
* Methods needs to specify return type and type involved if it returns another generic (ie .of(), .map(), .flatmap(), etc)
* Only reference types are allowed as type arguments for U. ie cannot <int>

## Autoboxing/Unboxing

Java automatically boxes primitives into their class types. Ie placing an int into a <Integer> causes the int to be boxed as an Integer. When taking the value out, it is auto-unboxed from Integer to int again.

## Variance (Based on LSP)

S <:T => S is a subtype of T. ie S extends T.

**Covariance (? extends T)**

Subtype relationship preserved in complex types eg Java arrays are covariant S[] <: T[]

**Contravariant (? super T)**

subtype relation is reversed for complex types

**Invariant (Box<String> cannot be equal to Box<Integer>)**

its neither covariant nor contravariant. e.g. Java generics are invariant

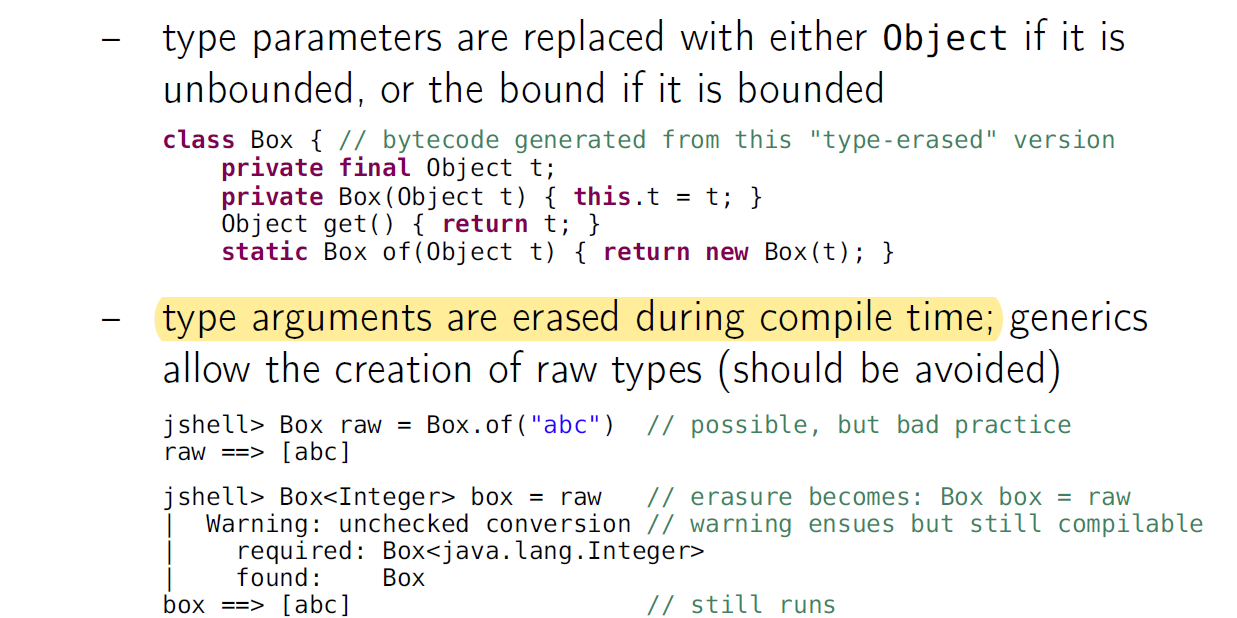
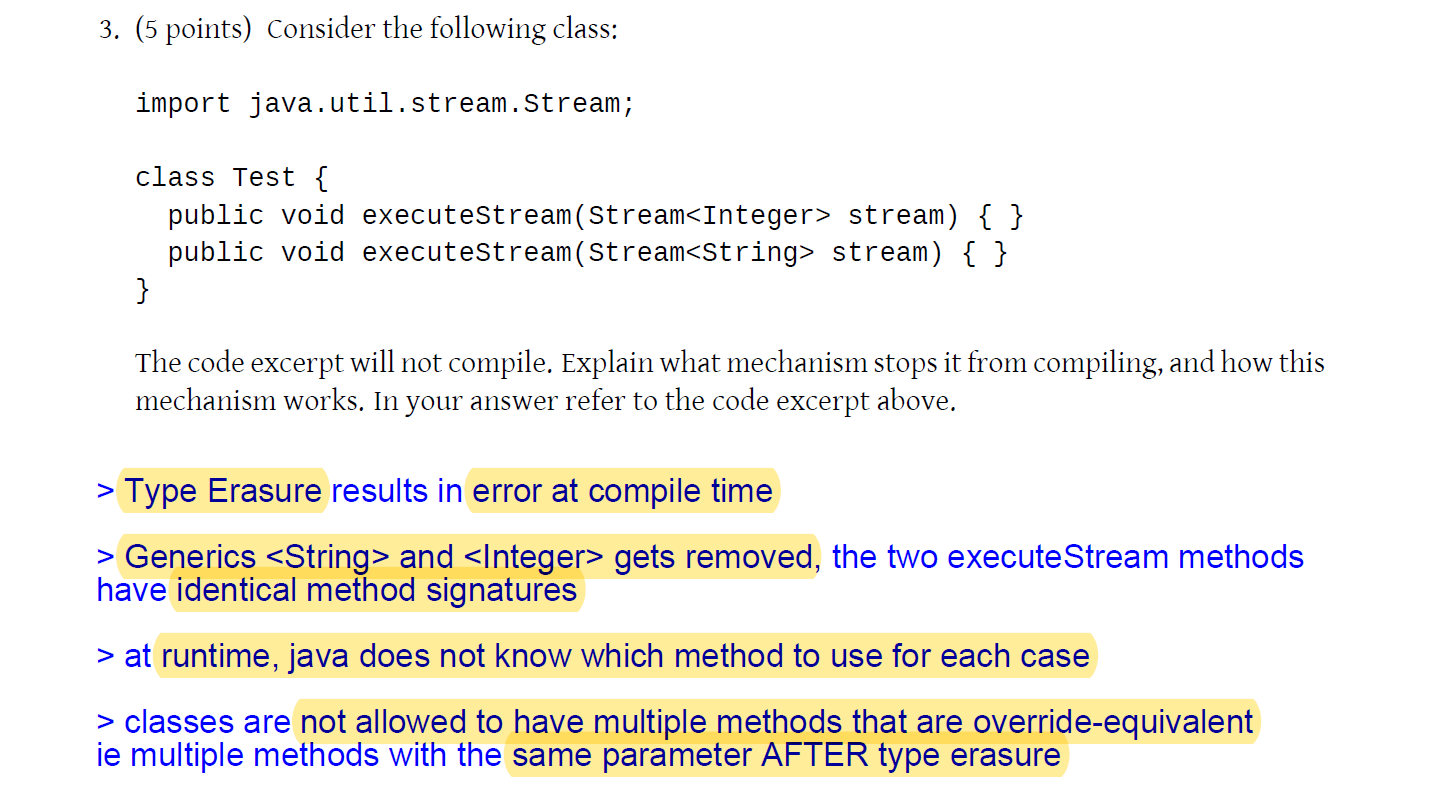
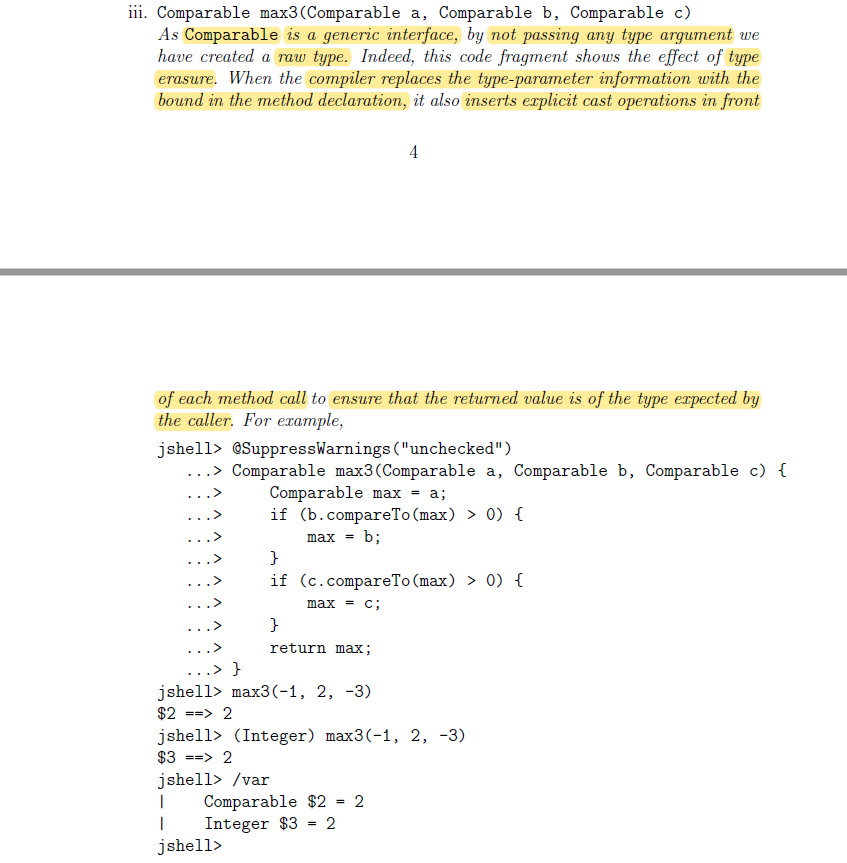
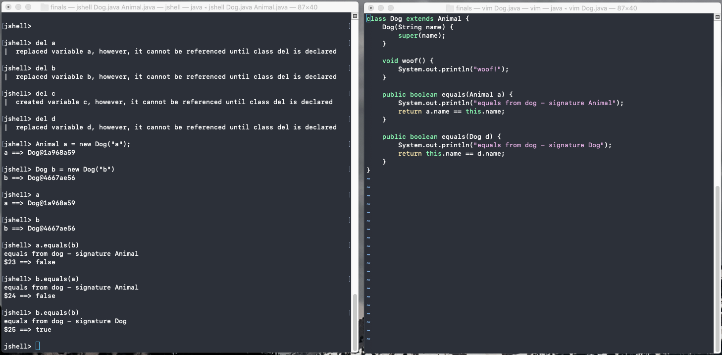
## Wildcards

<? extends T>, <? super U> etc

* Producer extends, Consumer super. Extends is to get, super is to put.
* Function<? super T, ? extends U>
* Unchecked cast warning may come up

## Type Erasure

Compiler performs type checking/inference, and generates non-generic bytecode (erasure) for backward compatibility

* type parameters are replaced with either Object if it is unbounded, or the bound if it is bounded
* 
* 
* 
* 

# Exception Handling / Static / Enum / Nested Classes

## Static

* Field is used by all objects having access to that class and shared by all objects of the class ie Class-level. Eg SelfCheckout.customerQueue
* No overriding as static methods are resolved at compile time

## Nested Classes

* Nested class is only useful for its enclosing class
* Non-static nested inner classes can access all (including private) members of the enclosing class
* Static nested inner classes can only access static members of enclosing class and top-level class cannot be made static.

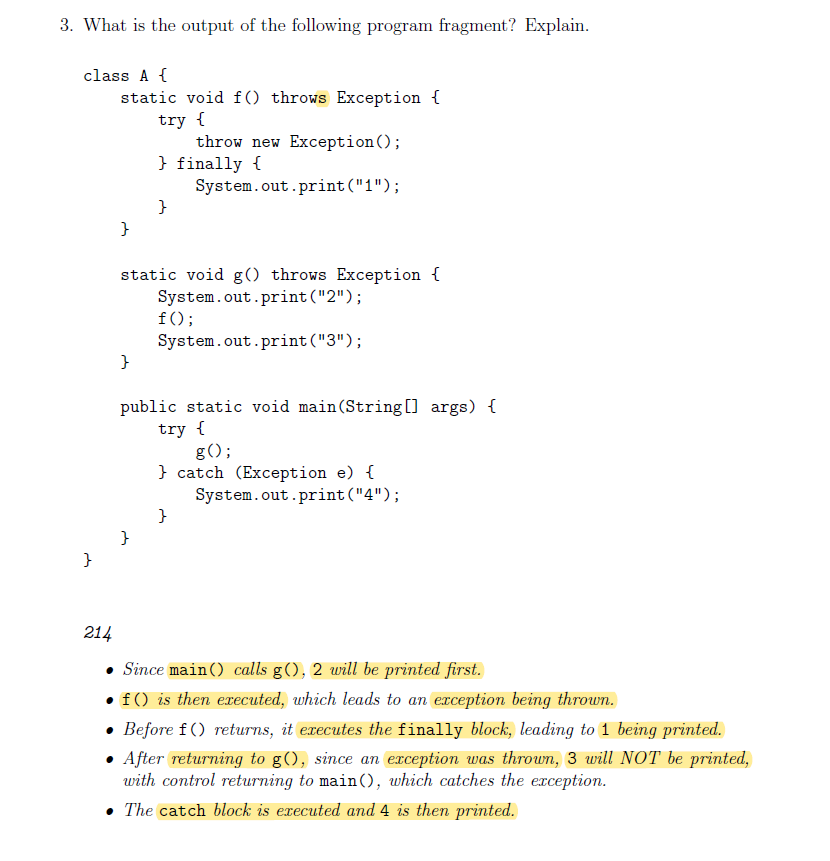
## Enum

* Special type of class used to define constants (ie are objects)
* All enums inherit from the class Enum<E> implicitly
* Is type-safe. Ie color – 1; is invalid.
* Each constant of an enum type is an instance of the enum class and is a field declared with public static final
* Constructors, methods, and fields can be defined in enums

## Access Modifiers

* Import/package statements at the top of each .java file (refer to projects 1 and 2)
* private (visible to the class only)
* default (visible to the package)
* protected (visible to the package and all sub-classes)
* public (visible to the world)

## Error Handling

* **throws** is signature (ie at the top where A throws NullPointerException)
* **throw** is a return statement (ie throw new NullPointerException(); )
* use try-catch blocks to handle exceptions (try, catch, finally)
* 

## Exceptions

**Checked Exception**

* occurs at compile time (eg returns null)

**Unchecked Exception (ie RuntimeException)**

* occurs at runtime (eg ArithmeticException)
* null could still be here…

# Functional Programming

Chaining methods, lambda expressions, etc.

Declarative (ie what to do) vs OOP’s imperative (ie how to do)

Good for parallel/concurrent computation

Memoization

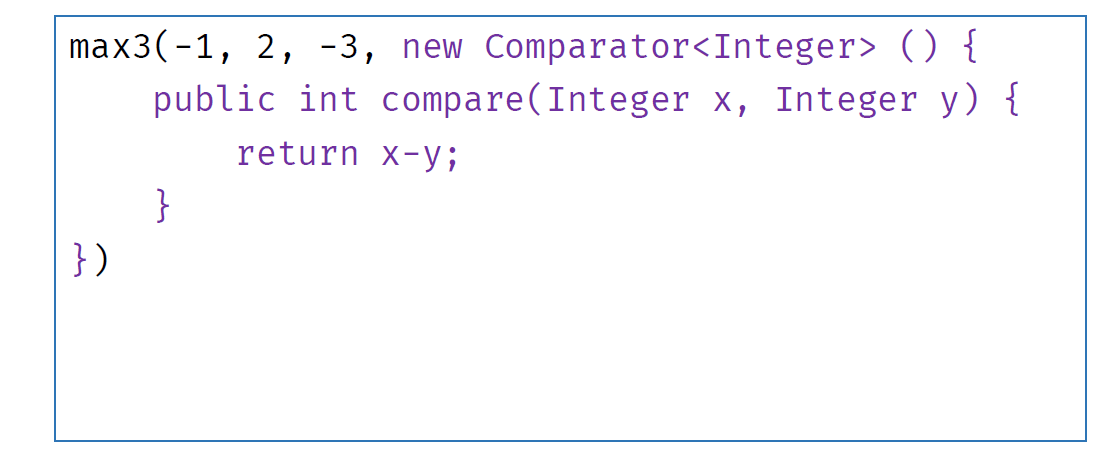
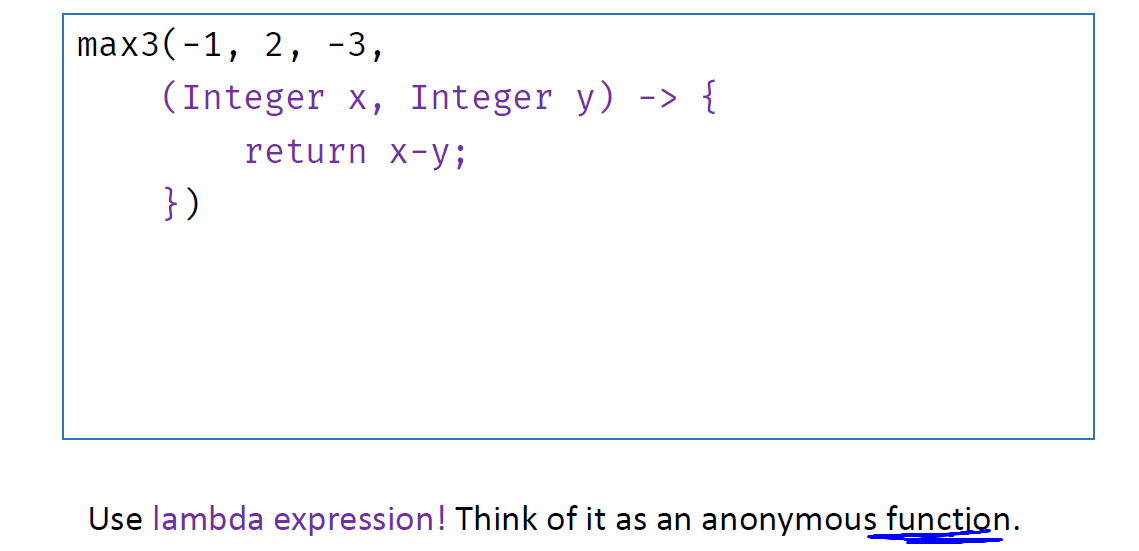
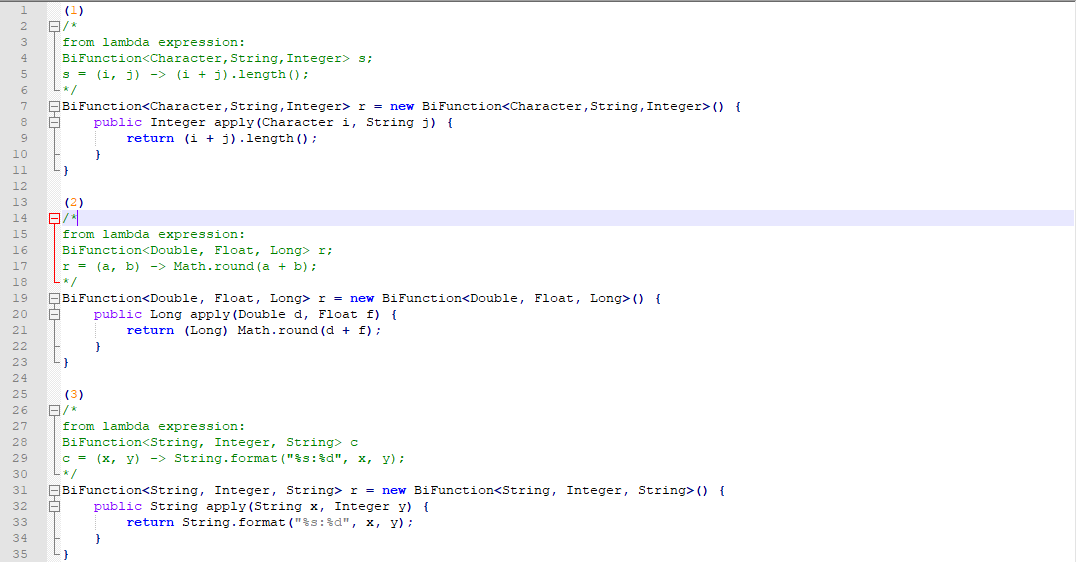
## Functional Interface (Anonymous Classes, Lambdas, Method Refs)

It is a Single Abstract Method that must be overridden by the user. Eg (Function, Predicate, Supplier, Consumer, etc)

**Purposes**

1. As a way of passing functions into other functions, or returning a function as a value, or for assigning a function to a variable. (Functions thus become first-class objects, like other objects)
2. For the compiler to check that the caller of the object is indeed using a type that has a the required method. The compiler checks that the type implements that interface.

**Classes**

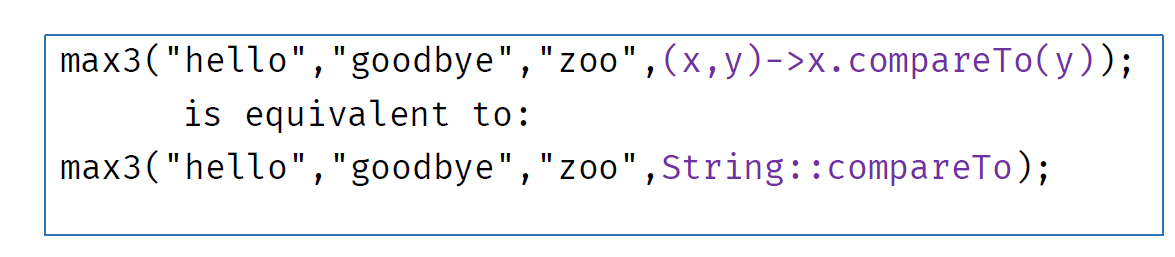
* Single-Use: create a class just to implement the interface and override the interface’s method (eg compare for Comparator<T>)
* Anonymous: create a class within the argument line. 
* Lambda-expressions: using the method’s original argument signature. 
* 

**Method reference**

Either use (Class::method)

or

x -> Class.method(x)



**Lambda Closure (Read under Java Memory Model)**

* The variables nand resultare copied from the stack into a closure object (in the heap).
* These variables must be final or effectively final(compiler will flag error if they are changed.) ie cannot do the following:

int n = 0;

Stream.of(1,2,3).forEach(x -> { n = n + x; });

n must be effectively final.

**Function Composition**

* (F o G)(x) == F(G(x))

**Read-Eval-Print Loop (REPL)**

* Use of while loop to just keep printing (ie project 1 and 2’s queue system)

**Data-Directed Programming**

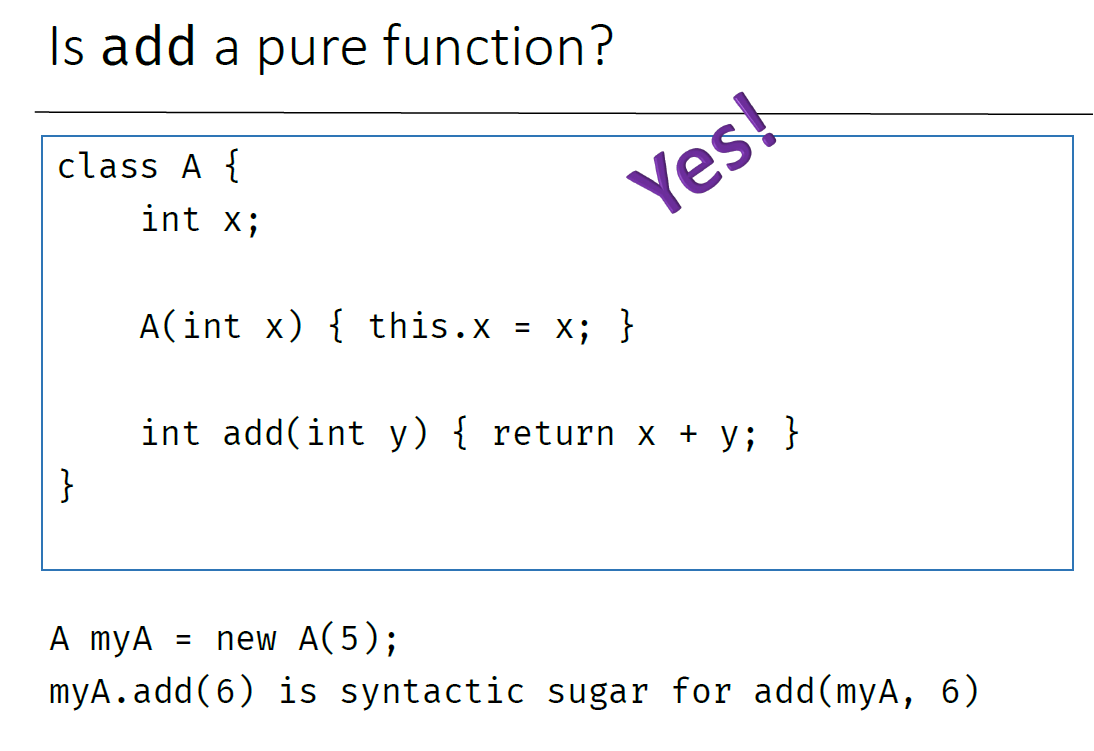
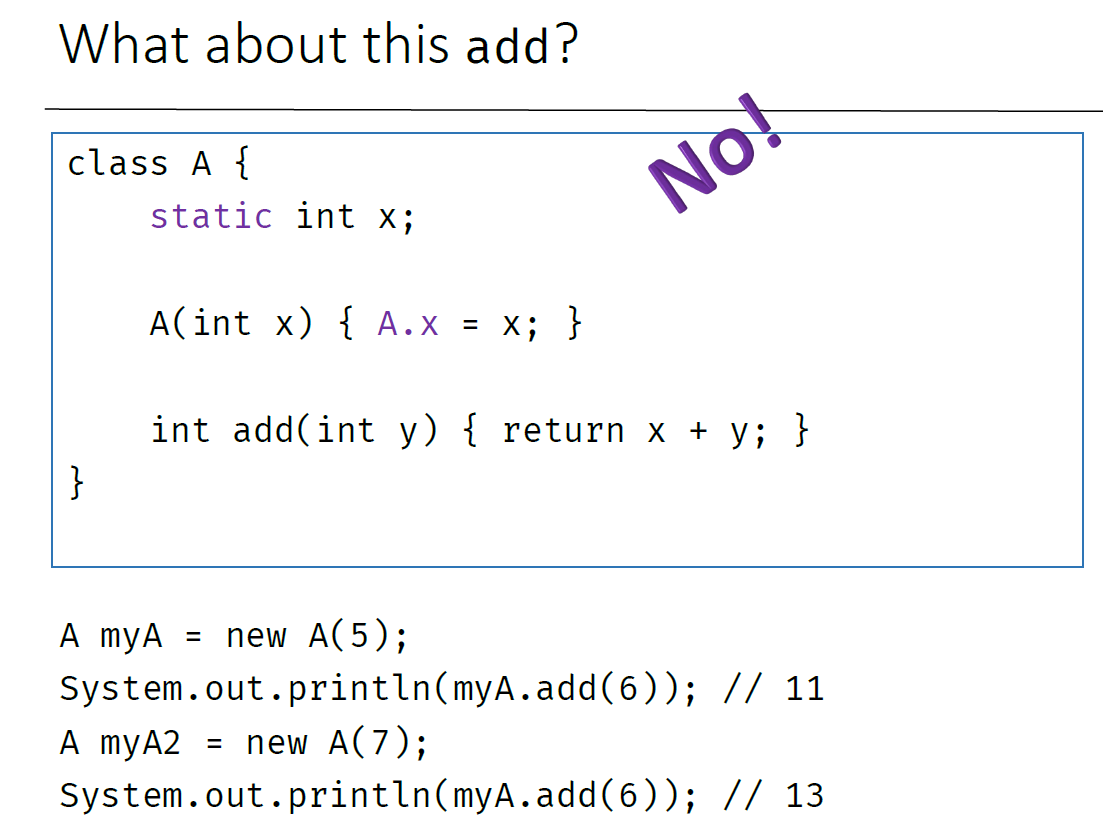
* Use of HashMap<K, V> etc for key -> value pairing.

## FP Principles

NO SIDE EFFECTS

### Pure Functions (Deterministic)

* **Have no side effects** eg. input/output, throwing exceptions, changing external state
* **Do not mutate data**. Instead, a change of state is represented by new data
* **Always return a value** Thus, a void function is not pure
* **That is the same for the same inputs**. Return value is fully determined by the input arguments; and nothing else

### Immutability

* Return a **new object** with the new values
* Original object, if called again, should retain original value (ie unmodified)
* Variables must be immutable

### Referential Transparency

* pure functions + immutable data = referential transparency
* allows for memoization

### First Class Functions

Functions as first-class entities can do the following:

* refer to it from constants and variables
* pass it as a parameter to other functions
* return it as result from other functions

### Higher-order Functions (Lambda expressions)

The function can either:

* takes one or more functions as arguments, or
* returns a function as its result

### Function Composition

* Reduce, map, filter should be present

## Functor

Functor is a box (aka context) to contain a thing, which then can be manipulated safely. Restores the partial function back into a pure one. Examples include Optional, Stream, ArrayList, Function, etc

Functor design pattern must have:

* Immutable generic box (ie. parametrized type)
* Constructor: (different names)
* Method: map

**Functor Laws**

1. Identity: make(t).map(x -> x).equals(make(t))
2. Associativity: make(t).map(f).map(g).equals(make(t).map(f.andThen(g)))

## Monads

parametrized type that contains a thing, along with a constructor (of, aka unit), and a method flatmap (aka bind)

**Monad Implements:**

1. A parameterized type M<T>: in Java terms, public class M<T>
2. flatMap (bind): a method that takes a monad as well as a function mapping an element to a monad, and returns the result of applying that function to the value wrapped in the monad
3. unit (ie identity, return, .of() method etc): which is a factory function to make a monad out of an element

**Monad Laws:**

1. Left Identity: applying the unit function to a value and then binding the resulting monad to function f is the same as calling f on the same value: let f be a function returning a monad

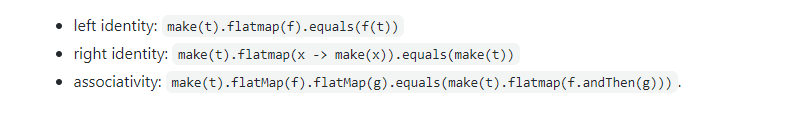
Ie bind(unit(value), f) === f(value)

1. Right Identity: binding the unit function to a monad doesn’t change the monad

Ie let m be a monadic value (an instance of M<T>), then bind(m, unit) === m

1. Associativity: if we have a chain of monadic function applications, it doesn’t matter how they are nested

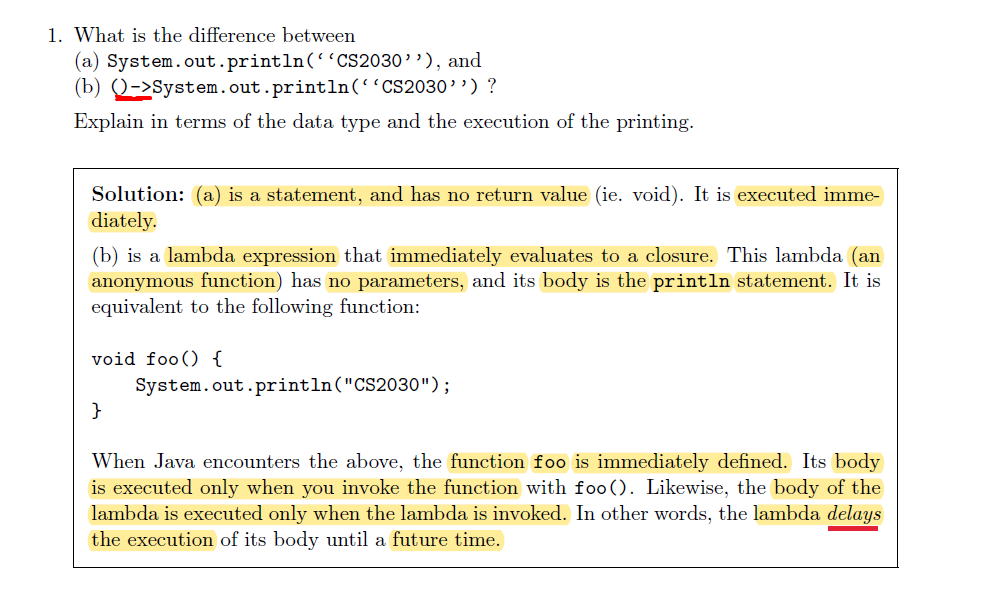
Ie bind(bind(m, f), g) === bind(m, x -> g(f(x)))



# Evaluation (Lazy vs Eager)

Lazy Evaluation benefits: Avoids unnecessary computation, Amortizes time complexity (when used with memoization(aka caching)), Allows for infinite data structures.

Use Supplier<T> to make an evaluation lazy and recursion!



## Memoization

An optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

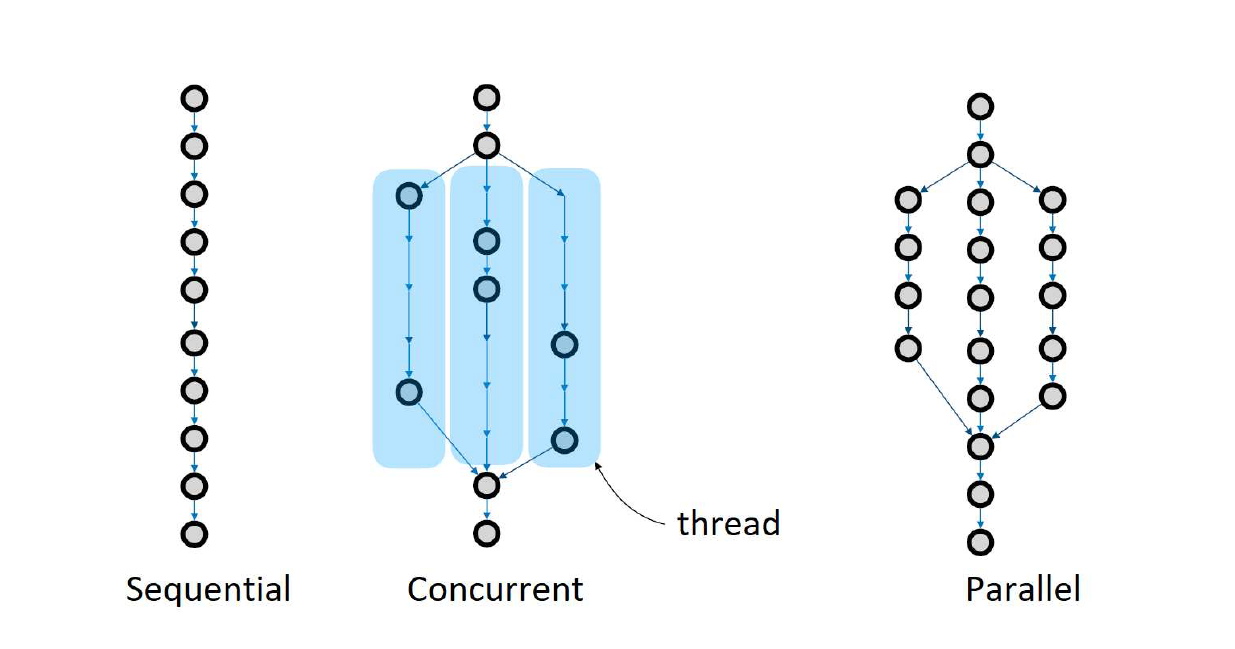
Read lecture 10’s slides.

# Sequential & Parallel Programming

## Stream

* A stream pipeline starts with a data source
* Intermediate operations specify tasks to perform (using lazy evaluation)
  + Stateless: filter, map, etc.
  + Stateful: sorted, limit, distinct etc
* Terminal operation initiates the processing of a stream pipeline so as to produce a result

## Concurrency vs Parallelism



* Parallelizing a trivial task actually creates more work in terms of parallelizing overhead (ie time required to spawn new threads). Only worthwhile if the task is complex enough that the benefit of parallelization outweighs the overhead incurred.
* Uses ForkJoinPool.
  + Fork(): concurrent event
  + Join(): blocks all processes until the threads reach completion
* 

# Asynchronous Programming

## Synchronous

* The method gets executed, and when the method returns, the result of the method (if any) becomes available
* might delay the execution of subsequent methods

## Asynchronous

* Creates a thread that does the work
* Passing a Runnable to the Thread constructor (Runnable is a SAM with abstract run() method)
* Start the thread with start() method to execute the computation, run() method to execute on the same thread
* Allows for unrelated task to be done while waiting for previous task’s completion (Busy waiting)
* Completion using join(). Throws an InterruptedException if current thread is interrupted
* Alternatively, can use a callback:
  + A callback (more aptly call-after) is any executable code that is passed as an argument to other code so that the former can be called back (executed) at a certain time
  + execution may be immediate (synchronous callback) or happen later (asynchronous callback)
  + Avoid repetitive checking to see if the asynchronous task completes
  + Callback may be invoked from a thread but is not a requirement
  + An observer pattern can be utilized where the callback can be invoked, say notifyListener / via a function

**CompletableFuture / CompletionStage**

* No need to figure out all the above-mentioned stuff yourself. Just use the in-built methods 😊. Otherwise we have got to know how long a process takes then approximately call the listener etc…