CS2030S Notes

Principles of OOP

Encapsulation	The process of restricting direct access to some of an object's components, as well as the bundling of the object's data with the object's methods (or other functions) operating on that data.
Abstraction	The process of separating the abstract properties of a data type and the concrete details of its implementation.
Inheritance	Use the `extends` keyword, to inherit attributes and methods from one class to another.
Polymorphism	**Polymorphism** uses methods inherited to perform different tasks. This allows us to perform a single action in different ways. -An object in Java that passes more than one IS-A tests is polymorphic in nature. -Every object in Java passes a minimum of two IS-A tests: one for itself and one for Object class. -Static polymorphism in Java is achieved by method overloading during compile time (same name different parameters). -Dynamic polymorphism in Java is achieved by method overriding during run-time (same method signature

Widening vs Narrowing Type Conversion

Widening	Narrowing
`float f = 2.5; int i = 5; f = i;` There is no need for casting as `int` <: `float`, thus it can be widened,	`float f= 2.5; int i = (int) f;` need casting for narrowing type conversion

For Narrowing Type Conversion, if there is sno explicit type casting, there would be complilation error.

Method Signature

The **method name**, **type** of arguments, **order** of arguments, and **number** of arguments.

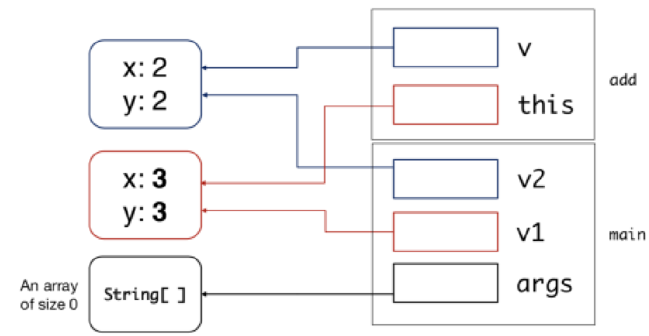
Stack and Heap

```
class Vector2D {
 private double x;
 private double y;
 Vector2D(double x, double y) {
   this.x = x;
   this.y = y;
```



```
 }
 void add(Vector2D v) {
   this.x += v.x;
   this.y += v.y;
 }
}

class Main {
   public static void main(String[] args) {
   Vector2D v1 = new Vector2D(1, 1);
   Vector2D v2 = new Vector2D(2, 2);
   v1.add(v2);
}
```



The Heap Space contains **all objects are created**, but Stack contains any reference to those objects. Objects stored in the Heap can be accessed throughout the application. Primitive local variables are only accessed the Stack Memory blocks that contain their methods.

Static Binding vs Dynamic Binding

Static Binding	Dynamic Binding
The binding of static, private and final methods is compile time. The reason is that the these method cannot be overridden and the type of the class is determined at the compile time.	When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed. The actual type of object is determined at the runtime so this is known as dynamic binding.

`class Human { public static void walk() { System.out.println("Human walks"); } } class Boy extends Human{ public static void walk(){ Systen.out.println("Boy walks"); } }`	**Step 1: At compile-time** `curr.equals(obj)` • Identify the compile-time type of `curr` (`Object`) • Check the class `Object` for methods called `equals` • There is only one method `Object::equals(Object)` • Store this method descriptor in the generated bytecode `circle.equals(circle);` What if there are multiple methods that can be correctly invoked? `boolean equals(Circle c) {:} boolean equals(Object c) {:}` We choose the *most specific* one. `equals(Circle)` is more specific than `equals(Object)`. We store this method descriptor in the generated bytecode. **Step 2: At run-time** `curr.equals(obj)` • The method descriptor from Step 1 is retrieved • The run-time type of the target `curr` is determined • Java then looks for an accessible method with the matching descriptor in that class • If no such method is found, the search will continue up the class hierarchy • The first matching descriptor will be the one that is executed Class methods do not support dynamic binding. The method to invoke is resolved statically during compile time.

what it is declared as.
Step ① : o1.equals (o2)
compile time - type
Object
- compiler will go to the def^n of 'Object' class and look for a method called equal
- Find the method descriptor, and write this in byte-code
∴ boolean equals (obj).

Step ② : Run - time - type.
- When it runs, it will look at what is o1 assigned to, ∴ it is a 'Circle' Class.
- Find method class., boolean equals (obj)

✿ At no point in this process, we do not look at the argument's class. ∴ This prints equals (Object) called

Step1 : CT
c1.equals (o2)
- c1 declared as a circle, thus CT is 'Circle' class.
- Look for methods named equals. 1) equals (obj) 2) equals (circle)
(can't guarantee argument is a specific object)
- compiler cannot be sure o2 (argument) is assigned as a 'circle' class. Thus compiler will assign o2 as an Object.
∴ most specific method will be equals (obj)
boolean equals (Object)

step 2 : RT : circle.

casting is COMPILE-TIME
C1.equals ((circle) o2) step 1 : Circle.
It will take o2 as a Circle class. As type casting is compile-time
∴ More specific one is
boolean equals (Circle)
step 2 : RT : Circle.

Liskov Substitution Principle

Let $\phi(x)$ be a property provable about objects x of type *T*. Then $\phi(y)$ should be true for objects y of type *S* where *S<:T*.
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

A *subclass* should not break the expectations set by the *superclass*. We can only substitute with an instance of the same *type* or a *subtype*.

How can a method in a subclass break a superclass method's contract?

1. Returning an object that's incompatible with the object returned by the superclass method.
2. Throwing a new exception that's not thrown by the superclass method.
3. Changing the semantics or introducing side effects that are not part of the superclass's contract.

Answer format:
Yes, it violates LSP. The subclass changes the behaviour of the superclass, so the property that (enter desirable property here) no longer holds for the subclass. Places in a program where the superclass is used cannot be simply replaced by the subclass.

`final` keyword

- Final Class: Cannot be subclassed
- Final Method: Cannot be overridden or hidden by subclasses
- Final Variable: A final variable **can only be initialized once**, either via an initializer or an assignment statement. It does not need to be initialized at the point of declaration: this is called a "blank final" variable. A blank final instance variable of a class must be definitely assigned in every constructor of the class in which it is declared; similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared; otherwise, a compile-time error occurs in both cases.

Variance of Types

Let $C(T)$ be a complex type based on type T. We say a complex type C is:
- *covariant* if S<: T implies $C(S)$ <: $C(T)$.
- *contravariant* if T <: S implies $C(S)$ <: $C(T)$.
- *invariant* if C is neither covariant nor contravariant.

Java array is covariant e.g. Integer[] <: Object[]

Diamond Problem

Since a single class can implement multiple interfaces, the diamond problem actually occurs when a class implements several interfaces with the same default methods and method signatures.

```
public interface Alarm {
 default String turnAlarmOn() { return "Turning the alarm on."; }
 default String turnAlarmOff() { return "Turning the alarm off."; }
}
public class Car implements Vehicle, Alarm {
 // ...
}

interface A {
  default void f();
}
interface B {
  default void f();
}
class AB implements A,B {} // this would not compile because there are 2 default methods
// to implement, there is ambiguity.
```

Overriding methods

```
class A {
  int x;
  A(int x) {
    this.x = x;
  }
  public A method() {
    return new A(x);
  }
}

class B extends A {
  B(int x) {
    super(x);
  }
  @Override
  public B method() {
    return new B(x);
  }
}

A a = new B();
A a2 = a.method();
//compile time type A     run time type B
//this works as B<:A, we can return a subclass of the overriden method.
//if a.method() returns a super class (returns A) of the overridden method (when this
//returns B in the original method overridden), this fails.
```

Generics

Limitations for Generics

You cannot set a variable of a generic type as a static field for a class:

```
| Error:
| non-static type variable T cannot be referenced from a static context
| static T y;
| ^
```

If you want a static method to return a generic type result, you need to write the method as:

```
static <X> X foo(X t) {
 // ...
}
```

if not you will get the following error:

```
| Error:
| non-static type variable T cannot be referenced from a static context
| static T foo (T t){return t;}
| ^
| Error:
| non-static type variable T cannot be referenced from a static context
| static T foo (T t){return t;}
| ^
```

You also cannot use a primitive for a generic type:

```
Pair<int> x = new Pair<>();
| Error:
| unexpected type
| required: reference
| found: int
| Pair<int>
| ^-^
```

Generics are ***invariant***

Supposed $T <: S$, this does not imply that $\texttt{Array<}T\texttt{>} <: \texttt{Array<}S\texttt{>}$ nor does it imply $\texttt{Array<}T\texttt{>} :> \texttt{Array<}S\texttt{>}$. Sub-type relationship does not imply immediately the generic typing relationships.

Bounded and Unbounded Generics

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
To declare a bounded type parameter:

- List the type parameter's name
- Along with the extends keyword
- and its Upper Bound

`<T extends superClassName>` or we can have multiple bounds with interfaces: `<T extends superClassName>`

How does Generics ensure type safety?

Generics allow classes and methods that use any reference type to be defined without resorting to using the Object type. It enforces type safety by binding the generic type to a specific given type argument at compile time. Attempt to pass in an incompatible type leads to compilation error.
OR
<> helps compiler in verifying type safety. Compiler makes sure that List<MyObj> holds objects of type MyObj at compile time instead of runtime. Generics are mainly for the purpose of type safety at compile time. All generic information will be replaced with concrete types after compilation due to type erasure.

Type Erasure

Type erasure is the process the compiler does in order to implement generics.

```
Pair<String,Integer> p = new Pair<String,Integer>("hello", 4);
Integer i = p.getSecond();
```

The compiler will change it in a way where it will turn into the following:

```
Pair p = new Pair("hello", 4);
Interger i = (Integer) p.getSecond(); // This is all done by the compiler.
```

In the Pair class type, when doing type checking, the compiler will erase all the types again and make it similar to how we defined it initially.


```
class Pair {
  private Object first;
  private Object second;
  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }
  Object getFirst() { return this.first; }
  Object getSecond() { return this.second; }
}
```

Suppose one of the type is bounded, that type will be replaced with the class it is bounded by. Type erasure will also do the casting if there needs to be one.

```
//Supposed class Pair<T, S extends Comparable<S>>
class Pair {
  private Object first;
  private Comparable second;
  public Pair(Object first, Comparable second) {
      this.first = first;
      this.second = second;
  }
  Object getFirst() { return this.first; }
  Comparable getSecond() { return this.second; }
}
```

When we lose these type information, we can no longer infer the types of these objects

```
Pair<String,Integer>[] pairArray;
Object[] objArray;
pairArray = new Pair<String,Integer>[2]; //cannot to instantiate a gen array like this
objArray = pairArray;
objArray[0] = new Pair<Double,Boolean>(3.14, true);

//AFTER TYPE ERASURE
Pair[] pairArray;
Object[] objArray;
pairArray = new Pair[2];
objArray = pairArray;
objArray[0] = new Pair(3.14, true); //Double type in the head pair
String str = pairArray[0].getFirst(); //Run time error

//Cannot mix ARRAYs and GENERICs
new Pair<String,Integer>[2]; // error
new Pair<S,T>[2]; // error
new T[2]; // error
```

Let’s build `Array<T>`

```
class Array<T> {
  private T[] array;

  public Array(int length) {
    // The only way we can put an object into array is through the method set() and we
    // only put object of type T inside. So it is safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked") //Instructions to the compiler.
    // this.array = (T[]) new Object[length]; This does not work, since we cannot
    // instatiate the array immediately.
    T[] temp = (T[]) new Object[size];
    this.array = temp;//
  }

  public void set(int index, T t) {
    this.array[index] = t;
  }

  public T get(int index) {
    return this.array[index];
  }
}

Array<String> strArray = new Array<String>(2);
Object[] objArray = strArray.getArray();
objArray[0] = 5;
String s = strArray.get(0);
```

Upper-Bounded Wildcard

```
Array<? extends Shape>
// ? can be substituted by a subtype of Shape.
A<Shape> <: A<? extends Shape>
A<Circle> <: A<? extends Shape>
A<Square> <: A<? extends Shape>
```

- Covariance if S<:T, then A<? extends S> <: A<? extends T>
  - e.g A<? extends Circle> <: A<? extends Shape> since Circle <: Shape
- For any type `S`, `A<S>` <: `A<? extends S>`
  - A<Circle> <: A<? extends Circle>
  - These rules are transitive.

Lower-Bounded Wildcard

```
Array<? super Shape>
// ? can be substituted by a supertype of Shape
```

- Contravariance if `S` <: `T`, then `A<? super T>` <: `A<? super S>`
  - e.g Circle <: Shape, then A<? super Shape> <: A<? super Circle>
- For any type `S`, `A<S>` <: `A<? super S>`

Unbounded Wildcard

```
Array<?>
// This wildcard can can be substituted by any type
```

- For any type S, `A<S>` <: `A<?>`
- For any type S, `A<? super S>` <: `A<?>`
- For any type S, `A<? extends S>` <: `A<?>`

`A<?>` is the super type of any upper/lower bounded wildcards, the mother of all types of this generic class A<S>.

PECS

Remember PECS: "Producer Extends, Consumer Super".

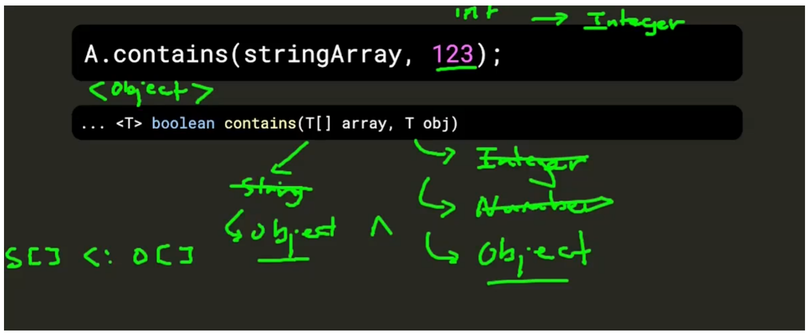
- "Producer Extends" - If you need a List to produce T values (you want to read Ts from the list), you need to declare it with ? extends T, e.g. List<? extends Integer>.
  But you cannot add to this list.
- "Consumer Super" - If you need a List to consume T values (you want to write Ts into the list), you need to declare it with ? super T, e.g. List<? super Integer>. But there are no guarantees what type of object you may read from this list.
- If you need to both read from and write to a list, you need to declare it exactly with no wildcards, e.g. List<Integer>.

```
new Array<Fruit>(0);
  //copy Fruit -> Array
  //copy Orange -> Array (any subtype of fruit)
  //Array<Fruit>, Fruit or any of its subtypes

  public void copyFrom(Array<? extends T> src) {...}
```

```
  //from source we are getting information -> this is a producer, hence EXTENDS.

//similar for 'super'
static <T extends Comparable<? super T>> T max3(List<T> arr) {
  T max = arr.get(0);
  if (max.get(1).compareTo(max) > 0) {
    max = arr.get(1);
  }
}
// this Comparable<T> which is max is being taken in by the compareTo method,
//hence this is a consumer, thus <? super T>What type should the object be?
```



The most specific one that overlaps both types is of type **OBJECT**, hence java would infer their objects to be Object.

```
Shape s = A.findLargest(new Array<Circle>(0));
... <T extends GetAreable> T findLargest(Array<? extends T> array)
// 1 T extends GetAreable
// T here can match any subtype of GA -> Shape -> Circle
// GA being an interface can also be implemented by other objects

// 2 T findLargest
// T here can match any subtype of Shape -> Circle/Square/Triangle etc
// In compile time Shape s is defined to be a Shape, thus i can
// take on any subtype

//3 Array<? extends T>
// T here can match any super type of Circle
// Wildcard extends Circle, thus any super type of circle can be
// taken on here. A<Circle> <:
// A<? extends Circle/Shape/any super type of circle>
```

Exception Handling


```
class A {
  static void f() throws Exception {
    try {
      throw new Exception();
    } finally {
      System.out.print("1");
    }
  }

  static void g() throws Exception {
    System.out.print("2");
    f();
    System.out.print("3");
  }

  public static void main(String[] args) {
    try {
      g();
    } catch (Exception e) {
      System.out.print("4");
    }
  }
}

//Output is 214
```

In main, `try g()` was first executed, this first printed **"2".**

Within `g()`, after printing **"2"**, `f()` was executed.

Within `f()`, a new `Exception` was thrown, before exiting `f()`, `finally` block was executed, hence printing **"1"**.

Since `f()` threw the exception, the `catch` caught the exception throw, and hence printed out **"4"** within the `catch` block, before the last line in `g()` was executed.

Checked vs Unchecked Exceptions

Unchecked Exceptions

- caused by programmer errors
- should not happen if code is perfectly written
- not explicitly caught or thrown
- cause run-time errors
- `IllegalArugumentException`, `NullPointerException`, `ClassCastException`

Checked Exceptions

- Something programmer has no control over
- Anticipate and handle them
- e.g File Cannot open → `FileNotFoundException`
- must be handled or program would not

- In general, unchecked exceptions are   compile subclasses of `RunTimeException`

```
public boolean checkGuestStatus(String name) {
  try {
    lookUpGuest(name);
    return true;
  } catch (NoSuchGuestException e) { //NoSuchGuestException thrown from lookUpGuest
    return false;
  } catch (Exception e) {
    return false;
  }
}
```

Design Problems →

1. Programmer is using exception as flow control. A better way would be for `lookUpGuest` to return a **boolean** to indicate if a guest exists.
2. The second `catch` catches all possible exceptions. This is the Pokémon exception and should be avoided.
3. `NoSuchGuestException` is something that the programmer anticipates so it should be a checked exception instead.

Immutability

- Immutable class → an instance cannot have any visible changes outside its abstraction barrier
- **Advantages:**
  1. Ease of understanding
  2. Enabling safe sharing of Objects
  3. Enabling safe sharing of Internals
  4. Enabling safe conucrrent execution
- To update something `final` we can explicitly reassign the item.


```
final class Circle {
  final private Point c;
  final private double r;

  public Circle moveTo(double x, double y) {
    return new Circle (c.moveTo(x, y), r);
  }
}
```

Nested Classes

Inner classes - associated with an instance

Can access ALL fields/methods of containing class

```
class A {
  private int x;
  static int y;
  class B { // this is not static
    void foo() {
      x = 1; // accessing x from A is OK
      y = 1; // accessing y from A is OK
    }
  }
}
```

Static nested classes - associated with the containing class

can ONLY access static fields/methods of containing class

```
class A {
  private int x;
  static int y;
  static class C {
    void bar() { // We cannot access non-static fields from here
      x = 1; // accessing x from A is not OK (instance of class A A.x)
      y = 1; // accessing y is OK, y is static
    }
  }
}
```

`B.this`

```
class A {
 private int x;
 class B {
```

```
    void foo() {
      this.x = 1; // error -> instead we need to use A.this.x
      //this.x refers to class B's current declartion of x, not present.
    }
  }
}
```

Since `this.x` is called within a method of `B`, `this` would refer to the instance of `B`, rather than the instance of `A`. Java has a piece of syntax called qualified `this` to resolve this. A qualified `this`
reference is prefixed with the enclosing class name, to differentiate between the `this` of the inner class and the `this` of the enclosing class. In the example above, we can access `x` from `A` through the `A.this` reference.

```
class A {
 private int x;

 class B {
   void foo() {
     A.this.x = 1; // ok
   }
 }
}
```

Local classes

A local class is a class defined **within** a method. This sorting is not particularly useful outside of this method, thus we define a *local* class within the method.

Can access: class and instance variables from the enclosing class (use qualified `this`) + local variables of the enclosing method

- can only access variables declared `final` or *effectively final*
- *effectively final* → variable does not change after initialization

Comparator is a 3rd party that compares 2 objects. It has a `compare()` method taking in 2 arguments of type T and returns an `int`.

```
void sortNames(List<String> names) {
    class NameComparator implements Comparator<String> {
      @Override
      public int compare(String s1, String s2) {
        return s1.length() - s2.length();
      }
    }
```



```
    names.sort(new NameComparator());
}
```

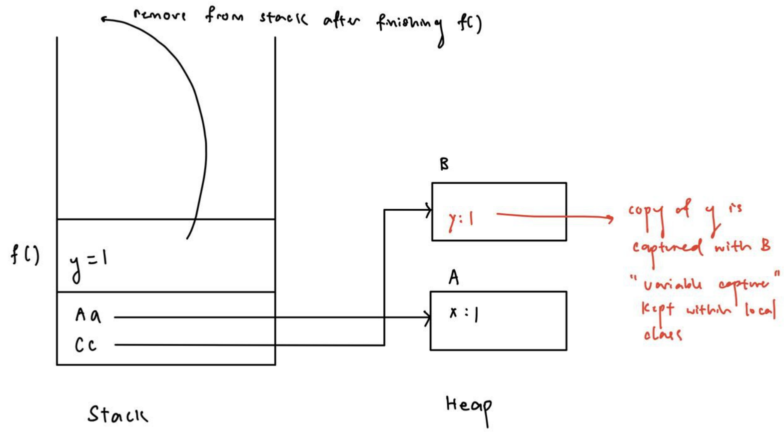
Within local class, we can use parameters from within the scope of the method. e.g a `int x = 0;` defined within the `sortNames()` function, the local class `NameComparator` can access the local variable within the enclosing method.

```
interface C { void g(); }
class A {
    int x = 1;
    C f() {
      int y = 1;
      class B implements C {
        void g() {
          x = y; // accessing x and y is OK.
        }
      }
      B b = new B();
      return b;
    }
}
```

Variable capture

Calling `A.f()` will give us a reference to an object of type B. But now after calling `c.g()`, the value of y is captured. The local class can access the local variables in the enclosing method, as the local class makes *a copy of variables inside itself*. This is how the local class captures the variables.

```
A a = new A();
C c = A.f(); // this returns Object of type B
c.g();
```



Anonymous classes

Anonymous classes do not have names!

- cannot implement more than one interface
- cannot extend a class an implement an interface at the same time
- same rules as local classes for variable access

```
void sortNames(List<String> names) {
    names.sort(new Comparator<String>() {
        public int compare(String s1, String s2) {
          return s1.length() - s2.length();
       }
    });
}
```

Effectively Final

Never changed after declaration. The following code would throw an error, as it is being captured by the `NameComparator` class, thus having 2 versions of `ascendingOrder` would cause ambiguity in capture.

```
void sortNames(List<String> names) {
    boolean ascendingOrder = true;
    class NameComparator implements Comparator<String> {
      public int compare(String s1, String s2) {
        if (ascendingOrder)
          return s1.length() - s2.length();
        else
          return s2.length() - s1.length();
      }
    }
    ascendingOrder = false;
    names.sort(new NameComparator());
}
```

Java only allows a local class to access variables that are explicitly declared `final` or implicitly final (or *effectively* final). An implicitly final variable does not change after initialization. `ascendingOrder` is not effectively final so the code above does not compile.

Lambda expressions can use variables defined in an outer scope. We refer to these lambdas as *capturing lambdas*. They can capture static variables, instance variables, and local variables, but only **local variables must be final or effectively final.**

Pure Functions

No side effects, i.e. don't:

- print to the screen
- write to files
- throw exceptions
- change other variables
- modify the values of the arguments

```
//Pure functions:
int square(int i) {
  return i * i;
}
int add(int i, int j) {
  return i + j;
}

//Not Pure functions
int div(int i, int j) {
  return i / j;  // if j = 0, this will throw an exception
}
int incrCount(int i) {
  return this.count + i;  // this.count is not guranteed final, thus its
}                         // not deterministic, hence also not pure.
```

Functions as first-class citizens

You can create an "instance" of a function, as have a variable reference that function instance, just like a reference to a String, Map or any other object.

```
Transformer<Integer, Integer> square = new Transformer<>() {
  @Override
  public Integer transform(Integer x) {
    return x * x;
  }
};
```

- `@FunctionalInterface` annotation (only one abstract method)

```
@FunctionalInterface
interface Transformer<T, R> { R transform(T t); }
```

Lambda Expressions

We can use this to rewrite **functional interfaces**.

Original code writtren with an anonymous class:

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = new Transformer<>() {
    @Override
    public Double transform(Point p) {
        return origin.distanceTo(p);
    }
}
```

Using a lambda expression:

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = p -> origin.distanceTo(p);
```

Method Reference

but since `distanceTo` takes in one parameter and returns a value, it already fits as a transformer, and we can write it as:

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = origin::distanceTo;
```


- static method in a class: `ClassName:staticMethodName`
- instance method of a class/interface: `instanceName::methodName`
- instance method of an object of a particular type: `Type::methodName`
- constructor of a class `ClassName::new`
- **at compile time**: Java searches for the matching method - performs type inference to find the method that matches that method reference. **Multiple matches/Ambiguous match** ⇒ compilation error.

Curried functions

- translate a general *n-ary* functions to n unary functions
- stores the data from the environement where it is defined
  - closure → a construct that stores a function together with the enclosing enviroment

```
Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);

add.transform(1).transform(1);

//Add is a function that takes in an Integer object and returns a unary Function
//over Integer. So add.transform(1) returns the function y -> 1 + y.
//We could assign this to a variable:

Transformer<Integer,Integer> incr = add.transform(1);
incr.transform(1); // returns 2
```

Note that `add` is no longer a function that takes two arguments and returns a value. It is a *higher-order function* that takes in a single argument and returns another function.

Lazy Evalutation

Producer && Task Interface - also Functional Interfaces (only 1 abstract method)

```
interface Producer<T> {
  T get();
}

interface Task {
  void run();
}
```

```
Producer<Integer> p = () -> 3;
p.get();
//$4 ==> 3

Producer<Integer> p = () -> {System.out.println("Execute"); return 3;};
//p ==> $Lambda$18/0x00000008000b7040@7d9d1a19

p.get();
Execute
//$7 ==>3     --> as we can see here we have delayed the evaluation as we only evaluate
// this when we use the get() method.
```

How to be lazy

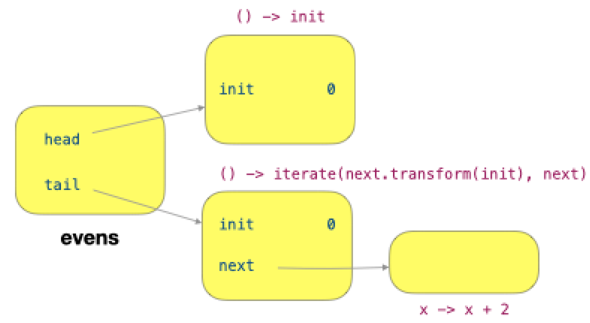
- Procrastinate until the last minute
- Never repeat yourself

Memoization

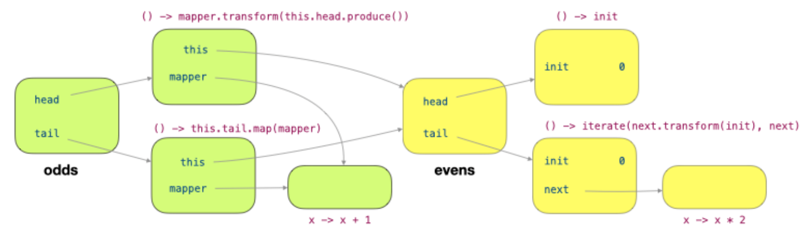
- If we have computed the value of a function before, we can cached (or memoize) the value, keep it somewhere, so that we don't need to compute it again.
- Only if the function is pure → regardless of how many times we invoke the function, it always returns the same value, and invoking it has no side effects on the execution of the program.

InfiniteList

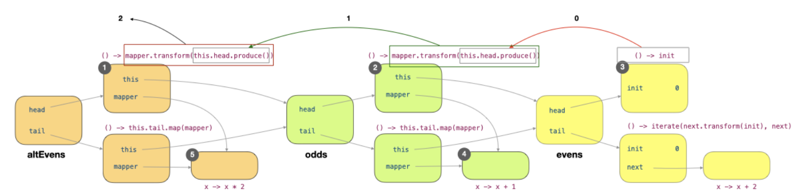
```
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2); // 0, 2, 4, 6, ...
```


```
InfiniteList<Integer> odds = evens.map(x -> x + 1); // 1, 3, 5, ...
```




```
InfiniteList<Integer> altEvens = odds.map(x -> x * 2); // 2, 6, 10, ..
```



Streams

A stream pipeline starts with a *data source*, then *intermediate operations*, and finally a *terminal operation*.

⇒ can only be **consumed once**, `IllegalStateException` if consumed more than once.

Data sources

- Static factory methods `of`, `generate`, `iterate`
- Convert an array into a Stream using `Arrays::stream`
- Convert a list instance into a Stream using `List::stream`

Terminal Operations

- `forEach`

More Terminal Operations

- `reduce`
- `noneMatch`, `allMatch`, `anyMatch`
- `count`

Intermediate Operations

- `filter`, `map`
- `flatMap`
- `limit`, `takeWhile`

Some operations are *stateful*

These operations have to keep track of the state of the stream when executing, keeping track of all of the elements in the stream.

- `sorted`
- `distinct`

Peeking into our `Stream`

```
Stream.iterate(0, x -> x + 1)
      .peek(System.out::println)
      .takeWhile(x < 5)
      .forEach(x -> {});
```

```
s = Stream.iterate(0, x -> x + 1).takeWhile(x -> x < 5)
s ==> java.util.stream.WhileOps$1@5bcea91b

s.forEach(System.out::println)
```



```
0
1
2
3
4

s.forEach(System.out::println)
| Exception java.lang.IllegalStateException: stream has already been operated upon
|                                            or closed
|       at BastractPipeline.evaluate (AbstractPipeline.java:229)
|       at ReferencePipeline.forEach (RefernecePipeline.java: 491)
|       at (#51:1)

//The stream has already been evaluated once, hence we would need to create a new
//instance of the stream to evaluated it again
```

Monads

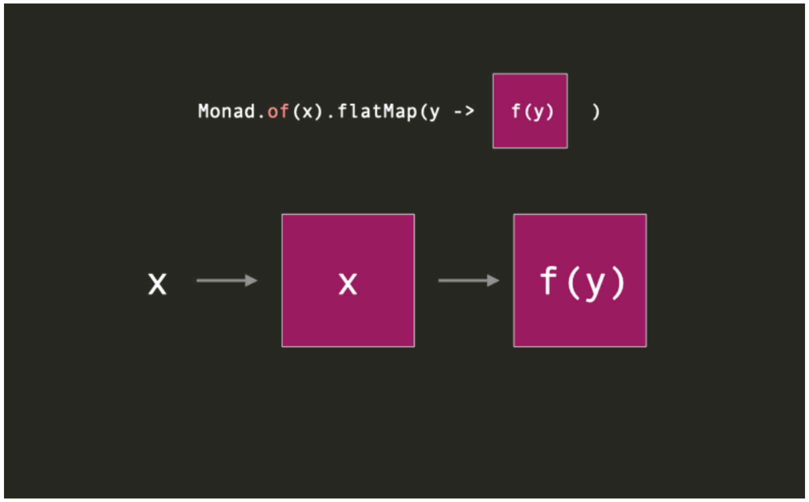
- `Maybe<T>`: item might be missing
- `Lazy<T>`: item is evaluated on demand
- `Loggable<T>`: item is logged
- These 3 above are Monads
- `InfiniteList<T>`: items in a lazily-evaluated list
- `Array<T>`: items in an array
- `Box<T>`: item in a box

Monads are an abstraction that needs to follow certain rules. A monad has two methods `of` and `flatMap`, `of` to intialize the value and side information, and `flatMap` to update the value and side information. They obeys the three laws below.

Left Identity Law

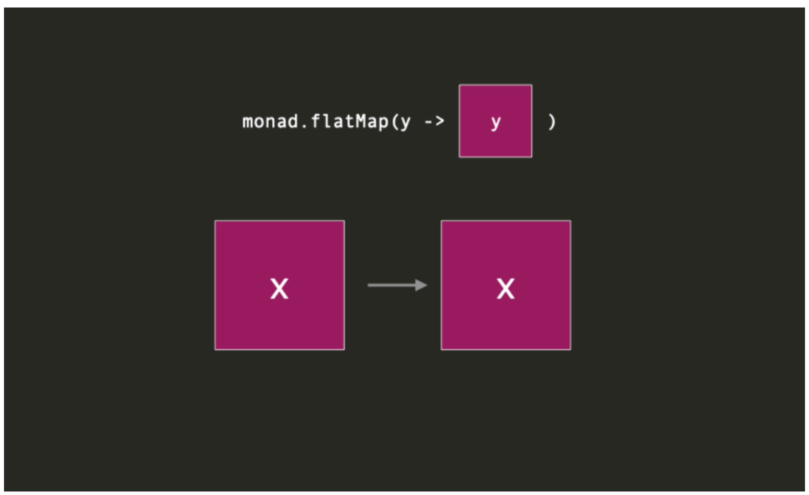
`Monad.of(x).flatMap(y -> f(y))` is equivalent to `f(x)`.

`Monad.of` should not do anything different than creating a new instance of the monad, thus it should be the same as applying another method directly to an already created monad.



Right Identity Law

`monad.flatMap(y -> Monad.of(y))` is equivalent to `monad`.

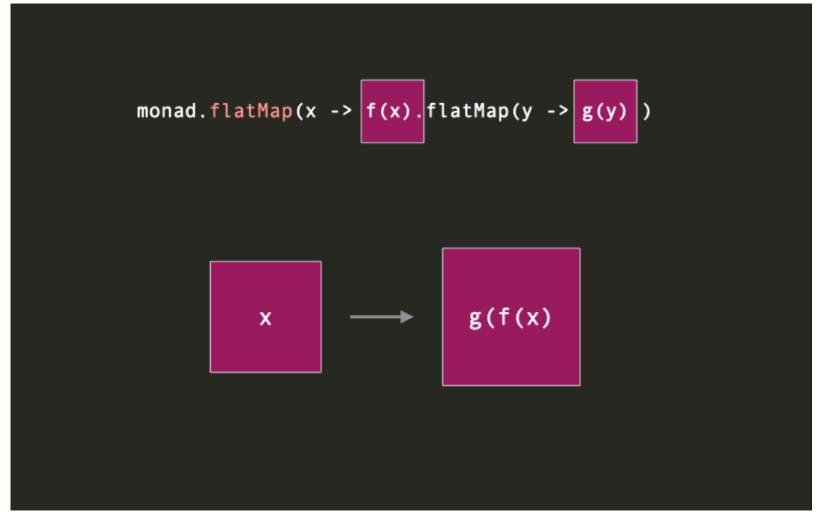


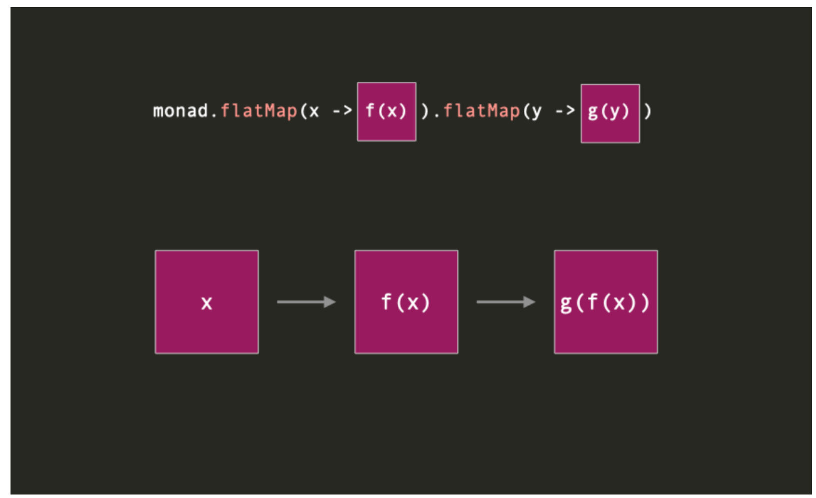
Associative Law - same result regardless of how it is composed

`monad.flatMap(x -> f(x)).flatMap(x -> g(x))`

is equivalent to

`monad.flatMap(x -> f(x).flatMap(x -> g(x)))`





$(A+B)+C$ should be equivalent to $A+(B+C)$, similarly, for a Monad, however order you apply the `flatMap` transformation should not have any effect on its output.

Functor

A functor obeys two laws.

1. `functor.map(x -> x)` is just `functor` (**preserving identity**)
2. `functor.map(x -> f(x)).map(x -> g(x))` is just `functor.map(x -> g(f(x)))` (**preserving composition**)

Parallel and Concurrent Programming

add `.parellel()` before the terminator or use `parellelStream()` in order to turn the stream to be processed parallel.

Stream operations must not interfere with the stream data, most of the time must be *stateless*. Side effects should be kept to a minimum.

- **Interference**

  one of the stream operations modifies the source of the stream during execution of the terminal operation (throws `ConcurrentModificationException`) → also applies if `stream()` used

- **Stateful vs Stateless**

  A stateful lambda is one where the result depends on any state that might change during the execution of the stream.

  e.g ⇒ `generate` and `map` are stateful as they depend on the state of input. Parallelizing may lead to inconrrect output.

  ```
  Stream.generate(scanner::nextInt).map(i -> i + scanner.nextInt())
      .forEach(System.out::println)
  ```

- **Side effects**

  Side effects can lead to incorrect results in parallel execution.

  ```
  List<Integer> list = new ArrayList<>(Arrays.asList(1,3,5,7,9,11,13,15,17,19));
  List<Integer> result = new ArrayList<>();
  list.parallelStream().filter(x -> isPrime(x)).forEach(x -> result.add(x));
  ```


`forEach` lambda generates a side effect as result is modified. `ArrayList` is non-thread safe, two threads manipulating it at the same time would cause result to be incorrect

  1. `.collect()` method

     ```
     list.parallelStream().filter(x -> isPrime(x)).collect(Collectors.toList())
     ```

  2. thread safe data structure → `CopyOnWriteArrayList`

     ```
     List<Integer> result = new CopyOnWriteArrayList<>();
     list.parallelStream().filter(x -> isPrime(x)).forEach(x -> result.add(x));
     ```

- **Associativity**

  The `reduce` operation is inherently parallelizable, as we can easily reduce each sub-stream and then use the `combiner` to combine the results. Consider this example:

  ```
  Stream.of(1,2,3,4).reduce(1, (x, y) -> x * y, (x, y) -> x * y);
  // where the identitiy is 1, the combiner is (x, y) -> x * y and the
  // accumulator is (x, y) -> x * y
  ```

  To allow us to run `reduce` in parallel, however, there are several rules that the `identity`, the `accumulator`, and the `combiner` must follow:

  - `combiner.apply(identity, i)` must be equal to `i`.
  - The `combiner` and the `accumulator` must be associative -- the order of applying must not matter.
  - The `combiner` and the `accumulator` must be compatible -- `combiner.apply(u, accumulator.apply(identity, t))` must equal to `accumulator.apply(u, t)`

  The multiplication example above meetings the three rules:

  - `i * 1` equals `i`
  - `(x * y) * z` equals `x * (y * z)`
  - `u * (1 * t)` equals `u * t`

Performance of Parallel Stream

Parallelizing a stream does not always improve the performance. Creating a thread to run a task incurs some overhead, and the overhead of creating too many threads might outweigh the benefits of parallelization.

Ordered vs. Unordered Source

Order of the stream also plays a role in the performance of parallel stream operations.

- ordered → created from `iterate`/ordered collections (e.g., `List` or arrays)/ `of`
- unordered →created from `generate` /unordered collections (e.g., `Set`)

`distinct` and `sorted` preserve original order (a.k.a it is STABLE) → ONLY for **finite streams**.

If we have an ordered stream and respecting the original order is not important, we can call `unordered()` as part of the chain command to make the parallel operations much more efficient.

Threads (`java.lang.Thread`)

Threads → single flow of execution in a program.

```
new Thread(Runnable) //constructor takes in a Runnable

new Thread(() -> {
  for (int i = 1; i < 100; i += 1) {
    System.out.print("_");
  }
}).start();

new Thread(() -> {
  for (int i = 2; i < 100; i += 1) {
    System.out.print("*");
  }
}).start();
```

- `start()` is returned immediately, not only upon lambda expression completes its execution, either of the above Threads operations could run and finish first.
- `Runnable` → functional interface with `run()` methods (returns void)
- `isAlive()` → returns boolean representing if the thread is alive
- `Thread.currentThread()` → returns reference of current running thread (for name: `Thread.currentThread().getName()`)


```
System.out.println(Thread.currentThread().getName());
    new Thread(() -> {
      System.out.print(Thread.currentThread().getName());
      for (int i = 1; i < 100; i += 1) {
        System.out.print("_");
      }
    }).start();
```

thread called `main` also printed, which is a thread created automatically for us every time our program runs and the class method `main()` is invoked.

```
Stream.of(1, 2, 3, 4)
    .parallel()
    .reduce(0, (x, y) -> {
        System.out.println(Thread.currentThread().getName());
        return x + y;
    });

//output
main
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-9
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-3

//If you remove the parallel() call, then only main is printed, showing the
//reduction being done sequentially in a single thread.
```

- `Thread.sleep(ms)` → pauses execution of the current thread for `ms` amount of time.
  After the sleep timer is over, the thread is ready to be chosen by the scheduler to run again.

CompletableFuture Monad

Creating a CompletableFuture

- `completedFuture()` → equivalent to creating a task that is already completed and return us a value.
- `runAsync()` → takes in a `Runnable` lambda expression. `runAsync` has the return type of `CompletableFuture<Void>`. The returned `CompletableFuture` instance completes when the given lambda expression finishes.

- `supplyAsync()` → takes in a `Supplier<T>` lambda expression. `supplyAsync` has the return type of `CompletableFuture<T>`. The returned `CompletableFuture` instance completes when the given lambda expression finishes.

Chaining a CompletableFuture

- `thenApply`, which is analogous to `map` | Async method → `thenApplyAsync`
- `thenCompose`, which is analogous to `flatMap` | Async method → `thenComposeAsync`
- `thenCombine`, which is analogous to `combine` | Async method → `thenCombineAsync`

The methods above run the given lambda expression in the same thread as the caller. Async methods may cause the given lambda expression to run in a different thread (thus more concurrency).

- concurrency → divides computation into subtasks called threads
  - separate unrelated tasks into threads; write each thread separately
  - improves utilization of the processor (can switch between threads)
- parallelism → multiple subtasks are truly running at the same time
- parallelism $\subseteq$ concurrency

Getting the result

`get()` and `join()` blocks the result so it ensures these results are returned first before the computation of other results.

`get()` → `CompletableFuture::get` throws a couple of checked exceptions: `InterruptedException` and `ExecutionException`, which we need to catch and handle.

`join()` → does not thrown exceptions.

```
int findIthPrime(int i) {
  return Stream.iterate(2, x -> x + 1).filter(x -> isPrime(x)).limit(i)
  .reduce((x, y) -> y).orElse(0);
}

CompletableFuture<Integer> ith = CompletableFuture.supplyAsync(() -> findIthPrime(i));
CompletableFuture<Integer> jth = CompletableFuture.supplyAsync(() -> findIthPrime(j));

CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);

diff.join(); //to actually get the value of the difference
```

Handling exceptions

`handle()` method → The first parameter to the `BiFunction` is the value, the second is the exception, the third is the return value.

```
CompletableFuture.<Integer>supplyAsync(() -> null).thenApply(x -> x + 1).join();
//would throw a CompletionException with a NullPointerException contains within it

cf.thenApply(x -> x + 1).handle((t, e) -> (e == null) ? t : 0).join();
```

Drawback of Programming with `Thread` directly

- Overhead - (additional costs + each Thread can only be used once).
- Exception handling?
- Not easy to pass information around. (How do we communicate between threads).

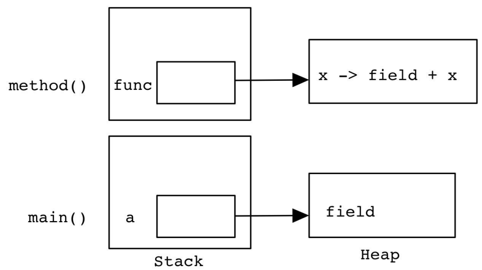
Past Year Paper Questions

2017/2018 S1

1. **Stack and Heap**

```
class A {
  int field;
  void method() {
    Function<Integer, Integer> func = x -> field + x;
  }
}
```

- `A` is a <u>class</u> so the instance field `field` stored on the **heap**.
- `func` is a <u>local variable</u>, so it would go on the **stack**.
- `func` REFERS to a <u>lambda expression</u>, internally implemented as an <u>anonymous class</u> → **heap.**
- `x` is an argument that is **NOT STORED ANYWHERE**.



if `A a = new A(); a.method();` was executed, ⇒ only `a` (hence the inner instance field `field`) remains on the heap, as the `func` would have been popped off the stack.

2. **CompletableFuture**

```
import java.util.concurrent.CompletableFuture;
class CF {
  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> {
      doSomething();
      System.out.print(i);
    });
  }
  public static void main(String[] args) {
```



```
    printAsync(1);
    CompletableFuture.allOf(printAsync(2), printAsync(3)).join();
  }
}
```

Which of the following are possible output printed by the program if main runs to completion normally?
(i) 123 (ii) 213 (iii) 1 (iv) 32
<u>D. Only (i), (ii) and (iv)</u> `printAsync(1)` is not synchronized so it can get printed anytime, or not printed at all before the program exits. But, there is a call to join() for printAsync(2) and printAsync(3). So we know for sure that both 2 and 3 will be printed (in any order).

3. **Parrallelizing streams.reduce()**

```
Stream.of(1,2,3,4).parallel().reduce(0, (result, x) -> result * 2 + x);
//has a different output from (the unparallelized form)
Stream.of(1,2,3,4).reduce(0, (result, x) -> result * 2 + x);
```

Reduction operation is not associative as `2 * result + x` is <u>non-associative</u>. Result depends on order of reduction.

4. `Function<? super Integer, Integer>` is better than `Function<Integer, Integer>`. Why?

W*hy it is better*: The argument is declared this way sothat we can pass in any function that operates on the superclass of Integer (such as `Number` and `Object`) (e.g., `Function<Object,Integer> h = x -> x.hashCode()`) into map.

2017/2018 S2

5. **No compilation error for the following methods below**

```
class A {
  static int x;
  static int foo() {  return 0; }
  int bar() { return 1; }
  static class B {}
  public static void main(String[] args) {:}
}
x = 1;
A a = new A();
B b = new B();
new A().foo();
new A().bar();
```

6. **Compile Time and Run type**

Compile Time

(i) type inference – inferring the type of a variable whose type is not specified.
(ii) type erasure – replacing a type parameter of generics with either Object or its bound.
(iii) type checking – checking if the value matches the type of the variable it is assigned to.

only (i) and (ii) happens during compile time as Type checking happens during runtime since the value is not always available at compile time.

Run Time

(i) late binding – determine which instance method to call depending on the type of a reference
object.
(ii) type casting – converting the type of one variable to another.
(iii) accessibility checking – checking if a class has an access to a field in another class.

only (i). Note that type casting happens during compile time but is checked during runtime. Accessibility checks happen both during run time and compile time.

- **Type casting** - **Compile time** but could not be catched and throw `ClassCastException` (Java Type Casting)
- **Late binding** - **Runtime** in general and **Compile time** for calls to final, private, or static methods (Late Binding in Java)
- **Accessibility checking** - In Java, accessibility checking is enforced at **runtime** and **compile time,** because Java also has a runtime typesystem, and it can dynamically (at runtime) create classes. So it needs to enforce access at runtime too for types it doesn't know about at compile time.
- **Type inference** - **Compile time** - Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable
- **Type erasure** - **Compile time** - Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime
- **Type checking** - **Compile time** but facilitating **runtime** type checking eg via `instanceof`

7. **Parallelizing and variable capture**

```
void foo() {
  int sum = 0;
  Stream.of(1, 2, 3, 4, 5).parallel().forEach(i -> {sum = sum + i;}); //line 3
  System.out.println(sum); //line 4
}
```

This is not a good demonstration, because:
A. `foo()` always prints 15 since `forEach()` will be executed sequentially.
B. `foo()` always prints 15 since the lambda expression passed to `forEach()` has no side effect.
C. `foo()` will not compile because Java expects `sum` to be either final or effectively final.
D. `foo()` always prints 0 since, due to variable capture, there are two copies of sum now, and the captured version of `sum` is being incremented in Line 3.
E. `foo()` may print different results every time `foo()` is invoked, but not due to side effects. The reason is that `System.out.println` on Line 5 is invoked in parallel with the code on Line 3. The code should call `join()` to wait for all the elements in the stream to be added to `sum` before printing.

8. **Explain why Line A would lead to a compiler warning of unchecked cast.**

```
public <R> Undoable<R> undo() {
  Deque<Object> newHistory = new LinkedList<>(this.history);
  R r;
  try {
    r = (R)newHistory.removeLast(); // Line A
  } catch (NoSuchElementException e) {
  // Missing line B }
  return new Undoable<R>(r, newHistory);
}
```

This is a narrowing type conversion, from R to Object. But since R is erased during compile time, the runtime system cannot safely check the type to make sure that it matches. Note that saying that it is a narrowing type conversion is not enough and is wrong – narrowing type conversion does not cause a compiler warning (you have seen this many times (e.g., in equals) so you should know this!).

```
Undoable<Integer> i = Undoable.of("hello").flatMap(s -> length(s));
Undoable<Double> d = i.undo();
```

The code runs without error. Even though we assign an Undoable<String> to Undoable<Double>, during runtime, it is stored as an Object reference, and the reference can refer to String. An error would occur only if we try to apply function that operates on Double to the Undoable, in which case it will throw a ClassCastException.

2019/2020 S2

1. **Turn lambda expression into anonymous class**

```
BiFunction<Character,String,Integer> s;
s = (i, j) -> (i + j).length();

->
s = new BiFunction<Character, String, Integer>() {
  @Override // need to override the apply method in BiFunction
  public Integer apply(Character i, String j) {
  return (i + j).length();
  }
};
```

2. **CompletableFuture**

```
import java.util.concurrent.CompletableFuture;
  class CF {
  static void doSomething() { .. }
  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> { doSomething(); System.out.print(i); });
  }
  public static void main(String[] args) {
    printAsync(1).join();
    CompletableFuture.allOf(printAsync(2), printAsync(3)).thenRun(() -> printAsync(4));
    doSomething();
  }
}
```

- `join()` method is blocking, so all CompletableFutures will wait for `printAsync(1)` to be completed, and hence printed before continuing evaluation. Hence 1 will be printed before anything else.
- `allOf()` method waits for all CompletableFutures in its parameters to finish completing before printing them). Since `thenRun(() -> printAsync(4)))` is after `allOf()` call, should 4 be printed, 2 and 3 must be printed (in any order) before 4 is printed. 4 may or may not be printed.

- if `allOf()` is `anyOf()`, upon completion of a single CompletableFuture, 4 may be printed afterwards.

4 if printed, must be either **12** or **3** are printed. **2**, if printed, must be after **1** is printed.

```
public static void main(String[] args) {
  CompletableFuture.anyOf(printAsync(1).thenRun(() -> printAsync(2)), printAsync(3))
    .thenRun(() -> printAsync(4));
  doSomething();
}
```

2020/2021 S2

1. **Consider the following code:**

```
Integer i = 0;
Object o = (Number) i;
```

Which of the following statement(s), if any, is true?
A. The runtime type of o is Integer .
B. The compile-time type of i is Object .
C. i 's compile-time type is inferred to be Number .
D. The type of i has been erased to Object .
E. The code would cause a compilation error.
F. The code would cause a run-time error

The compile-time type of i is Integer ; for o is Object. Since i points to 0 (zero), o will also points to 0 (zero), o has the runtime type of Integer . Type inference is not involved since all the types are spelled out. Type erasure is not involved since generics are not used. The code would compile fine since we have a widening type conversion from Number to Object .

2. **Overriding**

```
class ParentException extends Exception {}
class Parent<T> {
  public <R> Parent<R> foo(Parent<? extends T> p) throws ParentException {:}
  // insert class Child here
}
```

```
private class Child<S> extends Parent<S> {
  public <R> Parent<R> foo(Parent<? extends S> p) throws ParentException {
```

```
    return null;
  }
} //No @Override → It is OK to leave out @Override , which is just an annotation
```

```
private class Child<S> extends Parent<S> {
  @Override
  public <R> Parent<R> foo(Parent<? extends S> p) throws Exception {
    return null;
  } //Different exception thrown →
}   //Error, child method throws more general exception, violate LSP
```

```
private class Child<S> extends Parent<S> {
  @Override
  private <R> Parent<R> foo(Parent<? extends S> p) throws ParentException {
    return null;
  } //Different access modifier -> Error, child method is trying to throw a more
}   //general exception. Violates LSP.
```

```
private class Child<S> extends Parent<S> {
  @Override
  public <R> Parent<R> foo(Parent<? extends S> p) {
    return null;
  } //No exception thrown -> A Child<S> instance that does not throw
}   //an exception can substitute a Parent . No LSP is violated.
```

```
private class Child<S> extends Parent<S> {
  @Override
  public <R> Child<R> foo(Parent<? extends S> p) throws ParentException {
    return null;
  } //Different return type ->
}   //Can return a more specific type (i.e., a subtype) when we override.
```

```
private class Child<S> extends Parent<S> {
  @Override
  public <R> Parent<R> foo(Parent<S> p) throws ParentException {
    return null;
  } //Different parameter type -> Error.Parent<S> and Parent<? extends S> considered
}   //two different types at compile time (both erased to Parent but still diff).
```

```
private class Child<S> extends Parent<S> {
  @Override
```



```
  public <T> Parent<T> foo(Parent<? extends S> p) throws ParentException {
    return null;
  } //Different type variable -> Different name used for type param, still OK.
}
```

3. **Overloading**

Following code will not compile, we cannot overload two methods with the same method signature (after type erasure). Both are erased to

```
public class ArraySort {
  public int sort(Array<String> arrayString) {...}
  public int sort(Array<Integer> arrayInteger) {...}
}
```

4. **Compose 2 functions using lambda expression**

```
Transformer<Integer, Integer> f = x -> x + 1;
Transformer<Integer, Integer> g = x -> x * 2;

Box<Transformer<Integer, Integer>> box; //put f in box
box = Box.of(f);

box = box.map(***** ff -> x -> g.transform(ff.transform(x)) *****);
box.get().transform(4); // should return 10;
```

Monad (10 marks)

Consider the following monad, `Monad<T>` ,

```java
import cs2030s.fp.Transformer;

class Monad<T> {
  private T x;

  private Monad(T x) {
    this.x = x;
  }

  public static <T> Monad<T> of(T x) {
    return new Monad<>(x);
  }

  public T get() {
    return x;
  }

  public <R> Monad<R> flatMap(Transformer<? super T, ? extends Monad<? extends R>> f) {
    return new Monad<>(f.transform(this.x).get());
  }

  public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
      return flatMap(???);
  }
}
```

22. Complete the implementation of `map` using only `flatMap` so that the resulting `Monad<T>` that satisfies the functor laws.

Solution: `x -> Monad.of(f.transform(x))`

23. Show that the monad `Monad<T>` preserves composition and therefore meets one of the requirements of being a functor. The skeleton of the proof is given below. Fill in the blanks by completing the Expressions A, B, and C, as well as two monad laws we use.

Suppose we have an instance of `Monad<T>` called `m` and two functions `f` and `g` .

`m.map(x -> f(x)).map(x -> g(x))`

is equivalent to the following Expression A based on the implementation above:

`m.flatMap(x -> Monad.of(f(x))).flatMap(x -> Monad.of(g(x)))` (Expression A)

Invoking Monad's __**Associative**__ Law, Expression A is equivalent to Expression B below,

`m.flatMap(x -> Monad.of(f(x)).flatMap(x -> Monad.of(g(x))))` (Expression B)

Invoking Monad's __**Left Identity**__ Law, Expression B is equivalent to Expression C below,

`m.flatMap(x -> Monad.of(g(f(x))))` (Expression C)

which, by our implementation, is equivalent to

`m.map(x -> g(f(x)))`

Therefore, the composition of functions is preserved in our implementation.

5. **CompletableFuture**


```
CompletableFuture<Integer> cf = ten.thenApply(plus(1));
cf.thenApply(plus(10)).join();
cf.thenApply(plus(5)).join();

the program will always print     11  21  16     every time it is executed.
```

TRUE. `join()` ensures that the program exists only after all three numbers are printed. Due to the `join()` in the second line, the third line will always be executed after the second line completes. Thus, the numbers will always be printed and printed in order.

```
CompletableFuture<Integer> cf = ten.thenApply(plus(1));
cf = cf.thenApplyAsync(plus(10));
cf.thenApplyAsync(plus(5)).join();

the program will always print      11  21  26     every time it is executed.
```

TRUE. `join()` ensures that the program exists only after all three numbers are printed. Since the completable future chains them in order, the numbers will be printed in order.

```
CompletableFuture.allOf(ten.thenApplyAsync(plus(1)), ten.thenApplyAsync(plus(10)),
ten.thenApplyAsync(plus(5))).join();

the program will always print      11  20  15     every time it is executed.
```

FALSE. `join()` only ensures that the program exists only after all three numbers are printed, but not the order of the numbers are printed.

Midterm 2022 S2

1. **Type Checking**

```java
class Store<T> {
  T x;
  void keep(T x) {
    this.x = x;
  }
  T get() {
    return this.x;
  }
}

Store<String> stringStore = new Store<>();
```

```
Store store = stringStore;
store.keep(123); // Line A
String s = stringStore.get(); // Line B
```

Why compiler give compilation warning in Line A →

The compiler will give us an unchecked warning, as the compiler is not sure if this line is a type safe operation. Specifically, as we are using the Raw Type `Store` , the compiler can not check if it is safe to pass an `Integer` to the `keep` method. It will allow us to do this, but warn the programmer.

What happens during run time in Line A. →

During runtime, the program will assign the value 123 to the variable x. This is allowed due to type erasure, i.e. T x will become `Object x` after type erasure, and `Object` can point to an `Integer` .

What happens during run time in Line B. →

The program will have a `ClassCastException` as it tries to cast an Integer to a `String`. This is the result of the previous line in which we did an unsafe operation.

2. **Method Override Generics**

```
class A<T extends Comparable<T>> {
  public T foo() {
    return null;
  }
}
class B<T> extends A<T> {
  public T foo() {
    return null;
  }
}
```

False

- The `T` in `A` requires `T` to be a subclass of `Comparable<T>`. But the `T` in `B` does not have
  such as constraint so the `T` from `B` cannot be passed as type argument to `A` .
- Alternatively, `T` in `A` is erased to `Comparable` but `T` in `B` is erased to `Object`. The return
  type of the overriding `foo` cannot be a supertype of the return type of the overridden `foo`.

3. **Polymorphism**

`doTask(A)` in class `B` . The compile-time type of `c` and `d` are A and D respectively. There is only one `doTask` method in `A` , so the method descriptor `void doTask(A)` is chosen. During run-time, c has the run-time type of `C`. But there is no method `doTaskA` defined in `C`, travelling up the class hierarchy, `doTask(A)` is found in class `B` , so it is invoked.

```
class A {
    public void doTask(A a) { }
}

class B extends A {
    @Override
    public void doTask(A a) { }

    public void doTask(B b) { }
}

class C extends B {
    public void doTask(C c) { }
}

class D extends B {
    @Override
    public void doTask(A a) { }
}
```

```
A a = new A();
B b = new B();
A c = new C();
D d = new D();
```

9. (2 points) Consider the following code excerpt:

```
c.doTask(d);
```

Which `doTask` will be invoked?

A. `doTask(A)` in class `A`
B. `doTask(A)` in class `B`
C. `doTask(B)` in class `B`
D. `doTask(C)` in class `C`
E. `doTask(A)` in class `D`
F. None of the above.