Java

Structure of mtds

1. Access Modifier 2. Return Type 3. Mtd Name 4. Parameters in () 5. Mtd body in {}

1. *public protected private abstract default static final transient volatile synchronized native strictfp*

Access Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

abstract class //cannot be instantiated
abstract mtd //must be implemented in subclass
final class ... //cannot create subclass of this final class(String, Integer)
final mtd //cannot override it
final attribute //cannot change value
- static : no access to object attributes-> no additional data other than the parameters

2. Return data type

Primitive Data Type	Size	Description	Wrapper Class (treat primitive as objs)
byte	1 byte	Integers from -128 to 127	Byte
short	2 bytes	Integers from -32,768 to 32,767	Short
int	4 bytes	Integers from -2,147,483,648 to 2,147,483,647	Integer
long	8 bytes	Integers btw ±9,223,372,036,854,775,808(7)	Long
float	4 bytes	Fractional numbers. Up to 6 to 7 dp	Float
double	8 bytes	Fractional numbers. Up to 15 dp	Double
char	2 bytes	Single character/letter or ASCII values	Character
boolean	1 bit	True or false values	Boolean
byte <: short <: int <: long <: float <: double primitive data type stored directly, but classes stored by reference			

char <: int	//e.g. char x = 'A', char x = '\u00A9', char x = 65, default is '\u0000'
void : return nothing	
Object	//super of all classes
String	//char x = text.charAt(index), where text is a String
int num = 1_000_000	//can use underscore

E.g.
public static float expt(float x, int n) {...}
public static void main (String[] args) { // when program is run, execution start from this main line
...
}

Convention
- name class with SomeClass
- name attributes with someAttribute, if final attribute : SOME_ATTRIBUTE

//single line comment
/*
multi
line
comment`
*/

/** * bar does bar. * Link to {@link Foo}. * Link to foo in Foo {@link Foo#foo}. * Link to anotherBar in Bar {@link #anotherBar}. * Link to an overloaded method: {@link Foo#foo(int)}. * @param a some integer a. * @return bar. * @throws RuntimeException if exception. */	java docs To view javadoc javadoc <file>.java <webbrowser (chromium)> index.html

()		
++, --	Increment/ decrement by 1	

<table>
<tr><td>unary +, -</td><td>Assign sign to number</td><td>+5 - +7 (+ is unary, - is binary)</td></tr>
<tr><td>%</td><td></td><td></td></tr>
<tr><td>*, /</td><td>4 / 3 = 1;   4.0 / 3 = 1.333</td><td>* cannot work with string</td></tr>
<tr><td>+, -</td><td></td><td></td></tr>
<tr><td>=, +=, -=, *=, /=, %=</td><td></td><td></td></tr>
<tr><td>True && False</td><td>and</td><td></td></tr>
<tr><td>True || False</td><td>or</td><td></td></tr>
<tr><td>^</td><td>bitwise exclusive???</td><td></td></tr>
<tr><td>~True OR !True</td><td>not</td><td></td></tr>
</table>

```
Math.pow(2,3)    // 8
Math.abs(-1)                //1
Math.PI                     //3.141592
4!=5 <==> !(4==5)
Integer i = new Integer(4);                    //deprecated, so just Integer i = 4;
int j = i.intValue();
```
To change data type : (double) x --> x is of type double (typecasting)

```
Object o = new Object();
o.equals(o)
o.toString()
o instanceof Class            //boolean of whether obj is of class Class

var a = 7;
var a = "Hello";                //var is like wildcard, java will figure out data type automatically
double value = 1.23456
system.out.printf("%.1f Hello %.2f\n", value, value);    //1.2 Hello 1.23     %f if no change for float
```
%s for strings   %d for digits(%4d to left justify 4 spaces) %b for boolean

String

<table>
<tr><td>String result = "";
String += String.format("%d %s\n", i, array[i]);
String text = new String("Hello");
String text = "Hello"
string1.equalsIgnoreCase(string2)
string.length()
"some string".charAt(3)
String.valueOf(false)</td><td>
//to concatenate strings using string format
//default constructor for class types
//shortcut constructor | only works for string
//Ignore casing of strings
//length of string
//'s'
//"false"   provide string of arg passed</td></tr>
<tr><td>Stringbuilder sb = new Stringbuilder();
sb.append("I am");
sb.append(" 21");
String result = sb.toString();
Stringbuilder sb2 = new Stringbuilder("I ");
sb2.append("am ").append("a ").append("giraffe");</td><td>//using += to concatenate strings will use more memory than stringbuilder as += will store original and new string

//mtd chaining (works cus append return the string itself)</td></tr>
</table>

Subtype

T <: T (reflexive)

if S <: T and T <: U, then S <: U (transitive)

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);
double value = scanner.nextFloat();     //nextLine(), nextInt(), nextDouble()

System.exit(0) //exits programme immediately
```

Memory Usage

<table>
<tr><td>stuff
stack - variables in method/ fn calls ()

free space-
heap space- new instances of classes</td><td>
//primitive types as byte code, objects as location reference.
//whenever a mtd is called, new stack frame is created

//variables of the objects</td></tr>
</table>

if else statements:

int time = 22 if (time < 10) {	Ternary Operators int time = 20

System.out.println('Good morning');
} else if (time < 20) {
System.out.println('Good day');
} else {
System.out.println('Good evening');
}
// Good evening
	String result = (time < 18) ? 'Good morning' : 'Good evening';
	System.out.println(result);
	//Good evening
	
	System.out.println(a ? "yes":"no");
	//if a is true, print yes else print no

```
int option = 1;
switch (option) {                    //switch by default uses .equals() method for strings whereas if else == won't work
case 0:
        System.out.println("Option 0");
        break;
case 1:
        System.out.println("Option 1");
        break;
default:
        System.out.println("Invalid option");
        break;
}
```

<u>Loops</u>

do { System.out.println("Hello"); } while (condition);	//do something first, then check if true

while (condition) {
doSomething()
}
	//check if true, then do something

for(int i, i<N, i++) { }	//for i in range(N)

for (String num: nums) {
System.out.println(num);
}
	//for num in nums

<u>Arrays</u>

import java.utils.Arrays; int[] numbers = {2,3,4,5};	

int min = Integer.MAX_VALUE;
for (int currentValue : numbers) {
min = currentValue < min ? currentValue: min;
}
numbers.length
int[] numbers = new int[3]
String[] numbers = {"one", "two", "three"};
String[][] texts = {
{"one", "two"},
{"three", "four"},
{"five", "six", "seven"},
};
Arrays.toString(numbers);
	// set to largest value that int can hold, MIN_VALUE exist as well
	//find min value in numbers array
	
	//length is a property here
	//reserving space in memory for array of length 3
	
	//Multidimensional arrays
	
	//print out array

<u>Classes</u>

class Person {
private String name;
private static int count = 0;
public Person(String name) {
this.name = name;
Person.count++;
}
public Person () {
this("No name");
}
public void setName(String name) {
this.name = name;
}
	
	//static means associated with class and not with instance, keeps track of how many objects created
	//constructor, optional, can have multiple constructors that takes in diff inputs
	
	//constructor if no name provided
	//use the previous constructor with name as "No name"
	
	//setter

public String getName() {
return name;
}
public static int getCount() {
return Person.count;
}
public void sayHello() {
System.out.println("Hello");
}
public String sumNames(Person p) {
return this.name + p.name;
}
}
	//getter
	
	//static method can only access static variables like count
	
	//method
	
	//even though name is private, as both self and p are in the same class, self can access p name without a getter

public class Banker extends Person {
private String job;
public Banker(String name) {
super(name);
job = "Banker";
@Override
public void sayHello() {
System.out.println("Hello Banker");
}
public void sayHello(String name) {
System.out.println("Hello" + name);
}
}
new Person.sayHello();
super.method()
	//Inheritance where Person is superclass
	
	//Person constructor called first, then banker constructor
	
	//annotation, check we are actually overriding a method(check that such method actually exist in superclass, optional
	
	//method overloading, where methods have same name but diff args (which mtd to call depends on args passed)
	
	Person person1 = newPerson;
	person1.sayHello();
	//to reuse method in superclass in subclass

public class test {
public static void main(string[] args) {
Person person1 = new Person();
person1.setName("John")
person1.name = "Mary";
person1.sayHello();
System.out.println(person1);
Person person2 = new Person("Joe");
System.out.println(Person.getCount());
}
}
	
	//creating instance of a person class(name = "No name")
	
	//Won't work as attribute name is set to private
	//Hello
	//automatically calls toString method(can call person1.toString() if you want
	//instance with name set
	//associated with class, but use getter as count set to private
	//if count set to public static, can just Person.count, or (new Person).count as instance can access static fields

Static & Dynamic Binding

- Method signature: name of mtd and (num, order and type of arguments)
- compile type decides what mtd can be called by instance
- run time type decides which mtd is actually called (if same mtd signature, decide by which mtd args are more specific)

Person person1 = new Banker();
 	
Banker person2 = new Banker();
((Banker)person1).mtd();
	// At compile time, person1 of type Person, hence only can use Person Mtd (but technically Banker mtd work in run time)
	//method in banker but not in person would work
	//now method works as we type cast person1 to banker

Point p = new Point()
Object o = new Object()
o.toString()
o = p
o.toString()
	//uses object to string mtd(run time type is object)
	
	//At compile time, o is of type Object, hence toString mtd of Object is used.
	//However, at run time, type is Point and since Point has a @Override mtd of the same toString mtd in Object, toString mtd in Point is called

ColoredCircle c = new Circle();
 	
Circle c = new ColoredCircle();
	//Error as ColoredCircle <: Circle
	//c compile time type is ColoredCircle, run time type is Circle
	//Works, compile time is Circle, run time is ColoredCircle

Abstract Keyword

public abstract class GameObject {
 	
public abstract void describe();
}
	//cannot be instantiated, but useful to group subobjects in like arrays (opp of Abstract class is Concrete class)
	//forces subclass to implement this mtd
	//can have non abstract mtd

public class Player extend GameObject{
@Override
public void describe() {
}
}
 	
GameObject[] objs = {new Player(), new Monster()}
for (GameObject obj: objs) {
obj.describe();
}
	
	//have to implement this mtd
	
	//use to define type in arrays

Interface //models what an entity can do

interface GetAreable {
public abstract double getArea();
}
	//convention for name to end in -able
	//public abstract(optional as automatically will be as such)
	//to implement mtd, add default keyword(then implementation of mtd in class becomes optional)

abstract class Shape implements GetAreable {
}
OR
class Flat extends RealEstate implements GetAreable {
private int unit;
 	
@Override
public double getArea() { }
}
	//automatically have getArea mtd (abstract class)
	
	//concrete class
	
	//must implement mtd

double findLargest(GetAreable[] array) {
double maxArea = 0;
for (GetAreable curr : array) {
...
}
GetAreable g = findLargest(shapes)
Circle c = findLargest(shapes)
Object o = findLargest(shapes)
	//interface also considered a type
	
	//ok
	//cannot compile
	//ok as any obj in shapes must be a class and hence would be an object

- A class can only extend from one superclass, but it can implement multiple interfaces(A implements I1, I2, I3)
A a = new A(); (can use all mtds in class A)
I1 a = new A(); (can only use mtd in interface I1) (have to type cast if want to use other mtds)
- An interface can extend from one or more other interfaces, but an interface cannot extend from another class.
- Abstract class more for common root class, interface for common mtds

Variance
Let C(T) be a complex type based on type T (e.g. T[])
- covariant if S <: T implies C(S) <: C(T)
- contravariant if T <: S implies C(S) <: C(T)
- invariant if neither covariant nor contravariant

Exceptions / Errors

public void someMtd throws <Exception> {
if () { throw new Exception('msg'); }
}
	//when u want program to stop executing (throws E1, E2, E3 if multiple exceptions)
	//if mtd is called in another mtd, another mtd also need to throws

try { ... }
catch(<Exception> e) {
System.err.println("..." + e);
}
finally {
scanner.close();
}
	//for program to run differently with errors
	
	//can catch multiple; catch(E1 | E2 | E3 e)
	//print in error stream instead of normal console output??

class MyException extends Exception {
public MyException(String msg) {
super(msg);
}
@Override
public String toString...
}
	//Own exception

//usually throw exception somewhere, then have to catch it from elsewhere and handle it
//if throw RuntimeException (then no need throws) but will still be caught by catch statement
//unchecked exception are runtime exception caused by programmer, RuntimeException
//checked exception are exception programmers anticipate would happen and would catch them in advance
NullPointerException //when variable is referencing a null object (forget to initialise)
AutoCloseable //interface for classes that need to be closed(e.g. scanner)

public class Database implements AutoCloseable {
    public Database(..) throws Exception
    public void getData() throws Exception
    @Override
    public void close() throws Exception
}
	
	//constructor
	//mtd
	
	//close mtd

public static void main(String[] args) {
    try(Database db = new Database(..)) {
        db.getData();
    }
    catch (Exception e) {
        System.out...
    }
	//try with resources (auto call close mtd even if exceptions in getData)
	
	//any exceptions in try block will be caught here including those in close mtd

<u>Generics</u>

- for more general classes that can store different objects like arrays
- However, we want to specify what the class can store, hence we use <>, <S>, <S, T>
- Type Erasure: The type specified in <> is only used to check type during compile time
- However if type is bounded(S extends GetAreable), after type erasure, would become GetAreable

e.g. Cat <: Mammal <: Creature

public class Cat extends Mammal {
  public Cat(String name) { super(name); }
}
	
	//constructor no need <>, only in class

public class Wrapper {
  private Object obj;
  public void set(Object obj)
  public Object get()
	public class Wrapper<S> {
	  private S obj;
	  public void set(S obj)
	  public S get()

Wrapper wrapper = new Wrapper();
Cat cat = new Cat("Joe");
wrapper.set(cat);
Cat retrieved = (Cat)wrapper.get();
//not ideal as we cannot be certain only cats obj are in wrapper
	Wrapper<Cat> wrapper = new Wrapper<>();
	wrapper.set(new Cat("Joe"));
	wrapper.set(new Mammal("...")) ;
	//won't work as wrapper only accept Cat obj
	Cat retrieved = wrapper.get();
	//no need to type cast

	Wrapper<Mammal> wrapper ...//can set both cat and mammal objs as Cat <: Mammal

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

  Array(int size) {                              //Unchecked Warnings
    // 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")
    T[] a = (T[]) new Object[size];
    this.array = a;                              //cannot this.array = new T[size];
  }

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

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

class DictEntry<T> extends Pair<S, T>	DictEntry<T> is of single type parameter T that extends from Pair<S, T>, where T from DictEntry<T> is passed as the type argument for T of Pair<String,T>

public static <T> boolean contains(T[] array, T obj) {
  for (T curr : array) {
    if (curr.equals(obj)) { return true; }
  }
  return false;
}
	//Generic mtd
	//<T> is to specify what type to scope for
	A.<String>contains(strArray, 123); // type mismatch error
	A.<String> contains(strArray, 'a'); // works

```
//However generic types do not follow polymorphism
//feedAll(mammals) works but feedAll(cats) would not as although Cat <: Mammal, Array<Cat> !<: Array<Mammal>
//Contrast this to wrapper class where we can add Cat to a <Mammal> wrapper as set mtd is set(S obj) not in <>???
public static void feedAll(Array<Mammals> mammals)
```

```
//Wildcards
public static void feedAll(Array<? extend Mammals> mammals) //now feedAll would work for cats and mammals
A.<Shape>contains(shapeArray, circle); // ok
A.<Circle>contains(shapeArray, circle); // compilation error
A.<Shape>contains(circleArray, shape); // compilation error
A.<Circle>contains(circleArray, shape); // compilation error
```

```
public static <S, T extends S> boolean contains(Array<T> array, S obj) { .. }
A.<Shape,Circle>contains(circleArray, shape)  //ok
```

```
public static <T> T findLargest(T[] array) {
      double maxArea = 0;
      T maxObj = null;
      for (T curr : array) { ... }
      return maxObj;
   }
//wont compile as T might not have mtd getArea
```

```
//Bounded type
public static <T extends GetAreable> T findLargest(T[] array) {
      double maxArea = 0;
      T maxObj = null;
      for (T curr : array) { ... }
      return maxObj;
   }
```

```
public void copyFrom(Array<? extends T> src)   //can copy from a subtype array to self array of type T
public void copyTo(Array<? super T> dest)        //can copy to an array with type that is a superclass of self of type T
Producer Extends; Consumer Super (PECS)
```

<u>Type Inference</u>

```
Pair<String,Integer> p = new Pair<>()
A.contains(circleArray, shape);          //no need A<shape>.contains... as already stated in mtd and java can infer
//In inferencing, java would look for most specific class that satisfies, if not would be object
```

void foo(List<Integer> integers) {} void foo(List<String> strings) {}	same mtd signature after type erasure, cannot compile

class B<T> {
static T y;
T x;
}
	//cannot compile as static cannot access T
	//if we initialise B<String> and B<Integer>, how to know what is static T? so can't compile

class C<T> {
static int b = 0;
C() {
this.b++;
}
C<Integer> x = new C<>();
C<String> y = new C<>();
System.out.println(x.b);
System.out.println(y.b);
	//2,
	//2, as both C after type erasure is same class

static <T extends Comparable<T>> T max3(T[] arr) {
T max = arr[0];
if (arr[1].compareTo(max) > 0) {
max = arr[1];
}
if (arr[2].compareTo(max) > 0) { max = arr[2]; }
return max;
}
	static <T> Comparable<T> max3(Comparable<T>[] arr)
	// Mtd now returns a Comparable<T> obj. If we change type of max to Comparable<T>, then need to typecast the arg of compareTo mtd as it expects an arg of type T, e.g. arr[1].compareTo((T)max)
	
	static <T> T max3(Comparable<T>[] arr)
	//something of Comparable<T> might not be of type T, have to type cast to T
	
	static Comparable max3(Comparable[] arr)
	//have to type cast

class Fruit implements Comparable<Fruit> { } class Orange extends Fruit { } 1. static <T extends Comparable<T>> T max3(List<T> list)	// would work for List<Fruit> only, but not for List<Orange>, since Orange extends Comparable<Orange> does not hold.

2. static <T extends Comparable<T>> T max3(List<? extends T> list)
If T = Orange. <T extends Comparable<T>> would not work for List<Orange> but Orange <: Comparable<Orange> does not hold.
As although Orange <: Fruit <: Comparable<Fruit>, but Comparable<Fruit> and Comparable<Orange> are invariant.
If T = Fruit. Fruit extends Comparable<Fruit> holds. And is List<Orange> a sub-type of List<? extends Fruit>? Yes! This is a covari- ant relation. Thus, it is possible to put List<Orange> as an argument for max3 in this case because
List<Orange> <: List<? extends Fruit> and Fruit extends Comparable<Fruit>.

3. static <T extends Comparable<? super T>> T max3(List<T> list)
Orange <: Fruit <: Comparable<Fruit> <: Comparable<? super Orange> So T can be bounded to Orange! In this case,
Comparable<Fruit> <: Comparable<? super Orange> is contravariant.

4. static <T extends Comparable<? super T>> T max3(List<? extends T> list)	
void foo(List<?> list) { } foo(new ArrayList<String>());	Ok, since ArrayList<String> <: List<String> <: List<?>
void foo(List<? super Integer> list) { } foo(new List<Object>());	No, List is an interface. Ok if change to ArrayList<Object> since ArrayList<Object> <: List<Object> <: List<? super Object> <: List<? super Integer>
void foo(List<? extends Object> list) { } foo(new ArrayList<Object>());	Ok, since ArrayList<Object> <: ArrayList<? extends Object> <: List<? extends Object>
void foo(List<? super Integer> list) { } foo(new ArrayList<int>());	Error. A generic type cannot be primitive type.
void foo(List<? super Integer> list) { } foo(new ArrayList());	Compiles, but with an unchecked conversion warning. The use of raw type should also be generally be avoided.

<u>Enum</u>

public enum Fruit {
APPLE, BANANA, ORANGE
}
Fruit fruit1 = Fruit.APPLE;
Fruit fruit2 = Fruit.ORANGE;
Fruit[] fruits = Fruit.values();
for (Fruit fruit: fruits) {System.out.println(fruit)};
System.out.println(Fruit.BANANA.ordinal())
Fruit o = Fruit.valueOf("ORANGE");
System.out.println(ORANGE == ORANGE)
	
	//output enum values
	1
	ORANGE (type is now enum Fruit)
	//true (can use == for enums)

public enum Fruit {
APPLE("red"), BANANA("yellow"), ORANGE("orange");
private String color;
 	
Fruit(String color) {
this.color = color
}
}
	
	//enum constructor cannot have access modifier

switch(Fruit) { case APPLE: case ORANGE: case BANANA:	

static {
APPLE.color = "red";
Orange.color...
}
	//static constructor

<u>Final</u>

final class ImmutableArray<T> {
private final T[] array;
 	
// Only items of type T goes into the array.
@SafeVarargs
public static <T> ImmutableArray<T> of(T... items) {
return new ImmutableArray<>(items);
}
private ImmutableArray(T[] a) {
this.array = a;
}
 	
public T get(int index) {
return this.array[index];
}
}
	//final class cannot be inherited from
	//final attributes cannot be changed
	
	//when we need to use ... (explain & add anotation)
	//... -> variable num of args of type T
	
	//actual constructor
	
	ImmutableArray<Integer> a;
	a = ImmutableArray.of(1, 2, 3);
	a = ImmutableArray.of(1, 2, 3, 4, 5);

<u>Nested Class</u>

class A {
private int x;
static int y;
private class B {
A.this.x = 1;
	//class B can still access private attributes of class A
	
	//so class B cannot be used or accessed outside of class A
	//ok (if this.x ==> this is referring to B, so won't work)

y = 1;
}
static class C {
x = 1;
y = 1;
	//ok
	
	//cannot
	//ok

-Static nested class can only access static fields and static mtds -Non static nested class can access all fields and mtds and is known as inner class -classes defined within a mtd(within {}) are known as local class and can access all fields and mtds - local class can only access variables that are declared final, or does not change after initialisation (effectively final)	

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;
}
}
	Recall that when a method returns, all local variables of the methods are removed from the stack. But, an instance of that local class might still exist
	A a = new A();
	C b = a.f();
	b.g();
	will give us a reference to an obj of type B now. But, if we call b.g(), what is the value of y?
	Hence, even though a local class can access the local variables(B access y), the local class makes *a copy of local variables* inside itself (*capture* the variables)
	This ties it to effectively final, where does the local class capture the variable that was declared initially?, or the value that is changed afterwards? Hence, compiler wouldn't allow capture of not final var

Anonymous Class

void sortNames(List<String> names) {
class NameComparator implements Comparator<String> {
public int compare(String s1, String s2) { ... } }
names.sort(new NameComparator());
}
	//Comparator: interface: X
	names.sort(new Comparator<String>() {
	public int compare(String s1, String s2) {
	return s1.length() - s2.length();
	}
	});

-does not have a name, have format of new X (arguments) { body } -declare it and instantiate it in a single statement • *X* is a class that the anonymous class extends or an interface that the anonymous class implements. X cannot be empty. This also implies an anonymous class cannot extend another class and implement an interface at the same time. Furthermore, an anonymous class cannot implement more than one interface. • *arguments* are the args that you want to pass into the constructor of the anonymous class. If anonymous class is extending an interface, then there is no constructor, but we still need (). • *body* is the body of the class as per normal, except that we cannot have a constructor for an anonymous class.	

Comparator<String> cmp = new Comparator<String>() {
public int compare(String s1, String s2) { ... } };
names.sort(cmp);
	//same as above just giving it a name

Functions & Functional Programming(Only use pure fn)

Pure fn - get same output from input, does not print to screen, write to files, throw exceptions, change other variables, modify the values of the arguments. That is, a pure function does not cause any *side effect*.	
int add(int i, int j) { return i + j; }	//pure

int div(int i, int j) { return i / j; }
int incrCount(int i) { return this.count + i; }
void incrCount(int i) { this.count += i; }
int addToQ(Queue<Integer> queue, int i) { queue.enq(i); }
	// may throw an exception
	// assuming count is not final, diff i, diff result
	// does not return a value and has side effects on count
	// has side effects on queue

Lambda Expression

@FunctionalInterface	//use if interface only has one abstract mtd

interface Transformer<T, R> {
R transform(T t);
}
	interface Executor {
	int execute(int x, int y);
	}

Transformer<Integer, Integer> square = new Transformer<>() {
@Override
public Integer transform(Integer x) {
return x * x;
}
};
Transformer<Integer, Integer> incr = new Transformer<>() {
@Override
	Transformer<Integer, Integer> square =
	(x) -> { return x * x; };
	Transformer<Integer, Integer> incr =
	(x) -> { return x + 1; };
	
	//cus only one mtd to override, we dont need to explicitly state it
	//if no parameter, change (x) to ()

```
  public Integer transform(Integer x) {
    return x + 1;
  }
};
```

```
//Since body {} only has one return statement
Transformer<Integer, Integer> square = x -> x * x;
Transformer<Integer, Integer> incr = x -> x + 1;
square.transform(3)     //9
```

Suppose we already have an existing mtd that we want to use

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = new Transformer<>() {
   @Override
   public Double transform(Point p) {
      return origin.distanceTo(p);
   }
}
//Note as in captured variable, variable must be final or effectively final such as origin
//Lambda as Closure. Lambda hence can stores data from the environment where it is defined without ambiguity.
```

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

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

:: can be used for
(i) static method in a class, Class::mtd
(ii) instance method of a class or interface, Obj::mtd
(iii) constructor of a class, Class::new

So far, the lambda fn only accept one parameter, to accept more...aka curried fn

```
int add(int x, int y) {
  return x + y;
}
```

```
Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);
add.transform(1).transform(1);
//in python add = lambda x: lambda y: x+y
//so add.transform(1) is lambda y: 1+y
```

```
Executor exe = (x, y) -> x+y;
exe.execute(1, 1);
//in python exe = lambda x, y: x+y
```

-lazy evaluation: dont evaluate certain statement until required which is accomplished by lambda expressions
-can use memoization by having boolean evaluated and value to store evaluated value in a class

<u>Filter & Map</u>

-predicate interface, import java.util.function.Predicate;

Anonymous class

```
Predicate<String> p = new Predicate<> () {
   public boolean test(String s) { return s.length() < 4; }
};
```

Lambda expression

```
Predicate<String> p = s -> s.length() < 4;

System.out.println(p.test("abc"));              //true
```

```
List<Integers> numbers = new ArrayList<> (List.of(2,8,7,4));
numbers.removeIf(n -> n < 5);
numbers.forEach(System.out::println);
//forEach is meant to be used by stream so shld be
//numbers.stream().forEach(...) to convert to stream
//But forEach has been modified to work with list as well
```

removeIf expects a predicate to be passed as argument
8
7

```
numbers.replaceAll(n -> n * n);
numbers.forEach(System.out::println);
```

//mapping

<u>Optional</u>

- mtds may return object or null reference
- to handle this, we use class that internalise checks if return is an object or null

```
class Box<T> {
  private T item;
  public <U> Box<U> map(Transformer<? super T, ? extends U> transformer) {
    if (!isPresent()) { return empty(); }
    return Box.ofNullable(transformer.transform(this.item));
  }
  public Box<T> filter(BooleanCondition<? super T> condition) {
    if (!isPresent() || !(condition.test(this.item)) { return empty(); }
    return this;
  }
}
```

//check if obj is present

<u>Infinite List</u>

```
class EagerList<T> {
  private final T head;
  private final EagerList<T> tail;
  private static EagerList<?> EMPTY = new EmptyList();
```

//EagerList evaluation(generate whole list)
//List would be like [head, [tail, [tail*2, [tail*3...]]]
//Eager not ideal as won't stop if we try to generate an infinite list

public EagerList(T head, EagerList<T> tail) {
this.head = head;
this.tail = tail;
}
public T head() { return this.head; }
public EagerList<T> tail() { return this.tail; }
 	
public static <T> EagerList<T> empty() {
@SuppressWarnings("unchecked")
EagerList<T> temp = (EagerList<T>) EMPTY;
return temp;
}
public static <T> EagerList<T> generate(T t, int size) {
if (size == 0) { return empty(); }
return new EagerList<>(t, generate(t, size - 1));
}
public static <T> EagerList<T> iterate(T init, BooleanCondition<? super T> cond, Transformer<? super T, ? extends T> op) {
if (!cond.test(init)) { return empty(); }
return new EagerList<>(init, iterate(op.transform(init), cond, op));
}
 	
public <R> EagerList<R> map(Transformer<? super T, ? extends R> mapper) {
return new EagerList<>(mapper.transform(this.head()), this.tail().map(mapper));
}
public EagerList<T> filter(BooleanCondition<? super T> cond) {
if (cond.test(this.head())) {
return new EagerList<>(this.head(), this.tail().filter(cond));}
return this.tail().filter(cond);}
	//1st value in list
	
	//create empty list
	
	//generate list with same elements
	
	//generate list with elem changing according to *op* and stop when *cond* is false
	//recursively generate element in list
	
	//map
	//Transformer<consumer, producer>
	
	//filter
	BooleanCondition<consumer>

private static class EmptyList extends EagerList<Object> {
EmptyList() { super(null, null); }
@Override
public Object head() ...tail{ throw new java.util.NoSuchElementException(); }
@Override
map ...filter
}
	//static nested class
	
	//both head and tail mtd in EmptyList return the same thing
	//mtd signature same as superclass, just return empty();

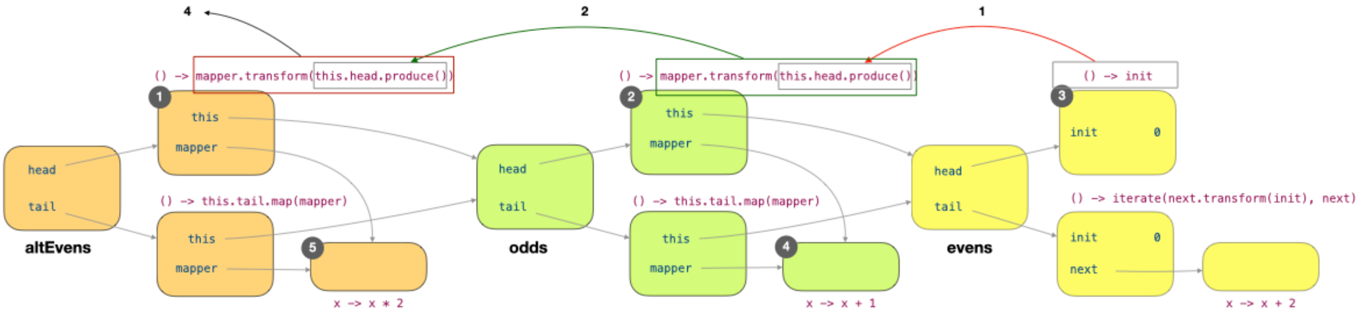
EagerList<Integer> l = EagerList.iterate(1, i -> i < 10, i -> i + 1)
.filter(i -> i % 3 == 0)
.map(i -> i * 2);
l.head();
l.tail().head();
l.tail().tail().head();
	// [1, ... 9]
	// [3, 6, 9]
	// [6, 12, 18]
	// 6
	// 12
	// 18

class InfiniteList<T> {
private Producer<T> head;
private Producer<InfiniteList<T>> tail;
 	
public InfiniteList(Producer<T> head, Producer<InfiniteList<T>> tail) {
this.head = head;
this.tail = tail;
}
public T head() { return this.head.produce(); }
public InfiniteList<T> tail() { return this.tail.produce(); }
 	
public static <T> InfiniteList<T> generate(Producer<T> producer) {
return new InfiniteList<T>(producer, () -> generate(producer));
}
public static <T> InfiniteList<T> iterate(T init, Transformer<T, T> next) {
return new InfiniteList<T>(() -> init, () -> iterate(next.transform(init), next));
}
public <R> InfiniteList<R> map(Transformer<? super T, ? extends R> mapper) {
return new InfiniteList<>(() -> mapper.transform(this.head()),
() -> this.tail().map(mapper));}}
	LazyList evaluation(only generate elements in list when required)
	//EmptyList not needed unless truncating list to a finite one
	//Producer just another functional interface like Transformer
	
	//actually producing value
	//actually producing value
	
	//only putting producer in head and tail
	
	//no cond as infinite list
	//only putting producer in head and tail

InfiniteList<Integer> ones = InfiniteList.generate(() -> 1); InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2);	// 1, 1, 1, 1, ... // 0, 2, 4, 6, ...

<table>
<tr><td>evens.head();
evens.tail().head();
odds = evens.map(x -> x + 1);
altEvens = odds.map(x -> x * 2);
altEvens.head() || evens.map(x -> x + 1).map(x -> x * 2).head();</td><td>// 0
// 2
//actual list not produced yet

// 2 (actual elem produced)</td></tr>
</table>



Evens is an instance of InfiniteList, with two fields, head and tail, each pointing to an instance of Producer<T>. The two instances of Producer<T> capture the variable init. The tail additionally captures the variable next, which itself is an instance of Transformer<T,T>.
Odds is an instance of InfiniteList, with two fields, head and tail, each pointing to an instance of Producer<T>. The two instances of Producer<T> capture the local variable this and mapper of the method map. mapper refers to an instance of Transformer<T, T>. Since the method map of *evens* is called, the this reference refers to the object *evens*.
1) () -> mapper.transform(this.head.produce())
2) () -> mapper.transform(this.head.produce())
3) () -> init
4) x -> x+1
5) x -> x*2
When we call *altEvens*.head() => this.head().produce() is invoked, where this refers to *altEvens*, which invokes 1).
This leads to this.head.produce() in 1) being called, where this refers to *odds*, which invokes 2)
Now, this refers to *evens*, and this.head.produce() invokes 3) to produce 0.
Result of 3) which is 0 is return to 2) where mapper refers to 4). This returns the value 1 and now 2) has value of 1.
Result of 2) which is 1 is return to 1) where mapper refers to 5). This returns the value 2, *altEvens*.head() = 2

<table>
<tr><td>public InfiniteList<T> filter(BooleanCondition<? super T> cond) {
  if (cond.test(this.head())) {
    return new InfiniteList<>(this.head, () -> this.tail().filter(cond));
  }
  return this.tail().filter(cond);
}</td><td>// wrong version of filter
//immediately produce value of head

//immediately produce tail</td></tr>
<tr><td>public InfiniteList<T> filter(BooleanCondition<? super T> cond) {
  Producer<T> newHead = () -> cond.test(this.head()) ? this.head() : null;
  return new InfiniteList<>(newHead, () -> this.tail().filter(cond));
}

public T head() {
  T h = this.head.produce();
  return h == null ? this.tail.produce().head() : h;
}

public InfiniteList<T> tail() {
  T h = this.head.produce();
  return h == null ? this.tail.produce().tail() : this.tail.produce();
}</td><td>//better version of filter
//filter in instance of Producer, not ideal as can return null

//new head mtd to adapt to filter

//if h == null, get next value

//new tail mtd to adapt to filter

//if h == null, ignore this head and get next tail</td></tr>
<tr><td colspan="2">To improve, use Lazy<Maybe<T>> for head (Lazy<T> uses memoization, Maybe<T> can refer to null or actual object)</td></tr>
</table>

<u>Streams</u>

<table>
<tr><td>- Stream is an interface and is lazy just like InfiniteList
- Limitation of stream is that it can only be used once unlike InfiniteList
Stream<Integer> s = Stream.of(1,2,3);
s.count();
s.count();</td><td>

//3
//IllegalStateException</td></tr>
<tr><td colspan="2">- We can use the static factory method of (e.g., Stream.of(1, 2, 3))
- We can use the generate and iterate methods (similar to our InfiniteList)
- We can convert an array into a Stream using Arrays::stream
- We can convert a List instance (or any Collection instance) into a Stream using List::stream</td></tr>
</table>

- Intermediate op are lazy(map, filter, flatMap(map then flatten))
Stream.of(1, 2, 3).forEach(System.out::println);
Stream.generate(() -> 1).forEach(System.out::println);
Stream.of("hello\nworld", "ciao\nmondo", "Bonjour\nle monde").map(x -> x.lines())
Stream.of("hello\nworld", "ciao\nmondo", "Bonjour\nle monde", "Hai\ndunia")
.flatMap(x -> x.lines())
names.stream.map(s -> new Person(s)).forEach(System.out::println)
OR names.stream.map(Person::new).forEach(System.out::println)
	
	// infinite loop
	// returns a stream of streams
	// return a stream of strings
	
	//print out Person object created from each name

- Some intermediate op are stateful -- they keep track of states(e.g. sorted, distinct) to operate
- sorted returns a stream with elements sorted. By default, sorts according to natural order by implementing the Comparable interface, else pass in own Comparator
- distinct returns a stream with only distinct elements in the stream.
- distinct and sorted are also known as bounded operations, since they should only be called on a finite stream

There are several intermediate operations that convert from infinite stream to finite stream:

- limit takes in an int n and returns a stream containing the first n elements of the stream;
- takeWhile takes in a predicate and returns a stream containing the elements of the stream, until the predicate becomes false. The resulting stream might still be infinite if the predicate never becomes false.

```
Stream.generate(() -> "hello").limit(3).forEach(System.out::println)      //limit infinite list to print hello 3 times
Stream.iterate(0, x-> x+1).takeWhile(x -> x<5);                            //create a (lazy) finite stream of elements 0 to 4
```

Another intermediate operation is peek, peek takes in a Consumer, and apply a lambda on a "fork" of the stream

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

- reduce aka fold/accumulate Stream.of(1, 2, 3).reduce(0, (x, y) -> x + y); Stream.of(1, 2, 3).reduce((x,y) -> x+y); IntStream.range(1, 4).sum();	 //6, sum all elements //6, start with 1st element in stream //6, same as above

- Terminal operation is an operation that triggers the evaluation of the stream
  - noneMatch returns true if none of the elements pass the given predicate.
  - allMatch returns true if every element passes the given predicate.
  - anyMatch returns true if at least one element passes the given predicate.

```
boolean isPrime(int x) {
  for (int i = 2; i <= x-1; i++) {
    if (x % i == 0) {
      return false;
    }
  }
  return true;
}
```

```
boolean isPrime(int x) {
  return IntStream.range(2, x)
      .noneMatch(i -> x % i == 0);
}
IntStream.iterate(2, x -> x+1)
    .filter(x -> isPrime(x))
    .limit(500)
    .forEach(System.out::println);
```

```
var result = Stream.of("one", "two", "three", "four")
            .collect(                                      //collect puts all element into a data structure
              () -> new ArrayList<String>(),               //initialise the data structure
              (list, item) -> list.add(item),              //what to do with each element
              (list1, list2) -> list1.addAll(list2));      //only useful when doing parallel threads (add result of diff threads tgt)
System.out.println(result);                                [one, two, three, four]
```

Priority Queue

```
Queue<String> testStringsPQ = new PriorityQueue<>();
testStringsPQ.add("abcd");
testStringsPQ.add("1234");
testStringsPQ.poll();                  //pop first item
testStringsPQ.peek()                   //return first item without popping
```

- To specify how to arrange the order in the queue, need to use comparator interface

```
public class CustomerOrder implements Comparable<CustomerOrder> {
   private int orderId;
  @Override                                          //o.orderId < this.orderId ? 1: -1 for ascending order
   public int compareTo(CustomerOrder o) {           //descending order
      return o.orderId > this.orderId ? 1 : -1;
   }
```

Monads

1. Identity Law (let Monad be a class and monad be an instance of it.)

The left identity law says:

- Monad.of(x).flatMap(x -> f(x)) must be the same as f(x)
- Monad.of(4).flatMap(x -> incrWithLog(x)) == incrWithLog(4) == Monad<Integer>(5, "incr 4;")

- Applying a method on a Monad.of == just calling the method

The right identity law says:

- monad.flatMap(x -> Monad.of(x)) must be the same as monad
- Applying the static factory method of on itself(instance) would just return itself

2. Associativity Law

- monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x).flatMap(y -> g(y)))
- regardless of the grouping, end result is the same

Counter Example
public static <T> Loggable<T> of(T value) {
  return new Loggable<>(value, "Logging starts: ")
}
	//all new Loggable message start with "Logging starts: "
	//However, chaining 2 mtd now would return "Logging starts:... Logging starts..." which is not what we want

Functor is simpler than monad in that it ensures lambdas can be applied sequentially to the value, without worrying about side info.
A functor needs to adhere to two laws:

- preserving identity: functor.map(x -> x) is the same as functor
- preserving composition: functor.map(x -> f(x)).map(x -> g(x)) is the same as functor.map(x -> g(f(x)).

Our classes from cs2030s.fp, Lazy<T>, Maybe<T>, and InfiniteList<T> are functors as well.

Parallel Streams

A single-core processor can only execute 1 instruction/*process*/application at any one time. The operating system switches btw diff processes, giving the illusion that they are running at the same time. For program to run *concurrently* -- divide computation into subtasks called *threads*. Such multi-thread programs: (i) separate unrelated tasks into threads, and write each thread separately; (ii) improves utilization of processor.

Parallel computing refers to scenario where multiple subtasks are truly running at the same time -- either have a processor that is capable of running multiple instructions at the same time, or have multiple cores executing diff tasks at the same time.
All parallel programs are concurrent, but not all concurrent programs are parallel.

IntStream.range(2_030_000, 2_040_000)
    .filter(x -> isPrime(x))
    .forEach(System.out::println);
	IntStream.range(2_030_000, 2_040_000)
	    .filter(x -> isPrime(x))
	    .parallel()
	    .forEach(System.out::println);

With .parallel(), the output has been reordered, although same set of numbers are produced. There is no coordination among the parallel tasks on order of printing => whichever parallel tasks that complete first will output the result to screen first
To produce the output in order of input, use forEachOrdered instead of forEach, but will lose some benefits of parallelization

//parallel() is a lazy op, merely marks the stream to be processed in parallel. Hence, can insert parallel() anywhere
//sequential() marks the stream to be process sequentially. If you call both parallel() and sequential(), last call "wins".
s.parallel().filter(x -> x < 0).sequential().forEach(..); //process sequentially
//Another way to create a parallel stream is to use parallelStream() instead of stream() of the Collector class.

IntStream.range(2_030_000, 2_040_000)
    .filter(x -> isPrime(x))
    .parallel()
    .count();
	Produces the same output regardless of parallelized or not.
	Note that code is *stateless*, does not produce *side effects* and each element does not *interfere* with other elements. Such computation is aka *embarrassingly parallel*. Only communication needed btw subtasks is combining result of count() into final count

Stream.generate(scanner::nextInt)
  .map(i -> i + scanner.nextInt())
  .forEach(System.out::println)
	A *stateful* lambda is one where result needs info on previous or next elements.
	The generate and map ops are stateful, as they depend on state of standard input.
	Parallelizing this may lead to incorrect output.

List<Integer> list = new ArrayList<>(Arrays.asList(1,9,17));
List<Integer> result = new ArrayList<>();
list.parallelStream()
  .filter(x -> isPrime(x))
  .forEach(x -> result.add(x));
	Side-effects can lead to incorrect results in parallel execution.
	The forEach lambda generates a side effect -- it modifies result. ArrayList is a non-thread-safe data structure. If two threads manipulate it at the same time, an incorrect result may result.

list.parallelStream()
  .filter(x -> isPrime(x))
  .collect(Collectors.toList())
	There are two ways to resolve this. One, we can use the .collect mtd

List<Integer> result = new CopyOnWriteArrayList<>();
list.parallelStream()
  .filter(x -> isPrime(x))
  .forEach(x -> result.add(x));
	Second, we can use a thread-safe data structure. Java provides several in java.util.concurrent package, including CopyOnWriteArrayList.

List<String> list = new ArrayList<>(List.of("Luke", "Leia", "Han")); list.stream() .peek(name -> { if (name.equals("Han")) { list.add("Chewie"); } }) .forEach(i -> {});	*Interference* means that one of the stream op modifies the source of the stream during execution of the terminal op. The code would cause ConcurrentModificationException to be thrown. Note that the non-interference rule applies even if we use stream() instead of parallelStream().

Stream.of(1,2,3,4) .reduce(1, (x, y) -> x * y, (x, y) -> x * y); - i * 1 equals i - (x * y) * z equals x * (y * z) - u * (1 * t) equals u * t	The reduce op is inherently parallelizable, as we reduce each sub-stream and then use the combiner to combine the results. However, to run reduce in parallel: - combiner.apply(identity, i) must be equal to i. - combiner and accumulator must be *associative* -- the order of applying must not matter. - combiner and accumulator must be *compatible* -- combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)

Whether stream elements are *ordered* or *unordered* also affect the performance of parallel stream operations.
- Ordered: Streams created from iterate, ordered collections (List or arrays), from of,
- Unordered: Stream created from generate or unordered collections (Set)
- Stream op that preserve *encounter order* (op is stable) : distinct and sorted
- parallel of findFirst, limit, and skip can be expensive for ordered stream, as it needs to coordinate btw streams to maintain order

Stream.iterate(0, i -> i + 7) .parallel() .unordered() .limit(10_000_000) .filter(i -> i % 64 == 0) .forEachOrdered(i -> { });	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.

Synchronous Programming
If method is still executing, rest of program stalls. Only after method returns can execution of rest of program continue.
We say that method *blocks* until it returns. Such a programming model is known as *synchronous programming*.
This is not very efficient, especially when there are frequent method calls that block for a long period

Threads
One way to overcome this is to use *threads*. A thread is a single flow of execution in a program.

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();
	new Thread(..) is the usual constructor
	The constructor takes a Runnable instance as an arg. A Runnable is a functional interface with a method run() that takes in no param and returns void.
	With each Thread instance, we run start(), which runs the given lambda expression. Note that start() returns immediately. It *does not return* only after lambda expression completes its execution. This is known as *asynchronous* execution.
	The two threads now run in two separate sequences of execution. The operating system has a scheduler that decides which threads to run when, and on which core/processor. Hence, different output everytime program is run
	Optional 2nd arg in {} for giving name to the thread

Instance mtd getName() : name of a thread
Class method Thread.currentThread() :reference of the current running thread.
All code will print name of the thread called main, which is a thread created automatically every time class method main() is run

Stream.of(1, 2, 3, 4) .parallel() .reduce(0, (x, y) -> { System.out.println(Thread.currentThread().getName()); return x + y; });	main ForkJoinPool.commonPool-worker-5 ForkJoinPool.commonPool-worker-5 ForkJoinPool.commonPool-worker-9 ForkJoinPool.commonPool-worker-3 ForkJoinPool.commonPool-worker-3 This shows 4 concurrent threads running (including main).
Stream.of(1, 2, 3, 4) .reduce(0, (x, y) -> { ...});	Without .parallel(), only main is printed, showing the reduction being done sequentially in a single thread.

Thread findPrime = new Thread(() -> {
System.out.println(Stream...);
});
findPrime.start();
while (findPrime.isAlive()) {
try {
Thread.sleep(1000);
System.out.print(".");
} catch (InterruptedException e) {
System.out.print("interrupted");}}
	Using sleep can cause the current execution thread to pause execution immediately for a given period (in milliseconds). After the sleep timer is over, the thread is ready to be chosen by the scheduler to run again.
	The following code prints a "." on-screen every second while another expensive computation is running.
	
	In this e.g., we use Thread.sleep() to pretend findPrime is a expensive computation
	- isAlive() is used to periodically check if another thread is still running.
	- The program exits only after all threads created are done

Asynchronous programming

E.g. To execute a series of tasks (A, B, C, D, E) *concurrently* where next task depend on result of previous task, we could put the tasks into separate threads. If one of the task have exception, the tasks not dependent on it should still complete.
Problems 1) no methods in Thread that return value 2) cannot specify the execution order 3) possibility of exceptions in the tasks 4) should reuse resources to create another Thread when one Thread is done

int foo(int x) {	If might have null value, use Maybe

```
  int a = taskA(x);
  int b = taskB(a);
  int c = taskC(a);
  int d = taskD(a);
  int e = taskE(b, c)
  return e; }
```

- For lazy computation, use Lazy<Integer>

```
Maybe<Integer> foo(int x) {
  Maybe<Integer> a = Maybe.of(x);
  Maybe<Integer> b = a.flatMap(i -> taskB(i));
  Maybe<Integer> c = a.flatMap(i -> taskC(i));
  Maybe<Integer> d = a.flatMap(i -> taskD(i));
  Maybe<Integer> e = b.combine(c, (i, j) -> taskE(i, j));
    return e; }
```

```
CompletableFuture<Integer> foo(int x) {
  CompletableFuture<Integer> a = CompletableFuture.completedFuture(x);
  CompletableFuture<Integer> b = a.thenComposeAsync(i -> taskB(i));
  CompletableFuture<Integer> c = a.thenComposeAsync(i -> taskC(i));
  CompletableFuture<Integer> d = a.thenComposeAsync(i -> taskD(i));
  CompletableFuture<Integer> e = b.thenCombineAsync(c, (i, j) -> taskE(i, j));
  return e; }
```

Can run concurrently.
We can then run foo(x).get() which will wait for all concurrent tasks to complete and return the result.
CompletableFuture<T> is a monad that encapsulates a value that is either there or not there *yet*. aka promise in other languages

Key property of CompletableFuture is whether the value it promises is ready -- i.e., the tasks that it encapsulates has *completed* or not.
To create a CompletableFuture<T> instance:

- use completedFuture method. Equivalent to creating a task that is already completed and return us a value.
- runAsync method that takes in a Runnable lambda expression. runAsync has return type of CompletableFuture<Void>. The returned CompletableFutureinstance completes when the given lambda expression finishes.
- supplyAsync method that 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.

To create a CompletableFuture that relies on other CompletableFuture instances, use allOf or anyOf mtds for this.
- allOf: completed only when all the given CompletableFuture completes.
- anyOf: completed when any one of the given CompletableFuture completes.

To run given lambda expression in same thread as caller	For asynchronous version (run in diff thread for more concurrency)
- thenApply == map	- thenApplyAsync
- thenCompose == flatMap	- thenComposeAsync
- thenCombine == combine	- thenCombineAsync

After we have set up all tasks to run asynchronously, we can call get() to get the result.
get() is a synchronous call, i.e., it blocks until CompletableFuture completes, hence only call get() as the final step
The method CompletableFuture::get throws a couple of checked exceptions: InterruptedException(thread interrupted) and ExecutionException (errros/exceptions during execution), which we need to catch and handle.
An alternative to get() is join(). join() behaves just like get() except that no checked exception is thrown.

E.g. to find diff btw the i-th prime number and the j-th prime number

```
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));
// This would launch 2 concurrent threads to compute i-th and j-th primes. The mtd supplyAsync
returns immediately without waiting for findIthPrime to complete.

CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);
// Create another CompletableFuture that compute diff btw the two prime nums asynchronously
At this point, we can move on to run other tasks, or to just wait until result is ready, use diff.join();
```

One of the advantages of using CompletableFuture<T> instead of Thread is its ability to handle exceptions.
CompletableFuture<T> has 3 mtds to deal with exceptions: exceptionally, whenComplete, and handle.
Since computation is asynchronous and could run in a diff thread, the qn of which thread should catch and handle exception arises.
CompletableFuture<T> keeps things simpler by storing the exception and passing it down the chain of calls, until join() is called. join() might throw CompletionException and whoever calls join() will be responsible for handling this exception.
The CompletionException contains info on the original exception.

```
CompletableFuture.<Integer>supplyAsync(() -> null)
 .thenApply(x -> x + 1)
 .join();
```

throw a CompletionException with NullPointerException contain within it

Suppose we want to continue chaining our tasks despite exceptions. We can use the handle mtd, to handle the exception.
The handle mtd takes in a BiFunction (similar to cs2030s.fp.Combiner). The 1st arg is the value, 2nd is exception, 3rd is return value.
Only one of the first two parameters is not null. If value is null, means that an exception has been thrown. Else, the exception is null

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

using handle to replace a default value
//t is value, e is exception

```
Queue<Runnable> queue;
new Thread(() -> {
 while (true) {
  if (!queue.isEmpty()) {
```

A thread pool consists of (i) a collection of threads, each waiting for a task to execute, (ii) a collection of tasks to be executed. Typically the tasks are put in a queue, and an idle thread picks up a task from the queue to execute.

```
    Runnable r = queue.dequeue();
    r.run(); } } }).start();
for (int i = 0; i < 100; i++) {
  queue.add(() -> System.out.println(i));
}
```

To illustrate this concept, here is a trivial thread pool with a single thread:
We assume that Queue<T> can be safely modified concurrently (i.e., it is thread-safe) in this e.g. Otherwise, just like the example you have seen in parallel streams with List, items might be lost.

```
//recursively sums up the content of an array
class Summer extends RecursiveTask<Integer> {
  private static final int FORK_THRESHOLD = 2;
  private int low;
  private int high;
  private int[] array;
  public Summer(int low, int high, int[] array) {
    this.low = low;
    this.high = high;
    this.array = array; }
  @Override
  protected Integer compute() {
    // stop splitting into subtask if array is already small.
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) { sum += array[i]; }
      return sum; }
    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();
    return right.compute() + left.join();
  }
}
```

The Fork-join model is essentially a parallel divide-and-conquer model
Done by breaking the problem into identical problems with smaller size (*fork*), solve this smaller problem, and combine the results (*join*).
This repeats recursively until problem size is small enough -- reach base case and just solve problem sequentially
We can use Java RecursiveTask<T> which has methods fork(), and join() (which waits for the smaller tasks to complete and return).
RecursiveTask<T> has an abstract method compute(), which we have to implement.
If the array is big enough, two new Summer instances, left and right, are created. We then call left.fork(), which adds the tasks to a thread pool so that one of the threads can call its compute() method. We subsequently call right.compute() (which is a normal method call). Finally, we call left.join(), which blocks until the computation of the recursive sum is completed and returned. We add the result from left and right together and return the sum.

```
Summer task = new Summer(0, array.length, array);
int sum = task.compute();
```

Logic of how Java manages the thread pool with fork-join tasks.

- Each thread has a queue of tasks.
- When a thread is idle, it checks its queue of tasks. If the queue is not empty, it picks up a task at the head of the queue to execute (e.g., invoke its compute() method). Otherwise, if the queue is empty, it picks up a task from the *tail* of the queue of another thread to run. The latter is a mechanism called *work stealing*.
- When fork() is called, the caller adds itself to the *head* of the queue of the executing thread. This is done so that the most recently forked task gets executed next, similar to how normal recursive calls.
- When join() is called, If the subtask to be joined hasn't been executed, its compute() method is called and the subtask is executed. If the subtask to be joined has been completed (some other thread *work steal*), then the result is read, and join() returns. If the subtask to be joined has been stolen and is being executed by another thread, then the current thread finds some other tasks to work on either in its local queue or steal a task from another queue.

```
left.fork();
right.fork();
return right.join() + left.join();
```
is more efficient than
```
left.fork();
right.fork();
return left.join() + right.join();
```

One implication of how ForkJoinPool adds and removes tasks from the queue is the order in which we call fork() and join(). Since the most recently forked task is likely to be executed next, we should join() the most recent fork() task first. In other words, the order of joining should be the reverse of the order of forking.

- to ensure that threads don't overlap when say incrementing a value
```
private synchronized void increment() {
        value++;
}
```

//ensure threads can only use increment mtd one at a time