1 Abstraction & Encapsulation

1.1 Access Modifiers

	Class	Package	Subclass	World
public	+	+	+	+
protected	+	+	+	
no modifier	+	+		
private	+			

1.2 Reference Type vs Primitive Type

- All objects are stored as references in Java.
- A variable of primitive type stores the value instead of a reference to the value.
- Java supports 8 primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`. However, `char` stores 16-bit Unicode, and `byte` stores 8-bit character.

2 Inheritance & Polymorphism

2.1 Syntax

- `class Rectangle extends Shape implements Shape, Printable`
- `extends` for inheritance, `implements` for implementing interface.
- `super` in sub-class refers to the parent class. For example, `super()` for constructor or `super.method()` or `super.property`
- `@Override` for methods meant to override a parent class' method

3 More on Inheritance

3.1 Relationship

- **HAS-A relationship** : use composition
- **IS-A relationship** : use inheritance

3.2 Liskov Substitution Principle (LSP)

- 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 is a subtype of T.
- This means that if S is a subclass of T, then an object of type T can be replaced by an object of type S without changing the desirable property of the program.
- Thus, not all IS-A relationship should be modeled with inheritance.

3.3 Overriding

- Useful for Polymorphism, e.g. for `equals(Object o)`.

```
Circle c = new Circle(10); Object o = c;
c.equals(c); // calls the one from Circle
o.equals(o); // calls the method from Circle if overridden, Object if not
```

3.4 Preventing Inheritance and Method Overriding

- Use `final` keyword (but also used for constant, eg. `public static final double PI = 3.141592653589793;`)
- `final class Circle {}`
- `class Circle { final public boolean contains(Point p) }`
- However, there is another use for `final` keyword: for constants, eg. `public static final double PI = 3.141592653589793;`

4 Types, Memory, Exception

4.1 Types

4.1.1 Type Conversion

- **Widening reference conversion** : for $T <: S$ (T is a subtype of S), converting T to S checked at compile time.
- **Narrowing reference conversion** : for $T <: S$, converting S to T, explicit casting needed.

4.1.2 Subtyping

`byte` <: `short` <: `int` <: `long` <: `float` <: `double`
`char` <: `int`

Widening Primitive Conversion

- `byte` to `short`, `int`, `long`, `float`, or `double`
- `short` to `int`, `long`, `float`, or `double`
- `char` to `int`, `long`, `float`, or `double`
- `int` to `long`, `float`, or `double`
- `long` to `float` or `double`
- `float` to `double`

Narrowing Primitive Conversion

- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`
- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

4.1.3 Variance of Types

- A is **Covariant** if $T <: S$ implies $A(T) <: A(S)$
- A is **Contravariant** if $T <: S$ implies $A(S) <: A(T)$
- A is **Bivariant** if both covariant and contravariant
- A is **Invariant** ifneither covariant nor contravariant

4.2 Heap and Stack

- **Heap** : region in memory where **all objects** are allocated in and stored.
- **Stack** : region in memory where all variables (**including primitives types and object references**) are allocated in and stored.

```
Circle c;
c = new Circle(new Point(1, 1), 8);
```

4.2.1 Call Stack

- JVM creates a **stack frame** for every method call, containing (i) the `this` reference (only for instance method, not for class method), (ii) the method arguments, (iii) local variables within the method. (iv) variable capture

```
1 class Point {
2  private double x;
3  private double y;
4  public double distanceTo(Point q) {
5   return Math.sqrt(Math.pow(q.x - this.x, 2) + Math.pow(q.y - this.y, 2);
6  }
7 }
8 Point p1 = new Point(0,0);
9 Point p2 = new Point(1,1);
10 p1.distanceTo(p2);
```

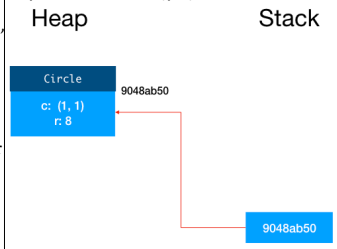

Figure 1: First Example

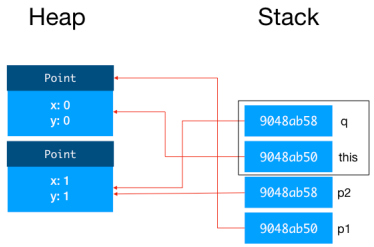

Figure 2: Call Stack

4.3 Exceptions

- Use `try/catch/finally` control statements.

```
1 try {
2   reader = new FileReader(filename);
3   scanner = new Scanner(reader);
4   numOfPoints = scanner.nextInt();
5 } catch (FileNotFoundException e) {
6   System.err.println("Unable to open " + filename + " " + e);
7 } catch (InputMismatchException e) {
8   System.err.println("Unable to scan for an integer");
9 } catch (NoSuchElementException e) {
10  System.err.println("No input found");
11 } finally {
12  if (scanner != null)
13  scanner.close();
14 }
```

- Lines 2-4 keep the basic tasks together, lines 12-13 group together all the cleanup tasks
- The `finally` block is always executed even when return or throw is called in a `catch` block.
- It is also possible to combine multiple catches

```
catch (FileNotFoundException | InputMismatchException |
↪  NoSuchElementException e) {
  System.err.println(e);
}
```

- There are 2 types of exceptions:
- **Checked exception** is something that the programmer should anticipate and handle.
- **Unchecked exception** is something that the programmer does not anticipate, and usually is a result of a bug.

4.3.1 Checked Exceptions

- We need to either catch all checked exceptions or let it propagate to the calling method. Otherwise, the program will not compile.
- All methods that throw checked exception need to specify the checked exception(s) using `throws`

```
1 public static int readIntFromFile(String filename) throws
  ↪  FileNotFoundException {
2   FileReader reader = new FileReader(filename);
3   Scanner scanner = new Scanner(reader);
4   int numOfPoints = scanner.nextInt();
5   scanner.close();
6   return numOfPoints;
7 }
```

4.3.2 Generating Exceptions

- When you override a method that throws a checked exception, the overriding method must throw only the same, or a more specific checked exception, than the overridden method.
- This rule enforces the Liskov Substitution Principle.

```
public Circle(Point p, Point q, double r, boolean centerOnLeft) {
  if (p.distanceTo(q) > 2*r) {
    throw new IllegalArgumentException("Input points are too far apart");
  }
  if (p.equals(q)) {
    throw new IllegalArgumentException("Input points coincide");
}
```

4.3.3 Good Practices for Exception Handling

- **Catch Exceptions to Clean Up** : It is better to catch the exception, handle the resource deallocation in a `finally` block. If need be, can always re-throw the exception.

```
public void m2() throws E2 {
  try {
    // Set up resources
  }
  catch (E2 e) {
    throw e;
  }
  finally {
    // clean up resources
  }
}
```

- **DO NOT catch all exceptions** :

```
try {}
catch (Exception e) {} // NO NO NO NO!!!!
// Or worse:
try {}
catch (Error e) {}      // NO NO NO NO!!!! U WOT M8!!!!
```

- **Overreacting** : Do not exit just because of exception, especially not silently.
- **Do Not Break Abstraction Barrier** : Sometimes, letting the calling method handles the exception causes the implementation details to be leaked, and make it harder to change the implementation later. We should, as much as possible, handle the implementation specific exceptions within the abstraction barrier.

```
class ClassRoster {
  public Students[] getStudents() throws FileNotFoundException {}
}
// Later on
class ClassRoster {
  public Students[] getStudents() throws SQLException {}
}
```

5 Generics and Collections

5.1 Abstract Class and Interface with Default Methods

- **Abstract Class** :

```
abstract class PaintedShape {
  Color fillColor;
  void fillWith(Color c) {
    fillColor = c;
  }
  abstract double getArea();
  abstract double getPerimeter();
}
```

- **Interface with default implementation** (introduced in Java 8): using the keyword `default`

```
interface Shape {
  public double getArea();
  public double getPerimeter();
  public boolean contains(Point p);
  default public boolean cover(Point p) {
    return contains(p);
  }
}
```

5.2 Generics

- Generic typing is a type of polymorphism, also known as **parametric polymorphism**.

```
1   class Queue<T> {
2     private T[] objects;
3     public Queue<T>(int size) {}
4     public boolean isFull() {}
5     public boolean isEmpty() {}
6     public void enqueue(T o) {}
7     public T dequeue() {}
8   }
9   Queue<Circle> cq = new Queue<Circle>(10);
10  cq.enqueue(new Circle(new Point(0, 0), 10));
11  cq.enqueue(new Circle(new Point(1, 1), 5));
12  Circle c = cq.dequeue();
```

- T is a **type parameter**. In line 9, `Circle` is passed as the **type argument** to T, creating a **parameterised type** `Queue<Circle>`.
- Thus, in line 12, we no longer need to cast because `cq.enqueue(new Point(1, 3));` will generate a compile-time error.

5.2.1 Variance of Generic Types

- Generics are **invariant** w.r.t. the type parameter.
- **Wildcards** :
- Subtyping among generic types are achieved through wildcard types ?, e.g. this is OK `Queue<?> qc = new Queue<Shape>();`
- However, ? is not a type, thus we cannot declare a class as such `class Bar<?> {}`
- Wildcards can be bounded using keywords `extends` (for covariant) and `super` (for contravariant)
- `Queue<Circle>` and `Queue<Shape>` are both subtypes of `Queue<? extends Shape>`, but not `Queue<Object>`
- `Queue<Object>` and `Queue<Shape>` are both subtypes of `Queue<? super Shape>`, but not `Queue<Circle>`

5.2.2 Type Erasure

- For backward compatibility. The type argument is erased during compile time, and the type parameter T is replaced with either `Object` (if unbounded) or the bound (if bounded)

Impacts:

1. No support for **generics of primitive types.**
2. No **distinction in method signature** for different type parameters:

```
class A {
  void foo(Queue<Circle> c) {}
  void foo(Queue<Point> c) {}
}
// After type erasure
class A {
  void foo(Queue c) {}
  void foo(Queue c) {}
}
```

3. Using **static methods/fields** become trickier.

```
1 class Queue<T> {
2   static int x = 1;
3   static T y;
4   static T foo(T t) {};
5 }
```

- `Queue<Circle>` and `Queue<Point>` are compiled to `Queue`, so they share the same x
- Lines 3-4 raise compilation error, since T could be both `Circle` and `Point`

4. We **cannot create an array** of parameterized types. After type erasure, if this were allowed, it would have been possible to add `Queue<Circle>` to `Queue<Point>[]` since after type erasure both are `Queue`

5.2.3 Raw Types

- For backward compatibility, Java allows us to use a generic class to be used without the type argument.

```
Queue<Circle> cq = new Queue();
```

- In Java 5 or later, if we just use `Queue`, we get a `Queue` of `Object`. This is called a raw type. Recent Java compilers will warn you if you use a raw type in your code.

5.2.4 Wrapper Class

- Java provides a set of wrapper classes, one for each primitive type: `Boolean`, `Byte`, `Character`, `Integer`, `Double`, `Long`, `Float`, and `Short`.
- Java 5 introduces **autoboxing** and **unboxing**, which creates the wrapper objects automatically (autoboxing) and retrieves its value (unboxing) automatically.

```
Queue<Integer> iq = new Queue<Integer>(10);
cq.enqueue(4);
cq.enqueue(8);
```

- Note that enqueue expects an `Integer` object, but we pass in an `int`. This would cause the `int` variable to automatically be boxed (i.e., be wrapped in `Integer` object) and put onto the call stack of enqueue.
- However, all primitive wrapper classes are immutable. Thus, performance penalty when updating (due to allocating memory and garbage collectipm)

5.2.5 Generic Methods

- The scope of the type parameter is limited to only the method itself. In this case, X (to prevent confusion. Conventionally, T is used instead)

```
1    class Queue<T> {
2      static <X> X foo(X t) { return t; };
3    }
4    Queue.<Point>foo(new Point(0, 0));
5    Queue.foo(new Point(0, 0));
```

- On line 5, Java compiler uses **type inference** to determine what X should be.
- Also for constructors using the **diamond operator** `Queue<Point> q = new Queue<>();`

5.3 Java Collections

5.3.1 Set and List

- `Set<E>` does not allow duplicates (but still does not care about order of elements) – a **set**. Useful classes implementation: `ArrayList`, `LinkedList`
- `List<E>` allows duplicates and the order of elements matters – a **sequence**. Useful classes implementation: `HashSet`

```
List<String> names = new ArrayList();
names.add("Cersei");
names.add("Joffrey");
names.add(0, "Gregor");
System.out.println(names.get(1)); // Prints Cersei
```

5.3.2 Comparator

- `List<E>` interface specifies `default void sort(Comparator<? super E> c)`
- Example Comparator class:

```
class NameComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return s1.compareTo(s2); // Sorting by lexicographical order
  }
}
```

5.3.3 Map

- store a unique key, value pair. `Map<K,V>` is generic with 2 type parameters.
- Useful class implementation: `HashMap`

```
Map<String,Integer> population = new HashMap<String,Integer>();
population.put("Oldtown",600000);
population.put("Kings Landing",500000);
population.put("Lannisport",300000);
population.get("Kings Landing"); // Returns 500000
```

5.3.4 Which one to use?

- **HashMap** if you want to keep a (key, value) pair for lookup later.
- **HashSet** no duplicates and order is not important. Quick `contain` check.
- **ArrayList** possible duplicates and order is important, and random access is more important than adding/removing.
- **LinkedList** possible duplicates and order is important, adding/removing is more important than random access.
- **PriorityQueue** Like a queue but order of poll is according to natural ordering

6 Hash Code, Nested Classes, Enum

6.1 Hash Code

- If `a.equals(b)` then `a.hashCode() == b.hashCode()` (not necessarily the converse)
- How to create `hashCode()`:

```
class A {
  public double x, y;
  @Override
  public int hashCode() {
    return Arrays.hashCode(new double[] {this.x, this.y});
  }
}
```

- If you have fields in a class of different types, pack them into different arrays, call `Arrays.hashCode()` for each of the arrays, and combine the hash code together with the XOR operator `^`

6.2 Nested Class

- A nested class can be either **static** or **non-static**.
- A **static nested class** is associated with the containing class, NOT an instance. So, it can only access static fields and static methods of the containing class.
- A **non-static nested class** can access all fields and methods of the containing class. A non-static nested class is also known as an **inner class**.

6.2.1 Local Class

- Classes that are declared inside a method (more precisely, inside a block of code between { and }) is called a **local class**.
- A local class is scoped within the method. It has access to the variables of the enclosing class and local variables of the enclosing method.

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

6.2.2 Variable Capture

- A local class **captures** the local variables in the enclosing method (the local class makes a copy of local variables in the enclosing method inside itself.).

6.2.3 Effectively final

- Java only allows a local class to access variables that are explicitly declared final or implicitly final (or effectively final).
- An implicitly final variable is one that does not change after initialization.

6.2.4 Anonymous Class

Format: `new X (arguments){ body }`

- **X** is a class that the anonymous class extends or an interface that the anonymous class implements. (cannot be empty, no multiple inheritance – cannot extend a class and implement an interface, cannot implement more than one interface)
- **arguments** are the arguments that you want to pass into the constructor of the anonymous class. (If extending an interface, there is no constructor, but we still need `()`)
- **body** the body of the class as per normal, except that we cannot have a constructor.

```
names.sort(new Comparator<String>() {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
});
// OR
Comparator<String> cmp = new Comparator<String>() {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
};
names.sort(cmp);
```

- The same rules to variable access as local class applies.

6.3 Enum

- Variable of an enum type can only be one of the predefined constants.
- Trying to assign anything other than the two predefined event type to eventType would result in compilation error.

```
enum EventType {
  CUSTOMER_ARRIVE,
  CUSTOMER_DONE
}
class Event {
  private double time;
  private EventType eventType;
}
```

6.3.1 Enum's Fields and Methods

- Each constant of an enum type is actually an instance of the enum class and is a field in the enum class declared with `public static final`.
- Since enum in Java is a class, we can define constructors, methods, and fields in enums.

```
enum Color {
  BLACK(0, 0, 0),
  WHITE(1, 1, 1),
  RED(1, 0, 0),
  BLUE(0, 0, 1),
  GREEN(0, 1, 0),
  YELLOW(1, 1, 0),
  PURPLE(1, 0, 1);
  private final double r;
  private final double g;
  private final double b;
  Color(double r, double g, double b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }
  public double luminance() {
    return (0.2126 * r) + (0.7152 * g) + (0.0722 * b);
  }
  public String toString() {
    return "(" + r + ", " + g + ", " + b + ")";
  }
}
Color a = Color.YELLOW;
System.out.println(a); // Prints (1.0, 1.0, 0.0)
a.luminance(); // Returns 0.9278
```

6.3.2 Custom Methods for Each Enum

- We can define custom methods for each of the enum constant, by writing constant-specific class body.

```
enum EventType {
  CUSTOMER_ARRIVE {
    void log(double time, Customer c) {
      System.out.println(time + " " + c + " arrives");
    }
  },
  CUSTOMER_DONE {
    void log(double time, Customer c) {
      System.out.println(time + " " + c + " done");
    }
  };
  abstract void log(double time, Customer c);
}
EventType.CUSTOMER_DONE.log(time, customer);
```

6.3.3 The Class Enum

- All enum inherits from the class `Enum` implicitly.

```
// The example in 6.3 is actually
public final class EventType extends Enum<EventType> {
  public static final EventType[] values { .. }
  public static EventType valueOf(String name) { .. }

  public static final EventType CUSTOMER_ARRIVE;
  public static final EventType CUSTOMER_DONE;
  :

  static {
    CUSTOMER_ARRIVE = new EventType();
    CUSTOMER_DONE = new EventType();
    :
  }
}
```

- To ensure that the generic type `E` actually inherits from `Enum<E>`, class `Enum` is defined to be generic type `Enum<E extends Enum<E>>`
- An enum is final. We cannot inherit from enum (those with constant-specific body are exceptions).
- A class in Java can contain fields of the same class.
- The block marked by static .. are static initializers, they are called when the class is first used. They are the counterpart to constructors for objects, and are useful for non-trivial initialization of static fields in a class.
- Two useful classes `EnumSet` and `EnumMap` – special cases of `HashSet` and `HashMap` respectively – the only difference is that we can only put enum values into EnumSet and enum-type keys into EnumMap.

7 Functions

7.1 Pure Functions

- **No side effect** (including but not limited to printing to the screen, writing to files, throwing exceptions, changing other variables, modifying the values of the arguments.)
- **Deterministic** (given same input, produces same output everytime)

7.2 `Function` interface and other related interfaces in Java 8

Interface	SAM
`Function<T, R>`	`R apply(T t)`
`Supplier<T>`	`T get()`
`Consumer<T>`	`void accept(T t)`
`BiFunction<T, U, R>`	`R apply(T t, U u)`

7.3 Lambda Expression

```
// All are equivalent
applyList(list, new Function<Integer,Integer>() {
  public Integer apply(Integer x) {
    return x * x;
  }
});
applyList(list, (Integer x) -> { return x * x; });
applyList(list, x -> { return x * x; });
applyList(list, x -> x * x);

// Method Reference
applyList(list, x -> Math.abs(x));
applyList(list, Math::abs);
```

7.4 Composing Functions

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after);
default <V> Function<V,R> compose(Function<? super V,? extends T> before);

Function<Integer,Integer> abs = Math::abs;
Function<Integer,Double> sqrt = Math::sqrt;
// All are equivalent
x -> Math.sqrt(Math.abs(x));
abs.andThen(sqrt));
sqrt.compose(abs));
```

7.5 PECS

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after);
```

We can make the method more general by allowing it to take a function with R or any superclass of R as input – **surely if the function can take in a superclass of R, it can take in R**. Thus, we can relax input type, or what the function consumes, from R to ? super R.

Similarly, if we are expecting the function after to return a more general type V, **it is fine if it returns V or a subclass of V**. Thus, we can relax the return type, or what the function produces, from V to ? extends V.

mnemonic "producer extends; consumer super", or PECS, for short.

8 Streams

8.1 Lambda as Closure

- A lambda expression can capture the variables of the enclosing scope since lambda expression is just a shorthand to an anonymous class.
- A captured variable must be either explicitly declared as final or is effectively final.

8.2 `Optional<T>` class

8.3 `Stream` class

9 FP Patterns

```
interface Thing<T> {
  public <R> Functor<R> f(Function<T,R> func);
  public <R> Monad<R> f(Function<T, Thing<R>> func);
}
```

9.1 Functor

- It will "unwrap" the value, applies the function, then re-"wrap" the value.
- e.g. `Optional` class.

9.2 Functor Laws

- if `func` is an identity function `x -> x`, then it should not change the functor.
- if `func` is a composition of two functions $g \cdot h$, then the resulting functor should be the same as calling f with h and then with g.

9.3 Monad

- It will "unwrap" the value, applies the function which returns an already wrapped value.
- e.g. `Stream` class

9.4 Monad Laws

- ∃ an of operation that takes (an) object(s) and wrap it / them into a monad:
  - **Left identity law**: `Monad.of(x).flatMap(f) == f(x)`
  - **Right identity law**: `monad.flatMap(x -> Monad.of(x)) == monad`
- **Associative law**: the `flatMap` operation should be associative – `monad.flatMap(f).flatMap(g) == monad.flatMap(x -> f(x).flatMap(g))`

9.5 Implementing Strategy / Policy

- OO using polymorphism through inheritance.
- FP using `Runnable`

9.6 Observer Pattern

Basically event-driven: add actions to do when triggerred.

10 Parallel Streams

By adding `parallel()` anywhere in the chain, e.g.:

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

10.1 Ensuring correctness of parallelisation

Criteria:

- **No interference**: stream operations must not modify the source of the stream during execution.
- **Stateless**: stream operations must not depend on any external state that might change during execution.
- **No side effects**
- **Associativity**: For `reduce`:
  - `combiner.apply(identity, i) == 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)) == accumulator.apply(u, t)`

```
// Associativity Example:
Stream.of(1,2,3,4).reduce(1, (x,y)->x*y, (x,y)->x*y);
// i * 1 == i
// (x * y) * z == x * (y * z)
// u * (1 * t) == u * t

// Using non-thread-safe data structure:
List<Integer> result =
    list.parallelStream()
        .filter(x -> isPrime(x))
        .collect(Collectors.toList())
// Or using a thread-safe data structure
List<Integer> result = new CopyOnWriteArrayList<>();
list.parallelStream()
  .filter(x -> isPrime(x))
  .forEach(x -> result.add(x));
```

10.2 Fork and Join

- `ForkJoinTask<V>` abstract class has 2 important methods `fork()` (submits task for execution) and `join()` (waits for computation to complete and returns the value)
- `RecursiveTask<V>` abstract class is a subclass of `ForkJoinTask<V>` with a method `V compute()`

11 Asynchronous

11.1 `Future<T>` Interface

Implemented by both `RecursiveTask` and `RecursiveAction`. Five simple operations:

- `get()` result of computation (waiting if needed)
- `get(timeout, unit)` same but waiting up to timeout period
- `cancel(interrupt)`
- `isCancelled()`
- `isDone()`

11.2 `CompletableFuture<T>` interface