Object Oriented Programming

To be maintainable is to be
- Simple to locate code responsible for a particular feature.
- Simple to understand what the code does.
- Simple to modify behaviour.
- Difficult to introduce bugs.

Types of Languages

Definitions
- **Declarative languages** specifies what to do (not how to do it).
  - Includes **functional, logic and reactive languages** (which reasons about streams of data).
  - E.g. HTML and SQL
  - OCaml: you give an example on how to achieve something, the compiler has the choice to do something completely different (e.g. tail recursion).
- **Imparative languages** specifies what and how to do it.
  - **Procedural languages** group things into functions.
  - **OOP** groups procedures together - dividing a large procedural program into many small chunks, which are easier to reason about.

Characteristics of OOP are:
- Encapsulation
- Abstraction
- Inheritance
- *Subtype* polymorphism

Languages often pick and mix concepts for the language creator to complete their tasks.
- E.g. procedural programming in OCaml.
- Functional programming in C++.

OOP is not always appropriate, smaller programs **don't need abstractions**, e.g. using Python for data science. But OOP is better for larger scale programs.

The JVM

Java was made for the web, where each machine visiting a page has different processors.

Model	Note
Traditional model	The compiler compiles source code for a particular machine, that can only run on that type of system. But how do you compile for every system and serve the correct binaries to visitors?
Using an interpreter	High level languages are not space efficient (not suitable for the 2000s internet speed), and source code is visible to everyone.
The Java model	The Java compiler compiles Java source code to **bytecode**, which is translated to machine code on the fly.

Definition
Bytecode is machine code for a fictional machine.

Translating from bytecode to machine code is quick, as the toughest part of compilation is going from source code to compiled code.

Bytecode is slower than native code, but compared to interpreted code:
- Bytecode is compiled and not easy to reverse engineer.
- JVM ships with libraries making the bytecode small.

The JIT

JVM uses a **JIT** which profiles your code. If it sees a code running over and over again, it compiles it to machine code. So Java programs become faster the longer you run them.

It is used for backend: servers are long running programs that the JVM can learn and optimise.

Java

Basic Language Features

Project Layout

There is **one class per file**, and the class name and file name must match (this is to make the code more maintainable).
- The **JRE** is what you need to run Java bytecode.
- The **JDK** is what you need to compile Java. The JRE is a subset of JDK.

Data Types

Java is strongly typed. Types are either built-in **primitive types** or **reference types**, which starts with a capital letter.

- `boolean`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`, which is the only **unsigned integer**.

Variables can be **promoted** or **narrowed** to another type.

(DANGER!) Inferring types on local variables can make the code more clear... or more confusing.

```
var courseName = "Java"; // clearly a String
var n = 1;               // what type is this?
```

Procedures

A Java *"function"* is made up of a **prototype** and a **body**, the prototype specifies the function name, arguments and return type. There are **no pure functions** in Java. There are only procedures which can manipulate state outside the functions.

Encapsulation

Definitions

Keyword	Definition
Object	A bundle of state and behaviour.
Class	A template for a specific type of object.

- The state of an object is called a **field**, the behaviour are called **methods**.
- A class defines a **type** and the **implementation of methods**.

Class

All code in Java are contained in classes.

For best readability, in each file declare in order.

1. Imports
2. Constants
3. Fieilds
4. Constructors
5. Methods

Constructors

Constructors have the same name as the class, have no return types (why?), and can be overloaded.

```
public class MyClass {
  public MyClass(int a) {}
  public MyClass(int a, int b) {}
}
```

> **Note**
> If you don't declare any constructors, Java will declare one for you.
>
> You can look at class file signatures with
>
> `javap -c MyClass.class`

Parameterised Classes

Polymorphism can be done through generics. Classes are defined with placeholder types.

```
public class Vec2D<T> {
  public T x;
  public T y;
}
```

We fill them in when we create an object with them.

```
Vec2D<Integer> l = new Vec2D<Integer>();
```

> **Note**
> The type parameter must be a **reference type**.

Statics

Objects don't allow global state as state is self contained.

> **Definition**
> A **static field** is created only once in the program's execution.

Pros	Cons
Auto synchronised cross instances.	Make code harder to understand.
Space efficient.	

> **Note**
> Make static fields `final` whenevr possible.

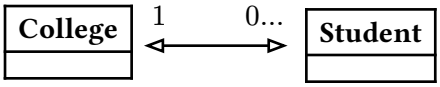
Static methods don't belong to an object, they are used to group related methods.

Universal Modelling Language

MyClass
- privateField + publicField
- privateMethod() + publicMethod()

Model **has-a** association with arrows.

- A college has zero or more students.
- A student has one college.



The college can store its students in a list, the student can store his college in a single variable.

Visibility

We want to provide an API that exposes what it needs to and hide everything else. Encapsulation allows us to decouple the API from the underlying state: changing how the state is represented internally will not affect code depending on it.

	Class	**Subclass**	**Package**	**Everywhere**
`public`	✓	✓	✓	✓
no modifier	✓	✓	✓	
`protected`	✓	✓		
`private`	✓			

Note
Encapsulation is also called information hiding.

Public state is almost never needed:

- Create private variables by default.
- Never make a public variable private, it can break code.

Immutability

Immutable	**Mutable**
Easier to reason about.	Updating values in place is more efficient.
Reduces scope for bugs.	
Thread safe.	

Definition
Immutable classes: state can be set at initialisation but cannot be changed.

Immutable classes can be achieved with the `final` modifier, or using Java **records** (since Java 16).

```
public record Vec2D(int x, int y) {}
```

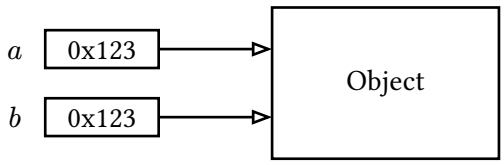
Immutable classes are everywhere in the JDK: `Integer`, `String`, `LocalDate`, `Optional`, etc.

Data Representation

Primitive and Reference Types

Definitions
- A **primitive value** is a chunk of memory that holds the value of the variable.
- **Objects** are referred to by reference - the variable links to a separate chunk of memory.

Changes to the block of memory being pointed to willl reflect in both reference variables.



Note
An array is a reference type.

String Optimisation

A string is immutable and stored in a read-only part of memory. Variables assigned the same string content will have the same reference address from optimisation.

The Call Stack

A function puts 3 things on the **call stack** when called.
- Local variables created by the function.
- Memory address to jump to when complete (return address).
- A copy of the arguments passed to it.

They are held in the function's **stack frame** in the call stack.

Definition
A **stack** is a FIFO data structure.

Local variables are deleted when a stack frame is popped. Static variables have global scope.

Note
There is no tail call optimisation in Java.

The Heap

We cannot resize variables in the stack: that would require moving everything above it on the stack. **The heap** is more flexible, there are gaps between objects in the heap.

- Primitives and references cannot change size, so they go on the stack.
- Everything else goes on the heap.

Pointers and References

Definition
A **pointer** is a chunk of memory that holds an address to another memory.

Pointers	References
Can be treated like numbers, can do arithmetic with it.	References are restricted pointers.
Gives us raw memory access (which could be dangerous).	
No way to test if a pointer is valid.	References can be tested.

C, C++ Supports both references and pointers, ML and Java only allow references.

Pass by Value/Reference

- **All Java parameters are pass by value.** These *values* passed to a function can include references that points to an object in heap.
- In C, the variables passed to a function can point to a variable from the calling function.

Inheritance

Inheritance is where a **subclass** inherits state and functionality from a **superclass**.

- Subclass use the `extend` keyword to inherit from a superclass.
- The subclass can directly access public and protected members of the superclass.

All Java classes must inherit from some other class, if not specfied it will inherit from `java.lang.Object`.

```
public class MyClass; // will be replaced by the compiler as:
public class MyClass extends Object;
```

In a UML diagram, inheritance is an unfilled arrow pointing towards the superclass.

```
class A;
class B extends A;
class C extends B;
```

When constructor is not explicitly written for `C`, when creating `C`, we will call:

1. Constructor of `C`
2. Which calls the constructor of `B`
3. Which calls the constructor of `A`

We can explicitly call the constructor of the superclass with `super();`

Type Casting

- **Widening conversion** cast up the tree, e.g. $C \to B \to A$
- **Narrowing conversion** is checked at runtime, if the type does not match will cause type errors.

```
B b1 = new C()
B b2 = new B();

C c1 = (C) b1; // ok
C c2 = (C) b2; // type error
```

Shadowing

If a variable in the subclass has the same name as the superclass, we can specify where the variable comes from.

```
// running in C
// these are all different variables
this.x = 10;
((B) this).x = 20;
((A) this).x = 30;
```

> **Note**
> This is very ugly, please avoid at all cost.

Overriding

Definition
Annotations are not part of the language, but hints to the compiler so it can perform stricter checks.

Use the `@Override` annotation to make sure it overrides a function from superclass.

```
@Override
public void myFunc();
```

Definitions
- An **abstract class** forces a subclass to override abstract methods.
- An **abstract method** has no implementation.

- In UML, an abstract class has *italic* name if drawn in computer, { curly braces } if drawn by hand.
- Interfaces just have the word "interface" in an UML diagarm.

Interface groups classes that implements a set of methods.

Inheritance vs Composition

- Inheritance represents an *is-a* relationship.
- Composition represents a *has-a* relationship.

Implementation of a stack with inheritance (inapproprite)

```
class Stack extends Vector;
```

A better implementation uses composition.

```
class stack {
  private Vector internal;
}
```

Polymorphism

- **Subtyping polymorphism** is where many kinds of objects provide the same method.
- **Parametric polymorphism** uses generics.
- **Ad-hoc polymorphism** uses overloading.

Static polymorphism decides the function to run at compile-time.

```
Person p = new Student();
p.dance(); // this runs Person::dance()
```

Java applies static polymorphism on state and static methods.

Dynamic polymorphism decides at runtime when we know the child's type.

```
Person p = new Student();
p.dance(); // this runs Student::dance()
```

Java applies dynamic polymorphism on methods.

Note
Dynamic polymorphism has a runtime overhead.

Polymorphism allows:
- Less changes needed to add features.
- So less likely to make mistakes.

Multiple Inheritance

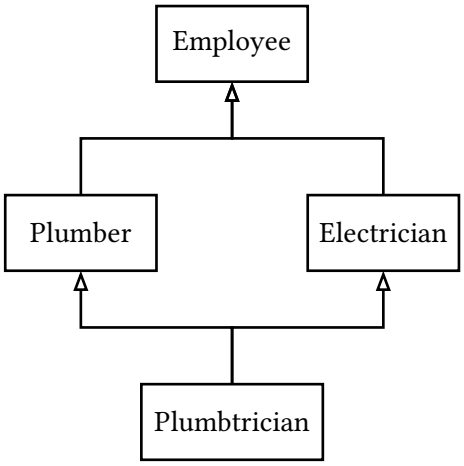
Inheriting from multiple classes causes name clash called the **diamond problem**.

There will be loads of state and method name clashes.

What should happen if we call `plumbtrician.doJob()`?

In Java
- Classes can have at most 1 direct parent.
- But multiple interfaces are allowed: interfaces don't have implementations, it just says the class must provide the methods.

  If the interfaces require the same method that's fine.



> **Note**
> Interfaces can have **default implementations**, so new functions can be added without breaking existing classes (that implements the interface).

Method Resolution

When there is a name clash, the resolution priority from highest to lowest is
1. Class methods
2. Interface methods
3. Methods from superclass

To call a default method from an interface, write

```
class MyClass implements InterA, InterB {
  public void myMethod() {
    InterA.super.myMethod(); // calls InterA implementation
    InterB.super.myMethod(); // calls InterB implementation
  }
}
```

OOP Principles

OCP: Open to extension, Close to modification
- Make classes easy to add new behaviour.
- But hard to change existing behaviour.

LSP: Liskov Substitution Principle

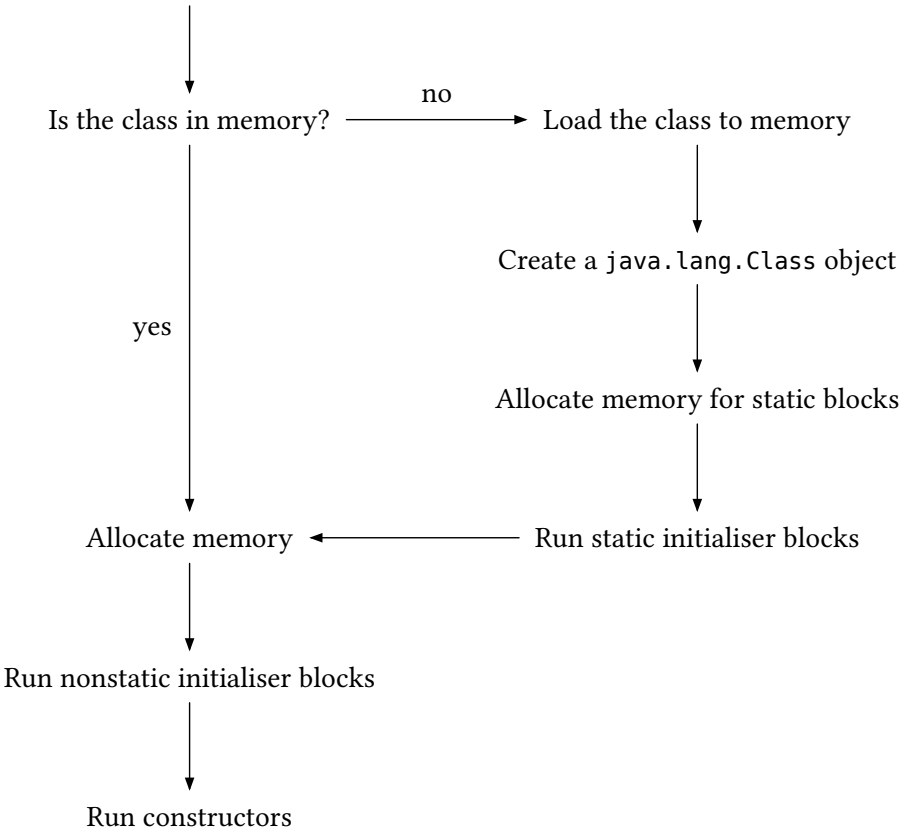
Subtypes must be behaviourally substitute to the parent type without negative side effects.

E.g. code below is not behaviourally substitute for the interface it is implementing.

```
class MyClass implements InterA {
  public void myMethod() {
    throw new UnsupportedOperationException("myMethod");
  }
}
```

Object Lifecycle

Object Creation



Note

Static blocks are code that runs only once when the class is loaded.

```
static {
  // statements
}
```

Object Deletion

Approach 1 The programmer specifies what to delete, it can cause **memory leaks** if objects are not deleted.

Approach 2 Garbage collection: if an object has a **reference count** of zero - it has no references pointing to it, so there is not way to access the object.

Once in a while Java stops the running process, follows all references and mark items that has a reference count of zero.

Deletion Strategies

Delete immediately If there is lots to delete, it can take while.

Queue for deletion If it is taking too long, delete them at the next GC pause.

After deletion, the GC may **compact the heap** to remove gaps in memory. This cannot be done while the program is running.

Note
Since most objects don't live that long, the heap is divided into
1. All objects are created in **eden**.
2. If they survived a few GCs, they are promoted to **survivors**.
3. A few more GCs and they go to **tenured**.

This is called **heap division**.

Garbage Collectors
You can select the GC to use.

Serial GC	Pauses the program for GC
Parallel GC	Same as serial, but uses multiple threads for deletion.
G1 (default)	Monitors while the program is running, and use the GC pause to do deletions.
Epsilon GC	A no-op GC, use only if completly certain that program uses constant memory.

Object Copying

Note
Hearing about a bunch of programmers failing to conform to a standard for 30 minutes in lecture is making me slightly angry.

Definitions
- A **shallow copy** copies the object, but the copied object references the same objects internally if it contains other objects.
- A **deep copy** copies everything, and is almost always what we want.

There are no way to copy a Java object! Yay!

Copy Constructors
The object has a constructor that copies itself.

```
class Vec2 {
  int x, y;

  public Vec2(Vec2 other) {
    this.x = other.x;
    this.y = other.y;
  }
}
```

But if we have the following code.

```
class Vec3 extends Vec2 {
  int z;

  public Vec3(Vec3 other) {
    this.x = other.x;
    this.y = other.y;
    this.z = other.z;
```

```
  }
}
```

```
Vec2 v1 = new Vec3(1, 2, 3);
Vec2 v2 = new Vec2(v1); // this will not use the constructor from Vec3
```

The other "solution" is to use the `Cloneable` interface which gives the `.clone()` function. There are two problems.

```
ArrayList<MyClass> arr1 = new ArrayList(...);
ArrayList<MyClass> arr2 = arr1.clone(); // since MyClass does not implement
                                        // Cloneable, this will crash
```

`Cloneable` is a **marker class**, it exists solely to tell us that the class can be clone, but contains no methods to override. The `.clone()` method is implemented by default in `java.lang.Object` *for some reason* so there are no way to tell if an object can actually be cloned except checking for the `Cloneable` interface (and crashing) at runtime.

Who tf designed this.

Covariant Return Type

If `class B extends A` and the parent class requires you to override `A myMethod()`, then you can override it with `B myMethod()` instead.

Common Interfaces

Collections

A collection is a grouping of objects that can be iterated over (i.e. `Collection<T> extends Iterable<T>`).

Common collections includes, `HashSet`, `ArrayList` and `LinkedList`

	ArrayList	LinkedList
get	$O(1)$	$O(n)$
add	$O(1)$ amortised	$O(1)$
contains	$O(n)$	$O(n)$
remove	$O(n)$	$O(n)$

LinkedList is suitable when there are a lot of add/removing from the head or the tail, but not accessing the middle elements.

- ArrayList have better time complexity in most cases.
- And allows the CPU to be more **cache sympathetic**.

You can create an `UnmodifiableList` with

```
List intList = Collections.unmodifiableList(list);
```

Iterators

The two pieces of code below don't do what you expect them to do.

```
for(int i = 0; i < list.length(); i++) {
  if(list.at(i) == 2)
    list.remove(i);
}
```

```
for(int n : list) { // compile error
  list.remove(n);
}
```

To allow modifying a collection while iterating over it, use the `Iterator<T>` class.

```
Iterator<Integer> iter = list.iterator();
while(iter.hasNext()) list.remove(iterator.next());
```

Queues

The queue interface is also implemented by LinkedList (but LinkedList is not necessarily FIFO).

```
q.offer(1);
q.offer(2);
q.poll(); // 1
q.poll(); // 2
```

Map

```
map.put("A", 1);
map.put("B", 2);
map.get("B"); // 2
```

HashMap

An object is given a single number (**hash**) in a range. Ideally two different objects have different hash. This is clearly not possibly because it is impossible to uniquely map all integers into a finite range.

A LinkedList is used to store multiple objects in the same array index.

	`get`	`put`
TreeMap	$O(\log n)$	$O(\log n)$
HashMap	$O(1)$	$O(1)$

Set

TreeSet and HashSet implements `get`, `add` and `contains`.

> **Note**
> An object added to a set or used as a key on a hash map should not be updated.
>
> To update a key or a set value, remove the entry, update it, then add it again.

Object Equality

`obj1 == obj2` only tests for reference equality. `obj1.equals(obj2)` tests for **value equality**.

- To support that, override the `equals` method with your own implementation.
- It *must satisfy the constraint* `a.equals(b)` $\implies$ `a.hashCode() == b.hashCode()`. So `equals` and `hashCode` must be implemented at the same time.

  `Objects.hash(o1, o2, o3, ...)` is a conventient way to implement the `hashCode` method.

Comparable

Comparable provides method `int compareTo(T obj)`

- `a.compareTo(b) < 0` if $a < b$
- `a.compareTo(b) > 0` if $a > b$
- `a.compareTo(b) = 0` if $a = b$ in terms of order.

We can declare a class implementing `Comparator` to sort a list in a particular order.

```
Collections.sort(list); // ascending order
Collections.sort(list, new MyComparator()); // custom order
```

Generics

Generics allows for better type safety: if we can assign more types to our code, then the compiler can stop us from doing silly things.

- **Static type checking** is done by the compile.
- **Dynamic type checking** cause crashes if the type is wrong at runtime.

Generics allows more type checking to be done at compile time.

We can have generic classes or function

```
public static <T> void myMethod(T value);
```

Generics Implementations

The two ways of implementing generics are:

Templates — Generate **pseudoclasses** for the class when you compile.

`class MyClass<T>` compiles to

- `class MyClass_int`
- `class MyClass_double`
- Etc

Type erasure — Do type checking at compile time, then delete all type information.

`ArrayList<Integer> value;` compiles to `Arraylist value;`

```
for(Integer i : list) {
  do_thing(i);
}
```

Compiles to ⟹

```
for(Object i : list) {
  do_thing((Integer) i);
}
```

Pros	Cons
Bytecode is backwards compatible (Java version 1 does not have generics)	No dynamic type checking.
Compile time type checking exists.	It can create confusing errors.
Less bloat than using templates.	

Weirdness with Type Erasure

You cannot do this

```
T value = new T();

T[] arr = new T[10];

// overloading
void addAll(List<String> items) {}
void addAll(List<Integer> items) {}
```

Because the compile bytecode looks like this

```
Object value = new Object(); // not what you wanted!

Object[] arr = new T[10]; // wrong type

// function signatures clash
void addAll(List items) {}
void addAll(List items) {}
```

Arrays violates **covariance** - if B is a subtype of A, you can use B everywhere you expected A. This is because Java arrays are **reified** (it knows its own type), and a cast will not change that.

```
Student[] students = new Student[10];
Person[] people = (Person []) students; // ok
people[0] = new Lecturer(); // runtime error, the underlying array is still Student[]
```

Bounded Wildcards

`<?>` matches anything, for example `List<?>` matches anything that is a list.

Lower bound	`List<? extends Number>`	A list of values that are a subclass of `Number` • You can read `Number` or its superclass from the list. • You cannot write to the list because you don't know the actual type of the `<? extends Number>`
Upper bound	`List<? super Number>`	A list of values that are a superclass of `Number` • You can write `Number` or its subclass to the list. • You cannot read from the list because it can be any of `Number`'s ancestor.

Coupling

Definition
Coupling is the degree to which different parts of the program depends on each other.

- High coupling rely on internal impl details.
- (Better) Low coubling rely on interfaces defined.

High coupling is caused by
- Reckless inheritance.
- Poor encapsulation (so internal variables can be accessed)

Boxing and Unboxing

Primitives can be boxed `int -> Integer` or unboxed `Integer -> int`: they are different types, `int` is stored in 4 bytes, `Integer` is stored in 16 bytes.

Java automatically boxes and unboxes to make life easier.

```
Integer n = 1; // 1 is an int
n = null;      // this is fine
int i = n;     // compiles, but runtime error
```

Errors

Traditionally, functions return special values called **return codes** to indicate error. This mixes the normal code with the error handling code.

Definition
An **exception** is an object that can be thrown and caught by the calling code.

This separates the normal and error handling code. There may be multiple catch blocks.

```
try {
  // statements
} catch (FileNotFound e) {}
  catch (IOException e) {}
```

To ensure a resource is closed even if there is an error, use a finally block.

```
try {
  // do stuff
} finally {
  // this gets ran no matter what
}
```

There is also a *try with resources* syntax that allows us to pass in anything implementing `java.lang.AutoCloseable` so it is closed after execution.

```
try (FileOutputStream f = new FileOutputStream("file.txt")) {
  // statements
}
```

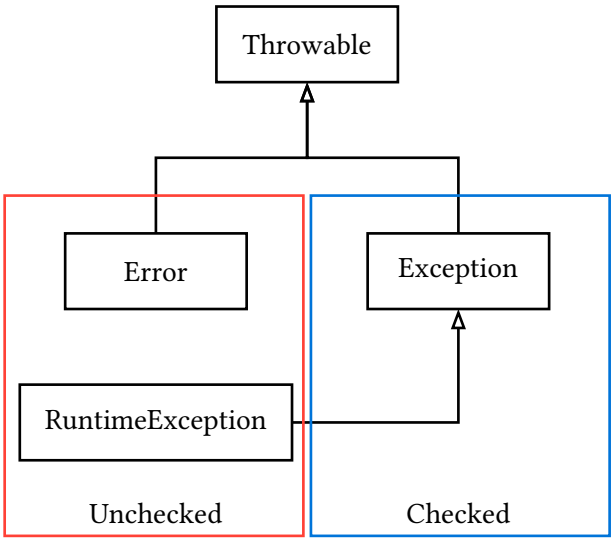
Types of Exceptions

Exceptions are good for

- Documentation: they are on the signature of a function.
- Type safety: same reason as above.
- Separation of concert: recovery logic is separated from normal logic.

Error comes from the JVM, they are not fixable.

- Unchecked errors are not fixable.
- Checked errors must be passed up or handled.



Rules for Exception Handling

1. Never ignore exceptions.
2. Do not catch `Exception`, that also catches `RuntimeException`.
3. Write JavaDoc to include exceptions that may occur.
4. Avoid exceptions specifying implementation details, as it breaks encapsulation.
5. **Don't use exceptions for control flow**.

Assertions

Assertions are enabled with the `-ea` flag.

- Used for **preconditions**: things that are assumed to be true at the start.

  They should only be used for private functions, use exceptions in public functions.
- Used for **postconditions** to ensure the algorithm is functioning correctly.

> **Note**
> Assertions can be turned off for production.

Design Patterns

> **Definition**
> A **design pattern** is a reusable solution to commonly occuring problems.

Principle	Description
The open-close principle	Classes are open for extension close to modification.
Composite	Group objects together, add methods to operate on the objects.
Decorator	Create a wrapper class that contains another class to add additional methods.
State	Include a state variable in the object, so the state can be changed without creating a new parent object.
Strategy	Selecting an algorithm at runtime.
Singleton	Allow only one instance of an object to be created by making the constructor private (e.g. a database connection handle).
Observer	When an object changes state, call all functions listening to the event.
Optional	Returns an optional value instead of null, so the return type not being present is shown in the type signature.

Lambda and Streams

An interface with only 1 function is called a **functional interface**.

```
interface MyPredicate {
  boolean test(Apple apple);
}
```

Then `filter(MyPredicate)` can be called with `filter((Apple apple) -> true)`.

The syntax for a **lambda** is

- `(parameters) -> expression`
- `(parameters) -> { statements; }`

Java provides the following (functional) interfaces.

Interface	Lambda signature
`Predicate<T>`	`T -> boolean`
`Consumer<T>`	`T -> void`
`Function<T, R>`	`T -> R`
`Supplier<T>`	`() -> T`
`UnaryOperator<T>`	`T -> T`
`BinaryOperator<T>`	`(T, T) -> T`
`BiFunction<T, U, R>`	`(T, U) -> R`

Method References

Method references allows us to use methods as first class objects.

Instead of writing `filter((Apple a) -> a.getWeight())`, we can just write `filter(Apple::getWeight)`.

Streams

Definition
A **stream** is a sequence of elements from a source that supports aggregate operations.

```
students.stream()
  .filter(student -> student.isFrom("Cambridge"))
  .count();
```

- **Intermediate operations** such as `filter` has type `Stream -> Stream`
- **Terminal operations** such as `count` has type `Stream -> T` where `T` is not a stream.

Stream evaluations are lazy: if a terminal operator is not called, it is not ran.

Benefits of using streams:
- Concise
- Lazy evaluation
- Short circuiting: does not look through the whole collection if the result is found before reaching the end.

END Object Oriented Programming