Java Quick Reference

By: Arsalaan Naeem

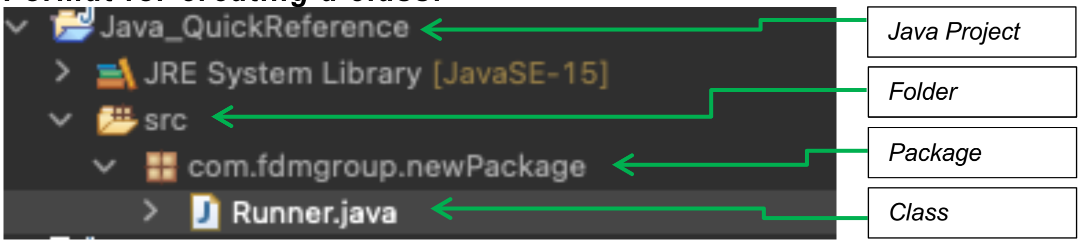
0. Basics (“Hello World!”)

In Java there are a few things you must implement in your program before you can run it. It is not as simple as typing print(‘Hello World!’) in an empty script like in python.

0.1. Classes

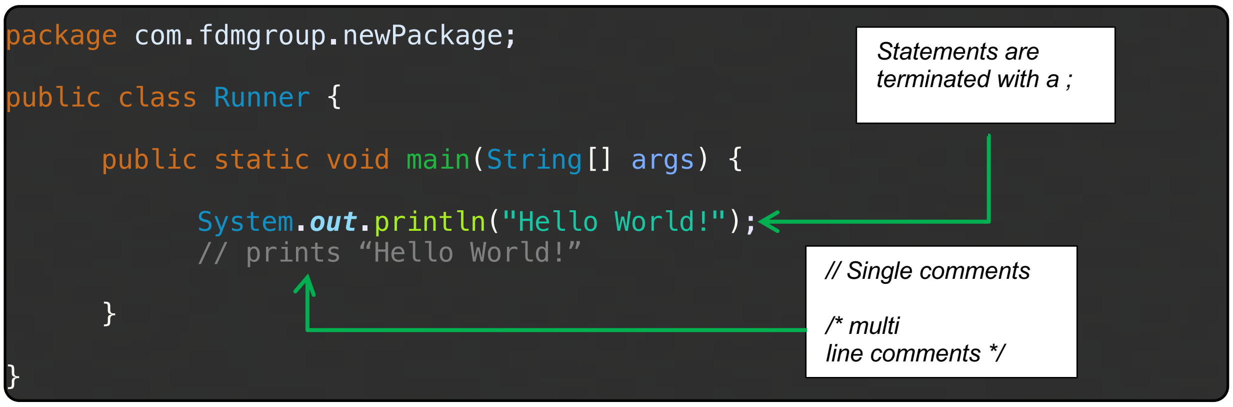
A class represents a single concept. It is advised to create a class in the following format in the project tree:

Format for creating a class:



A Java program must have one class whose name is the same as the program filename.

Runner.java program file:



```
package com.fdmgroup.newPackage;

public class Runner {

      public static void main(String[] args) {

            System.out.println("Hello World!");
            // prints “Hello World!”

      }

}
```

0.2. main() Method

In Java every application must contain a `main()` method, which is the entry point for the application. All other methods are invoked from the `main()` method. The signature of the method is `public static void main(String[] args) { }`. `public` defines the access modifier, and it must be public so Java can execute this method. It must be `static` so the JVM can load the class into memory and call the main method. `void` means the method does not return any value. `main` is the name of the method and is fixed when we create a Java program. `(String[] args)` means the method only accepts a single argument of type string array.

0.3. Print Line

In Java, `System.out.println` must be used to print to the console. `System` is a class from the core library provided by Java. `out` is an object that controls the output. `println()` is a method associated with that object that receives a single argument.

1. Data Types

1.1. Primitive Types

Java's most basic data types are known as primitive data types and are in the system by default.

Data Type	Default Value	Range	Size
`byte`	0	-128 to 127	8 bit
`short`	0	-32,768 to 32,767	16 bit
`int`	0	-2e+9 to 2e+9	32 bit
`long`	0L	-9e+18 to 9e+18	64 bit
`float`	0.0F	(+-)1.4e-45 to (+-)3.4e+38	32 bit
`double`	0.0	(+-)4.9e-324 to (+-)1.8e+308	64 bit
`boolean`	false	false to true	undefined
`char`	'\u0000' (space)	'\u0000' to '\uffff'	16 bit

Example:

```java
int age = 28;

char grade = 'A';

boolean late = true;

byte b = 20;

long num1 = 1234567;

short no = 10;

float k = (float)12.5;

double pi = 3.14;
```

1.2. Object Types

Java contains thousands of built-in object data types. These are more complex than primitive types and contain their own data and functionality. An unlimited number of custom object data types can be created.

1.3. Strings

Strings are an example of a built-in object data type within Java. They are immutable (cannot be changed). They can be created in two ways:

Making a string:

```java
String name = "Dog";

String name1 = new String("Dog");
```

Most common approach

This approach is in accordance with how you create an Object in Java.

There are many String methods one can use with each having their own functionality. Below are some common string methods:

length() – returns total number of characters.

```
      String petName = "Dog";
      int petNameLength = petName.length();

      System.out.println(petNameLength);
      // prints 3
```

concat() – appends one string to the end of another string.

```
      String firstName = "John";
      String lastName = " Smith";

      String fullName = firstName.concat(lastName);

      System.out.println(fullName);
      // prints John Smith
```

equals() and equalsIgnoreCase() – tests for equality between two strings.

```
      String s1 = "Hello";
      String s2 = "World";

      System.out.println(s1.equals("Hello"));
      // prints true

      System.out.println(s2.equals("Hello"));
      // prints false

      System.out.println(s2.equalsIgnoreCase("world"));
      // prints true
```

indexOf() – returns the first occurrence of a character or a substring.

```
      String str = "Hello World!";

      System.out.println(str.indexOf("l"));
      // prints 2

      System.out.println(str.indexOf("Wor"));
      // prints 6

      System.out.println(str.indexOf("z"));
      // prints -1
```

CharAt() – returns the character of a string at a specified index.

```
      String string = "This is a string";

      System.out.println(string.charAt(0));
      // prints 'T'

      System.out.println(string.charAt(15));
      // prints 'g'
```

toUpperCase() and toLowerCase() – returns the string converted to the case.

```
String myString = "Hello World!";

String uppercase = myString.toUpperCase();
// uppercase = "HELLO WORLD!"

String lowercase = myString.toLowerCase();
// lowercase = "hello world!"
```

substring() – returns a new string based on start and (optional) end index.

```
String stringy = "John Smith";

System.out.println(stringy.substring(5));
// prints Smith

System.out.println(stringy.substring(0,4));
// print John
```

char [] toCharArray() – returns a new array of chars from the string.

```
String str1 = "Java";
char[] arrayOfChars = str1.toCharArray();
// array of {'J', 'a', 'v', 'a'}
```

String [] split(String regex, int limit) – returns new array of strings from str.

```
String stringy1 = "Welcome to Java";

String [] words = stringy.split(" ");
// array of {"Welcome", "to", "Java"}

String [] words = stringy.split(" ", 2);
// array of {"Welcome", "to Java"}
```

More string methods available at: https://www.w3schools.com/java/java_ref_string.asp

2. Arrays and Lists

2.1. Arrays

In Java, an **array is used to store a list of elements of the same datatype**. They are fixed in their size and their elements are ordered. They are indexed from 0 and go up to one less than the total length of the array. They are also immutable in size.

Creating an Array:

```java
package com.fdmgroup.newPackage;

import java.util.Arrays;

public class Runner {

      public static void main(String[] args) {

            // create an array of 5 int elements
            int [] age = {22, 21, 24, 25, 20};

            System.out.println(Arrays.toString(age));
            // prints [22, 21, 24, 25, 20] in console

            int [] age1 = new int[5];
            age1[0] = 22;
            age1[1] = 21;
            age1[2] = 24;
            age1[3] = 25;
            age1[4] = 20;

            System.out.println(Arrays.toString(age1));
            // prints [22, 21, 24, 25, 20] in console

      }

}
```

To print array, need to import java.util.Arrays (library) to use the Arrays.toString() method.

Changing an element in an Array:

```java
            int[] nums = {1, 2, 0, 4};

            nums[2] = 3;
            // replaces the element at index 2 in nums array with 3
```

2.2. ArrayList

In Java, an **ArrayList is used to represent a dynamic list**. They are flexible as they allow elements to be added and/or removed (unlike with regular Arrays). To create and utilise this feature we must first import the `ArrayList` package.

Creating an ArrayList:

```
package com.fdmgroup.newPackage;

import java.util.ArrayList;

public class Runner {

      public static void main(String[] args) {

            // ArrayList called students, which initially holds []
            ArrayList<String> students = new ArrayList<String>();

      }

}
```

An ArrayList can easily be modified using built in methods. Some common ArrayList methods are:

sort() – sorts Array/ArrayList in alphabetical (str) or ascending (int) order.

add() – adds an element to the ArrayList.

```
            ArrayList<String> students = new ArrayList<String>();

            // add students to the ArrayList
            students.add("James");
            students.add("Samuel");
            students.add("John");
            students.add("William");
            // students is ["James", "Samuel", "John", "William"]
```

remove(index i) – removes an element to the ArrayList (can specify the index or the element itself)

```
            // remove James, and then Samuel from the ArrayList
            students.remove(0);
            students.remove("Samuel");
```

get(index i) – gets the element at the index.

```
            // gets the first element from ["John", "William"]
            students.get(0);
```

set(index i) – sets the element at the index.

```
            // replace William with Mark
            students.set(1, "Mark");
```

size() – returns the size of the array.

```
            // find length of ["John", "Mark"]
            students.size();
```

toArray() – returns an array filled with the elements in the list.

```
            // convert to array
            students.toArray();
```

NOTE: == and != only apply to primitives.

For all objects (not just strings) you should use object.equals(anotherObject) or !object.equals(anotherObject)

3. Conditionals

In Java, `if`, `else if` and `else` conditional statements are used in the same manner as other programming languages. When we want to execute a piece of code depending on a condition being met, we must use the comparison operators:

Equal to	Not equal to	Greater than	Greater than or equal to	Less than	Less than or equal to
==	!=	>	>=	<	<=

Furthermore, we can include more than one condition in our statements by using the conditional operators:

Conditional-AND	Conditional-OR	Conditional-NOT
&&	||	!

You can also nest conditional statements in Java. A basic example of all three conditional statements being used is shown below. No conditional operators or nested statements are present in the example below, but one can look online for examples.

if, else if and else statements:

```
package com.fdmgroup.newPackage;

public class Runner {

      public static void main(String[] args) {

            int overallGrade = 73;
            String classification;

            if (overallGrade >= 70) {
                  classification = "First-Class Honours";
            }

            else if (overallGrade >= 60) {
                  classification = "Upper Second-Class Honours";

            }

            else if (overallGrade >= 50) {
                  classification = "Lower Second-Class Honours";

            }

            else if (overallGrade >= 40) {
                  classification = "Third-Class Honours";
            }

            else {
                  classification = "Fail";
            }

            System.out.println(classification);
}
```

In Java, you can exclude the else statement and only write the code that was previously inside of it:

else if (…) {
…
}
classification = “Fail”;

3.1. Switch Statements

A `switch` statement **allows a variable to be tested for equality against a list of values**. Each value is called a `case`, and the variable being switched on is checked for each case.

Example:

```java
package com.fdmgroup.newPackage;

public class Runner {

    public static void main(String[] args) {

        automatedPhoneService(1);

    }


    public static void automatedPhoneService(int option) {

        switch(option) {

        case 1:

            System.out.println("You pressed option 1, your account
            balance is ...");
            break;

        case 2:

            System.out.println("You pressed option 2, you want to
            tell us about a chance of name or address");
            break;

        case 3:

            System.out.println("You pressed option 3, you want to
            report a problem with an online payment");
            break;

        default:

            System.out.println("You did not select a valid option,
            the valid options will now be repeated");

        }

    }

}
```

Without the break statement the code will continue through the other cases, even after the case is met.

Default case is optional.
No break is needed in this case.

Notice that a method automatedPhoneService() was created for this example.

To execute this method, the method was called upon in the main() method when the program was run.

4. Loops

Looping, or Iteration, is a feature which enables the **execution of a set of instructions or functions repeatedly while a condition evaluates to true.** Java provides different ways of executing loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

4.1. While Loop

Loops through a block of code as long as a specified condition is true. There is a variation called a Do/While Loop where it will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true. A Do/While Loop example will not be shown here.

While Loop Example:

```
int score = 0;

while (score < 20) {
        score += (int) (Math.random() * 10);
        System.out.println("score is "+score);
}
```

4.2. For Loop

Used to repeat a block of code if the number of iterations is fixed.

For Loop Example:

```
for (int i = 10; i <= 20; i++) {
        System.out.println(i);
}
```

Statement 1: Initialise.

Statement 2: Condition.

Statement 3: Increment Counter.

4.3. For/Each Loop

An enhanced For Loop that will iterate through all the elements of a collection or array.

For/Each Loop Example:

```
String[] allFruits = { "Apple", "Orange", "Banana", "Grape",
"Cherry" };

for (String fruit : allFruits) {
        System.out.println(fruit);
}
```

for (type variableName : collection/array) {
// code
}

4.4. Break & Continue

These two keywords can be used to **control the flow of the loop**. `Break` is used to stop all iterations of a loop. `Continue` stops the current iteration of a loop, and proceeds with the subsequent iterations.

5. Methods

A method is a **block of code which only runs when it is called**. You can pass data known as arguments into a method, the variable that then holds the value is the parameter. Methods are used to perform certain actions, and they are also known as functions.

Why use methods? To reuse code: define the code once and use it many times.

To create a method, it must be declared within a `class`. Below is an example of creating a method that adds two numbers together. To call a method in Java, you need to write the method's name followed by two parentheses () and a semicolon ;

Creating & Calling a Method (void):

```
package com.fdmgroup.newPackage;

public class Runner {

      static void add(int num1, int num2) {

            int sum = num1 + num2;
            System.out.println(sum);

      }

      public static void main(String[] args) {

            add(3, 2);

      }

}
```

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use any data type instead of void, and use the `return` keyword inside the method:

Creating & Calling a Method (return type):

```
public class Runner {

      static int add(int num1, int num2) {

            int sum = num1 + num2;
            return sum;

      }

      public static void main(String[] args) {

            System.out.println(add(3,2));

      }

}
```

To create a method that can accept any number of arguments, must use the varargs technique. Below is an example of the add() method but with any number of arguments:

varargs Method:

```
public class Runner {

      static int add(int... nums ) {
            int sum =0;
            for (int num : nums) {
                  sum += num;
            }
            return sum;
      }

      public static void main(String[] args) {
            System.out.println(add(3,2,5,6,7));
      }

}
```

6. Classes & Objects

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes (variables) and methods. For example: a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake. A Class is like an object constructor, or a "blueprint" for creating objects. To create a class, use the keyword `class`:

Creating a Class:

```java
package com.fdmgroup.newPackage;

public class Car {

	// Attributes (Variables & Constants)
	String companyName;
	String modelName;
	int year;
	int weight;
	String colour;

	// Behaviours/Methods (Functions)
	public void drive() {
		System.out.println("Car is driving.");
	}

	public void brake() {
		System.out.println("Car is braking.");
	}
}
```

In Java, an object is created from a class. We have already created the class named Car, so now we can use this to create objects. To create an object of Car, specify the class name, followed by the object name, and use the keyword `new`.

Creating an Object:

```java
package com.fdmgroup.newPackage;
public class Runner {
	public static void main (String[] args) {

	Car car1 = new Car(); // Empty Object 1
	Car car2 = new Car(); // Empty Object 2
	Car car3 = new Car(); // Empty Object 3

	// Adding Attributes to Object 1
	car1.companyName = "Ford";
	car1.modelName = "Fiesta";
	car1.year = 2013;
	car1.weight = 1200;
	car1.colour = "White";

	// Calling the drive() method from the Car class for Object 1
	car1.drive();

	}
}
```

Think of this as the data type.

Object name

Creating the empty object by calling the class

The format for creating an object for Car is similar to the one for creating ArrayList.

All objects follow this format.

6.1. Constructors

Creating a new object using the approach in the example above can be *repetitive* if you have multiple objects. We can create a custom constructor in the class, **to initialise an objects attributes**.

Creating a Constructor:

```
package com.fdmgroup.newPackage;

public class Car {

	// Attributes (Variables & Constants)
	String companyName;
	String modelName;
	int year;
	int price;
	int weight;
	String colour;

	// Constructor
	public Car(String companyName, String modelName, int year, int p
	rice, int weight, String colour) {
		super();
		this.companyName = companyName;
		this.modelName = modelName;
		this.year = year;
		this.price = price;
		this.weight = weight;
		this.colour = colour;
	}

	// Behaviours/Methods (Functions)
	public void drive() {
		System.out.println("Car is driving.");
	}

	public void brake() {
		System.out.println("Car is braking.");
	}

}
```

Creating an Object (after constructor added):

```
package com.fdmgroup.newPackage;

public class Runner {

	public static void main (String[] args) {

	Car car1 = new Car("Ford", "Fiesta", 2013, 5000, 1200, "White");

	System.out.println(car1.modelName);

	}

}
```

6.2. Static vs. Non-Static

You will often see Java programs that have either `static` or `public` attributes and methods. In the example above, we created a `public` method, which means that it can only be accessed by objects, unlike `static`, which can be accessed without creating an object of the class.

Difference between static and public methods:

```
package com.fdmgroup.newPackage;

public class Runner {

      // Static method
      static void myStaticMethod() {
            System.out.println("Static methods can be called without
            creating objects");
        }

       // Public method
       public void myPublicMethod() {
            System.out.println("Public methods must be called by creating
            objects");
        }

       // Main method
       public static void main(String[] args) {

            myStaticMethod(); // Call the static method
            // myPublicMethod(); This would compile an error

            Runner myObj = new Runner(); // Create an object of Runner
            myObj.myPublicMethod(); // Call the public method on the
            object
        }

}
```

In Java, static variables and static constant attributes have slightly different meanings.

The static variable can be used to refer to the common property of all objects (which is not unique for each object). In the example below, it is used as a counter to actively update the number of parking spaces available, as a new parking space object is created.

The static constant attribute is where a value has a hardcoded constant value. It allows the programmer to create a constant within a class and call it elsewhere without the need of creating an instance of that class. Example below demonstrates using the value of pi within a class called Circle.

Creating a static variable:

```
public class ParkingSpace {

      private static int numberOfAvailableSpaces = 0;

      public ParkingSpace() {
            numberOfAvailableSpaces ++;
      }

      public static int getNumberOfSpaces() {
            return numberOfAvailableSpaces;
      }

}
```

Variable is the same for ALL objects

Each time an object is created, the counter for the static variable increments.

This updated numberOfAvailableSpaces value is shared amongst ALL objects.

Creating a static constant attribute (final keyword):

```
public class Circle {

      private final static double PI = 3.1415;

      public static double getPi() {
            return PI;
      }

      // System.out.println(Circle.getPi()); in the main method will
return 3.1415

}
```

Final keyword used to make an attribute constant.

Can be used at class or method level and do NOT have setter methods.

6.3. Access & Non-Access Modifiers

6.3.1. Access Modifiers

Classes

Modifier	Description
`public`	The class is accessible by any other class.
`default`	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

Attributes, Methods and Constructors

Modifier	Description
`public`	The code is accessible for all classes.
`private`	The code is only accessible within the declared class.
`default`	The code is only accessible in the same package. This is used when you don't specify a modifier.
`protected`	The code is accessible in the same package and subclasses.

6.3.2. Non-Access Modifiers

Classes

Modifier	Description
`final`	The class cannot be inherited by other classes.
`abstract`	The class cannot be used to create objects. (To access an abstract class, it must be inherited from another class)

Attributes, Methods and Constructors

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified.
`static`	Attributes and methods belongs to the class, rather than an object.
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example abstract void run();. The body is provided by the subclass (inherited from).
`transient`	Attributes and methods are skipped when serialising the object containing them.
`synchronised`	Methods can only be accessed by one thread at a time.
`volatile`	The value of an attribute is not cached thread-locally and is always read from the "main memory".

6.4. Encapsulation (Getters & Setters) [Pillar 2 of OOD]

The meaning of Encapsulation is to **make sure that "sensitive" data is hidden from users**. It allows better control of class attributes and methods, increased security of data, and allows the programmer to make changes to one part of the code without affecting other parts.

A class groups together related attributes and controls the access to its attributes. The class also controls the values which can be given to its attributes. To achieve this, you must declare class attributes as private and provide public get and set methods to access and update the value of a private variable.

Get & Set Example:

```java
public class Example {

      // Attribute
      private String name;

      // Getter
      public String getName() {
            return name;
      }

      // Setter
      public void setName(String newName) {
            this.name = newName;
      }

}
```

Notice that no constructor was created for this class.

So the object created was empty, and we added the arguments afterwards.

```java
public class Runner {

      public static void main(String[] args) {

            // Creating empty Object
            Example person1 = new Example();

            // Setting the value of the name attribute
            person1.setName("John");

            // Getting (Accessing) the value of the name attribute
            System.out.println(person1.getName());

      }
}
```

6.5. Example of a Class (Complete)

Movie.jar

```
package com.fdmgroup.newPackage;

public class Movie {

        // Attributes
        private String title;
        private String director;
        private String rating;

        // Constructor
        public Movie(String title, String director, String rating) {
                this.title = title;
                this.director = director;
                this.rating = rating;
        }

        // Getters & Setters
        public String getTitle() {
                return title;
        }

        public void setTitle(String title) {
                this.title = title;
        }

        public String getDirector() {
                return director;
        }

        public void setDirector(String director) {
                this.director = director;
        }

        public String getRating() {
                return rating;
        }

        public void setRating(String rating) {

                if (rating.equals("G") || rating.equals("PG") || rating.equals("PG-
                13")) {
                        this.rating = rating;
                }
                this.rating = "N/A";

        }

}
```

ALL attributes private so Getters needed for all of them.

In many cases, not doing anything special BUT this controls the access and allows for easy manipulation for the future.

Method level attribute

Class level attribute

Controlling the values of rating.

Can only assign specific values, if any invalid value entered then a default "N/A" value will be used instead.

Runner.jar

```
public class Runner {

        public static void main(String[] args) {

                Movie mov1 = new Movie("Harry Potter", "Chris Columbus", "PG-13");
                System.out.println(mov1.getRating());

        }
}
```

7. Inheritance [Pillar 3 of OOD]

7.1. Classes

In Java, it is possible to **inherit attributes and methods from one class to another**. This is particularly useful for code reusability. We group the "inheritance concept" into two categories:

- subclass (child) - the class that inherits from another class
- superclass (parent) - the class being inherited from

To inherit from a class, use the `extends` keyword. In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

Inheritance of a Class Example (extends):

```java
package com.fdmgroup.newPackage;

class Vehicle {

      protected String brand = "Ford";          // Vehicle attribute
      public void honk() {                      // Vehicle method
      System.out.println("Tuut, tuut!");

        }

}

class Car extends Vehicle {

      private String modelName = "Mustang"; // Car attribute

      public static void main(String[] args) {

      // Create a myCar object
      Car myCar = new Car();

      // Call the honk() method (from the Vehicle class) on the myCar
      object
      myCar.honk();

      // Display the value of the brand attribute (from the Vehicle class)
      and the value of the modelName from the Car class
      System.out.println(myCar.brand + " " + myCar.modelName);

      }

}
```

We set the brand attribute in Vehicle to a `protected` access modifier. If it was set to `private`, the Car class would not be able to access it.

NOTE:
Using the `final` keyword prevents a class from being inherited. A class can only have 1 parent BUT can have multiple children. Constructors are not inherited.

7.1.1. Super keyword

The super keyword refers to superclass (parent) objects. It is **used to call superclass methods, and to access the superclass constructor.**

Using super. to access parent methods:

```java
class Animal { // Superclass (parent)

      public void animalSound() {

            System.out.println("The animal makes a sound");

        }
}

class Dog extends Animal { // Subclass (child)

      public void animalSound() {

            super.animalSound(); // Call the superclass method
            System.out.println("The dog says: bow wow");

        }
}

public class Main {

      public static void main(String args[]) {

            Animal myDog = new Dog(); // Create a Dog object
            myDog.animalSound(); // Call the method on the Dog object

        }
}
```

Using super() to access and pass arguments to parent constructor:

Bicycle.java:

```java
public class Bicycle {

      // the Bicycle class has three fields
      public int cadence;
      public int gear;
      public int speed;

      // the Bicycle class has one constructor
      public Bicycle(int startCadence, int startSpeed, int startGear) {
            gear = startGear;
            cadence = startCadence;
            speed = startSpeed;
      }

      // the Bicycle class has four methods
      public void setCadence(int newValue) {
            cadence = newValue;
```

```
    }

    public void setGear(int newValue) {
          gear = newValue;
    }

    public void applyBrake(int decrement) {
          speed -= decrement;
    }

    public void speedUp(int increment) {
          speed += increment;
    }

}
```

MountainBike.java:

```
public class MountainBike extends Bicycle {

      // the MountainBike subclass adds one field
      public int seatHeight;

      // the MountainBike subclass has one constructor
      public MountainBike(int startHeight, int startCadence, int
      startSpeed, int startGear) {

      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;

    }

      // the MountainBike subclass adds one method
      public void setHeight(int newValue) {
            seatHeight = newValue;
    }
}
```

The syntax for calling a superclass constructor is super(); or super(parameter list);

With super(), the superclass no-argument constructor is called.

With super(parameter list), the superclass constructor with a matching parameter list is called.

8. Abstraction [Pillar 1 of OOD]

Data abstraction is **the process of hiding certain details and showing only essential information to the user**. Abstraction can be achieved with either abstract classes or interfaces.

8.1. **Abstract Classes & Methods**

The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Why and when to use Abstract Classes and Methods? To achieve security - hide certain details and only show the important details of an object.

Creating an abstract class (with both abstract and regular methods):

Animal.java:

```java
public abstract class Animal {

	// Abstract method (no body)
	public abstract void animalSound();

	// Regular method
	public void sleep() {
		System.out.println("ZzzZzzZZzz");
	}

}
```

Accessing an abstract class and overriding the abstract method:

Dog.java:

```java
public class Dog extends Animal {

	@Override
	public void animalSound() {

		// The body of animalSound() is provided here
		System.out.println("The dog says: woof woof");
	}

}
```

Runner.java:

```
public class Runner {

      public static void main(String[] args) {

      // Creating objects
      // Cannot create "Animal myAnimal = new Animal();" object from
      abstract class
      Animal myDog = new Dog();

      // Calling the overridden abstract method and regular method from
      child class.
      myDog.animalSound();
      myDog.sleep();

      }
}
```

8.2. Interfaces

Another way to achieve abstraction in Java, is with interfaces. An interface is a **completely "abstract class" that is used to group related methods with empty bodies**:

Creating an Interface Example:

```
interface Animal {

      // interface methods (no body)
      public void animalSound();
      public void run();

}
```

To access the interface methods, the interface must be "implemented" (similar to inherited) by another class with the `implements` keyword (instead of extends). **The body of the interface method is provided by the "implement" class**:

Implementing an Interface Example:

```
// Interface
interface Animal {

      // interface method (does not have a body)
      public void animalSound();
      public void sleep();

}

// Bird "implements" the Animal interface
class Bird implements Animal {

      public void animalSound() {
            // The body of animalSound() is provided here
            System.out.println("The bird says: tweet tweet");
}
```

```java
      public void sleep() {
            // The body of sleep() is provided here
            System.out.println("Zzz");
      }
}

// Main method
class Main {

      public static void main(String[] args) {
            // Create a Bird object
            Bird myBird = new Bird();
            myBird.animalSound();
            myBird.sleep();
      }
}
```

Why and when to use Interfaces?
1) To achieve security - hide certain details and only show the important details of an object (interface).
2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces because the class can implement multiple interfaces.

Multiple Interfaces:

```java
interface FirstInterface {
      public void myMethod(); // interface method
}

interface SecondInterface {
      public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
      public void myMethod() {
            System.out.println("Some text..");
      }
      public void myOtherMethod() {
            System.out.println("Some other text...");
      }
}

class Main {
      public static void main(String[] args) {
      DemoClass myObj = new DemoClass();
      myObj.myMethod();
      myObj.myOtherMethod();
      }
}
```

To implement multiple interfaces, separate them with a comma

NOTE:
Interfaces cannot be used to create objects. Interface methods do not have a body. On implementation of an interface, you must override all of its methods. Interface methods are by default `abstract` and `public`. Interface attributes are by default `public`, `static` and `final`. An interface cannot contain a constructor.

9. Polymorphism [Pillar 4 of OOD]

Polymorphism means “many forms”, and it occurs when we have many classes that are related to each other by inheritance. Inheritance lets us inherit attributes and methods from another class. Polymorphism **uses those methods to perform different tasks**. This allows us to perform a single action in different ways. There are two types of polymorphism: Overriding and Overloading.

9.1. Overriding

This is when **code in the child class method completely replaces the code from the parent class method** that it overrides.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Cats, Dogs, Birds etc. – and they also have their own implementation of an animal sound (cat meows, dog barks, etc.):

Animal.java

```java
public class Animal {

      public void animalSound() {
            System.out.println("The animal makes a sound");
      }

}
```

Dog.java

```java
public class Dog extends Animal {

      public void animalSound() {
            System.out.println("The dog says: woof woof");
      }

}
```

Use extends keyword to inherit from a class

Cat.java

```java
public class Cat extends Animal {

      @Override
      public void animalSound() {
            System.out.println("The cat says: meow");
      }

}
```

Can add @Override annotation to check method is being overridden.

Runner.java

```java
public class Runner {

	public static void main(String[] args) {

	// Creating objects
	Animal myAnimal = new Animal();
	Animal myDog = new Dog();
	Animal myCat = new Cat();

	// Calling the same method() for each class.
	myAnimal.animalSound();
	myDog.animalSound();
	myCat.animalSound();

	}
}
```

Each print out different statements!

NOTE:

- To access a method from the parent class (that has been overridden in the child class), we must use the `super` keyword (as shown in 7.1.1 Super keyword section).
- Using the `final` keyword in a method header will prevent the method from being overridden in any child classes (if `final` already stated in class header, then no need for `final` in method header).
- Overriding the `equals()` and `hashcode()` methods is sometimes necessary to ensure they behave as ‘expected’ when comparing objects. Eclipse has a built-in option to enable this feature.

9.2. Overloading

This is when a class has **multiple methods with the same name but with different arguments**. Each version of the method will have slightly different functionality. The version called depends on the arguments passed in.

Overloading Method Example:

```java
public class Car {
      public void accelerate() {}

      public void accelerate(int speedLimit) {}

      public void accelerate(double hillGradient) {}
}
```

accelerate(40) will call the first method accelerate(2.5) will call the second method

Overloading Constructors:

```java
public class Car {

      public Car() {}

      public Car(String model) {}
}
```

A car object can now be created by calling either constructor:
Car car1 = new Car();
Car car2 = new Car("Tesla");

Horizontal Constructor Chaining:

```java
public class Car {

      public Car(int maxSpeed){
            // code to check maxSpeed is valid number
      }

      public Car(int maxSpeed, String model) {
            this(maxSpeed);
            this.model = model;
      }

}
```

Arguments passed to a different constructor in the same class using this().

this() used to avoid code duplication (instead of copying code from first constructor again)

10. Casting

Casting is **the act of storing one data type into a different data type**.

Why use Casting? It allows for multiple data types to be used for one method, accessing specific versions of a method in subclasses/superclasses, and used for storing multiple types of data in a collection.

10.1. **Primitive Casting**

Primitives are variables that hold raw data and have a certain number of bytes that can be stored into the data type. Primitive casting is **the process of converting one primitive data type to another.** There are two types: variable widening and variable narrowing.

10.1.1. Variable Widening

This is when a smaller primitive type value is automatically accommodated in a larger/wider primitive data type. In Java, this occurs automatically (but can also be done explicitly).

Widening Example:

```
public class CastingExamples {
      public static void main (String[] args) {

            int num = 10; // num = 10
            double numDouble = num; // (implicit) numDouble = 10.0
            float numFloat = (float) num; // (explicit) numFloat = 10.0

      }
}
```

10.1.2. Variable Narrowing

This is when a wider/bigger primitive type value is converted to a smaller primitive type value. This can only be carried out explicitly.

Narrowing Example:

```
public class CastingExamples {
      public static void main (String[] args) {

            double num = 10.5;
            // num = 10.5

            short numShort = (short) num; // explicit
            // numShort = 10

            int numInt = (int) num; // explicit
            // numInt = 10
      }

}
```

10.2. **Upcasting**

This is when a subclass object reference is assigned to a wider superclass object reference. This is synonymous with widening. It is the **act of casting to a supertype**. Upcasting is always allowed and occurs automatically (but can also be done explicitly).

Upcasting Example:

```
//Superclass
class Parent {
      public void message() {
            System.out.println("message from Parent");
      }
}

//Subclass
class Child extends Parent {
      public void message() {
            System.out.println("message from Child");
      }
}

public class UpcastingExample1 {

      //Main Method
      public static void main(String[] args) {

            Child child = new Child();
            Parent parent = child;
            parent.message(); // prints "message from Child"

            }
}
```

reference of a subclass type is widened to a reference of superclass type (implicitly).

10.3. **Downcasting**

This is when a superclass reference is assigned to a subclass reference (during inheritance). This is synonymous with narrowing. It is the **act of casting to a subtype**. Downcasting can only be carried out explicitly.

Downcasting Example:

```
//Superclass
class Parent {
      public void message() {
            System.out.println("message from Parent");
      }
}

//Subclass
class Child extends Parent1 {
      public void message() {
            System.out.println("message from Child");
      }

}

public class UpcastingExample1 {

      //Main Method
      public static void main(String[] args) {

            Parent parent = new Child();
            Child child = (Child)parent;
            child.message(); // prints "message from Child"

            }
}
```

An object of the subclass is referenced by a reference of the superclass.

A reference of the superclass type is downcasted to the reference of the subclass type.

NOTE:
The code above assumes that we know which type to downcast to. Usually this won't be the case. The `instanceof` keyword is used to determine the correct type of an object. This is especially useful when downcasting objects stored in an ArrayList of the parent type.

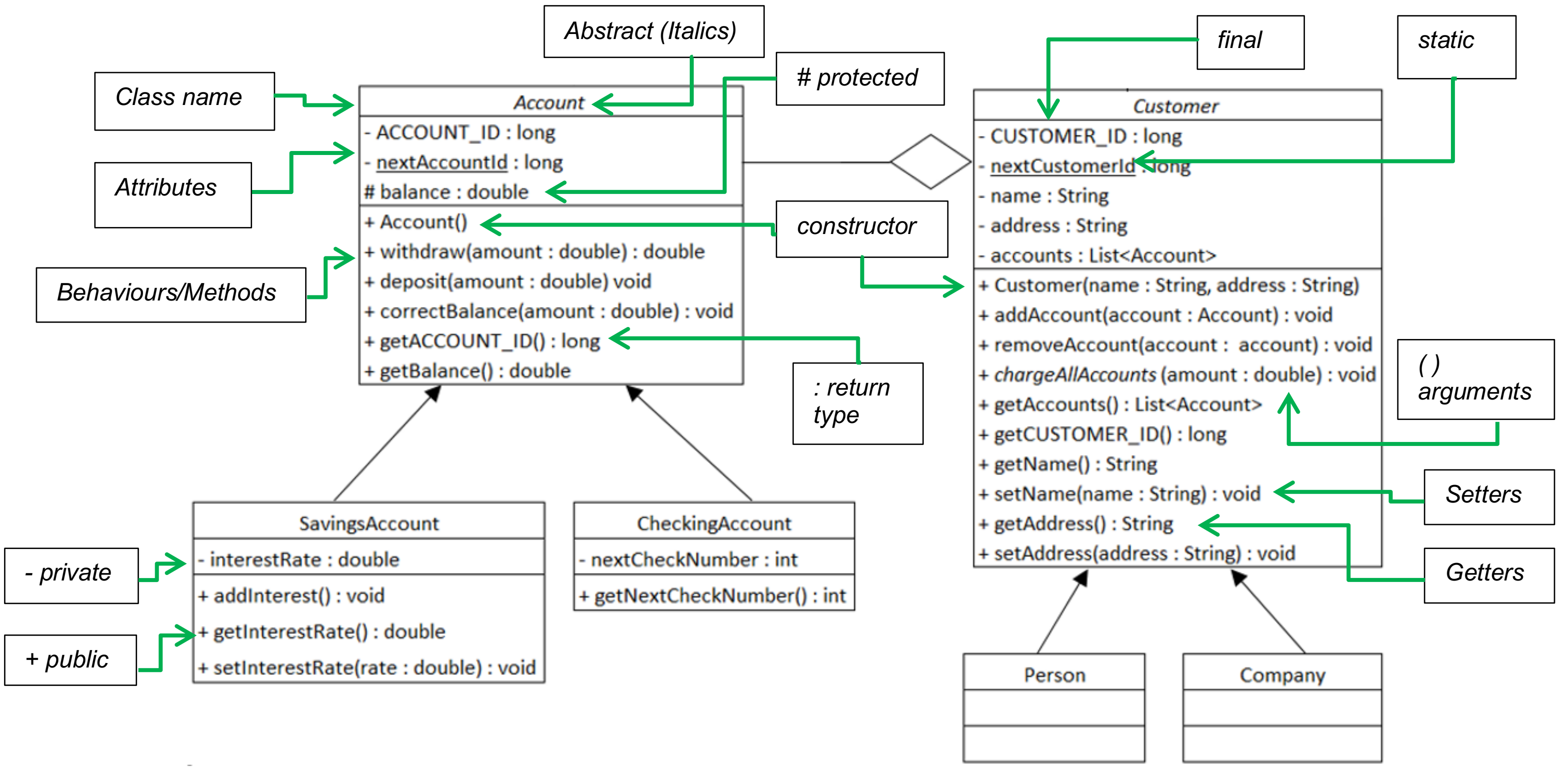
instanceof Example:

```
if (vehicle instanceof Car){
      Car car = (Car) vehicle;
      // code to call car methods
}

if (vehicle instanceof Plane){
      Plane plane = (Plane) vehicle;
      // code to call plane methods
}
```

11. UML Class Diagrams & Dependencies

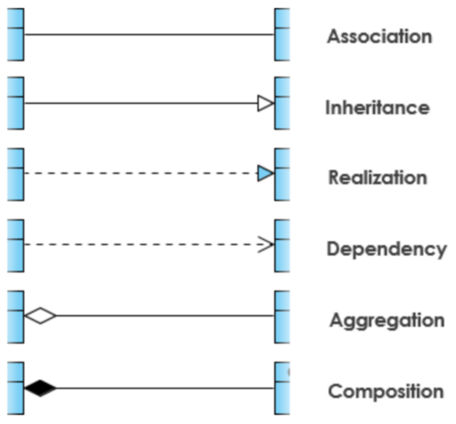
11.1. UML

The UML Class diagram is a **graphical notation used to construct and visualize object-oriented systems**. A class diagram in the Unified Modelling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's: classes, their attributes, behaviours (or methods), and the relationships among objects. Below is a UML diagram with its features labelled.

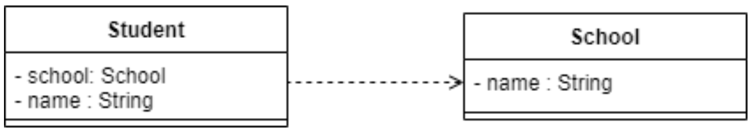

Abstract (Italics)
Class name
Attributes
Behaviours/Methods
# protected
constructor
: return type
- private
+ public
final
static
( ) arguments
Setters
Getters
Account
- ACCOUNT_ID : long
- nextAccountId : long
# balance : double
+ Account()
+ withdraw(amount : double) : double
+ deposit(amount : double) void
+ correctBalance(amount : double) : void
+ getACCOUNT_ID() : long
+ getBalance() : double
Customer
- CUSTOMER_ID : long
- nextCustomerId : long
- name : String
- address : String
- accounts : List<Account>
+ Customer(name : String, address : String)
+ addAccount(account : Account) : void
+ removeAccount(account : account) : void
+ chargeAllAccounts (amount : double) : void
+ getAccounts() : List<Account>
+ getCUSTOMER_ID() : long
+ getName() : String
+ setName(name : String) : void
+ getAddress() : String
+ setAddress(address : String) : void
SavingsAccount
- interestRate : double
+ addInterest() : void
+ getInterestRate() : double
+ setInterestRate(rate : double) : void
CheckingAccount
- nextCheckNumber : int
+ getNextCheckNumber() : int
Person
Company

11.2. **Dependencies**

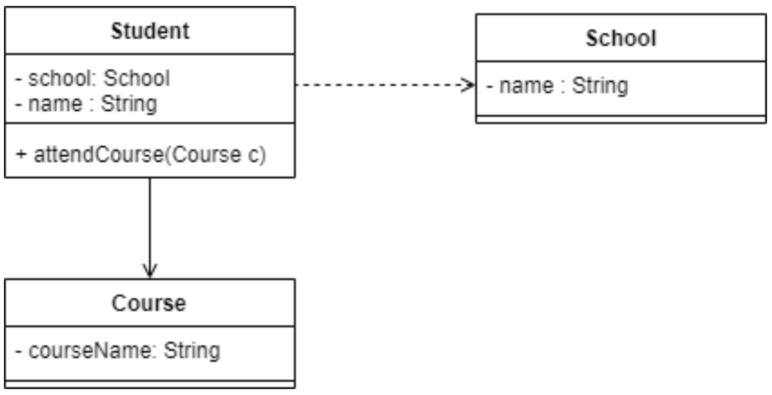
A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:



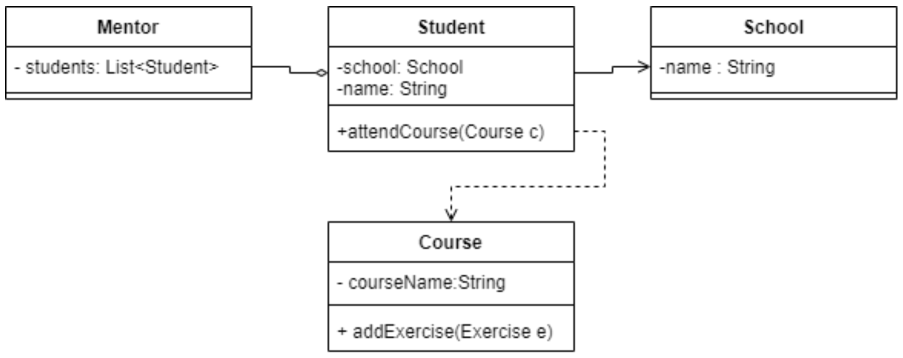
Association: One class has an object of another class as an attribute.



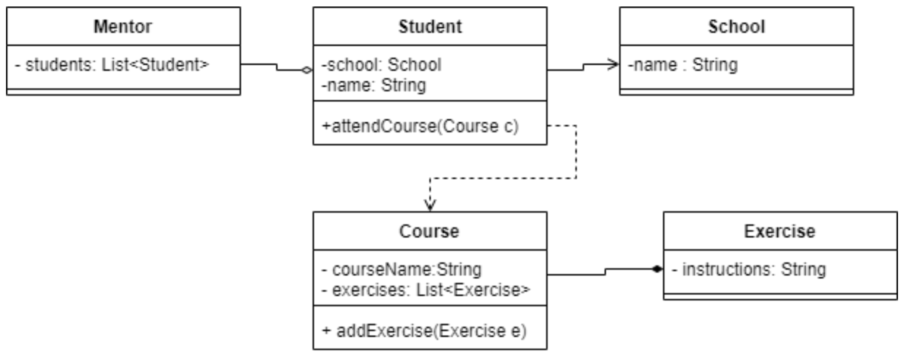
Dependency: One class uses an object of another class as a method argument or return type.



Aggregation: One class has multiple objects of another class as attributes.

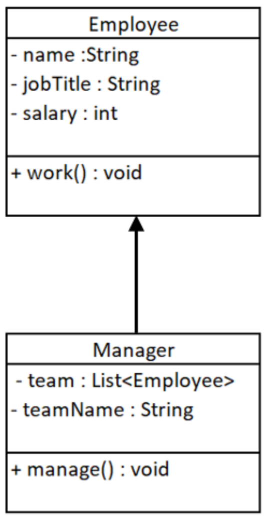


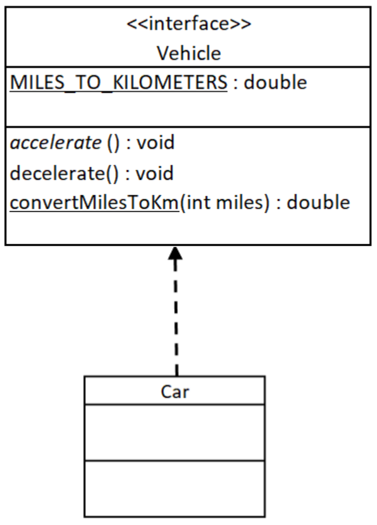
Composition: An object of one class belongs to an object of another class (And cannot exist without it).



Inheritance: One class (child) acquires the attributes and behaviours of another class (parent).

Realization: A class can inherit from one or more interfaces.

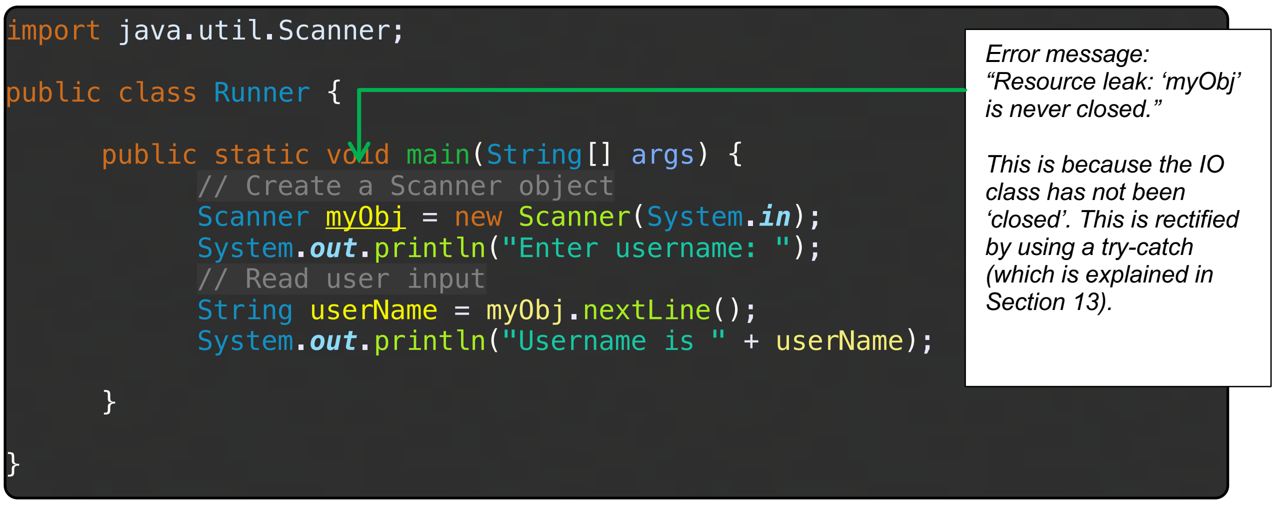



<<interface>>
Vehicle
MILES_TO_KILOMETERS : double
accelerate () : void
decelerate() : void
convertMilesToKm(int miles) : double
Car

12. User Input (Scanner)

The `Scanner` class is **used to get user input**, and it is found in the `java.util` package. To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the `nextLine()`method, which is used to read Strings:

Scanner example:



```
import java.util.Scanner;

public class Runner {

	public static void main(String[] args) {
		// Create a Scanner object
		Scanner myObj = new Scanner(System.in);
		System.out.println("Enter username: ");
		// Read user input
		String userName = myObj.nextLine();
		System.out.println("Username is " + userName);

	}

}
```

Error message: "Resource leak: 'myObj' is never closed."

This is because the IO class has not been 'closed'. This is rectified by using a try-catch (which is explained in Section 13).

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Input Types

Method	Description
`nextBoolean()`	Reads a boolean value from the user.
`nextByte()`	Reads a byte value from the user.
`nextDouble()`	Reads a double value from the user.
`nextFloat()`	Reads a float value from the user.
`nextInt()`	Reads a int value from the user.
`nextLine()`	Reads a String value from the user.
`nextLong()`	Reads a long value from the user.
`nextShort()`	Reads a short value from the user.

13. Exceptions (Try… Catch)

When executing Java code, different errors can occur such as coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: **Java will throw an exception** (throw an error).

13.1. Try and Catch

The `try` statement allows you to define a block of **code to be tested** for errors while it is being executed. The `catch` statement allows you to define a block of **code to be executed if an error occurs** in the try block. The `try` and `catch` keywords come in pairs:

Try and Catch Syntax:

```
            try {
                //  Block of code to try
                }
                catch(Exception e) {
                //  Block of code to handle errors
                }
```

Example of Error Catching:

```
public class Runner {

      public static void main(String[] args) {

            try {
                  int[] myNumbers = {1, 2, 3};
                  System.out.println(myNumbers[10]);
            }
            catch (Exception e) {
                  System.out.println("Something went wrong.");
            }

      }

}
```

This will generate an error, because myNumbers[10] does not exist.

Now the output will be: 'Something went wrong.'

As opposed to the generic error message.

13.2. Finally

The `finally` statement lets you **execute code after** `try...catch`, regardless of the result:

Example of using Finally:

```
public class Runner {

    public static void main(String[] args) {

        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]);
        } catch (Exception e) {
            System.out.println("Something went wrong.");
        } finally {
            System.out.println("The 'try catch' is finished.");
        }

    }

}
```

Now the output will be:
‘Something went wrong.’
‘The ‘try catch’ is finished.’

13.3. Throw

The `throw` statement allows you to **create a custom error** (red error message). The throw statement is used together with an exception type.

There are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `SecurityException`, etc:

Example of using Throw:

```
public class Runner {

    static void checkAge(int age) {

        if (age < 18) {
            throw new ArithmeticException("Access denied – You must
            be at least 18 years old.");
        }
        else {
            System.out.println("Access granted – You are old
            enough!");
        }

    }

    public static void main(String[] args) {

        checkAge(15); // Set age to 15 (which is below 18...)

    }

}
```

Now the output will be:
Exception in thread “main” java.lang.ArithmeticException: Access denied – You must be at least 18…

14. Collections

As seen when comparing Array's to ArrayList's, the *benefit of using a Collection is that they have methods*. Java provides a Collections Framework with some classes and interfaces, these include:

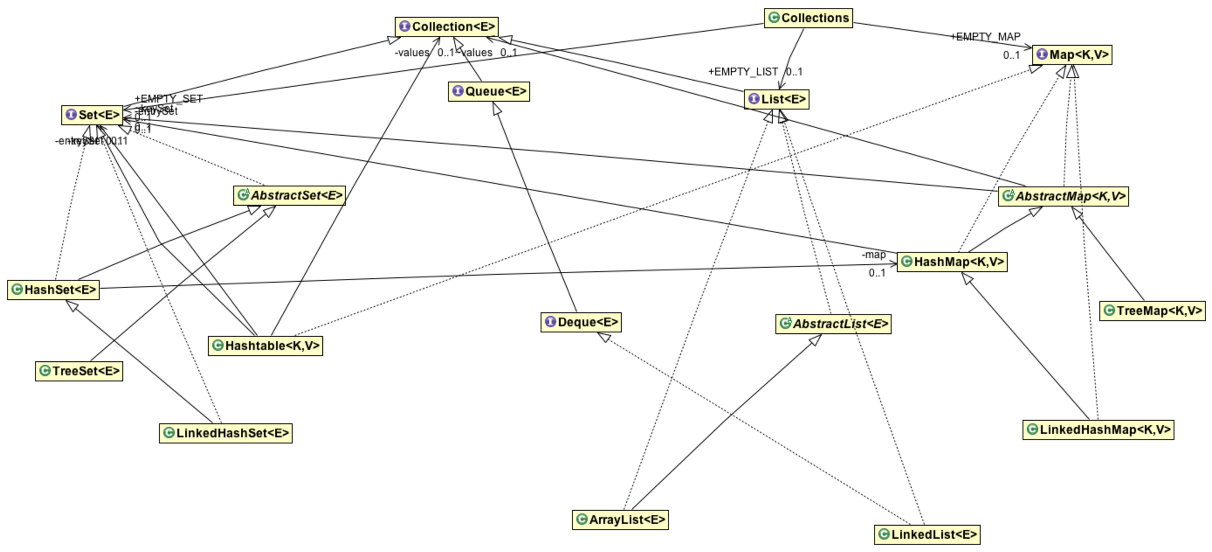
Collections Framework

Interfaces	Classes
`List`	ArrayList, Vector
`Set`	HashSet, TreeSet
`Map`	HashMap, TreeMap

A **collection is either ordered (Lists) or unordered (Sets and Maps). Ordered collections can either be sorted or unsorted**. We know how to sort alphabetically and numerically, but how do you sort an object? There is no natural order unless we use an interface called Comparable (Section 15.2) that defines how instances of a class can compare to each other.

Why use Java Collections Framework? Reduced development effort (comes with all common types of collections and useful methods), better quality (well-tested), reusability, *interoperability*, and reduced maintenance effort (as everybody knows Collection API classes).

The below class diagram shows the Collections Framework hierarchy:



For simplicity, a summary table of some of the interfaces will be provided but only LinkedList, HashSet and HashMap will have dedicated sections.

14.1. **Summary**

Collections Summary

Type	Collection	Duplicates	Ordered	Sorted
`Set`	HashSet	N	N	N
`Set`	LinkedHashSet	N	Y	N
`Set`	TreeSet	N	Y	Y
`List`	ArrayList	Y	Y	N
`List`	Vector	Y	Y	N
`List, Queue`	LinkedList	Y	Y	N
`Queue`	PriorityQueue	Y	Y	Y
`Map`	HashMap	Y	N	N
`Map`	TreeMap	Y	Y	Y
`Map`	LinkedHashMap	Y	Y	N
`Map`	HashTable	Y	N	N

NOTE:
Sets do not allow duplicates. Maps do not allow duplicate keys but they do allow duplicate values. HashSets, HashMaps and HashTables are not ordered. TreeSets, PriorityQueues and TreeMaps are sorted.

14.2. **LinkedList**

The LinkedList class is almost identical to the ArrayList.

Creating a LinkedList:

```
import java.util.LinkedList;

public class Runner {

      public static void main(String[] args) {
            LinkedList<String> cars = new LinkedList<String>();
            cars.add("Volvo");
            cars.add("BMW");
            cars.add("Ford");
            cars.add("Mazda");
            System.out.println(cars);
            }

}
```

14.2.1. ArrayList vs. LinkedList

The `ArrayList` class and the `LinkedList` class can be used in the same way, but they are built very differently.

The `ArrayList` class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, **larger array is created to replace the old one and the old one is removed**.

The `LinkedList` stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, **the element is placed into a new container and that container is linked to one of the other containers in the list**.

It is best to use an `ArrayList` when:

- You want to access random items frequently
- You only need to add or remove elements at the end of the list

It is best to use a `LinkedList` when:

- You only use the list by looping through it instead of accessing random items
- You frequently need to add and remove items from the beginning, middle or end of the list

14.2.2. LinkedList Methods

For many cases, the `ArrayList` is more efficient as it is common to need access to random items in the list, but the `LinkedList` provides several methods to do certain operations more efficiently:

LinkedList Methods

Method	Description
`addFirst()`	Adds an item to the beginning of the list.
`addLast()`	Add an item to the end of the list.
`removeFirst()`	Remove an item from the beginning of the list.
`removeLast()`	Remove an item from the end of the list.
`getFirst()`	Get the item at the beginning of the list.
`getLast()`	Get the item at the end of the list.

14.3. **HashMap**

In the `ArrayList` chapter (Section 2.2), you learned that Arrays store items as an ordered collection, and you have to access them with an index number (`int` type). A `HashMap` however, **store items in "key/value" pairs, and you can access them by an index of another type** (e.g. a `String`).

One object is used as a key (index) to another object (value). It can store different types: `String` keys and `Integer` values, or the same type, like: `String` keys and `String` values:

Creating and Modifying a HashMap:

```java
public class Runner {

      public static void main(String[] args) {

          // Create a HashMap object called capitalCities
          HashMap<String, String> capitalCities = new HashMap <String,
          String>();

          // Add keys and values (Country, Cities)
          capitalCities.put("England", "London");
          capitalCities.put("Germany", "Berlin");
          capitalCities.put("Norway", "Oslo");
          capitalCities.put("USA", "Washington DC");
          System.out.println(capitalCities);

          // Loop Through a HashMap (For each loop)
          // Print keys
          for (String i : capitalCities.keySet()) {
            System.out.println(i);
          }

          // Print values
          for (String i : capitalCities.values()) {
            System.out.println(i);
          }

          // Print keys and values
          for (String i : capitalCities.keySet()) {
            System.out.println("key: " + i + " value: " +
            capitalCities.get(i));
          }

          // Find size of HashMap
          capitalCities.size();
          // Access then Remove item
          capitalCities.get("England");
          capitalCities.remove("England");
          // Remove all items
          capitalCities.clear();


      }


}
```

Keys and values in a HashMap are actually objects.

14.4. **HashSet**

A HashSet is a **collection of items where every item is unique**, and it is found in the `java.util` package:

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper (Section 16) class: `Integer`.

Creating and Modifying a HashSet:

```java
public class Runner {

      public static void main(String[] args) {

          // Create a HashSet object called cars
          HashSet<String> cars = new HashSet <String>();

          // Add items to Set
          cars.add("Volvo");
          cars.add("BMW");
          cars.add("Ford");
          cars.add("BMW");
          cars.add("Mazda");
          System.out.println(cars);

          // Loop Through a HashSet (For each loop)
          for (String i : cars) {
              System.out.println(i);
          }

          // Find size of HashMap
          cars.size();

          // Check if an item exists
          cars.contains("England");

          // Remove an item
          cars.remove("England");

          // Remove all items
          cars.clear();

      }

}
```

even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

14.5. **Iterator**

An `Iterator` is an **object that can be used to loop through collections**, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

Why use an Iterator? Trying to remove items (from an ArrayList) using a for loop or a for-each loop would not work correctly because the collection is changing size at the same time that the code is trying to loop.

To use an Iterator, you must import it from the `java.util` package.

The `iterator()` method can be used to get an Iterator for any collection. To loop through a collection, use the `hasNext()` and `next()` methods of the `Iterator`:

Creating and Looping through an Interator:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Runner {

      public static void main(String[] args) {

          // Make a collection
          ArrayList<String> cars = new ArrayList<String>();
          cars.add("Volvo");
          cars.add("BMW");
          cars.add("Ford");
          cars.add("Mazda");

          // Get the iterator
          Iterator<String> it = cars.iterator();

          // Print the first item
          System.out.println(it.next());

          // Loop through a Collection
          while (it.hasNext()) {
            System.out.println(it.next());
          }

      }

}
```

Iterators are designed to easily change the collections that they loop through. The `remove()` method can remove items from a collection while looping.

Looping Through a Collection:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Runner {
      public static void main(String[] args) {

            ArrayList<Integer> numbers = new ArrayList<Integer>();
          numbers.add(12);
          numbers.add(8);
          numbers.add(2);
          numbers.add(23);

          Iterator<Integer> it = numbers.iterator();

          while(it.hasNext()) {

            Integer i = it.next();

            if(i < 10) {

                  it.remove();

            }

          }

          System.out.println(numbers);

      }

}
```

15. Comparisons

There are many different sorting algorithms: quick sort, merge sort, tim sort, insertion sort, selection sort, bubble sort. All sorting algorithms work by comparing pairs of elements. Java chooses the most appropriate algorithm for the collection or array which it is sorting.

However, Java must be told how to compare a pair of objects of a custom class. Comparable and Comparator both **are interfaces and can be used to sort collection elements by defining the sorting order** for custom classes. Below is an example of trying to use `Collections.sort` on objects and as expected, it results in a compile-time error.

Player.java

```java
public class Player {

      // Attributes
      private int ranking;
      private String name;
      private int age;

      // Constructor
      public Player(int ranking, String name, int age) {
            this.ranking = ranking;
            this.name = name;
            this.age = age;
      }

      // Getters & Setters
      public int getRanking() {
            return ranking;
      }

      public void setRanking(int ranking) {
            this.ranking = ranking;
      }

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public int getAge() {
            return age;
      }

      public void setAge(int age) {
            this.age = age;
      }

}
```

PlayerSorter.java

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PlayerSorter {

   public static void main(String[] args) {

      // Empty footballTeam list
     List<Player> footballTeam = new ArrayList<>();

     // Creating players (objects)
     Player player1 = new Player(93, "Messi", 34);
     Player player2 = new Player(92, "Ronaldo", 36);
     Player player3 = new Player(90, "van Dijk", 30);
     Player player4 = new Player(91, "Oblak", 28);
     Player player5 = new Player(89, "Ramos", 35);

     // Adding players to 5-a-side footballTeam
     footballTeam.add(player1);
     footballTeam.add(player2);
     footballTeam.add(player3);
     footballTeam.add(player4);
     footballTeam.add(player5);

     // Printing out the footballTeam before sort
     System.out.println("Before sorting: ");
     for (Player player : footballTeam) {

      System.out.println(player.getRanking() + " " +
                               player.getName() + " " +
                               player.getAge());

     }

     Collections.sort(footballTeam);

     // Printing out the footballTeam after sort
     System.out.println("After sorting: ");
     for (Player player : footballTeam) {

      System.out.println(player.getRanking() + " " +
                               player.getName() + " " +
                               player.getAge());
     }

   }

}
```

Compile error:
The method sort(List<T>) in the type Collections is not applicable for the arguments (List<Player>)

15.1. **Comparable**

Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. Comparable affects the original class (the actual class is modified). Comparable provides `compareTo()` method to sort elements.

Using compareTo() method via Comparable interface:

When PlayerSorter.java was run, the footballTeam was sorted from lowest to highest ranking.

Player.java

```java
public class Player implements Comparable<Player>{

      .
      .
      .
      .

      @Override
      public int compareTo(Player otherPlayer) {
            return Integer.compare(getRanking(), otherPlayer.getRanking());
      }

}
```

Comparable interface implemented + @Override compareTo() method added to original Player.java

The sorting order is decided by the return value of the `compareTo()` method. The `Integer.compare(x, y)` returns -1 if x is less than y, returns 0 if they're equal, and returns 1 otherwise.

15.2. **Comparator**

The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. Comparator doesn't affect the original class (the actual class is not modified). Comparator provides `compare()` method to sort elements.

Using compare() method via Comparator interface:

PlayerRankingComparator.java

```java
import java.util.Comparator;

public class PlayerRankingComparator implements Comparator<Player>{

      @Override
      public int compare(Player firstPlayer, Player secondPlayer) {
            return Integer.compare(firstPlayer.getRanking(),
                        secondPlayer.getRanking());
      }

}
```

PlayerAgeComparator.java:

```
import java.util.Comparator;

public class PlayerRankingComparator implements Comparator<Player>{

      @Override
      public int compare(Player firstPlayer, Player secondPlayer) {
            return Integer.compare(firstPlayer.getRanking(),
                        secondPlayer.getRanking());
      }

}
```

PlayerSorter.java:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class PlayerSorter {

      public static void main(String[] args) {

            .
            .
            .
            .

            // Sort by Ranking (Ascending)
            PlayerRankingComparator playerComparator1 = new
            PlayerRankingComparator();
            Collections.sort(footballTeam, playerComparator1);

            // Sort the SORTED list by Age (Ascending)
            PlayerAgeComparator playerComparator2 = new
            PlayerAgeComparator();
            Collections.sort(footballTeam, playerComparator2);

            // Printing out the footballTeam after sort
            System.out.println("After sorting: ");
            for (Player player : footballTeam) {

            System.out.println(player.getRanking() + " " +
                                          player.getName() + " " +
                                          player.getAge());
          }

      }

}
```

16. Wrapper Classes

Wrapper classes **provide a way to use primitive data types** (`int`, `boolean`, etc.) **as objects**. The table below shows the primitive type and the equivalent wrapper class:

Wrappers

Primitive Data Type	Wrapper Class
`byte`	Byte
`short`	Short
`int`	Integer
`long`	Long
`float`	Float
`double`	Double
`Boolean`	Boolean
`char`	Character

Sometimes you must use wrapper classes, for example when working with Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

Creating ArrayList with Integer Wrapper:

```
public class Runner {
      public static void main(String[] args) {

            //ArrayList<int> myNums = new ArrayList<int>(); // Invalid
            ArrayList<Integer> myNums = new ArrayList<Integer>(); // Valid

      }
}
```

To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

Creating Wrapper Objects:

```
public class Runner {
      public static void main(String[] args) {
            Integer myInt = 5;
            Character myChar = 'A';
            System.out.println(myInt);
            System.out.println(myChar);
      }
}
```

NOTE:
Since you're now working with objects, you can use certain methods to get information about the specific object. `intValue()`, `byteValue()`, `booleanValue()` etc. are used to get the value associated with the corresponding wrapper object. Another useful method is the `toString()` method, which is used to convert wrapper objects to strings.

17. Files

File handling is an **important part of any application**. Java has several methods for creating, reading, updating, and deleting files.

17.1. File Handling

The `File` class from the `java.io` package, allows us to work with files. To use the File class, create an object of the class, and specify the filename or directory name:

Creating a File Object:

```
import java.io.File;

public class Runner {

	public static void main(String[] args) {

		File myObj = new File("filename.txt"); // Specify the filename

	}

}
```

The `File` class has many useful methods for creating and getting information about files. For example:

File Methods

Method	Type	Descriptions
`canRead()`	Boolean	Tests whether the file is readable or not.
`canWrite()`	Boolean	Tests whether the file is writable or not.
`createNewFile()`	Boolean	Creates an empty file.
`delete()`	Boolean	Deletes a file.
`exists()`	Boolean	Tests whether the file exists.
`getName()`	String	Returns the name of the file.
`getAbsolutePath()`	String	Returns the absolute pathname of the file.
`length()`	Long	Returns the size of the file in bytes.
`list()`	String[]	Returns an array of the files in the directory.
`mkdir()`	Boolean	Creates a directory.

17.2. Create and Write to Files

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an `IOException` if an error occurs (if the file cannot be created for some reason):

Creating a File:

```java
import java.io.File;
import java.io.IOException;

public class Runner {
      public static void main(String[] args) {
            try {

                  File myObj = new File("filename.txt");

                  if (myObj.createNewFile()) {
                        System.out.println("File created: " +
                        myObj.getName());
                  }

                  else {
                        System.out.println("File already exists.");
                  }

            } catch (IOException e) {
                  System.out.println("An error occurred.");
                  e.printStackTrace();
            }
      }
}
```

File myObj = new File("C:\\Users\\MyName\\filename.txt");

The output will be:
File created: filename.txt

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

Writing to a File:

```java
import java.io.FileWriter;
import java.io.IOException;

public class Runner {
      public static void main(String[] args) {
            try {
            FileWriter myWriter = new FileWriter("filename.txt");
            myWriter.write("Files in Java might be tricky, but it is fun
            enough!");
            myWriter.close();
            System.out.println("Successfully wrote to the file.");
            }

            catch (IOException e) {
                  System.out.println("An error occurred.");
                  e.printStackTrace();
            }
      }
}
```

The output will be this

17.3. **Read Files**

In the previous chapter, you learned how to create and write to a file. In the following example, we use the `Scanner` class to read the contents of the text file we created in the previous chapter:

Read a File:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Runner {

      public static void main(String[] args) {

          try {
            File myObj = new File("filename.txt");
              Scanner myReader = new Scanner(myObj);
              while (myReader.hasNextLine()) {
                  String data = myReader.nextLine();
                  System.out.println(data);
              }

              myReader.close();

          } catch (FileNotFoundException e) {
              System.out.println("An error occurred.");
              e.printStackTrace();
          }

      }

}
```

The output will be:
Files in Java might be tricky, but it is fun enough!

To get more information about a file, use any of the `File` methods:

Get File Information:

```
import java.io.File;

public class Runner {
      public static void main(String[] args) {
            File myObj = new File("filename.txt");
          if (myObj.exists()) {
            System.out.println("File name: " + myObj.getName());
            System.out.println("Absolute path: " +
            myObj.getAbsolutePath());
            System.out.println("Writeable: " + myObj.canWrite());
            System.out.println("Readable " + myObj.canRead());
            System.out.println("File size in bytes " + myObj.length());
          } else {
            System.out.println("The file does not exist.");
          }
      }
}
```

The output will be:
File name: filename.txt
Absolute path:
C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0

17.4. **Delete Files**

To delete a file in Java, use the `delete()` method:

Delete a File:

```java
import java.io.File;

public class Runner {

      public static void main(String[] args) {

            File myObj = new File("filename.txt");

          if (myObj.delete()) {
              System.out.println("Deleted the file: " + myObj.getName());
          }

          else {
              System.out.println("Failed to delete the file.");
          }

      }

}
```

You can also delete a folder. However, it must be empty:

Delete a Folder:

```java
import java.io.File;

public class Runner {

      public static void main(String[] args) {

          File myObj = new File("C:\\Users\\MyName\\Test");

          if (myObj.delete()) {
            System.out.println("Deleted the folder: " + myObj.getName());
          }

          else {
            System.out.println("Failed to delete the folder.");
          }

      }

}
```

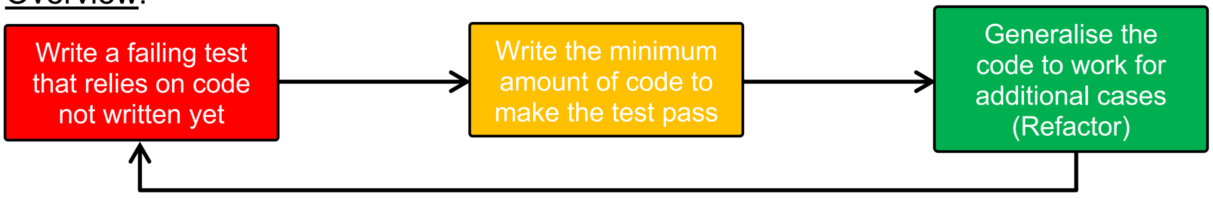
18. Test Driven Development

“Test-driven development” refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring). It is a technique where the **programmer writes a test before any production code, and then writes the code that will make that test pass.**

Unit Testing – software development process in which the **smallest testable parts** of an application, called units, are **individually and independently tested** to validate that each unit of the software code performs as expected. It is done during the development (coding phase) of an application by the developers.

Refactoring – disciplined technique for **restructuring an existing body of code, altering its internal structure without changing its external behaviour**. Practically, refactoring means making code clearer, cleaner, simpler and more robust.

Overview:


Write a failing test that relies on code not written yet
Write the minimum amount of code to make the test pass
Generalise the code to work for additional cases (Refactor)

The 5 Steps of TDD

1. Write the test
2. Make the test compile
3. Watch the test fail
4. Do just enough to get the test to pass
5. Refactor and generalise

Qualities of a Good Test

Focused - It should only test one thing.
Easy to read - The test name should make it clear what the test is doing.
Simple - Don’t overcomplicate your test with loops and decisions.
Independent - Individual tests should not affect each other in any way.
Flexible - A test and the code it is testing should be able to be re-used in different projects without having to change anything.

18.1. JUnit

JUnit is a unit testing framework for Java and plays a crucial role in TDD. JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented. It increases the productivity of the programmer and the stability of program code, which in turn reduces the stress on the programmer and the time spent on debugging.

The pattern for writing unit tests is:

- Arrange – Set up preconditions
- Act – Call the code under test
- Assert – Check that expected results have occurred

18.1.1. Enabling JUnit5

To enable JUnit5, you must update your compiler to version 1.8 or above and add the JUnit dependencies to your pom.xml file.

Setting Up JUnit:

1. Create Project:

(Project Explorer) → New → Project → Maven Project → Next → Create a simple project → Next → Group Id: com.fdmgroup → Artifact Id: MavenProjectName → Finish

2. Update Java and Add Dependencies:

MavenProjectName (drop down menu) → pom.xml → Copy & paste the red text in position shown:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.fdmgroup</groupId>
  <artifactId>BookStoreTDD</artifactId>
  <version>0.0.1-SNAPSHOT</version>

<properties>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
<dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.7.0</version>
      <scope>test</scope>
</dependency>

<dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-runner</artifactId>
      <version>1.7.0</version>
      <scope>test</scope>
</dependency>
</dependencies>

</project>
```

3. Update Project:

MavenProjectName → Maven → Update Project

18.1.2. Creating JUnit Test Files

To create a JUnit test you must create a test file.

Creating a JUnit Test File:

🖱 MavenProjectName (drop down) → 🖱 src/test/java → New → 🖱 Other → 🖱 JUnit Test Case → 🖱 Next → Package: ⌨ TestPackageName → Name: ⌨ TestFileName → 🖱 Finish

18.1.3. Assertions

Assertions are methods and **can verify whether a test should pass or fail by checking its return value**. Below are examples of some common assertions:

`assertEquals(expectedObject, actualObject)`

`assertNotEquals(expectedObject, actualObject)`

`assertTrue(boolean)`

`assertNull(object)`

`assertFalse(boolean)`

`assertNotNull(object)`

`assertArraryEquals(expectedArray,actualArray)`

18.1.4. Annotations

JUnit Annotations is a special form of syntactic meta-data that can be added to Java source code for *better code readability and structure*. Variables, parameters, packages, methods and classes can be annotated. Below are examples of some common annotations:

`@Test` – Indicates that a method is a test

`@BeforeEach` – The annotated method should be run before each test

`@BeforeAll` – The annotated method should be run once before any tests

`@AfterEach` – The annotated method should be run after each test

`@AfterAll` – The annotated method should be run once after all tests

18.1.5. Writing and Running JUnit Test Cases (Example)

To demonstrate how JUnit5 works an example problem will be shown.

Problem) A grade calculator service requires an implementing class that fulfils the interface contract. The implementing class must return the appropriate String value when a double representing a mark is passed in. They are as follows:

- "fail" when less than 75.
- "pass" when greater than or equal to 75 but less than 80.
- "merit" when greater than or equal to 80 but less than 90.
- "distinction" when greater than or equal to 90 but less than or equal to 100.

The interface contract is:

GradeCalculatorService.java

```
package com.fdmgroup.tdd.gradecalculator;

public interface GradeCalculatorService {
       public String getClassification(double mark);

}
```

Solution)
Step 1 – Enable JUnit5 (Section 18.1.1)
Step 2 – Create JUnit Test Case File (Section 18.1.2) [Select setUp() method stub]

GradeCalculatorServiceTest.java

```
package com.fdmgroup.tdd.gradecalculator;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class GradeCalculatorServiceTest {

       @BeforeEach
       void setUp() throws Exception {
       }

       @Test
       void test() {
              fail("Not yet implemented");
       }

}
```

Step 3 – Write an object of the 'ghost' class as an attribute. This will prompt you to create the actual class you have referenced.

GradeCalculatorServiceTest.java

```
class GradeCalculatorServiceTest{
       GradesClass gradesClass;
       @BeforeEach
       void setUp() throws Exception {
       }
       @Test
       void test() {
              fail("Not yet implemented");
       }
}
```

Step 4 – Ensure ‘Source folder’ of the Java Class being created is situated in the src/main/java directory.

GradesClass.java

```
public class GradesClass {

}
```

Step 5 – Create the object in the Test Case file in the setUp() method to finalise the setting up process (Arrange step).

GradeCalculatorServiceTest.java

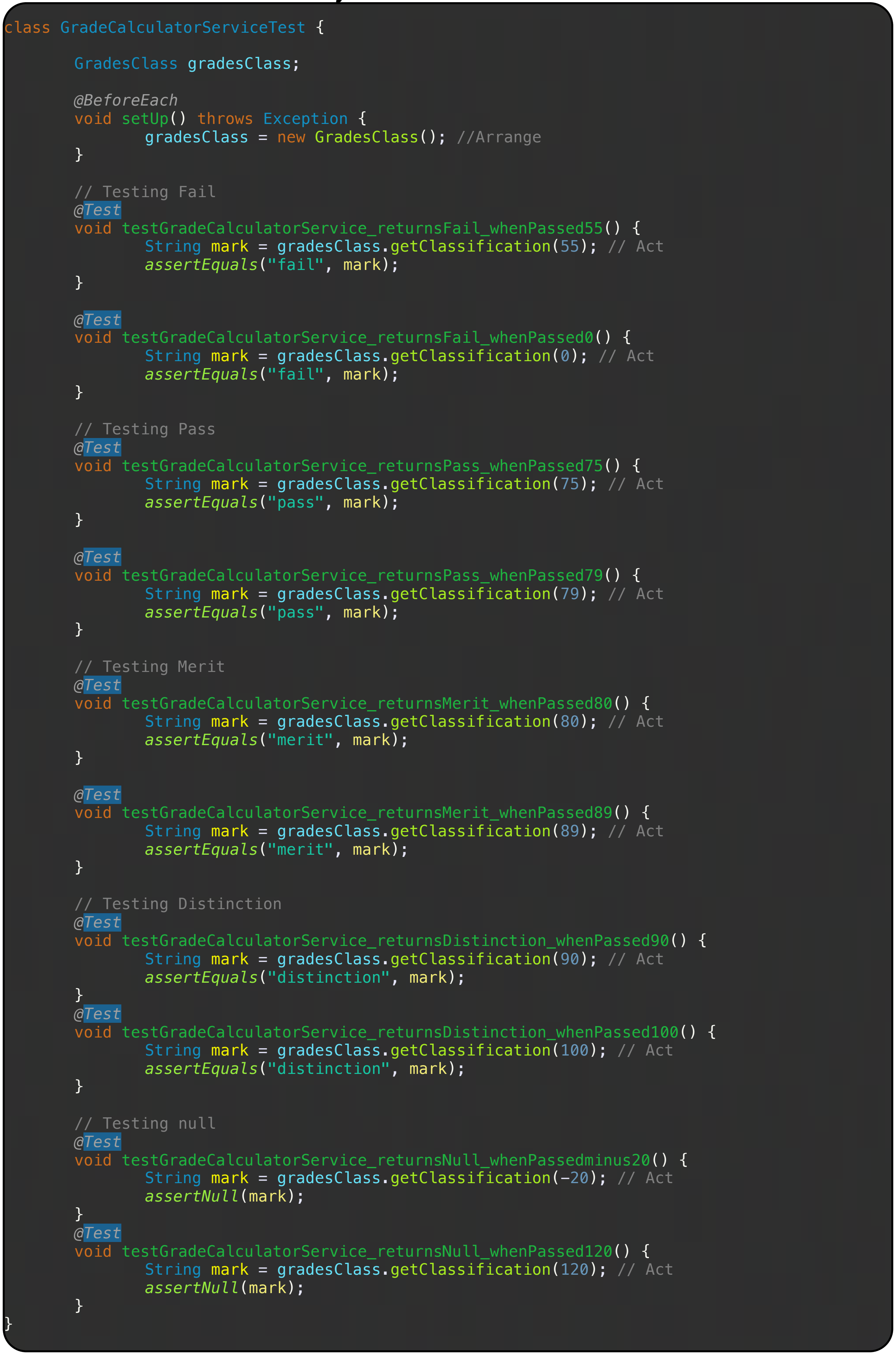
```
class GradeCalculatorTest {

	GradesClass gradesClass;

	@BeforeEach
	void setUp() throws Exception {
		gradesClass = new GradesClass(); //Arrange
	}

	@Test
	void test() {
		fail("Not yet implemented");
	}

}
```

Step 6 – Now you must write the GradesClass method according to the Problem and carry out unit tests to **ensure the program not only passes the tests but passes a range of tests consistently and with minimal coding**.

GradesClass.java

```
public class GradesClass implements GradeCalculatorService {

	@Override
	public String getClassification(double mark) {

		if (mark < 75 && mark >= 0) {
			return "fail";
		}

		else if (mark >= 75 && mark < 80) {
			return "pass";
		}

		else if (mark >= 80 && mark < 90) {
			return "merit";
		}

		else if (mark >= 90 && mark <= 100) {
			return "distinction";
		}

		return null;

	}

}
```

GradeCalculatorServiceTest.java



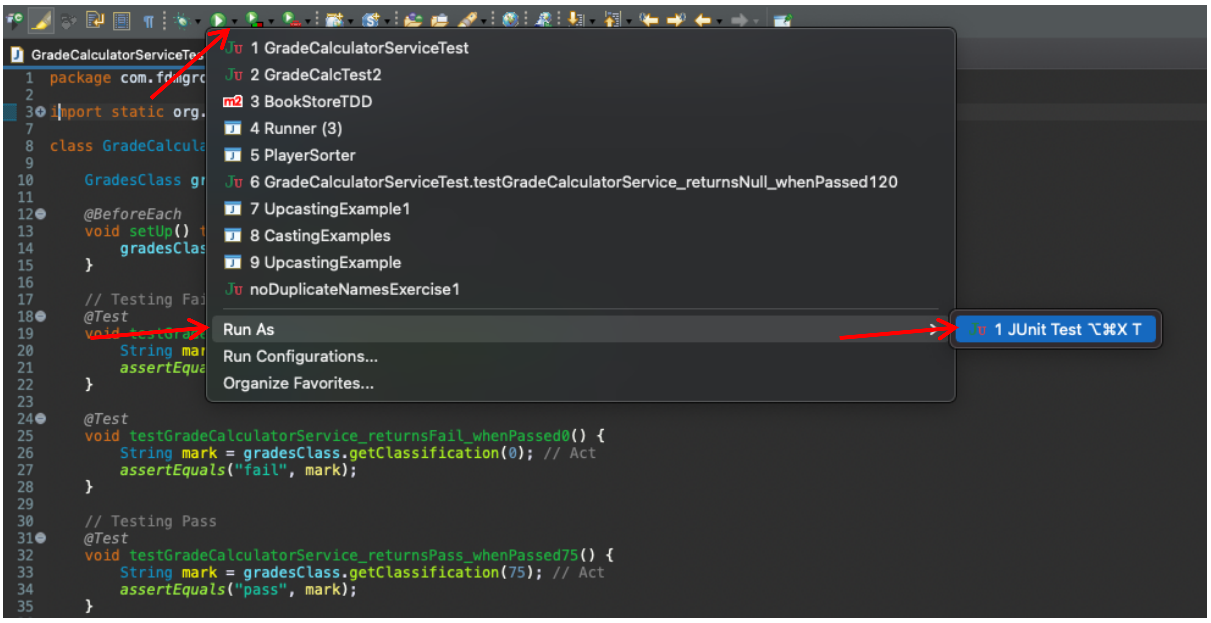
```java
class GradeCalculatorServiceTest {

        GradesClass gradesClass;

        @BeforeEach
        void setUp() throws Exception {
                gradesClass = new GradesClass(); //Arrange
        }

        // Testing Fail
        @Test
        void testGradeCalculatorService_returnsFail_whenPassed55() {
                String mark = gradesClass.getClassification(55); // Act
                assertEquals("fail", mark);
        }

        @Test
        void testGradeCalculatorService_returnsFail_whenPassed0() {
                String mark = gradesClass.getClassification(0); // Act
                assertEquals("fail", mark);
        }

        // Testing Pass
        @Test
        void testGradeCalculatorService_returnsPass_whenPassed75() {
                String mark = gradesClass.getClassification(75); // Act
                assertEquals("pass", mark);
        }

        @Test
        void testGradeCalculatorService_returnsPass_whenPassed79() {
                String mark = gradesClass.getClassification(79); // Act
                assertEquals("pass", mark);
        }

        // Testing Merit
        @Test
        void testGradeCalculatorService_returnsMerit_whenPassed80() {
                String mark = gradesClass.getClassification(80); // Act
                assertEquals("merit", mark);
        }

        @Test
        void testGradeCalculatorService_returnsMerit_whenPassed89() {
                String mark = gradesClass.getClassification(89); // Act
                assertEquals("merit", mark);
        }

        // Testing Distinction
        @Test
        void testGradeCalculatorService_returnsDistinction_whenPassed90() {
                String mark = gradesClass.getClassification(90); // Act
                assertEquals("distinction", mark);
        }
        @Test
        void testGradeCalculatorService_returnsDistinction_whenPassed100() {
                String mark = gradesClass.getClassification(100); // Act
                assertEquals("distinction", mark);
        }

        // Testing null
        @Test
        void testGradeCalculatorService_returnsNull_whenPassedminus20() {
                String mark = gradesClass.getClassification(-20); // Act
                assertNull(mark);
        }
        @Test
        void testGradeCalculatorService_returnsNull_whenPassed120() {
                String mark = gradesClass.getClassification(120); // Act
                assertNull(mark);
        }
}
```

18.1.6. Running Test Cases

1. <u>Run As</u>

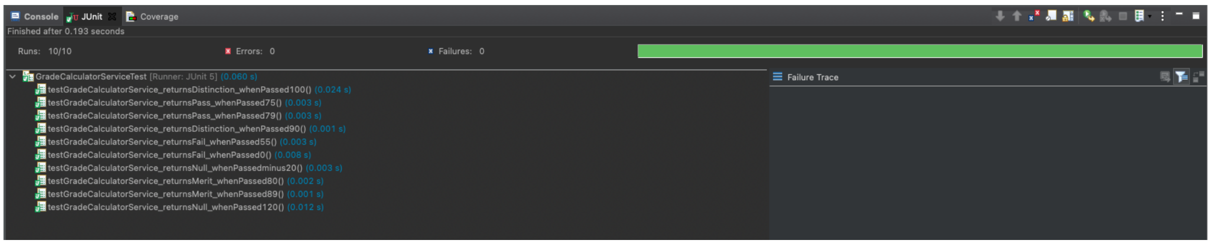
- Details the passed/failed tests and the expected/actual results from each test.

To execute a Run As JUnit Test:



```
GradeCalculatorServiceTe
package com.fdmgro
import static org.
class GradeCalcula
    GradesClass gr
    @BeforeEach
    void setUp() t
        gradesClas
    }
    // Testing Fai
    @Test
    void testGrade
        String mar
        assertEqua
    }
    @Test
    void testGradeCalculatorService_returnsFail_whenPassed0() {
        String mark = gradesClass.getClassification(0); // Act
        assertEquals("fail", mark);
    }
    // Testing Pass
    @Test
    void testGradeCalculatorService_returnsPass_whenPassed75() {
        String mark = gradesClass.getClassification(75); // Act
        assertEquals("pass", mark);
    }
```

The JUnit console:

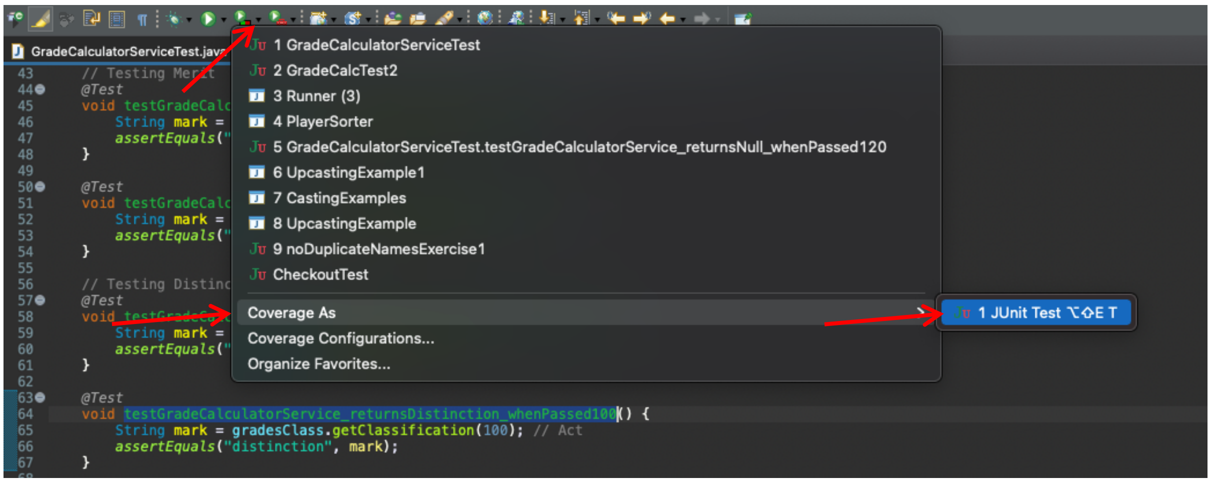


```
Finished after 0.193 seconds
Runs: 10/10          Errors: 0          Failures: 0

GradeCalculatorServiceTest [Runner: JUnit 5] (0.060 s)
    testGradeCalculatorService_returnsDistinction_whenPassed100() (0.024 s)
    testGradeCalculatorService_returnsPass_whenPassed75() (0.003 s)
    testGradeCalculatorService_returnsPass_whenPassed79() (0.003 s)
    testGradeCalculatorService_returnsDistinction_whenPassed90() (0.001 s)
    testGradeCalculatorService_returnsFail_whenPassed55() (0.003 s)
    testGradeCalculatorService_returnsFail_whenPassed0() (0.008 s)
    testGradeCalculatorService_returnsNull_whenPassedminus20() (0.003 s)
    testGradeCalculatorService_returnsMerit_whenPassed80() (0.002 s)
    testGradeCalculatorService_returnsMerit_whenPassed89() (0.001 s)
    testGradeCalculatorService_returnsNull_whenPassed120() (0.012 s)
```

2. Coverage As

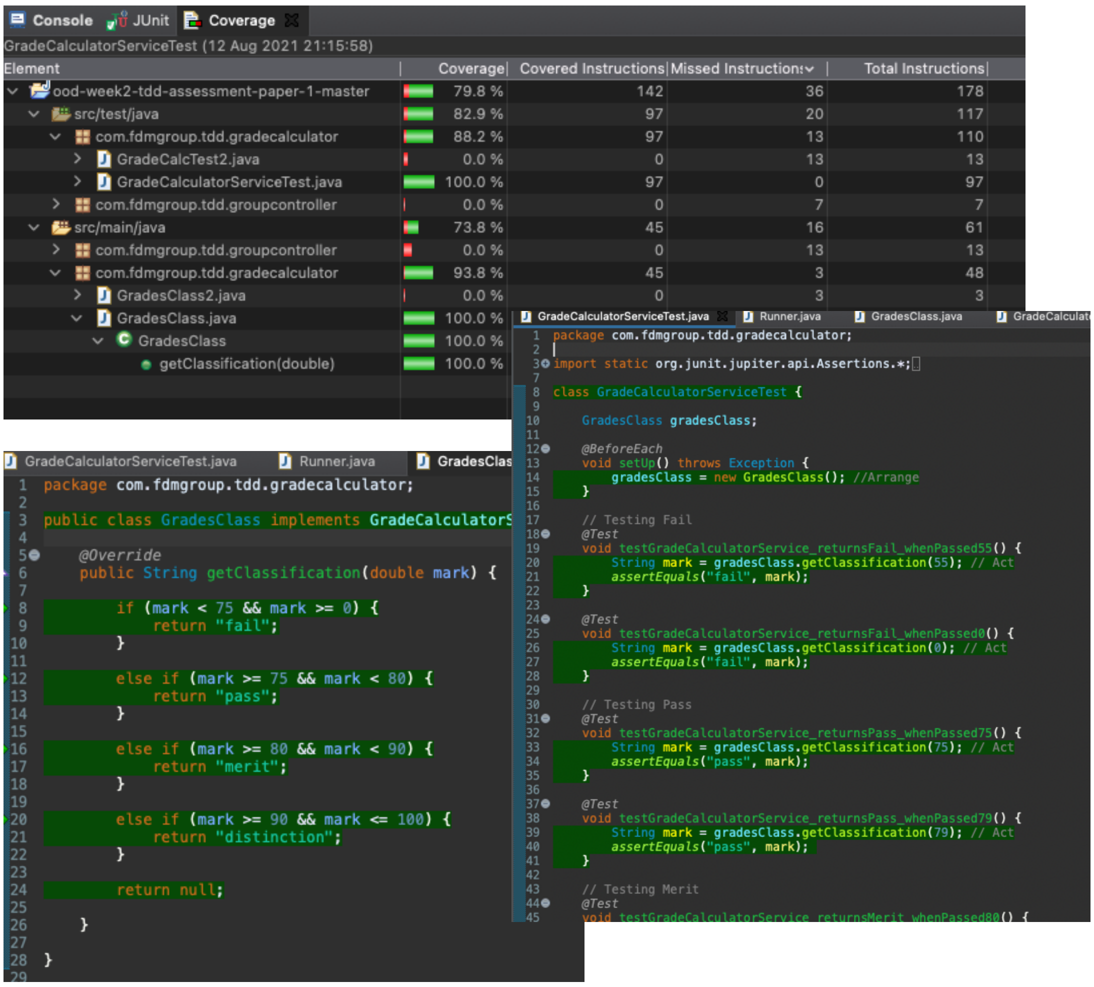
- Shows which lines of code have been executed when test cases are run.
- It can be used to identify and remove obsolete code.
- It shows the proportion of code in each class which is executed.

To execute a Coverage As JUnit test:


GradeCalculatorServiceTest.java
1 GradeCalculatorServiceTest
2 GradeCalcTest2
3 Runner (3)
4 PlayerSorter
5 GradeCalculatorServiceTest.testGradeCalculatorService_returnsNull_whenPassed120
6 UpcastingExample1
7 CastingExamples
8 UpcastingExample
9 noDuplicateNamesExercise1
CheckoutTest
Coverage As
Coverage Configurations...
Organize Favorites...
1 JUnit Test ⌥⇧E T

```
    // Testing Merit
    @Test
    void testGradeCalc
        String mark =
        assertEquals("
    }

    @Test
    void testGradeCalc
        String mark =
        assertEquals("
    }

    // Testing Distinc
    @Test
    void testGradeCalc
        String mark =
        assertEquals("
    }

    @Test
    void testGradeCalculatorService_returnsDistinction_whenPassed100() {
        String mark = gradesClass.getClassification(100); // Act
        assertEquals("distinction", mark);
    }
```

The JUnit Console and Java Programs:



GradeCalculatorServiceTest (12 Aug 2021 21:15:58)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ood-week2-tdd-assessment-paper-1-master	79.8 %	142	36	178
src/test/java	82.9 %	97	20	117
com.fdmgroup.tdd.gradecalculator	88.2 %	97	13	110
GradeCalcTest2.java	0.0 %	0	13	13
GradeCalculatorServiceTest.java	100.0 %	97	0	97
com.fdmgroup.tdd.groupcontroller	0.0 %	0	7	7
src/main/java	73.8 %	45	16	61
com.fdmgroup.tdd.groupcontroller	0.0 %	0	13	13
com.fdmgroup.tdd.gradecalculator	93.8 %	45	3	48
GradesClass2.java	0.0 %	0	3	3
GradesClass.java	100.0 %			
GradesClass	100.0 %			
getClassification(double)	100.0 %			

```java
package com.fdmgroup.tdd.gradecalculator;

public class GradesClass implements GradeCalculatorS

    @Override
    public String getClassification(double mark) {

        if (mark < 75 && mark >= 0) {
            return "fail";
        }

        else if (mark >= 75 && mark < 80) {
            return "pass";
        }

        else if (mark >= 80 && mark < 90) {
            return "merit";
        }

        else if (mark >= 90 && mark <= 100) {
            return "distinction";
        }

        return null;

    }

}
```

```java
package com.fdmgroup.tdd.gradecalculator;

import static org.junit.jupiter.api.Assertions.*;

class GradeCalculatorServiceTest {

    GradesClass gradesClass;

    @BeforeEach
    void setUp() throws Exception {
        gradesClass = new GradesClass(); //Arrange
    }

    // Testing Fail
    @Test
    void testGradeCalculatorService_returnsFail_whenPassed55() {
        String mark = gradesClass.getClassification(55); // Act
        assertEquals("fail", mark);
    }

    @Test
    void testGradeCalculatorService_returnsFail_whenPassed0() {
        String mark = gradesClass.getClassification(0); // Act
        assertEquals("fail", mark);
    }

    // Testing Pass
    @Test
    void testGradeCalculatorService_returnsPass_whenPassed75() {
        String mark = gradesClass.getClassification(75); // Act
        assertEquals("pass", mark);
    }

    @Test
    void testGradeCalculatorService_returnsPass_whenPassed79() {
        String mark = gradesClass.getClassification(79); // Act
        assertEquals("pass", mark);
    }

    // Testing Merit
    @Test
    void testGradeCalculatorService_returnsMerit_whenPassed80() {
```

18.2. **Mockito**

When testing in JUnit we test methods which return values. If we have *methods which have a return type of void* then we cannot use JUnit to test these methods. Instead, we can use Mockito.

Mocks let us test for interactions between components. Mockito uses mock objects. Mock objects **mimic the behaviour of real objects**. When we test using Mockito we test one class at a time. If a class depends on other classes we can mock those other classes. This means we can focus on one particular component of our application at a time.

When to Mock?
Using a mock object may be preferred to using an instance of the class in several scenarios:

- The class does not yet exist (the code has not been completed).
- If it uses a lot of resources to create an instance of the class.
- It may be slow to interact with an instance of the class (e.g. a database).
- The test requires a specific scenario that is hard to reproduce.

When Not to Mock?
Some developers will overuse mock objects. Keep the following in mind when mocking:

- Don't mock every class
- Don't mock entities
- Don't mock the class under test

18.2.1. Enabling Mockito

Follow the same general actions described in 18.2.1 but copy & paste this dependency to your pom.xml file:

Setting-up Mockito:

```
<dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>3.7.7</version>
      <scope>test</scope>
</dependency>

<dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-junit-jupiter</artifactId>
      <version>3.7.7</version>
      <scope>test</scope>
</dependency>
```

18.2.2. Creating a Mock

In Java, since we can substitute any parent class with a child class, this means we can replace the real object with the mock one as it fulfils the contracts set by its parent or interface. Mock objects can be injected into the class using dependency injection.

There are two ways to create a mock in Mockito:

1. Use @Mock

Insert the following annotation above the class definition:

```
@ExtendWith(MockitoExtension.class)
```

The annotation @Mock is placed before each mock member variable in the test class. Mock objects will be automatically created.

2. Use the mock method in the Mockito class

This is the traditional way to create mocks:

```
MyClass myClass = mock(MyClass.class);
```

18.2.3. Stubbing

We can add limited behaviour to the methods of our mock objects. We can also return specific values from the methods of our mock objects.

When creating a mock object, Mockito creates a sub class of that class, it then overrides the implementation of the methods and returns the default value for the return type.

To stub in Mockito use the code:

```
when(mockObject.method(3)).thenReturn(“Third”);
```

When the method is called on the mock object with an argument of 3, it will return the String "Third". If you call the method with a different value then null will be returned.

We can also stub our mock object so that it throws an exception:

```
when(mockObject.method(4)).thenThrow(new CustomException()));
```

When the method is called with an argument of 4, it will throw a CustomException.

18.2.4. Verification

Verification is the **foundation of testing in Mockito**. We can test methods which return void, checking they are being called the correct number of times. We can check that a method is called on a mock object, how many times it is called and what arguments are being passed to it.

We can also ensure a method is called:

- a specific number of times,
- at least once,
- at most once,
- we can also use the checks atLeast or atMost with ANY number.

<u>Using Verification</u>

Lets imagine we have the following program code:

```
mockObject.myMethod("one");
```

We can verify this is being called the correct number of times using the following code:

```
verify(mockObject, times(1)).myMethod("one");
```

Lets look at the separate parts:

- The verify method is in the Mockito framework
- The variable mockObject is the mock object you created
- times(1) allows you to check how many times myMethod has been called. Here we are checking it was called once.
- The method times(..) is in the Mockito framework.
- times(3) means the test expects the method to be called 3 times.

Lets imagine we have the following program code:

```
mockObject.myMethod("one");
mockObject.myMethod("two");
mockObject.myMethod("two");
```

We can use the following variations of the verify method:

```
verify(mockObject).myMethod("one");                    // PASS
verify(mockObject, times(1)).myMethod("one");          // PASS

OTHERS:
verify(mockObject, atMost(1)).myMethod("one");         // PASS
verify(mockObject, atLeast(2)).myMethod("two");        // PASS
verify(mockObject, never()).myMethod("never called"); // PASS
```

NOTE: instead of using one of JUnit's assert methods, we are instead using Mockito's verify method as our actual test condition.

18.2.5. Writing and Running Mockito Test Cases (Example)

Basket.java

```java
import java.util.List;

public interface Basket {

	List<Book> getBooksInBasket();

	void addBook(Book book);

	void removebook(Book book);
}
```

Book.java

```java
public class Book {

	private double price;

	public double getPrice() {
		return price;
	}

	public void setPrice(double price) {
		this.price = price;
	}

}
```

Checkout.java

```java
import java.util.List;

public class Checkout {

	public double calculatePrice(Basket basket) {
		List<Book> books = basket.getBooksInBasket();

		double total = 0;

		for (Book book : books) {
			total += book.getPrice();
		}

		return total;
	}

	public void emptyBasket(Basket basket) {
		List<Book> books = basket.getBooksInBasket();

		for (Book book : books) {
			basket.removebook(book);
		}

	}

}
```

CheckoutTest.java

```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

@ExtendWith(MockitoExtension.class) // creates mock objects for everything marked @Mock
class TestCheckout {

        // The class we're testing is the only real object in the whole testcase
        Checkout checkout;

        // All other objects are mock objects:
        @Mock
        Basket mockBasket;

        @Mock
        Book mockBook1, mockBook2, mockBook3;

        @BeforeEach
        void setUp() throws Exception {
                checkout = new Checkout();
        }

        @Test
        void test_calculatePrice_callsMockBaskets_getBooksInBasketMethod() {
               checkout.calculatePrice(mockBasket); // We don't care about the return value!
               verify(mockBasket).getBooksInBasket(); // passes if getBooksInBasket was called
        }

        @Test
        void test_calculatePrice_returns20point75WhenBooksCosting9point5and11point25Passed() {
                when(mockBook1.getPrice()).thenReturn(9.5);
                when(mockBook2.getPrice()).thenReturn(11.25);
                List<Book> mockBooks = new ArrayList<>(Arrays.asList(mockBook1,mockBook2));
                when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);

                double total = checkout.calculatePrice(mockBasket); // Act
                assertEquals(20.75,total); // Assert
        }

        @Test
        void test_calculatePrice_returns25WhenBooksCosting9pnt5_11pnt25_pnt25PassedIn() {
                when(mockBook1.getPrice()).thenReturn(9.5);
                when(mockBook2.getPrice()).thenReturn(11.25);
                when(mockBook3.getPrice()).thenReturn(4.25);
                List<Book> mockBooks = new
                ArrayList<>(Arrays.asList(mockBook1,mockBook2,mockBook3));
                when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);

                double total = checkout.calculatePrice(mockBasket); // Act
                assertEquals(25.0,total); // Assert
        }

        @Test
        void test_emptyBasket_callsRemoveBook2Times_whenBasketWith2BooksPassedIn() {
                List<Book> mockBooks = new ArrayList<>(Arrays.asList(mockBook1,mockBook2));
                when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);
                checkout.emptyBasket(mockBasket);
                verify(mockBasket,times(2)).removebook(any(Book.class)); // passes if remove
                book is called twice
        }

        @Test
        void test_emptyBasket_callsRemoveBook3Times_whenBasketWith3BooksPassedIn() {
                List<Book> mockBooks = new
                ArrayList<>(Arrays.asList(mockBook1,mockBook2,mockBook3));
                when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);
                checkout.emptyBasket(mockBasket);
                verify(mockBasket,times(3)).removebook(any(Book.class)); // passes if remove
                book is called three times
        }

}
```

19. Expressions

19.1. Regular Expression

A regular expression is a **sequence of characters that forms a search pattern**. When you search for data in a text, you can use this search pattern to describe what you are searching for. A regular expression can be a single character, or a more complicated pattern. Regular expressions can be used to perform all types of text search and text replace operations.

Java does not have a built-in Regular Expression class, but we can import the `java.util.regex` package to work with regular expressions. The package includes the following classes:

- `Pattern` Class - Defines a pattern (to be used in a search)
- `Matcher` Class - Used to search for the pattern
- `PatternSyntaxException` Class - Indicates syntax error in a regular expression pattern

Example:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Runner {

      public static void main(String[] args) {

          Pattern pattern = Pattern.compile("Java",
                                  Pattern.CASE_INSENSITIVE);
          Matcher matcher = pattern.matcher("I am writing this in Java!");

          boolean matchFound = matcher.find();

          if (matchFound) {
            System.out.println("Match found");
          }

          else {
            System.out.println("Match not found");
          }

      }

}
```

The word "Java" is being searched for. Pattern is created using the Pattern.compile() method. First arg is the pattern being searched for. Second arg (optional) has a flag to state that the search should be case-sensitive

The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

The find() method returns true if the pattern was found in the string and false if it was not found.

Flags in the compile() method change how the search is performed. Here are a few of them:

- `Pattern.CASE_INSENSITIVE` - The case of letters will be ignored.
- `Pattern.LITERAL` - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters.
- `Pattern.UNICODE_CASE` - Use it together with the `CASE_INSENSITIVE` flag to also ignore the case of letters outside of the English alphabet.

The first parameter of the `Pattern.compile()` method is the pattern. It describes what is being searched for. Brackets are used to find a range of characters:

Brackets

Expression	Description
`[abc]`	Find one character from the options between the brackets.
`[^abc]`	Find one character NOT between the brackets.
`[0-9]`	Find one character from the range 0 to 9.

Metacharacters are characters with a special meaning:

Metacharacters

Primitive Data Type	Wrapper Class
`|`	Find a match for any one of the patterns separated by | as in: cat|dog|fish
`.`	Find just one instance of any character
`^`	Finds a match as the beginning of a string as in: ^Hello
`$`	Finds a match at the end of the string as in: World$
`\d`	Find a digit
`\s`	Find a whitespace character
`\b`	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
`\uxxxx`	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers define quantities:

Metacharacters

Primitive Data Type	Wrapper Class
`n+`	Matches any string that contains at least one n
`n*`	Matches any string that contains zero or more occurrences of n
`n?`	Matches any string that contains zero or one occurrences of n
`n{x}`	Matches any string that contains a sequence of X n's
`n{x,y}`	Matches any string that contains a sequence of X to Y n's
`n{x,}`	Matches any string that contains a sequence of at least X n's

NOTE:
If your expression needs to search for one of the special characters you can use a backslash (\ ) to escape them. In Java, backslashes in strings need to be escaped themselves, so two backslashes are needed to escape special characters. For example, to search for one or more question marks you can use the following expression: "\\?"

19.2. **Lambda Expression**

Lambda Expressions were added in Java 8. A lambda expression is a **short block of code which takes in parameters and returns a value**. Lambda expressions are similar to methods, but they *do not need a name and they can be implemented right in the body of a method*.

The simplest lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as `if` or `for`. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a `return` statement.

```
(parameter1, parameter2) -> { code block }
```

Lambda expressions are usually passed as parameters to a function:

Using a lambda expression in the ArrayList's forEach() method to print every item in the list:

```
import java.util.ArrayList;

public class Runner {

      public static void main(String[] args) {

          ArrayList<Integer> numbers = new ArrayList<Integer>();
          numbers.add(5);
          numbers.add(9);
          numbers.add(8);
          numbers.add(1);
          numbers.forEach( (n) -> { System.out.println(n); } );

      }

}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the `Consumer` interface (found in the `java.util` package) used by lists.

Using Java's Consumer interface to store a lambda expression in a variable:

```java
import java.util.ArrayList;
import java.util.function.Consumer;

public class Runner {

      public static void main(String[] args) {

          ArrayList<Integer> numbers = new ArrayList<Integer>();
          numbers.add(5);
          numbers.add(9);
          numbers.add(8);
          numbers.add(1);

          Consumer<Integer> method = (n) -> { System.out.println(n); };

          numbers.forEach(method);

      }

}
```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

Creating a method which takes a lambda expression as a parameter:

```java
interface StringFunction {
      String run(String str);
}

public class Runner {

      public static void main(String[] args) {

          StringFunction exclaim = (s) -> s + "!";
          StringFunction ask = (s) -> s + "?";
          printFormatted("Hello", exclaim);
          printFormatted("Hello", ask);

      }

      private static void printFormatted(String str, StringFunction
      format) {

            String result = format.run(str);
            System.out.println(result);

      }

}
```

20. Threads

Threads allows a program to **operate more efficiently** by **doing multiple things at the same time**. Threads can be used to perform complicated tasks in the background without interrupting the main program.

20.1. Creating a Thread

There are two ways to create a thread. It can be created by extending the `Thread` class and overriding its `run()` method:

Extend Syntax:

```
public class Main extends Thread {
      public void run() {
            System.out.println("This code is running in a thread");
      }
}
```

Another way to create a thread is to implement the `Runnable` interface:

Implement Syntax:

```
public class Main implements Runnable {
      public void run() {
            System.out.println("This code is running in a thread");
      }
}
```

20.2. Running Threads

If the class extends the `Thread` class, the thread can be run by creating an instance of the class and call its `start()` method:

Extend Syntax:

```
public class Runner extends Thread {

      public static void main(String[] args) {

            Runner thread = new Runner();
            thread.start();
            System.out.println("This code is outside of the thread");

      }

      public void run() {

            System.out.println("This code is running in a thread");

      }

}
```

If the class implements the `Runnable` interface, the thread can be run by passing an instance of the class to a `Thread` object's constructor and then calling the thread's `start()` method:

Implement Syntax:

```
public class Runner implements Runnable {

      public static void main(String[] args) {

            Runner obj = new Runner();
            Thread thread = new Thread(obj);
            thread.start();
            System.out.println("This code is outside of the thread");

      }

      public void run() {

            System.out.println("This code is running in a thread");

      }

}
```

Differences between "extending" and "implementing" Threads
The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class `MyClass` `extends` `OtherClass` `implements` `Runnable`.

20.3. Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

A code example where the value of the variable amount is unpredictable:

```
public class Runner extends Thread {
      public static int amount = 0;

      public static void main(String[] args) {
            Runner thread = new Runner();
            thread.start();
            System.out.println(amount);
            amount++;
            System.out.println(amount);
      }

      public void run() {
            amount++;
      }
}
```

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the `isAlive()` method of the thread to **check whether the thread has finished running before using any attributes that the thread can change**.

Use isAlive() to prevent concurrency problems:

```java
public class Runner extends Thread {

      public static int amount = 0;

      public static void main(String[] args) {

            Runner thread = new Runner();
          thread.start();

          // Wait for the thread to finish
          while(thread.isAlive()) {

            System.out.println("Waiting...");

          }

          // Update amount and print its value
          System.out.println("Main: " + amount);
          amount++;
          System.out.println("Main: " + amount);

      }

      public void run() {

            amount++;

      }

}
```