

- If you try to access any method of an object, but the object is null; then you'll get an **Null Pointer Exception**.
- **Remember: in java we initialize an array using `{ }` curly braces; not `[]` square brackets.**
- Consider the following scenario:
 - ⌘ I have 3 files:
 - ⌘ **Main.java**
 - * Contains the main method.
 - * It has an object of type **QuestionService**
 - ⌘ **QuestionService.java**
 - * It has an array of objects.
 - * Those objects are of type **Question**
 - ⌘ **Question.java**
 - * It is a normal class that contains some variables and getter, setter methods to build a question.
 - * Like, **id (int), question(String), options (String[]), answer (String)**
 - ⌘ Now, I implemented the **toString()** method inside **Question** class, so that whenever I print the object of type **Question** directly, it'll print something meaningful instead of the default *hashCode*.
 - ⌘ Then I compiled **Main.java** and ran this. But I couldn't see the changes that I did inside **Question.java**.
 - ⌘ When you compile a **.java** file, it'll generate **.class** file of all those Class which are linked to the **.java** file (in our case, all Classes i.e. **Main, QuestionService, Question** are linked).
 - ⌘ Let I run **javac Main.java**
 - ⌘ Now, it'll compile **Main.java** and create the **Main.class** file.
 - ⌘ And it'll check if there is **QuestionService.class** and **Question.class** already present.
 - ⌘ If present, then don't recompile those; otherwise compile those as well.
 - ⌘ In my case, I had already **QuestionService.class** and **Question.class** present; so it was not re-compiling those classes.
 - ⌘ So, every-time you do any changes, run the following command:
 - ⌘ **javac *.java** (it'll re-compile all the **.java** files present in the current directory)
 - ⌘ **java Main**

➤ To read input from users:

- ♣ Scanner is used to read the input from the user.

```
Scanner sc = new Scanner(System.in); // initializing scanner to read user input
int i = 0;
for (Question q : questions) {
    System.out.println("Question No: " + q.getId());
    System.out.println(q.getQuestion());
    String[] options = q.getOptions();
    for (String option : options)
        System.out.println(" - " + option);

    this.selections[i++] = sc.nextLine(); // reading user input and storing in array
}
sc.close(); // closing scanner to prevent resource leak
```

- ♣ Here, I created an object of Scanner, and passed **System.in**

- ♣ **System.in** is a static input stream provided by JVM.
- ♣ It represents standard input of your program.
- ♣ Since it is **static final**, there is only **System.in** object per JVM process.
- ♣ Once you close the scanner object using **sc.close()**, you can't read the input again.

- ♣ Below is an example of wrong usage of Scanner:

```
int i = 0;
for (Question q : questions) {
    System.out.println("Question No: " + q.getId());
    System.out.println(q.getQuestion());
    String[] options = q.getOptions();
    for (String option : options)
        System.out.println(" - " + option);

    Scanner sc = new Scanner(System.in); // create a Scanner object
    this.selections[i++] = sc.nextLine(); // read user input
    sc.close(); // close the scanner to prevent resource leak
    // but closing here will close System.in, causing issues on next iteration
}
```

- ♣ Here you'll get an exception after the first iteration, because **System.in** is already closed in the previous iteration.

- ♣ **sc.close()** is optional by the way.

➤ Abstract Class and Abstract Method

- ♣ In java, empty methods are valid.

```
class Car {
    // this method is valid but doesn't give error
    public void drive() {

    }

    public void playMusic() {
        System.out.println(x:"Playing music");
    }
}
```

- ⌘ These are some conditions in Java OOP:
 - ⌘ Abstract method inside Abstract class (✓)
 - ⌘ Abstract method inside Normal class (✗)
 - ⌘ Normal method inside Abstract class (✓)
 - ⌘ Normal method inside Normal class (✓ (default only))
- ⌘ In short:
 - ⌘ Abstract method ⇒ class must be abstract
 - ⌘ Normal methods ⇒ allowed anywhere
- ⌘ An abstract class may have:
 - ⌘ Only abstract methods
 - ⌘ Only normal methods
 - ⌘ A mix of abstract + normal methods
 - ⌘ Even no methods at all

```

abstract class Car {
    public abstract void drive();

    public void playMusic() {
        System.out.println(x:"Playing music");
    }
}

class Tesla extends Car {
    public void drive() {
        System.out.println(x:"Driving Tesla");
    }

    public void show() {
        System.out.println(x:"Show method in Tesla");
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Car car = new Tesla();
        car.drive();
        car.playMusic();
        // car.show(); // This will give an error because 'show' is not defined in Car

        Tesla myCar = new Tesla();
        myCar.drive();
        myCar.playMusic();
        myCar.show();
    }
}

```

- ⌘ If a class is inheriting an abstract class
 - ⌘ It must implements the abstract methods present inside the abstract class.
 - ⌘ The normal methods present inside the abstract class need not to be overridden.

♣ An abstract class can have constructor.

♣ The constructor can be called from the base classes using **super()**

♣ NOTE

- ♣ An abstract class can inherit another abstract class as well.
- ♣ And in this case, the child *abstract class* need not to implement the *abstract methods* inside the parent abstract class.

```
abstract class Car {  
    public abstract void drive();  
  
    public abstract void accelerate();  
  
    public void playMusic() {  
        System.out.println(x:"Playing music");  
    }  
}  
  
abstract class FastCar extends Car {  
  
    public void accelerate() {  
        System.out.println(x:"Accelerating FastCar quickly");  
    }  
}  
  
class Tesla extends FastCar {  
    public void drive() {  
        System.out.println(x:"Driving Tesla");  
    }  
  
    public void show() {  
        System.out.println(x:"This is a Tesla car");  
    }  
}
```

➤ Inner Class

- ♣ An inner class is a class defined inside another class.
- ♣ It is logically associated with its outer class and has access to its members (even private ones).

♣ The inner class's type will be: **OuterClassName.InnerClassName**

♣ And to instantiate the inner class, you need an instance of the outer class.

♣ To instantiate the inner class, you need to call like.

♣ **obj.new InnerClassName()**

♣ There are 4 types of Inner Class

- Non-Static Nested Inner Class
- Static Nested Inner Class
- Local Inner Class
- Anonymous Inner Class

• Non-Static Nested Inner Class:

```
class Outer {
    int age = 5;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show");
    }

    class Inner {
        // shadowing occurred
        int age = 30;

        public void config() {
            System.out.println("in Inner's config: inner age = " + age); // inner's age
            System.out.println("in Inner's config: outer age = " + Outer.this.age); // outer's age
            // both are correct
            System.out.println("in Inner's config: name = " + name);
            System.out.println("in Inner's config: name = " + Outer.name);
        }
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer out = new Outer();
        Outer.Inner ob = out.new Inner();

        // Outer.Inner ob = new Outer().new Inner();
        // it is also correct; outer object would be anonymous

        ob.config();

        System.out.println(ob.age);
    }
}
```

- Just imagine a non-static method. You can access this only by an object.
- Just like that, you can access the Non-Static Inner Class using an object of Outer Class only.
- It can access all the instance and static variables of the outer class (even private variables are accessible).
- In the above example, the instance variable **age** got shadowed inside the Inner class. To access the Outer class's age

* **OuterClassName.this.VariableName**

* Because, **this.age** would have given InnerClass's variable **age**

- Why not `new ob1.Inner()` ?
 - Think of it like, as the Inner class is non-static; so the Inner class's instance will be specific to the Outer class's instance.
 - So, to instantiate Inner class's instance inside the Outer class's instance (here `ob1`), we need to call `ob1.new Inner()`

Static Nested Inner Class:

- Declared with the static keyword.
- It does not need an instance of the outer class.
- Can access only **static members** of the outer class directly.

```
class Outer {
    int age;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show");
    }

    static class Inner {
        public void config() {
            // // error; as age is an instance variable
            // System.out.println("in Inner's config: age = " + age);
            // both are correct
            System.out.println("in Inner's config: name = " + name);
            System.out.println("in Inner's config: name = " + Outer.name);
        }
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer.Inner ob = new Outer.Inner();
        // new Outer().Inner() <= wrong

        ob.config();
    }
}
```

- Just like static method, we can access the static inner class using the Outer class directly without instantiating it.
- Here, `new Outer.Inner()` (not `Outer.new Inner()` or `new Outer().Inner()`)

Local Inner Class:

- When the Inner class is defined inside a method of Outer class, then it is Local Inner Class.

```
class Outer {
    int age = 5;
    static String name = "Outer Static";

    private void show() {
        System.out.println(x:"In outer's show..");
    }

    public void show2() {
        System.out.println(x:"in Outer's show2..");

        class Inner {
            int val = 10;

            private void displayVal() {
                System.out.println("displayVal: val inside Inner is: " + val);
            }

            public void displayVal2() {
                System.out.println("displayVal2: val inside Inner is: " + val);
            }
        }
        // it can access both private and public methods of inner class
        Inner obj = new Inner();
        obj.displayVal();
        obj.displayVal2();
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        // ob.show(); // error as method is private
        ob.show2();
    }
}
```

- It is strange that, the **displayVal** method is private; but still it was able to get called from outside of it i.e. inside the **show()** method.
- As the Inner class comes inside the scope of Outer class, so in this case, **all private things of Outer class and Inner class are accessible to each-other.**
- But the private method **show()** of the class **Outer** is not accessible outside.
- Because Inner lives inside the scope of Outer, they can freely access each other's private members.
- But Main is outside, so it cannot access **Outer.show()** or **Inner.displayVal()**.

Anonymous Inner Class:

```
class Test {
    public void greet() {
        System.out.println(x:"Hello from Test!");
    }
}

class Outer {
    int age = 5;
    static String name = "Outer Static";

    public void show() {
        System.out.println(x:"in Outer's show...");

        // case-1
        class AdvTest extends Test {
            public void greet() {
                System.out.println(x:"Hello from AdvTest!");
            }
        }

        Test ob = new AdvTest();
        ob.greet();

        // case-2 (anonymous inner class)
        // --- for this case: Test can be abstract as well ---
        Test ob2 = new Test() {
            public void greet() {
                System.out.println(x:"Hello from Anonymous Test!");
            }
        }; // semicolon is necessary here
        ob2.greet();
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        ob.show();
    }
}
```



```

class Test {
    public void greet() {
        System.out.println(x:"Hello from Test!");
    }
}

class Outer {
    int age = 5;
    static String name = "Outer Static";
    public Test obj = new Test() {
        public void greet() {
            System.out.println(x:"Hello from Outer => Test!");
        }
    };
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Outer ob = new Outer();
        ob.obj.greet();
    }
}

```

⚡

- ⚡ Its just like inheriting a Normal/Abstract class and instantiating directly without creating the inherited class.

Summary of Inner Classes

- In any type of inner class creation, both Outer and Inner classes can access each-other's private members.

• Non-Static Inner Class:

- * Assumption: Inner class's name: **Inner**, Outer class's name: **Outer**
- * Just like non-static method, the Non-Static Inner Class can access both instance variables and static variables of the Outer class.
- * If there is any type of shadowing of Outer class's variable then (let variable name is: **val**)
 - " **this.val** ⇒ Inner class's variable *val*
 - " **Outer.this.val** ⇒ Outer class's variable *val*
- * Just like Non-Static Method, we need an instance of the class to access the Non-Static Inner Class.
- * As the Inner class's instance will be a part of the Outer class's instance, so to instantiate this:
 - " **obOuter.new Inner()**

• Static Inner Class:

- * Assumption: Inner class's name: **Inner**, Outer class's name: **Outer**
- * Just like the Static Methods, the Static Inner Class can only access the *static* members of the Outer class.
- * If there is any shadowing: (let the variable name is **val**)
 - " **val** ⇒ Inner class's static variable
 - " **Outer.val** ⇒ Outer class's static variable

• Local Inner Class:

- * The Inner class is defined inside a method of the Outer Class.
- * The scope to access this Inner class is only the scope of that Method.

• Anonymous Inner Class:

- * Its just like extending a class (either Normal or Abstract) and creating an object out of that; without creating the Class.
- * The syntax is:
 - " **ClassName obj = new ClassName() { /* override method if want */ }**

•

➤ Interface

- By default the variables inside interfaces are: **public static final**.

- So, you need to initialize while declaring it.

```
interface A {  
    // by default variables are "public final static"  
    // so, you need to initialize this  
    int age = 23;  
    String area = "Bangalore";  
}
```

- You cannot override the variables that were declared and initialized in interface.

```
public static void main() {  
    System.out.println("A's area: " + A.area);  
    System.out.println("A's age: " + A.age);  
    //A.area = "Delhi"; // Error, because that is final  
}
```

- By default all the methods are **public abstract**; you don't need to explicitly write that.

```
interface A {  
    void show(); //same as => public abstract void show()  
    void config(); //same as => public abstract void config()  
}
```

- implements** is the keyword that is used to implement a interface to a class.
- Unlike classes, **multiple implementations** are allowed in case of interface.

```
class B implements A, X {  
    public void show() {  
        // ...  
    }  
}
```

(multiple implementation)

- Interfaces can inherit another interface.
- In this case, **multiple inheritance** is allowed.

```
interface Y extends X, A {  
    // ...  
}
```

- But the class which implements Y, has to override all the methods mentioned in interfaces X and A.

NOTE

- Interfaces cannot have constructors (because they can't be instantiated).
- But you can create a reference of an interface type pointing to a class object.
- Interfaces are used to achieve abstraction and multiple inheritance in Java.

```

interface A {
    void show();
    void config();
}
interface X {
    void run();
}
class B implements A, X {
    public void show() {
        System.out.println(x:"overridden 'A: show'");
    }
    public void config() {
        System.out.println(x:"overridden 'A: config'");
    }
    public void run() {
        System.out.println(x:"overridden 'X: run'");
    }
}
public class Interface {
    public static void main() {
        A obj = new B();
        obj.show();
        obj.config();
        // obj.run(); // Error: A doesn't have run

        X obj2 = new B();
        // obj.show(); // Error: X doesn't have show
        // obj.config(); // Error: X doesn't have config
        obj2.run();
    }
}

```

- In case of implementing 2 interfaces, creating object of one interface type and calling the method mentioned in the other interface will not be possible.
- We had seen this during Upcasting and Downcasting.

➤ Need of Interface

- ⌘ You can see the below example code.
- ⌘ Here if we didn't have implemented an interface, only **Laptop** or **Desktop** type of objects would have been acceptable inside the **codeApplication** method of **Developer** class.
- ⌘ Now we can think, instead of interface, **abstract class** can also be used;
- ⌘ But, just to write a abstract method, why to create an abstract class.
- ⌘ Interface is here simple and doing all the required things.

```
interface Computer {
    void code();
}

class Laptop implements Computer {
    public void code() {
        System.out.println(x:"Coding started: little slow");
    }
}

class Desktop implements Computer {
    public void code() {
        System.out.println(x:"Coding started: faster");
    }
}

class Developer {
    public void codeApplication(Computer comp) {
        comp.code();
    }
}

class Company {
    Run | Debug
    public static void main(String[] args) {
        Developer alok = new Developer();
        Developer kanha = new Developer();
        // we are giving alok a laptop
        // and kanha a desktop to code
        Computer laptop = new Laptop();
        Computer desktop = new Desktop();

        alok.codeApplication(laptop);
        kanha.codeApplication(desktop);
    }
}
```

➤ Enum

- enum is a special type of class in Java (its not same as Class; but similar).
- It's a **final class** which cannot be inherited by any other class.

```
enum Status {
    Running, Failed, Pending, Success;
}

public class Enum {
    Run | Debug
    public static void main(String[] args) {
        Status s = Status.Failed;
        System.out.println("s = " + s); // Failed

        // in Java, the indexing starts from 0 for the enums
        // Running: 0, Failed: 1, Pending: 2, Success: 3
        // the method "ordinal()" returns the index
        Status[] allVals = Status.values();
        for (Status val : allVals)
            System.out.println("index: " + val.ordinal() + ", value: " + val);
    }
}
```

- switch case** statement also supports **enum**, so it can be used to check the status.
- Consider the following example: (more than one constructor can be created)

```
enum Laptop {
    Macbook(price:2000), Dell(price:1200), Acer(price:1400);

    private int price;

    Laptop(int price) {
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}
```

```
final class Laptop extends Enum<Laptop> {
    public static final Laptop Macbook = new Laptop("Macbook", 0, 2000);
    public static final Laptop Dell = new Laptop("Dell", 1, 1200);
    public static final Laptop Acer = new Laptop("Acer", 2, 1400);

    private int price;

    private Laptop(String name, int ordinal, int price) {
        super(name, ordinal); // from java.lang.Enum
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}
```

* Behind the scene.


```
enum Laptop {
    // these are objects of Laptop class itself
    // as you are passing some value, so you need to create a constructor
    Macbook(price:2000), Dell(price:1200), Acer(price:1400);

    public int price;

    Laptop(int price) {
        this.price = price;
    }

    public int getPrice() {
        return this.price;
    }
}

public class Enum {
    Run | Debug
    public static void main(String[] args) {
        Laptop lap = Laptop.Macbook;
        System.out.println(lap.getClass()); // class Laptop
        System.out.println(lap.getClass().getSuperclass()); // class java.lang.Enum

        System.out.println("s = " + lap); // Macbook
        System.out.println("price = " + lap.getPrice()); // 2000
    }
}
```

➤ Annotations

- ⌘ Provides information to the compiler, tools, or runtime.
- ⌘ Think of it as a special marker/label you attach to classes, methods, variables, etc.
- ⌘ For example **@Override**
 - ⌘ It tells the compiler: “this method is supposed to override a method from its superclass.”
 - ⌘ If it doesn't, the compiler will show an error.

```
class A {  
    public void greet() {  
        System.out.println(x:"Hello from class A");  
    }  
}  
  
class B extends A {  
    public void greeet() {  
        System.out.println(x:"Hello from class B");  
    }  
}
```

- ⌘ Here you can see, I have made a *spelling error* in Class B.
- ⌘ Instead of **greet** I have written **greeet**

```
class A {  
    public void greet() {  
        System.out.println(x:"Hello from class A");  
    }  
}  
  
class B extends  
    @Override  
    public void greeet() {  
        System.out.println(x:"Hello from class B");  
    }  
}
```

The method greeet() of type B must override or implement the abstract method
void B.greeet()
View Problem (Alt+F8) Quick Fix... (Ctrl+.) Fix (Ctrl+I)

- ⌘ Now I used the annotation **@Override**, so now the compiler is showing me the error that this method doesn't exist in the superclass.

➤ Types of Interface

⌘ **Normal Interface**

- ⌘ Interface having **2 or more** methods

⌘ **Functional Interface / SAM (Single Abstract Method)**

- ⌘ Interface having only **1** method.

⌘ **Marker Interface**

- ⌘ Interface having **no** method.
- ⌘ used for tagging or marking classes (e.g., Serializable).

⌘ **Functional Interface:**

```
@FunctionalInterface
interface A {
    void show();

    private static int add(int a, int b) {
        return a + b;
    }

    default void display() {
        System.out.println(add(a:4, b:5));
    }
}
```

- * Abstract method should be only **1**.
- * Remaining static or default methods can be there.
- * Annotation: **@FunctionalInterface**

```
Interface.java Invalid '@FunctionalInterface' annotation; A is not a functional interface
A
@FunctionalInterface View Problem (Alt+F8) Quick Fix... (Ctrl+.) Fix (Ctrl+I)
interface A {
    void show();
    void config();

    private static int add(int a, int b) {
        return a + b;
    }

    default void display() {
        System.out.println(add(a:4, b:5));
    }
}
```

- * I added one more Abstract method, so it is showing me error.

➤ **Lambda Expression**

```
@FunctionalInterface
interface A {
    void show();
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // anonymous inner class concept
        A obj = new A() {
            public void show() {
                System.out.println(x:"in show A");
            }
        };
        obj.show();
    }
}
```

☞ This code is proper and it'll work fine.

```
public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // anonymous inner class concept
        A obj = new A() {
            public void show() {
                System.out.println(x:"in show A");
            }
        };
        obj.show();

        // lambda expression
        A obj2 = () -> {
            System.out.println(x:"in show A");
        };
        obj2.show();

        // if there is only single expression
        A obj3 = () -> System.out.println(x:"in show A");
        obj3.show();
    }
}
```

```
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj3 = (int a) -> System.out.println("in show A: " + a);
        obj3.show(a:4);
    }
}
```

☞ You can also pass the arguments.

```
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = (a) -> System.out.println("in show A: " + a);
        obj.show(a:4);
    }
}
```

- You don't even need to provide the data type; it'll take from the interface directly.

```
@FunctionalInterface
interface A {
    void show(int a);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = a -> System.out.println("in show A: " + a);
        obj.show(a:4);
    }
}
```

- If you have only one argument, don't need to give the *parenthesis* as well.

```
@FunctionalInterface
interface A {
    int add(int a, int b);
}

public class FuncInterface {
    Run | Debug
    public static void main(String[] args) {
        // if there is only single expression
        A obj = (a, b) -> a + b;
        int res = obj.add(a:4, b:5);
        System.out.println("Sum = " + res);
    }
}
```

- You can directly return the values like this.

• **Lambda Expression only works with the Functional Interface.**

• **Because if there are more than one method, which will be implemented.**

➤ Exceptions

- ⌘ Compile time error and Logical Errors can be fixed;
- ⌘ But Run Time error should be handled. So that the application won't stop in between.
- ⌘ Exception Handling is nothing but handling these Run Time error.

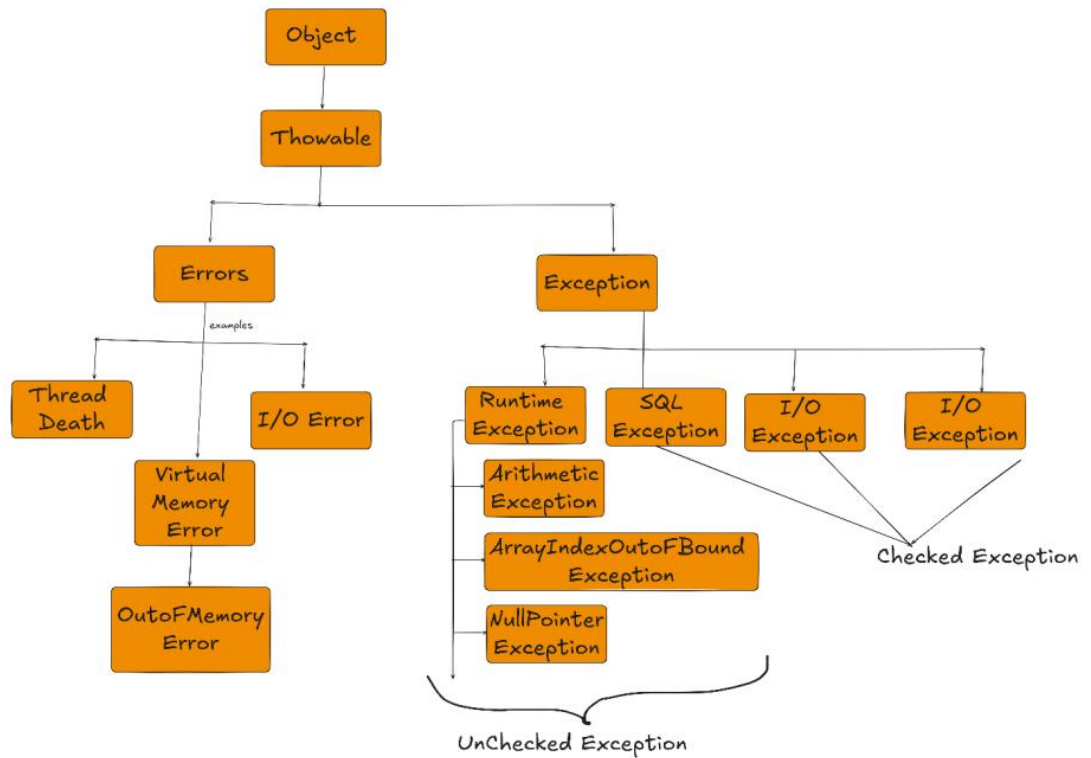
```
public class Exceptionss {  
    Run | Debug  
    public static void main(String[] args) {  
        int i = 0, j = 0;  
  
        try {  
            j = 20 / i; // as i is 0, it'll throw an error  
        } catch (Exception e) {  
            System.out.println("Something went wrong: " + e);  
        }  
        System.err.println(x:"Bye");  
    }  
}
```

- ⌘ Handling Exceptions using try catch block.

```
(main)  
$ java Exceptionss  
Something went wrong: java.lang.ArithmeticException: / by zero  
Bye
```

```
public class Exceptionss {  
    Run | Debug  
    public static void main(String[] args) {  
        int i, j;  
        i = 5;  
        j = 1;  
  
        int[] nums = new int[5];  
  
        try {  
            j = 20 / i;  
            System.err.println(nums[1]);  
            System.err.println(nums[5]); // error bcs out of bound  
        } catch (ArithmeticException e) {  
            System.out.println("divided by zero\n" + e);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.err.println("Out of bound\n" + e);  
        } catch (Exception e) { // if some other exception occurred  
            System.err.println("Something went wrong\n" + e);  
        }  
        System.err.println(x:"Bye");  
    }  
}
```

- ⌘ Using multiple catch blocks to catch different types of Exceptions.



2

- This is the hierarchy of Exception classes.
- **Checked** means the exceptions that are checked during compile-time. i.e. `IOException`, `ClassNotFoundException`, `SQLException`
- **Unchecked** means the exceptions that are occur during the run-time i.e. `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException` ..etc etc

➤ **throw** Keyword

```
public class Exceptionss {  
    Run | Debug  
    public static void main(String[] args) {  
        int i = 5, j = 0;  
  
        try {  
            if (j == 0)  
                throw new ArithmeticException(s:"J cannot be zero!");  
            j = 20 / i;  
        } catch (ArithmeticException e) {  
            j = 20 / 1;  
            System.err.println("Set j's value as default: " + e);  
        } catch (Exception e) {  
            System.err.println("Something went wrong\n" + e);  
        }  
        System.err.println("j = " + j);  
        System.err.println(x:"Bye");  
    }  
}
```

- We can throw any kind of exception we want by giving some customized error message.

```
class MyException extends Exception {  
    public MyException(String string) {  
        super(string);  
    }  
}  
  
public class Exceptionss {  
    Run | Debug  
    public static void main(String[] args) {  
        int i = 5, j = 0;  
  
        try {  
            if (j == 0)  
                throw new MyException(string:"J cannot be zero!");  
            j = 20 / i;  
        } catch (MyException e) {  
            j = 20 / 1;  
            System.err.println("Set j's value as default: " + e);  
        } catch (Exception e) {  
            System.err.println("Something went wrong\n" + e);  
        }  
        System.err.println("j = " + j);  
        System.err.println(x:"Bye");  
    }  
}
```

- It is a **custom exception**.

- You need to inherit the **Exception** class or can also inherit **RuntimeException** class; and pass the string to the super class's constructor because those Exception classes handles this message.

```
$ java Exceptionss
Set j's value as default: MyException: J cannot be zero!
j = 20
Bye
```

➤ **throws** keyword

- Let suppose, in a method **A** , you are calling 2 methods **B** and **C**.
- **B** and **C** both are having a critical expression that might throw the same Exception.
- So, instead of handling those inside **B** and **C** both, we can handle those inside **A** directly.
- For this, **throws** keyword will be used in **B** and **C**.
- It is used to forward the Exception to the method where the current method is called.

- Lets **A** is being called in some another method **X**, if you mention the keyword **throws** in the method **A** as well, then the Exception occurred from **B** and **C** will go to **A** then it'll go to **X**.

```
class A {
    public void show() {
        try {
            Class.forName(className:"NoClass");
            System.err.println(x:"Class found!");
        } catch (ClassNotFoundException e) {
            System.err.println("Not able to find the class; " + e);
        }
    }
}

public class Exceptionss {
    Run | Debug
    public static void main(String[] args) {
        A obj = new A();
        obj.show();
    }
}
```

- Simple code only; exception will be thrown inside the **show()** method and will be handled there only.

```

class A {
    public void show() throws ClassNotFoundException {
        Class.forName(className:"NoClass");
    }
}

public class Exceptionss {
    Run | Debug
    public static void main(String[] args) {
        A obj = new A();

        try {
            obj.show();
            System.err.println(x:"Class found!");
        } catch (ClassNotFoundException e) {
            System.err.println("Not able to find the class; " + e);
        }
    }
}

```

• So, like this you can use **throws** to forward the Exception to calling method.

```

class A {
    public void showA() throws ClassNotFoundException {
        Class.forName(className:"NoClass");
    }
}

class B {
    public void showB() throws ClassNotFoundException {
        A obj = new A();
        obj.showA();
    }
}

class C {
    public void showC() throws ClassNotFoundException {
        B obj = new B();
        obj.showB();
    }
}

public class Exceptionss {
    Run | Debug
    public static void main(String[] args) {
        C obj = new C();

        try {
            obj.showC();
            System.err.println(x:"Class found!");
        } catch (ClassNotFoundException e) {
            System.err.println("Not able to find the class; " + e);
        }
    }
}

```

• Here the exception flows from A to Exceptionss class:

* A's showA ⇒ B's showB ⇒ C's showC ⇒ Exceptionss's main

➤ User Input

- ⌘ When we write `System.out.println("...")` it prints something in the CLI.
- ⌘ Here, **System** is a class where there is a *static variable* which is **out**.

```
J System.class X
102 public final class System {
160     /*
161     public static final PrintStream out = null;
162
163     /**
```

- ⌘ And, this **out** variable is of type **PrintStream**.
 - ⌘ Inside the class **PrintStream**, there is a method which is **println**,
 - ⌘ This is how, **System.out.println** works.
- ⌘ Just like that **out** variable, another variable is there inside the **System** class which is **in**

```
System.class X
102 public final class System {
128     /*
129     public static final InputStream in = null;
130
131     /**
```

- ⌘ It is of type **InputStream**
- ⌘ Inside the class **InputStream**, so many methods are there like *read*, *readAllBytes* etc etc.

```
public class UserInput {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Hello");

        int val = System.in.read();
    }
}
Unhandled exception type IOException Java(16777384)
int java.io.InputStream.read() throws IOException
```

- ⌘ As we can see, it is saying that **read** method might throw **IOException** (it is a checked exception; so it'll give error during compilation)
- ⌘ Just to handle this temporarily, I am appending the **throws** keyword in the main method (It is not at all preferable; because if the **main** method throws the exception, it'll go to JVM directly and the application will stop).

```
import java.io.IOException;

public class UserInput {

    Run | Debug
    public static void main(String[] args) throws IOException {
        System.out.println(x:"Hello");

        int val = System.in.read();
        System.out.println("val = " + val);
    }
}
```

Now the compilation error gone.

```
alokr@Alok MINGW64
(main)
$ java UserInput
Hello
abcd
val = 97
```

* It'll just return the ASCII value of the first character (here 'a')

```
public static void main(String[] args) throws IOException {
    BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
    System.out.print(s:"Enter a string: ");
    String line = bf.readLine();
    System.out.println("line = " + line + "\n");

    System.out.print(s:"Enter a number: ");
    int val = Integer.parseInt(bf.readLine());
    System.out.println("val = " + val + "\n");

    BufferedReader bf2 = new BufferedReader(new FileReader(fileName:"./test.txt"));
    System.out.println(x:"Reading a file....");
    String fileLine = bf2.readLine();
    while (fileLine != null) {
        System.out.println("fileLine = " + fileLine);
        fileLine = bf2.readLine();
    }

    bf.close();
    bf2.close();
}
```

```
$ java UserInput
Enter a string: Alok Ranjan
line = Alok Ranjan

Enter a number: 974545
val = 974545

Reading a file....
fileLine = it is alok
fileLine = it is a normal text file.
```

(Output)

- ☛ It is how we can take input from user using **BufferedReader**.
- ☛ **BufferedReader** constructor takes an **Reader** type argument.
 - * First case, I took **InputStreamReader**, it'll be used to take user's input from terminal.
 - * Second case, I took **FileReader**, it'll be used to read a file.
- ☛ Here **bf** and **bf2** (**BufferedReader** instance) are resources. So whenever you create these, you have to close it as well.
 - * It'll not give any error, but it is a good idea to close the resources.

```
public static void main(String[] args) throws IOException {
    Scanner sc = new Scanner(System.in);

    System.out.print(s:"Enter a number: ");
    int num = sc.nextInt();
    System.out.println("num = " + num + "\n");

    sc.nextLine();

    System.out.print(s:"Enter a string: ");
    String str = sc.nextLine();
    System.out.println("str = " + str + "\n");

    sc.close();
}
```

```
$ java UserInput
Enter a number: 8547
num = 8547

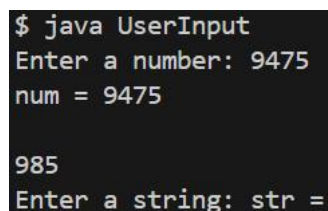
Enter a string: abcdef
str = abcdef
```

(Output)

- ☛ Here you must be thinking why we have written **sc.nextLine()** in between.
 - * When you give input and hit *Enter*, the next **sc.nextLine()** will take that as its input.
 - * So, you can't take the input for the string here because it'll take that **\n** (*Enter*) as its input.

```
$ java UserInput
Enter a number: 9475
num = 9475

985
Enter a string: str =
```

- *  (it would have occurred without that middle **sc.nextLine()**)

➤ Try with Resources

- ⌘ There is a keyword **finally**; this block executes even if the Exception occurred (catch) or not (try).
- ⌘ Even you can just run **try** and **finally** without **catch**.
- ⌘ The finally block is mostly used to *close the resources*.
- ⌘ Without this **finally** block, we would have to close the resources on both **try** and **catch** blocks.

```
public static void main(String[] args) {  
    Scanner sc = null;  
    try {  
        sc = new Scanner(System.in);  
        System.out.print(s:"Enter a number: ");  
        int num = sc.nextInt();  
        System.out.println("num = " + num + "\n");  
    } finally {  
        System.out.println(x:"Closing the resource...");  
        sc.close();  
    }  
}
```

- ⌘ This is how we can use the finally block to close the resources.

```
public static void main(String[] args) {  
  
    try (Scanner sc = new Scanner(System.in)) {  
        System.out.print(s:"Enter a number: ");  
        int num = sc.nextInt();  
        System.out.println("num = " + num + "\n");  
    }  
}
```

- ⌘ It is a short syntax.
- ⌘ Here, after the try block is completed, the resource will be closed automatically.

```
public final class Scanner implements Iterator<String>, Closeable {
```

```
public interface Closeable extends AutoCloseable {
```

- ⌘ You can see, the Scanner class's ancestor is the AutoClosable interface, so it'll be automatically closed.

➤ Threads

- There is a class **Thread** in java, which has a method called **start()**.
 - This **start()** method call a method whose name is **run()**.
 - So, if you want to run a method in a thread, then you need to give **run** as the method name.
 - Below is the example of threads:

```
class A extends Thread {
    public void run() {
        for (int i = 1; i <= 20; i++)
            System.err.println(x:"--A--A--A--");
    }
}

class B extends Thread {
    public void run() {
        for (int i = 1; i <= 20; i++)
            System.err.println(x:"B--B--B--B");
    }
}

public class ThreadPractice {
    Run | Debug
    public static void main(String[] args) {
        A ob1 = new A();
        B ob2 = new B();

        ob1.start();
        ob2.start();
    }
}
```

\$ java ThreadPractice
B--B--B--B
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
--A--A--A--
B--B--B--B
B--B--B--B
B--B--B--B

- Output is not continuous like --A--, --B--, --A--, --B-- like this
- If your CPU has **n** cores, then **n** threads can be run at a same time.
 - In modern systems, 1 core may be able to run 2 or more threads at a same time.

```
A ob1 = new A();
B ob2 = new B();

System.err.println(ob1.getPriority()); // 5
System.err.println(ob2.getPriority()); // 5
```

- The priority range is from **0** to **10**.
 - 0** is least priority and **10** is highest priority.
 - To set the priority, we can use **setPriority** method.

```

class A extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.err.println(x:"Hi");
            try {
                Thread.sleep(millis:10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class B extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.err.println(x:"Hello");
            try {
                Thread.sleep(millis:10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

public class ThreadPractice {
    Run | Debug
    public static void main(String[] a) {
        A ob1 = new A();
        B ob2 = new B();
        ob1.setPriority(Thread.MAX_PRIORITY);

        ob1.start();
        ob2.start();
    }
}

```

- * (System.out, not System.err (minor mistake :))
- * Here, I gave some sleep to make the output alternate Hi, Hello, Hi, Hello.. like this.

```

Hello
Hi
Hi
Hello
Hi
Hello

```

- * (you can just optimize it; how it'll work can't control)
- * In above case, might be both the run came to the scheduler to get executed after their respective sleep of 10 milliseconds (mentioned in code), then scheduler might have given someone.

➤ Runnable vs Thread

- ⌘ It is not a good idea to inherit the **Thread** class to make a thread.
- ⌘ Because, if the class has to inherit some other class, then it can't be done in this case.

```
public class Thread implements Runnable {  
    /* Make sure registerNatives is the first
```

- ⌘ The **Thread** class implements an *functional* interface **Runnable**, and the **run** method is present inside the **Runnable** interface only.

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

```
class A implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 10; i++)  
            System.out.println(x:"Hi");  
    }  
}  
  
class B implements Runnable {  
    public void run() {  
        for (int i = 1; i <= 10; i++)  
            System.out.println(x:"Hello");  
    }  
}  
  
public class RunnablePractice {  
    Run | Debug  
    public static void main(String[] a) {  
        /**  
         * Runnable bcs Thread constructor accepts Runnable  
         * We can even create with A, B in stead of Runnable,  
         * anyhow, only the "run()" will be called,  
         * so why to give a heavy object...  
         */  
        Runnable ob1 = new A();  
        Runnable ob2 = new B();  
        // A ob1 = new A(); // this is also correct  
        // B ob2 = new B(); // this is also correct  
  
        Thread t1 = new Thread(ob1);  
        Thread t2 = new Thread(ob2);  
  
        t1.start();  
        t2.start();  
    }  
}
```

- ⌘ It is how **Runnable** and **Thread** work.


```

public class RunnablePractice {
    Run | Debug
    public static void main(String[] a) {
        Runnable ob1 = () -> {
            for (int i = 1; i <= 10; i++)
                System.out.println(x:"Hi");
        };
        Runnable ob2 = () -> {
            for (int i = 1; i <= 10; i++)
                System.out.println(x:"Hello");
        };

        Thread t1 = new Thread(ob1);
        Thread t2 = new Thread(ob2);

        t1.start();
        t2.start();
    }
}

```

(using lambda

expression)

➤ Some subtle links between Thread and Runnable

```

public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    this(group, target, name, stackSize, acc:null, inheritThreadLocals:true);
}

```

✎ This is the main constructor of the Thread class.

✎ **target is of Runnable type.**

✎ When we extend the Thread class by any custom class, by default the default constructor of the Thread class (non-parameterized constructor) gets called.

```

public Thread() {
    this(group:null, target:null, "Thread-" + nextThreadNum(), stackSize:0);
}

```

✎ It is the default constructor of Thread class.

✎ Here we can see, the target is null.

✎ **So, when we extends Thread class from our class, the target is null.**

✎ Also there is a **run()** method inside the Thread class which overrides the **run()** method of the interface **Runnable**.

```

@Override
public void run() {
    if (target != null) {
        target.run();
    }
}

```


- In case of extending Thread class, we override this **run()** method, so that our **run()** method (present in our class) will get executed.
- One more constructor inside Thread is there which accepts **target** .

```
public Thread(Runnable target) {  
    this(group:null, target, "Thread-" + nextThreadNum(), stackSize:0);  
}
```

- So, if we are not extending the class, we need to pass a **Runnable** type object inside the Thread constructor while initializing.
- Our custom class can implement the **Runnable** interface and that object can be passed inside the **Thread** class's constructor.

➤ **Race Condition**

- ⌘ When 2 threads are running, they should not modify one variable at the same time.
- ⌘ Like imagine transacting to 2 different persons from the same bank account at the same time, it'll cause issues.
- ⌘ When you start the threads inside a method, the method doesn't stop there and execute the remaining code after starting the thread.
 - ⌘ If you want to execute the statements after the threads are complete, then use **join()** method.

```
class Counter {
    int count;
    Counter() {
        this.count = 0;
    }
    public void increment() {
        this.count++;
    }
}

public class RacePractice {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Runnable ob1 = () -> {
            for (int i = 1; i <= 5000; i++)
                c.increment();
        };
        Runnable ob2 = () -> {
            for (int i = 1; i <= 5000; i++)
                c.increment();
        };

        Thread t1 = new Thread(ob1);
        Thread t2 = new Thread(ob2);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.err.println(c.count);
    }
}
```

- ⌘ This code should give the output *10000*, but the output will not be consistent.

```
$ for((i=1;i<=5;i++)); do java RacePractice; done
10000
8631
10000
10000
8705
```

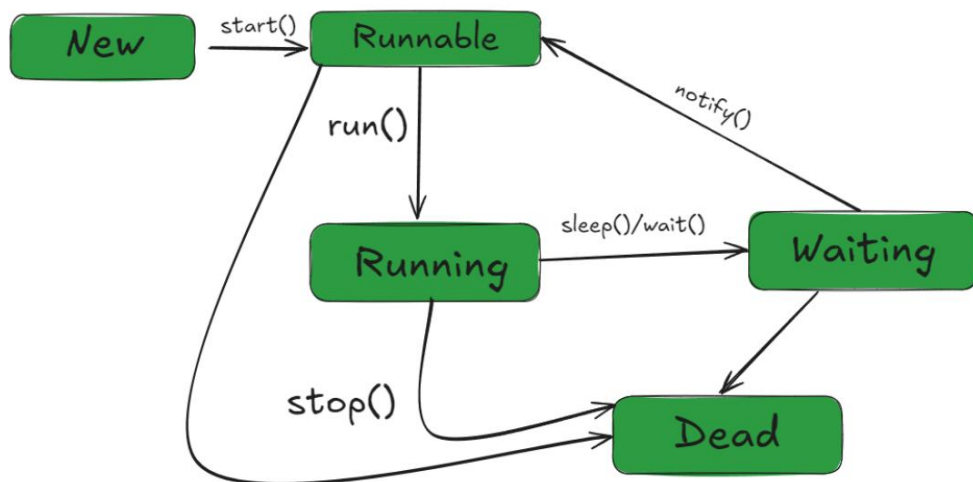
- I ran for 5 times, the results are inconsistent.
- It is happening because, at sometimes, both the threads are executing the **increment()** method at same time;
 - Lets value of count was 100 at a time, both executed *increment()* method at that time.
 - So, now, instead of 102, the value of count became 101.
 - This is the cause of the inconsistent result.
- There is a keyword called **synchronized**, it doesn't allow the method to be called 2 times at once.

```
public synchronized void increment() {
    this.count++;
}
```

```
dv_java (main)
$ for((i=1;i<=5;i++)); do java RacePractice; done
10000
10000
10000
10000
10000
```

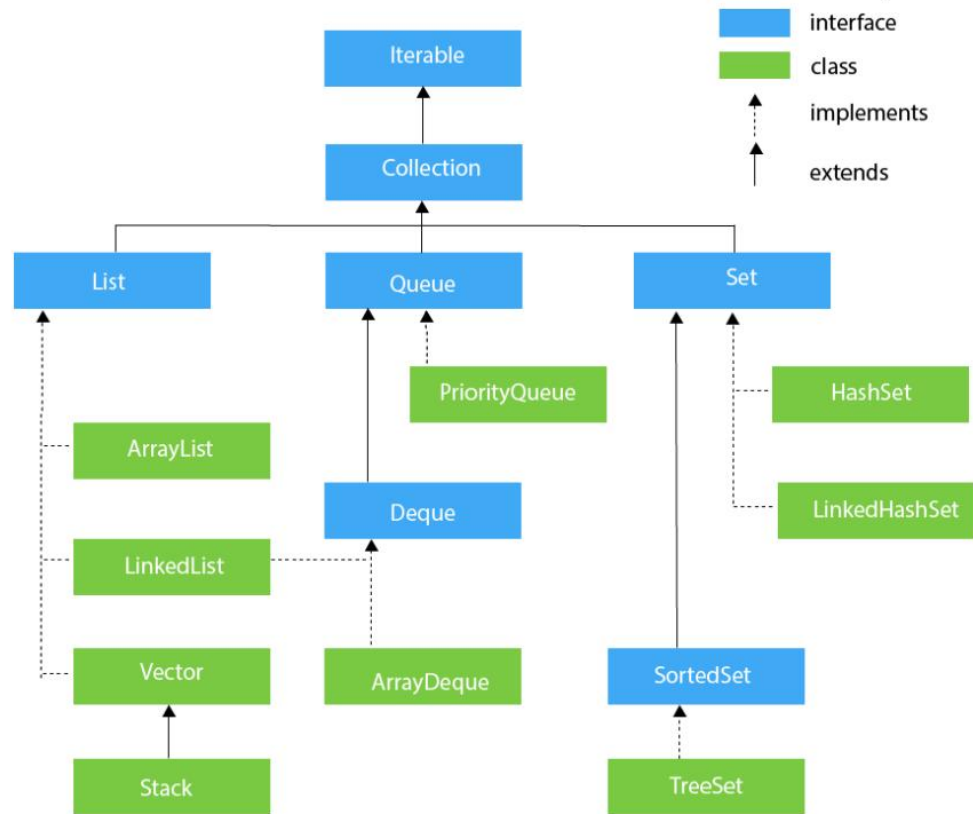
- Now, the result is consistent.

➤ Thread States



➤ **Collection API**

- ♣ Collection API : concept
- ♣ Collection : interface
- ♣ Collections : class



➤ ArrayList

```
public static void main(String[] args) {  
    Collection nums = new ArrayList();  
    for (int i = 1; i <= 10; i++)  
        nums.add(i);  
  
    System.err.println(nums);  
}
```

- Here, you can directly print the object.

```
$ java ArrayListPractice  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Here you can't access the values like **nums[2]**
- You can use **nums.get(2)** to get the value at index-2.
- In the above code, we have taken **nums** as **Collection** type. In that *Collection* interface, there is no method called **get()**.
- The type of **nums** should be **List** or **ArrayList** or something like that.

```
public static void main(String[] args) {  
    Collection<Integer> nums = new ArrayList<Integer>();  
    for (int i = 1; i <= 10; i++)  
        nums.add(i);  
  
    System.err.println(nums);  
}
```

- The warning that was coming before, was due to not specifying the type.
- Note: here you can use Wrapper type (Integer, Double etc.) not primitive type (int, double etc).

```
public static void main(String[] args) {  
    List<Integer> nums = new ArrayList<Integer>();  
    for (int i = 1; i <= 10; i++)  
        nums.add(i);  
  
    for (int i = 0; i <= nums.size(); i++)  
        nums.get(i);  
}
```

- Now we used **List** in place of **Collection**, so we can use the **get()** method now.

➤ Set

- ~ Whenever you add some values in a set, sometimes it looks like sorted; but it is **coincidence**.
 - ✦ Internally, it stores elements in a hash table based on the element's hash code.
 - ✦ The iteration order you see depends on how the hash table buckets are organized.

```
public static void main(String[] args) {  
    Set<Integer> st = new HashSet<Integer>();  
    int[] arr = { 62, 54, 82, 21 };  
  
    for (int x : arr)  
        st.add(x);  
  
    for (int val : st)  
        System.err.println(val);  
}
```

```
$ java SetPr  
82  
21  
54  
62
```

- ✦ Set is interface; **HashSet** is class
- ~ If you want a sorted set, then go for **TreeSet** instead of **HashSet**

```
public static void main(String[] args) {  
    Set<Integer> st = new TreeSet<Integer>();  
    int[] arr = { 62, 54, 82, 21 };  
  
    for (int x : arr)  
        st.add(x);  
  
    for (int val : st)  
        System.err.println(val);  
}
```

```
$ java  
21  
54  
62  
82
```


➤ **Map**

```
public static void main(String[] args) {
    Map<String, Integer> students = new HashMap<>();

    students.put(key:"Alok", value:24);
    students.put(key:"Kanha", value:28);
    students.put(key:"Ram", value:30);

    System.err.println(students); // {Alok=24, Kanha=28, Ram=30}
    System.err.println(students.get(key:"Alok")); // 24

    for (String key : students.keySet())
        System.err.print(key + " "); // Alok Kanha Ram
    System.err.println();

    for (int value : students.values())
        System.err.print(value + " "); // 24 28 30
    System.err.println(x:"\n");

    for (String key : students.keySet())
        System.err.println(key + " : " + students.get(key));
    // Alok : 24
    // Kanha : 28
    // Ram : 30
}
```

- The **HashMap** is not **synchronized**.
- To use **synchronized** version, use **HashTable**.

➤ Comparator vs Comparable

- ~ The **Collections** class contains **static methods** that operate or returns collections.

```
@SuppressWarnings({"unchecked", "rawtypes"})
public static <T> void sort(List<T> list, Comparator<? super T> c) {
    list.sort(c);
}
```

- ~ This is the **sort()** method.
- ~ Here **Comparator** is there; Comparator is nothing but a *Functional Interface* containing a **compare()** method.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- ~ In the below code snippet, the list nums will be sorted according to its values.

```
$ java Demo
[43, 31, 72, 29]
[29, 31, 43, 72]
```

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<Integer>();
    nums.add(e:43);
    nums.add(e:31);
    nums.add(e:72);
    nums.add(e:29);

    System.err.println(nums);

    // Collection(s); not Collection
    Collections.sort(nums);

    System.err.println(nums);
}
```

- I want to sort the values by their right most digit:

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<Integer>();
    nums.add(43);
    nums.add(31);
    nums.add(72);
    nums.add(29);

    System.err.println(nums);

    Comparator<Integer> comp = new Comparator<Integer>() {
        @Override
        public int compare(Integer v1, Integer v2) {
            return (v1 % 10) - (v2 % 10);
        }
    };

    Collections.sort(nums, comp); // [31, 72, 43, 29]

    System.err.println(nums);
}
```

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<Integer>();
    nums.add(43);
    nums.add(31);
    nums.add(72);
    nums.add(29);

    System.err.println(nums);

    Collections.sort(nums, (v1, v2) -> (v1 % 10) - (v2 % 10)); // [31, 72, 43, 29]

    System.err.println(nums);
}
```

- Using lambda expression.

```
public static void main(String[] args) {
    List<Student> students = new ArrayList<Student>();
    students.add(new Student(21, "Alice"));
    students.add(new Student(19, "Bob"));
    students.add(new Student(22, "Charlie"));
    students.add(new Student(20, "Diana"));

    for (Student st : students)
        System.err.println(st);

    System.err.println();

    Collections.sort(students);
}
```

- Here, in case of a list of objects, **sort()** method is not working without passing the **comparator**.

```
@SuppressWarnings("unchecked")
public static <T extends Comparable<? super T>> void sort(List<T> list) {
    list.sort(c:null);
}
```

- * It is the **sort()** method, it requires that the Type (i.e. Integer, Double etc etc) should be of **Comparable**.

```
@jdk.internal.ValueBased
public final class Integer extends Number
    implements Comparable<Integer>, Constable, ConstantDesc {
    /**
```

- * You can see, Integer is implementing **Comparable**; so it can be used in the **sort()** method without passing the *comparator*.

```
class Student implements Comparable<Student> {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "Student [age=" + age + ", name=" + name + "]";
    }

    @Override
    public int compareTo(Student that) {
        if (this.age > that.age)
            return 1;
        return -1;
    }
}
```

- * Now, we implemented the **Comparable** interface in the **Student** class, and overridden the **compareTo** method inside it.
- * Now, when we sort without using **comparator**, it'll not give any error.

➤ **forEach** method

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

- It accepts a Consumer typed object.
- Consumer is a functional interface.
- <? super T> means, type should be T or any of its super class.

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```

- This the abstract method that is to be overridden.

```
public static void main(String[] args) {
    Integer[] nums = { 3, 4, 2, 5, 3, 5 };
    List<Integer> ls = new ArrayList<Integer>();
    for (Integer num : nums)
        ls.add(num);

    ls.forEach(new Consumer<Integer>() {
        public void accept(Integer val) {
            System.err.println(val);
        }
    });
}
```

- It can also be written like this.

➤ Stream API

- ⌘ You'll get a method in the **List** objects called **stream()**.
- ⌘ It'll create a **Stream** object containing all the methods that **List** has.
- ⌘ Even if you change something inside that **Stream** object, nothing will affect the original list.
- ⌘ But the condition is: the Stream object can be used **once**. If you use it more than once, it'll give **Run time exception**.

```
public static void main(String[] args) {  
    List<Integer> ls = Arrays.asList(...a:4, 3, 5, 6, 4, 3, 5, 3, 5);  
  
    Stream<Integer> strm = ls.stream();  
  
    strm.forEach(val -> System.err.print(val + " "));  
    System.err.println();  
    strm.forEach(val -> System.err.print(val + " "));  
}
```

```
$ java Demo  
4 3 5 6 4 3 5 3 5  
Exception in thread "main" java.lang.IllegalStateException:  
stream has already been operated upon or closed  
    at java.base/java.util.stream.AbstractPipeline.sourceStageSplitter(AbstractPipeline.java:279)  
    at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:762)  
    at Demo.main(Demo.java:15)
```

- ⌘ Run-Time exception. Only once the **Stream** object can be used.

- ⌘ Using of stream provides so many methods like **map**, **filter**, **flatMap**, **sorted**, ..etc.

These are part of **Stream API**, not part of the **List** or something like that.

```
public static void main(String[] args) {  
    List<Integer> ls = Arrays.asList(...a:4, 3, 5, 6, 4, 3, 5, 3, 5);  
  
    Stream<Integer> s1 = ls.stream();  
    Stream<Integer> s2 = s1.filter(val -> val % 2 == 0);  
    // s1 is used now; can't use it again  
  
    s2.forEach(val -> System.err.print(val + " "));  
    // s2 is used now; can't use it again  
}
```



```

public static void main(String[] args) {
    List<Integer> ls = Arrays.asList(...a:4, 3, 5, 6, 4, 10, 5, 8, 5);

    Integer res = ls.stream()
        .filter(n -> n % 2 == 0)
        .map(n -> n * 2)
        .reduce(identity:0, (a, b) -> a + b);
    System.err.println(res);
}

```

- As the return types of these **stream api's methods** are Stream only, so we can chain these kind of methods.

➤ parallelStream in Java

```

public static void main(String[] args) {
    int size = 10_000; // just to make it eye catching
    List<Integer> nums = new ArrayList<Integer>(size);
    Random ran = new Random();
    for (int i = 1; i <= size; i++)
        nums.add(ran.nextInt(bound:100));

    long t1 = System.currentTimeMillis();
    int sum1 = nums.stream()
        .map(n -> {
            try {Thread.sleep(millis:3);} catch (Exception e) {}
            return n * 2;
        })
        .reduce(identity:0, (a, b) -> a + b);
    long t2 = System.currentTimeMillis();
    int sum2 = nums.parallelStream()
        .map(n -> {
            try {Thread.sleep(millis:3);} catch (Exception e) {}
            return n * 2;
        })
        .reduce(identity:0, (a, b) -> a + b);
    long t3 = System.currentTimeMillis();

    System.err.println("sum1 = " + sum1 + ", sum2 = " + sum2);
    System.err.println("stream : " + (long) (t2 - t1));
    System.err.println("parallelStream : " + (long) (t3 - t2));
}

```

```

$ java Demo
sum1 = 998040, sum2 = 998040
stream : 38450
parallelStream : 2445

```

- parallelStream ran faster (as multiple threads)

```
Thread 1 handles nums[0..2499] → produces sum1  
Thread 2 handles nums[2500..4999] → produces sum2  
Thread 3 handles nums[5000..7499] → produces sum3  
Thread 4 handles nums[7500..9999] → produces sum4
```

```
Final result = sum1 + sum2 + sum3 + sum4
```

⌘ (it works like this)

➤ Optional class

~ It's a class just to avoid the Null Pointer Exception.

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList("Alok", "Laxmi", "Ram", "Hari");  
  
    Optional<String> name = names.stream().filter(str -> str.contains("x"))  
    |  
    | .findFirst();  
    System.err.println(name.orElse("Not found!"));  
  
    // same only; findFirst() returns a object of tyoe "Optional"  
    String name2 = names.stream().filter(str -> str.contains("x"))  
    |  
    | .findFirst().orElse("Not Found");  
    System.err.println(name2);  
}
```

~

➤ Method Reference

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList(...a:"Alok", "Laxmi", "Ram", "Hari");  
    // converting all to upper case  
    List<String> uNames = names.stream()  
        .map(str -> str.toUpperCase())  
        .toList();  
    System.err.println(uNames);  
}
```

- Here, in the **lambda** expression passed inside the **map()** method, we are just calling a **method** i.e. **toUpperCase()** present inside that object i.e. **str**.
- So, when you are just calling a method of an object, no need to write the full lambda expression. Just pass the reference of that method.

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList(...a:"Alok", "Laxmi", "Ram", "Hari");  
    // converting all to upper case  
    List<String> uNames = names.stream()  
        .map(String::toUpperCase)  
        .toList();  
    System.err.println(uNames);  
}
```

- **::** is used to reference a instance method of a class.

```
public static void main(String[] args) {  
    List<String> names = Arrays.asList(...a:"Alok", "Laxmi", "Ram", "Hari");  
    names.forEach(System.out::println);  
}
```

- Now the confusion is: in both the cases, it is acting differently.
 - In first case, its calling that method using the **str** i.e. **str.toUpperCase()**
 - But in second case, its calling that method using the object (**System.out**) provided and passing the **str** inside it.

• **NOTE**

- If the reference is in the form **ClassName::methodName**, the method is called on the object provided by the function (e.g., **str.toUpperCase()**).
- If the reference is in the form **objectName::methodName**, the method is called on that object, with the function-provided value passed as an argument (e.g., **System.out.println(str)**).

➤ Constructor Reference

```
public static void main(String[] args) {
    List<String> names = Arrays.asList(...a:"Alok", "Laxmi", "Ram", "Hari");

    List<Student> students = new ArrayList<Student>();

    students = names.stream()
        .map(name -> new Student(name))
        .toList();

    System.err.println(students);
}
```

- Here, I am creating a **ArrayList** of type **Student**, from a **List** of type **String**, using **map** function.

```
public static void main(String[] args) {
    List<String> names = Arrays.asList(...a:"Alok", "Laxmi", "Ram", "Hari");

    List<Student> students = new ArrayList<Student>();

    students = names.stream()
        .map(Student::new)
        .toList();

    System.err.println(students);
}
```

- It'll do the work.

➤ Supplier interface

- ⌘ It is a *functional interface* that provides a `get` method to create something.

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

```
Supplier<List<Integer>> sp1 = new Supplier<List<Integer>>() {
    public List<Integer> get() {
        return new ArrayList<Integer>();
    }
};

Supplier<List<Integer>> sp2 = () -> new ArrayList<Integer>();

List<Integer> ls1 = sp2.get();
List<Integer> ls2 = sp2.get();
System.err.println("ls1 = " + ls1);
System.err.println("ls2 = " + ls2);
```

- ⌘ Here, I used `sp2` to create 2 `ArrayList<Integer>`

```
$ java Demo
ls1 = []
ls2 = []
```

```
Supplier<List<Integer>> sp2 = ArrayList<Integer>::new;

List<Integer> ls1 = sp2.get();
List<Integer> ls2 = sp2.get();
System.err.println("ls1 = " + ls1);
System.err.println("ls2 = " + ls2);
```

- ⌘ It can also be written like this using the **Constructor reference**.