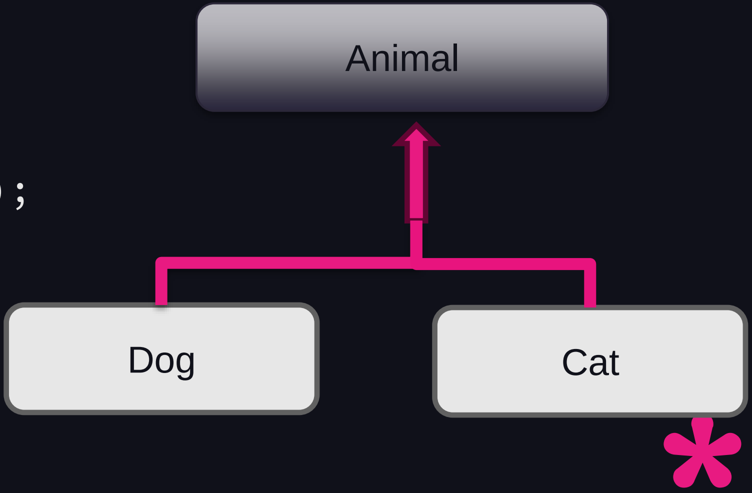
Hierarchical Inheritance

Hierarchical inheritance is a concept in object-oriented programming (OOP) where a single class serves as the base class for multiple derived classes. In other words, it's a type of inheritance where a class inherits from a single superclass, but multiple subclasses inherit from that same superclass. Each subclass can have its own additional methods and properties, as well as inherit the properties and methods of the common superclass.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}
```



Multiple Inheritance

While Java **does not support multiple inheritance of classes** (a class cannot extend more than one class), it does support multiple inheritance of interfaces. A class can implement multiple interfaces, allowing it to inherit method signatures from each interface. This is a way to achieve some level of multiple inheritance while avoiding the complexities and ambiguities associated with multiple inheritance of classes.

Java Interface

In Java, an interface is a reference type that is like a class, but it is a collection of abstract methods. An interface defines a contract or a set of method signatures that classes implementing the interface must provide concrete implementations for. It allows you to achieve abstraction, multiple inheritance of method signatures, and the definition of common behavior across unrelated classes.

Abstract methods are those types of methods that don't require implementation for its declaration

```
Syntax : interface {interface Name}
```

```
// Define an interface
interface Vehicle {
    void start();
    void stop();
}
```

Interface Implementation

```java
// Define an interface
interface Vehicle {
    void start();
    void stop();
}

// Implement the interface in classes
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car starting...");
    }

    @Override
    public void stop() {
        System.out.println("Car stopping...");
    }
}
```

The @Override annotation is used to indicate that the methods are intended to override the methods from the interface, although it's not required by the language; it's a good practice for clarity.

Multiple Inheritance with Interface

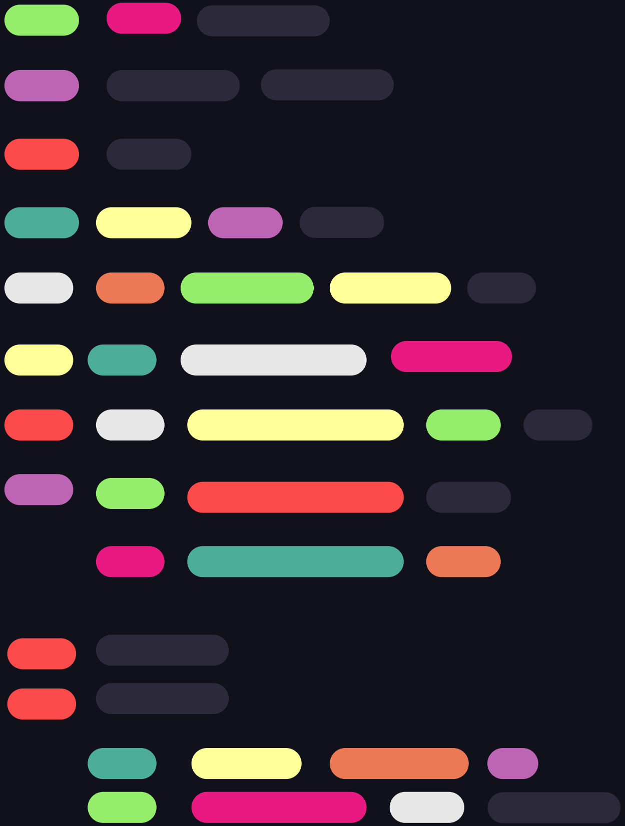
Multiple inheritance is not allowed among classes to avoid complexities and ambiguities that can arise from inheriting from multiple sources. However, Java provides a mechanism to achieve similar functionality through interfaces.

```
class MyClass implements A, B {
    @Override
    public void methodA() {
        System.out.println("Implementation of methodA");
    }

    @Override
    public void methodB() {
        System.out.println("Implementation of methodB");
    }
}
```

```
interface B {
    void methodB();
}

interface A {
    void methodA();
}
```

Access Modifier Scope cont.

	Private	Deful t	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package	No	Yes	Yes	Yes
Same Package - Sub Class	No	Yes	Yes	Yes
Different Package	No	No	No	Yes
Diff Package - Sub Class	No	No	Yes	Yes



Polymorphism

Polymorphism is a core concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass or interface. It enables you to write code that can work with different types of objects in a unified way, enhancing code flexibility and reusability. explain more

There are two main types of polymorphism in Java:

- **Compile-time Polymorphism (Method Overloading)**
- **Run-time Polymorphism (Method Overriding)**

Compile-time Polymorphism (Method Overloading)

This occurs when you have multiple methods in the same class with the same name but different parameter lists (different number or types of parameters). The appropriate method to be called is determined at compile-time based on the method signature.

```
class Calculator {

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

        double add(double a, double b) {
                return a + b;
        }
double add(int a, double b) {
        return a + b;
```

Run-time Polymorphism (Method Overriding)

This occurs when you have a class that inherits from another class or implements an interface, and it provides a specific implementation for a method that is already declared in the superclass or interface. The method to be called is determined at run-time based on the actual object's type

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Java Getters & Setters

In Java, getters and setters are methods used to access and modify the private fields (properties) of a class. They are a common practice for encapsulating the fields and providing controlled access to them. Getters are used to retrieve the values of fields, and setters are used to modify the values of fields.

Getters: A getter method returns the value of a private field.
Setters: A setter method sets the value of a private field.

```
public class Person {
    private int age;

    public int getAge() {
        return this.age;
    }
}
```

```
public class Person {
    private int age;

    public void setAge(int age) {

        this.age=age;
    }
}
```

Exercise -01

In Java, getters and setters are methods used to access and modify the private fields (properties) of a class. They are a common practice for encapsulating the fields and providing controlled access to them. Getters are used to retrieve the values of fields, and setters are used to modify the values of fields.

Getters: A getter method returns the value of a private field.
Setters: A setter method sets the value of a private field.

```java
public class Person {
    private int age;

    public int getAge() {
        return this.age;
    }
}
```

```java
public class Person {
    private int age;

    public void setAge(int age) {

         this.age=age;
    }
}
```

{

Java Programming Week-04

<Mohan De Zoysa –IIT JAVA CERTIFICATION>

● ● ●

}

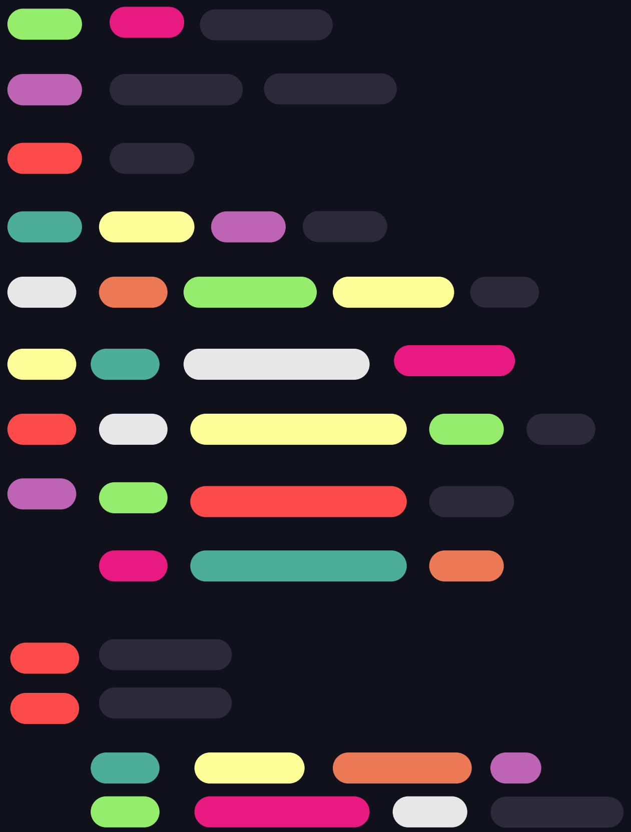
Outline

Arrays and Exception Handling

- Working with arrays and arrayLists
- Exception handling and error propagation
- Handling checked and unchecked exceptions
- Custom exception classes

Arrays and ArrayLists

OOP



Java Array

An array is a container object that holds a fixed number of values of a single type.
The length of an array is established when the array is created.

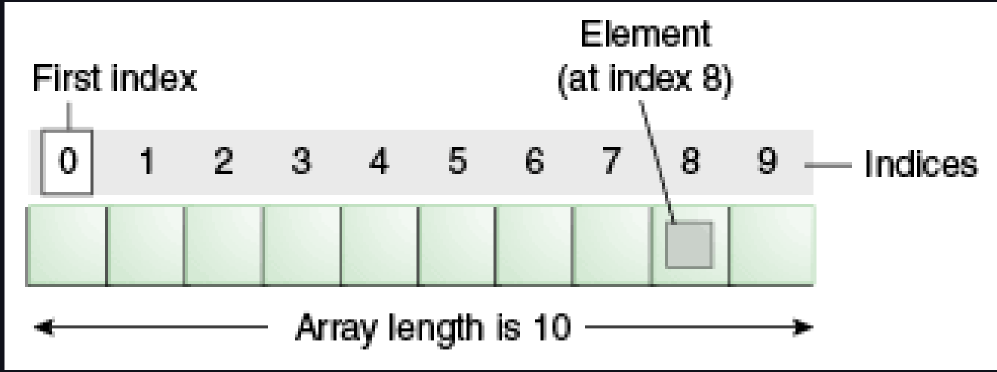
After creation, its length is fixed.

In Java, an array is used to store a list of elements of the same datatype.

Java Array - Index

- An index refers to an element’s position within an array.
- The index of an array starts from 0 and goes up to one less than the total length of the array.

```
int[] marks = {50, 55, 60, 70, 80};
System.out.println(marks[0]);
// Output: 50
System.out.println(marks[4]);
// Output: 80
```



Array creation in Java

In Java, an array can be created in the following ways:

Using the {} notation, by adding each element all at once.

```
// Create an array of 5 int elements
int[] marks = {10, 20, 30, 40, 50};
```

Using the new keyword and assigning each position of the array individually.

```
int[] marks = new int[3];
marks[0] = 50;
marks[1] = 70;
marks[2] = 93;
```

Array creation in Java cont.

Array of Integer

```java
int[] myNum = {10, 20, 30, 40};
```

Array of strings

```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Array of Characters

```java
char[] JavaCharArray = {'r', 's', 't', 'u', 'v'};
```

Working with Array

Array of Integer

```
int[] myNum = {10, 20, 30, 40};
```

Array of strings

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Array of Characters

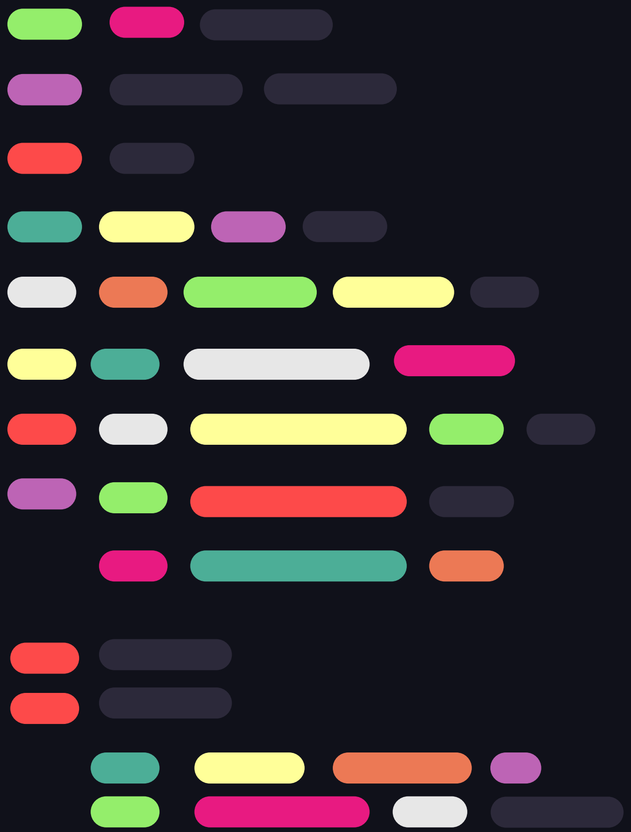
```
char[] JavaCharArray = {'r', 's', 't', 'u', 'v'};
```

Working with Array

Changing an Element Value

To change an element value, select the element via its index and use the assignment operator to set a new value.

```
int[] nums = {1, 2, 0, 4};
// Change value at index 2
nums[2] = 3;
```



Java ArrayList

In Java, an ArrayList is used to represent a dynamic list.

While Java arrays are fixed in size (the size cannot be modified), an ArrayList allows flexibility by being able to both add and remove elements.

Create an ArrayList

Changing an Element Value

To change an element value, select the element via its index and use the assignment operator to set a new value.

```
// import the ArrayList package
import java.util.ArrayList;

// create an ArrayList called students
ArrayList<String> students = new ArrayList<String>();
```

Modifying ArrayLists in Java

An ArrayList can easily be modified using built in methods.

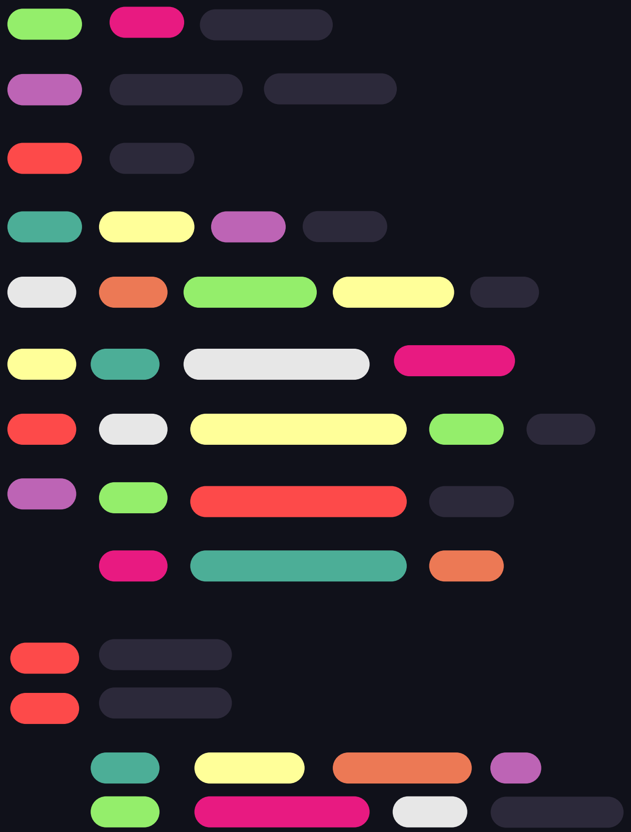
To add elements to an ArrayList, you use the add() method. The element that you want to add goes inside of the ().

To remove elements from an ArrayList, you use the remove() method.

Inside the () you can specify the index of the element that you want to remove. Alternatively, you can specify directly the element that you want to remove.

Modifying ArrayLists in Java cont.

```
// create an ArrayList called studentList, which initially holds []
     ArrayList<String> studentList = new ArrayList<String>();

    // add students to the ArrayList
    studentList.add("John");
    studentList.add("Lily");
    studentList.add("Samantha");
    studentList.add("Tony");

    // remove John from the ArrayList, then Lily
    studentList.remove(0);
    studentList.remove("Lily");
```



Java Exceptions

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

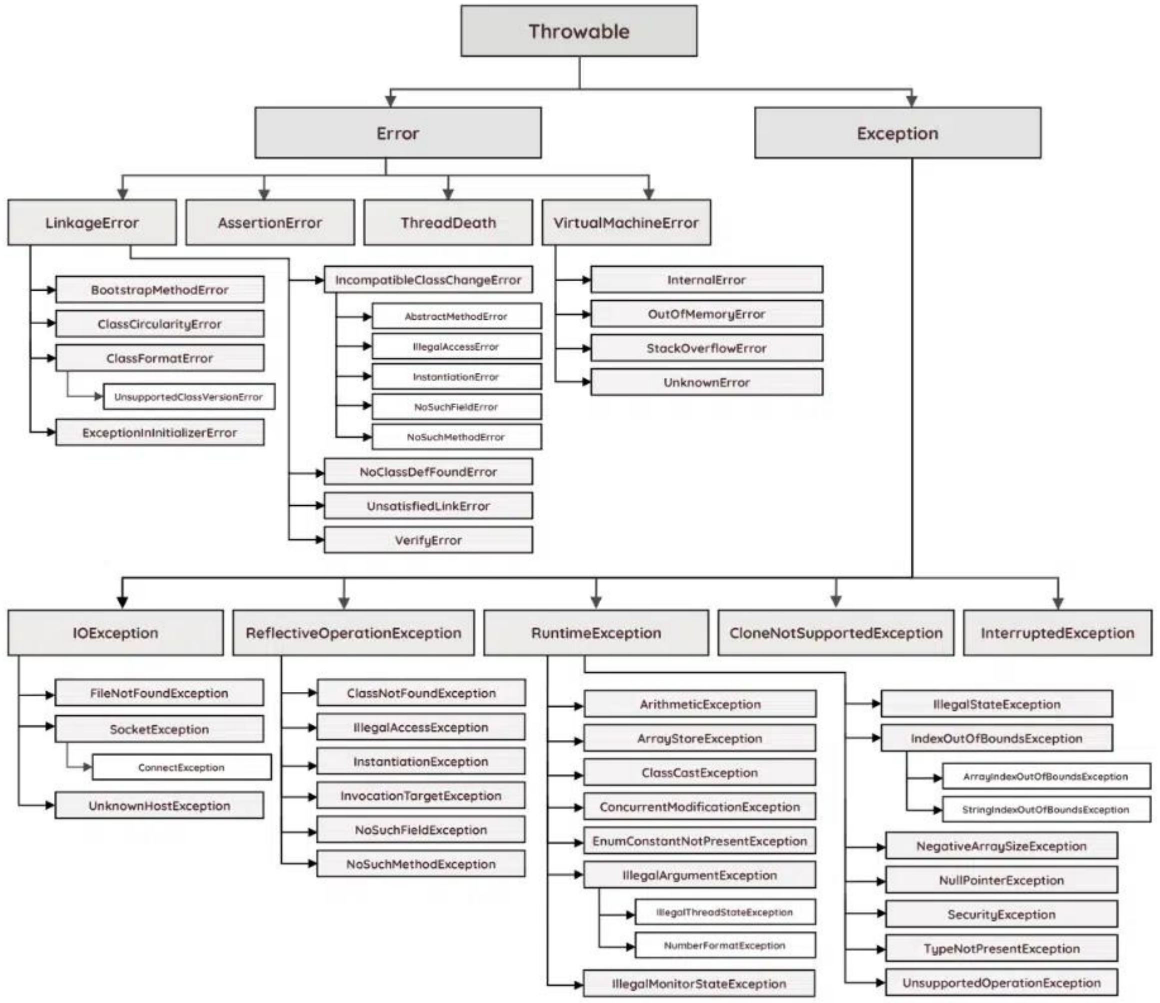
The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Java Exceptions Hierarchy

The class at the top of the exception class hierarchy is the Throwable class, which is a direct subclass of the Object class. Throwable has two direct subclasses - Exception and Error.

The Throwable class is the superclass of all errors and exceptions in the Java language.

Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.


Throwable
Error
Exception
LinkageError
AssertionError
ThreadDeath
VirtualMachineError
BootstrapMethodError
ClassCircularityError
ClassFormatError
UnsupportedClassVersionError
ExceptionInInitializerError
IncompatibleClassChangeError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
IOException
ReflectiveOperationException
RuntimeException
CloneNotSupportedException
InterruptedException
FileNotFoundException
SocketException
ConnectException
UnknownHostException
ClassNotFoundException
IllegalAccessException
InstantiationException
InvocationTargetException
NoSuchFieldException
NoSuchMethodException
ArithmeticException
ArrayStoreException
ClassCastException
ConcurrentModificationException
EnumConstantNotPresentException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
TypeNotPresentException
UnsupportedOperationException

Java Errors vs Exceptions

The Exception class is used for exception conditions that the application may need to handle. Examples of exceptions include IllegalArgumentException, ClassNotFoundException and NullPointerException.

The Error class is used to indicate a more serious problem in the architecture and should not be handled in the application code. Examples of errors include InternalError, OutOfMemoryError and AssertionError.

According to the official documentation, an error “indicates serious problems that a reasonable application should not try to catch.” This refers to problems that the application can not recover from - they should be dealt with by modifying application architecture or by refactoring code.

Java Errors vs Exceptions

The Exception class is used for exception conditions that the application may need to handle. Examples of exceptions include IllegalArgumentException, ClassNotFoundException and NullPointerException.

The Error class is used to indicate a more serious problem in the architecture and should not be handled in the application code. Examples of errors include InternalError, OutOfMemoryError and AssertionError.

According to the official documentation, an error “indicates serious problems that a reasonable application should not try to catch.” This refers to problems that the application can not recover from - they should be dealt with by modifying application architecture or by refactoring code.

Java Errors Sample

Example of a method that throws a error, which is not handled in code:

```
public static void print(String myString) {

        print(myString);
}
```

In this example, the recursive method “print” calls itself over and over again until it reaches the maximum size of the Java thread stack, at which point it exits with a StackOverflowError:

```
Exception in thread "main" java.lang.StackOverflowError
at StackOverflowErrorExample.print(StackOverflowErrorExample.java:6)
```

Checked Exceptions

Exceptions that can occur at compile-time are called checked exceptions since they need to be explicitly checked and handled in code.

Classes that directly inherit Throwable - except RuntimeException and Error - are checked exceptions e.g. IOException, InterruptedException etc.

```
public void writeToFile() {
try (BufferedWriter bw = new BufferedWriter(new FileWriter("myFile.txt"))) {
        bw.write("Test");
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

Unchecked Exceptions

Unchecked exceptions can be thrown "at any time" (i.e. run-time). Therefore, methods don't have to explicitly catch or throw unchecked exceptions. Classes that inherit RuntimeException are unchecked exceptions e.g. ArithmeticException, NullPointerException.

```
public static void main(String[] args) {
      String stringToPrint=null;
      System.out.println(stringToPrint.length());

}
```

```
Exception in thread "main" java.lang.NullPointerException
        at Main.main(Main.java:11)
```

Java try block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch
```

Java catch block

Catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){
}
```

Finally block

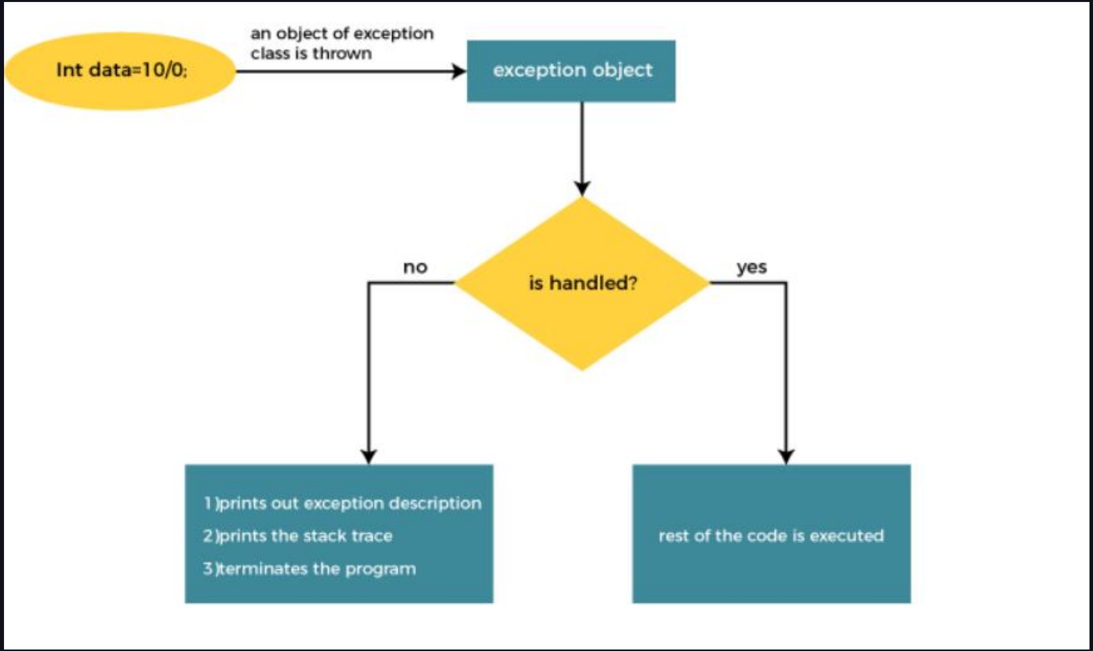
Java finally block is a block used to execute important code such as closing the connection, etc.
Java finally block is always executed whether an exception is handled or not.

Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
The finally block follows the try-catch block.

Syntax of Java try-catch

```
try {
      int x= 50/0;
} catch (ArithmeticException e) {
      //Catch block
} finally {
      // this block will execute after all catch blocks
}
```

Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Example without Exception handing

```
public class TryCatchExample1 {

  public static void main(String[] args) {

     int data=50/0; //may throw exception
     System.out.println("rest of the code");

  }

}
```

Output :

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Example without Exception handing

```
public class TryCatchExample2 {

        public static void main(String[] args) {
            try
            {
            int data=50/0; //may throw exception
            }
                //handling the exception
            catch(ArithmeticException e)
            {
                System.out.println(e);
            }
            System.out.println("rest of the code");
        }

    }
```

Output :

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Throws

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{

    //Method Code
}
```

Throws

Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.

Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only

The throw keyword is followed by an instance of Exception to be thrown.

Syntax of Java throw

```
if (num < 1) {
  throw new ArithmeticException("\nNumber is negative, cannot calculate square");
}
```

Custom Exception Class

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.