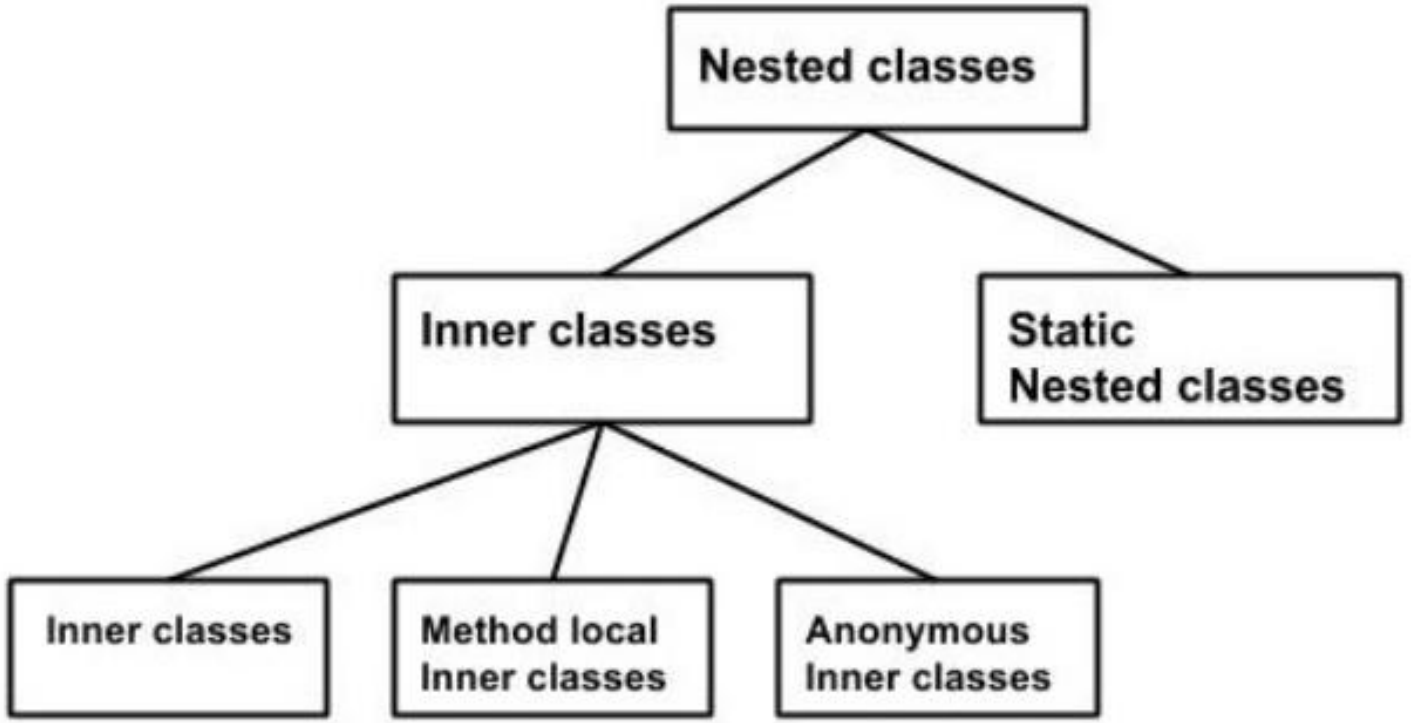
Unit 3

Nested Classes in Java

- In Java, it is possible to define a class within another class, such classes are known as nested classes.
- They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation and creates more readable and maintainable code.
- The scope of a nested class is bounded by the scope of its enclosing class. Thus in the below example, the class *NestedClass* does not exist independently of the class *OuterClass*.
- A nested class has access to the members, including private members, of the class in which it is nested. But the enclosing class does not have access to the member of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package-private*(default).
- Nested classes are divided into two categories:
  - **static nested class:** Nested classes that are declared *static* are called static nested classes.
  - **inner class:** An inner class is a non-static nested class.

- **Syntax**

```
class OuterClass
{
...
    class NestedClass
    {
        ...
    }
}
```



- **Types of Inner Classes**

There are basically four types of inner classes in java.

1. Nested Inner Class
2. Method Local Inner Classes
3. Static Nested Classes
4. Anonymous Inner Classes

1. Nested Inner Class

- It can access any private instance variable of the outer class.
- Like any other instance variable, we can have access modifier private, protected, public, and default modifier.
- Like class, an interface can also be nested and can have access specifiers.
- Example1:

```
// Java Program to Demonstrate Nested class

// Class 1
// Helper classes
class Outer {

    // Class 2
    // Simple nested inner class
    class Inner {

        // show() method of inner class
        public void show()
        {

            // Print statement
            System.out.println("In a nested class method");
        }
    }
}
```

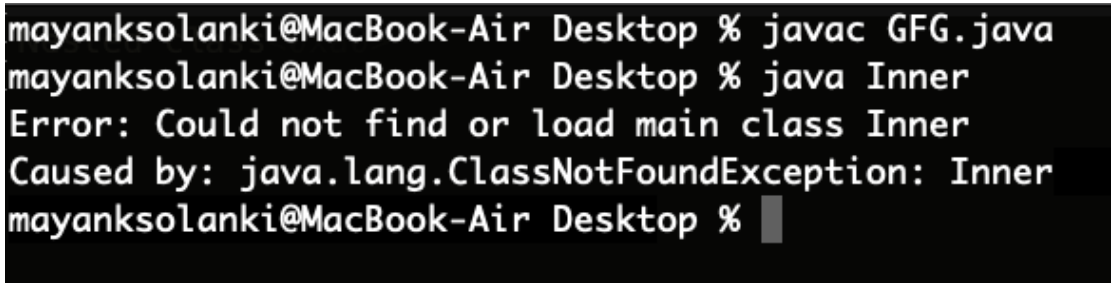
```
// Class 2
// Main class
class Main {

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

        // Note how inner class object is created inside
        // main()
        Outer.Inner in = new Outer().new Inner();

        // Calling show() method over above object created
        in.show();
    }
}
```

- **Output:**
  - In a nested class method

- Example2:

```
// Java Program to Demonstrate Nested class
// Where Error is thrown

// Class 1
// Outer class
class Outer {

    // Method defined inside outer class
    void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");
    }
// Class 2
    // Inner class
    class Inner {

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

            // Display message for better readability
            System.out.println("inside inner class Method");
        }
    }
}
```

- **Output:**



2. Method Local Inner Classes

- Inner class can be declared within a method of an outer class which we will be illustrating in the below example where Inner is an inner class in outerMethod().
- Example1:

```
// Java Program to Illustrate Inner class can be
// declared within a method of outer class

// Class 1
// Outer class
class Outer {

    // Method inside outer class
    void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");
```

```
    // Class 2
        // Inner class
        // It is local to outerMethod()
        class Inner {

            // Method defined inside inner class
            void innerMethod()
            {

                // Print statement whenever inner class is
                // called
                System.out.println("inside innerMethod");

    }
        }

        // Creating object of inner class
        Inner y = new Inner();

        // Calling over method defined inside it
        y.innerMethod();
    }
}
```

```
    // Class 3
    // Main class
    class GFG {

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

            // Creating object of outer class inside main()
            // method
            Outer x = new Outer();

            // Calling over the same method
            // as we did for inner class above
            x.outerMethod();
        }
    }
```

- **Output:**

inside outerMethod

inside innerMethod

- Method Local inner classes can’t use a local variable of the outer method until that local variable is not declared as final.
- For example, the following code generates a compiler error.
- Example2:

```
class Outer {
  void outerMethod() {
    int x = 98;
    System.out.println("inside outerMethod");
    class Inner {
      void innerMethod() {
        System.out.println("x= "+x);
      }
    }
    Inner y = new Inner();
    y.innerMethod();
  }
}

class MethodLocalVariableDemo {
  public static void main(String[] args) {
    Outer x=new Outer();
    x.outerMethod();
  }
}
```

- Output:

```
inside outerMethod
x= 98
```

- But the following code compiles and runs fine
- Example 3:

```
class Outer {
   void outerMethod() {
      final int x=98;
      System.out.println("inside outerMethod");
      class Inner {
         void innerMethod() {
            System.out.println("x = "+x);
         }
      }
```

```
Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
    public static void main(String[] args){
      Outer x = new Outer();
      x.outerMethod();
    }
}
```

- Output:

```
inside outerMethod
x = 98
```

- The main reason we need to declare a local variable as a final is that the local variable lives on the stack till the method is on the stack but there might be a case the object of the inner class still lives on the heap.
- Method local inner class can’t be marked as private, protected, static, and transient but can be marked as abstract and final, but not both at the same time.

3. Static Nested Classes

- Static nested classes are not technically inner classes. They are like a static member of outer class.
- Example:

```
// Java Program to Illustrate Static Nested Classes


// Importing required classes
import java.util.*;


// Class 1
// Outer class
class Outer {

  // Method
    private static void outerMethod()
    {

        // Print statement
        System.out.println("inside outerMethod");
    }
```

```
// Class 2
    // Static inner class
    static class Inner {

        public static void display()
        {

            // Print statement
            System.out.println("inside inner class Method");

            // Calling method inside main() method
            outerMethod();
        }
    }
}
```

```
// Class 3
// Main class
class GFG {

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

        // Calling method static display method rather than an instance of that class.
        Outer.Inner.display();
    }
}
```

- **Output:**

```
inside inner class Method
inside outerMethod
```

4. Anonymous Inner Classes

- Anonymous inner classes are declared without any name at all.
- They are created in two ways.
  - As a subclass of the specified type
  - As an implementer of the specified interface

1. As a subclass of the specified type

```
// Java Program to Illustrate
Anonymous Inner classes
// Declaration Without any Name
// As a subclass of the specified type


// Importing required classes
import java.util.*;
// Class 1
// Helper class
class Demo {


    // Method of helper class
    void show()
    {
        // Print statement
        System.out.println(
            "i am in show method of
super class");
    }
}
```

```
// Class 2
// Main class
class Flavor1Demo {

   //  An anonymous class with Demo as base class
   static Demo d = new Demo() {
      // Method 1
      // show() method
      void show()
      {
         // Calling method show() via super keyword
         // which refers to parent class
         super.show();

         // Print statement
         System.out.println("i am in Flavor1Demo
class");
      }
   };

   // Method 2
   // Main driver method
   public static void main(String[] args)
   {
      // Calling show() method inside main() method
      d.show();
   }
}
```

- **Output:**

```
i am in show method of super class
i am in Flavor1Demo class
```

- In the above code, we have two classes Demo and Flavor1Demo.
- Here demo act as a super-class and the anonymous class acts as a subclass, both classes have a method show().
- In anonymous class show() method is overridden.

2. As an implementer of the specified interface

```
// Java Program to Illustrate Anonymous Inner Classes
// Declaration Without Any Name
// As an implementer of Specified interface

// Interface
interface Hello {

    // Method defined inside interface
    void show();
}

// Main class
class GFG {

    // Class implementing interface
    static Hello h = new Hello() {

        // Method 1
        // show() method inside main class
        public void show()
        {
            // Print statement
            System.out.println("i am in anonymous class");
        }
    };
```

```
// Method 2
    // Main driver method
    public static void main(String[] args)
    {
        // Calling show() method inside
main() method
        h.show();
    }
}
```

- Output:

i am in anonymous class

- **Explanation:**
- In the above code, we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello.
- Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement an interface at a time.

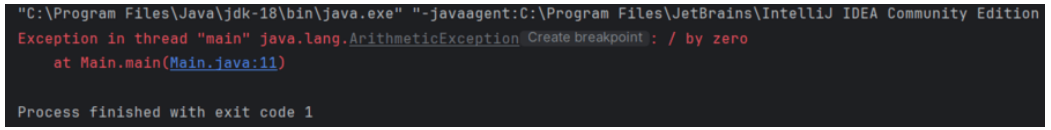
- **The benefits of using inner classes in Java are:**
  - Encapsulation: Inner classes can access private variables and methods of the outer class. This helps to achieve encapsulation and improves code readability.
  - Code Organization: Inner classes allow you to group related code together in one place. This makes your code easier to understand and maintain.
  - Better Access Control: Inner classes can be declared as private, which means that they can only be accessed within the outer class. This provides better access control and improves code security.
  - Callbacks: Inner classes are often used for implementing callbacks in event-driven programming. They provide a convenient way to define and implement a callback function within the context of the outer class.
  - Polymorphism: Inner classes can be used to implement polymorphism. You can define a class hierarchy within the outer class and then create objects of the inner classes that implement the different subclasses.
  - Reduced Code Complexity: Inner classes can reduce the complexity of your code by encapsulating complex logic and data structures within the context of the outer class.
  - Overall, the use of inner classes can lead to more modular, maintainable, and flexible code.

- **Java Exception Handling**
  - Exception handling in Java allows developers to manage runtime errors effectively by using mechanisms like try-catch block, finally block, throwing Exceptions, Custom Exception handling, etc.
  - An Exception is an unwanted or unexpected event that occurs during the execution of a program (i.e., at runtime) and disrupts the normal flow of the program's instructions.
  - It occurs when something unexpected things happen, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.
  - Exception in Java is an error condition that occurs when something wrong happens during the program execution.

- Example:

```
import java.io.*;

class Geeks {
    public static void main(String[] args)
    {
        int n = 10;
        int m = 0;

        int ans = n / m;

        System.out.println("Answer: " + ans);
    }
}
```

- Output:



- Exception handling in Java is an effective mechanism for managing runtime errors to ensure the application's regular flow is maintained.
- Some Common examples of exceptions include ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- By handling these exceptions, Java enables developers to create robust and fault-tolerant applications.

- Example:

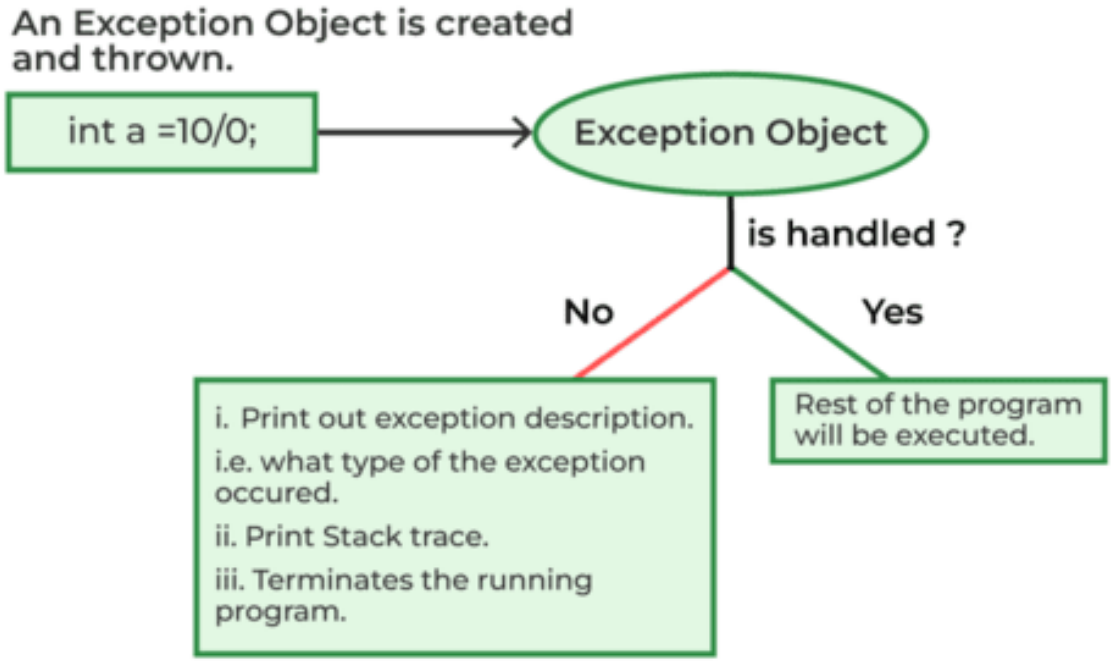
```
// Java program to demonstrates handling
// the exception using try-catch block
import java.io.*;


class Geeks {
    public static void main(String[] args)
    {
        int n = 10;
        int m = 0;
```

```
try {

        // Code that may throw an exception
        int ans = n / m;
        System.out.println("Answer: " + ans);
    }
    catch (ArithmeticException e) {

        // Handling the exception
        System.out.println(
            "Error: Division by zero is not allowed!");
    }
    finally {
        System.out.println(
            "Program continues after handling the
exception.");
    }
  }
}
```

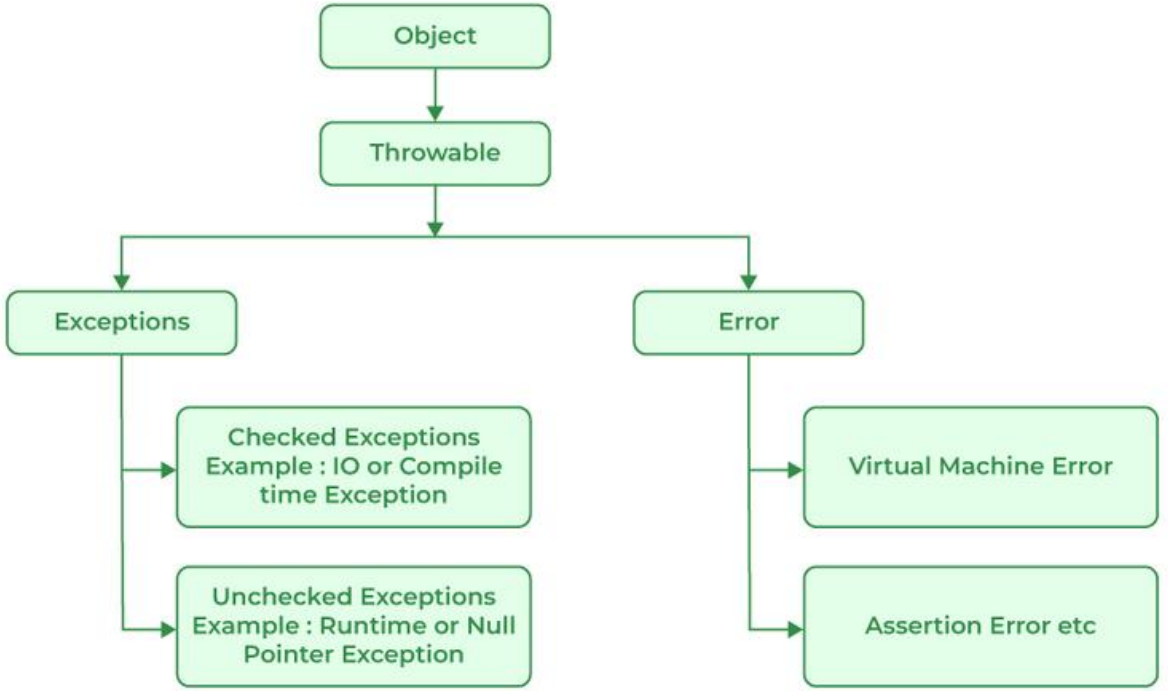
- Output:

Error: Division by zero is not allowed!

Program continues after handling the exception.



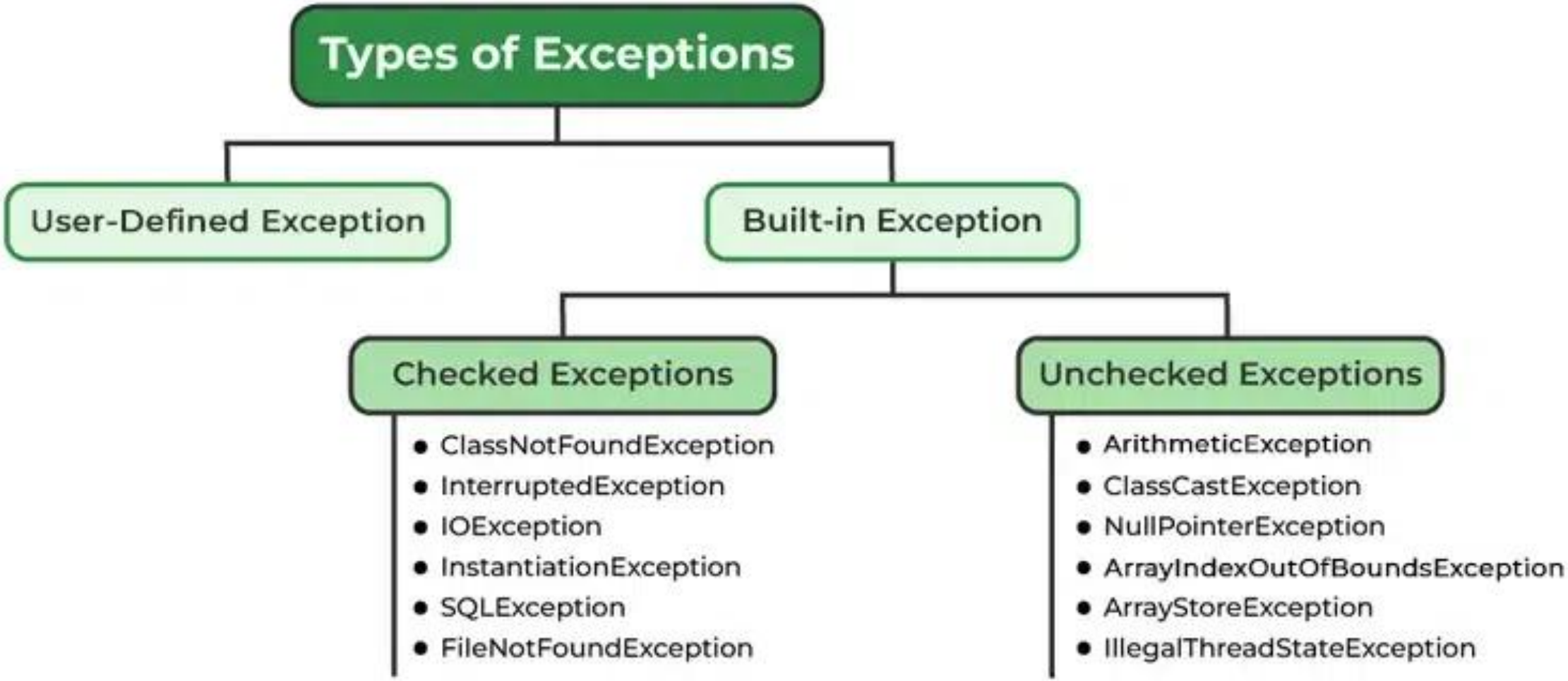
- **Java Exception Hierarchy**
- All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy.
- One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch.
- NullPointerException is an example of such an exception.
- Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).



- Exceptions can occur due several reasons, such as:
  - Invalid user input
  - Device failure
  - Loss of network connection
  - Physical limitations (out-of-disk memory)
  - Code errors
  - Out of bound
  - Null reference
  - Type mismatch
  - Opening an unavailable file
  - Database errors
  - Arithmetic errors

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
- Errors are usually beyond the control of the programmer, and we should not try to handle errors.
- Difference Between Exception and Error

Aspect	Error	Exception
Definition	An Error indicates a serious problem that a reasonable application should not try to catch.	Exception indicates conditions that a reasonable application might try to catch
Cause	Caused by issues with the JVM or hardware.	Caused by conditions in the program such as invalid input or logic errors.
Examples	OutOfMemoryError StackOverFlowError	IOException NullPointerException


Types of Exceptions
User-Defined Exception
Built-in Exception
Checked Exceptions
ClassNotFoundException
InterruptedException
IOException
InstantiationException
SQLException
FileNotFoundException
Unchecked Exceptions
ArithmeticException
ClassCastException
NullPointerException
ArrayIndexOutOfBoundsException
ArrayStoreException
IllegalThreadStateException

- Exceptions can be categorized in two ways:
  - Built-in Exceptions
    - Checked Exception
    - Unchecked Exception
  - User-Defined Exceptions
- **Built-in Exception**
  - Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution.
  - **Checked Exceptions**
    - Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler. Examples of Checked Exception are listed below:
      - ClassNotFoundException: Throws when the program tries to load a class at runtime but the class is not found because its not present in the correct location or it is missing from the project.
      - InterruptedException: Thrown when a thread is paused and another thread interrupts it.
      - IOException: Throws when input/output operation fails
      - InstantiationException: Thrown when the program tries to create an object of a class but fails because the class is abstract, an interface, or has no default constructor.
      - SQLException: Throws when there's an error with the database.
      - FileNotFoundException: Thrown when the program tries to open a file that doesn't exist

- **Unchecked Exceptions**
  - The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.
  - Examples of Unchecked Exception are listed below:
    - ArithmeticException: It is thrown when there's an illegal math operation.
    - ClassCastException: It is thrown when you try to cast an object to a class it does not belongs to.
    - NullPointerException: It is thrown when you try to use a null object (e.g. accessing its methods or fields)
    - ArrayIndexOutOfBoundsException: It occurs when we try to access an array element with an invalid index.
    - ArrayStoreException: It happens when you store an object of the wrong type in an array.
    - IllegalThreadStateException: It is thrown when a thread operation is not allowed in its current state

- **User-Defined Exception**
  - Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called “user-defined Exceptions“.
  - **Methods to Print the Exception Information**

Method	Description
printStackTrace()	Prints the full stack trace of the exception, including the name, message, and location of the error.
toString()	Prints exception information in the format of the Name of the exception.
getMessage()	Prints the description of the exception.

- **Try-Catch Block**
  - A try-catch block in Java is a mechanism to handle exception. The try block contains code that might thrown an exception and the catch block is used to handles the exceptions if it occurs.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

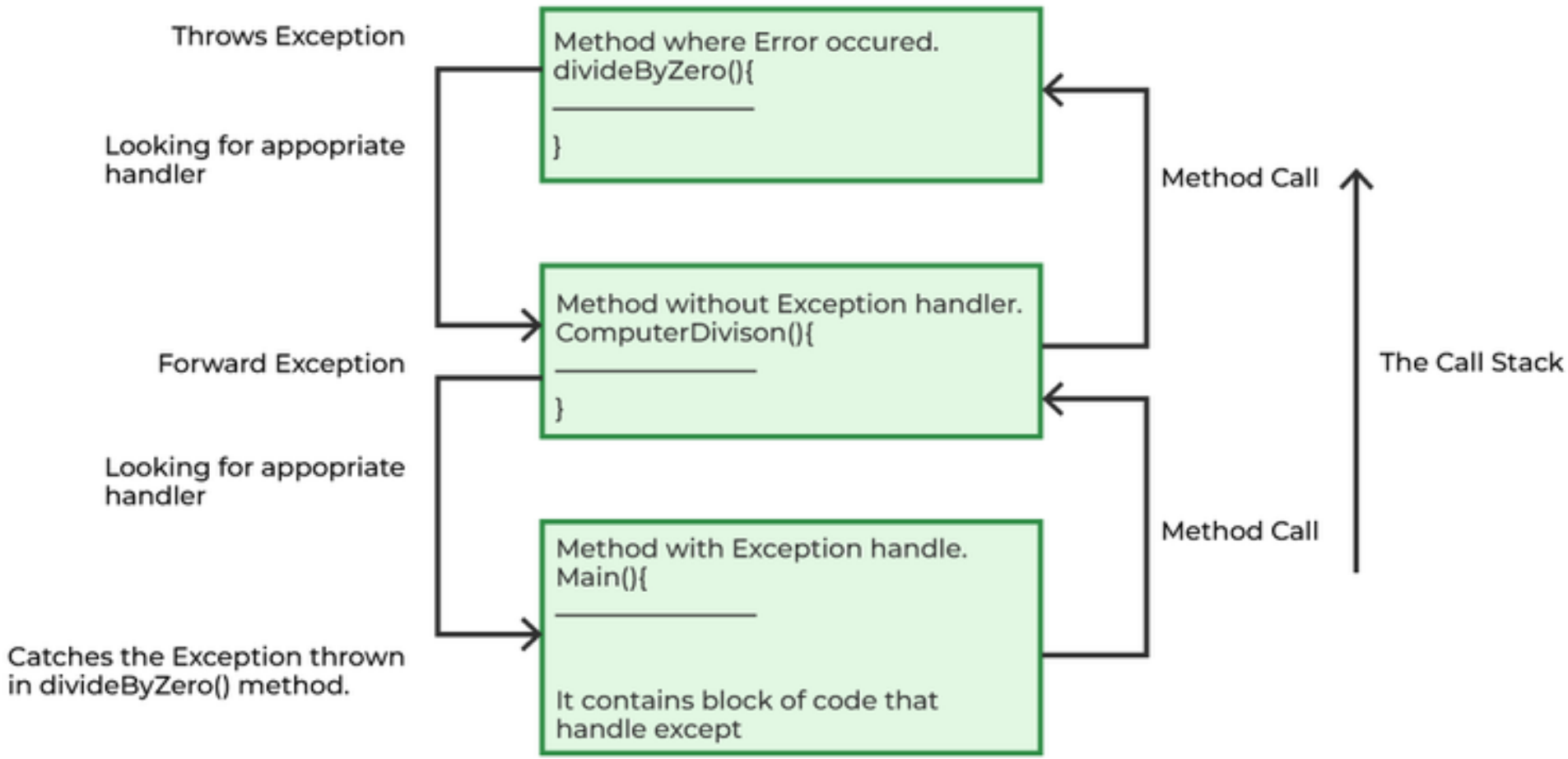
- **finally Block**
  - The finally Block is used to execute important code regardless of weather an exception occurs or not.
  - finally block is always executes after the try-catch block. It is also used for resource cleanup.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}finally{
// cleanup code
}
```

- **Handling Multiple Exception**
  - We can handle multiple type of exceptions in Java by using multiple catch blocks, each catching a different type of exception.

```
try {
    // Code that may throw an exception
} catch (ArithmeticException e) {
    // Code to handle the exception
} catch(ArrayIndexOutOfBoundsException e){
    //Code to handle the anothert exception
}catch(NumberFormatException e){
     //Code to handle the anothert exception
}
```

- **Default Exception Handling**
- When an Exception occurs, the JVM Creates an exception object containing the error name, description, and program state. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred.
- This ordered list of methods is called Call Stack. Now the following procedure will happen.
  - The run-time system searches the call stack for an Exception handler
  - It starts searching from the method where the exception occurred and proceeds backward through the call stack.
  - If a handler is found, the exception is passed to it.
  - If no handler is found, the default exception handler terminates the program and prints the stack trace.

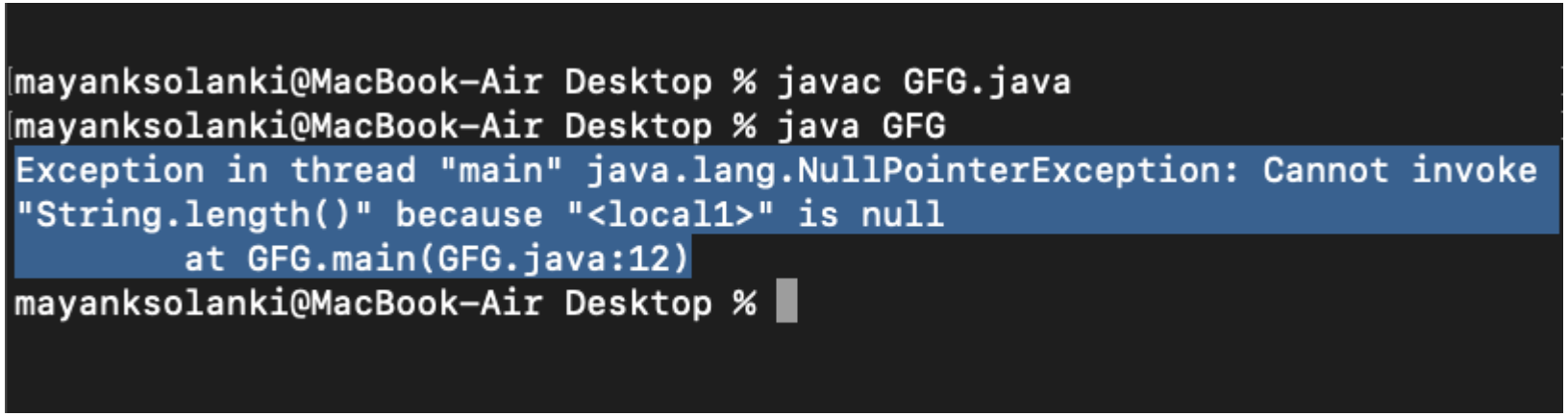


The Call Stack and searching the call stack for exception handler.

- Example:

```
class Geeks{
        public static void main(String args[]){
    // Taking an empty string
    String s = null;
    // Getting length of a string
    System.out.println(s.length());
   }
}
```

- Output:



- Example for a run-time system searches for appropriate exception handling code on the call stack.

```
// Class
// ExceptionThrown
class Geeks {

    // It throws the Exception(ArithmeticException)
    // Appropriate Exception handler is not found
    // within this method
    static int divideByZero(int a, int b)
    {

        // this statement will cause ArithmeticException
        // (/by zero)
        int i = a / b;

        return i;
    }

    // The runTime System searches the appropriate
    // Exception handler in method also but couldn't have
    // found. So looking forward on the call stack
    static int computeDivision(int a, int b)
    {

        int res = 0;

        // Try block to check for exceptions
        try {

            res = divideByZero(a, b);
        }
// Catch block to handle NumberFormatException
        // exception doesn't matches with
        // ArithmeticException
```

```
catch (NumberFormatException ex) {

        System.out.println(
            "NumberFormatException is occurred");
    }
    return res;
  }
// Found appropriate Exception handler
  // i.e. matching catch block.
  public static void main(String args[])
  {

    int a = 1;
    int b = 0;

    // Try block to check for exceptions
    try {
      int i = computeDivision(a, b);
    }
```

```
    // Catch block to handle ArithmeticException
    // exceptions
    catch (ArithmeticException ex) {

      // getMessage() will print description
      // of exception(here / by zero)
      System.out.println(ex.getMessage());
    }
  }
}
```

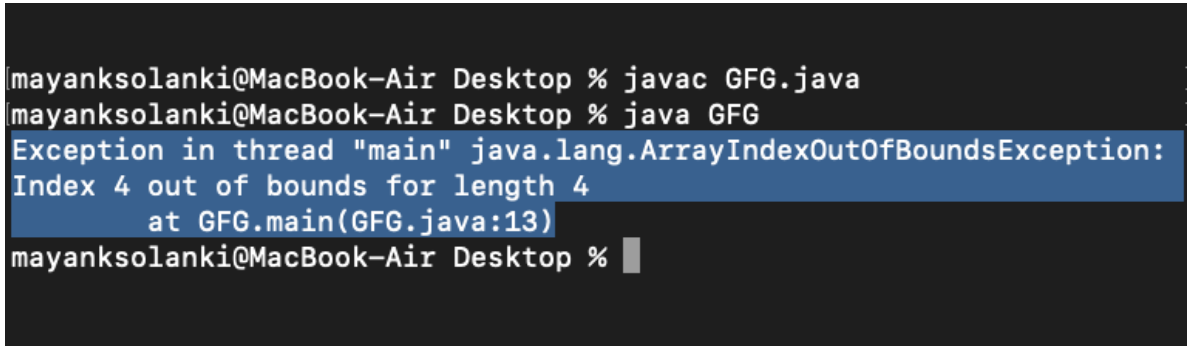
- Output:

/ by zero

- Customized Exception Handling
  - Java exception handling uses five keywords: try, catch, throw and throws, and finally. Code that might cause an exception goes in the try block.
  - If an exception occurs, it is caught using catch. We can throw exceptions manually with throw, and methods must declare exceptions they can throw using throws.
  - The finally block is used for code that must run after try, whether an exception occurs or not.
- Need for try-catch clause (Customized Exception Handling)

```
// Java Program to Demonstrate
// Need of try-catch Clause
class Geeks {
    public static void main(String[] args) {
        // Taking an array of size 4
        int[] arr = new int[4];
        // Now this statement will cause an exception
        int i = arr[4];
        // This statement will never execute
        // as above we caught with an exception
        System.out.println("Hi, I want to execute");
    }
}
```

- Output:



- Explanation:

In the above example, an array is defined with size i.e. we can access elements only from index 0 to 3. But we trying to access the elements at index 4 (by mistake) that is why it is throwing an exception. In this case, JVM terminates the program abnormally. The statement System.out.println("Hi, I want to execute"); will never execute. To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

- **Advantages of Exception Handling**
  - Provision to Complete Program Execution
  - Easy Identification of Program Code and Error-Handling Code
  - Propagation of Errors
  - Meaningful Error Reporting
  - Identifying Error Types