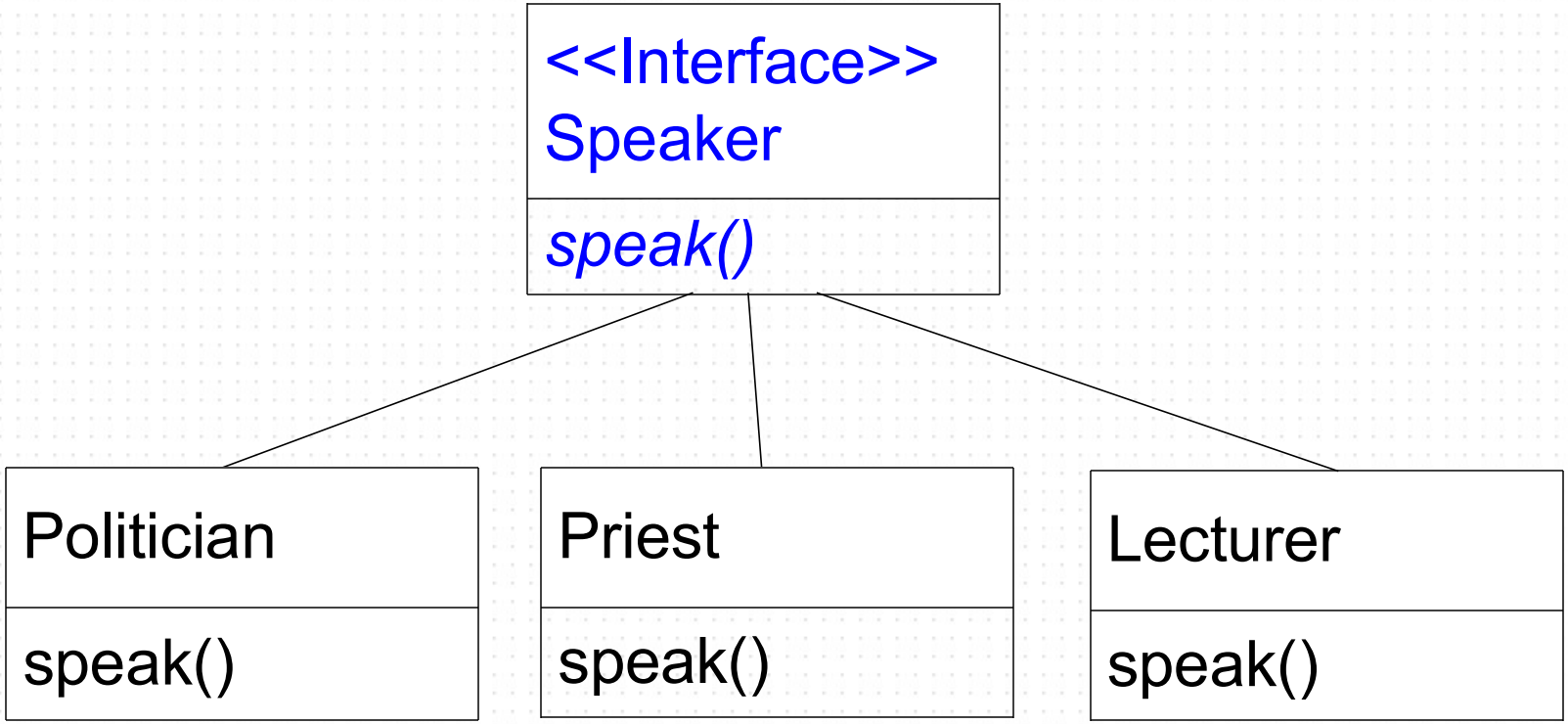
UNIT III

TABLE OF CONTENTS

INTERFACES

- An interface is basically a kind of class—it contains methods and variables, but they have to be only *abstract classes and final fields/variables*.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can *implement any number of interfaces*, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

Interface - Example



Interfaces Definition

Syntax (appears like abstract class):

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration
}
```

An interface is declared by using the interface keyword.

Example:

```
interface Speaker
    { public void
    speak( );
}
```


Implementing Interfaces

Interfaces are used ***like super-classes*** who properties are inherited by classes.

This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, ...]
{
    // Body of Class
}
```

Implementing Interfaces Example

```
class  Politician implements Speaker
        { public void speak(){
            System.out.println("Talk politics");
        }
}
```

```
class  Priest implements Speaker
        { public void speak(){
            System.out.println("Religious Talks");
        }
}
```

```
class  Lecturer implements Speaker
        { public void speak(){
            System.out.println("Talks Object Oriented Design and Programming!");
        }
}
```

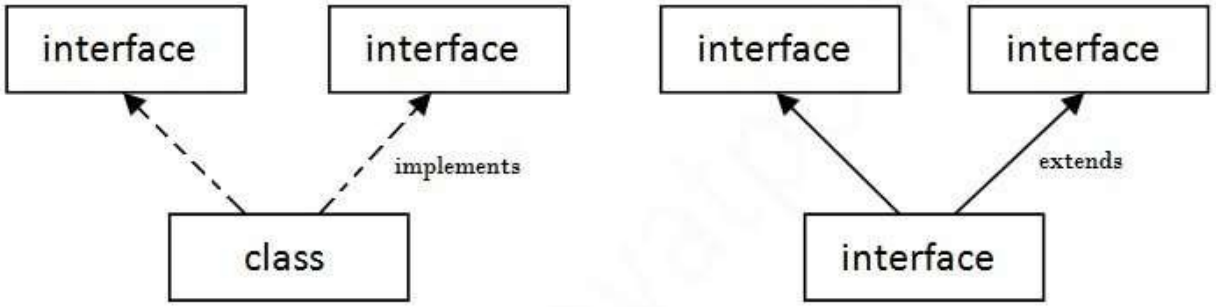
Java Interface Example

```
interface printable{
void print();
}
class A6 implements
printable{ public void print()
{System.out.println("Hello");
}

public static void main(String
args[]){ A6 obj = new A6();
obj.print();
 }
}
```

Multiple inheritance in Java by interface

If a class implements *multiple interfaces*, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements
Printable,Showable{ public void
print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String
args[]){ A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- multiple inheritance is not supported in the case of class because of ambiguity.
- However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
 class TestInterface3 implements Printable,
Showable{ public void
print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no

Interface inheritance/ Extending Interfaces

- A class implements an interface, but one interface extends another interface.
- Like classes, interfaces can also be extended.
- The new sub-interface will inherit all the members of the superinterface in the manner similar to classes.
- This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

```
interface Printable{
void print();
}
interface Showable extends
Printable{ void show();
}
class TestInterface4 implements
Showable{ public void
print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String
args[]){ TestInterface4 obj = new
TestInterface4(); obj.print();
obj.show();
 }
}
```

Inheritance and Interface Implementation

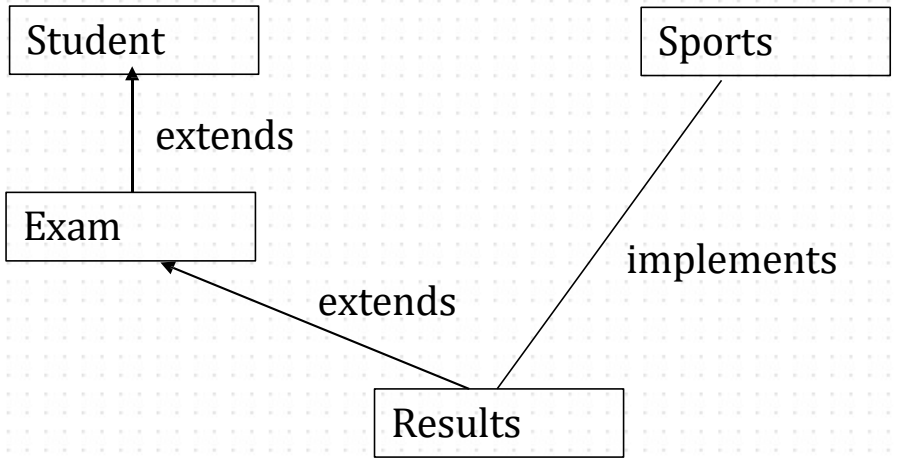
- A general form of interface implementation:

```
class ClassName extends SuperClass implements InterfaceName [, InterfaceName2, ...]
{
    // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces.

Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks.
- Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks.
- A class diagram representing this scenario is as follow:



Software Implementation

```
class Student
{
        // student no and access methods
}
interface Sport
{
        // sports grace marks (say 5 marks) and abstract methods
}
class Exam extends Student
{
        // example marks (test1 and test 2 marks) and access methods
}
class Results extends Exam implements Sport
{
        // implementation of abstract methods of Sport interface
        // other methods to compute total marks = test1+test2+sports_grace_marks;
        // other display or final results access methods
}
```


Packages

A **java package** is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Reusability

How To Create Package

Open your text editor and create a new file that will contain the Simple class in the mypack. Type in the following Java statements:

package nameofthepackage;

Ex: package pack1;

As the first statement in java file.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String
   args[]){ System.out.println("Welcome to
   package");
  }
}
```


How to compile & Run

Compile:

javac -d . File.java

Run:

java packagename.file

```
G:\Sukanya Griet\java\labprogs\packages>javac ./pack/A.java

G:\Sukanya Griet\java\labprogs\packages>javac ./mypack/Simple.java

G:\Sukanya Griet\java\labprogs\packages>javac ./mypack/B.java

G:\Sukanya Griet\java\labprogs\packages>java mypack.B
Hello

G:\Sukanya Griet\java\labprogs\packages>
```


How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;

In above examples Selected or all classes in packages can be imported:

3. fully qualified name.

   Using fully qualified class name

   java.lang.Math.sqrt(x);

   Import package and use class name directly.

   import java.lang.Math Math.sqrt(x);

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible.

The import keyword is used to make the classes and interface of another package accessible to the current package.

```
import pack.*;
class B{
  public static void main(String
   args[]){ A obj = new A();
   obj.msg();
  }
}
```


Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

EX: **import** pack.A;

Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Java provides a large number of classes groped into different packages based on their functionality.

The six foundation Java packages are:

java.lang

Contains classes for primitive types, strings, math functions, threads, and exception

java.util

Contains classes such as vectors, hash tables, date etc.

java.io

Stream classes for I/ O

java.awt

Classes for implementing GUI – windows, buttons, menus etc.

java.net

Classes for networking

java.applet

Classes for creating and implementing applets

Access Modifiers

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Exception Handling

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

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?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

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. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed.

Common Runtime Errors

Dividing a number by zero.

Accessing an element that is out of bounds of an array.

Trying to store incompatible data elements.

Using negative value as array size.

Trying to convert from string data to a specific data value (e.g., converting string "abc" to integer value).

File errors:

- opening a file in "read mode" that does not exist or no read permission
- Opening a file in "write/update mode" which has "read only" permission.

Corrupting memory: - common with pointers

Any more ….

Without Error Handling - Example 1

```
class NoErrorHandling{
        public static void main(String[] args){
                int a,b;
                a = 7;
                b = 0;
                System.out.println("Result is " + a/b);

                System.out.println("Program reached this line");

        }
}
```

Program does not reach here

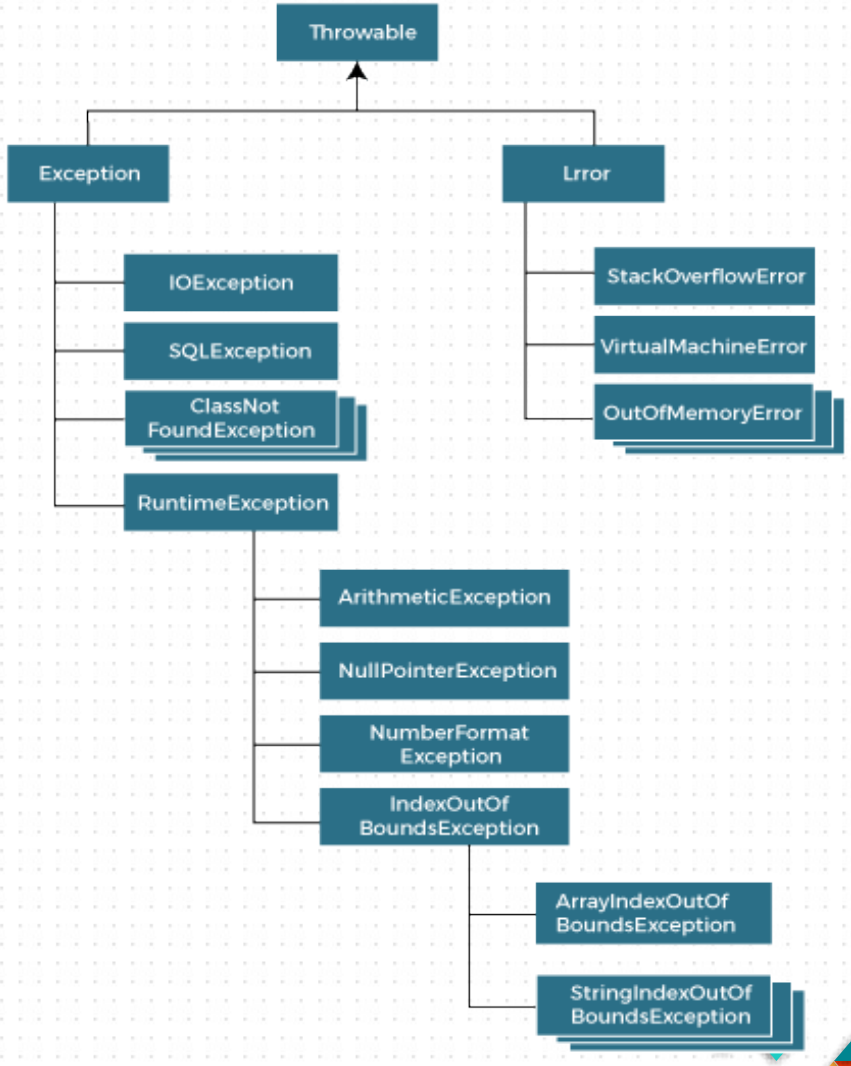
No compilation errors. While running it reports an error and stops without executing further statements:

Output: java.lang.ArithmeticException: / by zero at Error2.main(Error2.java:10)

```
//Without Error Handling – Example 2
class Without
{
        public static void main(String[] args)
        {
                System.out.println("open the files");
                int n=args.length;
                System.out.println("Number of args are"+n);
                int a=45/n;
                System.out.println(a);
                int a1[]={10,20};
                a1[3]=40;
                System.out.println("The value of a1 is"+a[1]);
                System.out.println("close the file");
                }
        }
```

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



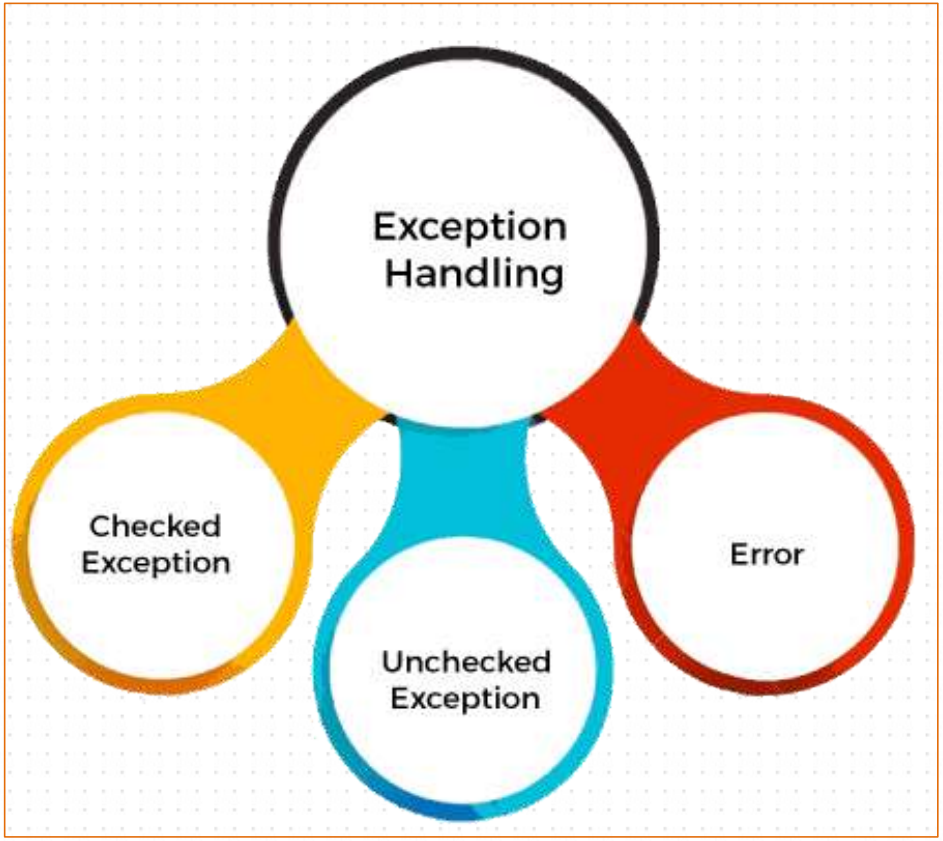
Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

Checked Exception

Unchecked Exception

Error



1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

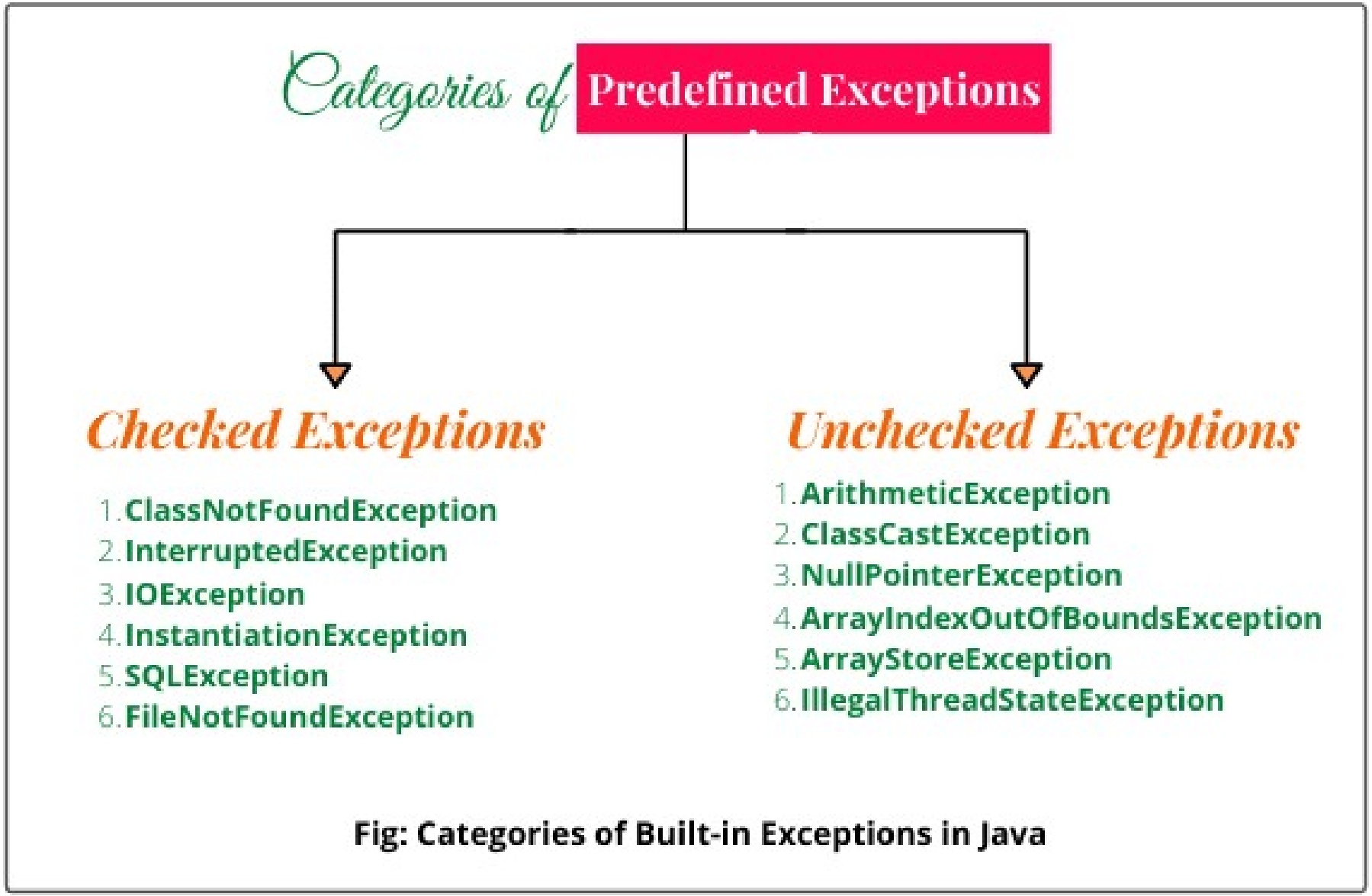


Fig: Categories of Built-in Exceptions in Java

Java Exception Keywords

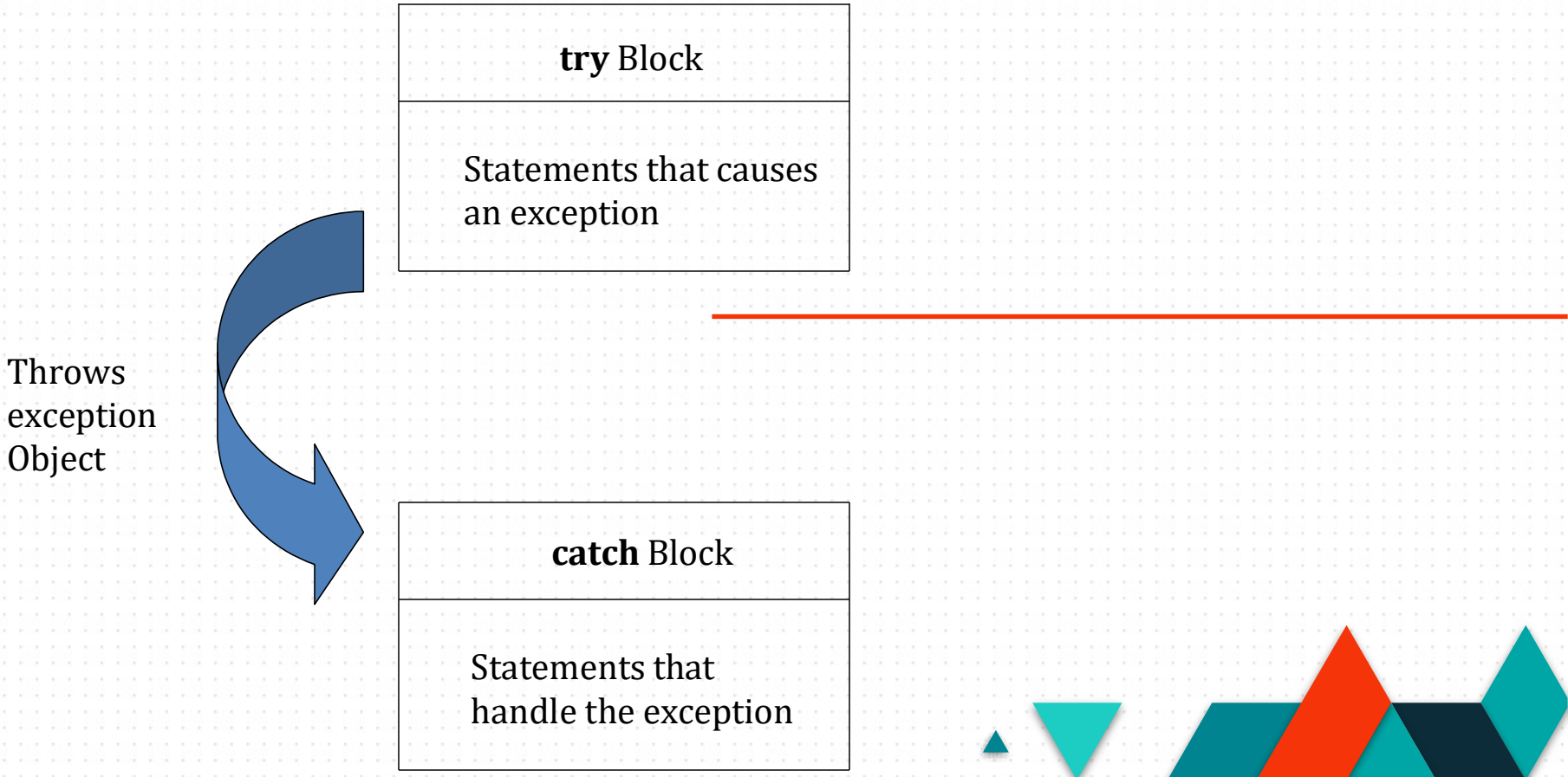
Java provides five keywords that are used to handle the exception.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

```
public class
  JavaExceptionExample{ public static
  void main(String args[]){ try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Exception Handling Mechanism- Java try-catch 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	Syntax of try-finally block

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

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter.

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

```
public class TryCatchExample {

   public static void main(String[] args) {
      try
      {
      int arr[]= {1,3,5,7};
      System.out.println(arr[10]); //may throw exception
      }
         // handling the array exception
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Let's see an example to handle checked exception.

```
import java.io.*;
 public class TryCatchExample10
    {  public static void main(String[]
    args) {
FileReader fr;
      try {
         fr = new FileReader("jtp1.txt"); //may throw exception
         System.out.println("saved");
      }
// providing the checked exception handler
 catch (FileNotFoundException e) {
          System.out.println(e);
      }
   System.out.println("
   }
}
```

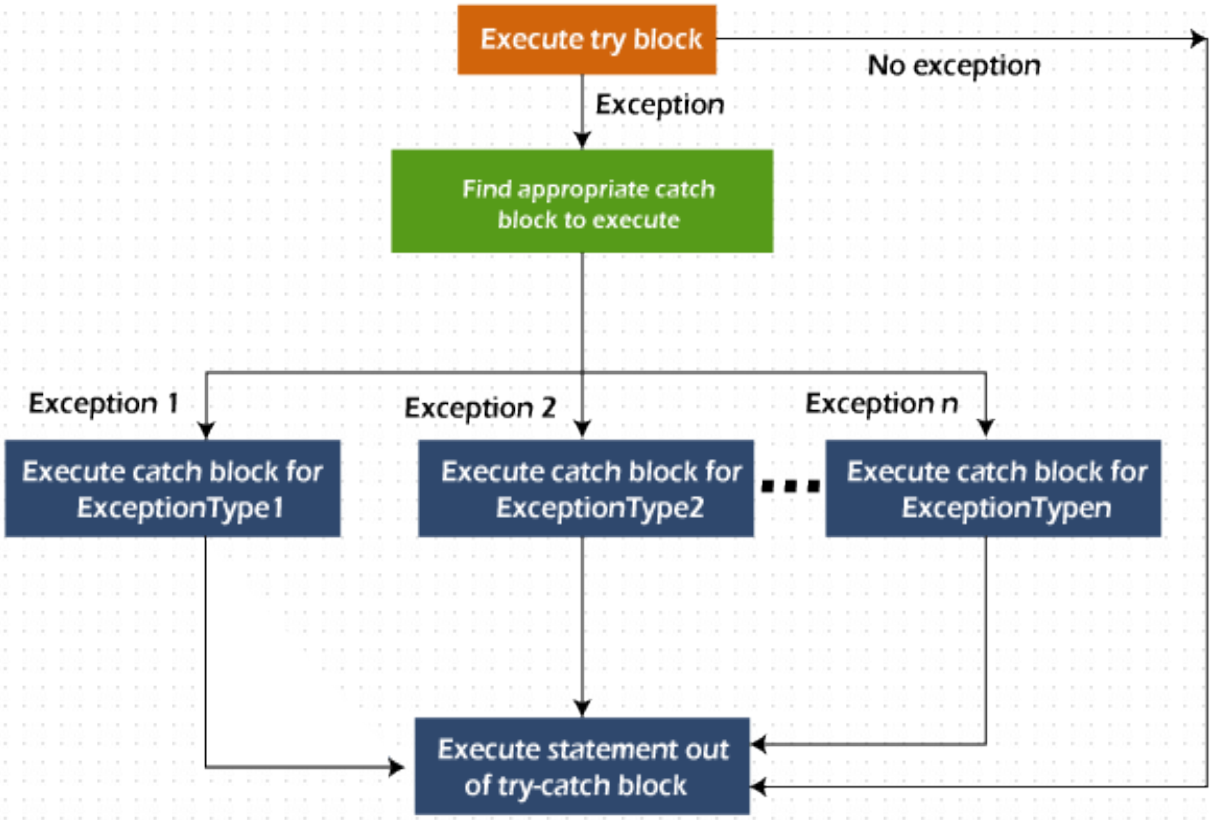
```
java.io.FileNotFoundException: jtp1.txt (The system canno
t find the file specified)
File saved successfully
```

Java Catch Multiple Exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler.

At a time only one exception occurs and at a time only one catch block is executed.



```
public class MultipleCatchBlock1 {

   public static void main(String[] args) {


        try{
            int a[]=new int[5];
            a[4]=30/0;
    // System.out.println(a[10]);
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
```

```
Arithmetic Exception occurs
rest of the code
```

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block.
For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

```
try
{
    statement 1;
    statement 2;
    try    //try catch block within another try block
    {
        statement 3;
        statement 4;
        try    //try catch block within nested try block
        {
            statement 5;
            statement 6;
     }
        catch(Exception e2)
        {    //exception message          }
      }
    catch(Exception e1)
    {    //exception message       }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
```