# JAVA PROGRAMMING

## UNIT - V

## Syllabus

- **Packages and Interfaces:** Packages, Access Protection, Importing Packages, Interfaces.
- **Exception Handling:** Exception-Handling Fundamentals, Exception Types, Uncaught Exceptions, using try and catch, Multiple catch Clauses, throw, throws, finally, Java’s Built-in Exceptions.

## Packages

- Packages are containers for classes, interfaces and sub packages.
- Ie. Is used to group related classes and interfaces.
- Packages are used for avoiding name conflicts, to control access, to make searching/locating and usage of classes, interfaces, and annotations easier, etc.

# Defining a Package

- To create a package: include a package command as the first statement in a Java source file.

  **package NameOfPackage;**

- Example - creates a package called **MyPack.**

```
package MyPack;
```

- Java uses file system directories to store packages. For example, the package **MyPack** must be stored in a directory called **MyPack**. the directory name must match the package name exactly.

## Example

```
//save by A.java
package MyPack;
public class A {
    int a;
    public void dispa(int x) {
    a=x;
    System.out.println("MyPack a = "+ a);
   }
 }
```

## How to access package from another package?

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

1. **import packageName.*;**
2. **import packageName.ClassName;**
3. **fully qualified name.**

## 1) Using packagename.*

- use **import PackageName.*;**
- Hence all the classes and interfaces of this package will be accessible but not sub packages.

## Example

```
import MyPack.*;
class B{
        public static void main(String args[]){
        A Obj = new A();
        Obj.dispa(5);
       }
}
```

## 2) PackageName.ClassName : only named class of this package will be accessible.

```
import MyPack.A;
  class B{
  public static void main(String args[]){
   A Obj = new A();
   Obj.dispa(5);
  }
}
```

## 3) Using fully qualified name

- There is no need to import.
- But 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

## Example

```
class B {
  public static void main(String args[]){
   MyPack.A Obj = new MyPack.A();
   Obj.dispa();
  }
}
```

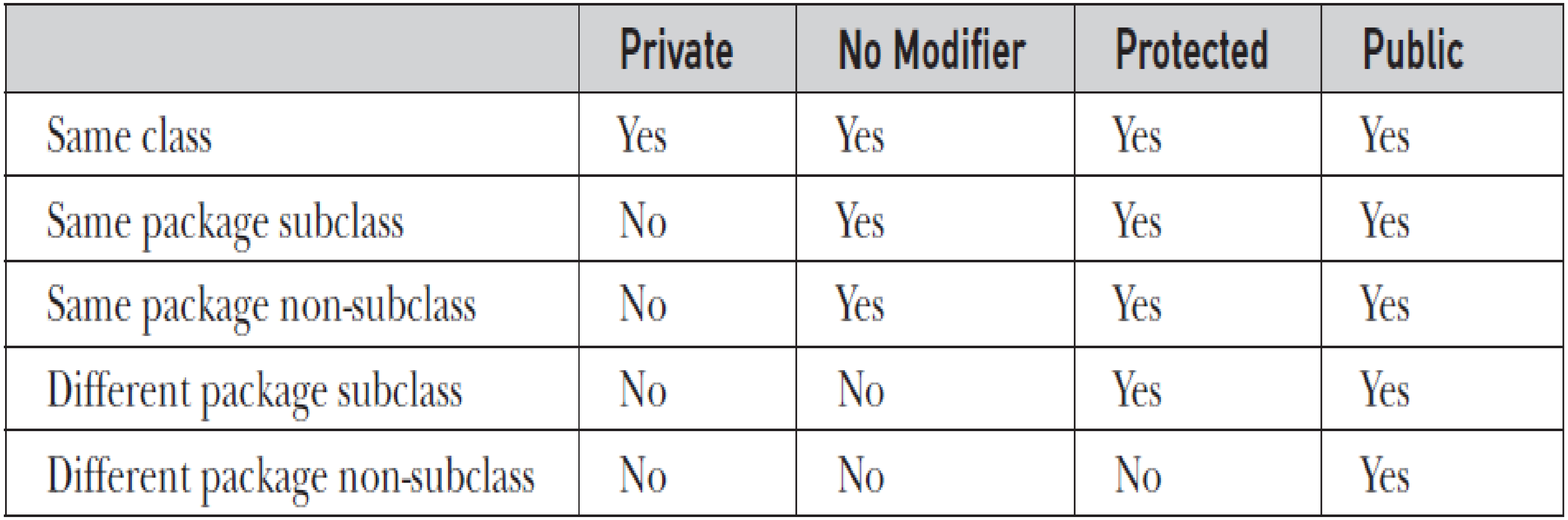
## Access Protection

- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access modifiers, **private, public, and protected**.

## Access Protection

- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

## Access Protection



| | 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 |

- The public members can be accessed everywhere.
- The private members can be accessed only inside the same class.
- The protected members are accessible to every child class (same package or other packages).
- The default members are accessible within the same package but not outside the package.

## Interfaces

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

```
interface Animal {
            public void animalSound();
            public void run();
}
```

- Using the keyword **interface**, you can fully abstract a class’ interface from its implementation.

## Defining an Interface

- The general form of an interface is:

```
interface InterfaceName {
            return-type method-name1(parameter-list);
            return-type method-name2(parameter-list);
            type final-varname1 = value;
            type final-varname2 = value;
            //...
}
```

## …

- Using **interface,** you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes
- All the variables present in the interface are by default "**public static** and **final**".
- Methods in interfaces are declared without any body.
- "**implements**" keyword is used for implementing interfaces to the class.
- Interfaces are designed to support dynamic method resolution at run time.

## Example

```
interface A{
        void method1();
}
class B implements A{
        public void method1(){
                System.out.println("Method 1");
    }
}
class MyClass{
        public static void main(String[] args) {
        B obj = new B();
        obj.method1();
    }
}
```

## Extending Interfaces

- One interface can inherit another by use of keyword extends.
- When a class implements an interface that inherit another interface.

## Example

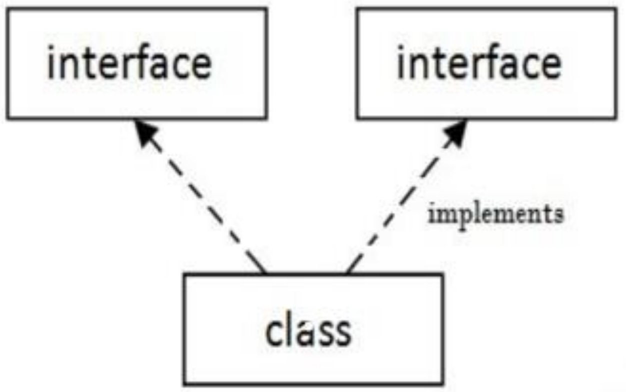
```
interface A{
      void method1();
      void method2();
}
interface B extends A{    // B now includes method1 and method2
       void method3();
}
                                         //the class must implement all method of A and B.
class C implements B{
      public void method1(){
                    System.out.println("Method 1");
    }
      public void method2(){
                    System.out.println("Method 2");
    }
      public void method3(){
                    System.out.println("Method 3");
    }
}
```

## Example

```
class MyClass {
                public static void main(String args[]){
                C obj = new C();
                obj.method1();
                obj.method2();
                obj.method3();
        }
    }
```

# Multiple inheritance in Java by interface

- If a class implements multiple interfaces, it is known as multiple inheritance.



## Multiple inheritance in Java by interface - Example

```
interface A{
                    void method1();    }
interface B{
                    void method2();    }
class C implements A, B{
                    public void method1(){    System.out.println("Method 1");
                              }
                    public void method2(){    System.out.println("Method 2");
                              }

            }
class MyClass {
                    public static void main(String args[]){
                    C obj = new C();
                    obj.method1();
                    obj.method2();
           }
      }
```

## Default Method in Interface

- Can have method body in interface.
- But need to make it, default method.

```
interface A{
            void method1();
            default void method2(){    System.out.println("Method 2");
                }
        }
class C implements A{
        public void method1(){    System.out.println("Method 1");
                }
        }
class MyClass {
                public static void main(String args[]){
                A obj = new C();
                obj.method1();
                obj.method2();
        }
    }
```

## **Static Method in Interface** - we can have static method in interface.

```
interface A{
            void method1();
            static void method2(){    System.out.println("Method 2");
                }
        }
class C implements A{
        public void method1(){    System.out.println("Method 1");
                }
        }
class MyClass {
                public static void main(String args[]){
                A obj = new C();
                obj.method1();
                A.method2();
        }
    }
```

## Differences Between – Class and Interface

| Class | Interface |
|---|---|
| The keyword used to create a class is "class" | The keyword used to create an interface is "interface" |
| A class can be instantiated i.e, objects of a class can be created. | An Interface cannot be instantiated i.e, objects cannot be created. |
| Classes does not support multiple inheritance. | Interface supports multiple inheritance. |
| It can be inherit another class. | It cannot inherit a class. |
| It can be inherited by another class using the keyword 'extends'. | It can be inherited by a class by using the keyword 'implements' and it can be inherited by an interface using the keyword 'extends'. |
| It can contain constructors. | It cannot contain constructors. |
| It cannot contain abstract methods. | It contains abstract methods only. |
| Variables and methods in a class can be declared using any access specifier(public, private, default, protected) | All variables and methods in a interface are declared as public. |
| Variables in a class can be static, final or neither. | All variables are static and final. |

## Exception Handling - Introduction

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a run-time error.

## Exception Handling - Fundamentals

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is caught and processed.

## …

- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.

## …

- In Java exception handled by Five keywords: **try, catch, throw, throws, and finally.**
- Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.
- Your code can catch this exception (using catch) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause.
- Any code that absolutely must be executed after a try block completes is put in a finally block.

## General form of an exception-handling block:

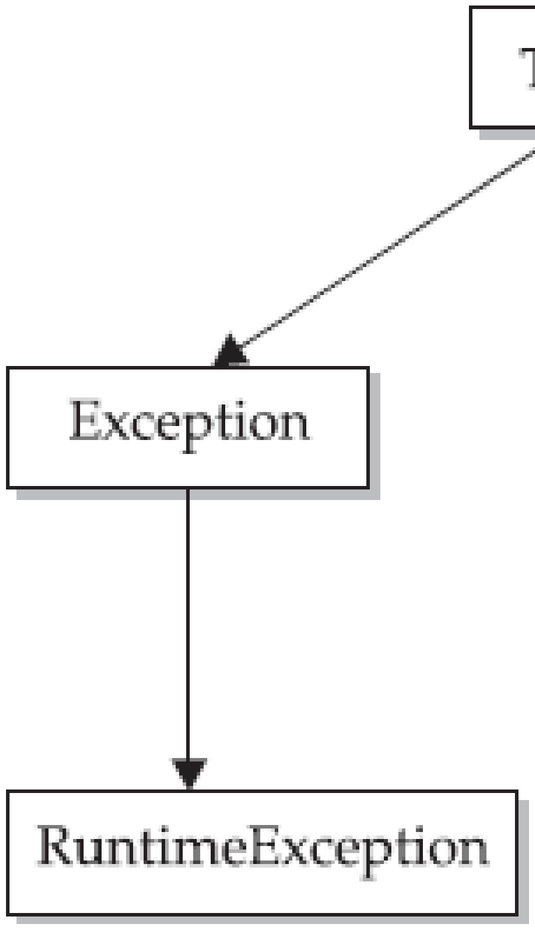
```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

Here, **ExceptionType** is the type of **exception that has occurred**.

## Note:

1. try must be followed by catch
2. try & catch must be perfect pairs.
3. A try can be followed by multiple catches.

# Exception Types



## …

- All exception types are subclasses of the built-in class **Throwable**.
- Exception is a subclass of Throwable. Exception class is used for catching exceptional conditions arise in the user programs. This is also the class that you will subclass to create your own custom exception types.
- There is an important subclass of **Exception**, called **RuntimeException**.

## …

- **Error** is another subclass of Throwable. These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise.

## Three Categories of Exceptions

- **1. Checked exceptions:** A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
- For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

## Three Categories of Exceptions

- **2. Un checked exceptions:** An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an

  ArrayIndexOutOfBoundsExceptionexception occurs.

## Three Categories of Exceptions

- **3. Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## Uncaught Exceptions

```
class Exc0 {
        public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

## …

- Exception generated by the above program: **java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)**
- Notice that here Exc0 is a class name; the method name main(); and the filename is Exc0.java; and the line number, 4, are all included in the simple stack trace.
- Also, notice that the type of exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.

## Using try and catch

Benefits of using user defined Exception handler:

1. It allows you to fix the error.
2. It prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error

## . . .

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;
          try {
              d = 0;
              a = 42 / d;
              System.out.println("This will not be printed.");
             }
           catch (ArithmeticException e) {
                      System.out.println("Division by zero.");
              }
          System.out.println("After catch statement.");
    }
 }
```

Output:  Division by zero.

After catch statement.

## …

- Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.
- Put differently, catch is not "called," so execution never "returns" to the try block from a catch.
- Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.

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

## Problem without exception handling

```
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**

- As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).
- There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

## Solution by exception handling

```
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
```

- As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

## Example 3

```
public class TryCatchExample3 {
     public static void main(String[] args) {
       try
       {
       int data=50/0; //may throw exception
                     // if exception occurs, the remaining statement will not exceute
       System.out.println("rest of the code");
       }
            // handling the exception
       catch(ArithmeticException e)
       {
          System.out.println(e);
       }
       }
}
```

- Output:   java.lang.ArithmeticException: / by zero
- Here, we can see that if an exception occurs in the try block, the rest of the try block code will not execute.

## Example 4 - example to print a custom message on exception.

```
public class TryCatchExample5 {
     public static void main(String[] args) {
       try
       {
       int data=50/0; //may throw exception
       }
            // handling the exception
       catch(Exception e)
       {
                 // displaying the custom message
           System.out.println("Can't divided by zero");
       }
    }
   }
```

Output: Can't divided by zero

## Example to resolve the exception in a catch block.

```
public class TryCatchExample6 {
      public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

Output: **25**

## Example - example to handle unchecked exception.

```
public class TryCatchExample9 {
   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");
   }
}
```

Output : java.lang.ArrayIndexOutOfBoundsException: 10

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

## Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.
- The following example traps two different exception types:

## Example

```
class MultiCatch {
            public static void main(String args[]) {
              try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                }
             catch(ArithmeticException e) {
                                    System.out.println("Divide by 0: " + e);
              }
            catch(ArrayIndexOutOfBoundsException e) {
                                    System.out.println("Array index oob: " + e);
            }
            System.out.println("After try/catch blocks.");
      }
}
```

# Throw - used to throw an exception explicitly.

- Normally JRE will identify the Exception and throw the Exception.
- However, it is possible for your program to throw an exception explicitly, using the **throw** or “**throws**”.
- General form:
  - **throw ThrowableInstance;**
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.
- We can also define our own set of conditions and throw an exception explicitly using throw keyword.

## Throw..

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.
- There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.
- Example

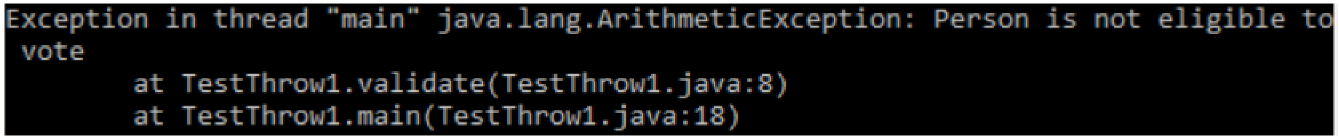
```
 throw new exception_class("error message");
throw new IOException("sorry device error");
```

## Throw..

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement.
- If not, then the next enclosing try statement is inspected, and so on.
- If no matching catch is found, then the default exception handler halts the program and Prints the error message.

## Example 1: Throwing Unchecked Exception

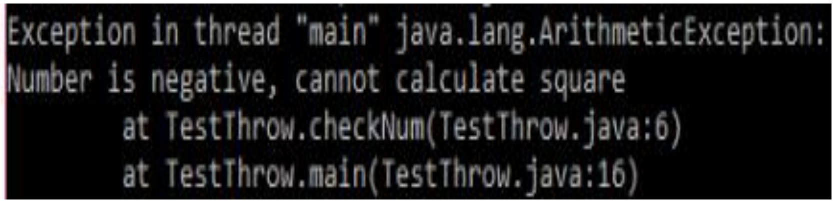
```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
    public static void main(String args[]){
                validate(13);
                 System.out.println("rest of the code...");
          }
}
```



```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

# Java throw Example

```java
public class TestThrow {
   //defining a method
   public static void checkNum(int num) {
      if (num < 1) {
         throw new ArithmeticException("\nNumber is negative, cannot calculate square
");
      }
      else {
         System.out.println("Square of " + num + " is " + (num*num));
      }
   }
   //main method
   public static void main(String[] args) {
         TestThrow obj = new TestThrow();
         obj.checkNum(-3);
         System.out.println("Rest of the code..'
   }
}
```



```
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

## A sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
        static void demoproc() {
                try {
                        throw new NullPointerException("demo");
                }
                catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                }
        }
        public static void main(String args[]) {
                try {
                        demoproc();
                }
                catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
                }
        }
}
```

- This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ).
- The demoproc( ) method then sets up another exception handling context and immediately throws a new instance of NullPointerException, which is caught on the next line.
- The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

- The program also illustrates how to create one of Java's standard exception objects.

```
throw new NullPointerException("demo");
```

- Here, new is used to construct an instance of NullPointerException.
- Many of Java's built in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception.

## Throws - is used to declare an exception.

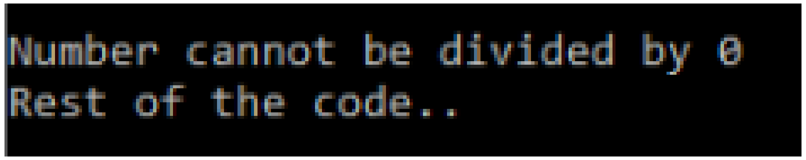
- Exceptions that a method can throw must be declared in the throws clause.
- The general form of a throws clause is:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

# throws Example

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
                System.out.println(obj.divideNum(45, 0));
            }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }
       System.out.println("Rest of the code..");
    }
}
```



```
Number cannot be divided by 0
Rest of the code..
```

## Throws

```
class ThrowsDemo
{
        static void throwOne() throws IllegalAccessException
        {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                try {
                        throwOne();
                }
                catch (IllegalAccessException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

**Output: inside throwOne**
**caught java.lang.IllegalAccessException: demo**

## throw and throws Example

```
public class TestThrowAndThrows
{
      // which throws ArithmeticException
   static void method() throws ArithmeticException
   {
      System.out.println("Inside the method()");
      throw new ArithmeticException("throwing ArithmeticException");
   }
   public static void main(String args[])
   {
      try   {
               method();
      }
      catch(ArithmeticException e)    {
         System.out.println("caught in main() method");
      }
   }
}
// System.out.println("caught in main() method" + e);
```

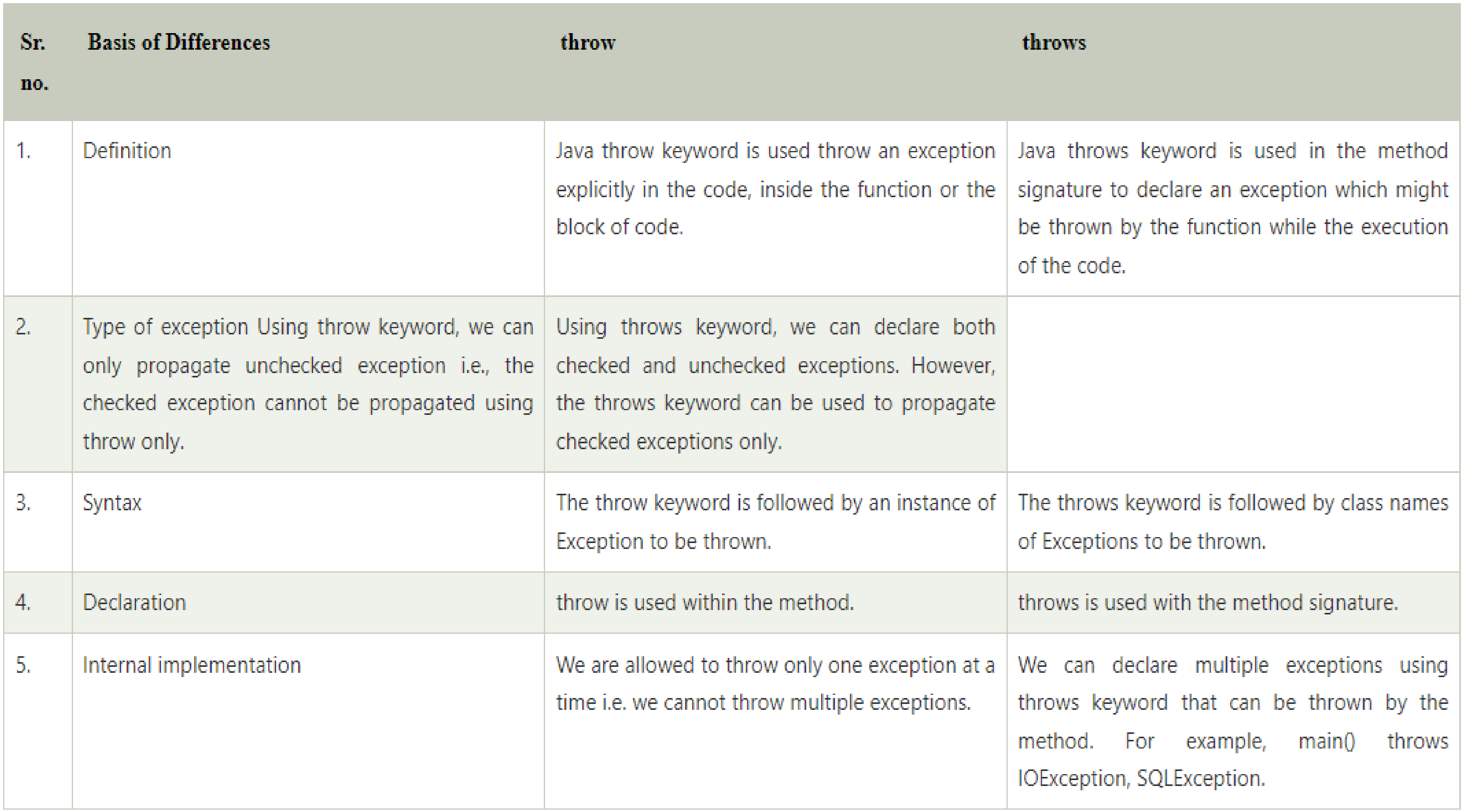
Output:

```
Inside the method()
caught in main() method
```

Output:

```
Inside the method()
caught in main() method
java.lang.ArithmeticException: throwing ArithmeticException
```

# Difference between throw and throws

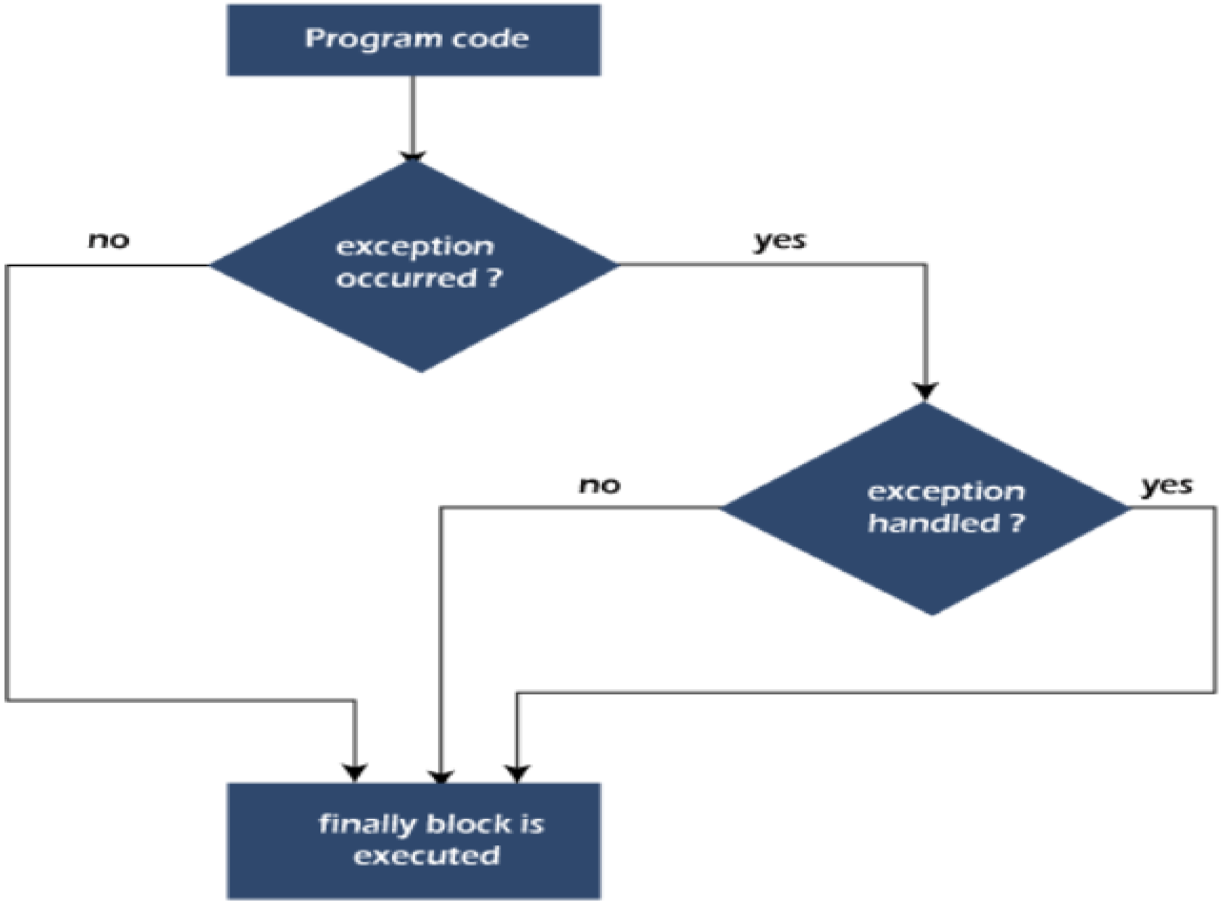


| Sr. no. | Basis of Differences | throw | throws |
|---|---|---|---|
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. | |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

## finally

- **Java finally block** is a block used to execute important code such as closing the opened file, 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.

# Flowchart of finally block



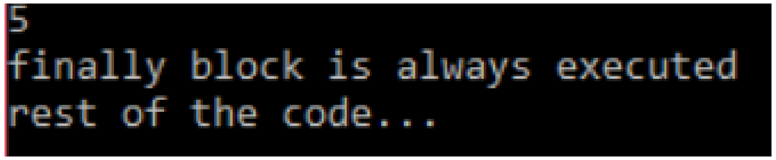
Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

## Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

## Case 1: When an exception does not occur

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
        //below code do not throw any exception
        int data=25/5;
        System.out.println(data);
      }
      //catch won't be executed
      catch(NullPointerException e){
            System.out.println(e);
       }
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}
  System.out.println("rest of the code...");
 }
}
```



```
5
finally block is always executed
rest of the code...
```

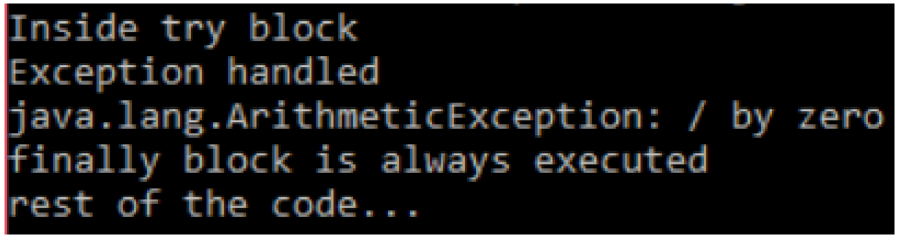
Case 2: When an exception occur but not handled by the catch block - Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```
public class TestFinallyBlock1{
     public static void main(String args[]){
       try {
               System.out.println("Inside the try block");
                //below code throws divide by zero exception
               int data=25/0;
                System.out.println(data);
               }
               //cannot handle Arithmetic type exception  , can only accept Null Pointer type exception
                catch(NullPointerException e){
                    System.out.println(“Error”);
               }
             //executes regardless of exception occurred or not
           finally {
                         System.out.println(“Finally block ");
                  }
         System.out.println("rest of the code...");
     }
   }
```

```
Inside the try block
finally block
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

Case 3: When an exception occurs and is handled by the catch block. The Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

```java
public class TestFinallyBlock2{
   public static void main(String args[]){
    try {
        System.out.println("Inside try block");
        //below code throws divide by zero exception
         int data=25/0;
        System.out.println(data);
      }
    //handles the Arithmetic Exception / Divide by zero exception
   catch(ArithmeticException e){
    System.out.println("Exception handled");
    System.out.println(e);
   }
    //executes regardless of exception occured or not
   finally {
         System.out.println("finally block is always executed");
   }
    System.out.println("rest of the code...");
  }
}
```



```
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

## finally

```
// Demonstrate finally.
class FinallyDemo {
        // Throw an exception out of the method.
        static void procA() {
                try {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                }
                finally {
                        System.out.println("procA's finally");
                }
        }
        // Return from within a try block.
        static void procB() {
                try {
                        System.out.println("inside procB");
                        return;
                }
                finally {
                        System.out.println("procB's finally");
                }
        }
```

# finally

```
        // Execute a try block normally.
        static void procC() {
                try {
                        System.out.println("inside procC");
                }
                finally {
                        System.out.println("procC's finally");
                }
        }

        public static void main(String args[]) {
                try {
                        procA();
                }
                catch (Exception e) {
                        System.out.println("Exception caught");
                }
                procB();
                procC();
        }
}
```

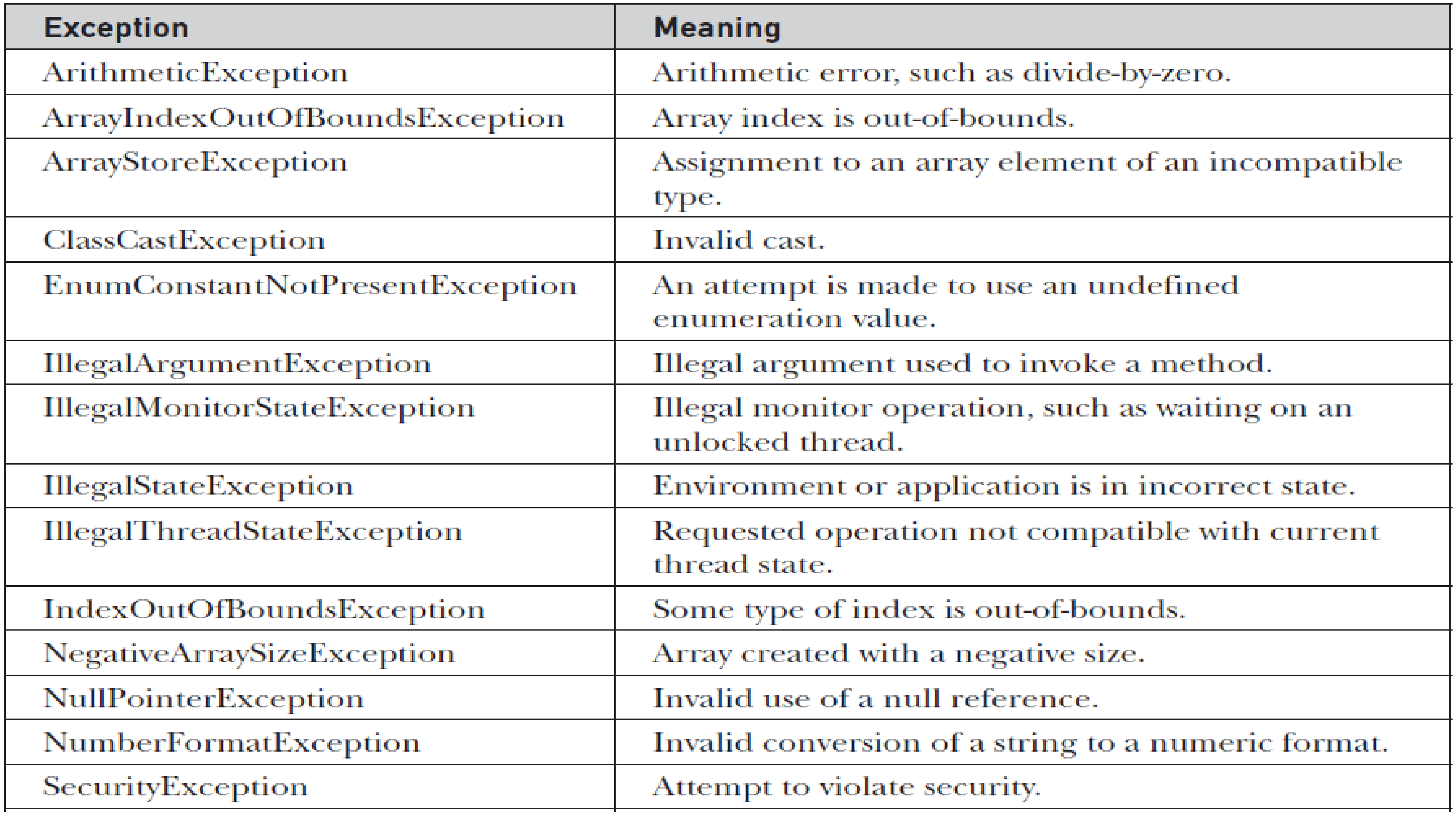
- In the above example, procA( ) prematurely breaks out of the try by throwing an exception.
- The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement.
- The finally clause is executed before procB( ) returns.
- In procC( ), the try statement executes normally, without error.
- However, the finally block is still executed.
- REMEMBER If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.
- Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

# Java's Built-in Exceptions

- Java defines several exception classes inside the standard package **java.lang**.
- Unchecked exception or RuntimeException need not be included in the method's **throws** list. Bcz Compiler does not check to see if a method handles or throws these exceptions.
- Checked exception must be included in the method's **throws** list.

## Unchecked Runtime Exception Subclasses in java.lang



| Exception | Meaning |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |

## Java’s Checked Exceptions in java.lang

| Exception | Meaning |
|---|---|
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the **Cloneable** interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |
| ReflectiveOperationException | Superclass of reflection-related exceptions. |

## THANK YOU