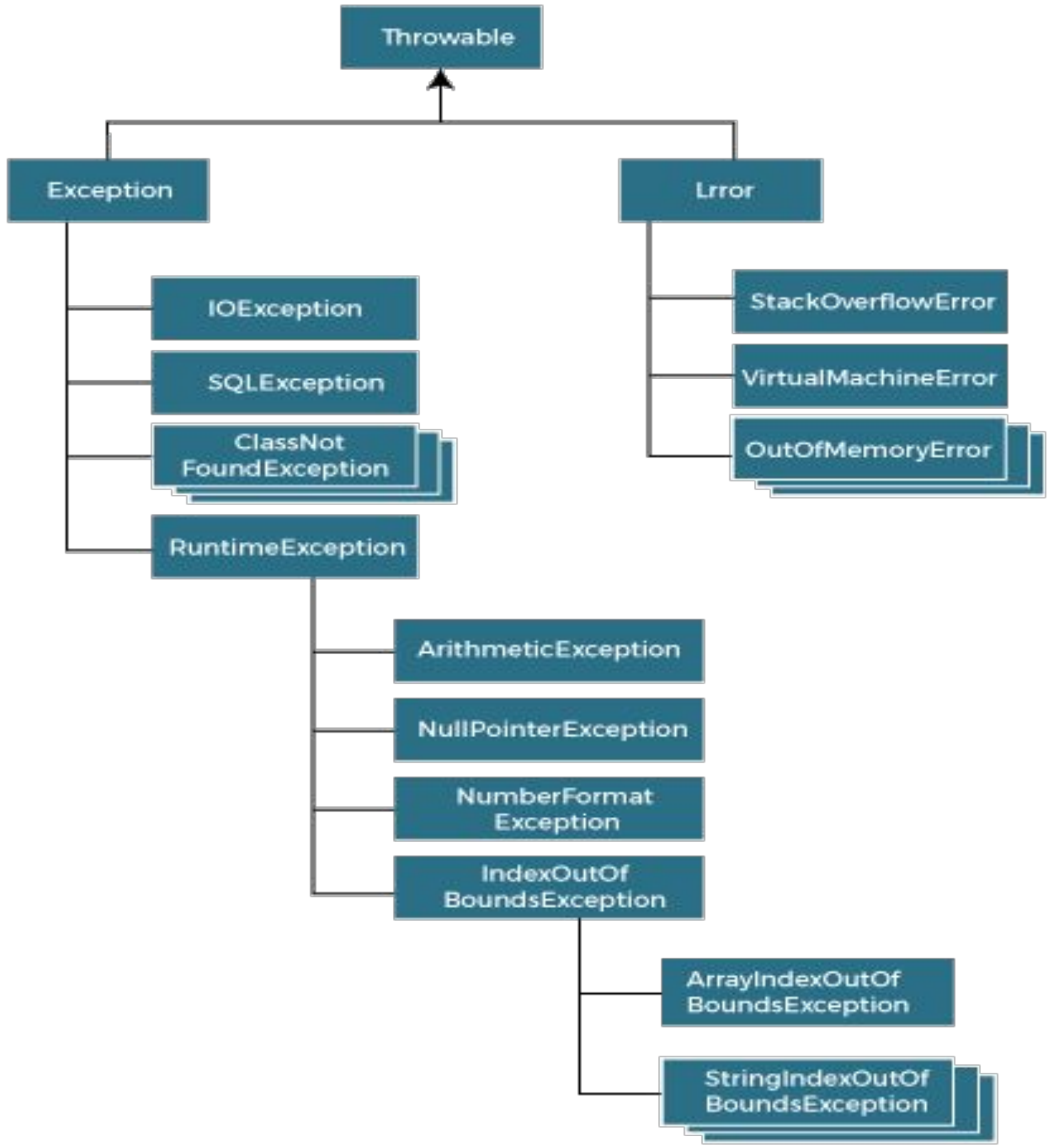
# Unit 5
# Handling Error/Exception

## Java Exceptions

- ❖ Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions.
- ❖ When an Exception occurs the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- ❖ An exception can occur for many different reasons. Following are *some* scenarios where an exception occurs.
  - A user has entered an invalid data.
  - Device failure
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
  - Code error

## 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.
- ❖ Errors represent irrecoverable conditions such as JVM running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
- ❖ Errors are generally beyond the control of programmer and we should not try to handle errors.



Java Exception Hierarchy

## Types of Java Exceptions

❖ Exceptions can be categorized into three types:

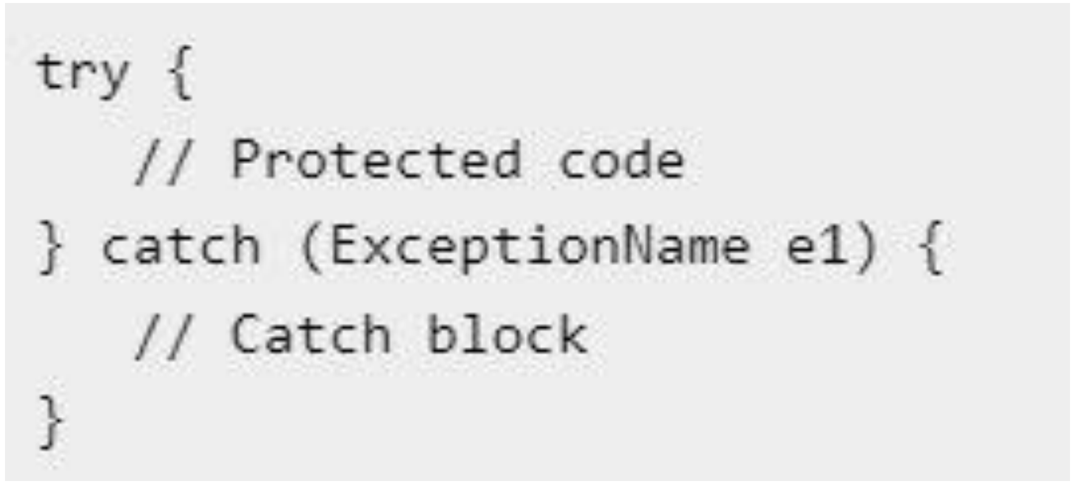
1. **Checked exceptions**
   - A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as **compile time exceptions**.
   - These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.
2. **Unchecked 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.
3. **Errors**
   - These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

## Java Exception handling

- ❖ Exception Handling is a mechanism to handle exceptions such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- ❖ 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.
- ❖ A method catches an exception using a combination of the **try** and **catch** keywords.
- ❖ A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code

## Java Exception handling

❖ Syntax for exception handling:



```
try {
   // Protected code
} catch (ExceptionName e1) {
   // Catch block
}
```

❖ The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it.

❖ Every try block should be immediately followed either by a catch block or finally block.

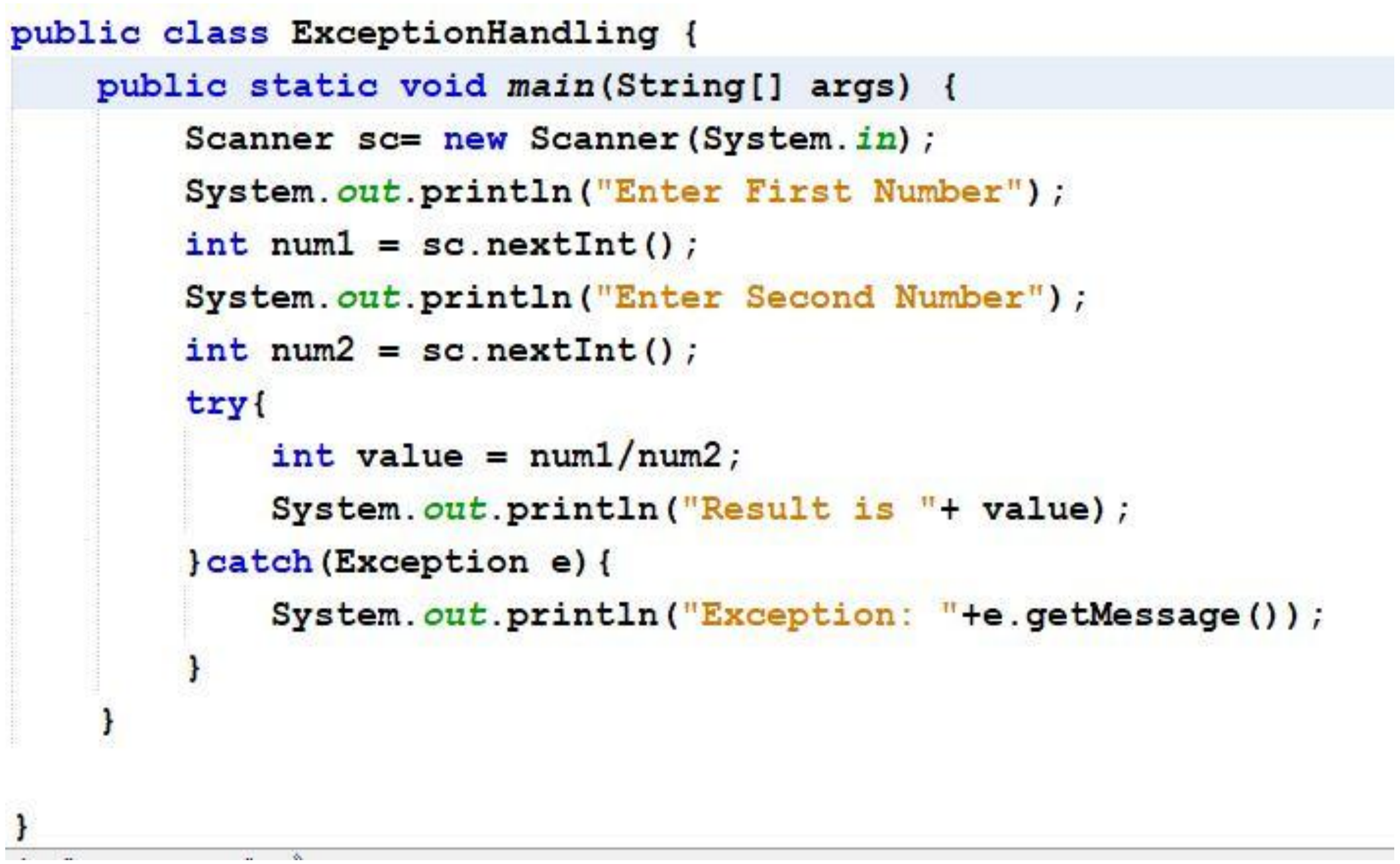
❖ The catch block cannot be used without the try block

## Java Exception handling-Exceptions Methods

❖ Following is the list of important methods available in the Throwable class.

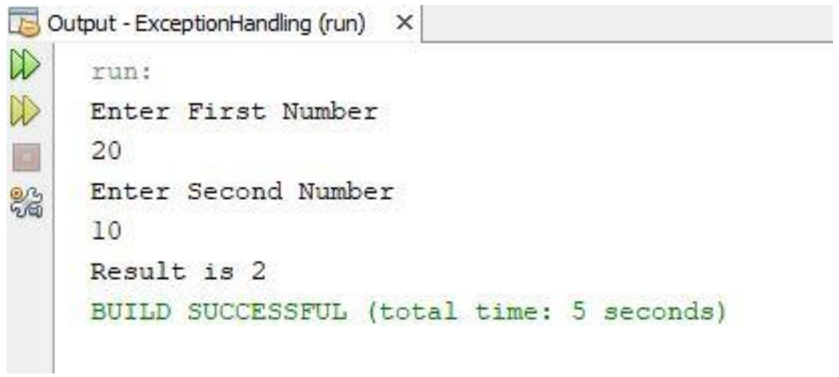
| S.N.. | Method & Description |
|---|---|
| 1 | **public String getMessage()**<br>Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()**<br>Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()**<br>Returns the name of the class concatenated with the result of getMessage(). |
| 4 | **public void printStackTrace()**<br>Prints the result of toString() along with the stack trace to System.err, the error output stream. |

# Java Exception handling-Example



```
public class ExceptionHandling {
    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter First Number");
        int num1 = sc.nextInt();
        System.out.println("Enter Second Number");
        int num2 = sc.nextInt();
        try{
            int value = num1/num2;
            System.out.println("Result is "+ value);
        }catch(Exception e){
            System.out.println("Exception: "+e.getMessage());
        }
    }

}
```

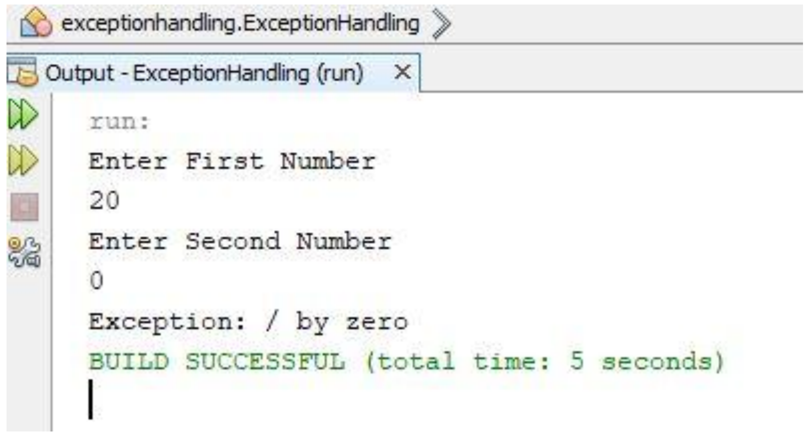
Output when num2 =10



Output - ExceptionHandling (run)

```
run:
Enter First Number
20
Enter Second Number
10
Result is 2
BUILD SUCCESSFUL (total time: 5 seconds)
```

Output when num2 =0



exceptionhandling.ExceptionHandling

Output - ExceptionHandling (run)

```
run:
Enter First Number
20
Enter Second Number
0
Exception: / by zero
BUILD SUCCESSFUL (total time: 5 seconds)
```

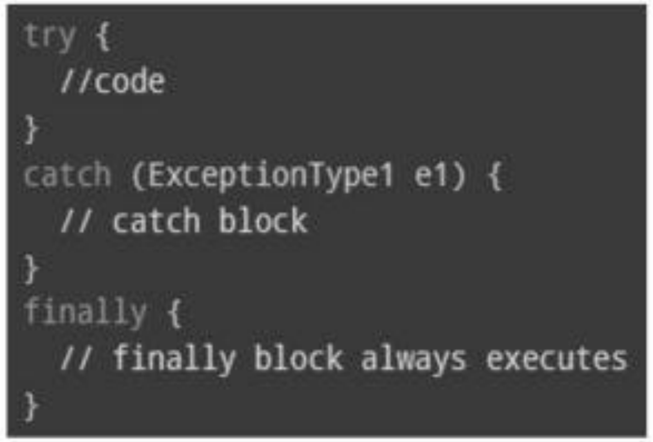
## Java Exception Handling-Finally Block

❖ In Java, the finally block is always executed no matter whether there is an exception or not.

❖ The finally block is optional. And for each try block there can be only one finally block.

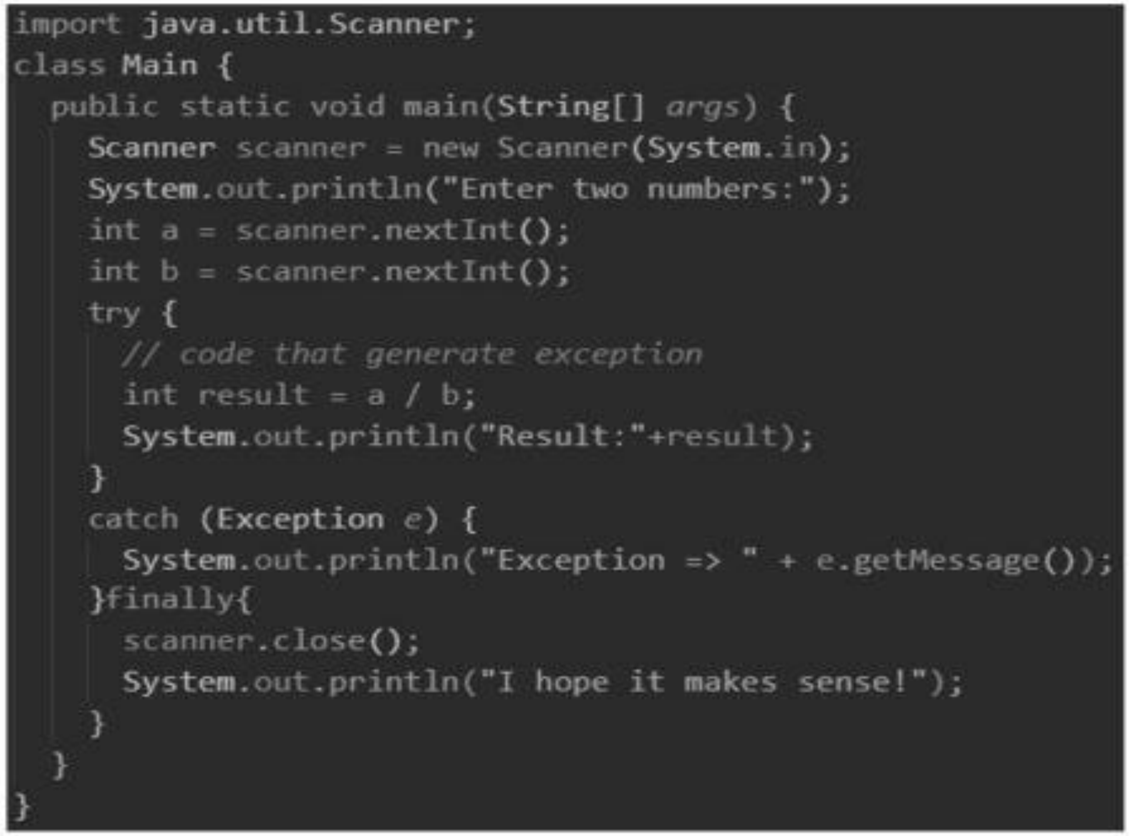
❖ It is a good practice to use the finally block. It is because it can include important cleanup codes like:

- Code that might be accidently escaped by return, continue or break.
- Closing a file or connection

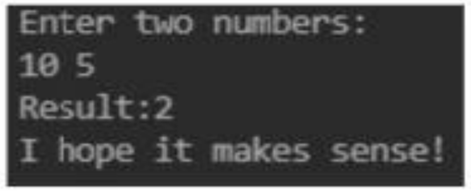


```
try {
  //code
}
catch (ExceptionType1 e1) {
  // catch block
}
finally {
  // finally block always executes
}
```

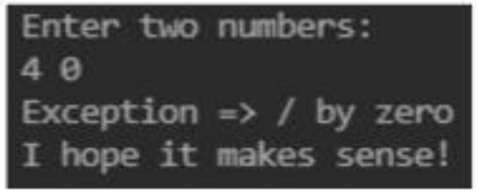
## Java Exception Handling-Finally Block



```java
import java.util.Scanner;
class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter two numbers:");
    int a = scanner.nextInt();
    int b = scanner.nextInt();
    try {
      // code that generate exception
      int result = a / b;
      System.out.println("Result:"+result);
    }
    catch (Exception e) {
      System.out.println("Exception => " + e.getMessage());
    }finally{
      scanner.close();
      System.out.println("I hope it makes sense!");
    }
  }
}
```

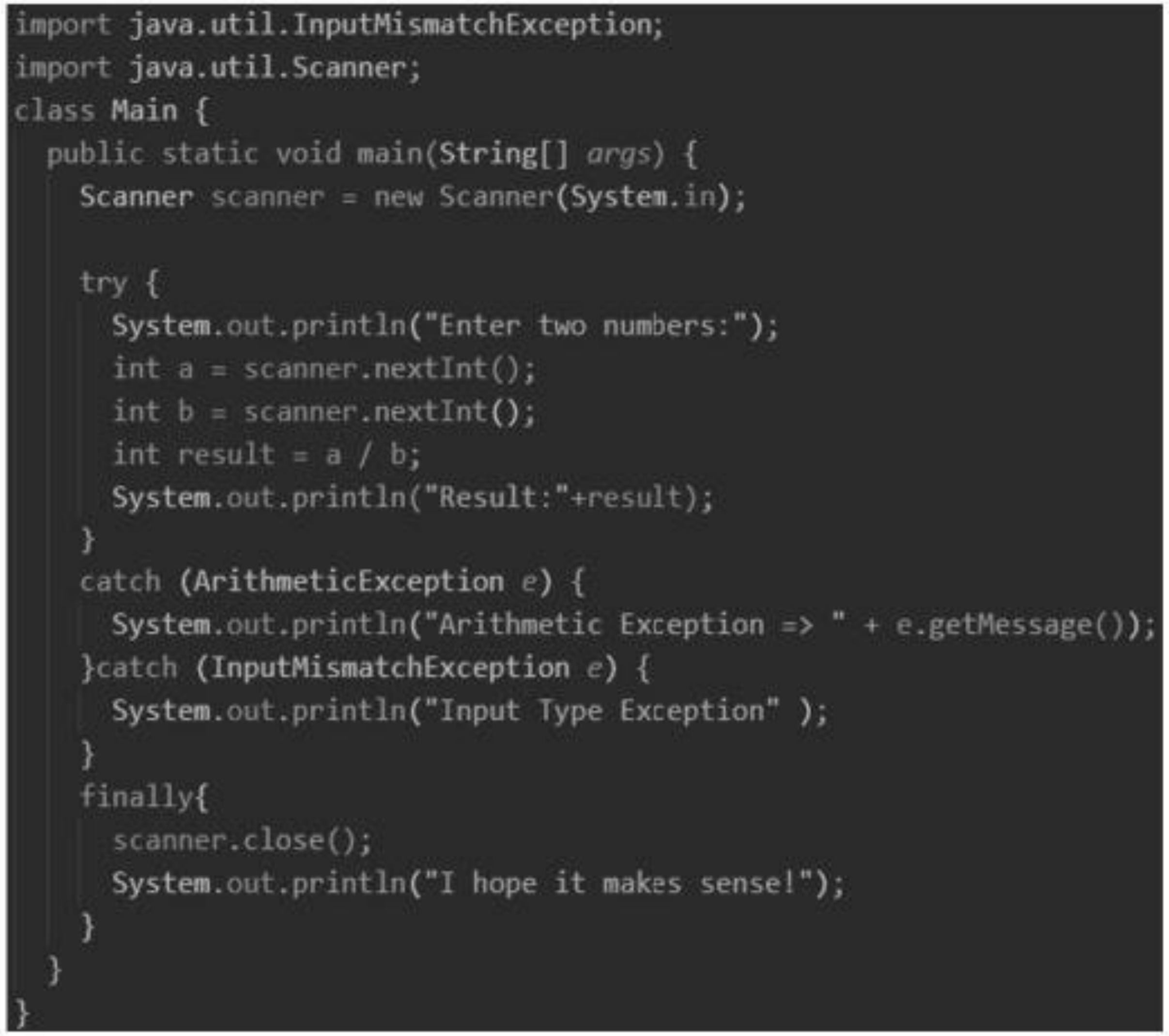


```
Enter two numbers:
10 5
Result:2
I hope it makes sense!
```

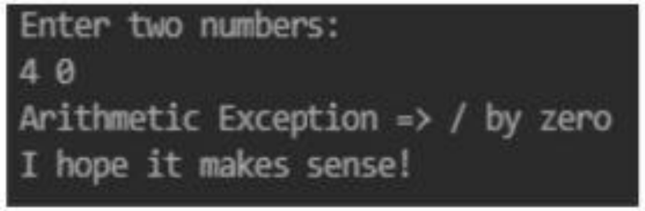


```
Enter two numbers:
4 0
Exception => / by zero
I hope it makes sense!
```

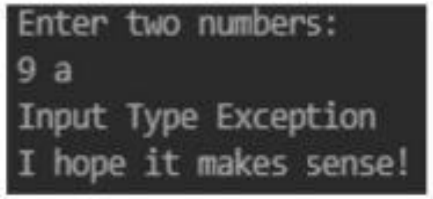
## Java Exception Handling-Multiple Catch Blocks



```
import java.util.InputMismatchException;
import java.util.Scanner;
class Main {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    try {
      System.out.println("Enter two numbers:");
      int a = scanner.nextInt();
      int b = scanner.nextInt();
      int result = a / b;
      System.out.println("Result:"+result);
    }
    catch (ArithmeticException e) {
      System.out.println("Arithmetic Exception => " + e.getMessage());
    }catch (InputMismatchException e) {
      System.out.println("Input Type Exception" );
    }
    finally{
      scanner.close();
      System.out.println("I hope it makes sense!");
    }
  }
}
```



```
Enter two numbers:
4 0
Arithmetic Exception => / by zero
I hope it makes sense!
```

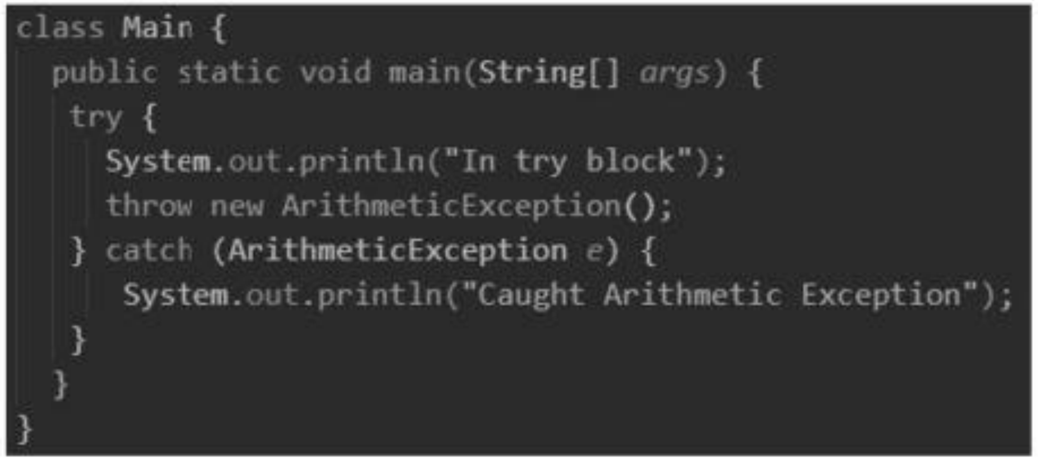


```
Enter two numbers:
9 a
Input Type Exception
I hope it makes sense!
```

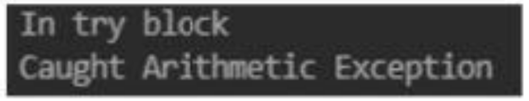
## Java Exception Handling-throw and throws keywords

- ❖ Java throw keyword is used to explicitly throw a single exception
- ❖ When we throw an exception, the flow of the program moves from the try block to the catch block
- ❖ Similarly the throws keyword is used to declare the type of exceptions that might occur within the method.
- ❖ It provides information to the caller of the method about the exception.

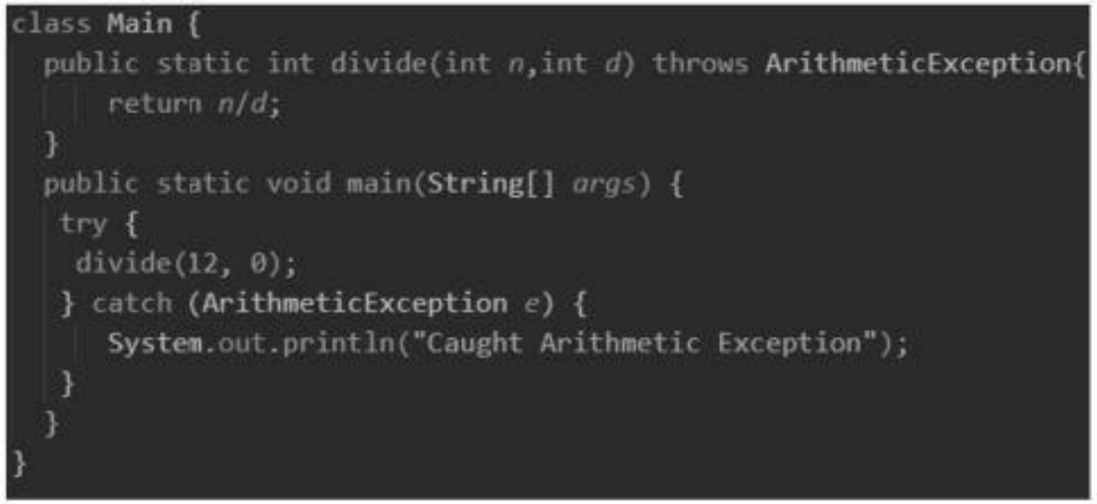
## Java Exception Handling-throw and throws keywords



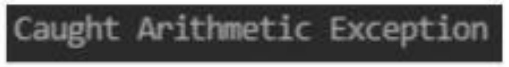
```
class Main {
  public static void main(String[] args) {
    try {
      System.out.println("In try block");
      throw new ArithmeticException();
    } catch (ArithmeticException e) {
      System.out.println("Caught Arithmetic Exception");
    }
  }
}
```



```
In try block
Caught Arithmetic Exception
```



```
class Main {
  public static int divide(int n,int d) throws ArithmeticException{
      return n/d;
  }
  public static void main(String[] args) {
   try {
    divide(12, 0);
   } catch (ArithmeticException e) {
      System.out.println("Caught Arithmetic Exception");
   }
  }
}
```

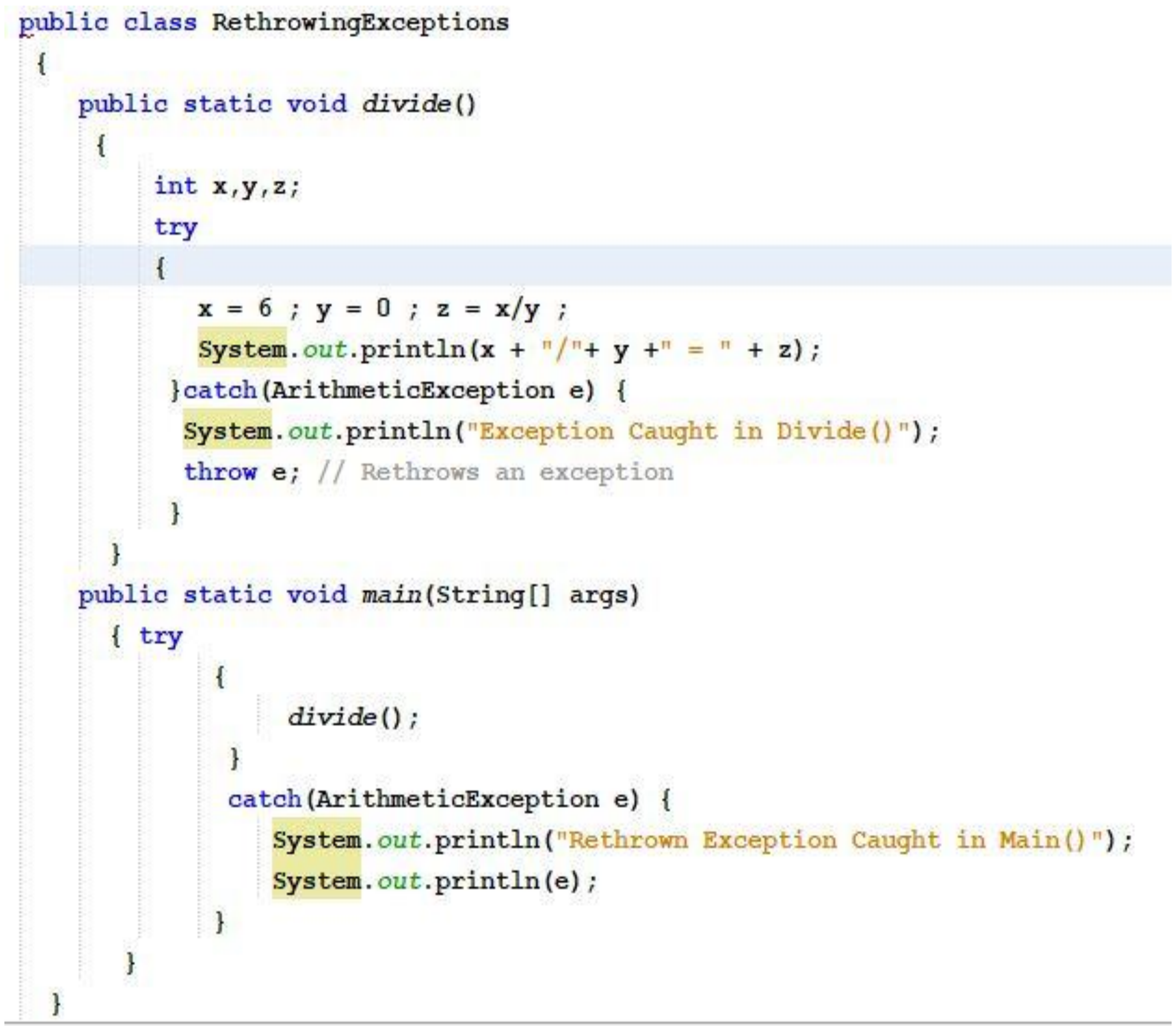


```
Caught Arithmetic Exception
```

## Java Re-throwing Exception

- ❖ Sometimes we may need to rethrow an exception in Java.
- ❖ If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.
- ❖ The rethrow expression causes the **originally thrown object to be rethrown.**
- ❖ Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block.
- ❖ Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred.

# Java Re-throwing Exception-Example



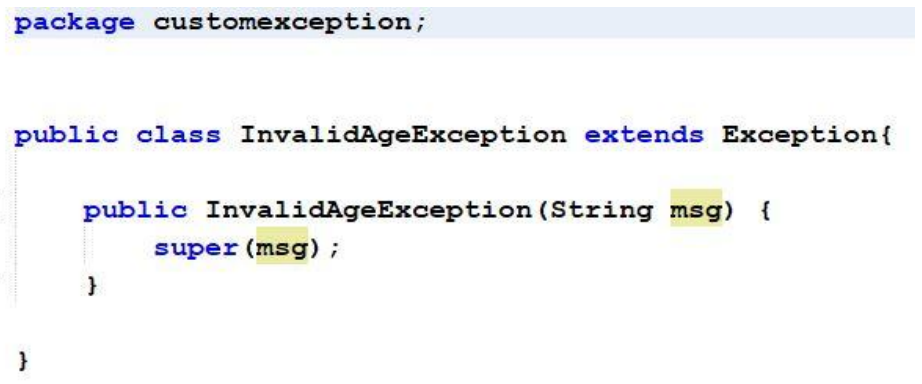
```
public class RethrowingExceptions
 {
    public static void divide()
     {
         int x,y,z;
         try
         {
           x = 6 ; y = 0 ; z = x/y ;
           System.out.println(x + "/"+ y +" = " + z);
          }catch(ArithmeticException e) {
          System.out.println("Exception Caught in Divide()");
          throw e; // Rethrows an exception
          }
      }
    public static void main(String[] args)
      { try
            {
                divide();
             }
            catch(ArithmeticException e) {
               System.out.println("Rethrown Exception Caught in Main()");
               System.out.println(e);
            }
       }
 }
```

## Java Custom/User Defined Exception

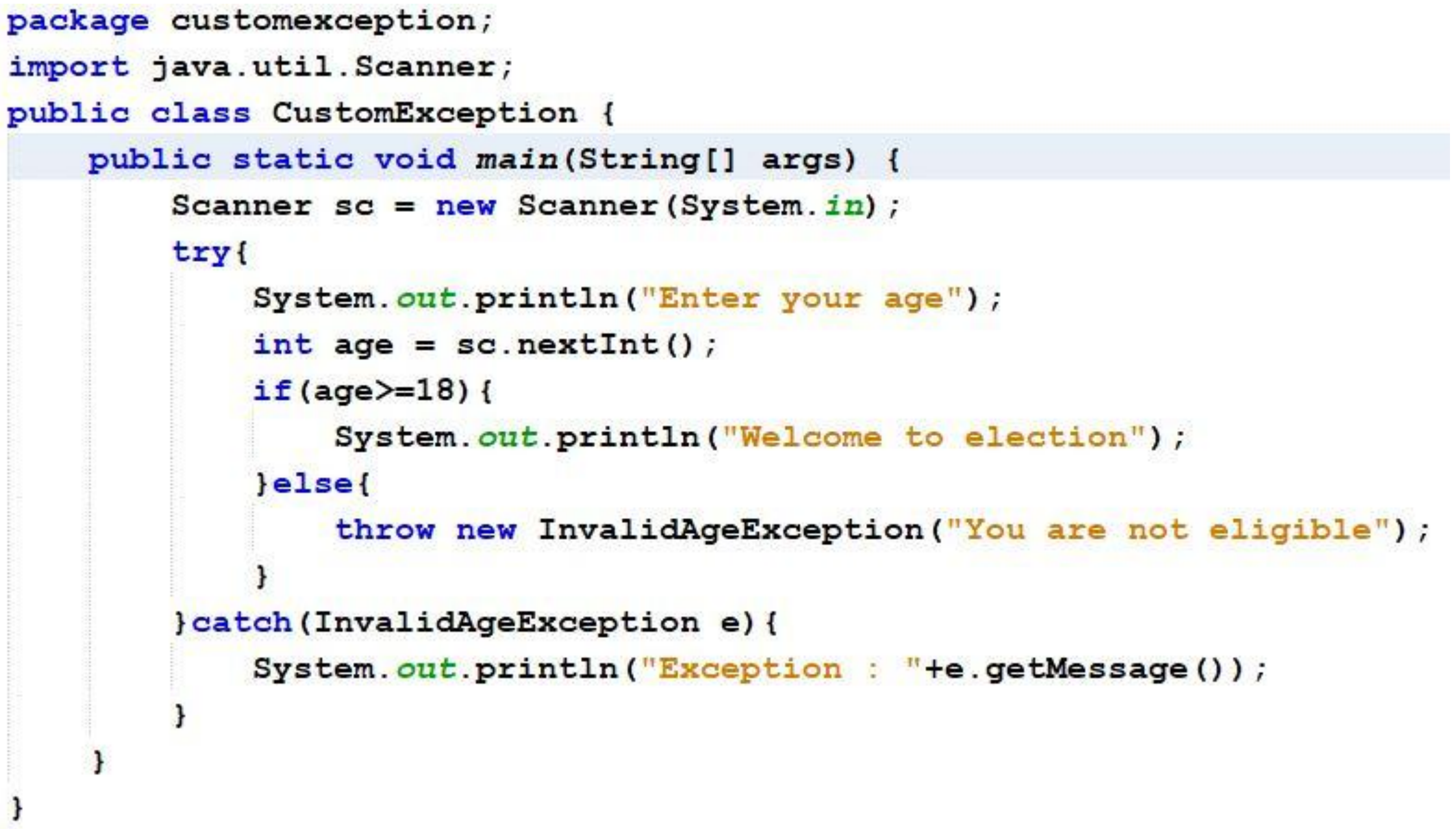
- ❖ In Java, we can create our own exceptions that are derived classes of the Exception class.
- ❖ Creating our own Exception is known as custom exception or user-defined exception.
- ❖ Basically, Java custom exceptions are used to customize the exception according to user need.
- ❖ Following are few of the reasons to use custom exceptions:
  - To catch and provide specific treatment to a subset of existing Java exceptions.
  - Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

## Java Custom/User Defined Exception-Example

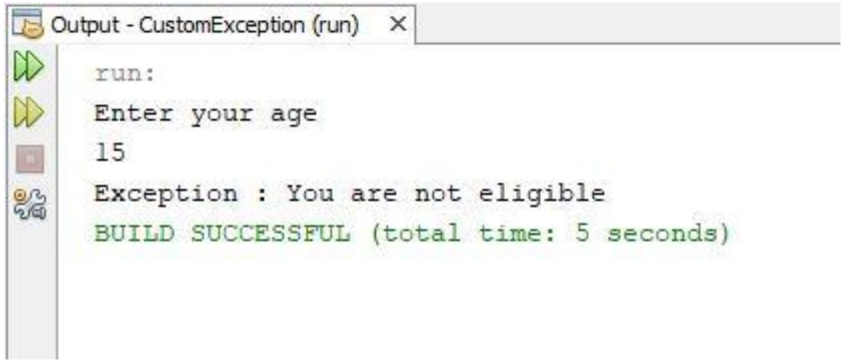
User Defined Exception InvalidAgeException



```
package customexception;


public class InvalidAgeException extends Exception{

    public InvalidAgeException(String msg) {
        super(msg);
    }

}
```



```
package customexception;
import java.util.Scanner;
public class CustomException {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        try{
            System.out.println("Enter your age");
            int age = sc.nextInt();
            if(age>=18){
                System.out.println("Welcome to election");
            }else{
                throw new InvalidAgeException("You are not eligible");
            }
        }catch(InvalidAgeException e){
            System.out.println("Exception : "+e.getMessage());
        }
    }
}
```



```
run:
Enter your age
15
Exception : You are not eligible
BUILD SUCCESSFUL (total time: 5 seconds)
```