Java Exceptions

Agenda:

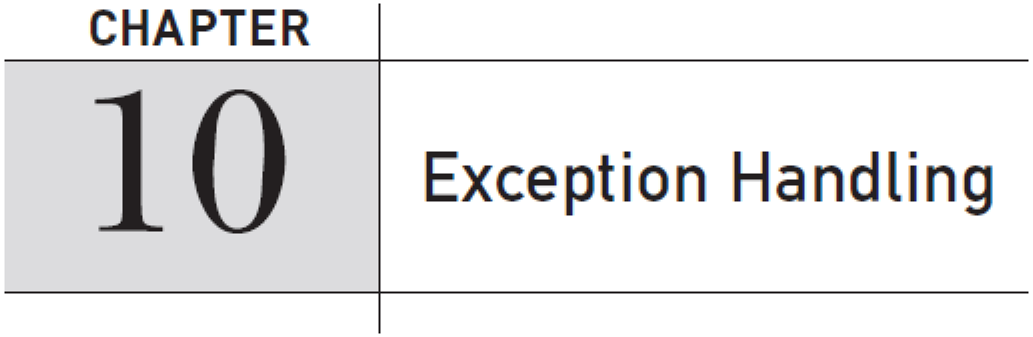
Exception Handling
(try, catch, throw, throws and finally)

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

Exception is an abnormal condition. Exception handling is to maintain the normal flow.

Reference Text : CHAPTER - 10

Exception Handling



CHAPTER

10

Exception Handling

Keywords:

Exception

Exception Handling

Advantages of Exception Handling

Exception Types

try, catch, throw, throws and finally

Multiple exception

Catching all exception

Exception Propagation

throw Vs throws

Md. Mamun Hossain
B. Sc. (Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

Exception and Exception Handling

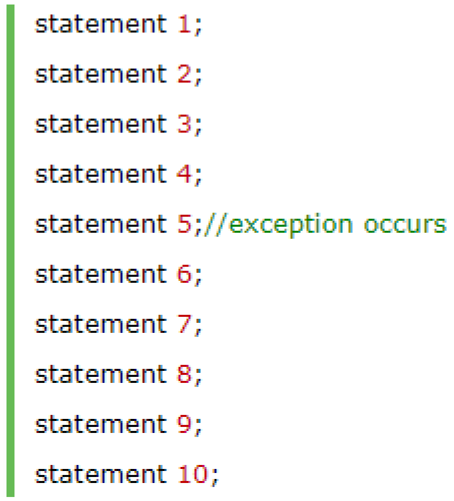
- **Exception**
  - ✓ Exception is an abnormal condition.
  - ✓ In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

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

Md. Mamun Hossain
B. Sc. (Engg.) & M. Sc. (Thesis) in CSE, SUST
Asst. Professor, Dept. of CSE , BAUST

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 use exception handling. Let's take a scenario:

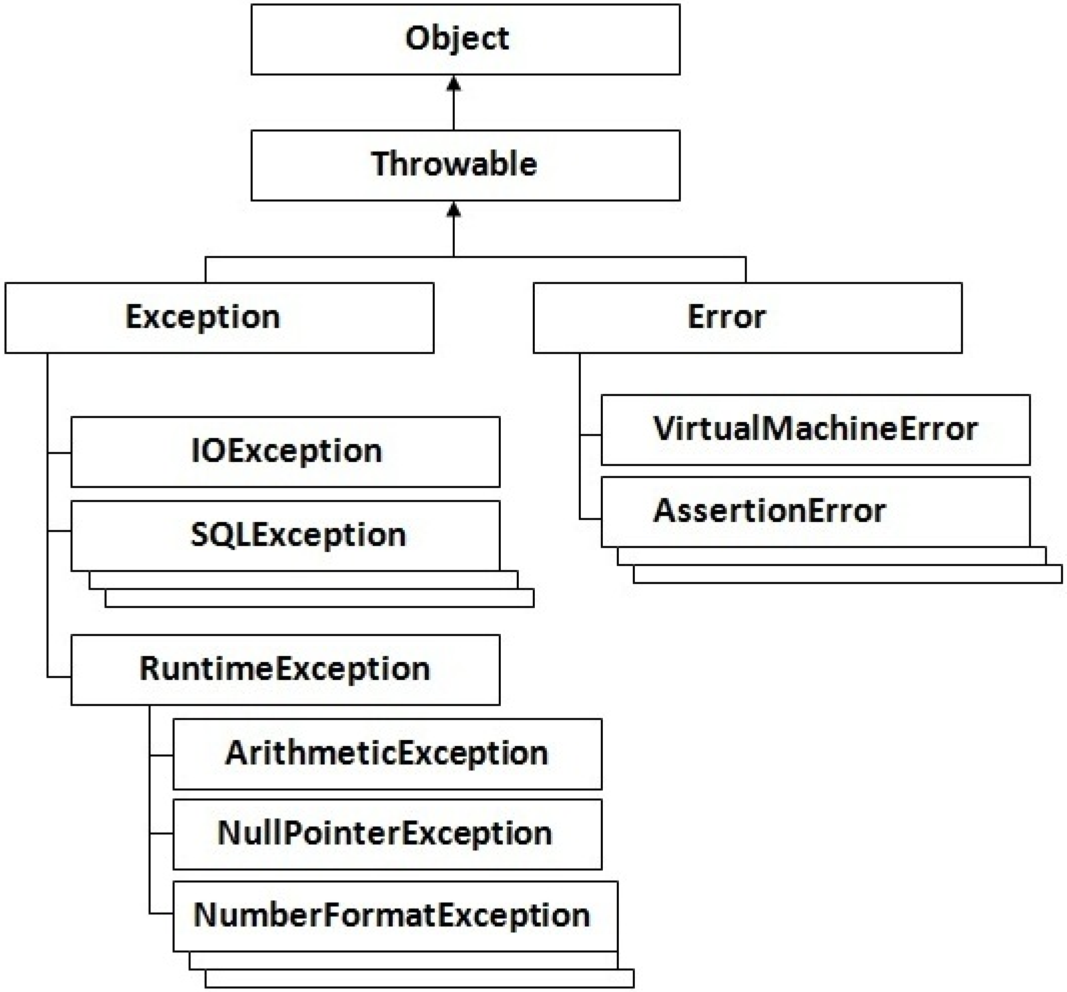


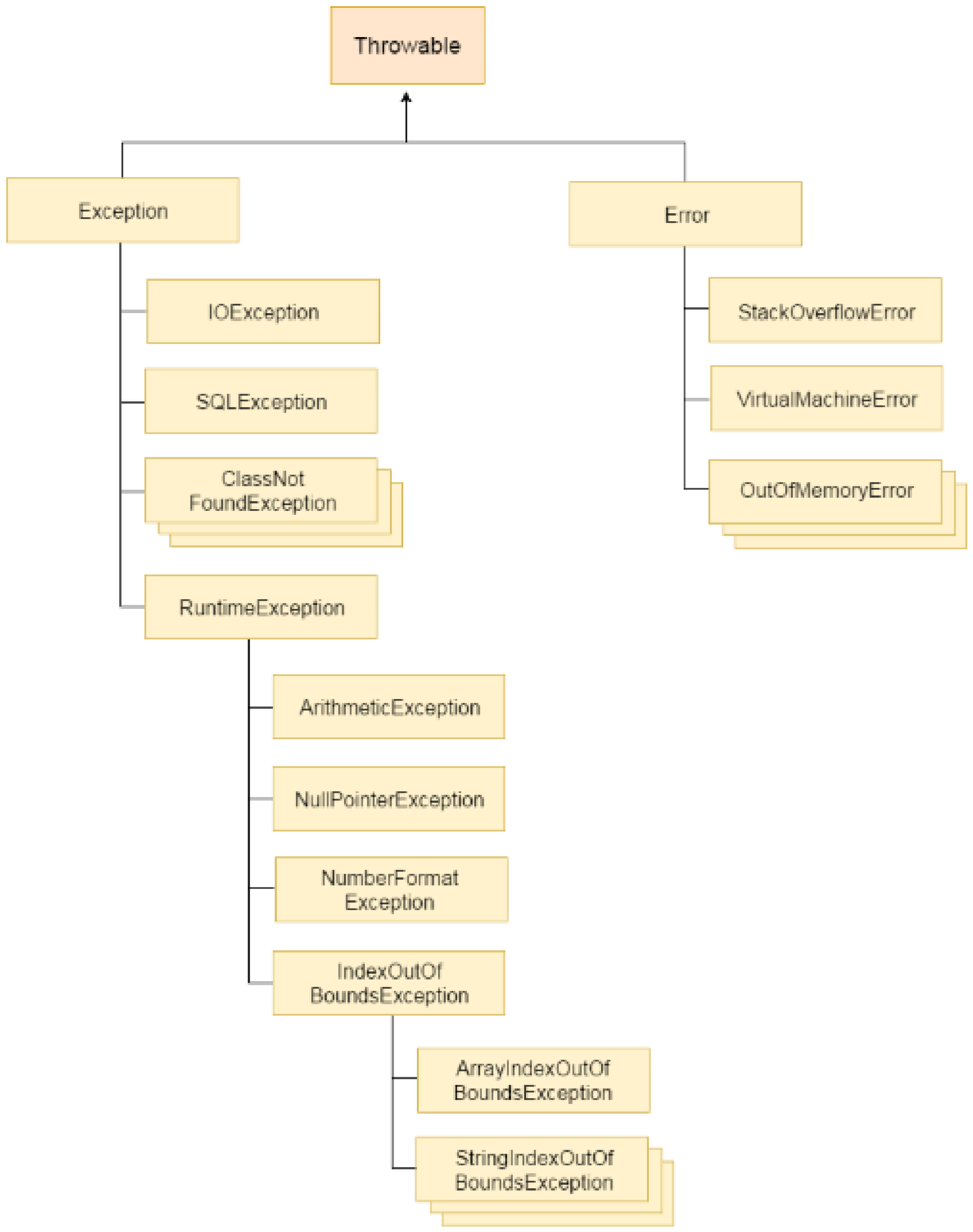
```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

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



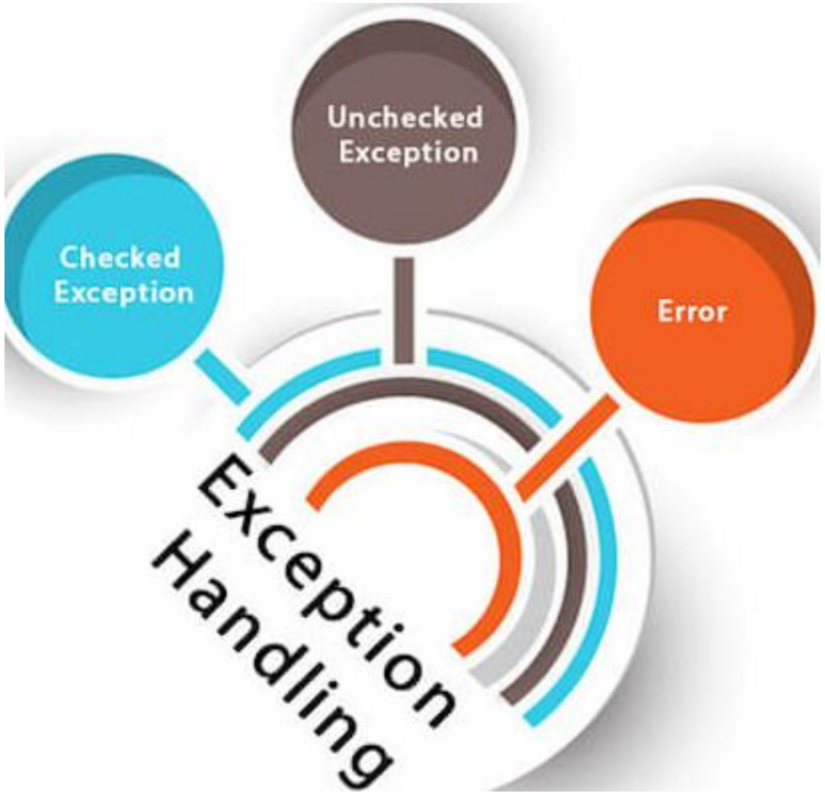

Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException
StackOverflowError
VirtualMachineError
OutOfMemoryError

Type of Java Exception

- There are mainly two types of exceptions:
  - **checked and**
  - **unchecked.**

Here, an **error** is considered as the unchecked exception.

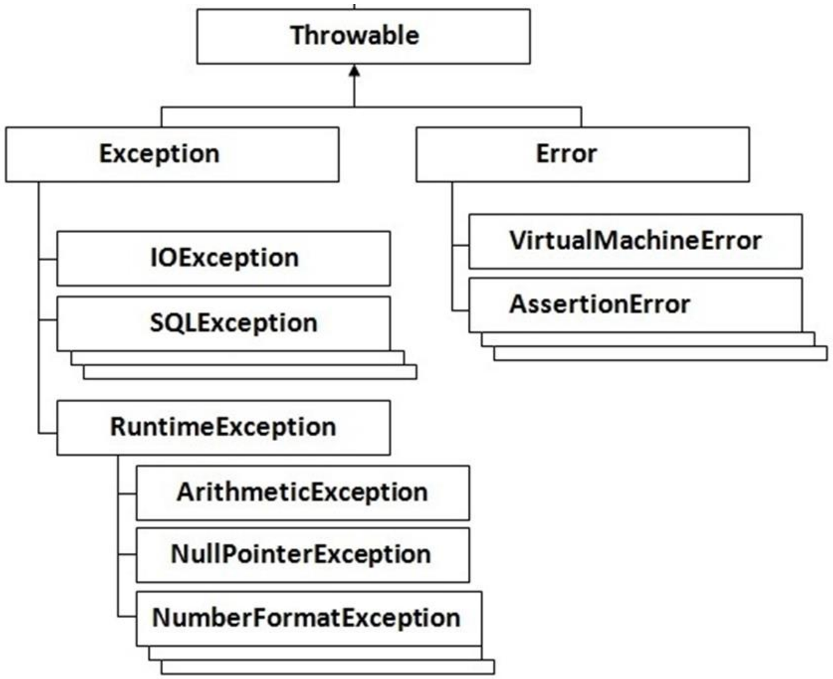
According to Oracle, there are three types of exceptions:



Checked Vs. Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.



2) Unchecked Exception

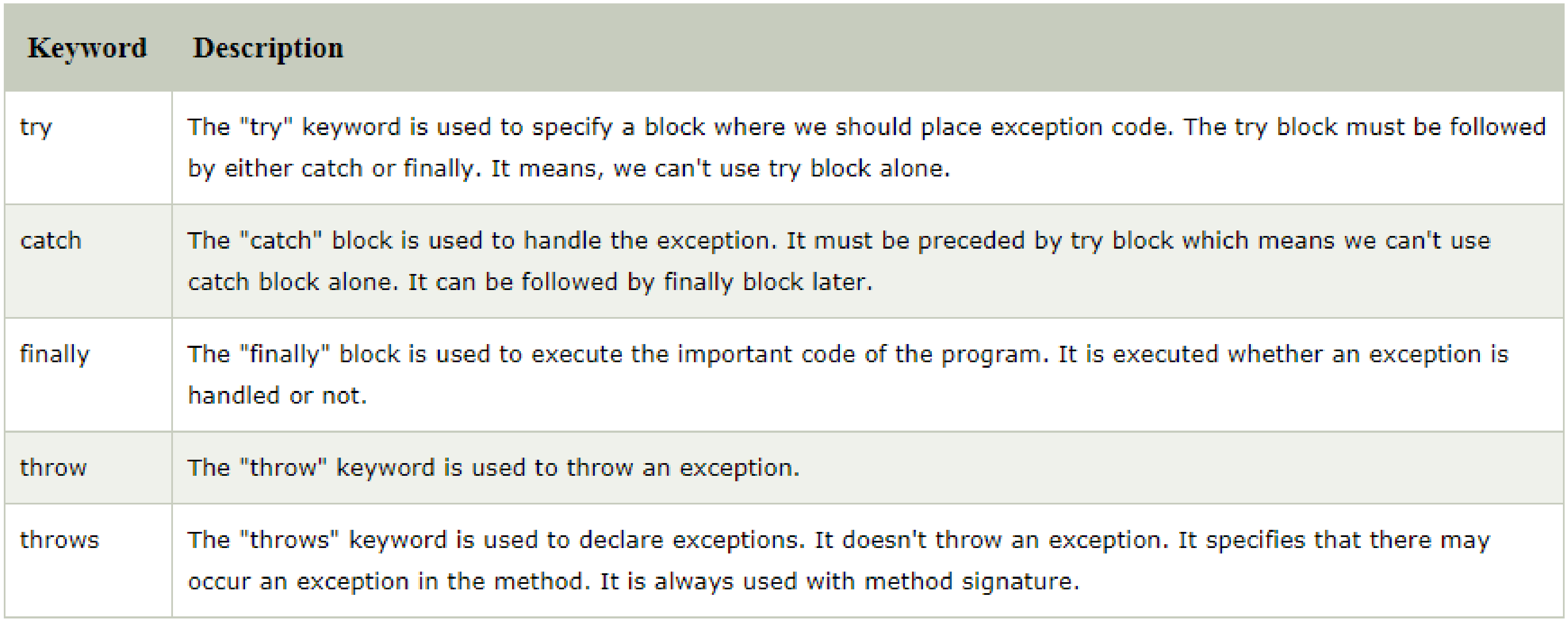
The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime. etc.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError

Java Exception Keywords

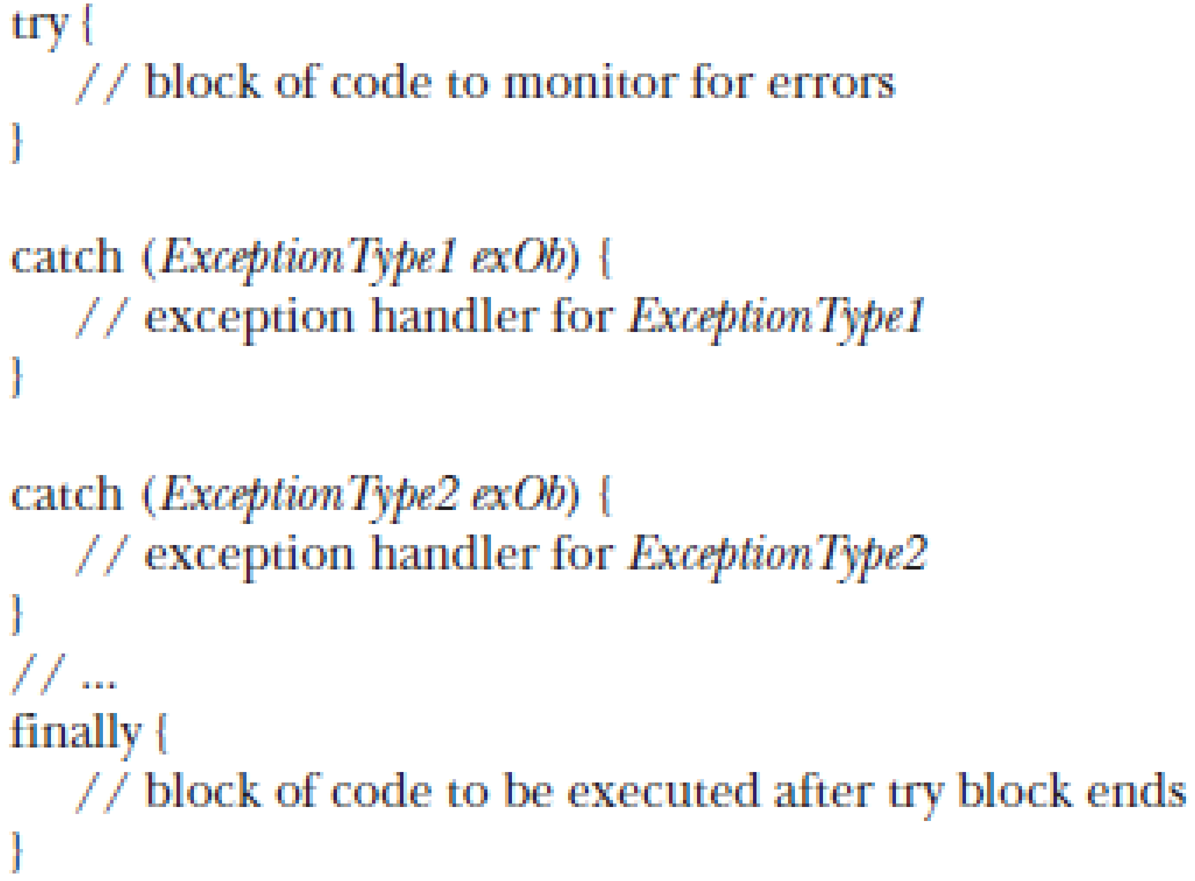
There are 5 keywords which are used in handling exceptions in Java.



Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

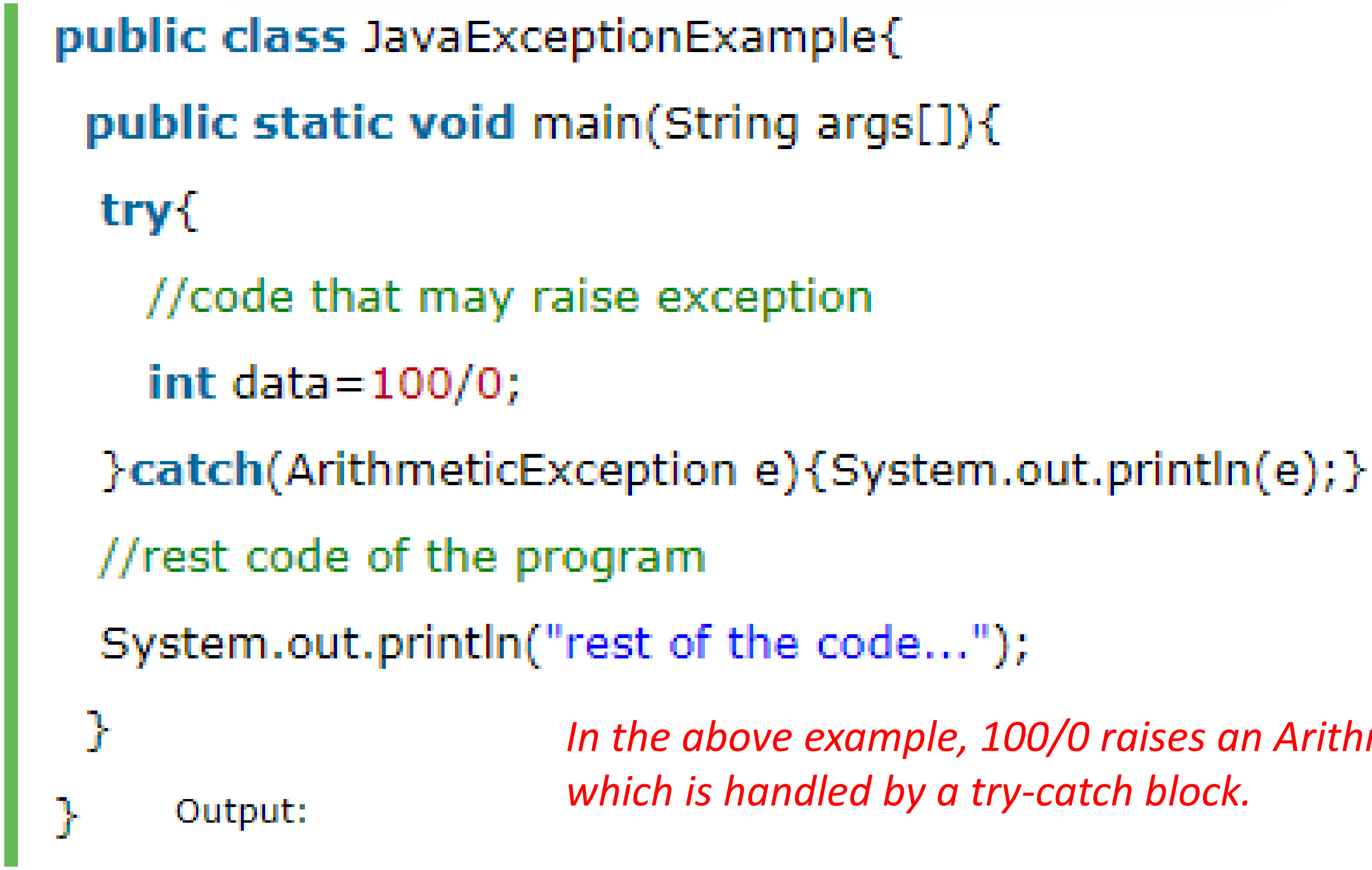
General form : exception-handling

This is the 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
}
```

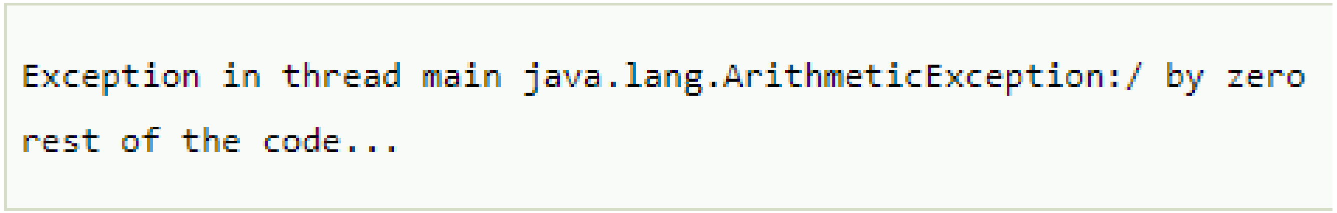
Exception Handling Example



```
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...");
  }
}
```

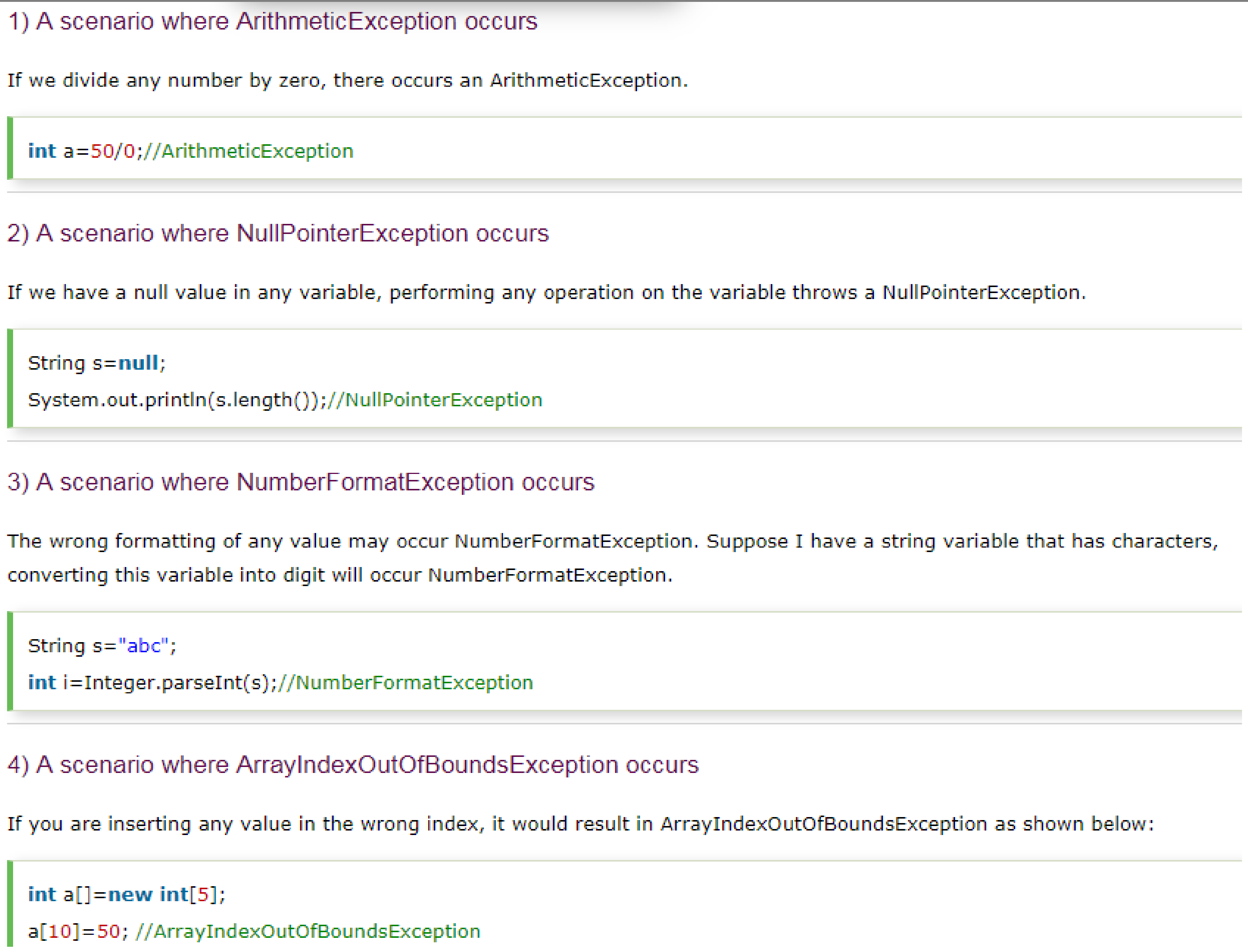
In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Output:



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

Common Scenarios of Java Exceptions



1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try block

Java try block

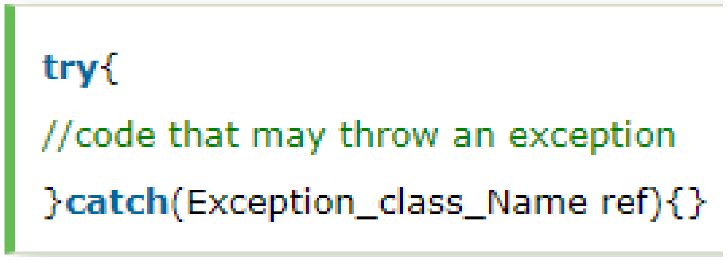
Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute.

So, it is recommended not to keeping the code in try block that will not throw an exception.

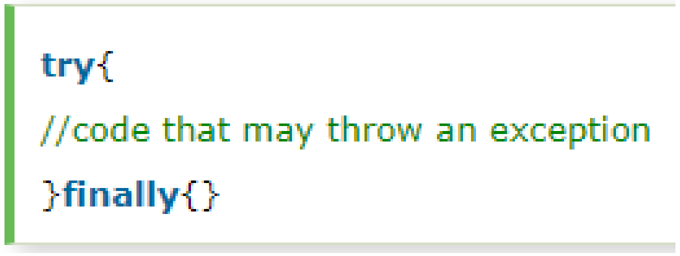
Java try block must be followed by either **catch** or **finally** block

Syntax of Java try-catch



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

Syntax of try-finally block



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

Java try catch block

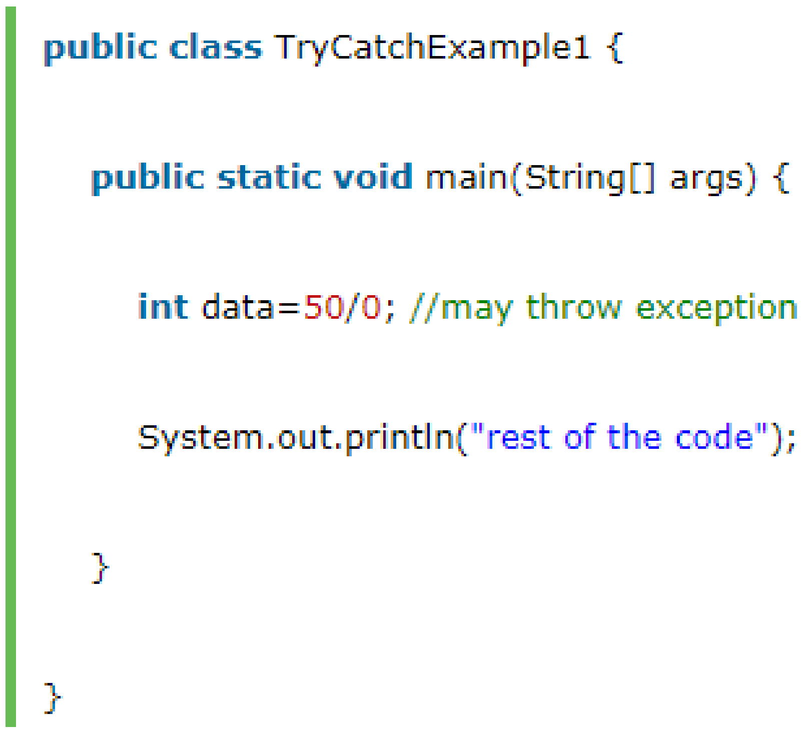
Java Catch block

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

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

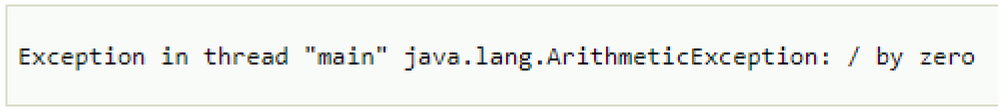
try-catch: Problem -Solution

Problem without exception handling



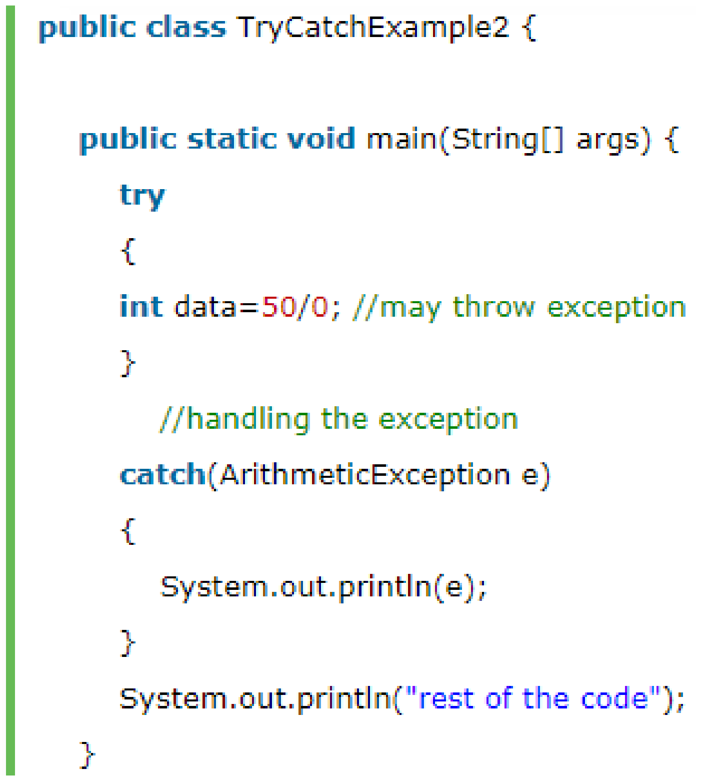
```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

Output:



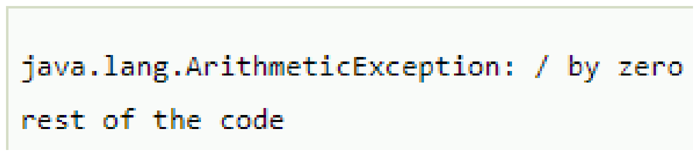
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Solution by exception handling



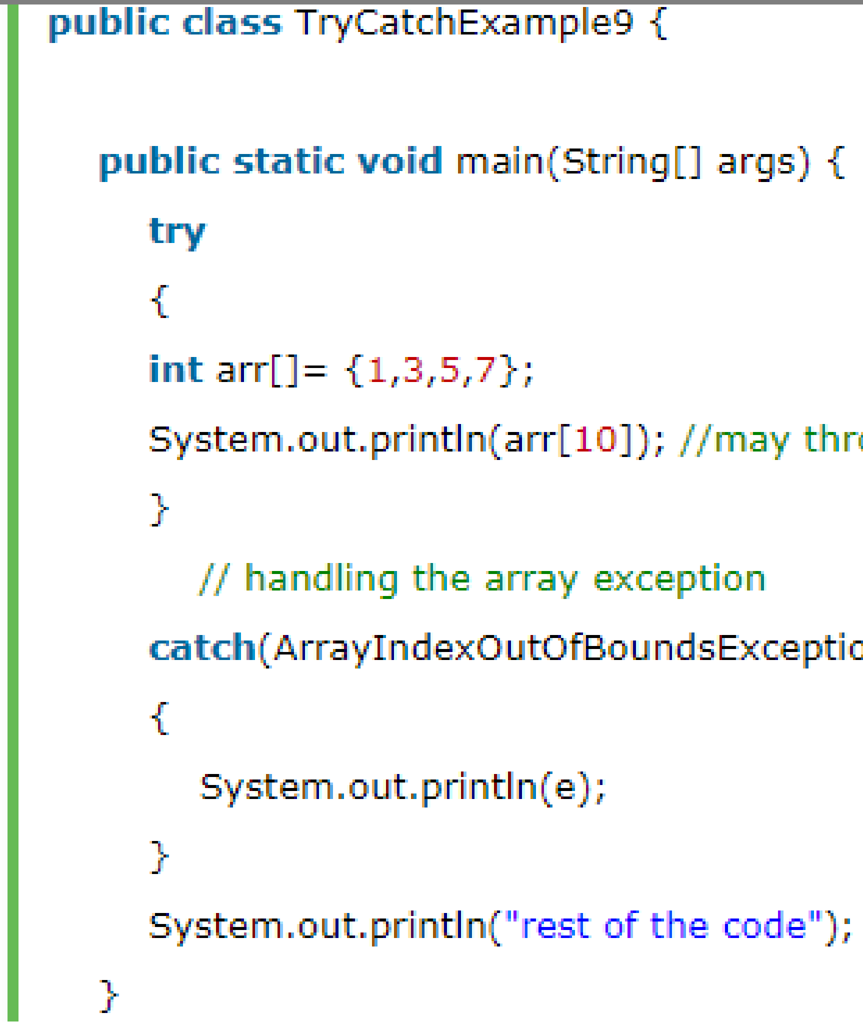
```
public class TryCatchExample2 {

    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:

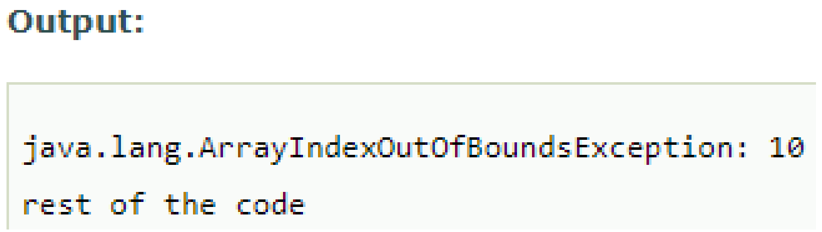


```
java.lang.ArithmeticException: / by zero
rest of the code
```

ArrayIndexOutOfBoundsException



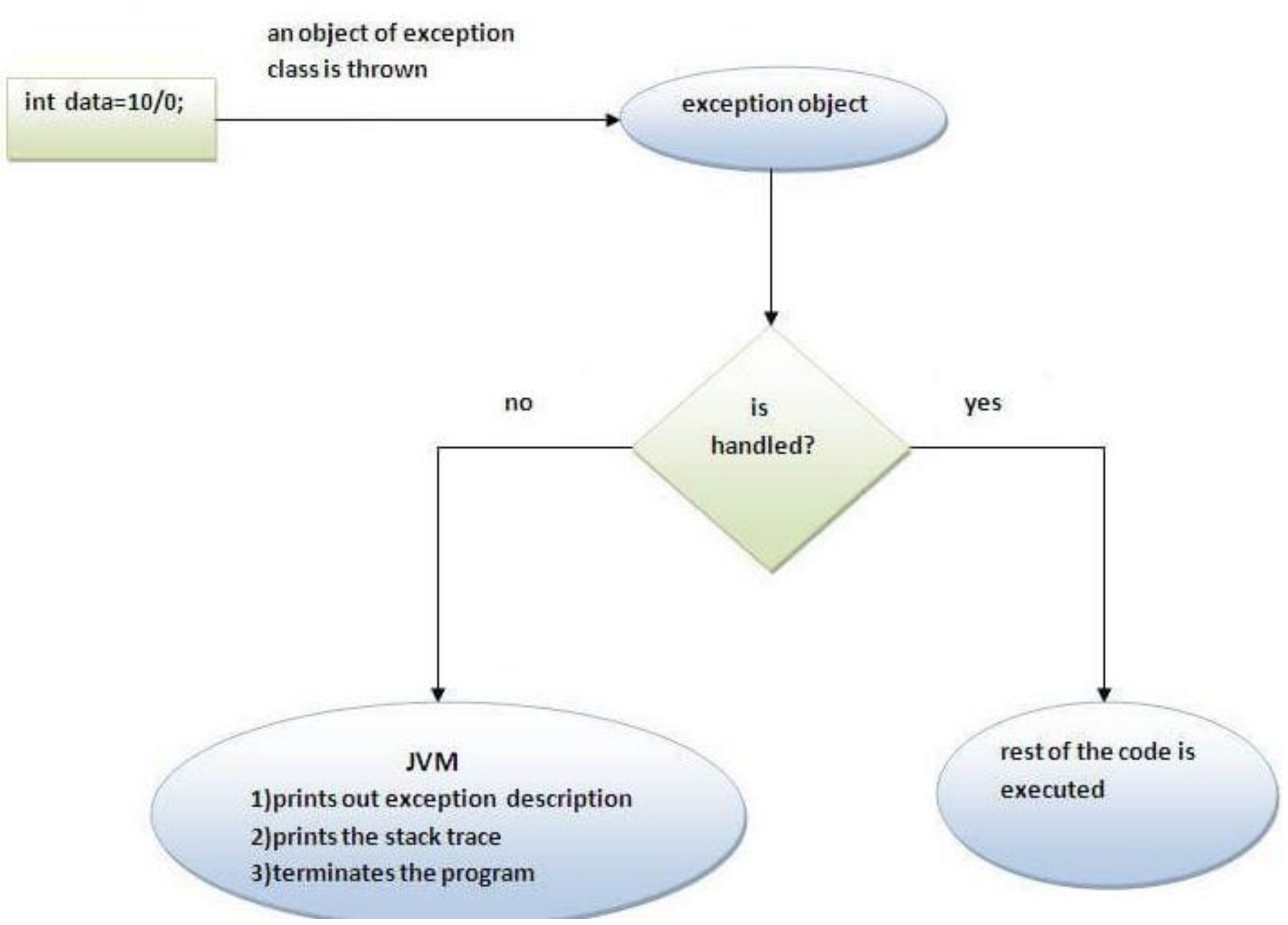
```
public class TryCatchExample9 {

    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
rest of the code
```

Internal working of java try-catch block


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

Java catch multiple exceptions

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler.
- So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

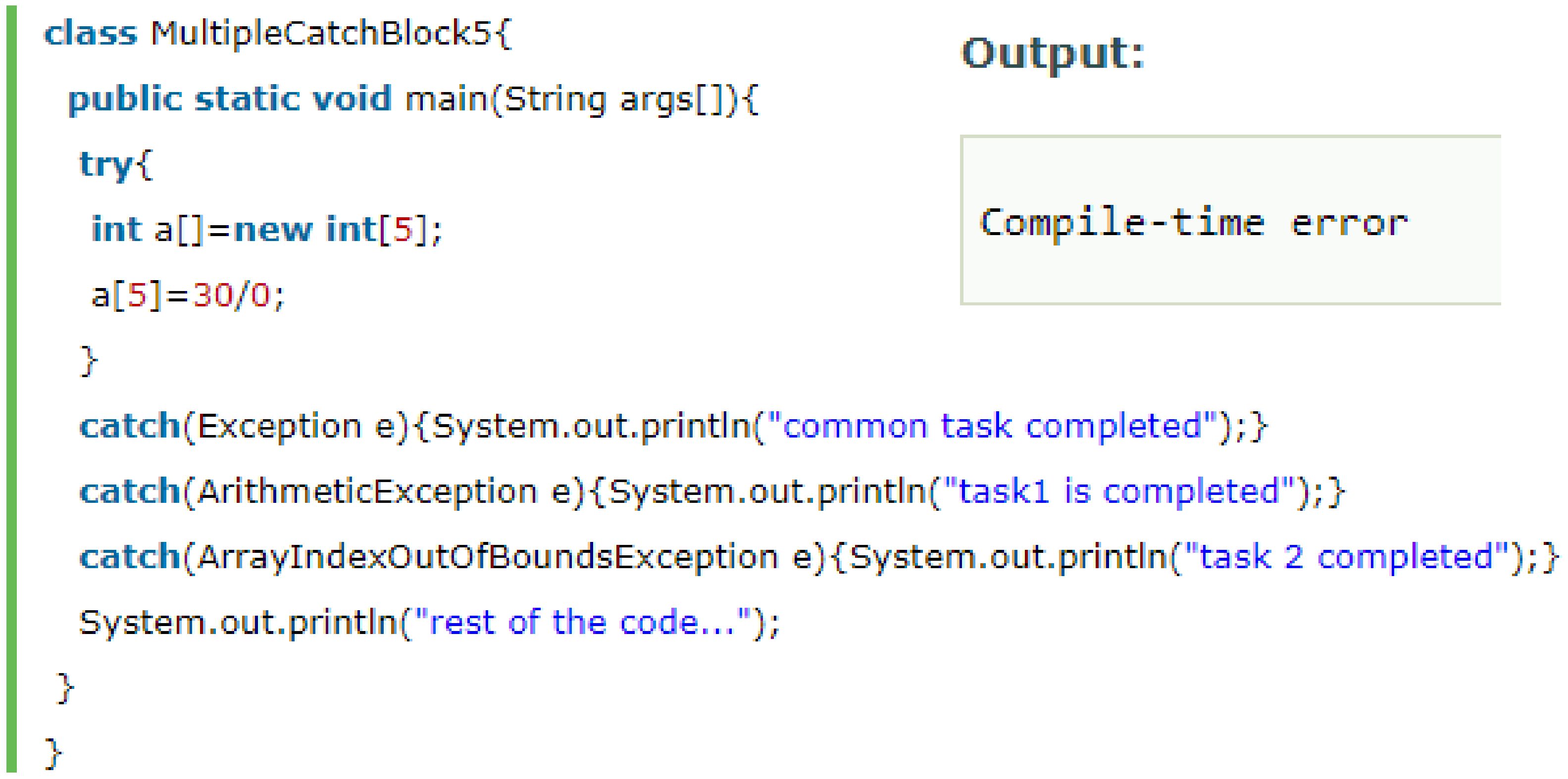
- ✓ At a time only one exception occurs and at a time only one catch block is executed.
- ✓ All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Java catch multiple exceptions

```
try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   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");
```

```
try{
    String s=null;
    System.out.println(s.length());
   }
   catch(ArithmeticException e)
     {
      System.out.println("Arithmetic Exception occurs");
     }
   catch(ArrayIndexOutOfBoundsException e)
     {
      System.out.println("ArrayIndexOutOfBounds Excep
     }
   catch(Exception e)
     {
      System.out.println("Parent Exception occurs");
     }
```

Without most specific to most general !



```
class MultipleCatchBlock5{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
 }
}
```

Output:

```
Compile-time error
```

Java Nested try - catch block

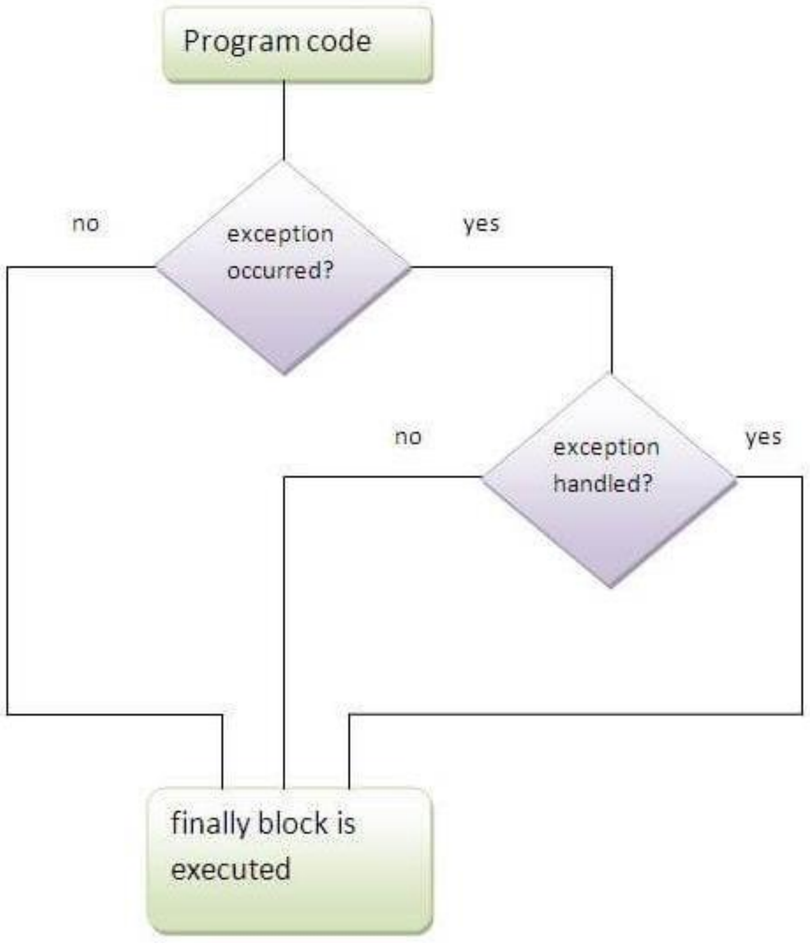
```
public static void main(String args[]){
 try{
   try{
    System.out.println("going to divide");
    int b =39/0;
   }catch(ArithmeticException e){System.out.println(e);}

   try{
   int a[]=new int[5];
   a[5]=4;
   }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

   System.out.println("other statement);
 }catch(Exception e){System.out.println("handeled");}
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.



Java finally block : Example

Java finally block is always executed whether exception is handled or not.

exception doesn't occur.

```
try{
 int data=25/5;
 System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
```

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

exception occurs and not handled.

```
try{
 int data=25/0;
 System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
```

```
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

```
try{
 int data=25/0;
 System.out.println(data);
}
catch(ArithmeticException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
```

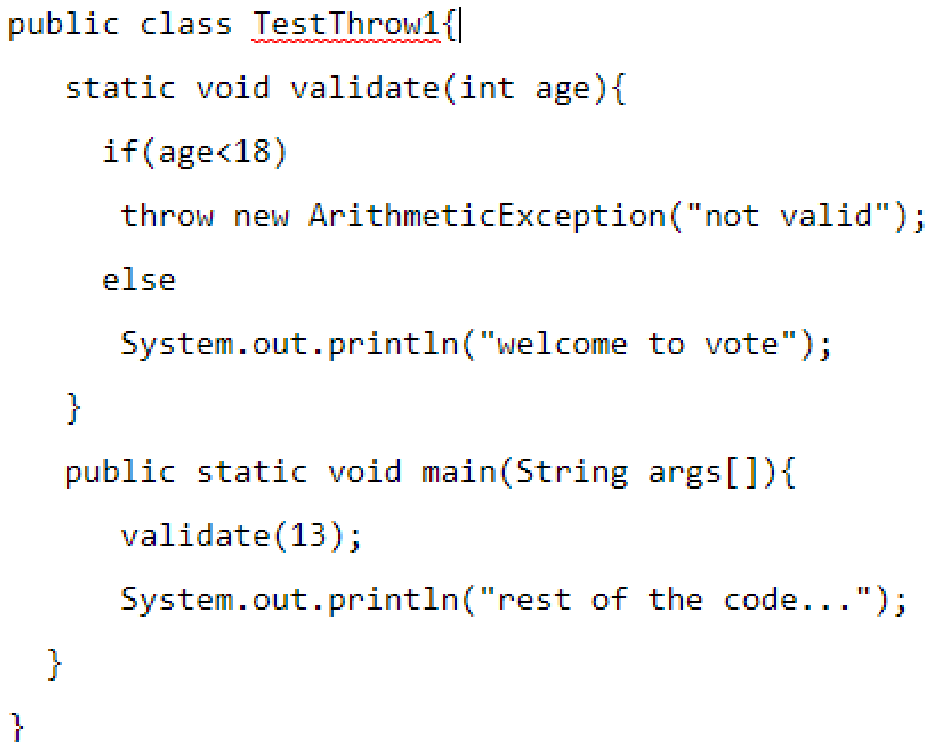
```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

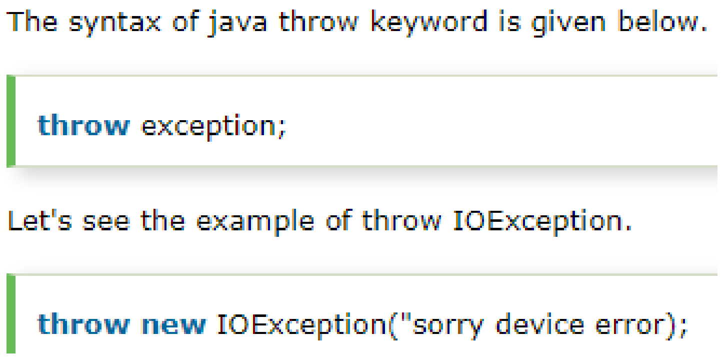
The throw keyword in Java

The Java throw keyword is used to ***explicitly throw*** an exception.

We can throw either ***checked or uncheked*** exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. Here we can see custom exceptions.



```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
 }
}
```



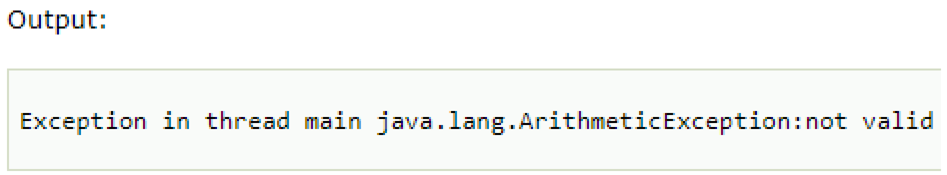
The syntax of java throw keyword is given below.

```
throw exception;
```

Let's see the example of throw IOException.

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

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message



Output:

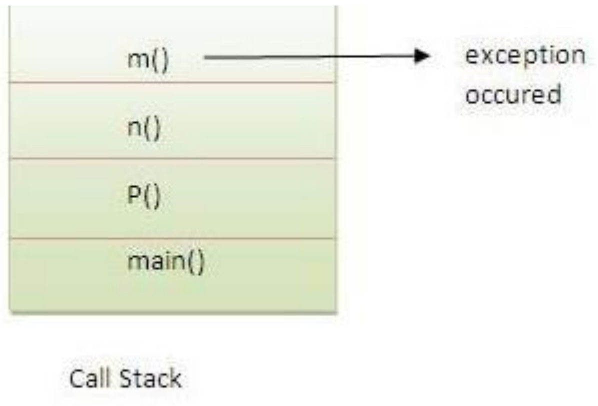
```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

```
void m(){
  int data=50/0;
}
void n(){
  m();
}
void p(){
 try{
  n();
 }catch(Exception e){System.out.println("exception handled");}
}
public static void main(String args[]){
 TestExceptionPropagation1 obj=new TestExceptionPropagation1();
 obj.p();
```



```
Output:exception handled
      normal flow...
```

In the example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Java Exception propagation

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

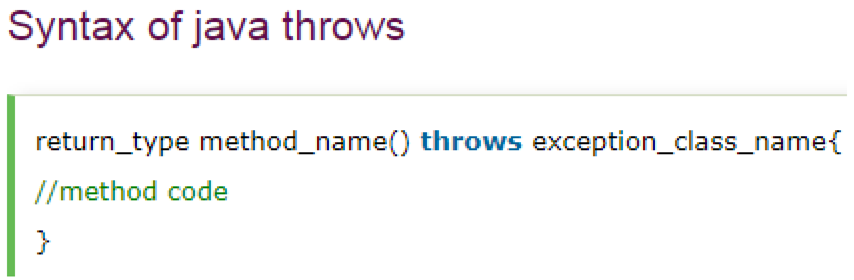
Program which describes that checked exceptions are not propagated

```
class TestExceptionPropagation2{
 void m(){
  throw new java.io.IOException("device error");//checked exception
 }
 void n(){
  m();
 }
 void p(){
  try{
  n();
  }catch(Exception e){System.out.println("exception handeled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation2 obj=new TestExceptionPropagation2();
  obj.p();
  System.out.println("normal flow");
 }
}
```

```
Output:Compile Time Error
```

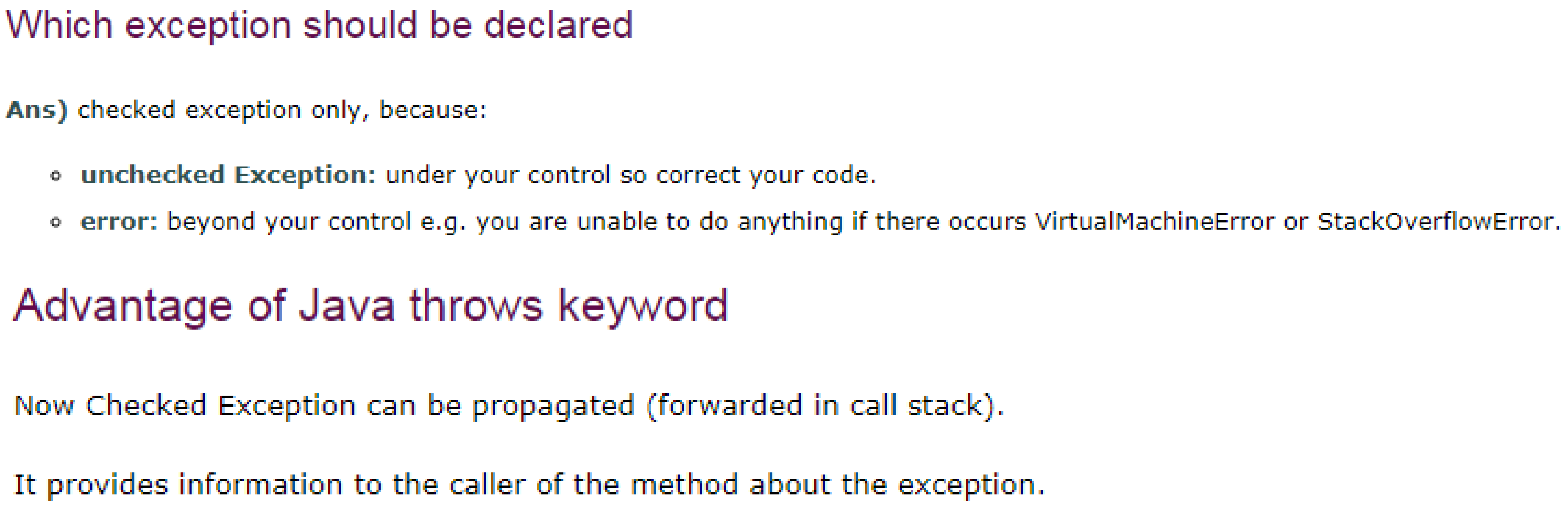
The throws keyword in Java

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



Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```



Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

throws : Example

```
import java.io.IOException;
class Testthrows1{
 void m()throws IOException{
  throw new IOException("device error");//checked exception
 }
 void n()throws IOException{
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handled");}
 }
 public static void main(String args[]){
  Testthrows1 obj=new Testthrows1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

throws : example

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

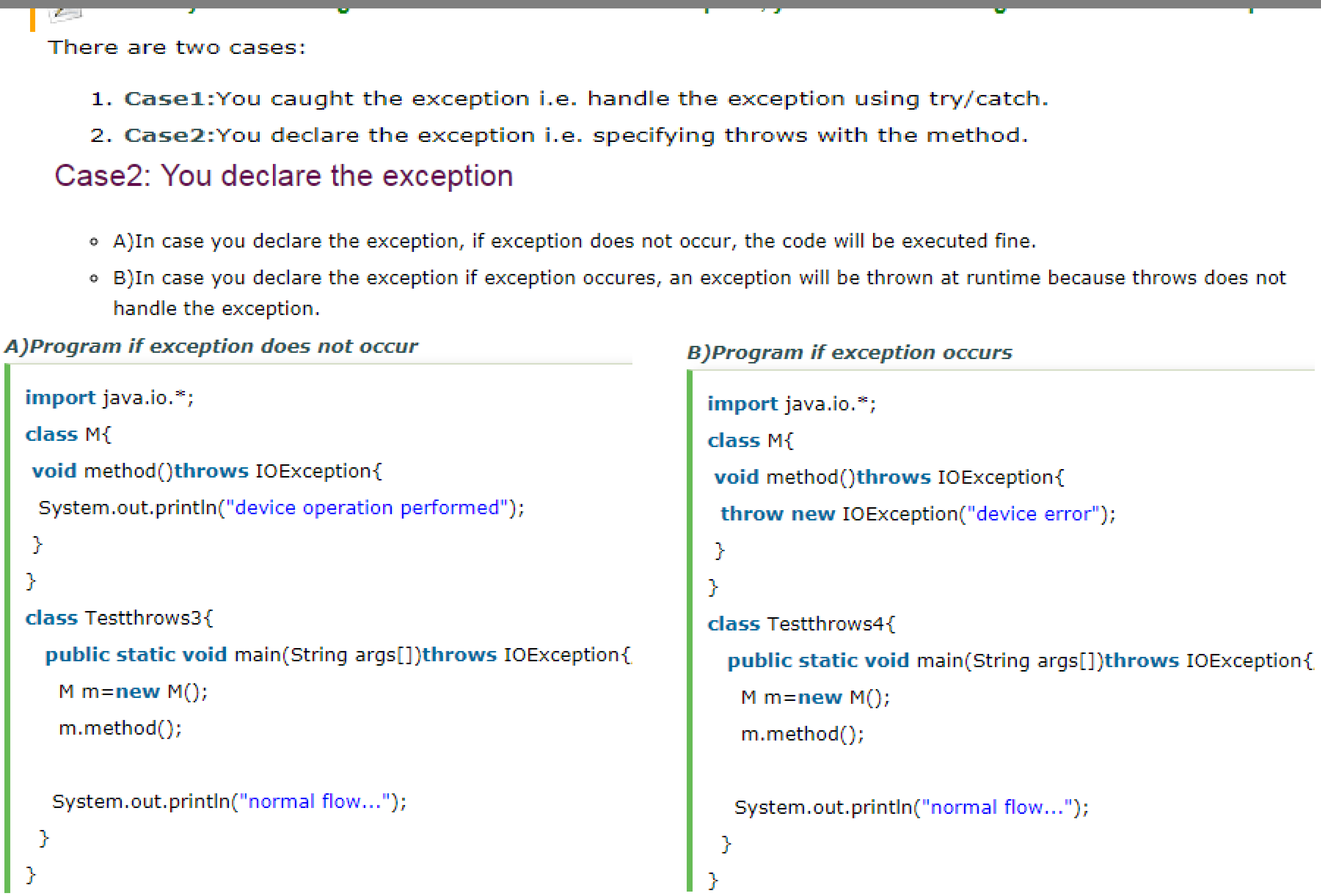
Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

```
Output:exception handled
       normal flow...
```

throws : example



There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

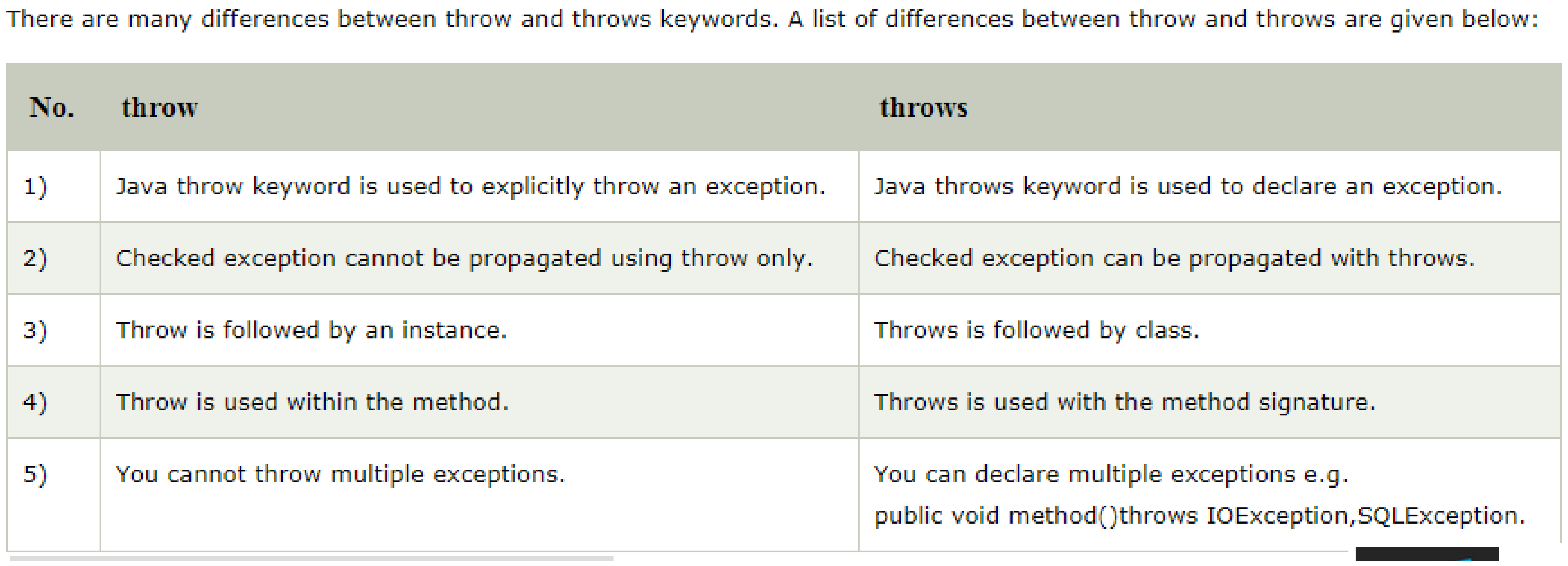
A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

B)Program if exception occurs

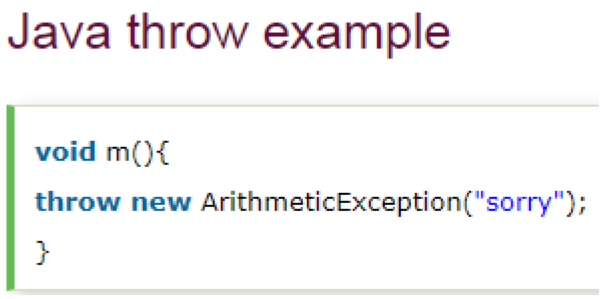
```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Difference between throw and throws in Java



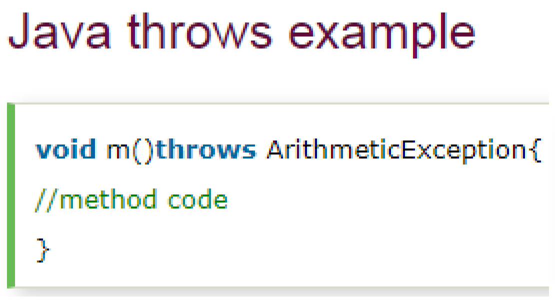
There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.



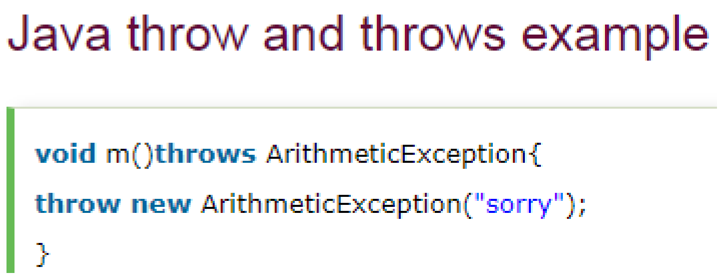
Java throw example

```
void m(){
throw new ArithmeticException("sorry");
}
```



Java throws example

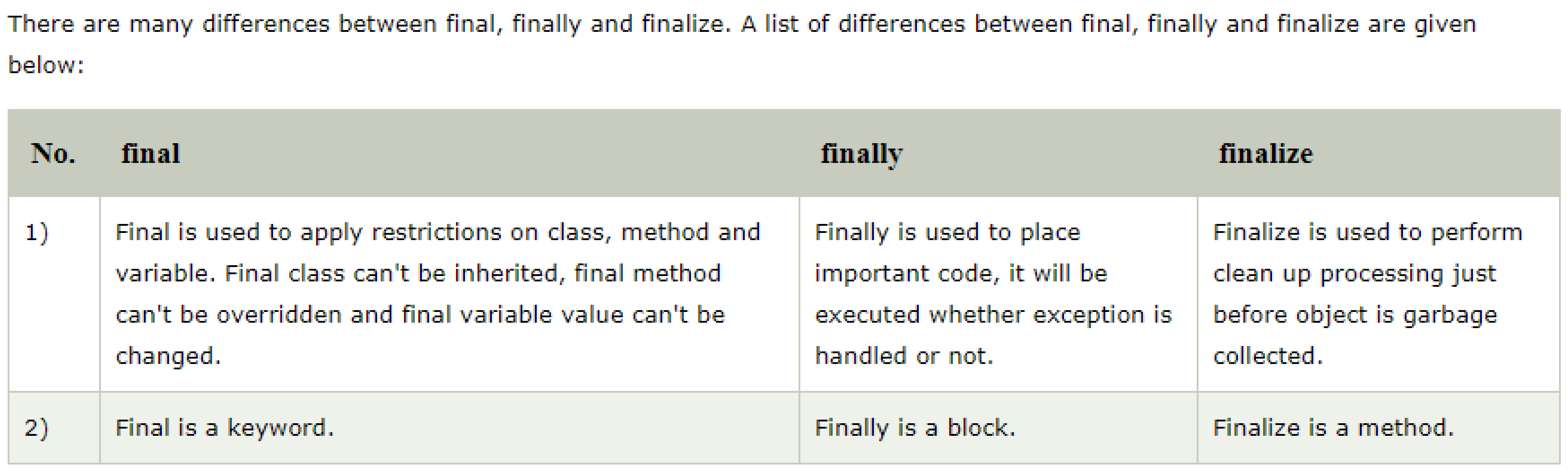
```
void m()throws ArithmeticException{
//method code
}
```



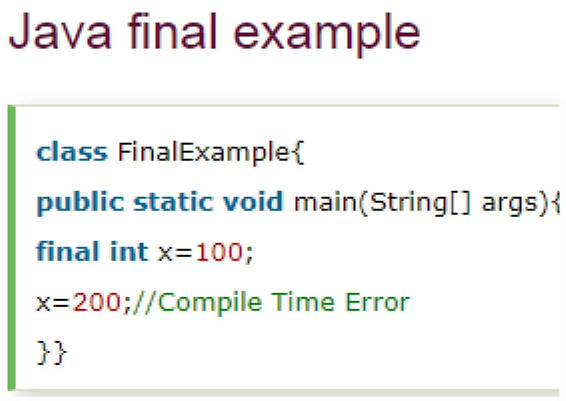
Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

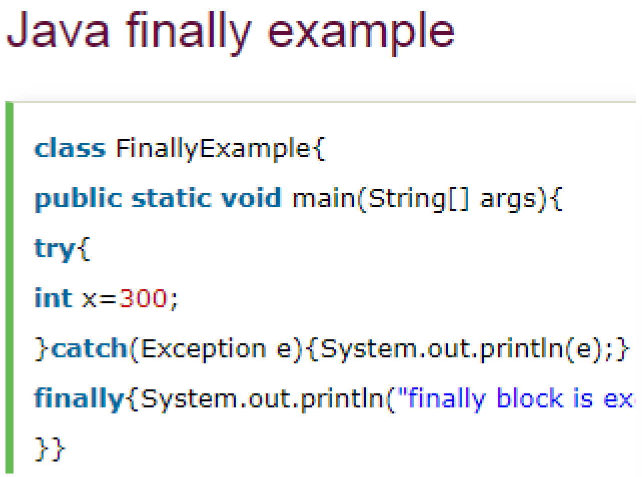
Difference between final, finally and finalize


There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

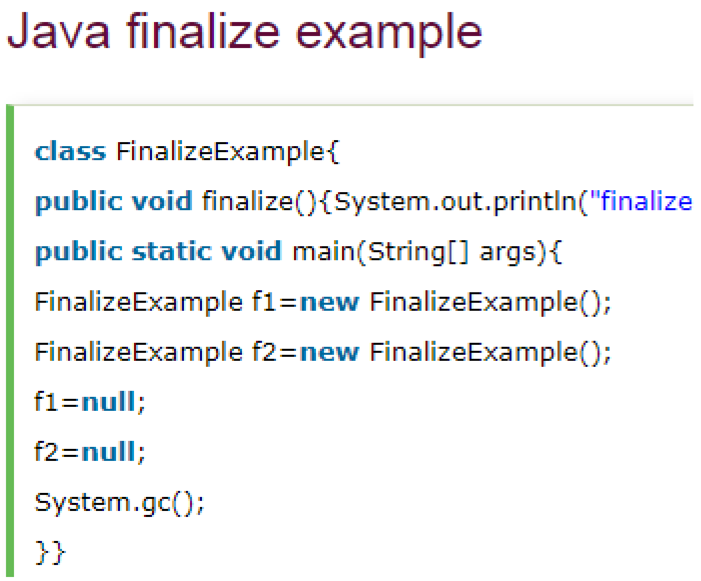
No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.


Java final example

```
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}}
```

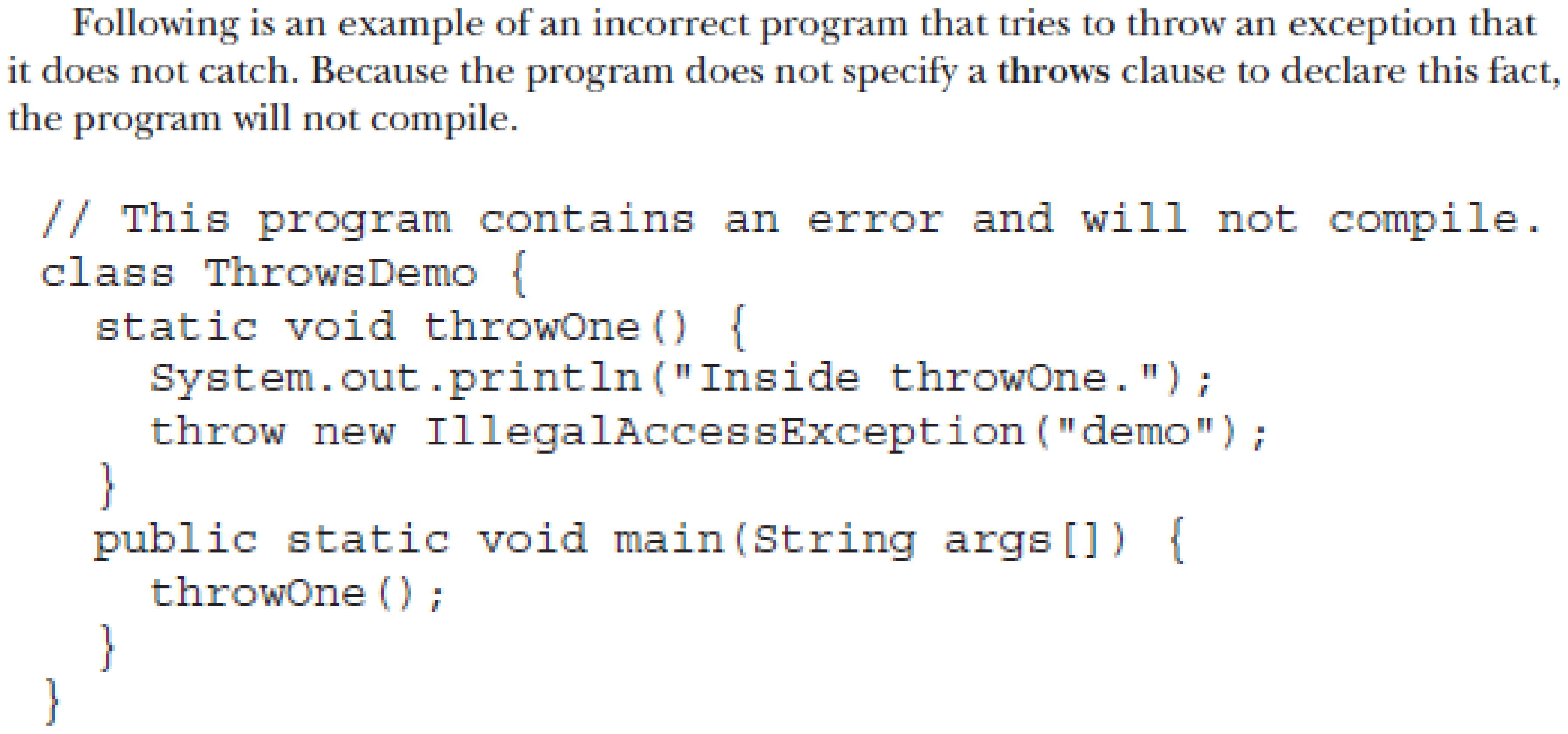

Java finally example

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is ex
}}
```


Java finalize example

```
class FinalizeExample{
public void finalize(){System.out.println("finalize
public static void main(String[] args){
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}}
```

throw-throws : Exercise



Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

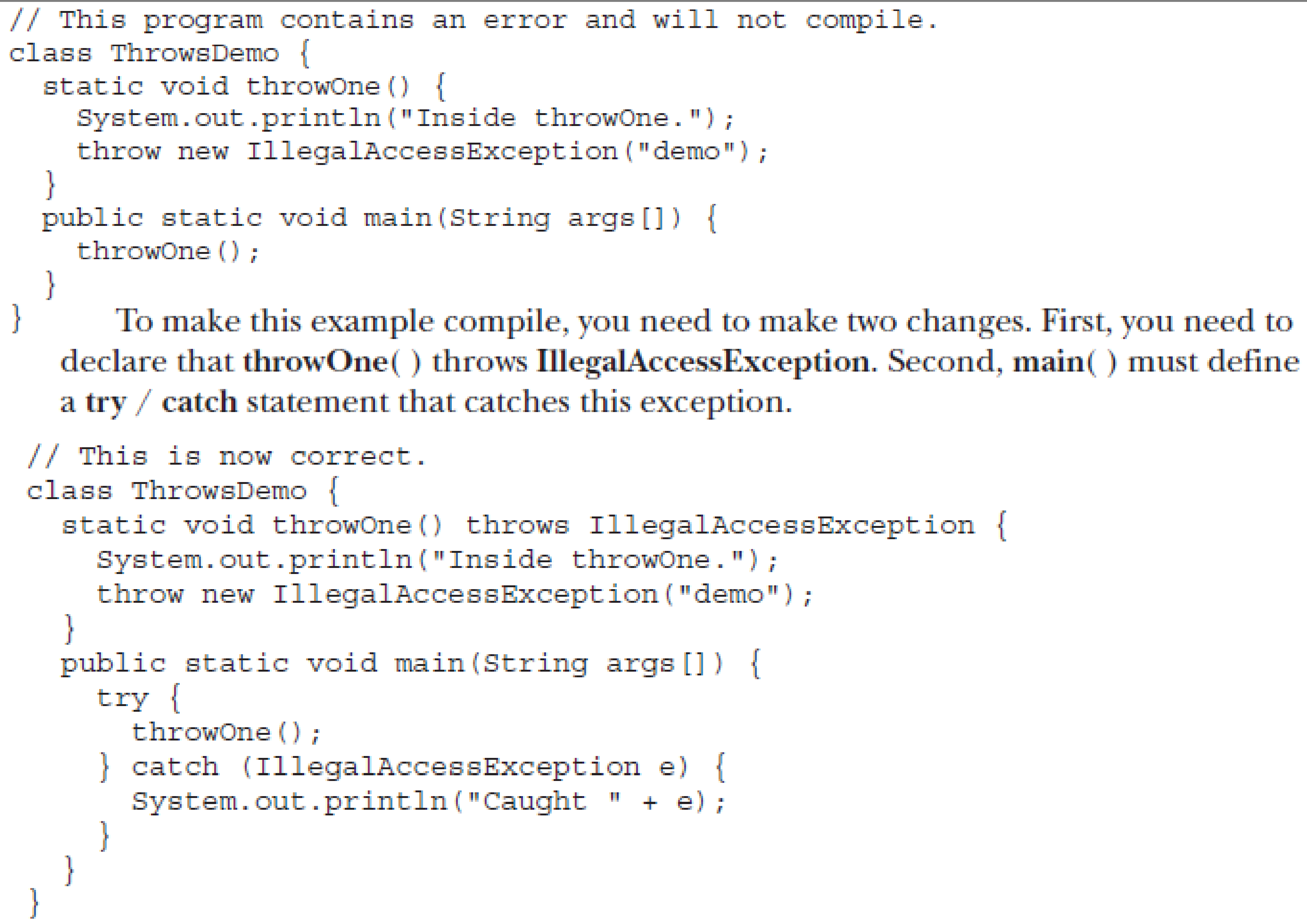
```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

throw-throws: Solution

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try** / **catch** statement that catches this exception.

```
// This is now correct.
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);
    }
  }
}
```

throw-throws : Solutions



```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try** / **catch** statement that catches this exception.

```
// This is now correct.
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);
    }
  }
}
```