Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf
yousuf@juniv.edu

Exception Handling-2

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.**

```
public class ExceptionExample {
   public static void main(String argv[]) {
       int num1 = 10;
       int num2 = 0;
       int result = 0;
       int arr[] = new int[5];
       try {
            arr[0] = 0;
            arr[1] = 1;
            arr[2] = 2;
            arr[3] = 3;
            arr[4] = 4;
            arr[5] = 5;
      result = num1 / num2;
      System.out.println("Result of Division : " + result);
      }catch (ArithmeticException e)
                  { System.out.println("Err: Divided by Zero"); }
       catch (ArrayIndexOutOfBoundsException e)
                  { System.out.println("Err: Array Out of Bound"); }
} }
```

Example 1

Explanation of Example 1

- Output :

  Err: Array Out of Bound
- In the above example we have two lines that might throw an exception i.e

```
arr[5] = 5;
```

above statement can cause array index out of bound exception and

```
result = num1 / num2;
```

this can cause arithmetic exception.

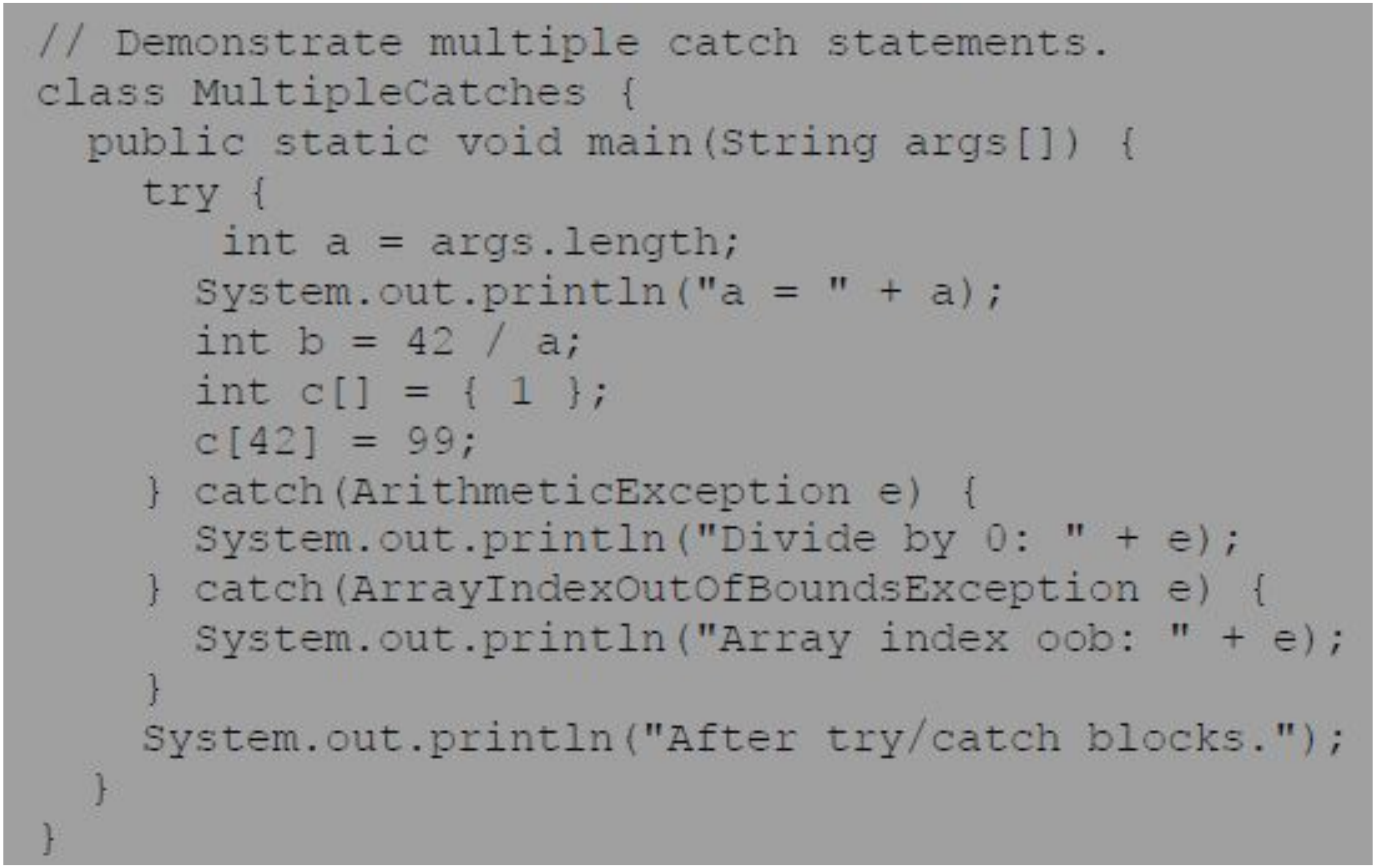
Explanation of Example 1

To Handle these two different types of exception we have included two catch blocks for single try block.

```
}catch (ArithmeticException e)
        { System.out.println("Err: Divided by Zero"); }
 catch (ArrayIndexOutOfBoundsException e)
       { System.out.println("Err: Array Out of Bound"); }
```

- Now Inside the try block when exception is thrown then type of the exception thrown is compared with the type of exception of each catch block.
- If type of exception thrown is matched with the type of exception from catch then it will execute corresponding catch block.

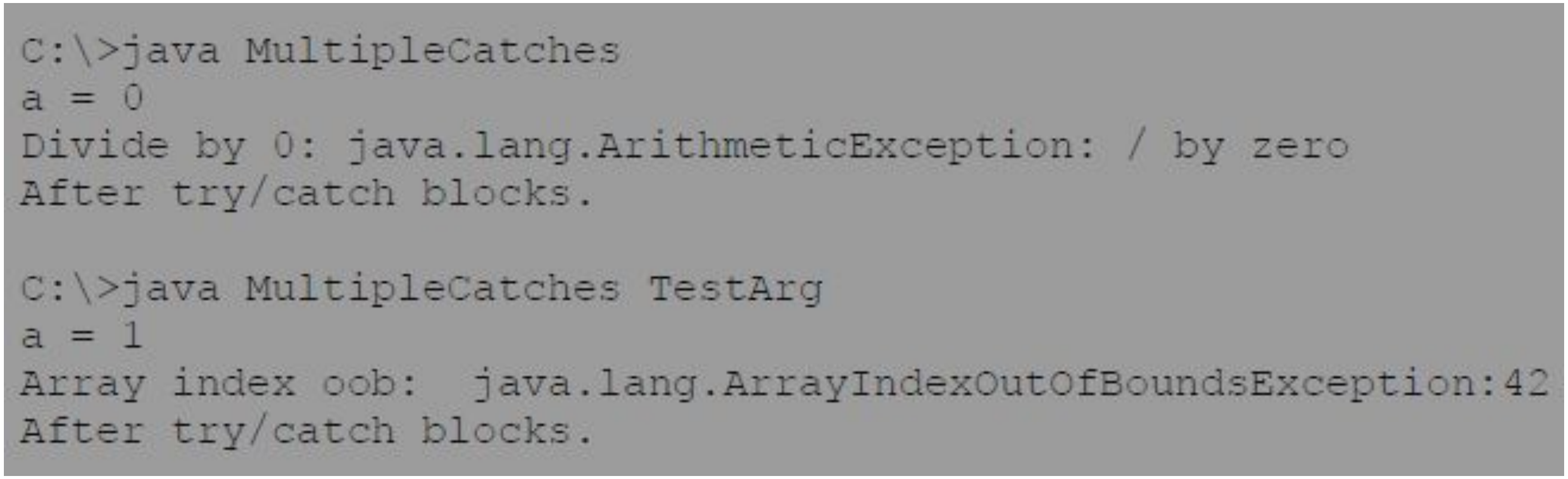
Multiple catch Clauses : Example 2



```
// Demonstrate multiple catch statements.
class MultipleCatches {
  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.");
  }
}
```

Multiple catch Clauses: Example 2

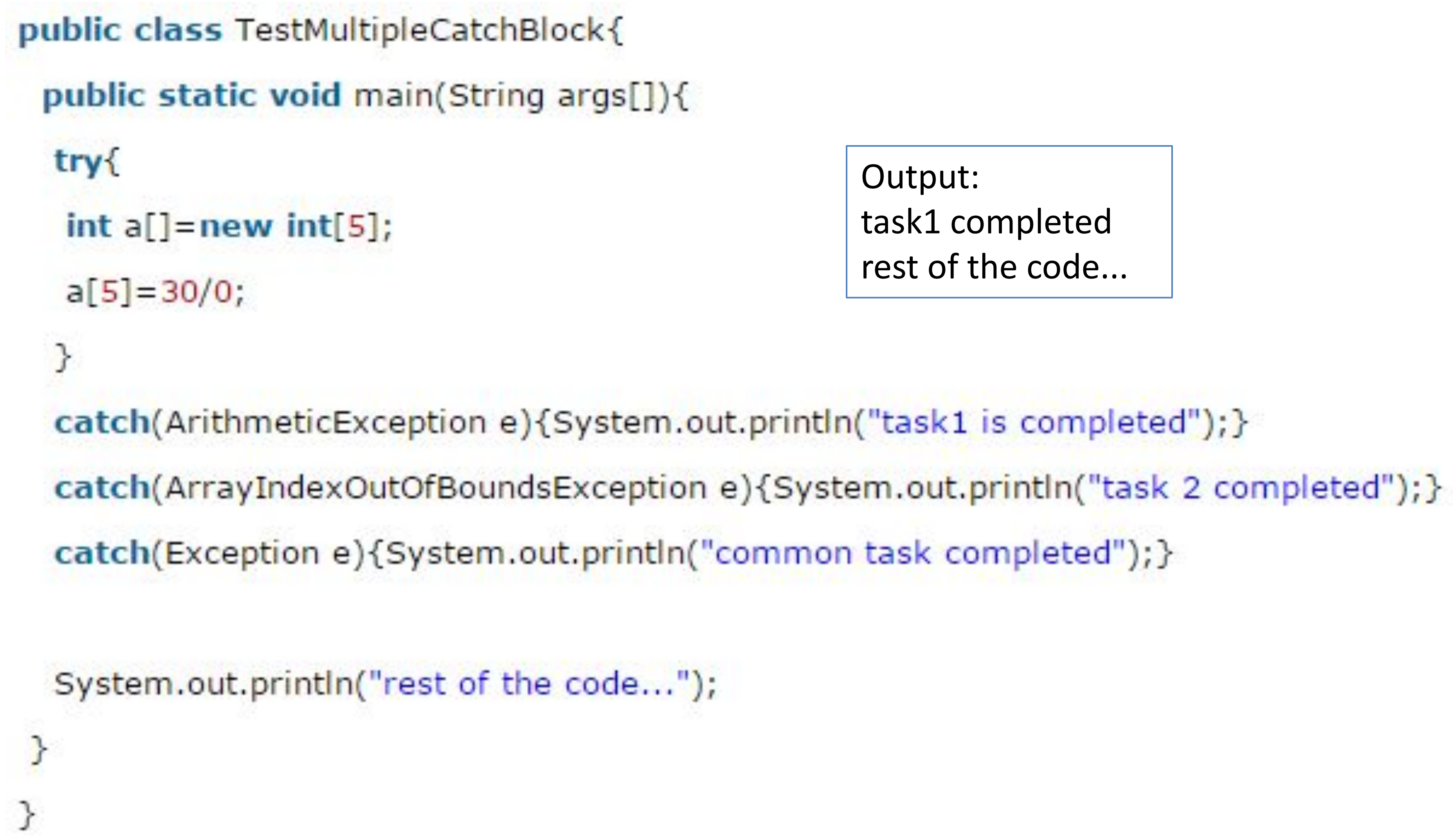
- The output generated by running it both ways:



```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob:  java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Multiple catch Clauses: Example 2



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

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

Output:

```
task1 completed
rest of the code...
```

Multiple catch Clauses

- **Rule 1: At a time only one Exception is occurred.**
- Rule 2: At a time only **single catch block can be executed**. After the execution of catch block control goes to the statement next to the try block.
- **Rule 3:** When you use multiple **catch statements, it is important to remember that exception** subclasses must come before any of their super-classes.

Multiple catch Clauses

```
/* This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
     } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
     catch(ArithmeticException e) { // ERROR - unreachable
       System.out.println("This is never reached.");
     }
  }
}
```

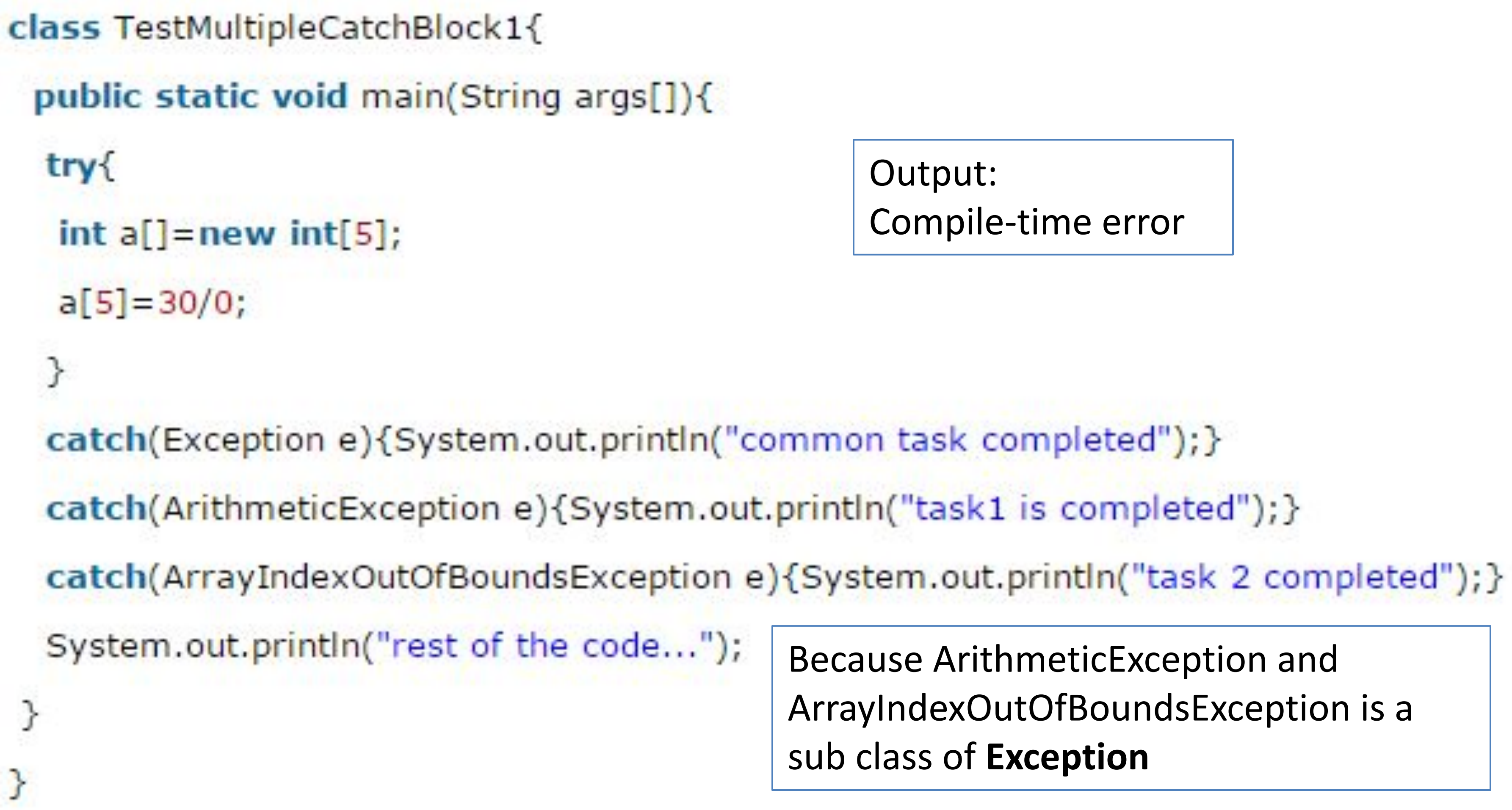
Multiple catch Clauses

- If you try to compile this program, you will receive an error message stating that the second **catch statement is unreachable because the exception has already been caught.**

- Since **ArithmeticException is a subclass of Exception,** the first catch statement will handle all **Exception-based errors, including ArithmeticException.**

- This means that the second **catch statement will never execute**. To fix the problem, reverse the order of the catch statements.

Multiple catch Clauses



```
class TestMultipleCatchBlock1{
 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

Because ArithmeticException and ArrayIndexOutOfBoundsException is a sub class of **Exception**

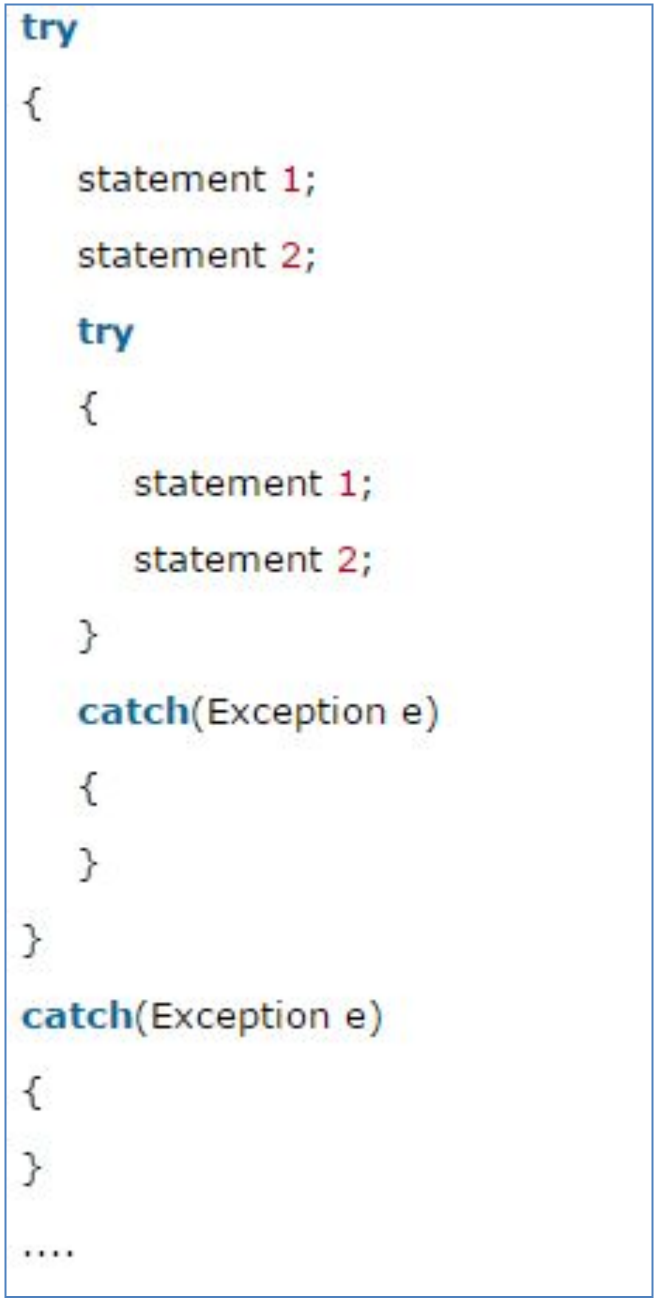
Nested try Statements

- The try block within a try block is known as nested try block in java.

Why use nested try block:

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

- Syntax:



```
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

```
class Excep6{
 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");}


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

throw

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the **throw statement.**

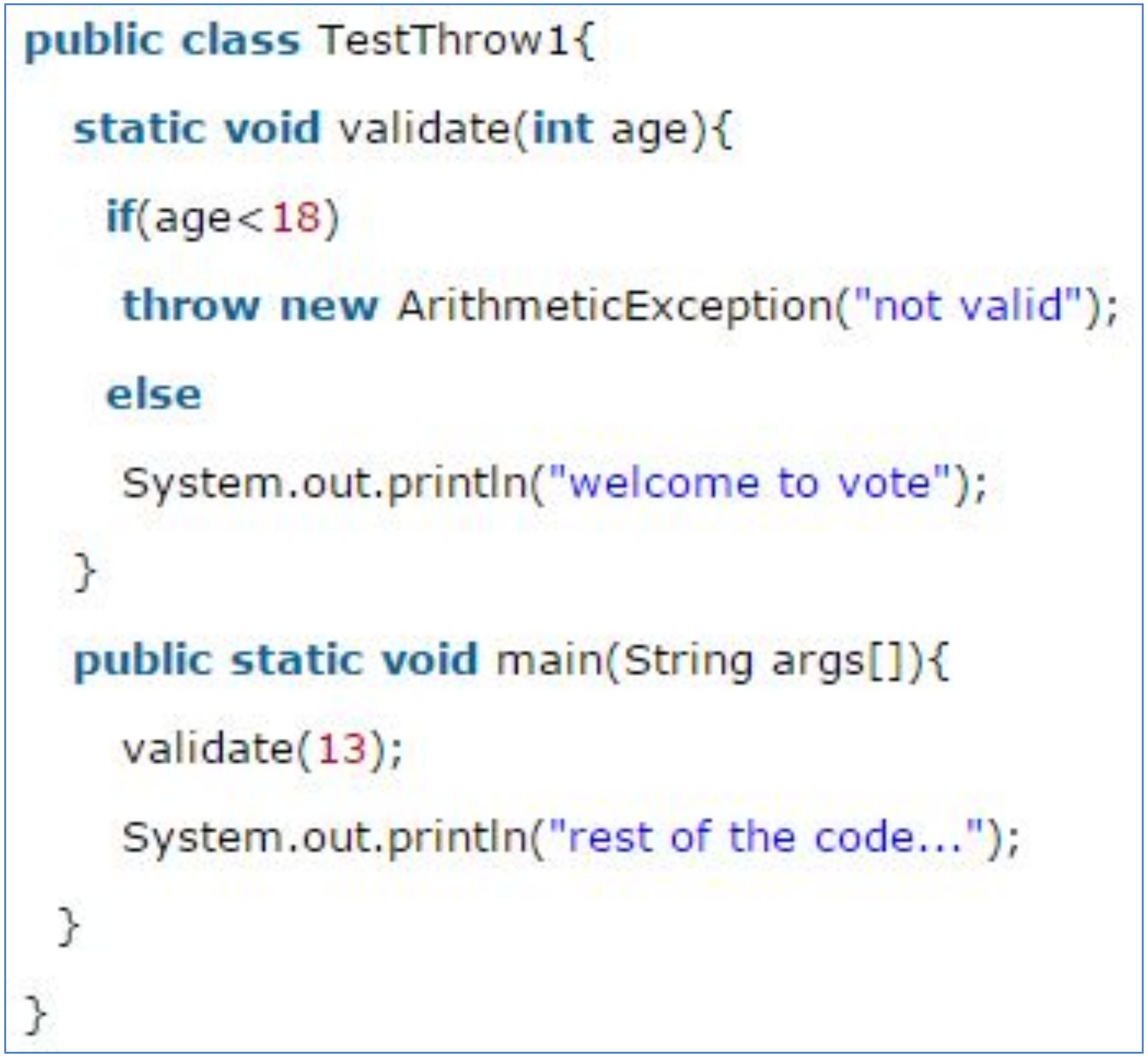
- **The general form of throw is shown here:**

  throw exception;

  Let's see the example of throw IOException.

  throw new IOException("sorry device error”);

throw



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

Sample program that creates and throws an exception.

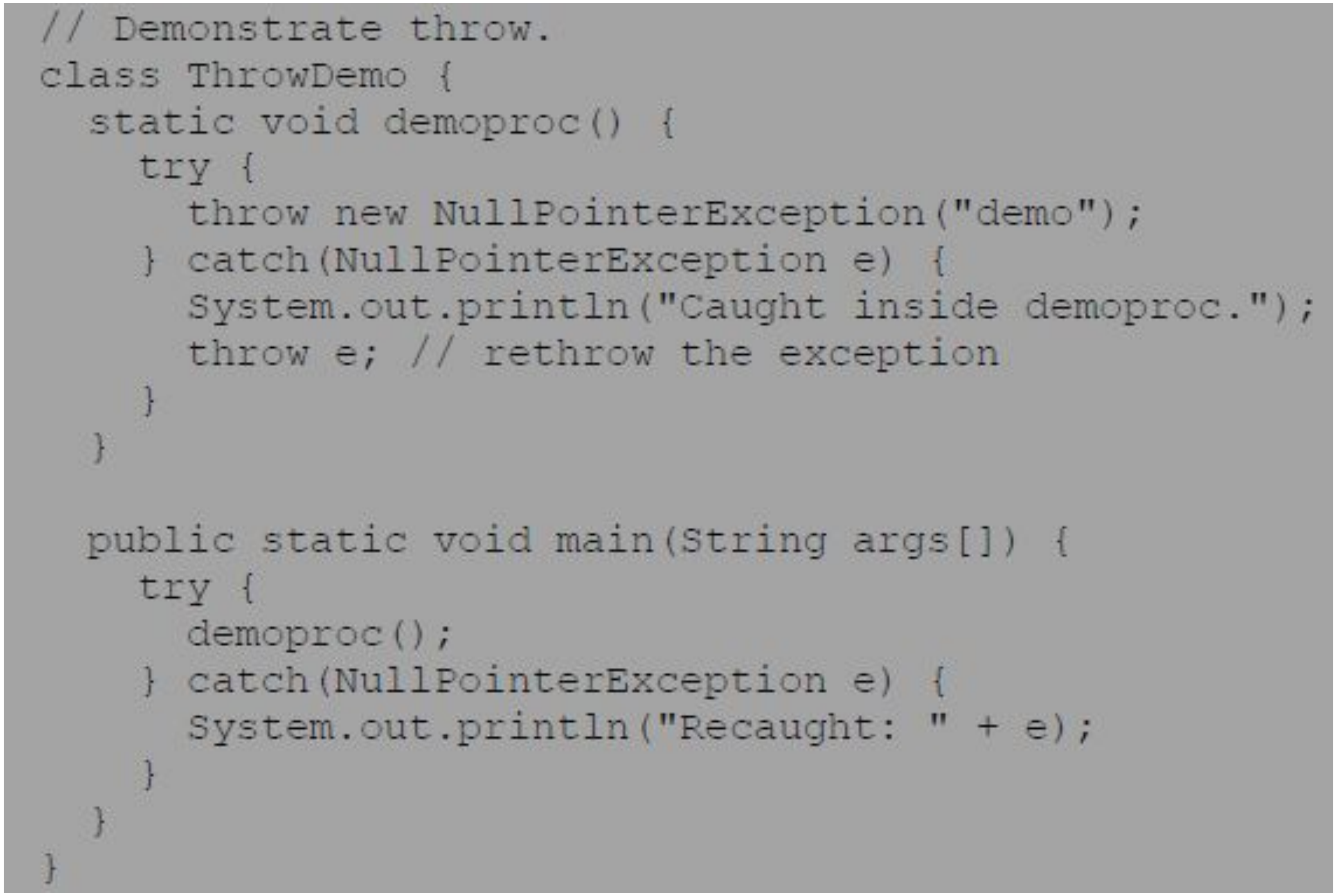
In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message

Output:

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

throw

- Sample program that creates and throws an exception.



```
// 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);
    }
  }
}
```

throw

- 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
```

throw

- 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. This string is displayed when the object is used as an argument to **print( ) or println( ).**

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a **throws clause in the method's declaration.**
- **A throws clause** lists the types of exceptions that a method might throw.

throws

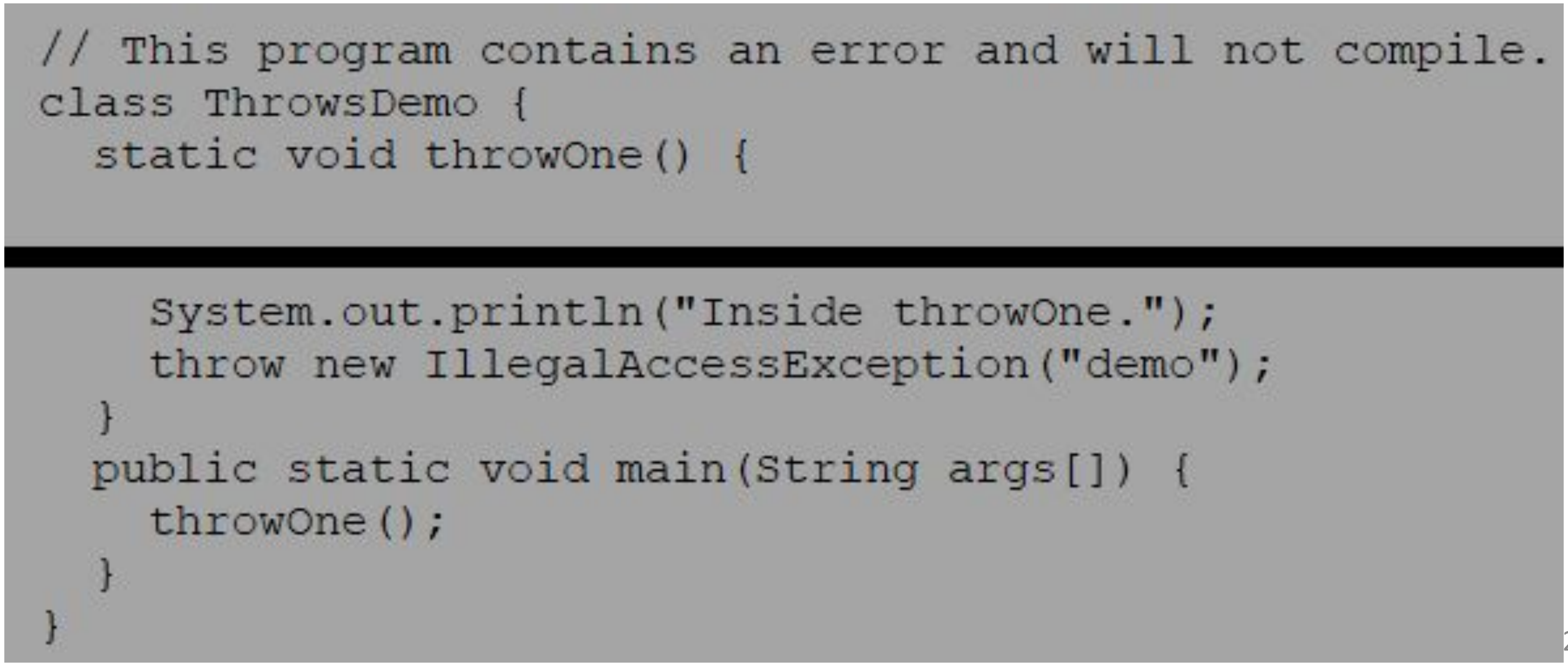
- This is the general form of a method declaration that includes a **throws clause:**

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

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

throws

- 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.

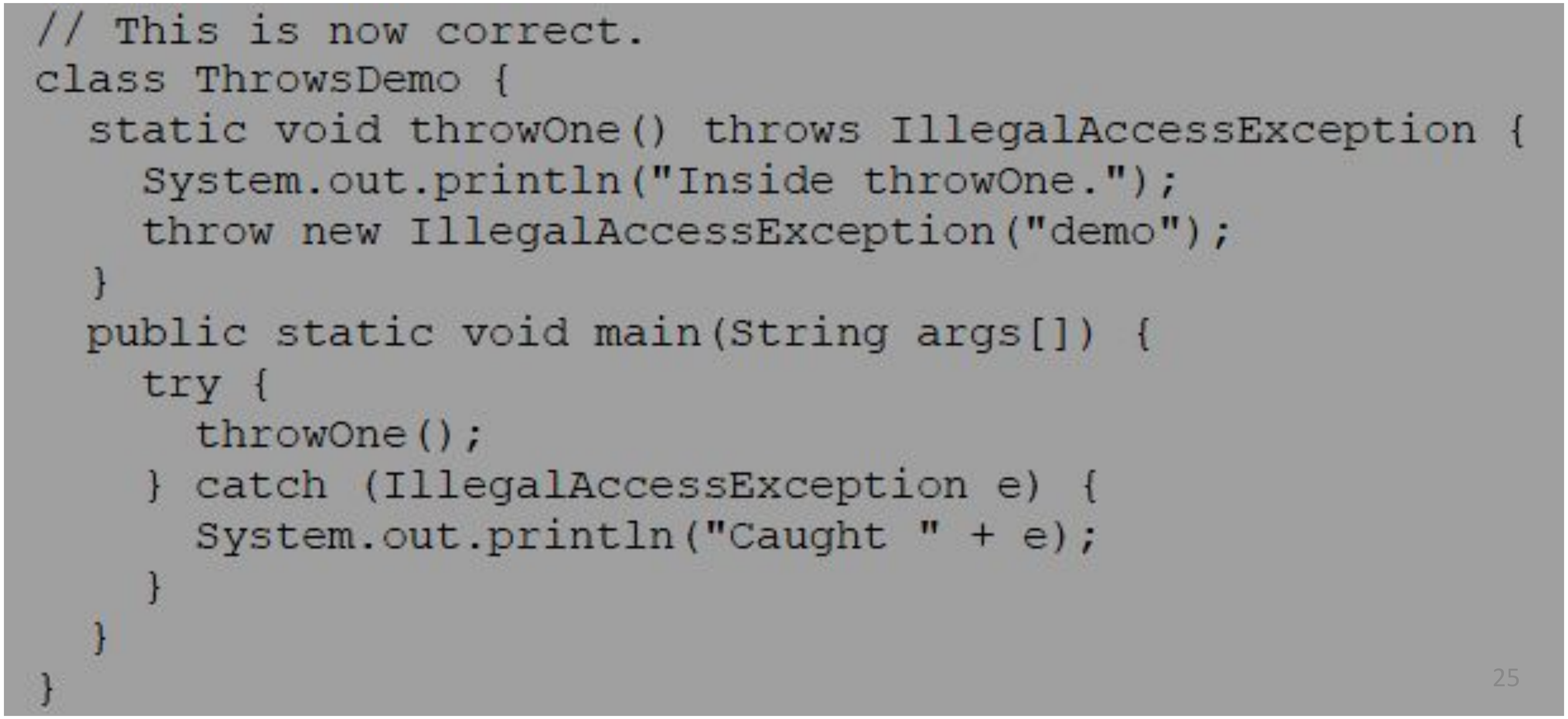


```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {

    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

throws

- To make this example compile, you need to make two changes. First, you need to declare that **throwOne( ) throw IllegalAccessException. Second, main( ) must define** a **try / catch statement that catches this exception.**
- **The corrected example is shown here:**



```
// 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);
    }
  }
}
```

throws

- Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

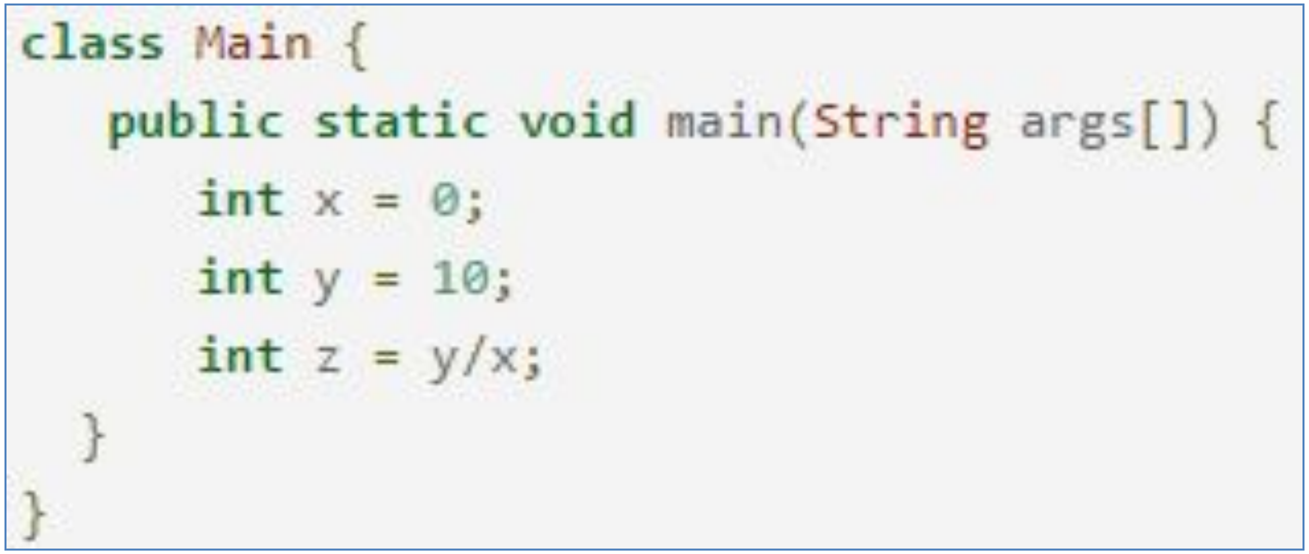
Checked and unchecked exception

- **Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

- **Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
- In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

Checked and unchecked exception

- Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because ***ArithmeticException*** is an unchecked exception.



```
class Main {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

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**.

Java Exception (unchecked) propagation: Example

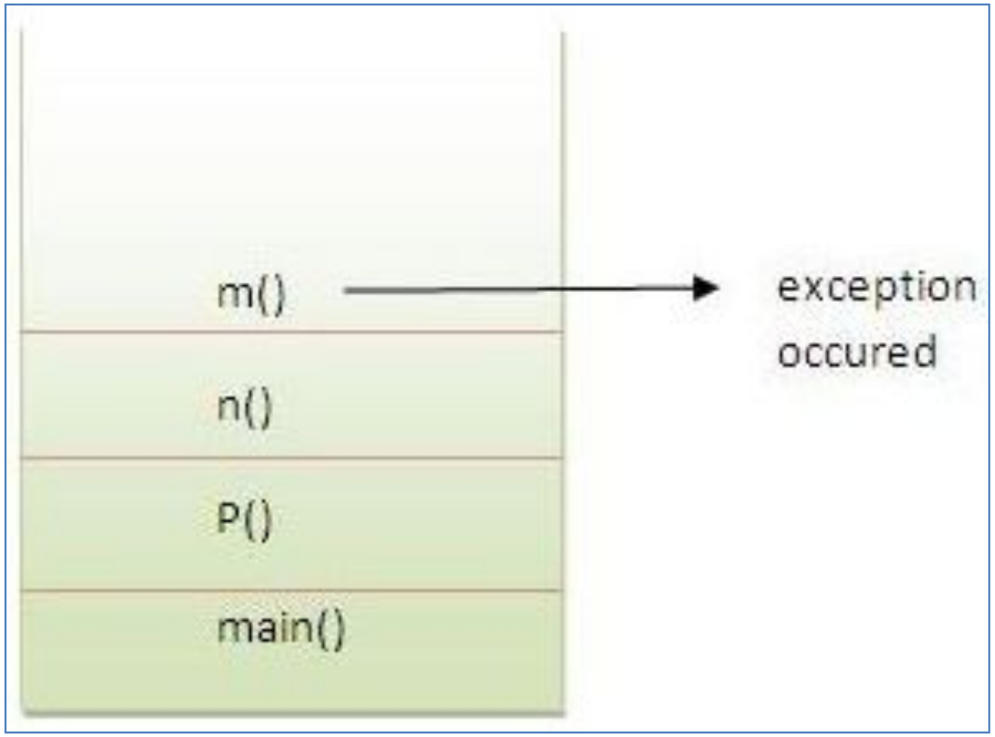
```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();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

Output:

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

Java Exception (unchecked) propagation: Example

- In the above 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.**



Checked exceptions are not propagated

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked excepti
on
  }
  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
```

Java Exception propagation

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

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

Checked Exception can be propagated using throws

- Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

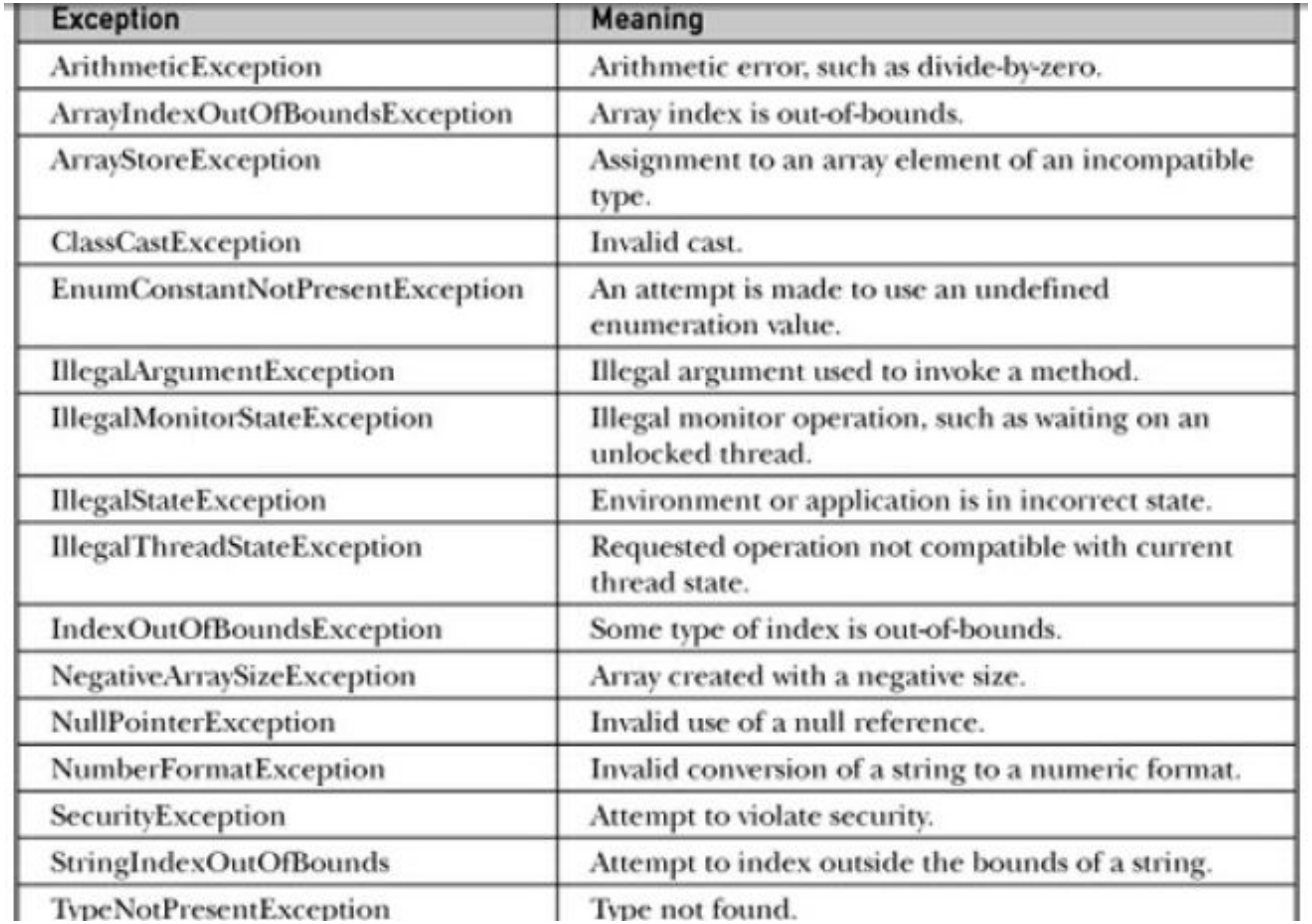
```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static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

Output:

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

Java's Built-in Exceptions

- Inside the standard package **java.lang, Java defines several exception classes.**



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.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.

Thank you