Java Programming

Lecture 11

Exception handling

- 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.

- It brings run-time error management into the object-oriented programming.

- 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.

- Java exception handling is managed via 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.

This is the general form of an exception-handling block:

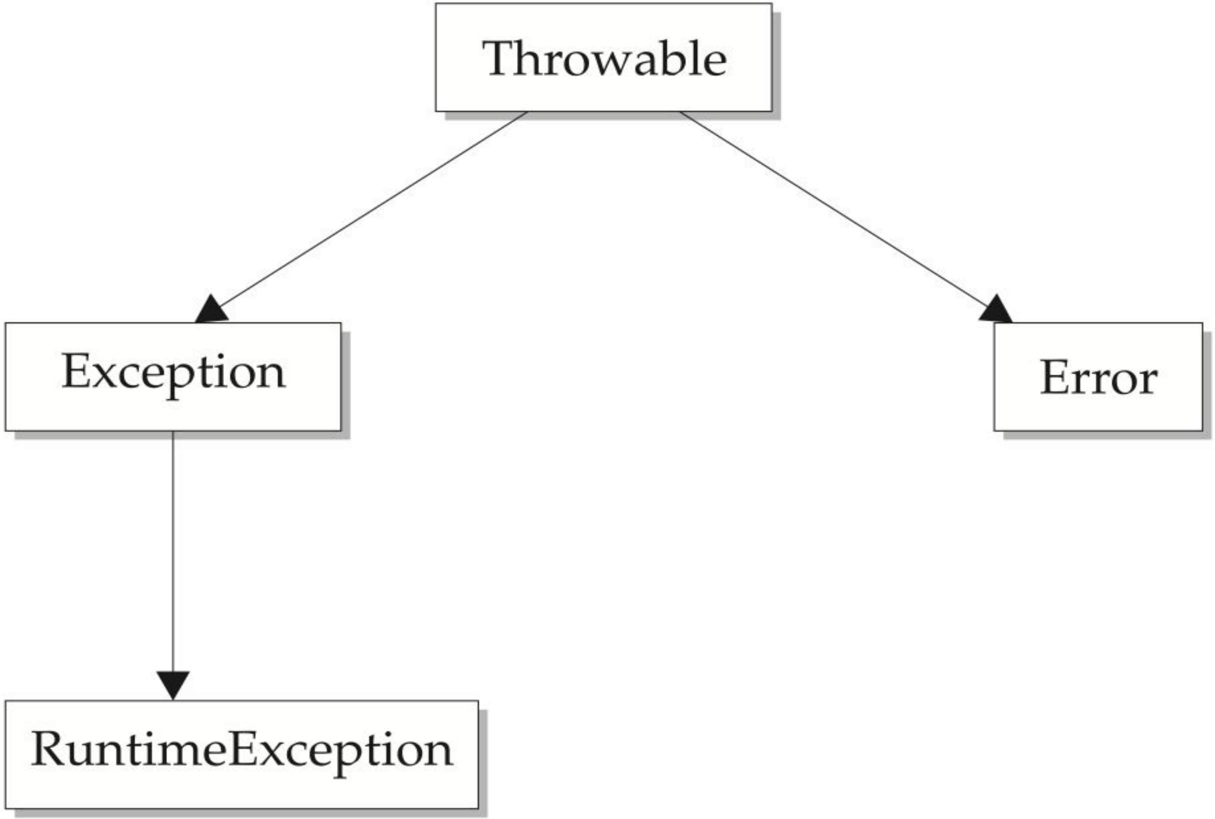
```
try {
    // block of code to monitor for errors
}

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
}
```

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

Exception Types

- All exception types are subclasses of the built-in class **Throwable**.
- Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.

The top-level exception hierarchy is shown here:



- One branch is headed by **Exception**.

- This class is used for exceptional conditions that user programs should catch.

- 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**.

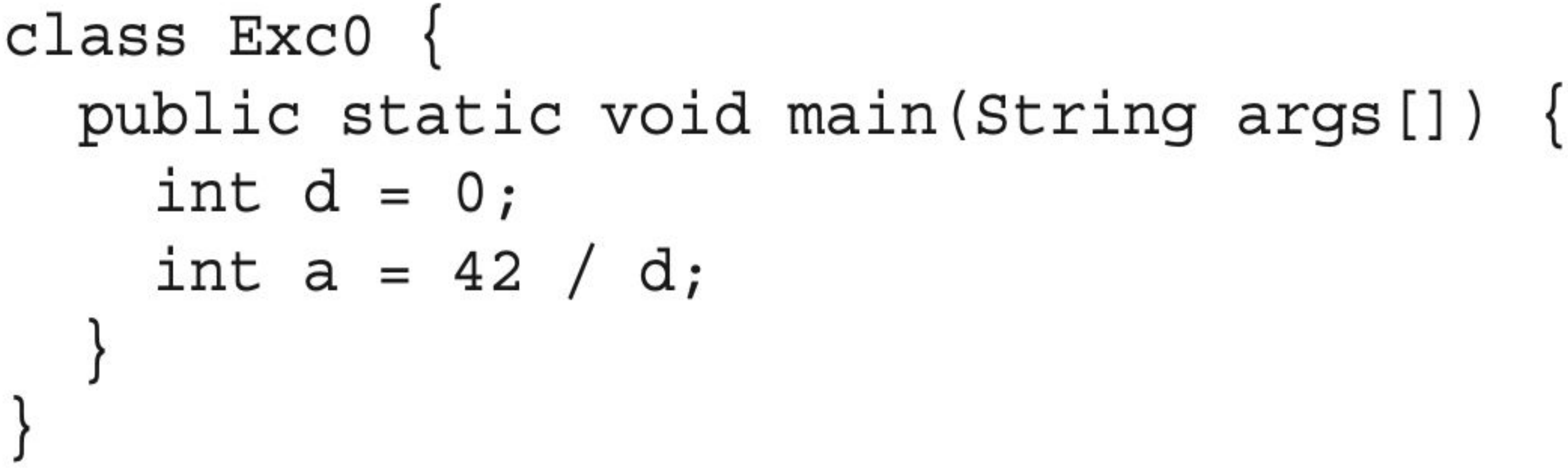
- Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.

- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

- Stack overflow is an example of such an error.

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.

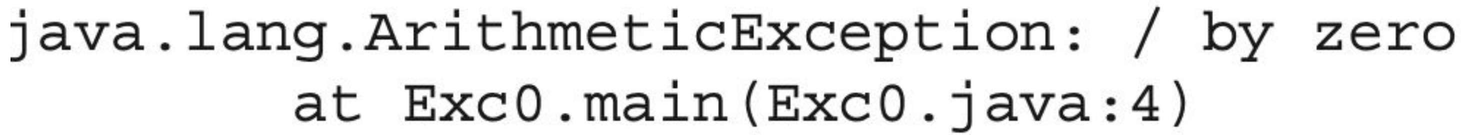
❖ This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.

❖ In this example, we haven’t supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

❖ Any exception that is not caught by your program will ultimately be processed by the default handler.

- ❖ The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

- ❖ Here is the exception generated when this example is executed:



```
java.lang.ArithmeticException: / by zero
        at Exc0.main(Exc0.java:4)
```

- Here, the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace.

- And, the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

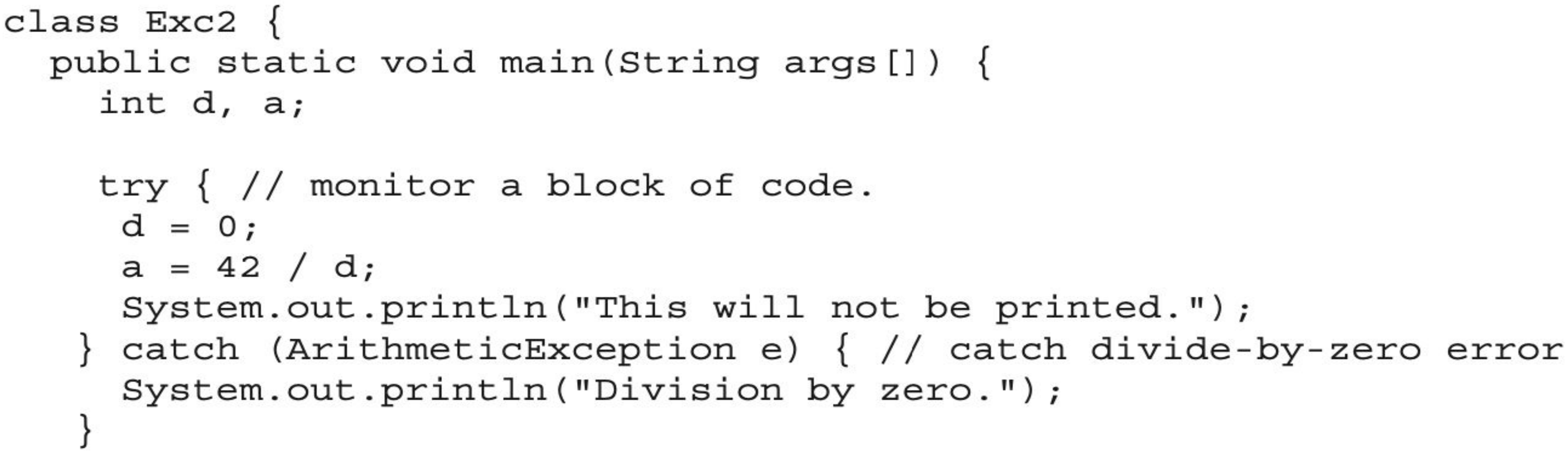
As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

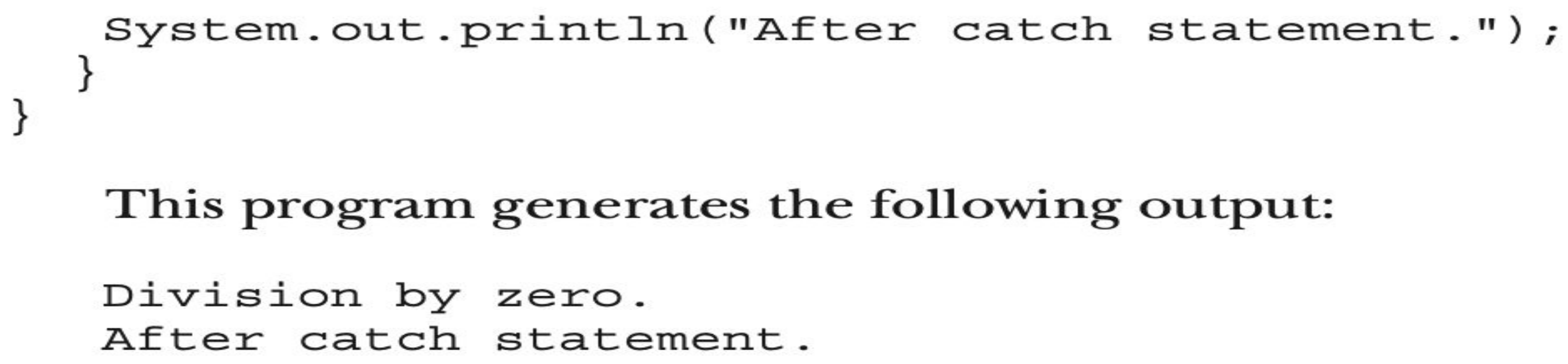
Handling an exception by ourself provides two benefits.

- ❖ First, it allows you to fix the error.
- ❖ Second, it prevents the program from automatically terminating.

Example of try and catch



```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
   } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
   }
```



```
      System.out.println("After catch statement.");
    }
}
```

This program generates the following 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.

- A **try** and its **catch** statement form a unit.

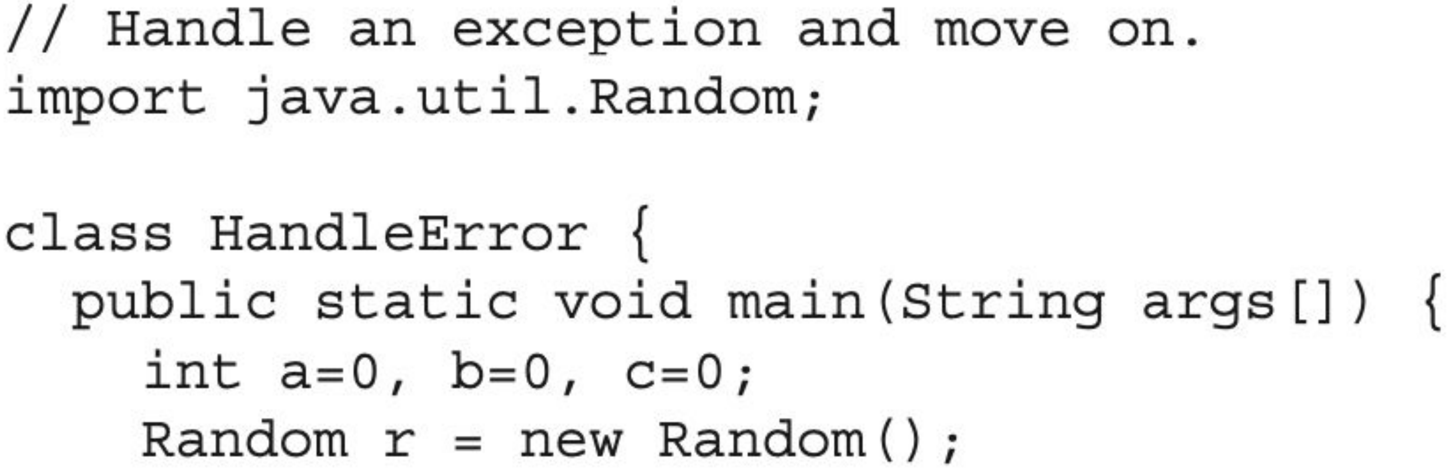
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

- A **catch** statement cannot catch an exception thrown by another **try** statement.

- The statements that are protected by **try** must be surrounded by curly braces.

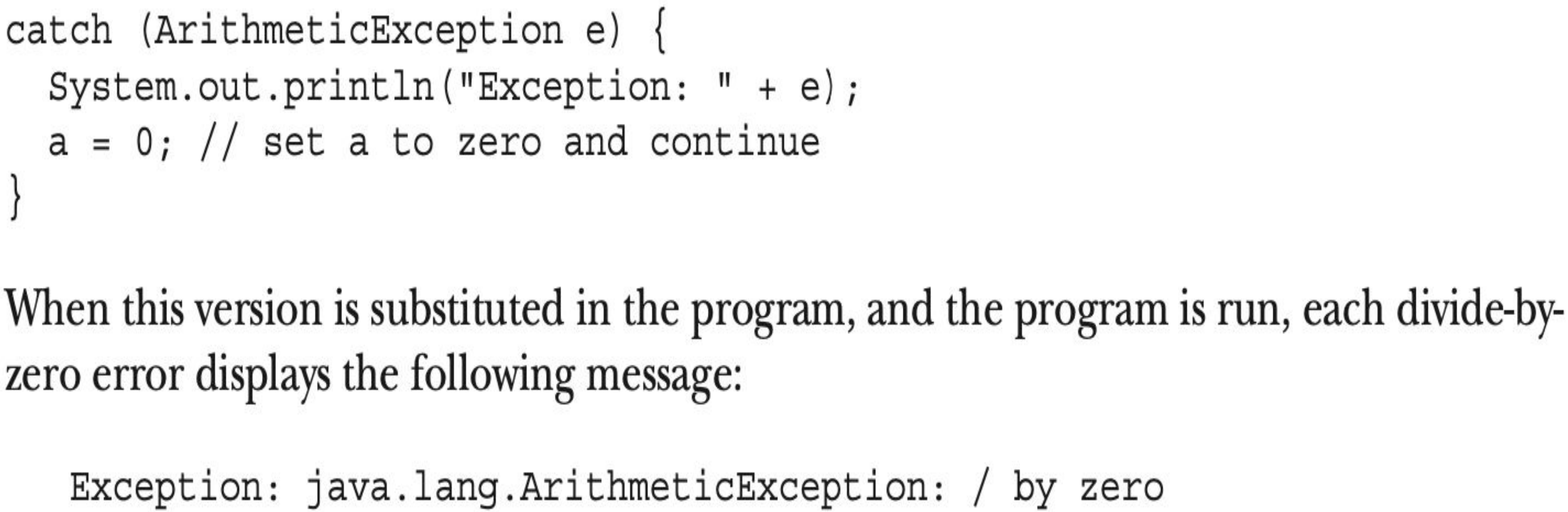
- You cannot use **try** on a single statement.

- The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.



```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

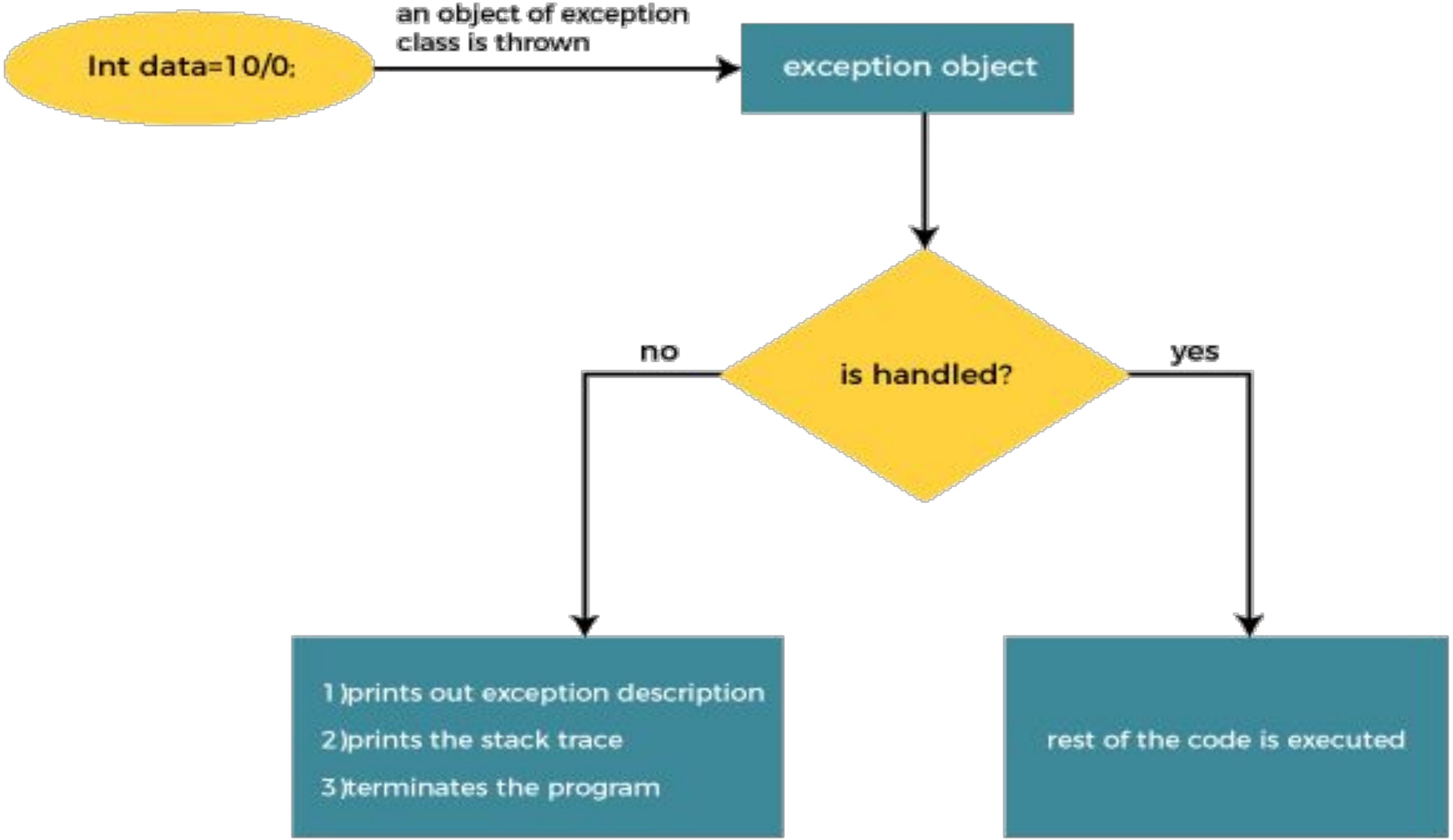
Another version of catch in previous program



```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```


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

Working of Java try-catch block

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.

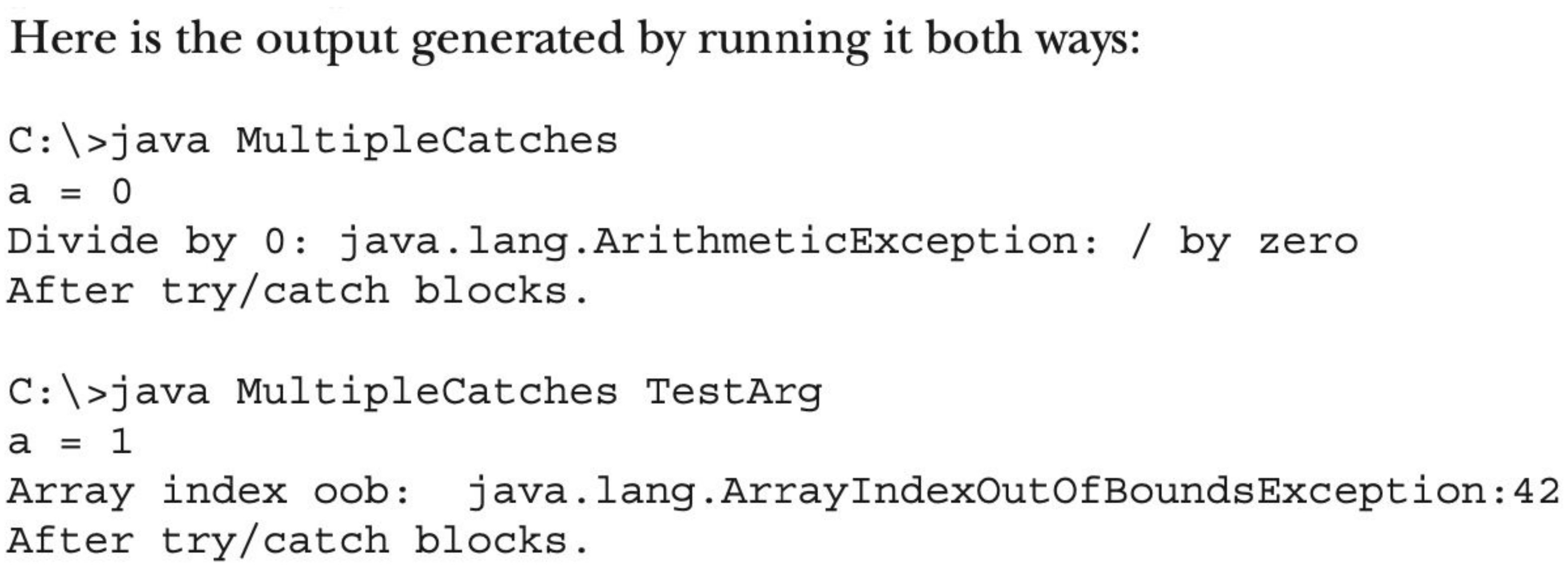
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block.

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

- This program will cause a division-by-zero exception if it is started with no command- line arguments, since **a** will equal zero.

- It will survive the division if you provide a command-line argument, setting **a** to something larger than zero.

- But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.



Here is 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.
```

❖ When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.

- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

- Thus, a subclass would never be reached if it came after its superclass.

- In Java, unreachable code is an error.

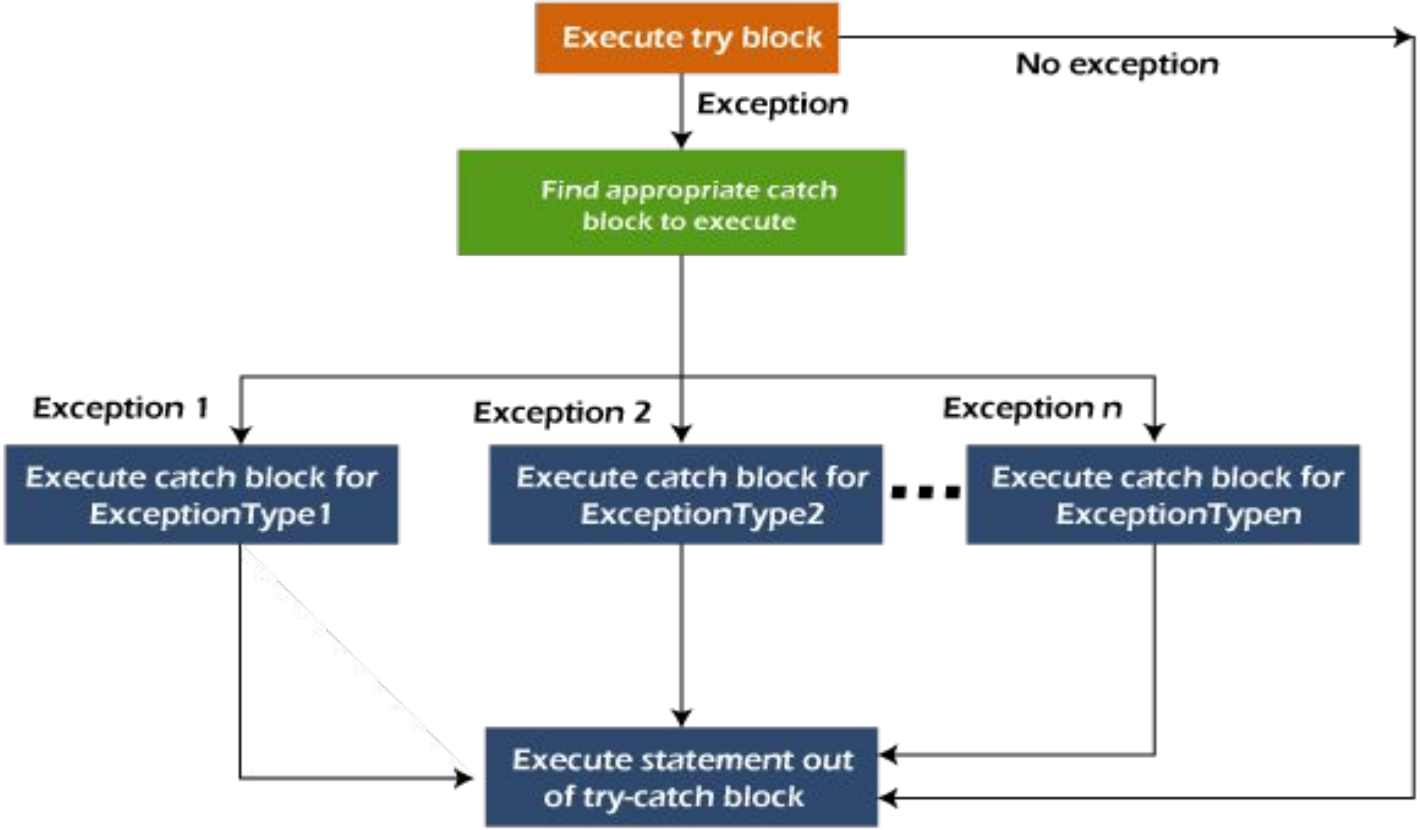
```
/* This program contains an error.

   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.");
    }
  }
}
```

- If you try to compile previous 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.


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
...
Execute catch block for ExceptionTypen
Execute statement out of try-catch block

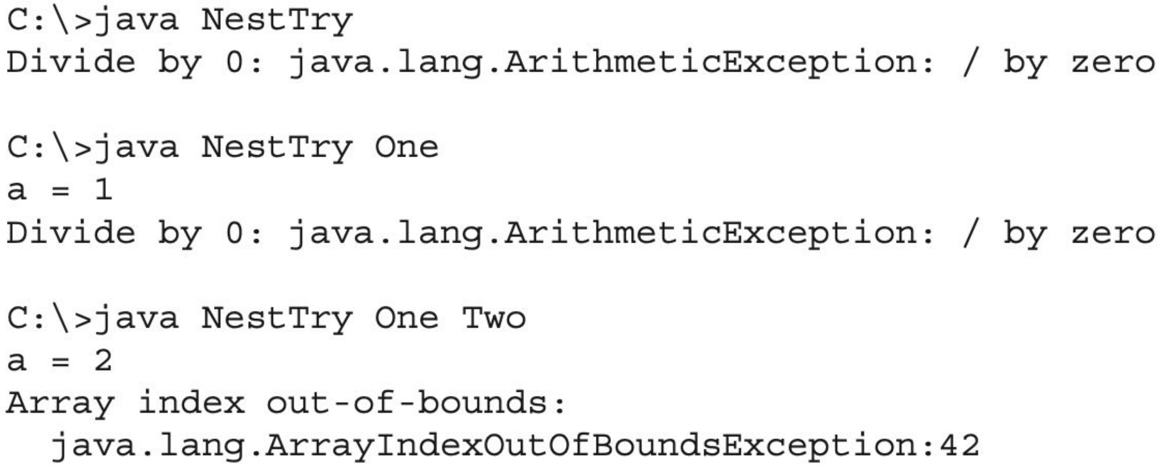
Working of Multi-catch Block

Nested try Statements

- The **try** statement can be nested.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement’s **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command-line args are present,
          the following statement will generate
          a divide-by-zero exception. */
      int b = 42 / a;

       System.out.println("a = " + a);

       try { // nested try block
         /* If one command-line arg is used,
            then a divide-by-zero exception
            will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command-line args are used,
           then generate an out-of-bounds exception. */
        if(a==2) {
           int c[] = { 1 };
           c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

Output:



```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:42
```

throw

- It is possible for your program to throw an exception explicitly, using the **throw** statement.

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

  throw *ThrowableInstance*;

- 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.

❖ 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 stack trace.

```
// 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
  ```

- We can also define our own set of conditions and throw an exception explicitly using throw keyword.

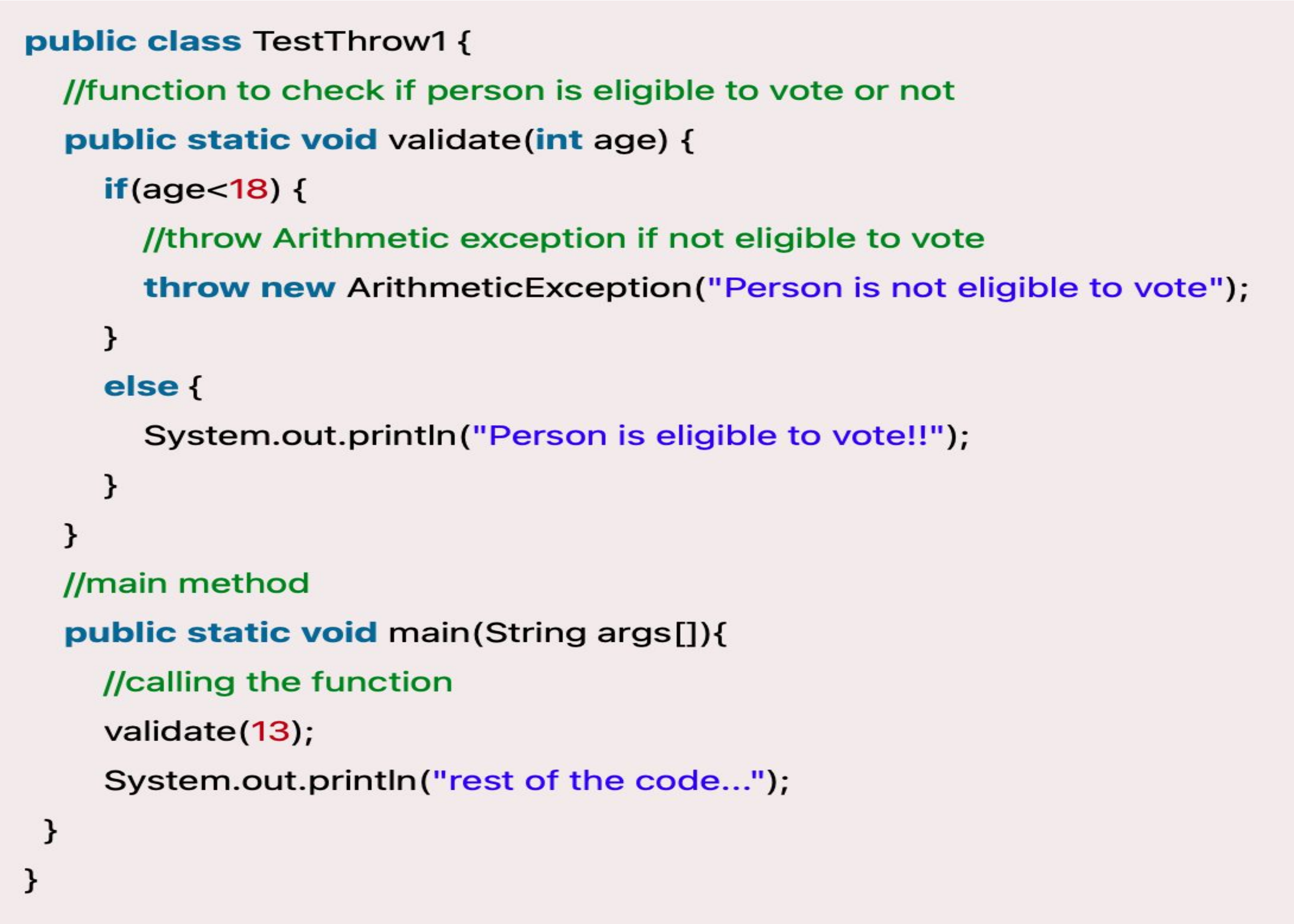
- For example, we can throw ArithmeticException if we divide a number by another number.

- Here, we just need to set the condition and throw exception using throw keyword.

Example:

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

Throwing Unchecked Exception Example:

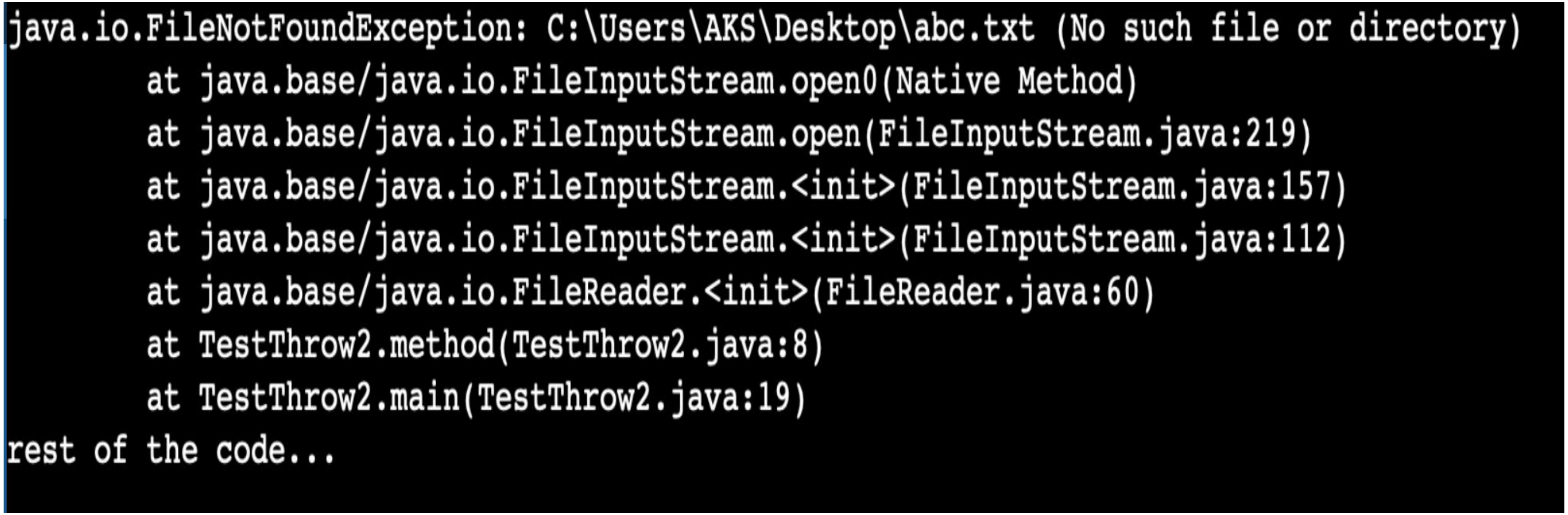


```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
   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

Throwing Checked Exception

```java
import java.io.*;
public class TestThrow2 {
    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {
        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);
        throw new FileNotFoundException();
    }
```

```
 public static void main(String args[]){
    try
    {
      method();
    }
    catch (FileNotFoundException e)
    {
      e.printStackTrace();
    }
    System.out.println("rest of the code...");
 }
}
```



```
java.io.FileNotFoundException: C:\Users\AKS\Desktop\abc.txt (No such file or directory)
	at java.base/java.io.FileInputStream.open0(Native Method)
	at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
	at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
	at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
	at java.base/java.io.FileReader.<init>(FileReader.java:60)
	at TestThrow2.method(TestThrow2.java:8)
	at TestThrow2.main(TestThrow2.java:19)
rest of the code...
```

User Defined Exception

```java
// class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        // Calling constructor of parent Exception
        super(str);
    }
}
```

```
public class TestThrow3
{
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            System.out.println("Caught the exception");
            // Print the message from MyException object
            System.out.println(ude.getMessage());
        } } }
```

Output:

```
Caught the exception
This is user-defined exception
```

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.

- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.

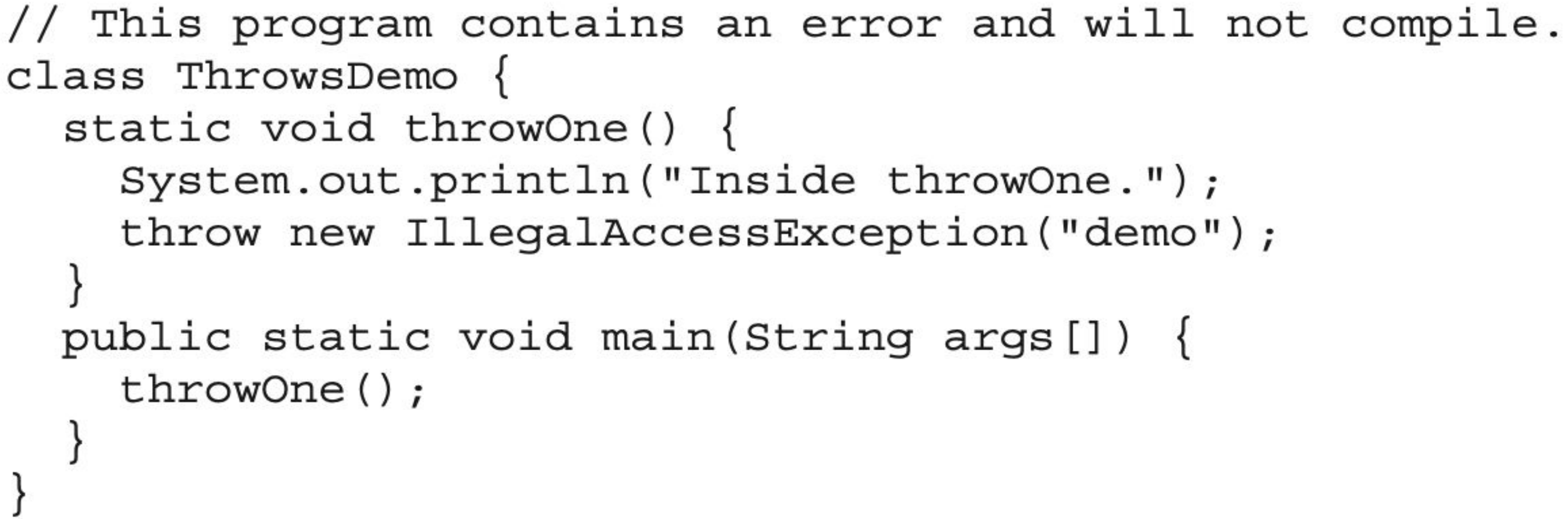
- All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

- 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-list* is a comma-separated list of the exceptions that a method can throw.



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

Here is the output generated by running this example program:

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

finally

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.

- The **finally** block will execute whether or not an exception is thrown.

- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

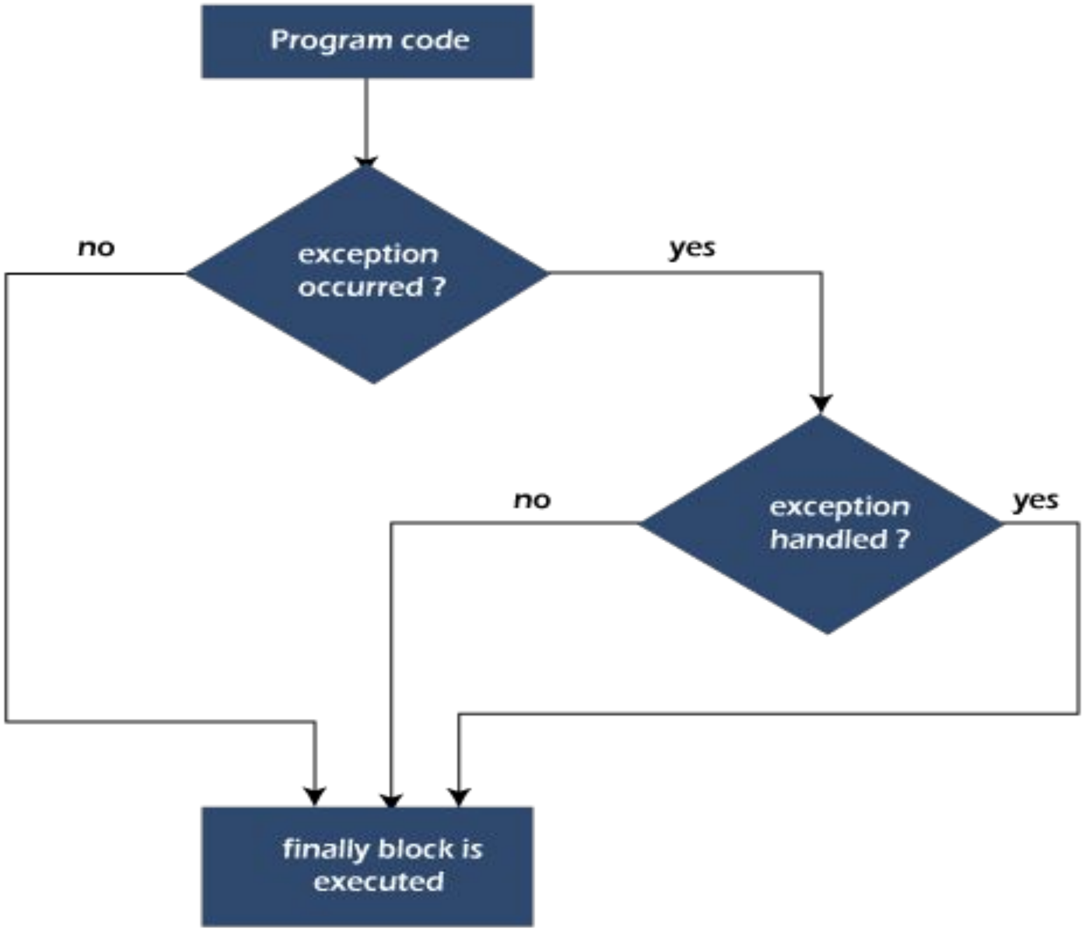
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- **finally** block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.

- The **finally** clause is optional.

- However, each **try** statement requires at least one **catch** or a **finally** clause.

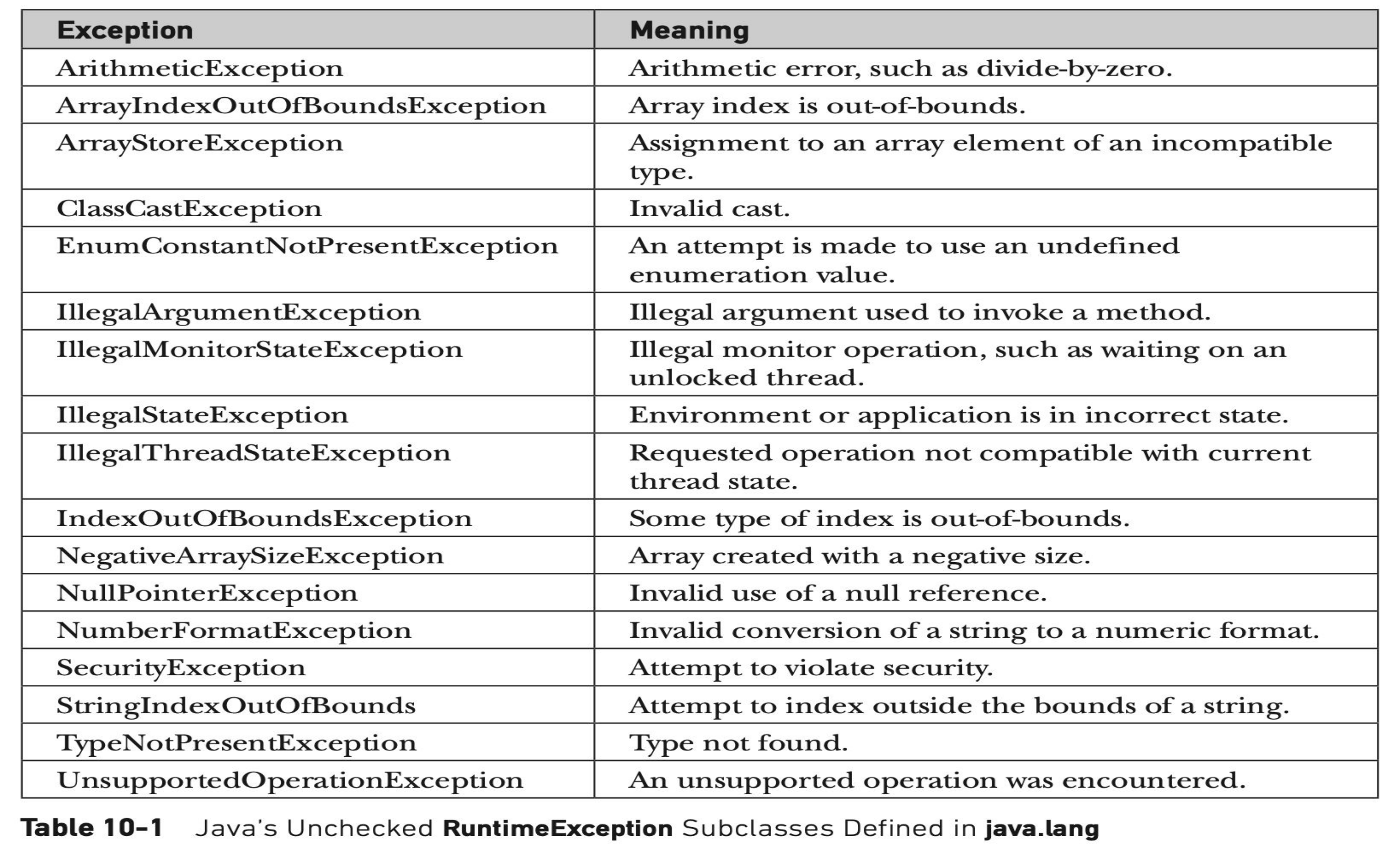


Working of finally block

```
// 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");
    }
  }
  // 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();
  }
}
```

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
```

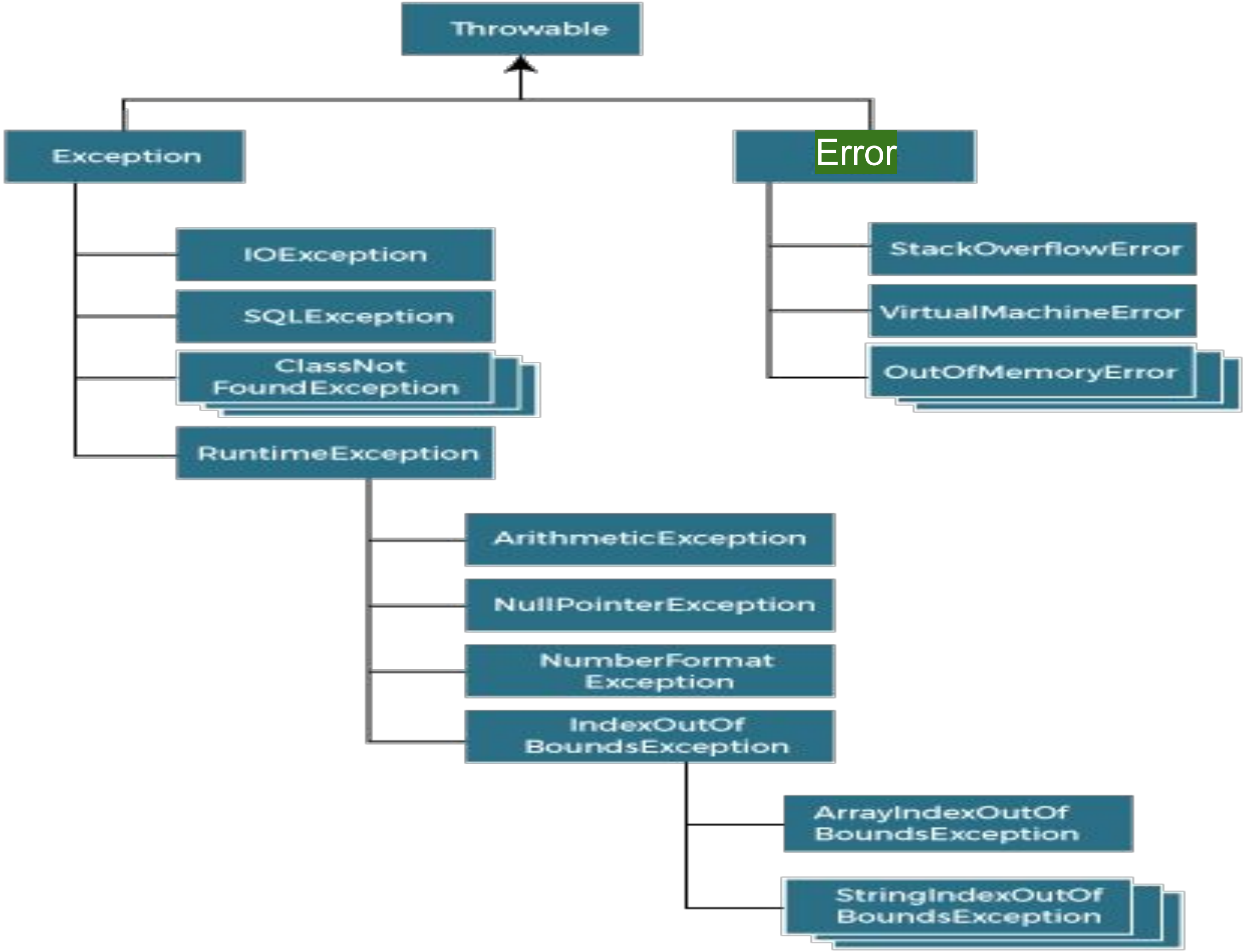


Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made 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.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined 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.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**


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

Thank you