UNIT - IV

Exception-Handling

&

Multithreading

Exception-Handling and Multithreading

Exception-Handling: Exception handling basics, different types of exception classes, use of try & catch with throw, throws & finally, creation of user defined exception classes.

Multithreading: *Basics of multithreading, main thread, thread life cycle, creation of multiple threads, thread priorities, thread synchronization, inter-thread communication, deadlocks for threads, suspending & resuming threads.*

Exception-Handling

- Basics
- Different types of exception classes
- Use of try & catch with throw
- throws & finally
- Creation of user defined exception classes

Exception-Handling

Basics:

The three categories of errors

- ***Syntax errors*** arise because the rules of the language have not been followed. They are detected by the compiler.
- ***Runtime errors*** occur while the program is running if the environment detects an operation that is impossible to carry out.
- ***Logic errors*** occur when a program doesn't perform the way it was intended to.

Exception-Handling

Basics: ...

- 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.***
- In computer languages that do not support exception handling, errors must be checked and handled manually - typically through the use of error codes, and so on.
- 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.

Exception-Handling

Basics: ...

- Exceptions can be generated by the Java run-time system, or they can be manually generated by our 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**.

Exception-Handling

Basics: ...

- 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. Our 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.

Exception-Handling

Basics: ...

- 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

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.

- 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 our 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.

Exception-Handling

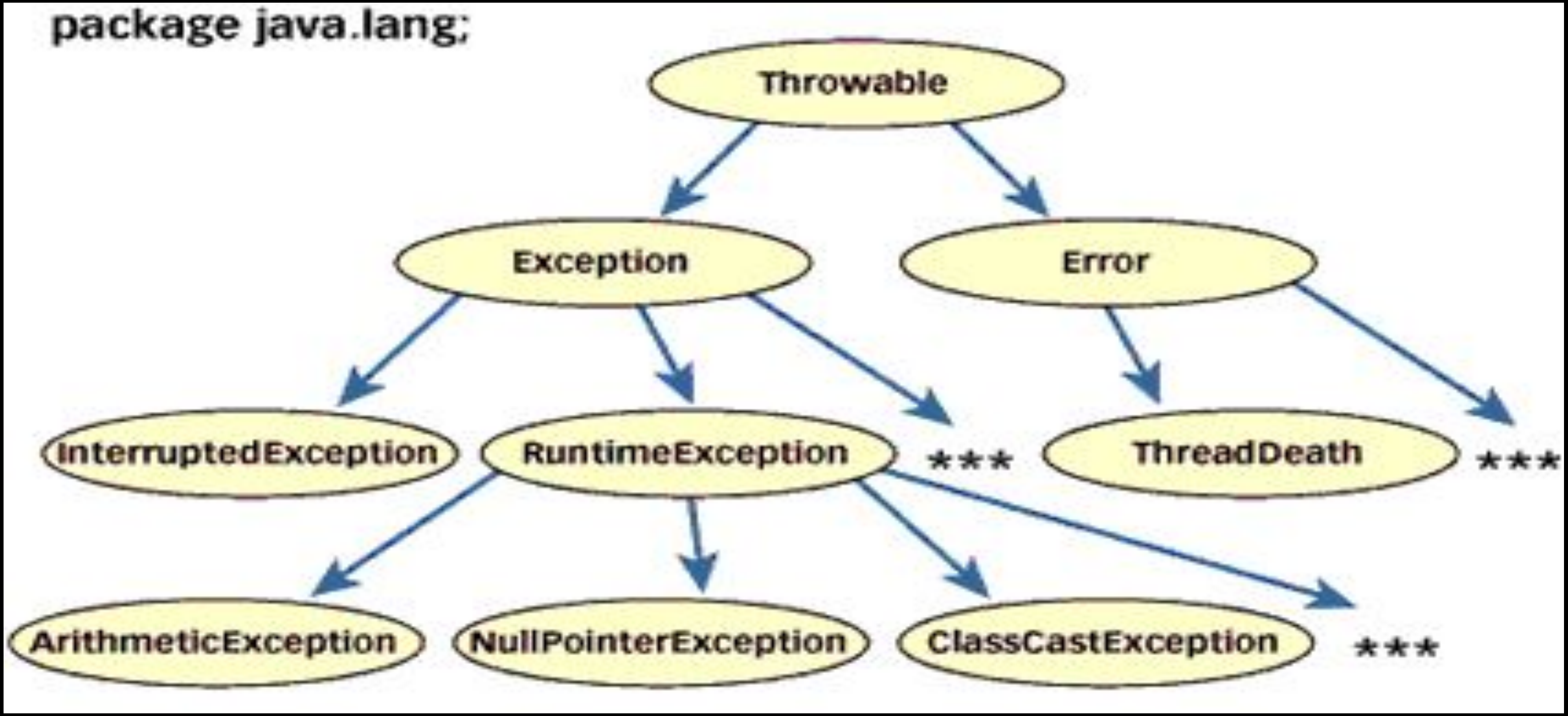
Exception Types: ...

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by our 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.
- This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to **catastrophic** failures that cannot usually be handled by our program.

Exception-Handling

Exception Types: ...

- *Every Exception type is basically an object belonging to class* ***Exception***
- ***Throwable*** *class is the root class of Exceptions.*
- ***Throwable*** *class has two direct subclasses named* ***Exception, Error***



Exception-Handling

Exception Types: ...

Checked Exceptions

•All Exceptions that extends the Exception or any one its subclass except RunTimeException class are checked exceptions.

- Checked Exceptions are checked by the Java compiler.
- There are two approaches that a user can follow to deal with checked exceptions.

Inform the compiler that a method can throw an Exception.

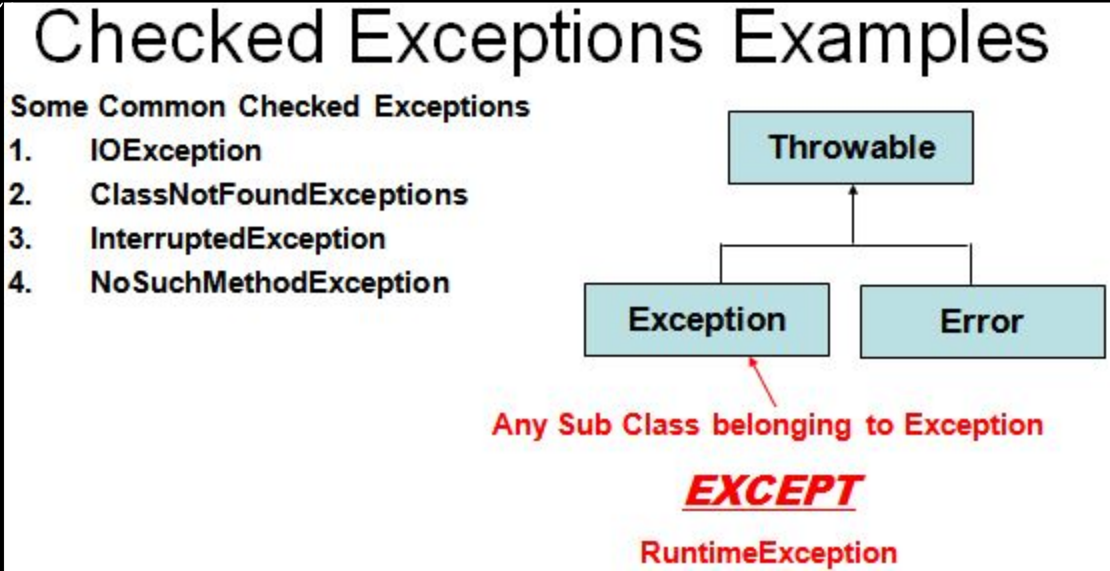
Catch the checked exception in try catch block.

Exception-Handling

Exception Types: ...

Checked Exceptions…

- If Checked exception is caught then exception handling code will be executed and program's execution continues.
- If Checked exception is not caught then java interpreter will provide the default handler. But in this case execution of the program will be stopped by displaying the name of the exceptions object.



Checked Exceptions Examples

Some Common Checked Exceptions

1. IOException
2. ClassNotFoundExceptions
3. InterruptedException
4. NoSuchMethodException

Exception-Handling

Exception Types: ...

Unchecked Exceptions

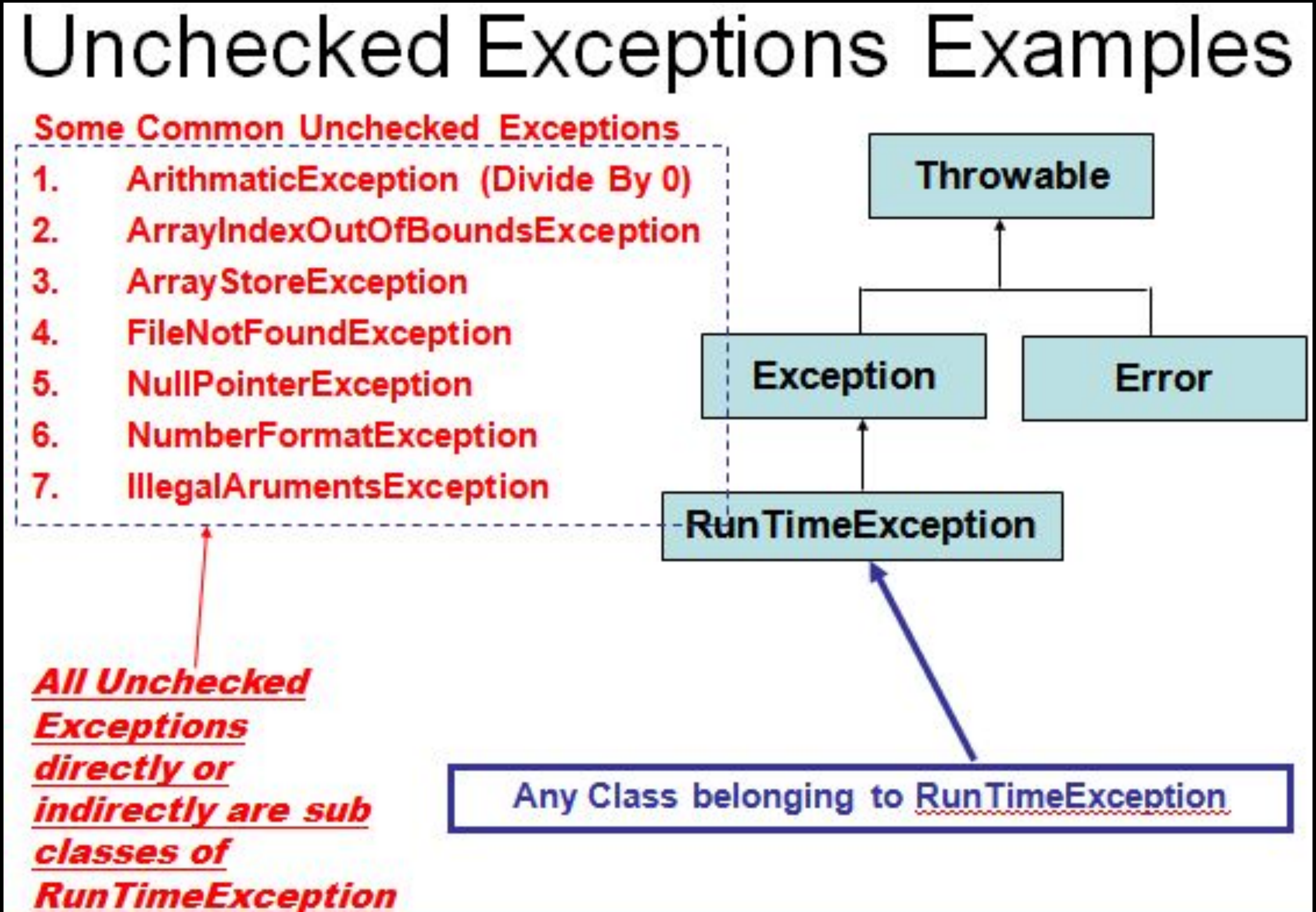
•All Exceptions that extend the RuntimeException or any one of its subclass are unchecked exceptions.

- Unchecked Exceptions are unchecked by compiler.
- Whether you catch the exception or not compiler will pass the compilation process.
- If Unchecked exception is caught then exception handling code will be executed and program's execution continues.
- If Unchecked exception is not caught then java interpreter will provide the default handler. But in this case execution of the program will be stopped by displaying the name of the exceptions object.

Exception-Handling

Exception Types: ...

Unchecked Exceptions…


Unchecked Exceptions Examples

Some Common Unchecked Exceptions

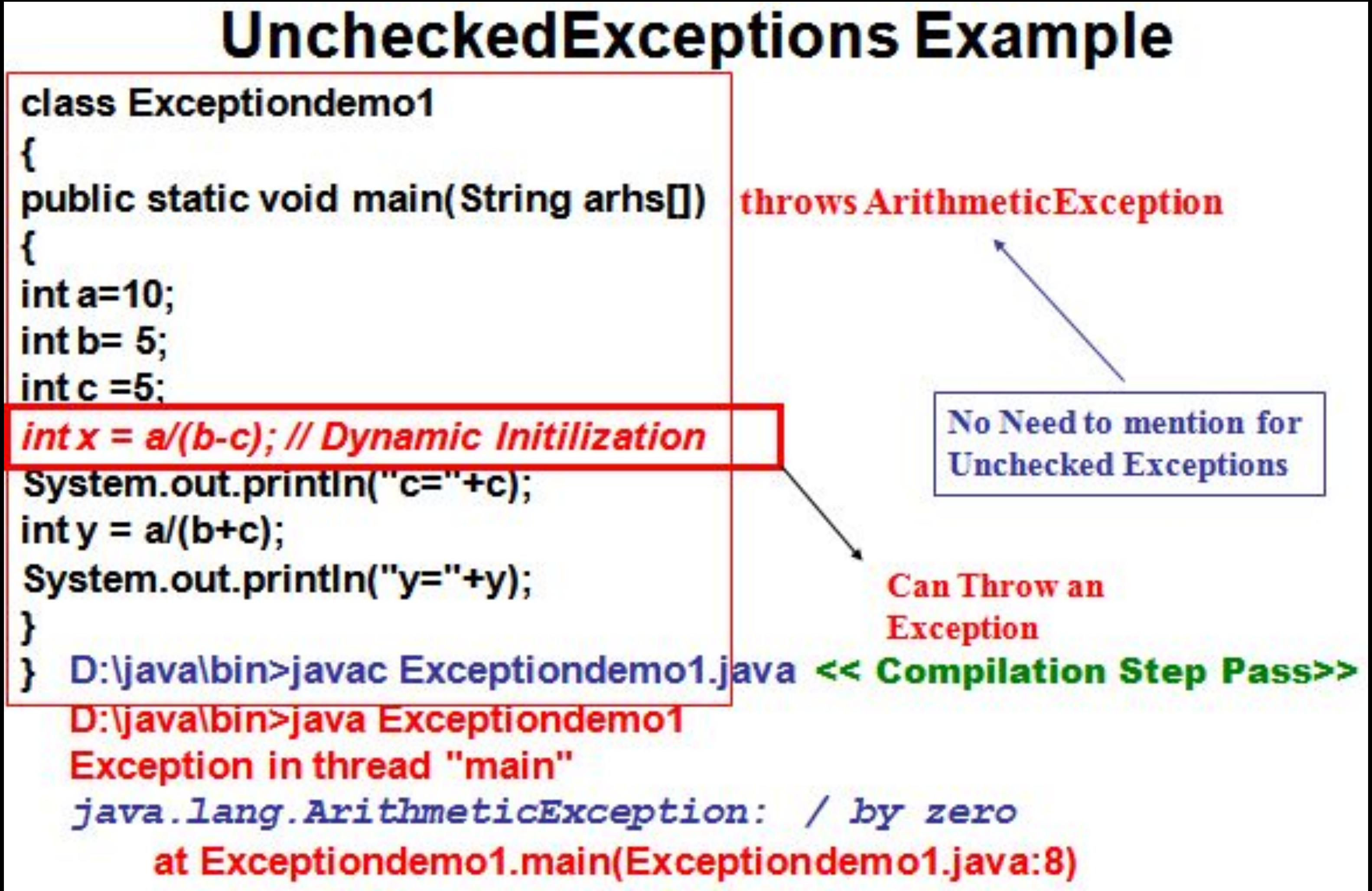
1. ArithmaticException (Divide By 0)
2. ArrayIndexOutOfBoundsException
3. ArrayStoreException
4. FileNotFoundException
5. NullPointerException
6. NumberFormatException
7. IllegalArumentsException

All Unchecked Exceptions directly or indirectly are sub classes of RunTimeException

Exception-Handling

Exception Types: ...

Unchecked Exceptions…



UncheckedExceptions Example

```
class Exceptiondemo1
{
public static void main(String arhs[])
{
int a=10;
int b= 5;
int c =5;
int x = a/(b-c); // Dynamic Initilization
System.out.println("c="+c);
int y = a/(b+c);
System.out.println("y="+y);
}
}
```

throws ArithmeticException

No Need to mention for Unchecked Exceptions

Can Throw an Exception

```
D:\java\bin>javac Exceptiondemo1.java
```

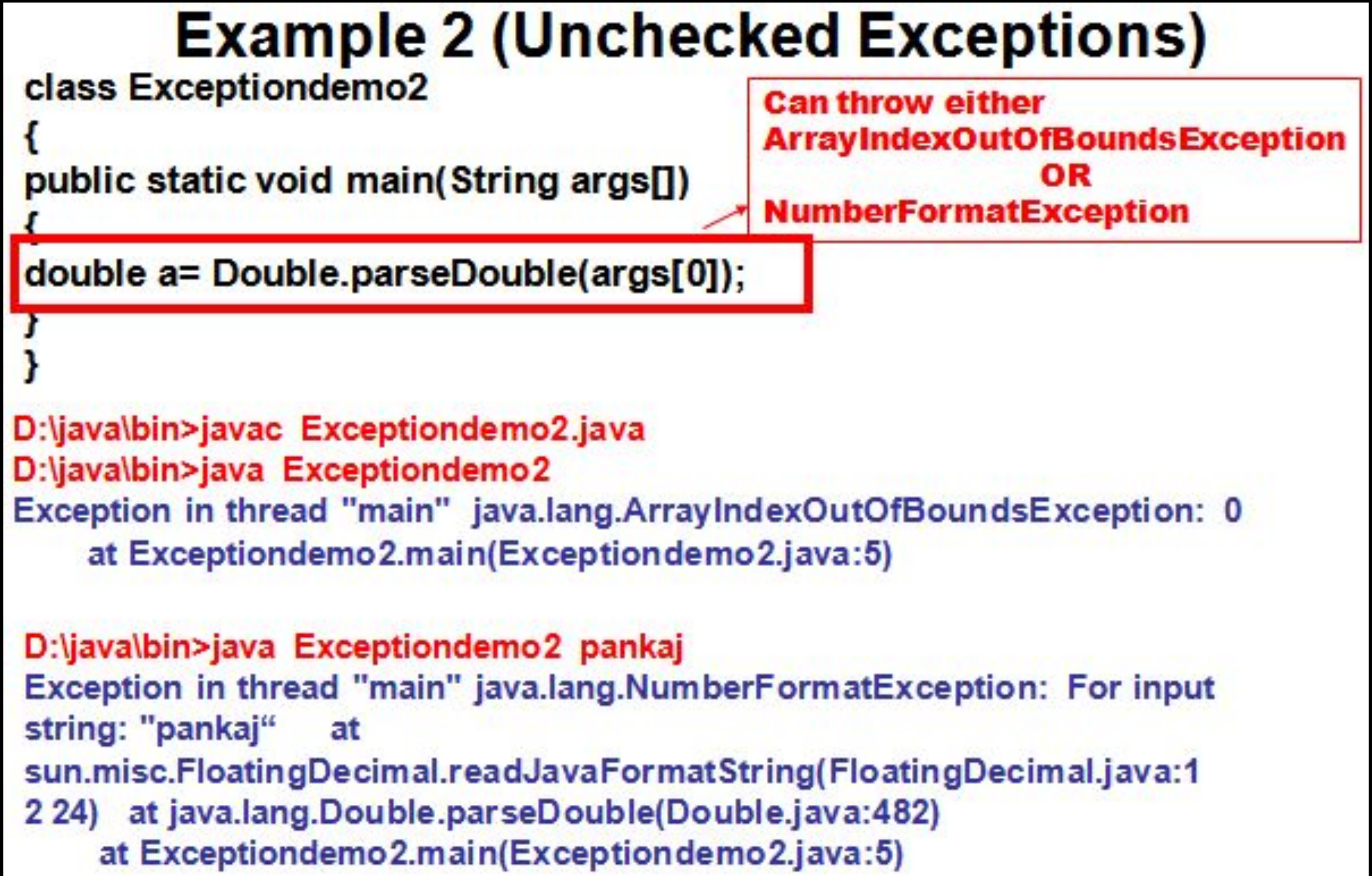
<< Compilation Step Pass>>

```
D:\java\bin>java Exceptiondemo1
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Exceptiondemo1.main(Exceptiondemo1.java:8)
```

Exception-Handling

Exception Types: ...

Unchecked Exceptions…



Example 2 (Unchecked Exceptions)

```
class Exceptiondemo2
{
public static void main(String args[])
{
double a= Double.parseDouble(args[0]);
}
}
```

Can throw either ArrayIndexOutOfBoundsException OR NumberFormatException

```
D:\java\bin>javac  Exceptiondemo2.java
D:\java\bin>java  Exceptiondemo2
Exception in thread "main"  java.lang.ArrayIndexOutOfBoundsException:  0
      at Exceptiondemo2.main(Exceptiondemo2.java:5)

D:\java\bin>java  Exceptiondemo2  pankaj
Exception in thread "main" java.lang.NumberFormatException:  For input
string: "pankaj“     at
sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1
2 24)   at java.lang.Double.parseDouble(Double.java:482)
      at Exceptiondemo2.main(Exceptiondemo2.java:5)
```

Exception-Handling

Exception Types: ...

Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Exception-Handling

Exception Types: ...

Checked Exceptions vs. Unchecked Exceptions…

- Exceptions which are checked for during compile time are called checked exceptions. **Eg:** SQLException or any userdefined exception extending the Exception class.
- Exceptions which are not checked for during compile time are called unchecked exception. **Eg:** NullPointerException or any class extending the RuntimeException class.

- All the checked exceptions must be handled in the program.
- The exceptions raised, if not handled will be handled by the Java Virtual Machine. The Virtual machine will print the stack trace of the exception indicating the stack of exception and the line where it was caused.

Exception-Handling

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.***

Exception-Handling

Uncaught Exceptions: ...

- Any exception that is not caught by our 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)
```

Exception-Handling

Uncaught Exceptions: ...

- The stack trace will always show the sequence of method invocations that led up to the error.

```
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)
```

Exception-Handling

Using try and catch:

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception ourself.
- Doing so provides two benefits:

First, it allows you to fix the error.

Second, it prevents the program from automatically terminating.

Exception-Handling

Using try and catch: ...

- 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.

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

Program output:

```
Division by zero.
After catch statement.
```

Exception-Handling

Using try and catch: ...

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

Exception-Handling

Using try and catch: ...

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

For example, in this program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

Exception-Handling

Using try and catch: ...

Displaying a Description of an Exception

- You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
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
```

Exception-Handling

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.

Exception-Handling

Multiple catch Clauses: ...

- **Eg:** To trap two different exceptions

```
// Demonstrate multiple catch statements.
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.");
        }
}
```

The output generated by running it both ways:

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

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

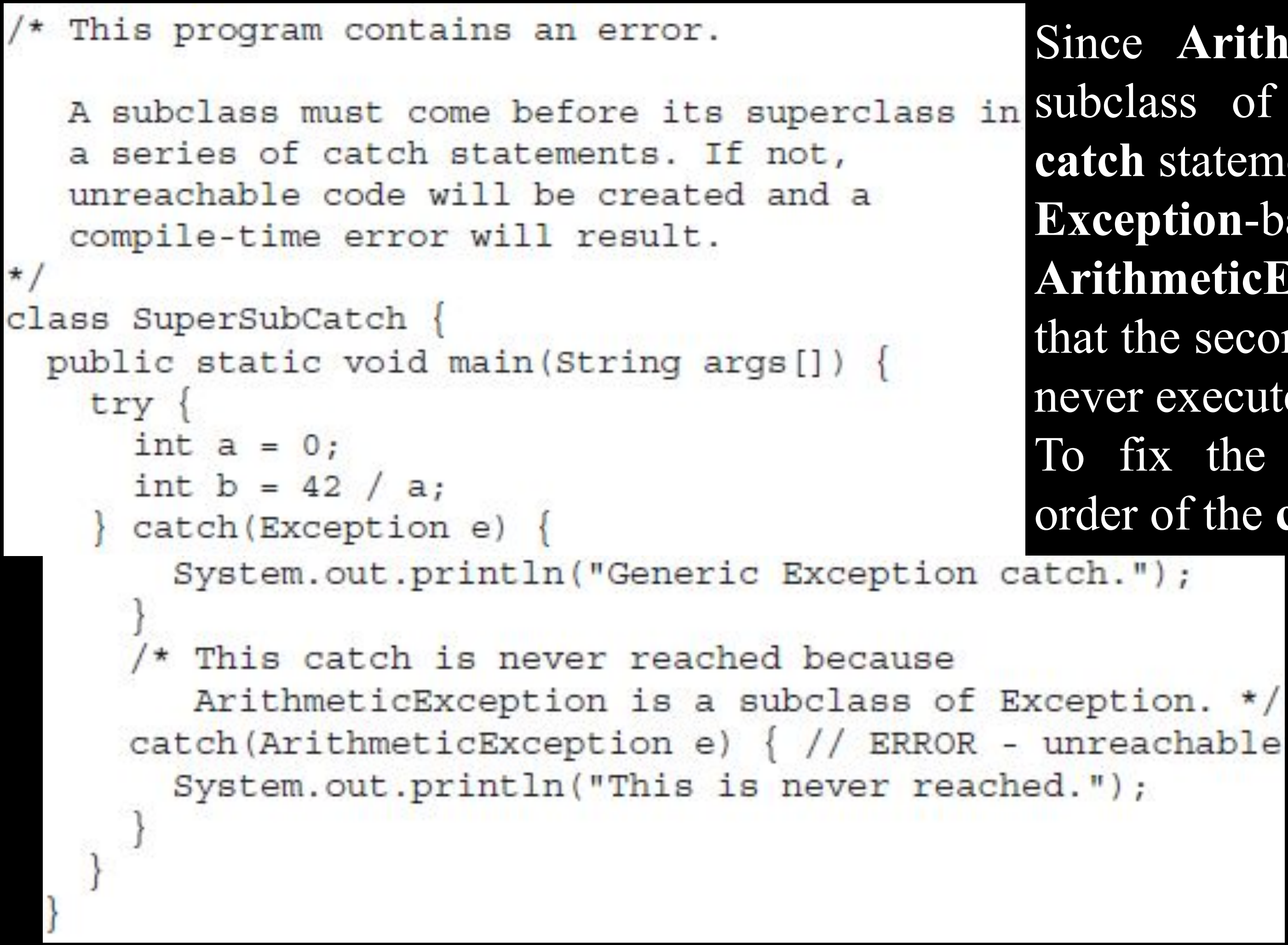
Exception-Handling

Multiple catch Clauses: ...

- 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.
- Further, in Java, unreachable code is an error.

Exception-Handling

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.");
    }
  }
}
```

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.

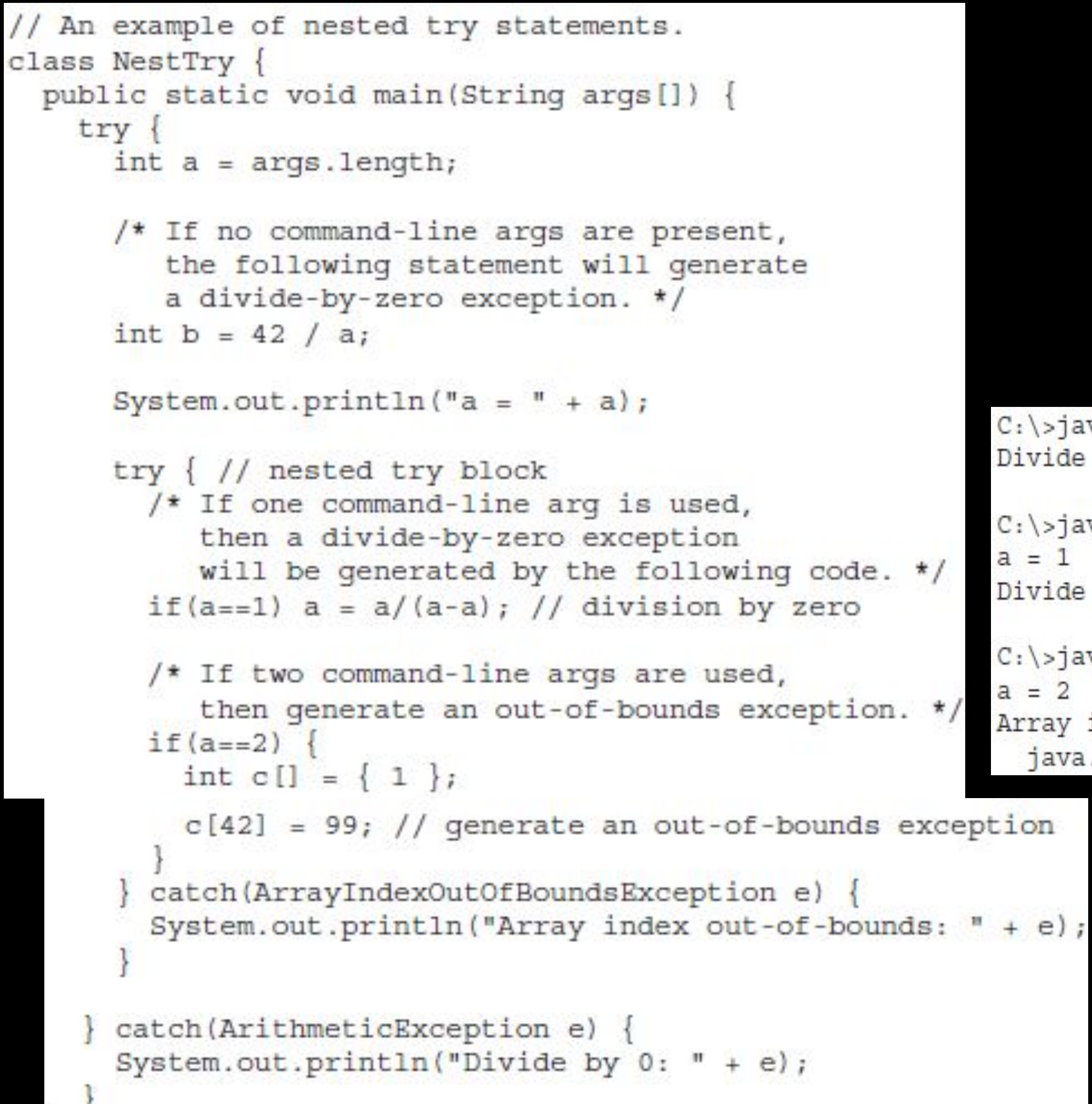
Exception-Handling

Nested try Statements:

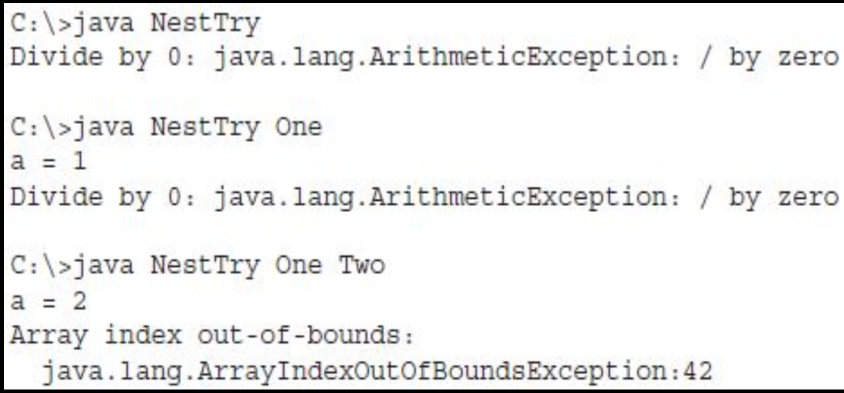
- A **try** statement can be inside the block of another **try.** Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **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.

Exception-Handling

Nested try Statements: ...



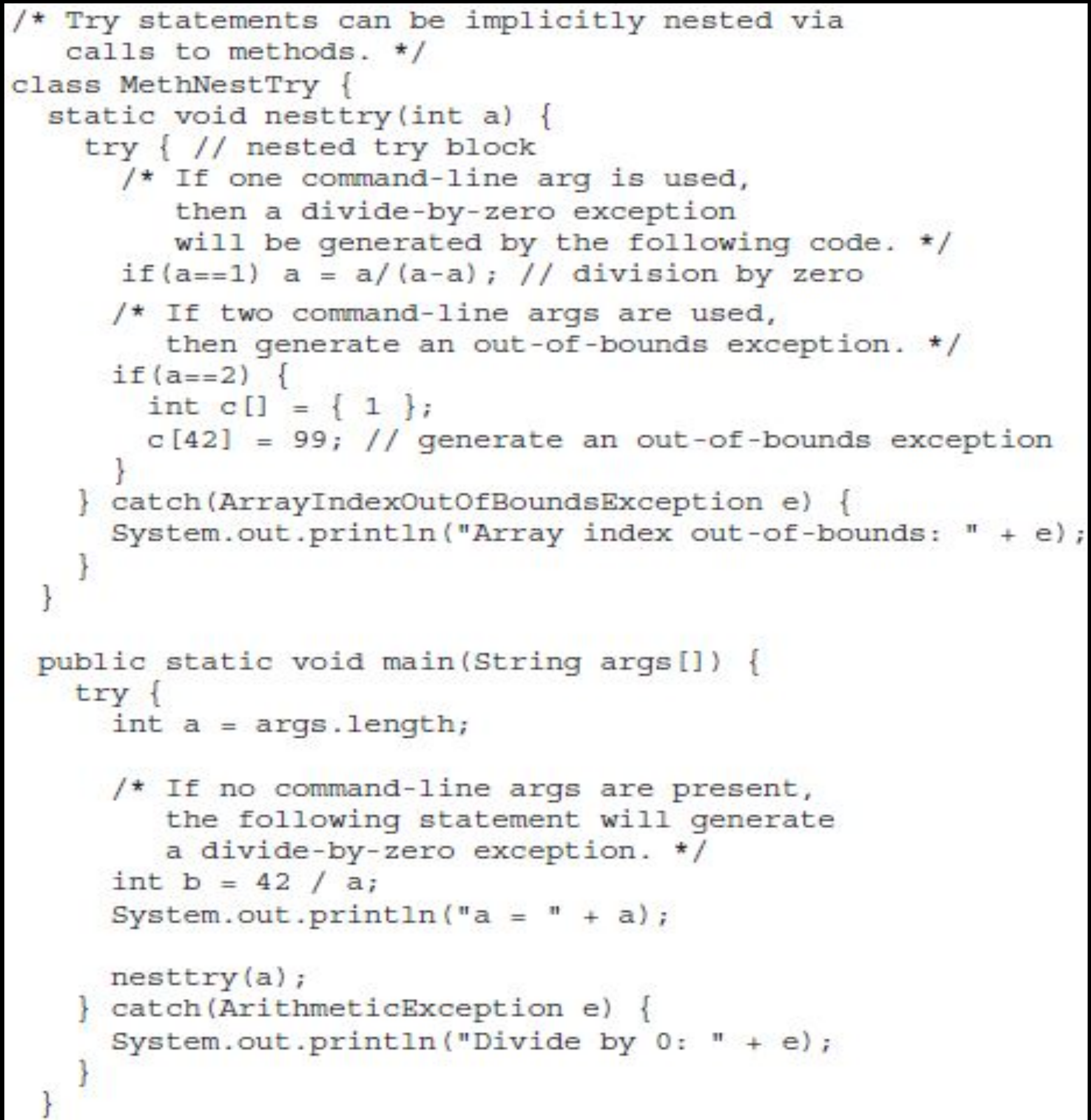
```
// An example of nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* 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);
    }
```



```
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
```

Exception-Handling

Nested try Statements: ...



```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
  static void nesttry(int 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);
    }
  }

  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);

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

The output of this program is identical to that of the preceding example.

Exception-Handling

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: ***throw*** ***ThrowableInstance;***
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable.** Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways to obtain a **Throwable** object:

using a parameter in a **catch** clause, or

creating one with the **new** operator.

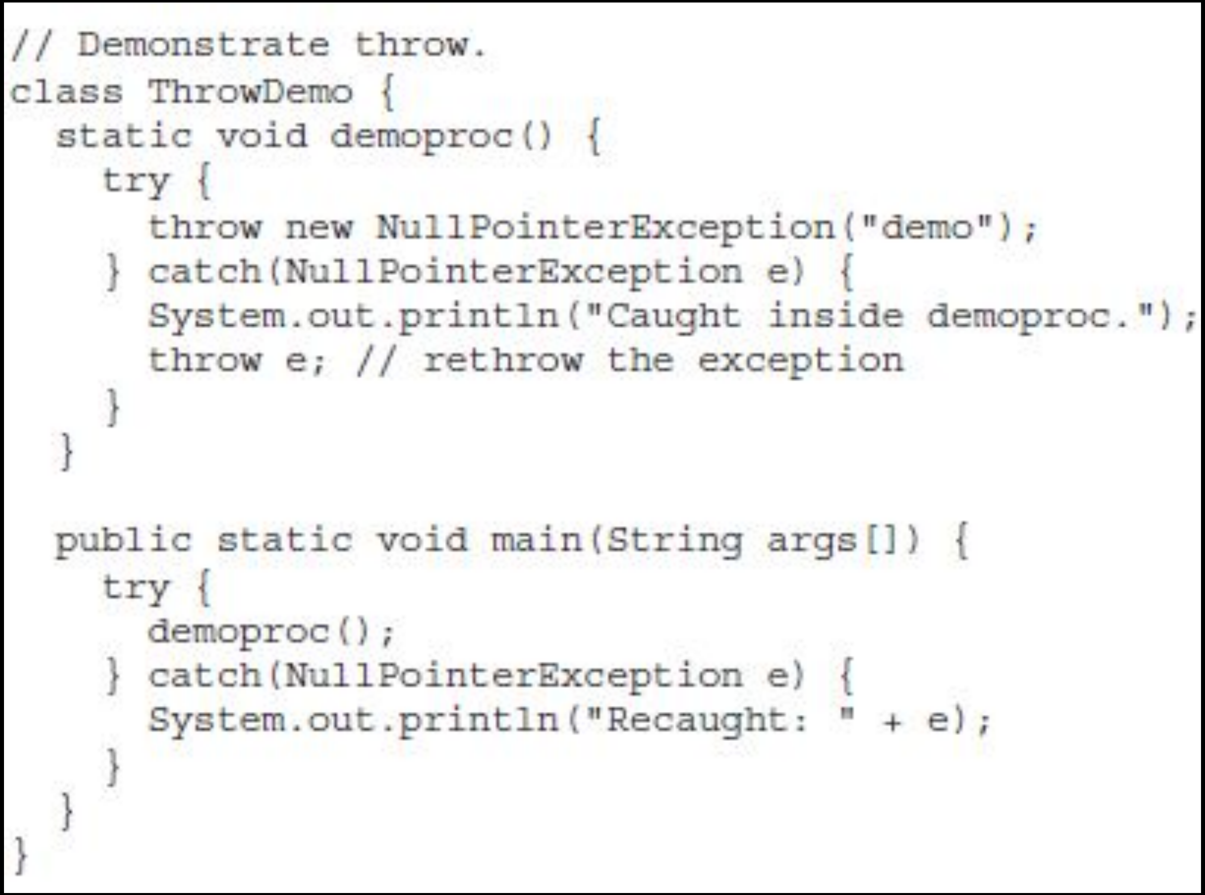
Exception-Handling

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 stack trace.
- Aa sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

Exception-Handling

throw: ...



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

The resulting output:

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

Exception-Handling

throw: ...

- The program also illustrates how to create one of Java’s standard exception objects. Pay close attention to this line:

  throw new NullPointerException("demo");

- Here, new is used to construct an instance of **NullPointerException.** Many of Java’s builtin 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( ).** It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable.**

Exception-Handling

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. 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.

Exception-Handling

throws:

- The general form of a method declaration that includes a **throws clause:**

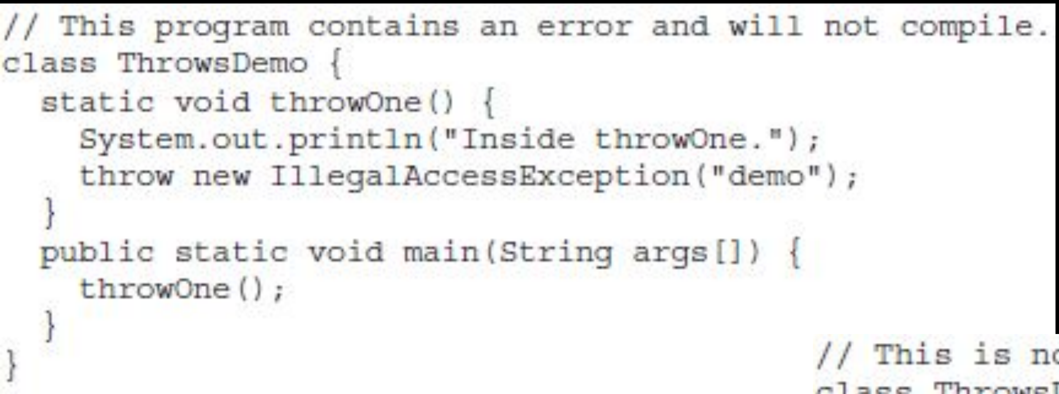
```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

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

Exception-Handling

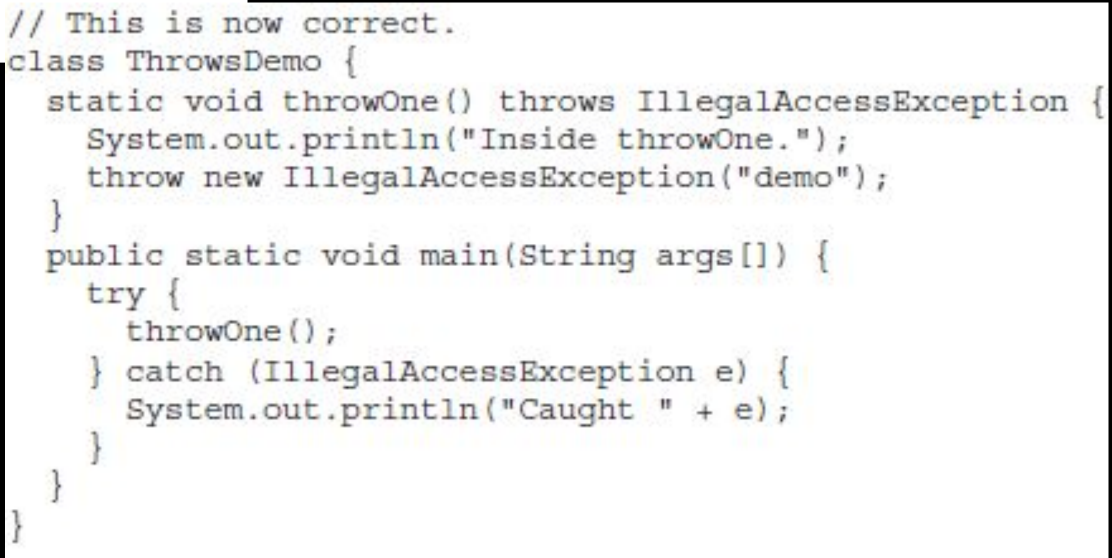
throws: ...

- 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();
  }
}
```

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);
    }
  }
}
```

Exception-Handling

finally:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- **Eg:** If a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

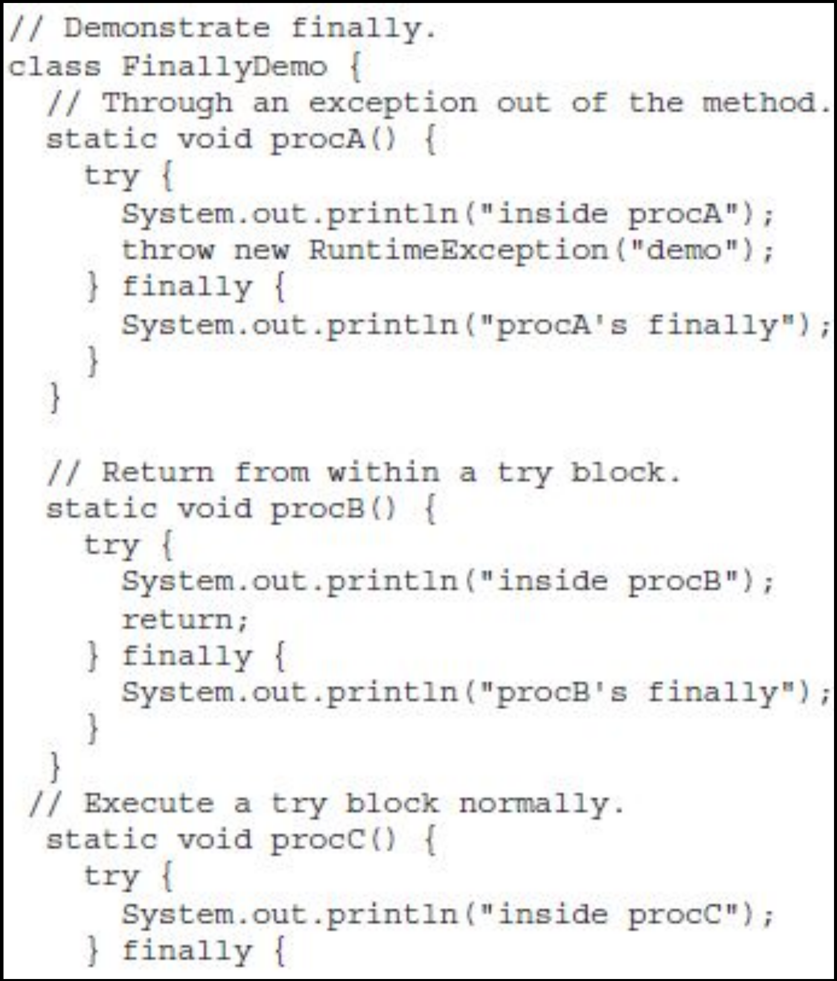
Exception-Handling

finally: ...

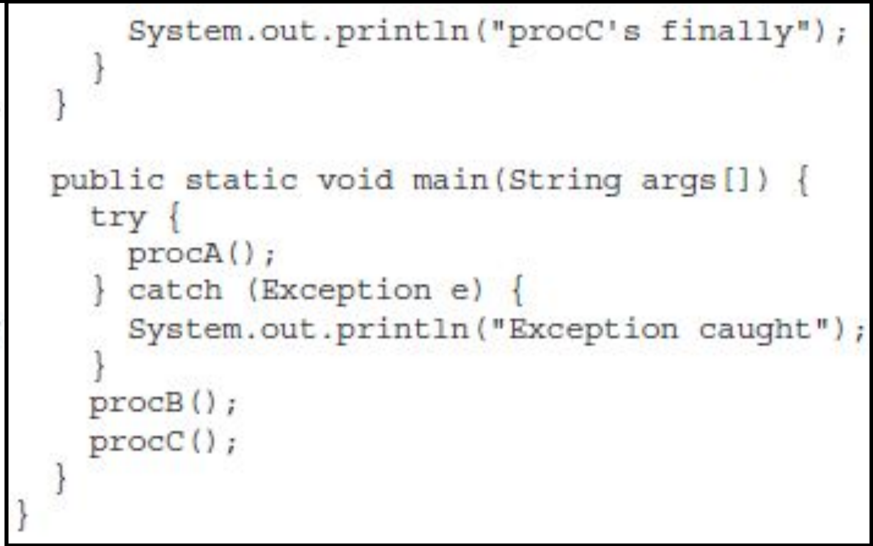
- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if 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. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Exception-Handling

finally: ...



```
// Demonstrate finally.
class FinallyDemo {
  // Through 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();
  }
}
```

In this 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**.*

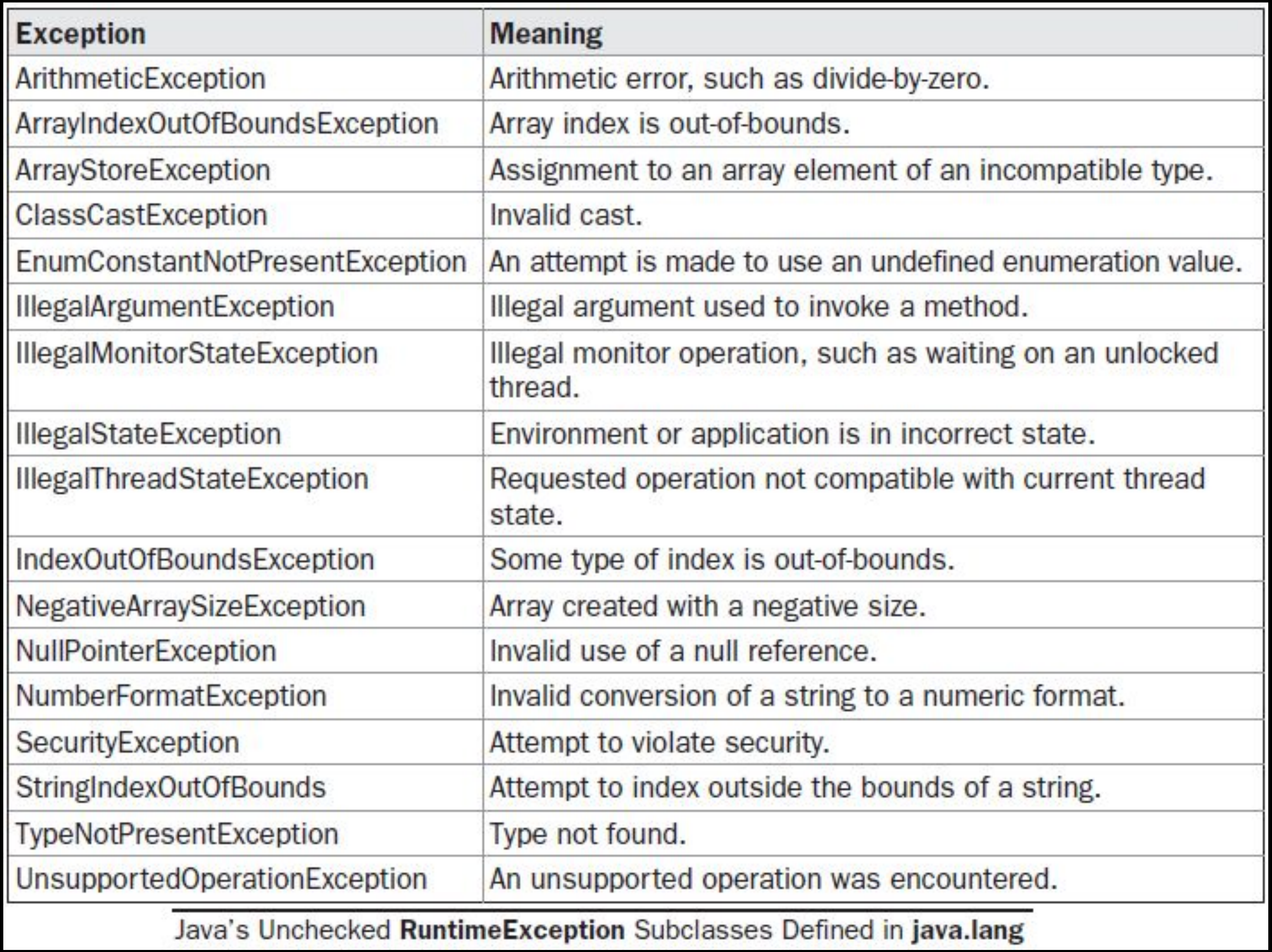
Exception-Handling

Java's Built-in Exceptions:

- Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's throws list. In the language of Java, these are called *unchecked* exceptions because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in **java.lang** are listed, lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. Java defines several other types of exceptions that relate to its various class libraries.

Exception-Handling

Java’s Built-in Exceptions: ...

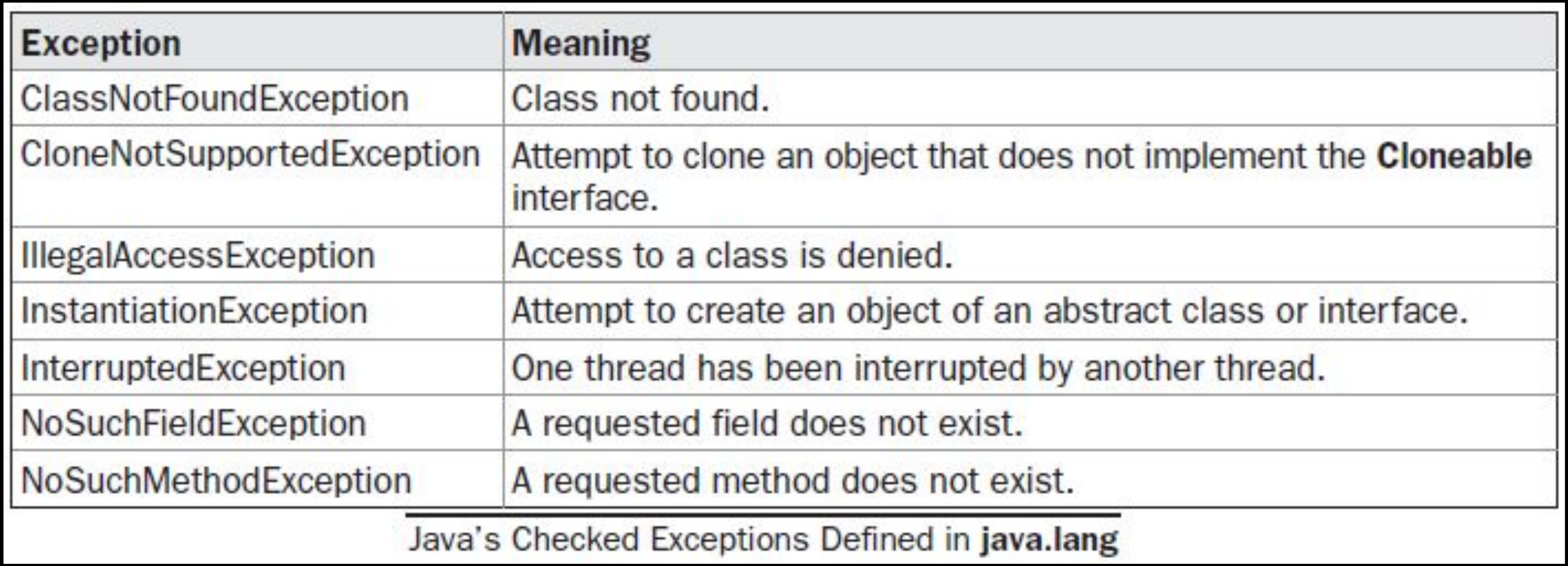


Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made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

Java’s Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception-Handling

Java's Built-in Exceptions: ...



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.

Java's Checked Exceptions Defined in **java.lang**

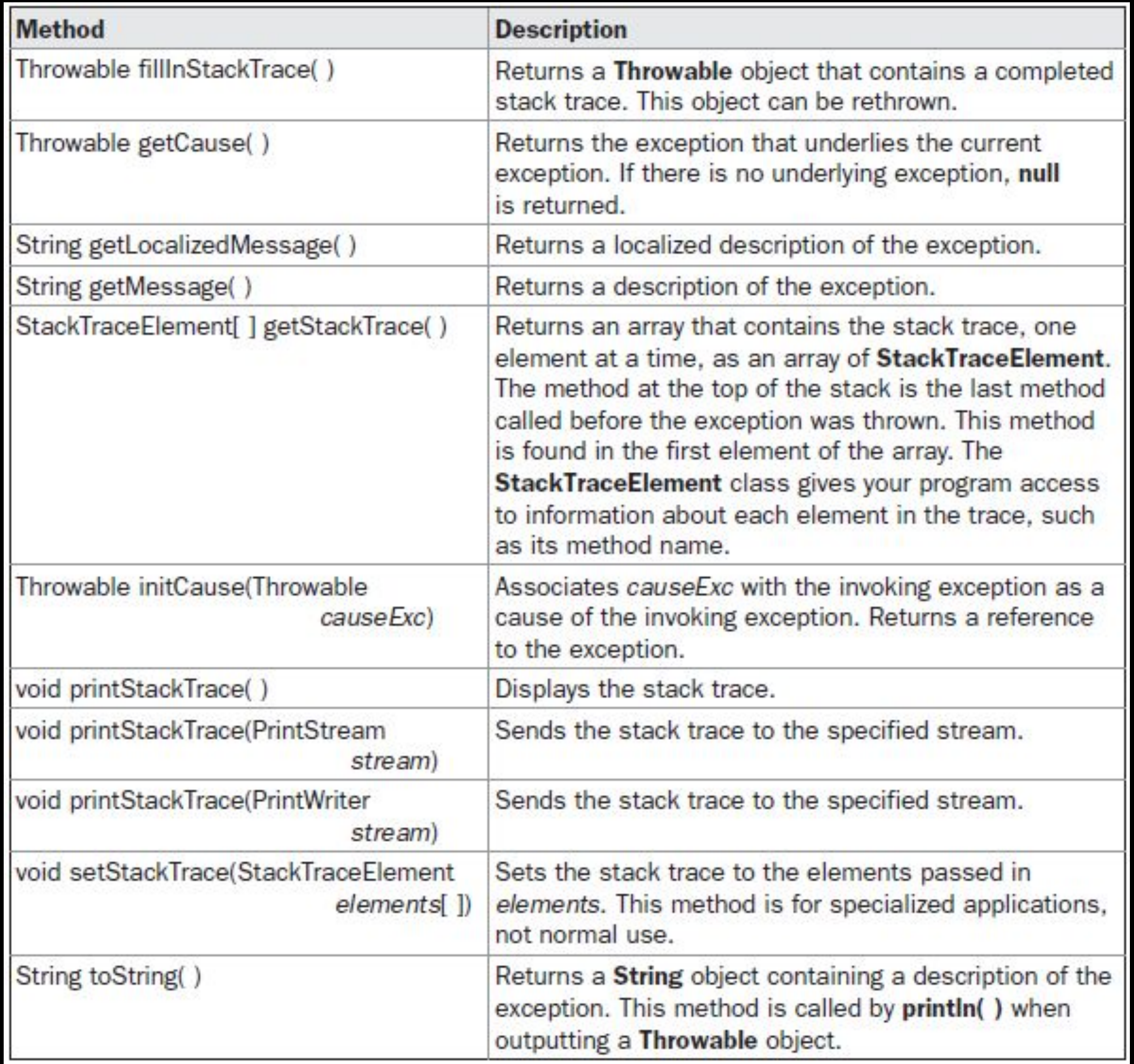
Exception-Handling

Creating our Own Exception Subclasses:

- A user defined exception should be a subclass of the **exception** class.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Exception-Handling

Creating our Own Exception Subclasses: ...



Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

Exception-Handling

Creating our Own Exception Subclasses: ...

Exception-Handling

Using Exceptions:

- Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics.
- It is important to think of **try, throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in our program's logic.
- Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.
- **One last point:** Java's exception-handling statements should not be considered a general mechanism for nonlocal branching.

Exception-Handling

Additional Resources:

Exception-Handling-1

Exception-Handling-2

Exception-Handling-3