Exception Handling – Introduction

- Exception handling enables a program to deal with exceptional situations and continue its normal execution.
- **Runtime errors** occur while a program is running if the JVM detects an operation that is impossible to carry out.
- For example, if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException**.
- If you enter a **double** value when your program expects an integer, you will get a runtime error with an **InputMismatchException**.

Exception Handling – Introduction (Cont.)

- In Java, runtime errors are thrown as exceptions.
- An **exception** is an **<u>object</u>** that represents an **<u>error</u>** or a **<u>condition</u>** that prevents execution from proceeding normally.
- If the exception is **<u>not handled</u>**, the program will **<u>terminate</u>** abnormally.
- How can you handle the exception so that the program can continue to run or else terminate gracefully?

Exception-Handling Overview (Cont.)

- Exceptions are thrown from a method.
- The caller of the method can catch and handle the exception.
- To demonstrate exception handling, including how an exception object is created and thrown, let’s begin with an example that reads in two integers and displays their quotient.

  (**`Quotient.java`**)

  (**`QuotientWithException.java`**)

Try-Catch Block

```
try {
} catch (ExceptionType name) {

} catch (ExceptionType name) {
}
```

Each catch block is an exception handler that handles the type of exception indicated by its argument

Try-Catch-Cont.

- The argument type, *ExceptionType*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with *name*.

Example

- The following are two exception handlers for the writeList method:

```
try {
 } catch (IndexOutOfBoundsException e) {
System.err.println("IndexOutOfBoundsException: " +
e.getMessage());
} catch (IOException e) { System.err.println("Caught IOException:
" + e.getMessage());
}
```

Exception-Handling Overview (Cont.)

- You should not let the method terminate the program—the **caller** should decide whether to terminate the program.
- Java enables a method to throw an exception that can be caught and handled by the caller.

```
throw new ArithmeticException("Divisor cannot be zero");
```

- **Example**

Exception-Handling Overview (Cont.)

- The value thrown is called an exception. The exception is an **object** created from an **exception class**.
- The execution of a throw statement is called throwing an exception.
- In this case, the exception class is java.lang.ArithmeticException.
- The constructor ArithmeticException(str) is invoked to construct an **exception object**, where str is a message that describes the exception.

try-catch

- When an exception is thrown, the normal execution flow is **interrupted**. As the name suggests, to “throw an exception” is to pass the exception from one place to another.
- The **statement for invoking the method** is contained in a **try** block.
- The **try** block contains the code that is executed in **normal** circumstances.
- The exception is caught by the **catch** block. The code in the **catch** block is executed to **handle the exception**.

try-catch (Cont.)

- The **throw** statement is analogous to a method call, but instead of calling a method, it calls a **catch** block.
- In this sense, a **catch** block is like a method definition with a **parameter** that **matches** the **type** of the value being thrown.
- Unlike a method, after the **catch** block is executed, the program control **does not return** to the **throw** statement; instead, it executes the next statement **after** the **catch** block.

try-catch (Cont.)

- The identifier **ex** in the **catch**–block header acts very much like a parameter in a method.
- Thus, this parameter is referred to as a **catch**–block parameter.
- The type (e.g., **ArithmeticException**) preceding **ex** specifies what kind of **exception** the **catch** block can catch.
- Once the exception is caught, you can access the **thrown** value from this **parameter** in the body of a **catch** block.

```
catch (ArithmeticException ex)
```

try-throw-catch block

```
try {
        Code to run;
        A statement or a method that may throw an exception;
        More code to run;
}
catch (type ex) {
        Code to process the exception;
}
```

try-throw-catch block (Cont.)

- An exception may be thrown **directly** by using a **throw** statement in a **try** block(e.g throw new), or by invoking a **method** that **may** throw an exception.
- The main method invokes **quotient**. If the quotient method executes normally, it returns a value to the caller.
- If the **quotient** method encounters an exception, it throws the exception back to its caller.
- The caller’s **catch** block **handles** the exception.(if they are the same type)

Exception Handling *advantage*

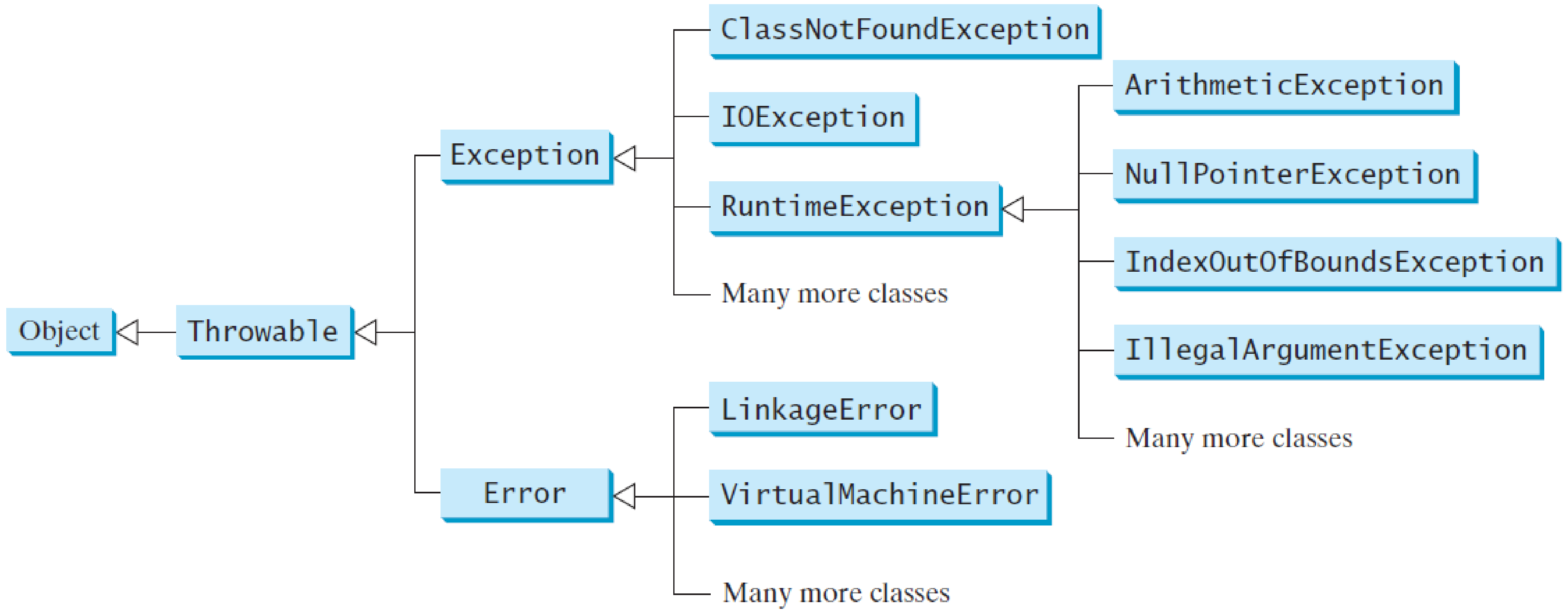
- It enables a method to **throw** an **exception** to its caller, enabling the caller to **handle** the **exception**. Without this capability, the called method itself must **handle** the exception or **terminate** the program.
- The library **method** can detect the error, but only the **caller** knows what needs to be done when an error occurs.
- The key benefit of exception handling is **separating** the **detection** of an error (done in a called method) from the **handling** of an error (done in the calling method).
- Many library methods throw exceptions.

try-throw-catch block – Example

- `InputMismatchExceptionDemo.java` handles an **InputMismatchException** when reading an input.
- When executing **input.nextInt()**, an **InputMismatchException** occurs if the input entered is not an integer.
- Suppose **3.5** is entered. An **InputMismatchException** occurs and the control is transferred to the **catch** block.

Exception Classes

- Exceptions are **objects**, and objects are defined using classes.
- The **root class** for exceptions is **java.lang.Throwable**.



Exception Classes

- The exception classes can be classified into three major types:
  1. system errors
  2. exceptions
  3. runtime exceptions.
- **System errors** are thrown by the JVM and are represented in the **Error** class.
- The **Error** class describes internal system errors, though such errors rarely occur.

Exceptions

- Exceptions are represented in the **Exception** class, which describes errors caused by your program and by **<u>external</u>** circumstances.
- These errors can be caught and handled by your program.

Exceptions – Example

- **ClassNotFoundException:** Attempt to use a class that does not exist.
- This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
- **IOException:** Related to input/output operations, such as invalid input and opening a nonexistent file.
- Examples of subclasses of **IOException** are **InterruptedIOException (invalid input)** and **FileNotFoundException**.

Runtime exceptions

- Runtime exceptions are represented in the **RuntimeException** class, which describes **programming errors**, such as bad casting, accessing an out-of-bounds array, and numeric errors.
- Runtime exceptions are generally thrown by the JVM.

Runtime Exceptions – Example

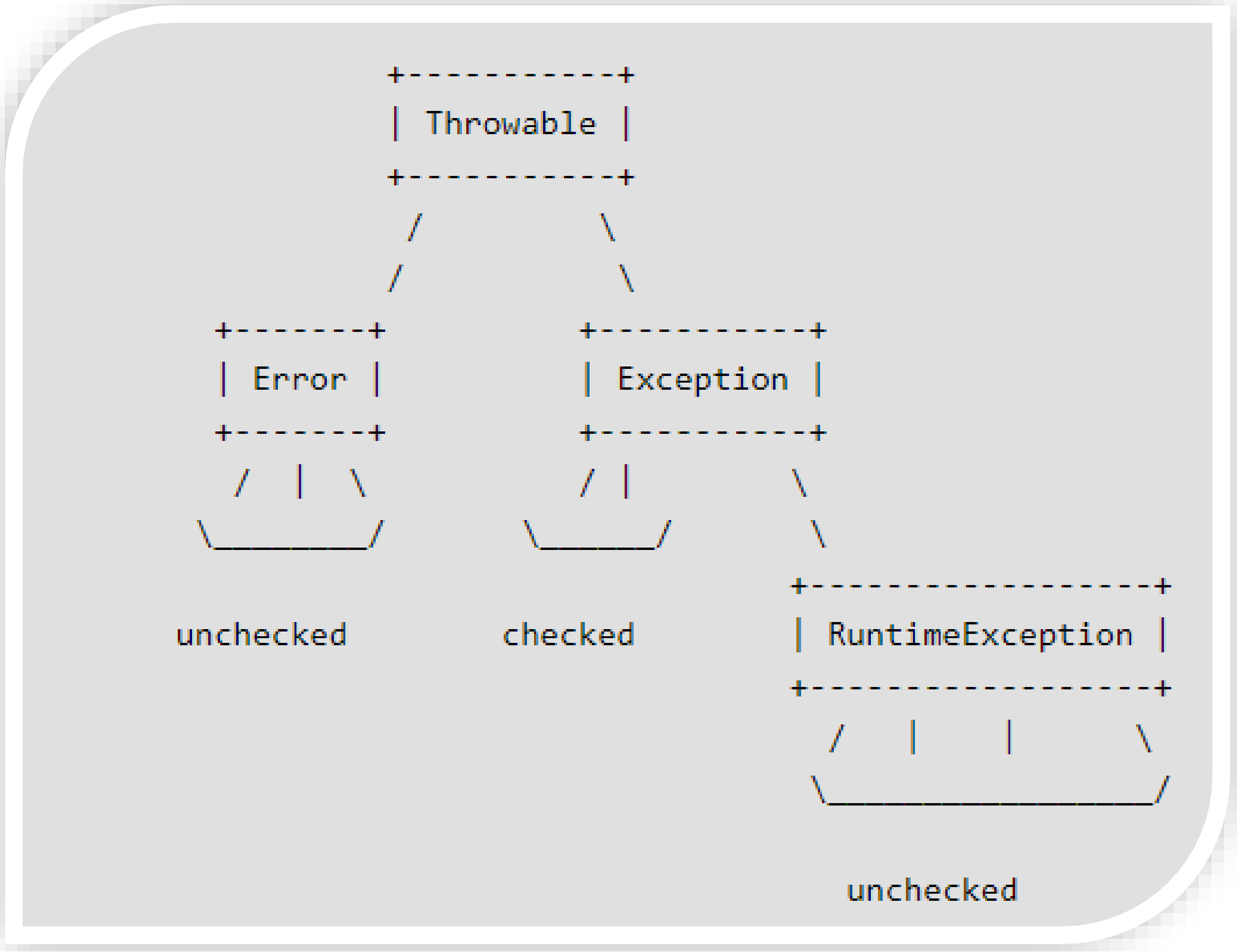
Class	*Reasons for Exception*
ArithmeticException	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions.
NullPointerException	Attempt to access an object through a **null** reference variable.
IndexOutOfBoundsException	Index to an array is out of range.
IllegalArgumentException	A method is passed an argument that is illegal or inappropriate.

unchecked/checked 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 them in a **try-catch** block or declare it in the method header.

Remember This……….

In Java exceptions
under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.



unchecked Exceptions

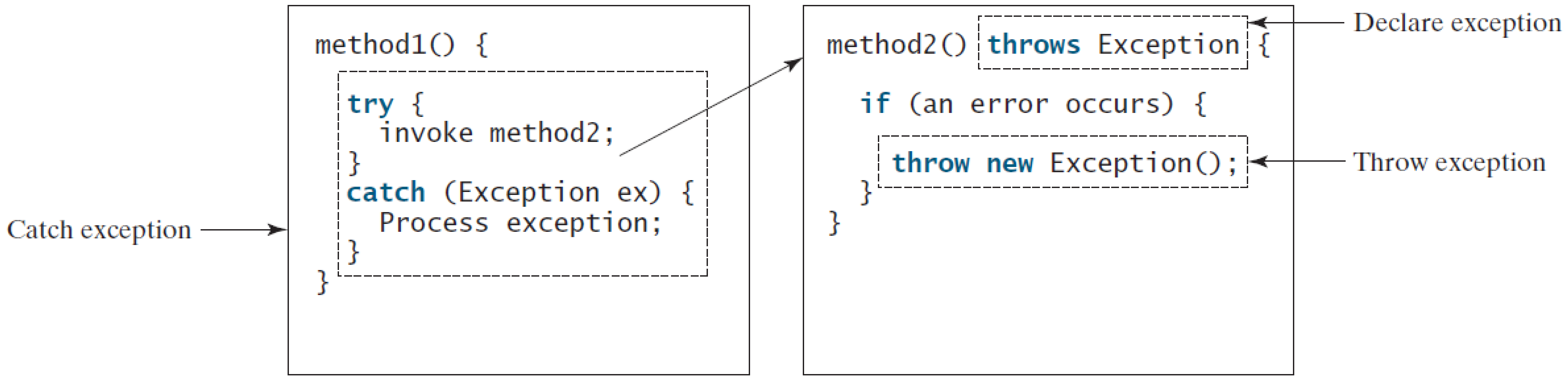
- In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable.
- For example, a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it.
- An **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array.
- These are logic errors that should be corrected in the program.
- To avoid cumbersome overuse of **try-catch** blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

Exception Handling (Cont.)

- A **handler** for an **exception** is found by propagating the exception backward through a chain of method calls, starting from the current method.
- Java’s exception-handling model is based on three operations:

1. declaring an exception
2. throwing an exception
3. catching an exception

Exception Handling Operations



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Declaring Exceptions

- Every method must state the types of **checked** exceptions it might throw. This is known as *declaring exceptions*.
- Because **system errors** and **runtime errors** can happen to any code, Java **does not** require that you declare **Error** and **RuntimeException** (unchecked exceptions) explicitly in the method.
- All other exceptions thrown by the method must be explicitly declared in the method header so that the caller of the method is informed of the exception.

Declaring Exceptions Syntax

- To declare an exception in a method, use the **throws** keyword in the method header

  public void myMethod() **throws** IOException

- The **throws** keyword indicates that **myMethod** might throw an **IOException**.
- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after **throws**:

  public void myMethod()

  throws Exception1, Exception2, ..., ExceptionN

Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as **throwing an exception**.
- Example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of **IllegalArgumentException** and throw it.

Throwing Exceptions – Syntax

```
IllegalArgumentException ex =
   new IllegalArgumentException("Wrong Argument");
throw ex;



throw new IllegalArgumentException("Wrong  Argument");
```

Throwing Exceptions – Notes

- **IllegalArgumentException** is an exception class in the Java API.
- In general, each exception class in the Java API has at least two constructors: a no-arg constructor, and a constructor with a **String** argument that describes the exception.
- This argument is called the *exception message*, which can be obtained using **getMessage()**.
- The keyword to declare an exception is **throws**, and the keyword to throw an exception is **throw**.

Catching Exceptions Syntax

- When an exception is thrown, it can be caught and handled in a **try-catch** block, as follows:

```
try {
        statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
        handler for exception1;
}
catch (Exception2 exVar2) {
        handler for exception2;
}
...
catch (ExceptionN exVarN) {
        handler for exceptionN;
}
```

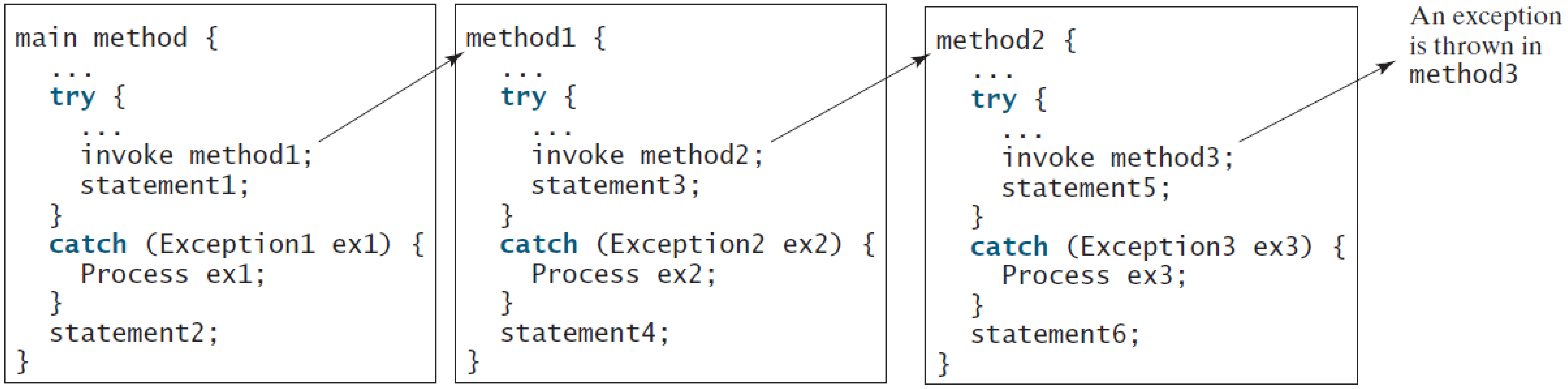
Catching Exceptions Process

- If no exceptions arise during the execution of the **try** block, the **catch** blocks are skipped.
- If one of the statements inside the **try** block throws an exception, Java skips the remaining statements in the **try** block and starts the process of finding the code to handle the exception.
- The code that handles the exception is called the *exception handler*; it is found by **propagating the exception** backward through a chain of method calls, starting from the current method.

Catching Exceptions Process (Cont.)

- Each **catch** block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the **catch** block.
- If so, the exception object is assigned to the variable declared, and the code in the **catch** block is executed.
- If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.
- If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.
- The process of finding a handler is called **catching an exception**.

Catching Exceptions Process



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Catching Exceptions – Notes

- Various exception classes can be derived from a common superclass. If a **catch** block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.
- The order in which exceptions are specified in **catch** blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type.

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

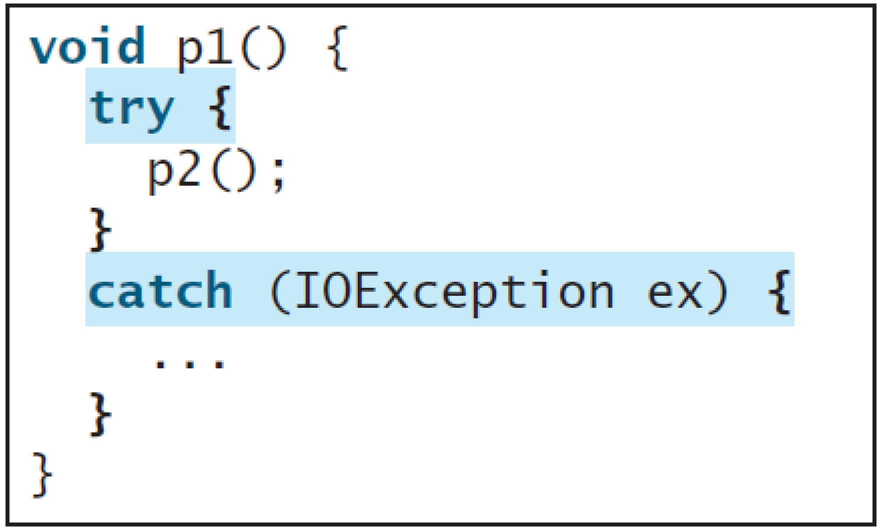
Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

Correct order

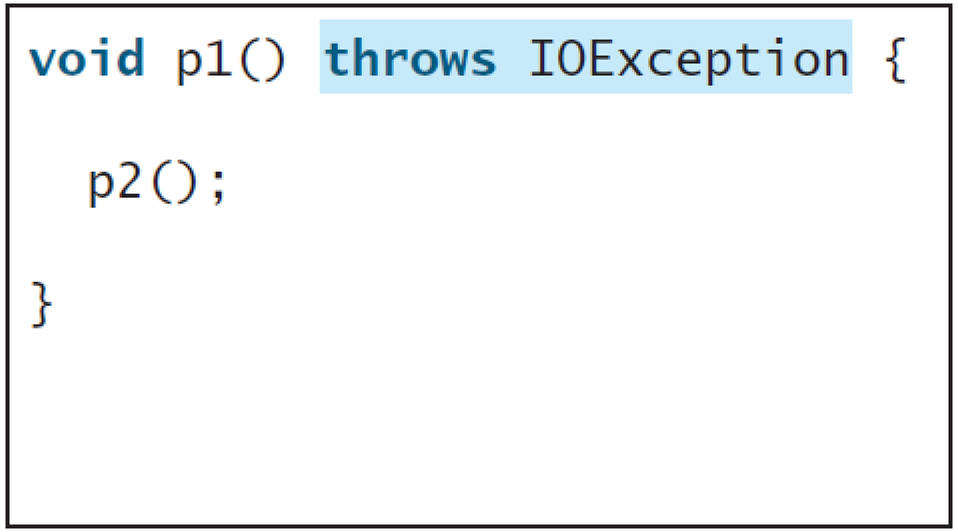
Catching Exceptions – Notes

- suppose that method **p1** invokes method **p2**, and **p2** may throw a checked exception (e.g., **IOException**);



```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

Catch exception



```
void p1() throws IOException {

  p2();

}
```

Throw exception

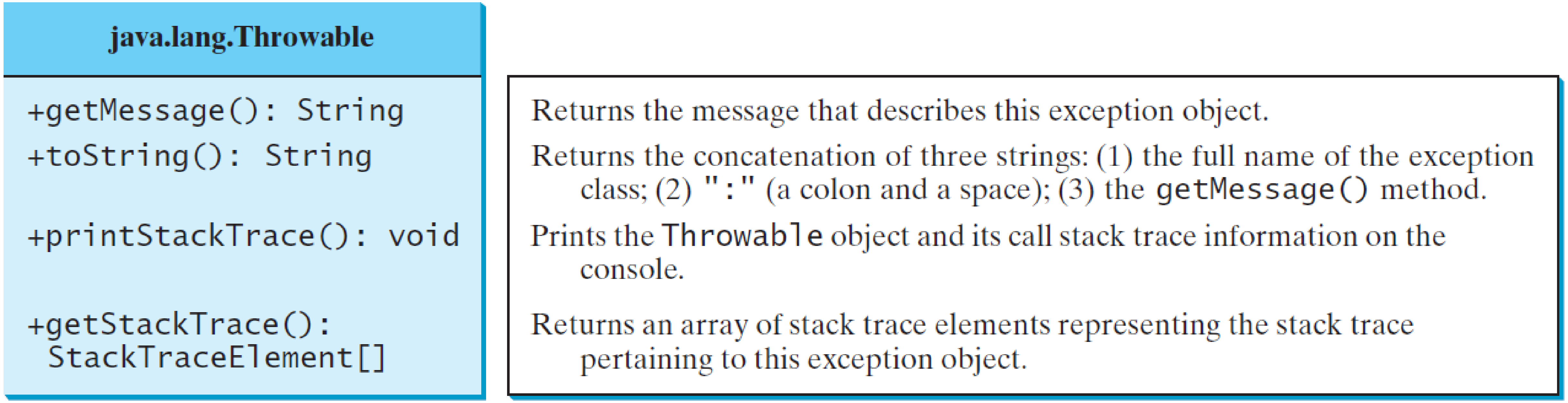
Catching Exceptions – Notes

- You can use the new JDK 7 multi-catch feature to simplify coding for the exceptions with the same handling code. The syntax is:

```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {
    // Same code for handling these exceptions
}
```

- Each exception type is separated from the next with a vertical bar (|).
- If one of the exceptions is caught, the handling code is executed.

Getting Information from Exceptions



java.lang.Throwable

`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `":"` (a colon and a space); (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

The **finally** Clause

- The **finally** clause is always executed regardless whether an exception occurred or not.
- Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught.
- Java has a **finally** clause that can be used to accomplish this objective.

The **finally** Clause (Cont.)

- The syntax for the **finally** clause might look like this:

```
try {
    statements;
}
catch (TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
```

The **finally** Clause (Cont.)

- The **finally** block executes even if there is a **return** statement prior to reaching the **finally** block.
- The code in the **finally** block is executed under all circumstances, regardless of whether an exception occurs in the **try** block or is caught. Consider three possible cases:
- If no exception arises in the **try** block, **finalStatements** is executed, and the next statement after the **try** statement is executed.

The **finally** Clause (Cont.)

- If a statement causes an exception in the **try** block that is caught in a **catch** block, the rest of the statements in the **try** block are skipped, the **catch** block is executed, and the **finally** clause is executed. The next statement after the **try** statement is executed.
- If one of the statements causes an exception that is not caught in any **catch** block, the other statements in the **try** block are skipped, the **finally** clause is executed, and the exception is passed to the caller of this method.

Rethrowing Exceptions

- Java allows an exception handler to rethrow the exception if the handler cannot process the exception or simply wants to let its caller be notified of the exception.

Rethrowing Exceptions

- The syntax for rethrowing an exception may look like this:

```
try {
      statements;
}
catch (TheException ex) {
      perform operations before exits;
      throw ex;
}
```

- The statement **throw ex** rethrows the exception to the caller so that other handlers in the caller get a chance to process the exception **ex**.

Chained Exceptions

- Throwing an exception along with another exception forms a chained exception.
- Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called chained exceptions.
- Example: (`ChainedExceptionDemo.java`)

Defining Custom Exception Classes

- You can define a custom exception class by extending the **java.lang.Exception** class.
- Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes.
- If you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from **Exception** or from a subclass of **Exception**, such as **IOException**.

Defining Custom Exception Classes – Example

- In the previous example, `CircleWithException.java`, the **setRadius** method throws an exception if the radius is negative.
- In this example, `InvalidRadiusException`, suppose you wish to pass the radius to the handler. In that case, you can define a custom exception class.
- `TestCircleWithCustomException.java`