Exceptions and Exception Handling

Exception

- An abnormal condition that occurs at runtime is called an *exception*.
- Java's Exception class and its subclasses provide an automatic and clean mechanism for handling exceptions.
- E.g. A file placed in the wrong directory, an array index out of bounds, an illegal argument, or division by zero are a few common exceptions that no programmer has escaped.

Exception Class

- The subclasses of Exception include
  - ClassNotFoundException ,
  - IOException ,
  - FileNotFoundException ,
  - EOFException (End of File Exception) ,
  - ArithmeticException ,
  - NullPointerException ,
  - IndexOutOfBoundsException , and
  - IllegalArgumentException.

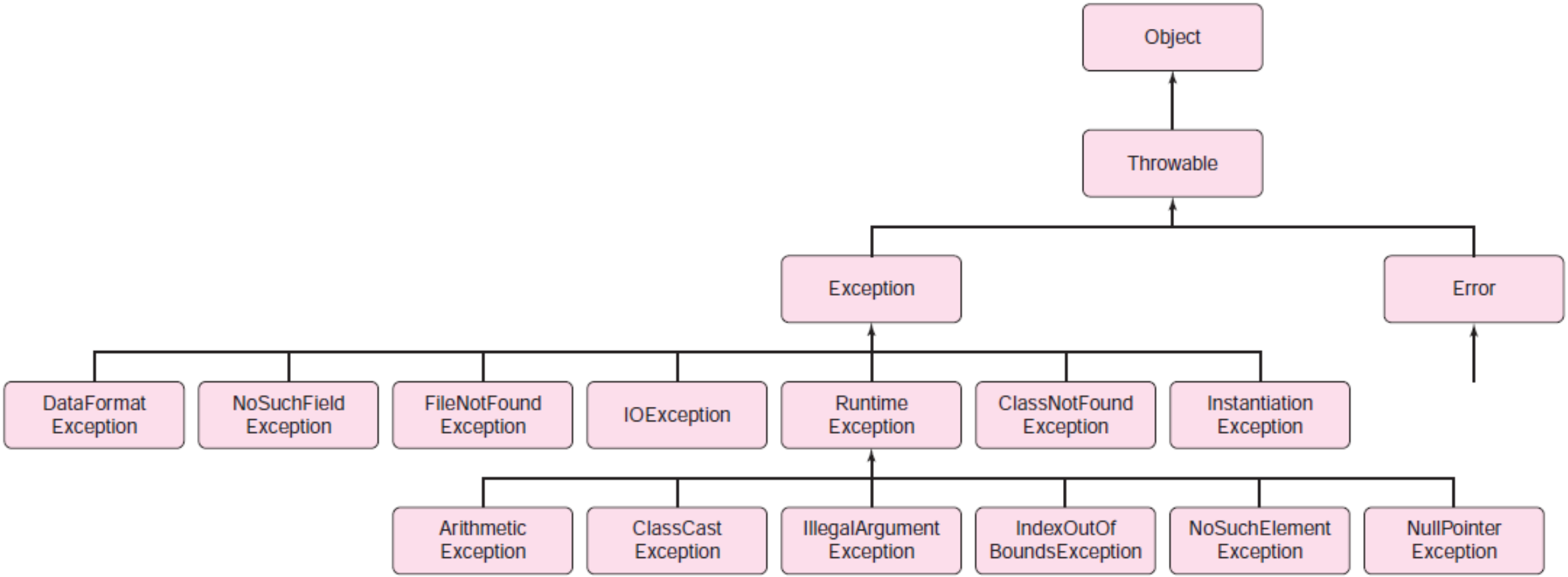

FIGURE 14.8 A partial view of the *Exception* hierarchy

Continue..

- Exception , along with Error , extends Throwable . The Error class encapsulates internal system errors such as the Java Virtual Machine running out of memory. There is not much you can do about system errors so we do not discuss such errors.

Exception Handling with if-else

- Without handling this exception, division by zero causes a program crash. Handling the exception allows the program to deal with the error more gracefully. Checking for exceptional conditions with if statements is certainly one method for handling exceptions, but Java's built-in mechanism is better.

Creating, Throwing & Catching Exceptions

- To handle exceptions uniformly and efficiently, Java provides the *try-throw-catch* construction.
- Generally speaking, when an exception occurs,
  - an Exception object that holds information about the exception is instantiated, and
  - the Exception object is passed, or *thrown* , to a section of code called a *catch block* that handles the exception.
- This scenario implies that, when an exception occurs, program control, along with an Exception object containing information about the exception, is passed, like a parameter, to the catch block, and the catch block takes control or handles the exception.

Continue..

- **The try block:**
  - try
  - {
    - code
    - **instantiate an Exception object, e**
    - **throw e // pass *e* to the *catch* block**
    - code
  - }
- **The catch block:**
  - catch (**Exception e**)
  - {
    - code that handles the exception
  - }

Continue..

- When an Exception object is *thrown,* the program branches to the corresponding catch block.
- The object e , belonging to Java's Exception class, is called the *catch block parameter*. Although the term *parameter* is used in this context, a catch block parameter is not really a parameter, nor is a catch block a method. A catch block is a section of code that executes when an Exception is passed to it.
- Every catch block must have an associated try block.
- After the code of the catch block executes, the program continues with any statements that follow the catch block.

Continue..

- **getMessage() :**
  - The Exception class has two constructors:
    - Exception(String s) , which instantiates an Exception object with a message;
    - Exception() , the default constructor, which instantiates an Exception object with a “ null message.”
- The method
  - String getMessage()
    - returns the string stored in an Exception object or else null.

System Generated Exceptions

- It is the case that, when a “standard” exception occurs, the Java Virtual Machine automatically creates and throws the Exception object. No explicit instantiation or throw statement is required. The JVM takes the initiative.
- For example, when division by zero occurs, the JVM instantiates and throws an ArithmeticException object that holds information about the error; if a program accesses a null reference, a NullPointerException object is automatically instantiated and thrown; or if an application passes an illegal argument to a method, the JVM creates and throws an IllegalArgumentException object.

Continue..

- “All exceptions are thrown, but not every exception necessitates an explicit throw statement. If a standard system exception occurs (file not found, array out of bounds, IO exception, arithmetic exception, etc.) the Java system automatically instantiates and throws the exception.”
- However, if the Java Virtual Machine throws an exception, no customized message can be attached to the Exception object, although an error message can be printed in the catch block.
- An Exception object is thrown by JVM, but how is it caught?
- If there is no catch block, the exception is nonetheless caught by the Java Virtual Machine, which handles the exception by terminating the program and issuing the message.

Continue..

- “You should be as specific as possible when throwing an exception.”

Multiple Catch Blocks

- Multiple catch blocks should be written in order from most specific to least specific exception
- try
  - {
    - statements
  - }
- catch (ArithmeticException e)
  - {
    - statements
  - }
- catch (NullPointerException e)
  - {
    - statements
  - }
- catch (Exception e)
  - {
    - statements
  - }

Continue..

- The final catch block is a “catch all.” Notice that the catch blocks are purposely written in order from most specific to least specific. If the “ catch(Exception e) block” had come first, then all exceptions would be caught by that block, and the code would not distinguish a NumberFormatException from another type of Exception . The compiler, in fact, forbids this ordering.

Unchecked and Checked Exceptions

- UNCHECKED EXCEPTION
  - Java's Exception hierarchy divides exceptions into two categories, *unchecked* exceptions and *checked* exceptions.
  - RuntimeException exceptions fall into the category of unchecked exceptions. Unchecked exceptions can occur almost anywhere in any method and are the most common types of exceptions.
  - An unchecked exception usually occurs due to some program flaw such as an invalid argument, division by zero, an arithmetic error, or an array out of bounds.

Some unchecked RuntimeException exceptions

ArithmeticException	Some arithmetic error, e.g., division by zero.
ArrayIndexOutOfBoundsException	Invalid index value for an array.
ArrayStoreException	Invalid type for an array element.
ClassCastException	Invalid cast.
IllegalArgumentException	Invalid argument when calling a method.
NullPointerException	Attempt to dereference (access) a null pointer.
NumberFormatException	Invalid string in a conversion to a number.
StringIndexOutOfBounds	Invalid index value for a string.

FIGURE 14.9 Some common unchecked *RuntimeException* exceptions

Continue..

- An unchecked exception, such as an out of bounds array index, is one that usually cannot be handled during runtime.
- If an unchecked exception occurs, the JVM automatically creates an Exception object and throws the object, but a program need not catch and handle the exception. In fact, a method that generates an unchecked error can usually do nothing productive to recover from the error. Therefore, Java does not insist that a program handle unchecked exceptions.
- Every unchecked exception *is* eventually caught and handled. If the program does not explicitly handle the exception, then it is caught and handled by the Java Virtual Machine. Indeed, this is the more common scenario. When the segment executes, the JVM throws and catches the exception. The JVM handles the exception by terminating the program and issuing the message.

Continue..

- CHECKED EXCEPTION
  - An exception that is not unchecked is called a *checked exception*. A checked exception is one from which a method *can* reasonably be expected to recover.
  - All exceptions derived from Exception , except for RuntimeException , are checked exceptions. For example, bad input data such as an invalid file name might generate a FileNotFoundException exception. This is a checked exception.
  - In contrast to an unchecked exception, a checked exception cannot be ignored. If a checked exception is thrown in a method, the method *must* either handle the exception or pass it back to the caller to handle.

Continue..

- In particular, the method must either
  - catch the exception with a catch block, or
  - pass the exception back to the caller for handling by explicitly listing the exception in a throws clause appended to the method signature.

The *throws* clause

- If a method does not explicitly catch and handle a checked exception, the method, by including a throws clause in its heading, passes the exception back to the caller, and it becomes the caller's responsibility to handle or throw the exception.
- Not catching a checked exception *and* leaving out the throws clause generates a compilation error.
- If a method does not catch a checked exception, the Exception object is passed to the caller, via the throws clause. Checked exceptions can be passed along the chain of method calls right up to the main(...) method and finally to the system, until they are eventually caught and handled.

Checked Exceptions Handled in 3 ways

1. Using a try-catch construction, the exception is handled directly by the method that generates the exception.
2. The exception is thrown back to the calling method and handled by the caller.
3. The exception is passed (thrown) all the way back through the caller to the Java system and handled by the system.

The *throws* clause

- Most of the standard exceptions encountered are unchecked. IOExceptions are the exception, so to speak. If an exception is checked and you fail to catch it or include a throws clause, the Java compiler will persistently remind you.
- Notice the difference between throw and throws. The former passes or throws an exception, and the latter indicates that the method does not handle a particular exception, but instead, passes the exception back to the caller. One letter changes the meaning. Be careful.

catch can *thorw*

- A system-generated exception includes a system-generated message, which may be a bit cryptic or uninformative. It is possible, however, for a method to catch an exception , create a new exception with a message, and then throw (or rethrow) the new exception to the caller.

```
1.  public void myMethod(String filename) throws FileNotFoundException
2.  {
3.     try
4.     {
5.        File file = new File(filename);
6.        // other code
7.     }
8.     catch (FileNotFoundException e)
9.     {
10.       String message = "File not found error in MyMethod : " + filename);
11.       FileNotFoundException e1 = new FileNotFoundException(message);   // add a message
12.       throw e1;
13.    }
14. }
```

Creating Your own Exception Classes

- You can create your own exception class by extending Exception or any subclass of Exception . For example, many applications require that input data consist of positive integers.
- Any class derived from Exception is checked, unless it is derived from RuntimeException, in which case it is unchecked.

finally block

- A *finally block* is a block of code that always executes, regardless of whether or not an exception is thrown. A finally block is paired with either a try-catch pair or a try block.
- The finally block is used as a cleanup device. Without the finally block in Example 14.8, cleanup would be replicated in both the try and the catch blocks. Moreover, a single try block may have multiple catch blocks, so the replication could even be more cumbersome. A finally block is a cleaner solution.
- Variables declared within a try block are known only within that block and are not visible to the finally block. If the variables of a try block must be accessed in a finally block, declare such variables outside the try block.

Example: What is output?

```
1.  import java.io.*;

2.  public class UsingFinally
3.  {
4.      public int add(int a, int b)
5.      {
6.          try
7.          {
8.              return (a + b);
9.          }
10.         finally
11.         {
12.             return 0;
13.         }
14.     }

15.     public static void main(String[] args)
16.     {
17.         UsingFinally x = new UsingFinally();
18.         System.out.println(x.add(3,4));
19.     }
20. }
```

Answer:

- The value that is printed is 0, the value returned by the finally block. When the return in the try block statement is encountered, control immediately passes to the finally block, and the value 0 is returned by the method.
- This occurs because the code in the finally block *must* execute. If the statement return(a b) , in method add() executed before the finally block, then the add() method would immediately terminate and the finally block would never execute. Remember, a method returns a single value and then terminates. When control jumps to the finally block, 0 is returned, and the method terminates. Thus, the try block never gets a chance to return 7, the expected value. A return statement in a finally block effectively precludes a return statement in a try block.
- In general, a finally block should not be used to return a value.