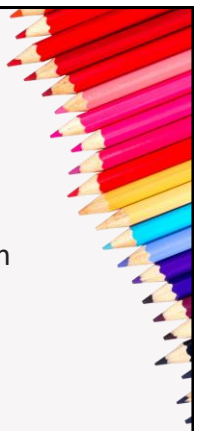
Chapter 12
Exception Handling and Text IO

Motivations

- When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully?



Objectives(1)

- To get an overview of exceptions and exception handling (§12.2).
- To explore the advantages of using exception handling (§12.2).
- To distinguish exception types: Error (fatal) vs. Exception (nonfatal) and checked vs. unchecked (§12.3).
- To declare exceptions in a method header (§12.4.1).
- To throw exceptions in a method (§12.4.2).
- To write a try-catch block to handle exceptions (§12.4.3).
- To explain how an exception is propagated (§12.4.3).

Objectives(2)

- To obtain information from an exception object (§12.4.4).
- To develop applications with exception handling (§12.4.5).
- To use the finally clause in a try-catch block (§12.5).
- To use exceptions only for unexpected errors (§12.6).
- To rethrow exceptions in a catch block (§12.7).
- To create chained exceptions (§12.8).
- To define custom exception classes (§12.9).

Objectives(3)

- To discover file/directory properties, to delete and rename files/directories, and to create directories using the File class (§12.10).
- To write data to a file using the PrintWriter class (§12.11.1).
- To use try-with-resources to ensure that the resources are closed automatically (§12.11.2).
- To read data from a file using the Scanner class (§12.11.3).
- To understand how data is read using a Scanner (§12.11.4).
- To develop a program that replaces text in a file (§12.11.5).
- To read data from the Web (§12.12).
- To develop a Web crawler (§12.13).

5

12.1 Introduction

- *Runtime errors* occur while a program is running if the JVM detects an operation that is impossible to carry out.
- In Java, runtime errors are thrown as exceptions.
- An *exception is an object that represents* an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally.
- *Exception handling enables a program to deal with exceptional situations and continue its normal execution.*

12.2 Exception-Handling Overview

- *Exceptions are thrown from a method. The caller of the method can catch and handle the exception.*

Quotient	Run
QuotientWithIf	Run
QuotientWithMethod	Run
QuotientWithException	Run

Exception Advantages

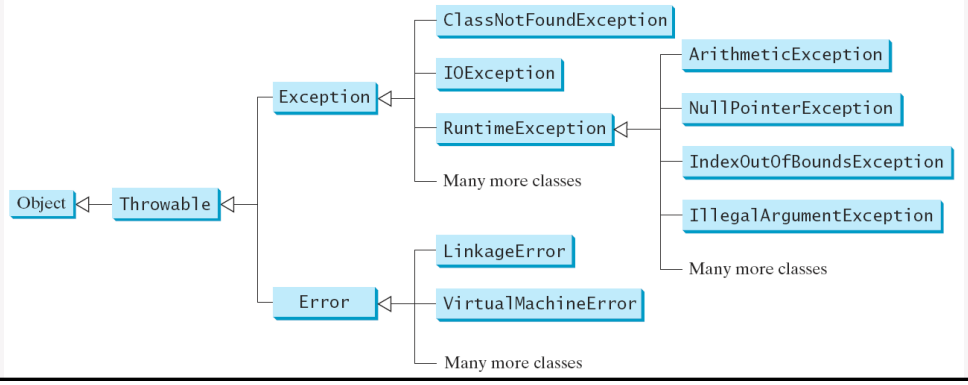
- Now you see the *advantages* of using exception handling.
- It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.
- Handling InputMismatchException
  - By handling InputMismatchException, your program will continuously read an input until it is correct.

InputMismatchExceptionDemo	Run

8

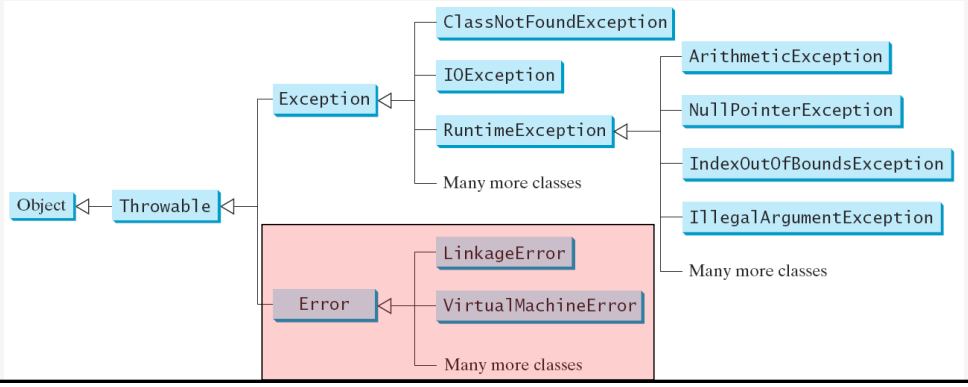
12.3 Exception Types

- *Exceptions are objects, and objects are defined using classes. The root class for exceptions is* ***java.lang.Throwable.***
- Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.



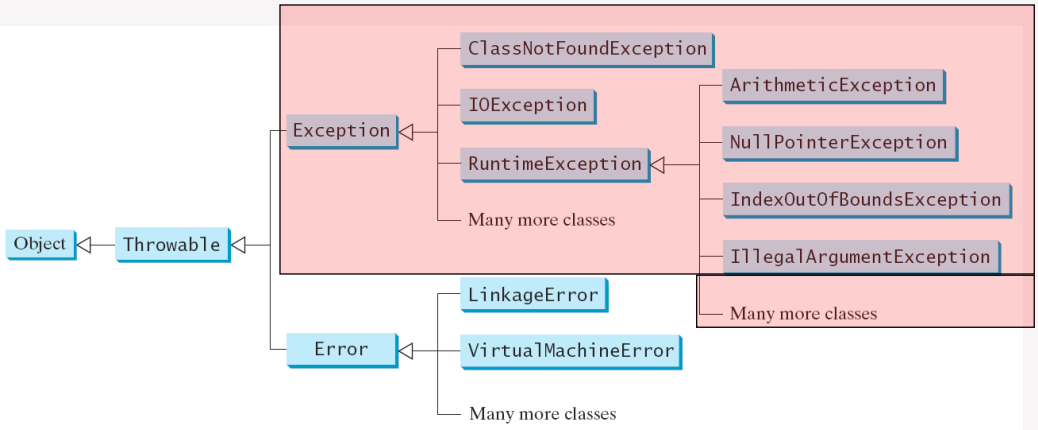
System Error

- System errors are thrown by JVM and represented in the *Error* class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.



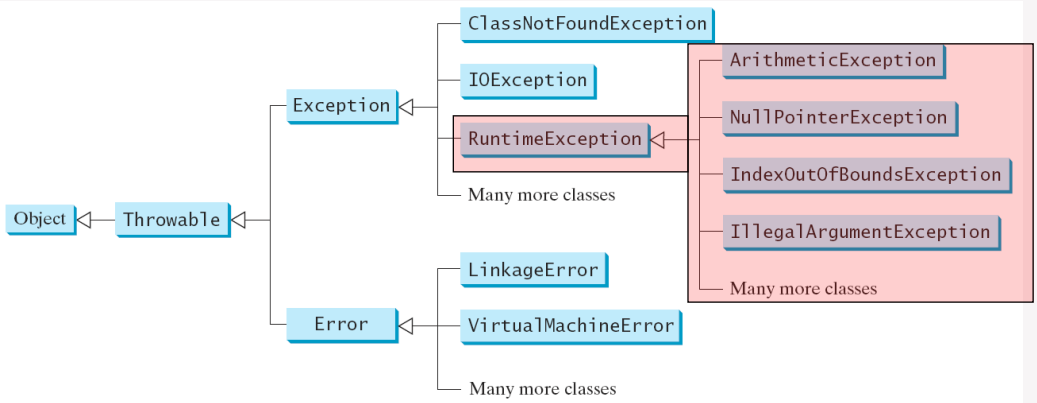
Exceptions

- Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.

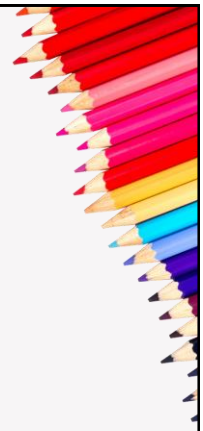


Runtime Exceptions

- RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.



Checked Exceptions vs. Unchecked Exceptions



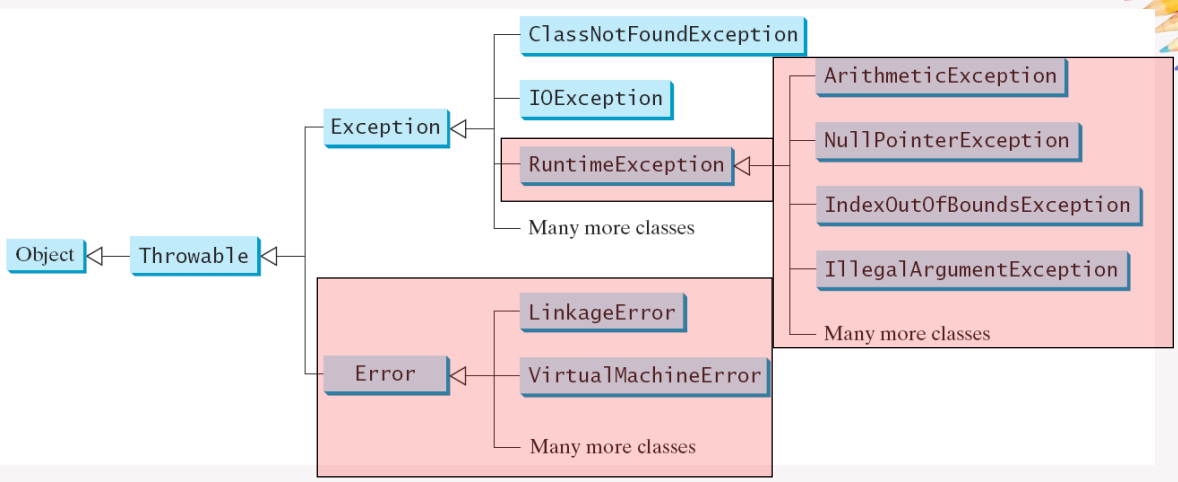
- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
  - 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 the logic errors that should be corrected in the program.
- Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

14

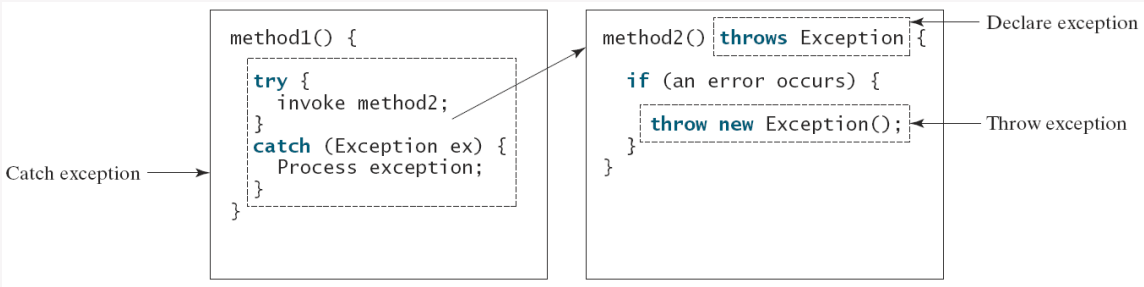
Unchecked Exceptions



15

12.4 Declaring, Throwing, and Catching Exceptions

- Java's exception-handling model is based on three operations: *declaring an exception, throwing an exception, and catching an exception.*



Declaring Exceptions

- Every method must state the types of ***checked exceptions*** it might throw. This is known as *declaring exceptions*.
- To declare an exception in a method, use the **throws** keyword in the method header.

```
public void myMethod() throws IOException


public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```

Throwing Exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*.

```
throw new TheException();

TheException ex = new TheException();

throw ex;
```

- Here is an example,

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

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

18

Throwing Exceptions Example

```
  /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

19

Catching Exceptions

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

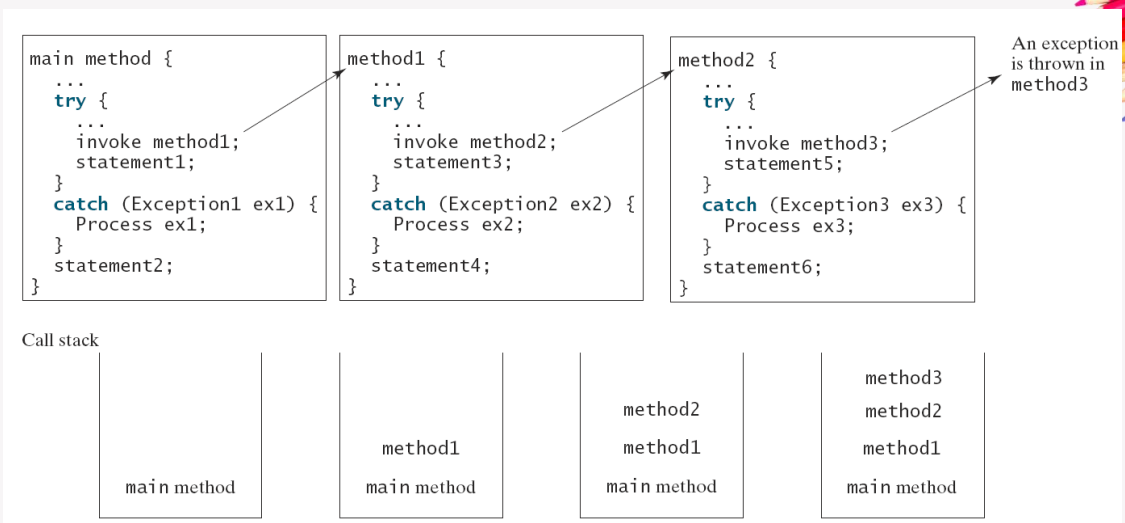
```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

Exception Handler and Exception Propagation

- 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. 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.

21

Catching an Exception



NOTE

- 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) {
  ...
}
```

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

Catch or Declare Checked Exceptions

- Java forces you to deal with checked exceptions.
- If a method declares a checked exception, you must invoke it in a try-catch block or declare to throw the exception in the calling method.
- For example, suppose that method p1 invokes method p2, and p2 may throw a checked exception (e.g., IOException); you have to write the code as shown in (a) or (b) below.

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

(a) Catch exception

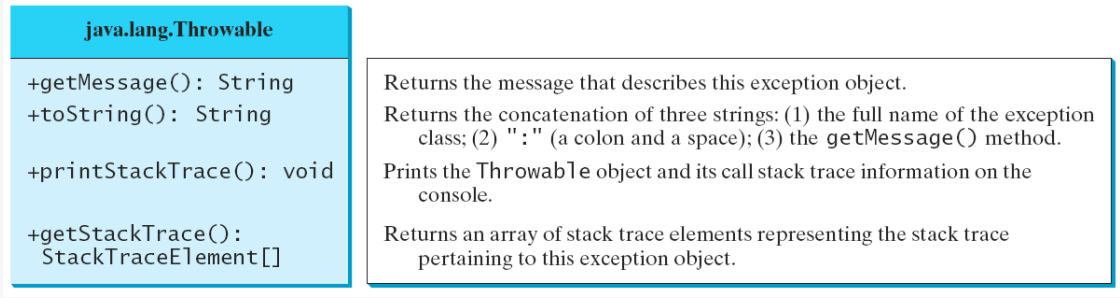
```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

24

Getting Information from Exceptions

- An exception object contains valuable information about the exception. You may use the following instance methods in the java.lang.Throwable class to get information regarding the exception.



TestException

Run

Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 8. The new setRadius method throws an exception if radius is negative.

CircleWithException

TestCircleWithException

Run

26

12.5 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 syntax for the finally clause might look like this:

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

finally {
  finalStatements;
}
```

12.6 When to Use Exceptions

- When should you use the try-catch block in the code? You should use it to deal with *unexpected error conditions*.
- Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

28

12.7 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.*
- The syntax for rethrowing an exception may look like this:
  - 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.**

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

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

30

animation

Trace a Program Execution

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

The final block is always executed

31

animation

Trace a Program Execution

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

Next statement in the method is executed

32

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

33

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
Next statement;
```

The exception is handled.

34

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
Next statement;
```

The final block is always executed.

35

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
Next statement;
```

The next statement in the method is now executed.

36

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {  handling ex; }
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {   finalStatements; }
Next statement;
```

statement2 throws an exception of type Exception2.

37

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {  handling ex; }
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {   finalStatements; }
Next statement;
```

Handling exception

38

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {  handling ex; }
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {   finalStatements; }
Next statement;
```

Execute the final block

39

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {   handling ex; }
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {   finalStatements; }
Next statement;
```

Rethrow the exception and control is transferred to the caller

40

12.8 Chained Exceptions

- Throwing an exception along with another exception forms a chained exception.
- In the preceding section, the catch block rethrows the original exception. Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called chained exceptions.

ChainedExceptionDemo Run

12.9 Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending **`Exception`** or a subclass of Exception.
- In Listing 13.8, the setRadius method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler, you have to create a custom exception class.

InvalidRadiusException CircleWithRadiusException

TestCircleWithRadiusException Run

42

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.
- When to Throw Exceptions?
  - An exception occurs in a method.
  - If you want the exception to be processed by its caller, you should create an exception object and throw it.
  - If you can handle the exception in the method where it occurs, there is no need to throw it.

43

Text IO

44

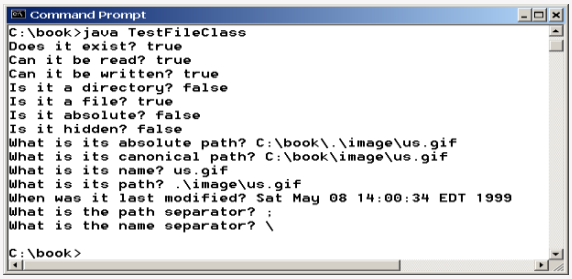
12.10 The `File` Class

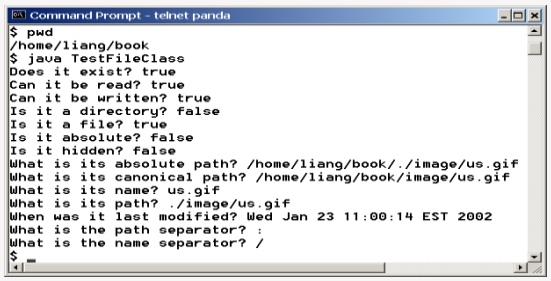
- The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- The filename is a string.
- The `File` class is a wrapper class for the file name and its directory path.

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

Example: Explore File Properties

- Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix.





TestFileClass Run

47

12.11 File Input and Output

- A **`File`** object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the **`Scanner`** and **`PrintWriter`** classes.

48

Writing Data Using PrintWriter

java.io.PrintWriter	
`+PrintWriter(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file-name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
Also contains the overloaded `println` methods.	A `println` method acts like a `print` method; additionally, it prints a line separator. The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix.
Also contains the overloaded `printf` methods.	The `printf` method was introduced in §4.6, "Formatting Console Output."

WriteData Run

Closing Resources Automatically Using try-with-resources

- Programmers often forget to close the file. JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

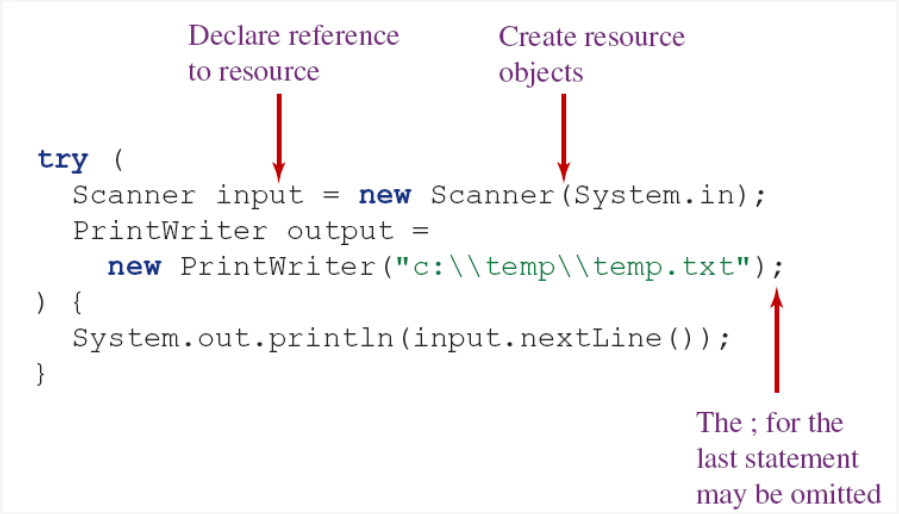
```
try (declare and create resources) {
  Use the resource to process the file;
}
```

WriteDataWithAutoClose Run

Using try-with-resources

- A resource is declared and created in the parentheses following the keyword **try**.
- The resources must be a subtype of **AutoCloseable** such as a **PrinterWriter** that has the **close()** method.
- A resource must be declared and created in the same statement, and multiple resources can be declared and created inside the parentheses.
- The statements in the block immediately following the resource declaration use the resource. After the block is finished, the resource's **close()** method is automatically invoked to close the resource.

Using try-with-resources



Using try-with-resources

- Using try-with resources can not only avoid errors, but also make the code simpler. Note the catch clause may be omitted in a try-with-resources statement.
- Note

  (1) you have to declare the resource reference variable and create the resource altogether in the **try(...)** clause;

  (2) the semicolon (**;**) in last statement in the **try(...)** clause may be omitted;

  (3) You may create multiple **AutoCloseable** resources in the the **try(...)** clause;

  (4) The **try(...)** clause can contain only the statements for creating resources.

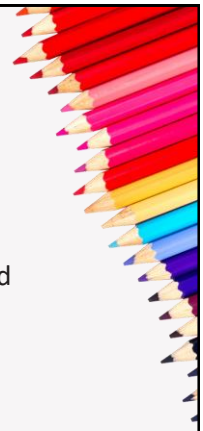
Reading Data Using Scanner

java.util.Scanner	
`+Scanner(source: File)`	Creates a `Scanner` that scans tokens from the specified file.
`+Scanner(source: String)`	Creates a `Scanner` that scans tokens from the specified string.
`+close()`	Closes this scanner.
`+hasNext(): boolean`	Returns true if this scanner has more data to be read.
`+next(): String`	Returns next token as a string from this scanner.
`+nextLine(): String`	Returns a line ending with the line separator from this scanner.
`+nextByte(): byte`	Returns next token as a `byte` from this scanner.
`+nextShort(): short`	Returns next token as a `short` from this scanner.
`+nextInt(): int`	Returns next token as an `int` from this scanner.
`+nextLong(): long`	Returns next token as a `long` from this scanner.
`+nextFloat(): float`	Returns next token as a `float` from this scanner.
`+nextDouble(): double`	Returns next token as a `double` from this scanner.
`+useDelimiter(pattern: String): Scanner`	Sets this scanner's delimiting pattern and returns this scanner.

ReadData Run

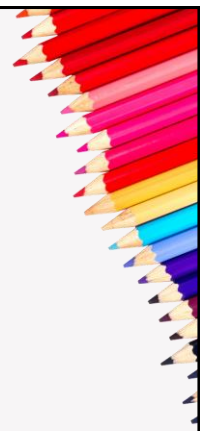
54

How Does **Scanner** Work?



- The token-based input methods read input separated by delimiters. By default, the delimiters are whitespace characters.
  - **nextByte()**, **nextShort()**, **nextInt()**, **nextLong()**, **nextFloat()**, **nextDouble()**, and **next().**
  - You can use the **useDelimiter(String regex)** method to set a new pattern for delimiters.
- A token-based input first skips any delimiters then reads a token ending at a delimiter.
  - The token is then automatically converted into a value. For the **next()** method, no conversion is performed.
  - If the token does not matchthe expected type, a runtime exception **java.util.InputMismatchException** willbe thrown.

How Does **Scanner** Work?



- Both methods **next()** and **nextLine()** read a string. The **next()** method reads a string separated by delimiters and **nextLine()** reads a line ending with a line separator.
- The token-based input method does not read the delimiter after the token.
  - If the **nextLine()** method is invoked after a token-based input method, this method reads characters that start from this delimiter and end with the line separator.
    - *You should not use a line-based input after a token-based input.*
  - The line separator is read, but it is not part of the string returned by **nextLine()**.

Scan a String

- You can read data from a file or from the keyboard using the **Scanner** class. You can also scan data from a string using the **Scanner** class.

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum is " + sum);
```

displays

```
Sum is 27
```

Case Study: Replacing Text

- Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

- For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder
  StringBuffer
```

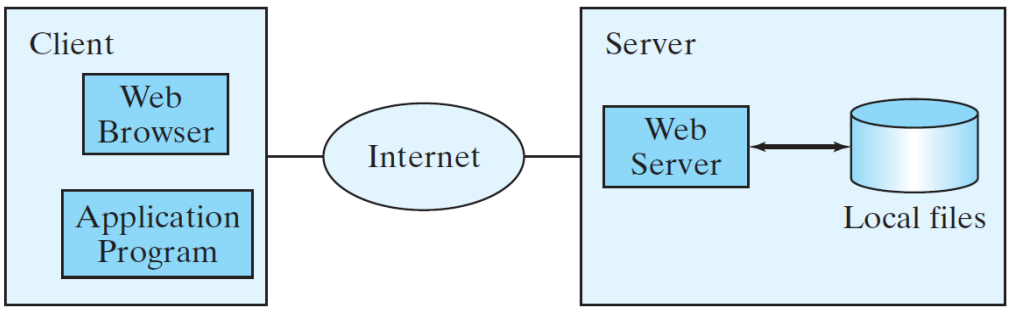
- replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

ReplaceText Run

58

12.12 Reading Data from the Web

- Just like you can read data from a file on your computer, you can read data from a file on the Web.



59

Reading Data from the Web

- For an application program to read data from a URL, you first need to create a **URL** object using the **java.net.URL** class with this constructor:

```
public URL(String spec) throws MalformedURLException
```

- A **MalformedURLException** is thrown if the URL string has a syntax error.
- For example, the following statement creates a URL object for http://www.google.com/index.html.

```
1  try {
2    URL url = new URL("http://www.google.com/index.html");
3  }
4  catch (MalformedURLException ex) {
5    ex.printStackTrace();
6  }
```

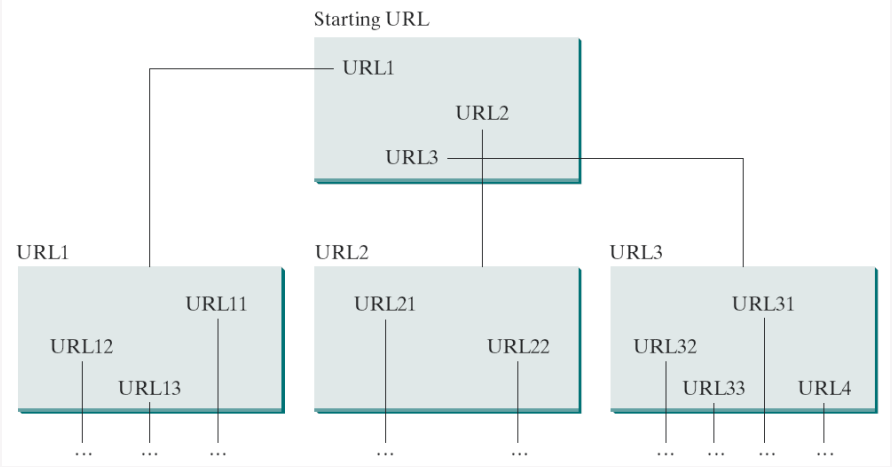
ReadFileFromURL

Run

60

Case Study: Web Crawler

- This case study develops a program that travels the Web by following hyperlinks.



61

Case Study: Web Crawler

- The program follows the URLs to traverse the Web.
- To avoid that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed.
- The algorithm for this program can be described as follows:

62

Case Study: Web Crawler

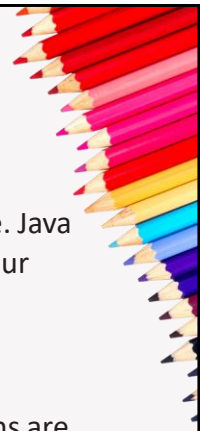
```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty and size of listOfTraversedURLs
<= 100 {
  Remove a URL from listOfPendingURLs;
  if this URL is not in listOfTraversedURLs {
    Add it to listOfTraversedURLs;
    Display this URL;
    Read the page from this URL and for each URL contained in the page {
      Add it to listOfPendingURLs if it is not in listOfTraversedURLs;
    }
  }
}
```

WebCrawler Run

63

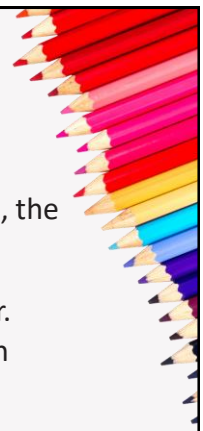
Chapter Summary

Chapter Summary



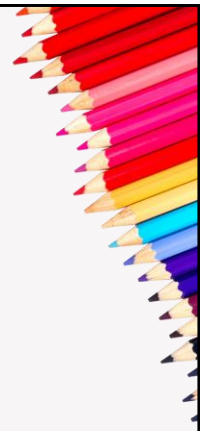
- Exception handling enables a method to throw an exception to its caller.
- A Java exception is an instance of a class derived from java.lang.Throwable. Java provides a number of predefined exception classes. You can also define your own exception class by extending Exception.
- Exceptions occur during the execution of a method.
- RuntimeException and Error are unchecked exceptions; all other exceptions are checked.
- When declaring a method, you have to declare a *checked exception if the method might* throw it, thus telling the compiler what can go wrong.
- The keyword for declaring an exception is throws, and the keyword for throwing an exception is throw.

Chapter Summary



- To invoke the method that declares checked exceptions, enclose it in a try statement. When an exception occurs during the execution of the method, the catch block catches and handles the exception.
- If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the main method.
- The code in the finally block is executed under all circumstances, regardless of whether an exception occurs in the try block or whether an exception is caught if it occurs.
- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

Chapter Summary



- The File class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.
- You can use Scanner to read string and primitive data values from a text file and use PrintWriter to create a file and write data to a text file.
- You can read from a file on the Web using the URL class.

Programming Exercises

1, 2, 3, 6, 7, 8, 9, 10, 11,

16, 19, 23, 25, 28, 30