Ch-6
Exception Handling, I/O, abstract classes and interfaces

Prepared By
Hruta N. Desai

Introduction:

- *Runtime errors occur while a program is running if the JVM detects an operation that is* impossible to carry out.

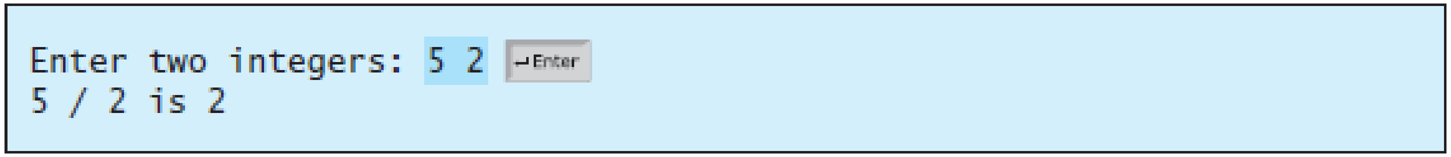
For example,
1. if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException.**
2. If you enter a double value when your program expects an integer, you will get a runtime error with an **InputMismatchException.**

- 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.

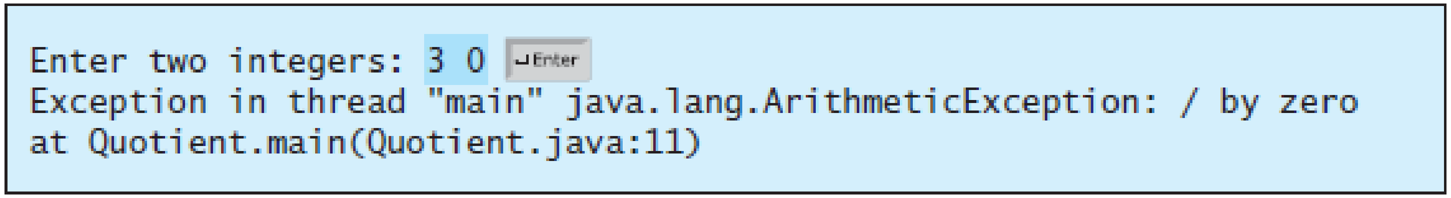
Exception Handling Overview

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

```
public class Quotient {
public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.print("Enter two integers: ");
            int number1 = input.nextInt();
            int number2 = input.nextInt();
             System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
            }
}
```



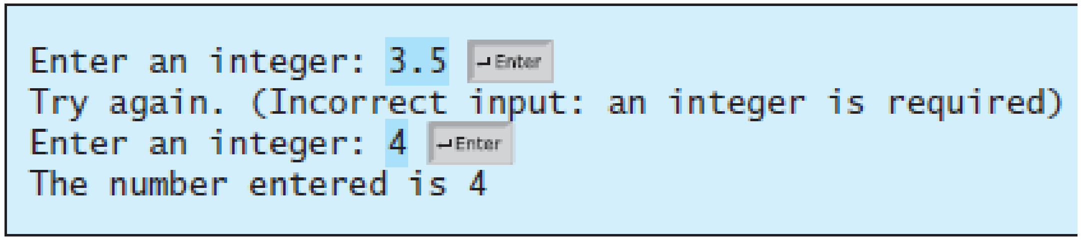
```
Enter two integers: 5 2 ↵Enter
5 / 2 is 2
```



```
Enter two integers: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:11)
```

```
public class InputMismatchExceptionDemo {
 public static void main(String[] args) { Scanner input = new Scanner(System.in);
boolean continueInput = true;
 do {
        try {
                System.out.print("Enter an integer: ");
                int number = input.nextInt();
                // Display the result
                System.out.println("The number entered is " + number);
                continueInput = false;
        }
        catch (InputMismatchException ex) {
                System.out.println("Try again. (" + "Incorrect input: an integer is
required)");
                input.nextLine(); // Discard input
        }
} while (continueInput);
}
}
```

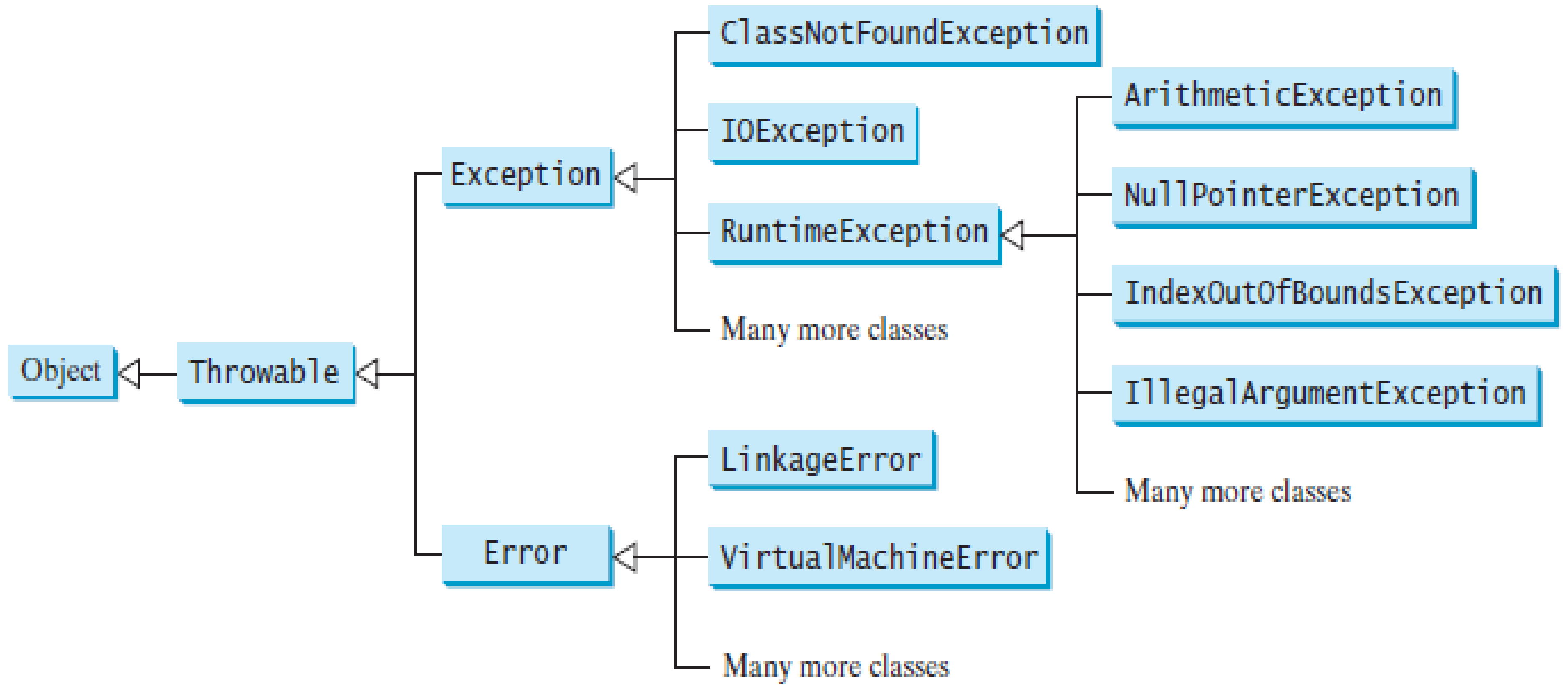
If an InputMismatch Exception occurs



```
Enter an integer: 3.5 ↵Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 ↵Enter
The number entered is 4
```

Exception Types

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



The **Throwable class is the root of exception classes. All Java exception classes inherit** directly or indirectly from **Throwable. You can create your own exception classes by extending Exception or a subclass of Exception.**

Continued..

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

1. ***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. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Example:

- **LinkageError:** A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
- **VirtualMachineError:** The JVM is broken or has run out of the resources it needs in order to continue operating.

Continued..

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

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, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedIOException, EOFException (EOF is short for End of File), and FileNotFoundException.

Continued..

*3. **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.

Example:

- **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.

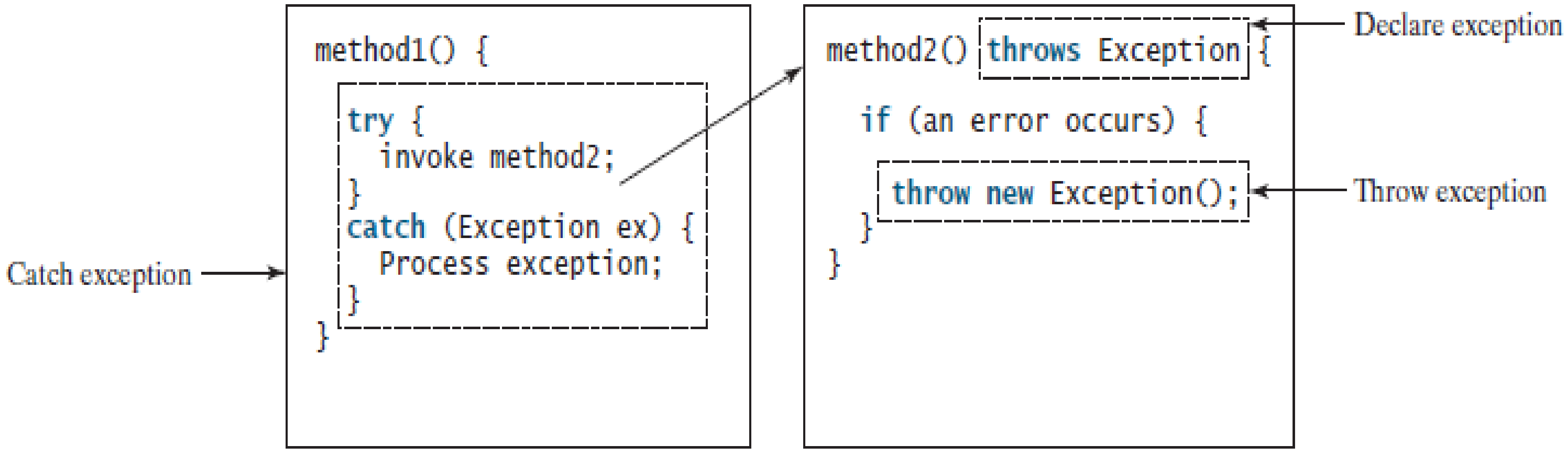
Continued..

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.

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. Unchecked exceptions can occur anywhere in a 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

- *A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.*
- Exception-handling model is based on three operations: *declaring an exception, throwing an exception, and catching an exception.*



```
method1() {

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

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Declaring Exception

1. **Declaring Exceptions:** The Java interpreter invokes the main method to start executing a program. 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. However, 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.

Example of declaring exception

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

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

2. ***Throwing exception:*** 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.*

• 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, as follows:

IllegalArgumentException ex = **new IllegalArgumentException("Wrong Argument");**

throw ex;

Or, if you prefer, you can use the following:

throw new IllegalArgumentException("Wrong Argument");

Catching Exceptions

3. **Catching Exceptions:** You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a **try-catch block.**

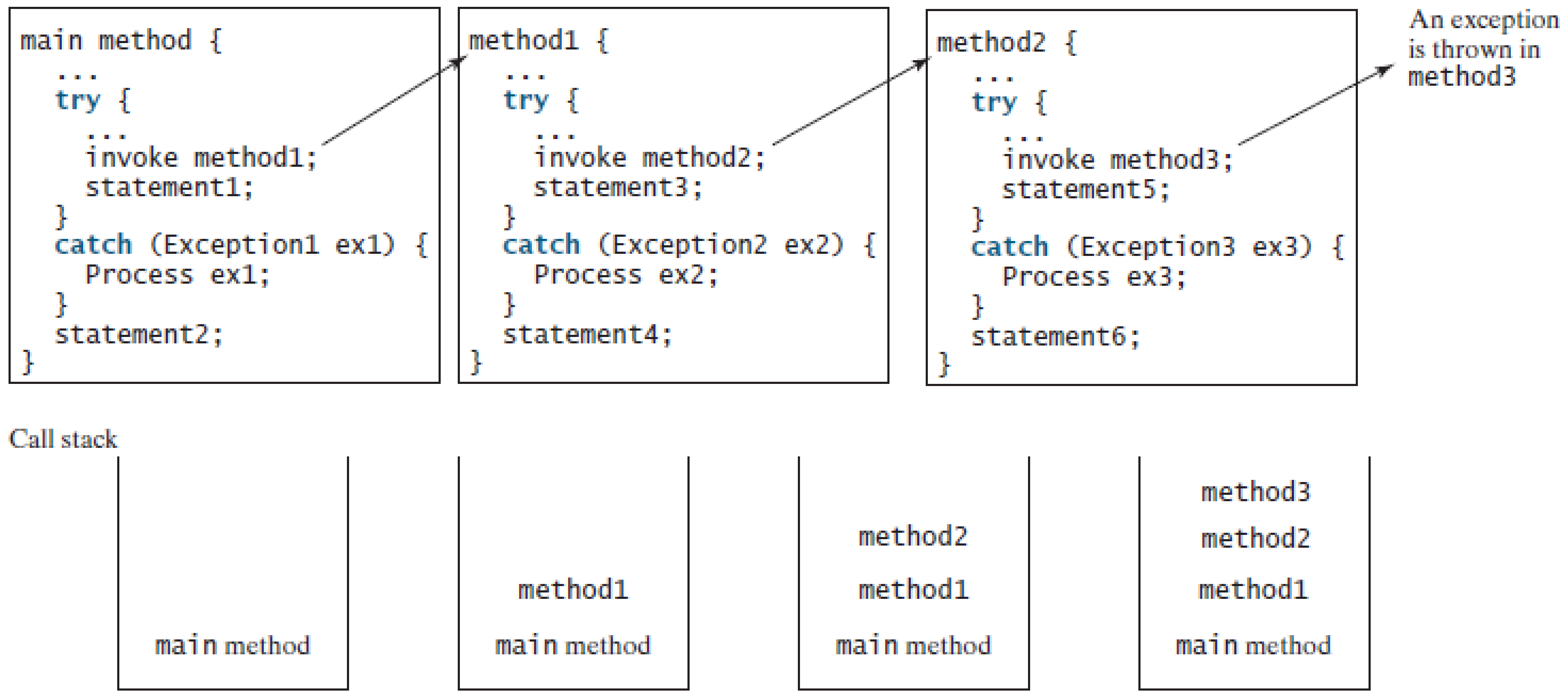
```
try {
statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
handler for exception1;
}
catch (Exception2 exVar2) {
handler for exception2;
}
...
catch (ExceptionN exVarN) {
handler for exceptionN;
}
```

Continued..

- 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.*

Exception handling and method call

Suppose the **main method invokes method1, method1 invokes method2, method2** invokes **method3, and method3 throws an exception. Consider the** following scenario:



```
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;
}
```

Continued..

- If the exception type is Exception3, it is caught by the catch block for handling exception ex3 in method2. statement5 is skipped, and statement6 is executed.
- If the exception type is Exception2, method2 is aborted, the control is returned to method1, and the exception is caught by the catch block for handling exception ex2 in method1. statement3 is skipped, and statement4 is executed.
- If the exception type is Exception1, method1 is aborted, the control is returned to the main method, and the exception is caught by the catch block for handling exception ex1 in the main method. statement1 is skipped, and statement2 is executed.
- If the exception type is not caught in method2, method1, or main, the program terminates, and statement1 and statement2 are not executed.

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.**

Exception Handling

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

(a) Wrong order

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

(b) Correct order

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

(a) Catch exception

```java
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

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.**

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

Continued..

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 3 possible cases:

■ If no exception arises in the try block, final Statements is executed, and the next statement after the try statement is executed.

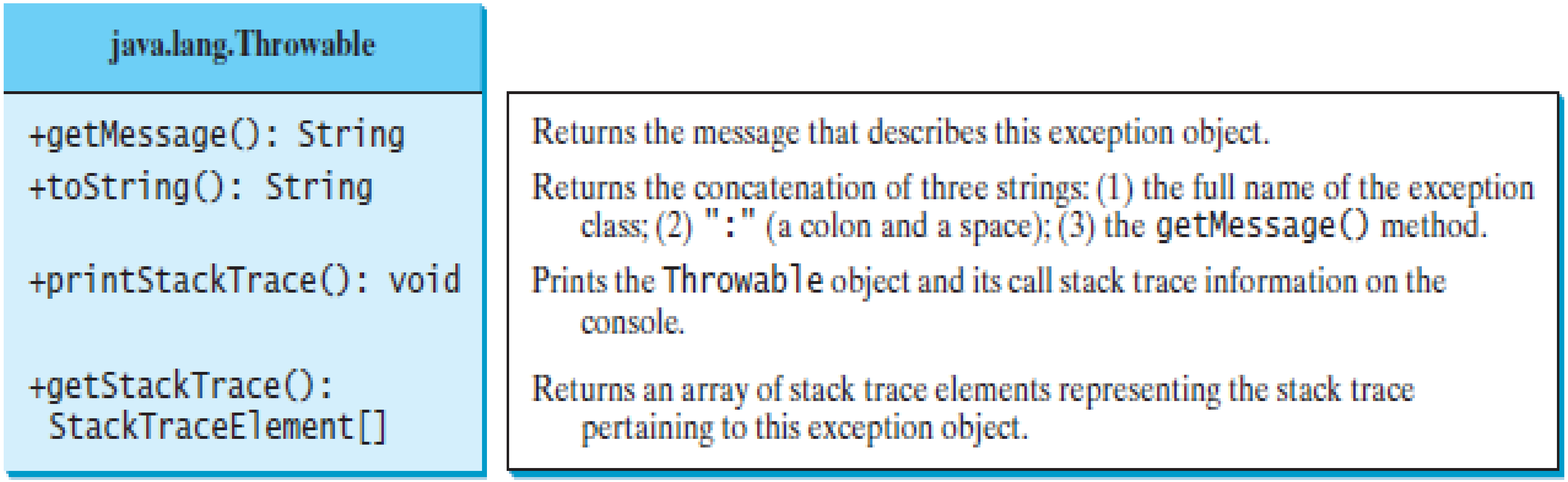
■ If a statement causes an exception in the try block that is caught in a catch block, the rest of the statements in the try block are skipped, the catch block is executed, and the finally clause is executed. The next statement after the try statement is executed.

■ 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.

The finally block executes even if there is a return statement prior to reaching the finally block.

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



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.

When to Use Exceptions

- The **try block contains the code that is executed in normal circumstances. The catch block** contains the code that is executed in exceptional circumstances.
- 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 exception through the chain of methods invoked to search for the handler.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled without throwing exceptions. This can be done by using **if statements to check for errors.**

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

is better replaced by

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

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.

Example:

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

Note: 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.

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.*

Example Chained Exception

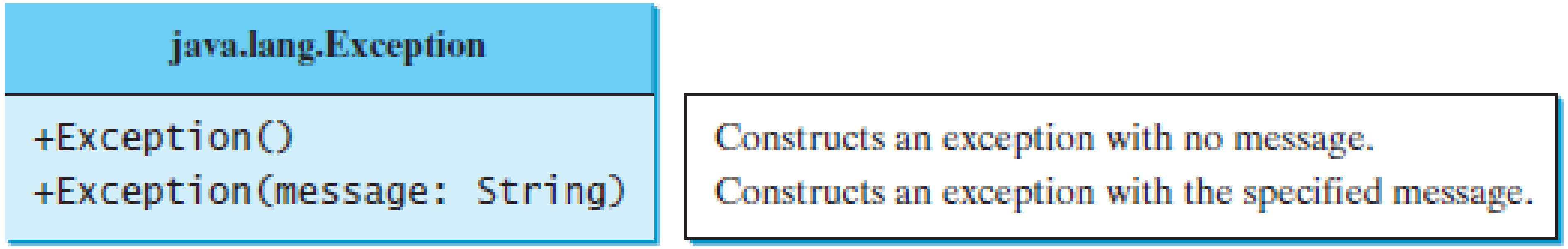
```
public static void main(String[] args) {
 try {
		method1();
}
catch (Exception ex) {
		ex.printStackTrace();
 }
 }
 public static void method1() throws Exception {
 try {
		 method2();
 }
 catch (Exception ex) {
		 throw new Exception("New info from method1", ex);
 }
public static void method2() throws Exception {
throw new Exception("New info from method2");
}
```

Defining Custom Exception Classes

*You can define a custom exception class by extending the **java.lang.Exception** class.*

- 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.**

Constructors:



java.lang.Exception

```
+Exception()
+Exception(message: String)
```

Constructs an exception with no message.
Constructs an exception with the specified message.

Example of Custom Exception

```
public class InvalidRadiusException extends Exception {
 private double radius;
 /** Construct an exception */
 public InvalidRadiusException(double radius) {
          super("Invalid radius " + radius);
          this.radius = radius;
}
/** Return the radius */
 public double getRadius() {
          return radius;
}
}
```

The File Class

The ***File class contains the methods for obtaining the properties of a file/directory*** *and for renaming and deleting a file/directory.*

•Every file is placed in a directory in the file system. **An** ***absolute file name*** *(or full name) contains a file name with its complete path and drive letter.*

For example, ***c:\book*****\Welcome.java is the absolute file name for the file Welcome.java** on the Windows operating system.

Here **c:\book** is referred to as the ***directory path for the file.*** *Absolute* file names are machine dependent.

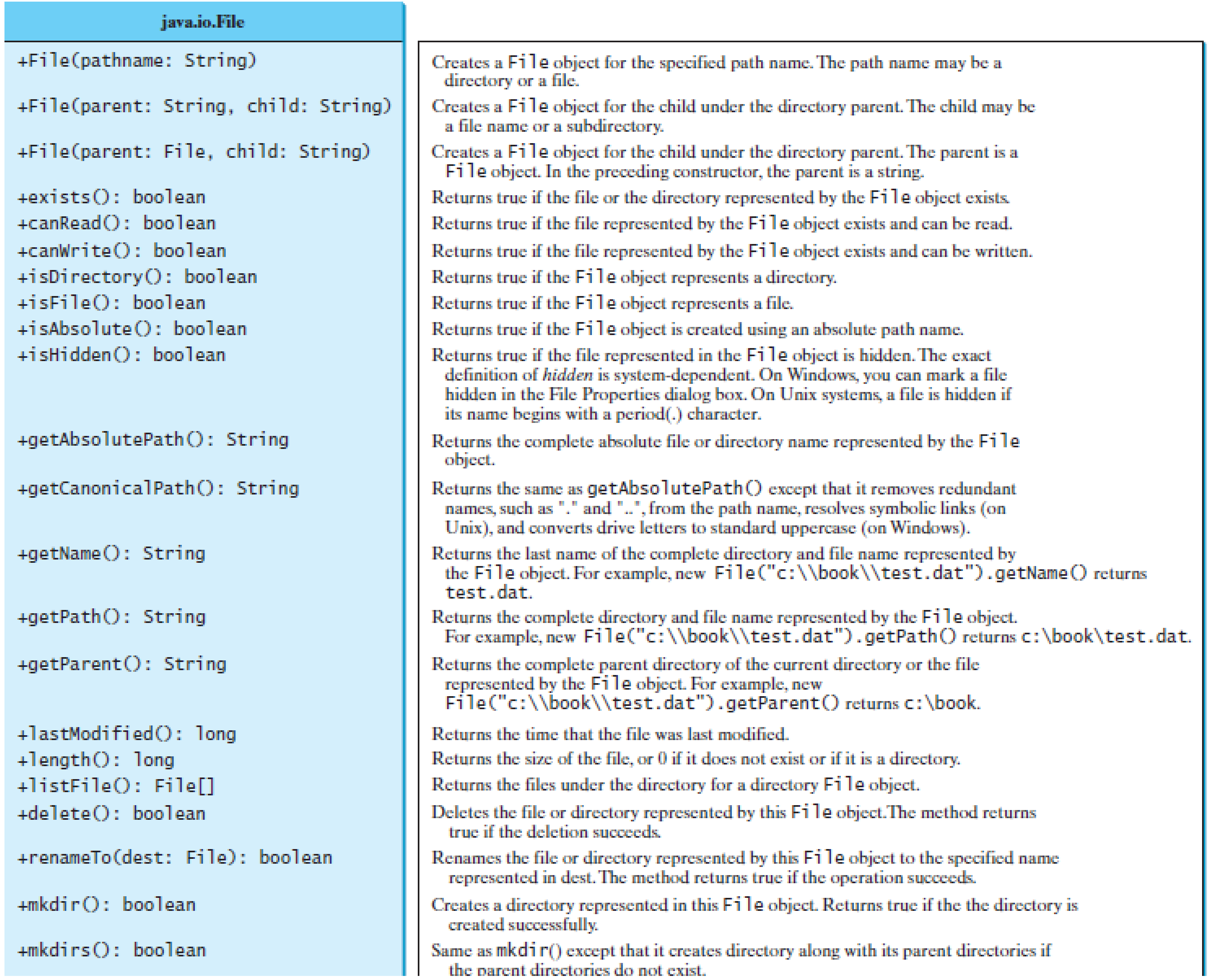
• A *relative file name is in relation to the current working directory.*

For example, **Welcome.java** is a **relative file name**.

Continued..

•*The* ***File class does not*** *contain the methods for reading and writing file contents.*

• The file name 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.

Methods used in File Class

File Input and Output

*Use the **Scanner class for reading text data from a file and the PrintWriter class** for writing text data to a file.*

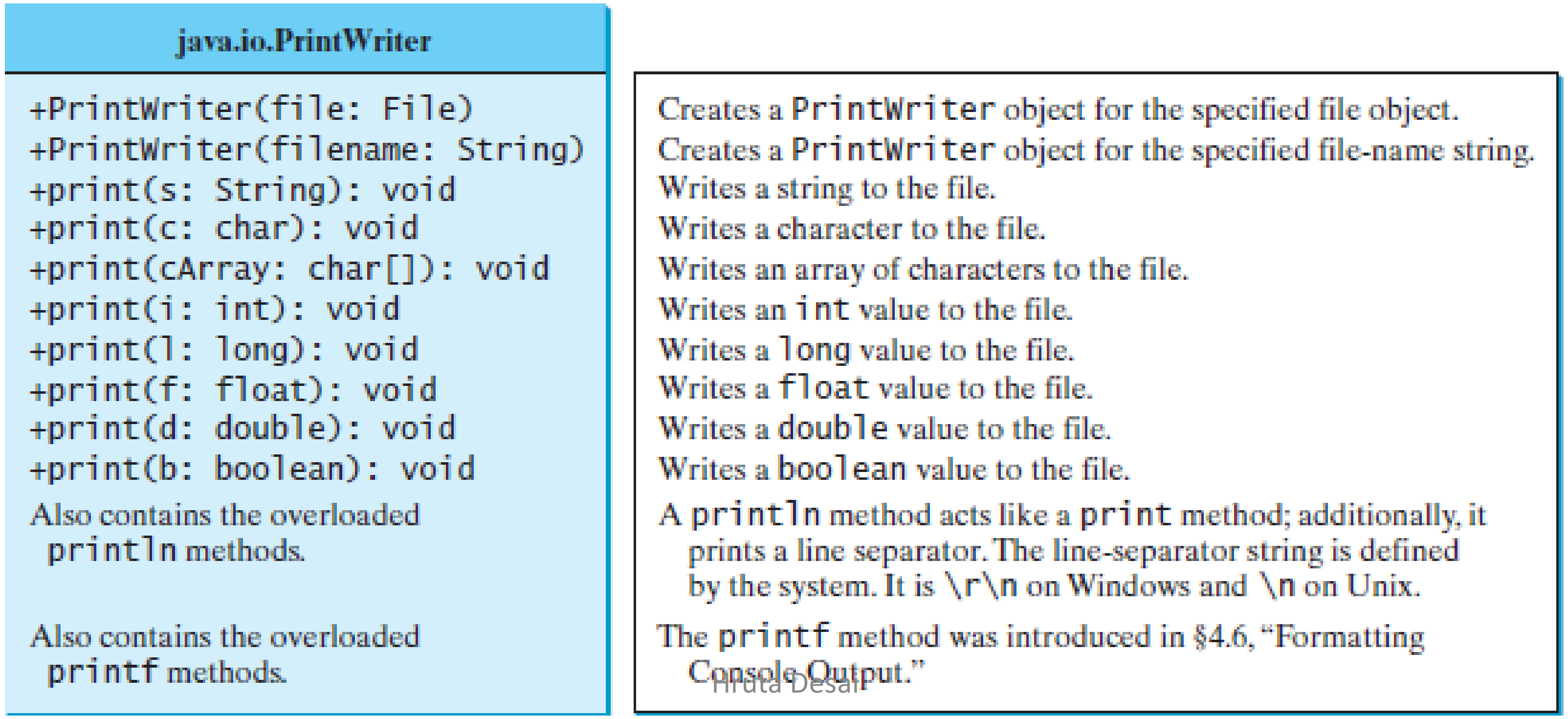
• A **File object encapsulates the properties of a file or a path, but it does not contain the methods** for creating a file or for writing/reading data to/from a file (referred to as data *input and output, or I/O for short).*
• *In order to perform I/O, you need to create objects using appropriate* Java I/O classes. The objects contain the methods for reading/writing data from/to a file.
• There are two types of files: text and binary. Text files are essentially characters on disk. This
section introduces how to read/write strings and numeric values from/to a text file using the
Scanner and PrintWriter classes.

Writing Data Using **PrintWriter**

The **java.io.PrintWriter class can be used to create a file and write data to a text file.**

PrintWriter output = **new PrintWriter(filename);**

Then, you can invoke the **print, println, and printf methods on the PrintWriter object** to write data to a file.



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."

Example

```
import java.util.Scanner;
public class ReadData {
 public static void main(String[] args) throws Exception {
        // Create a File instance
        java.io.File file = new java.io.File("scores.txt");

        // Create a Scanner for the file
        Scanner input = new Scanner(file);
        // Read data from a file
        while (input.hasNext()) {
                String firstName = input.next();
                String mi = input.next();
                String lastName = input.next();
                int score = input.nextInt();
                System.out.println( firstName + " " + mi + " " + lastName + " " + score);
         }
        // Close the file
         input.close();
 }
 }
```

How Does **Scanner Work?**

new Scanner(String) creates a Scanner for a given string. To create a Scanner to read data from a file, you have to use the java.io.File class to create an instance of the File using the constructor **new File(filename),** use **new Scanner(File)** to create a **Scanner for the file.**

• The **nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), next-Double(), and next() methods are known as *token-reading methods, because they read*** tokens separated by delimiters. By default, the delimiters are whitespace characters. You can use the **useDelimiter(String regex) method to set a new pattern for delimiters.**

Continued..

- Token-reading method first skips any delimiters (whitespace characters by default), then reads a token ending at a delimiter. The token is then automatically converted into a value of the **byte, short, int, long, float, or double type for nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), and nextDouble(),** respectively.
- For the **next() method, no conversion is performed. If the token does** not match the expected type, a runtime exception **java.util.InputMismatchException** will be thrown.

- The token-reading method does not read the delimiter after the token. If the **nextLine()** method is invoked after a token-reading method, this method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by **nextLine().**

Example of Scanner

Suppose a text file named test.txt contains a line
34 567
After the following code is executed,

```
        Scanner input = new Scanner(new File("test.txt"));
         int intValue = input.nextInt();
         String line = input.nextLine();
```

intValue contains 34 and line contains the characters ' ', 5, 6, and 7.
What happens if the input is *entered from the keyboard? Suppose you enter 34, press the Enter key, then enter 567 and press the Enter key for the following code:*

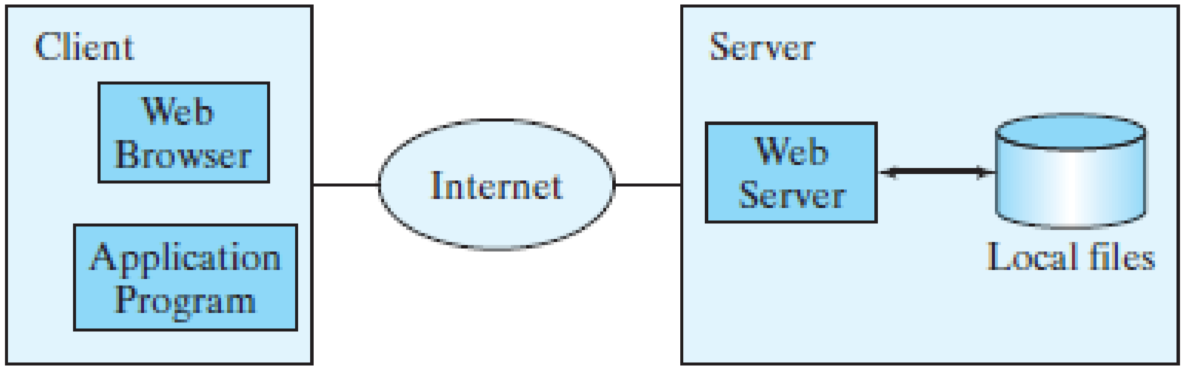
```
         Scanner input = new Scanner(System.in);
         int intValue = input.nextInt();
         String line = input.nextLine();
```

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.

• In addition to reading data from a local file on a computer or file server, you can also access
data from a file that is on the Web if you know the file’s URL (Uniform Resource Locator— the unique address for a file on the Web).

For example, www.google.com/index.html is the URL for the file index.html located on the Google Web server. When you enter the URL in a Web browser, the Web server sends the data to your browser, which renders the data graphically.



The client retrieves files from a Web server.

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

A MalformedURLException is thrown if the URL string has a syntax error.

For example:

The URL string “http:www.google.com/index.html” would cause a MalformedURLException runtime error because two slashes (//) are required after the colon (:). Note that the http:// prefix is required for the URL class to recognize a valid URL. It would be wrong if you replace line 2 with the following code:

```
URL url = new URL("www.google.com/index.html");
```

After a URL object is created, you can use the openStream() method defined in the URL class to open an input stream and use this stream to create a Scanner object as follows:

```
Scanner input = new Scanner(url.openStream());
```

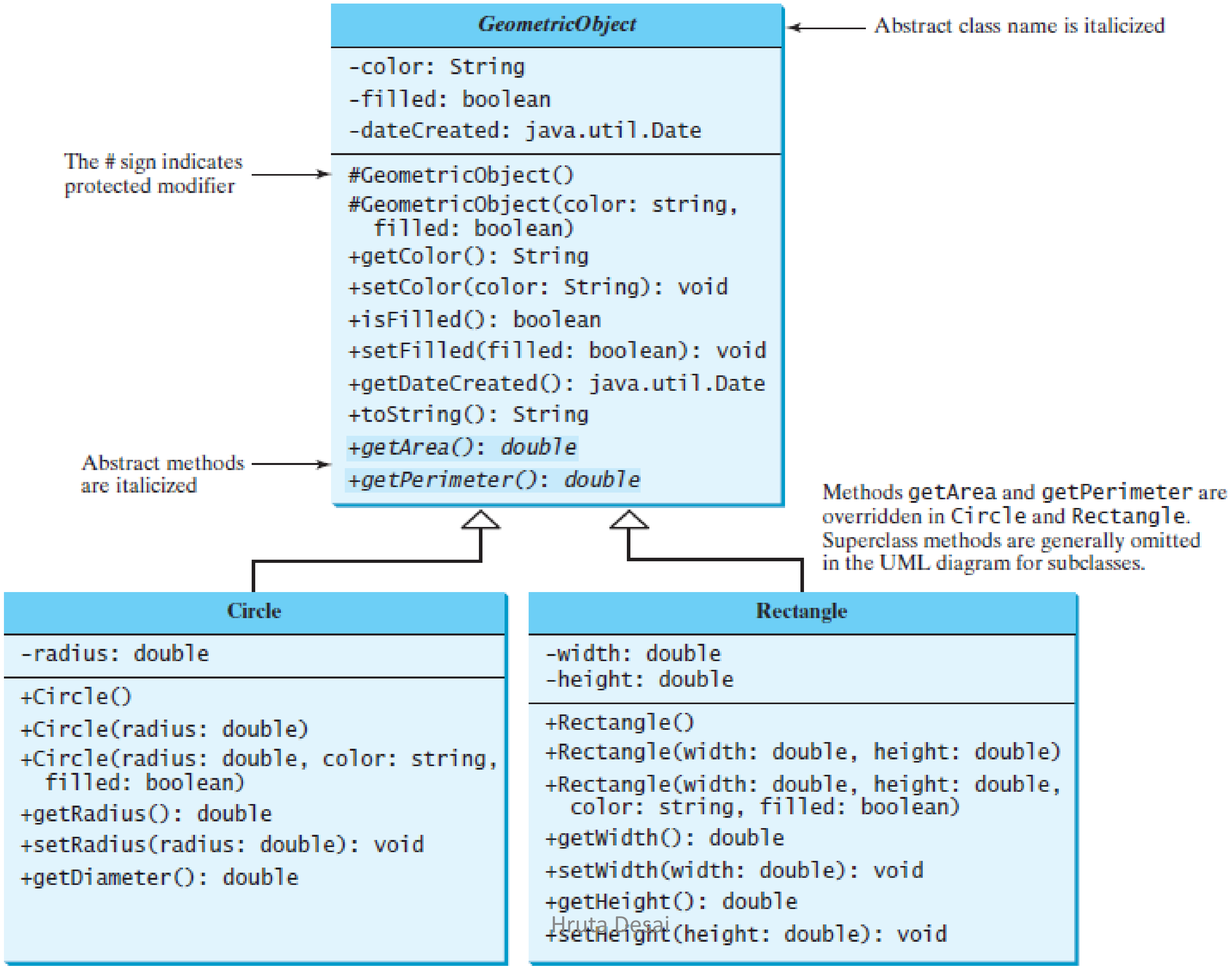
Now you can read the data from the input stream just like from a local file.

Abstract Classes

An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in concrete subclasses.

- Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an *abstract class.*
- Abstract classes are denoted using the **abstract modifier in the** class header.

Methods in abstract class

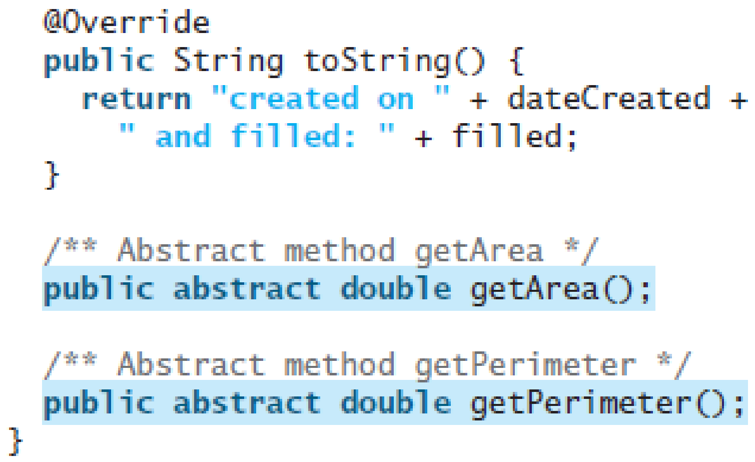


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Abstract class name is italicized
The # sign indicates protected modifier
Abstract methods are italicized
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

Example

```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }
```

Continued..



```
  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

• Abstract classes are like regular classes, but you cannot create instances of abstract classes using the **new operator. An abstract method is defined without implementation. Its implementation** is provided by the subclasses. A class that contains abstract methods must be defined as abstract.

• The constructor in the abstract class is defined as protected, because it is used only by subclasses. When you create an instance of a concrete subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

Interesting Points about Abstract class

- An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note that abstract methods are nonstatic.
- An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

Continued..

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn’t contain any abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining subclasses.
- A subclass can override a method from its superclass to define it as abstract. This is *very unusual, but it is useful when the implementation of the method in the superclass* becomes invalid in the subclass. In this case, the subclass must be defined as abstract.
- A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

Continued..

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the GeometricObject type, is correct.

GeometricObject[] objects = new GeometricObject[10];

You can then create an instance of GeometricObject and assign its reference to the array like this:

objects[0] = new Circle();

Interfaces

An interface is a class-like construct that contains only constants and abstract methods.

• In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes. For example, using appropriate interfaces, you can specify that the objects are comparable, edible, and/or cloneable.

• Syntax to define an interface:

modifier **interface InterfaceName {**
/** Constant declarations */
/** Abstract method signatures */
}

• Here is an example of an interface:

public interface Edible {
/** Describe how to eat */
public abstract String howToEat();
}

Continued..

- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.
- You can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a reference variable, as the result of casting, and so on.
- As with an abstract class, you cannot create an instance from an interface using the **new operator.**
- The relationship between the class and the interface is known as *interface inheritance. Since interface inheritance and class inheritance are essentially the* same, we will simply refer to both as *inheritance.*

The **Comparable Interface**

*The **Comparable interface defines the compareTo method for comparing objects.***

• Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares. In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable. Java provides the **Comparable interface for this purpose.**

```
// Interface for comparing objects, defined in java.lang
package java.lang;
public interface Comparable<E> {
public int compareTo(E o);
}
```

Continued..

- The compareTo method determines the order of this object with the specified object o and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o.
- The Comparable interface is a generic interface. The generic type E is replaced by a concrete type when implementing this interface. Many classes in the Java library implement Comparable to define a natural order for objects.

Example of Comparable Interfaces

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

You can use the **compareTo** method to compare two numbers, two strings, and two dates.

Example of Comparable Interfaces

```
System.out.println(new Integer(3).compareTo(new
Integer(5)));
 System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
 java.util.Date date2 = new java.util.Date(2012, 1, 1);
 System.out.println(date1.compareTo(date2));
```

Output:

```
-1
-2
1
```

The **Cloneable Interface**

*The **Cloneable interface specifies that an object can be cloned.***

- Often it is desirable to create a copy of an object. To do this, you need to use the **clone**
method and understand the **Cloneable interface.**
- An interface contains constants and abstract methods, but the **Cloneable interface is a** special case. The **Cloneable interface in the java.lang package is defined as follows:**

```
package java.lang;
public interface Cloneable {
}
```

Continued..

- This interface is empty. An interface with an empty body is referred to as a *marker interface.*
- A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties.
- A class that implements the **Cloneable interface** is marked cloneable, and its objects can be cloned using the **clone()** method defined in the Object class.

Example-1

```
Calendar calendar = new GregorianCalendar(2013, 2, 1);
Calendar calendar1 = calendar;
Calendar calendar2 = (Calendar)calendar.clone();
System.out.println("calendar == calendar1 is " +  (calendar ==
calendar1));
System.out.println("calendar == calendar2 is " + (calendar ==
calendar2));
System.out.println("calendar.equals(calendar2) is " +
calendar.equals(calendar2));
```

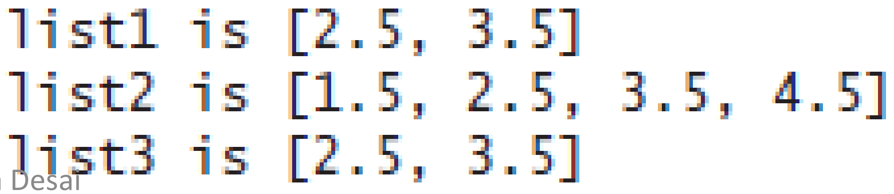
Output:

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```

Example - 2

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 is " + list1);
 System.out.println("list2 is " + list2);
System.out.println("list3 is " + list3);
```

Output:



```
list1 is [2.5, 3.5]
list2 is [1.5, 2.5, 3.5, 4.5]
list3 is [2.5, 3.5]
```

Example-3

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 is " + list1[0] + ", " + list1[1]);
System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

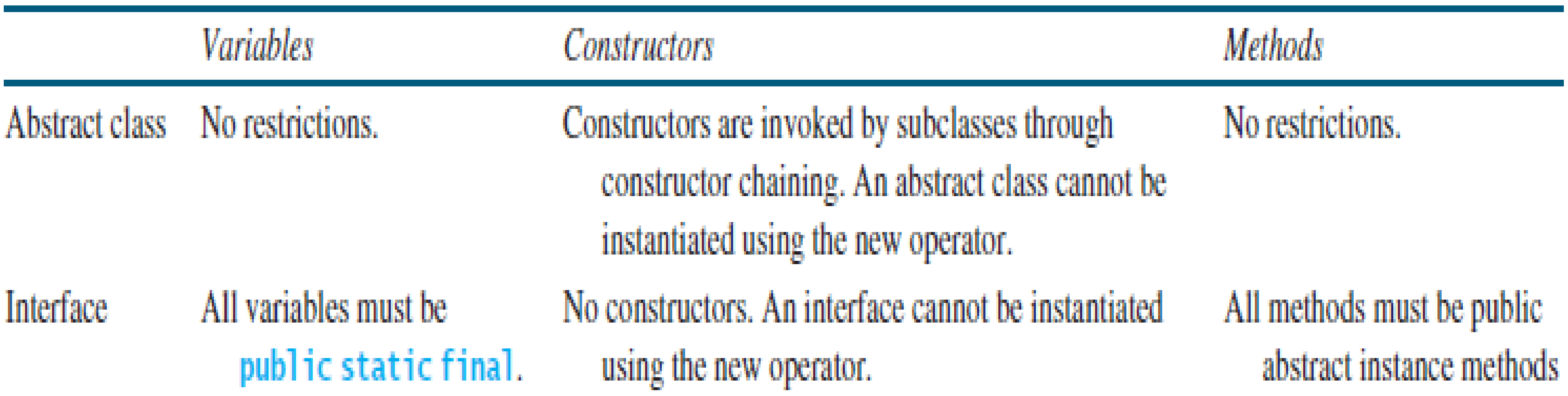
Output:

```
list1 is 7, 2
list2 is 1, 8
```

Interfaces vs. Abstract Classes

A class can implement multiple interfaces, but it can only extend one superclass.

• An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class.



	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

• Java allows only *single inheritance for class extension but allows multiple extensions for* interfaces. For example,

```
public class NewClass extends BaseClass
implements Interface1, ..., InterfaceN {
          ...
}
```

• An interface can inherit other interfaces using the **extends keyword. Such an interface is** called a *subinterface. For example,* ***NewInterface in the following code is a subinterface of*** **Interface1, . . . , and InterfaceN.**

```
public interface NewInterface extends Interface1, ... , InterfaceN {
          // constants and abstract methods
}
```

• A class implementing **NewInterface must implement the abstract methods defined in NewInterface, Interface1, . . . , and InterfaceN. An interface can extend other interfaces** but not classes. A class can extend its superclass and implement multiple interfaces.

Thank You