Chapter 12 Exception Handling and Text I/O

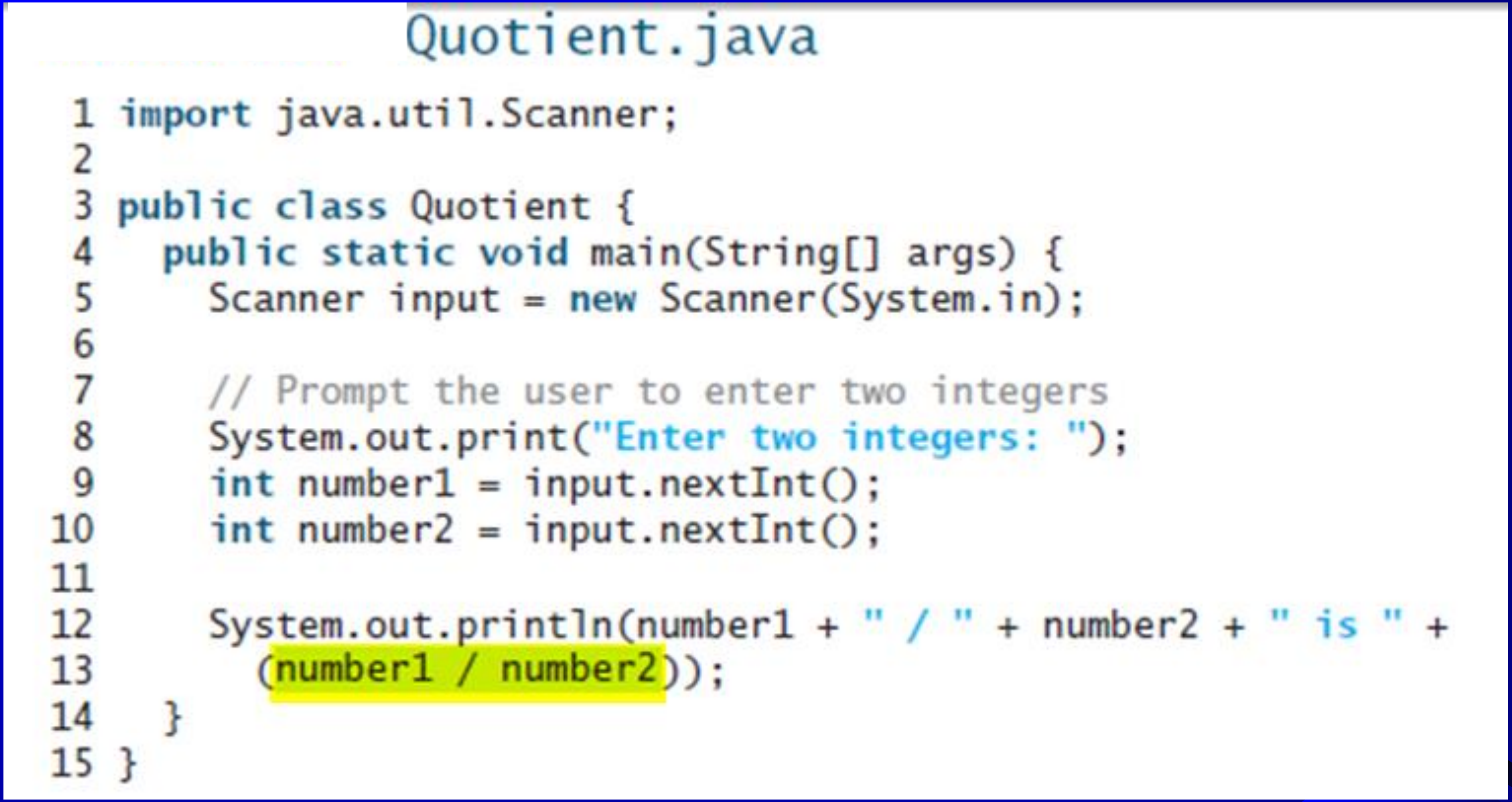
Motivations

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?

This is the subject we will introduce in this chapter.

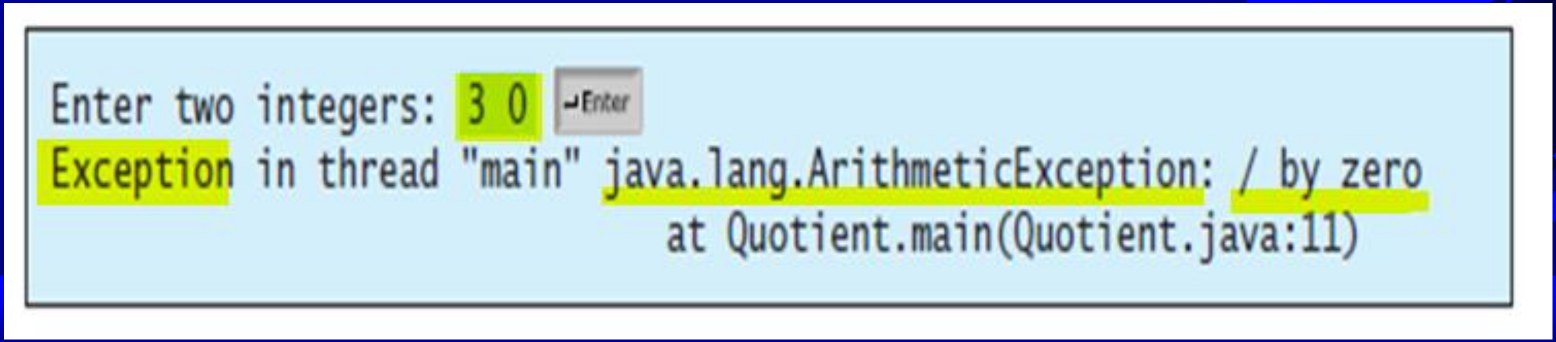
Exception-Handling Overview



Quotient.java

```java
 1 import java.util.Scanner;
 2 
 3 public class Quotient {
 4   public static void main(String[] args) {
 5     Scanner input = new Scanner(System.in);
 6 
 7     // Prompt the user to enter two integers
 8     System.out.print("Enter two integers: ");
 9     int number1 = input.nextInt();
10     int number2 = input.nextInt();
11 
12     System.out.println(number1 + " / " + number2 + " is " +
13       (number1 / number2));
14   }
15 }
```

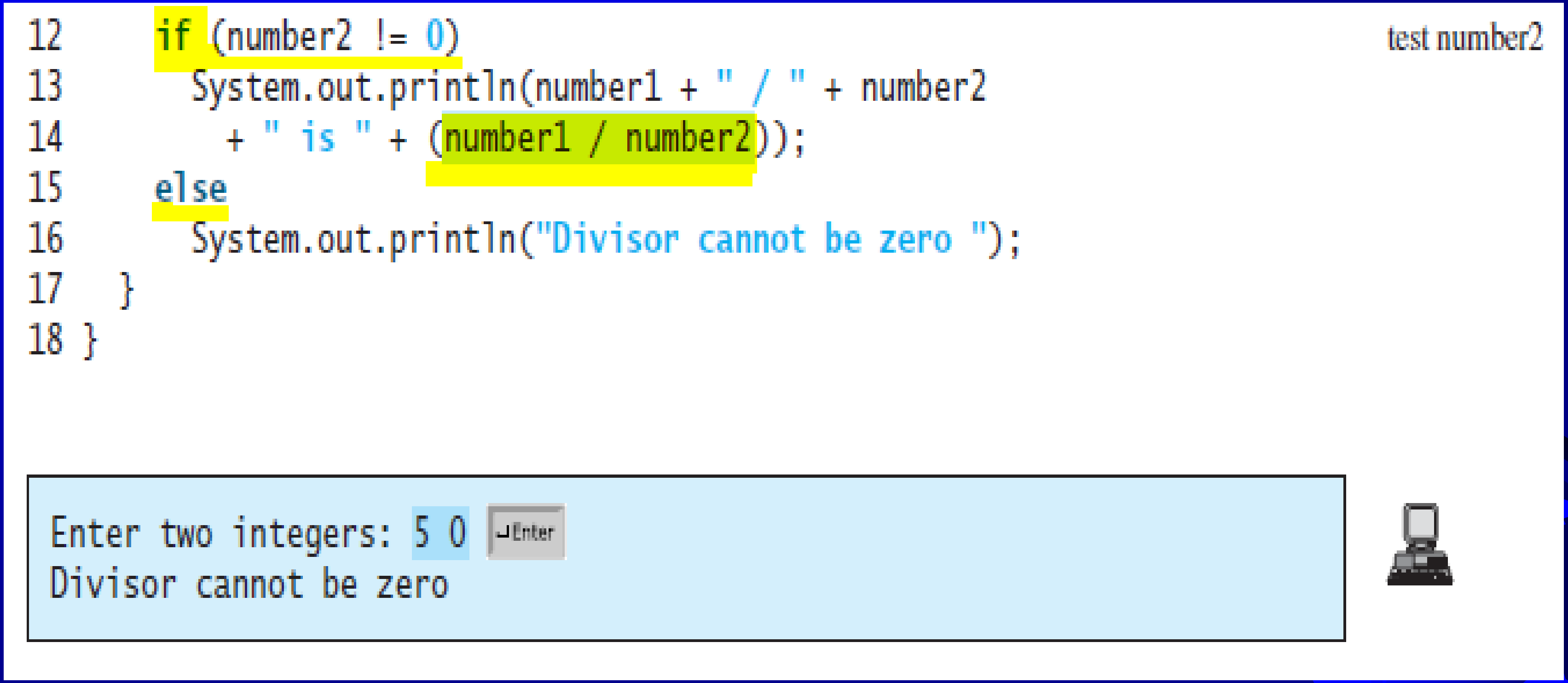
Abort， Crash！ Show runtime-error



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

Exception-Handling Overview

Fix runtime-error using an if statement

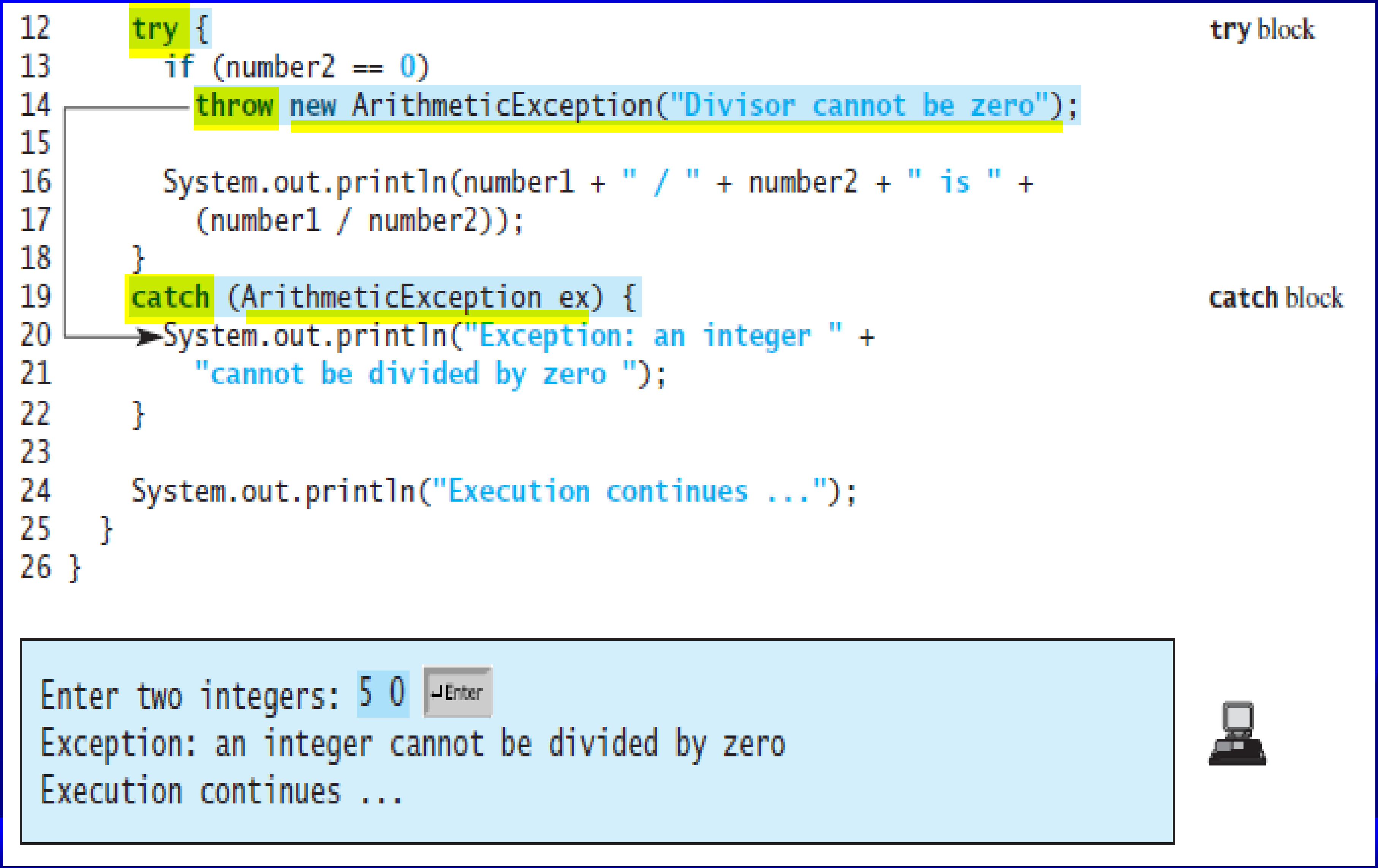


```java
    if (number2 != 0)
      System.out.println(number1 + " / " + number2
        + " is " + (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

```
Enter two integers: 5 0
Divisor cannot be zero
```

Exception-Handling Overview

Use exception-handling statement



```
    try {
      if (number2 == 0)
        throw new ArithmeticException("Divisor cannot be zero");

      System.out.println(number1 + " / " + number2 + " is " +
        (number1 / number2));
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

```
Enter two integers: 5 0
Exception: an integer cannot be divided by zero
Execution continues ...
```

Exception-Handling Overview

An exception:

new ArithmeticException("**Divisor cannot be zero**");

- an object of exception class **java.lang.ArithmeticException**
- Exception message: describe the exception

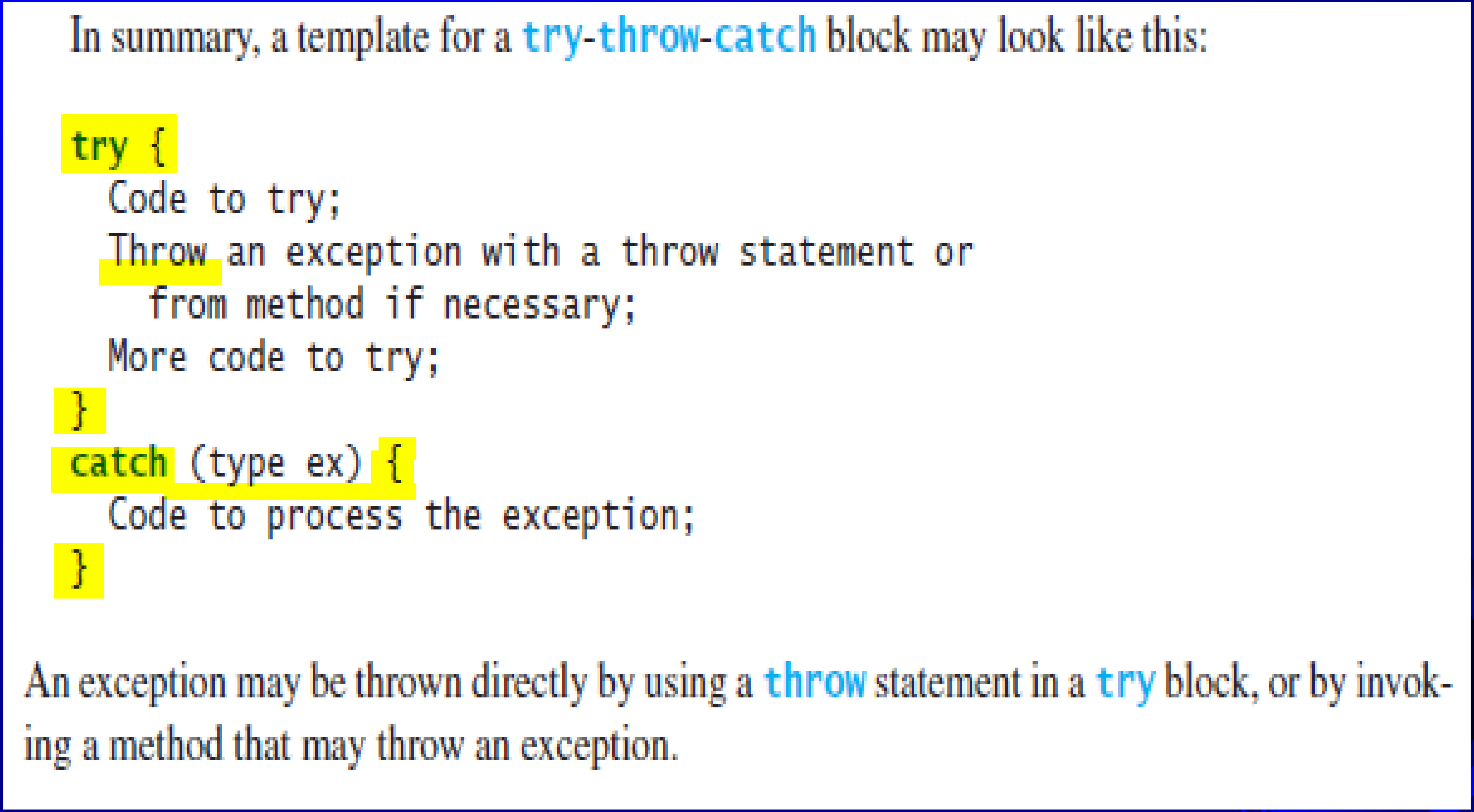
try block :

- Throw exception
- executed in normal circumstances.

catch block :

- Handle exception
- executed when **exception occurs**.
- Afterward, line24 is executed

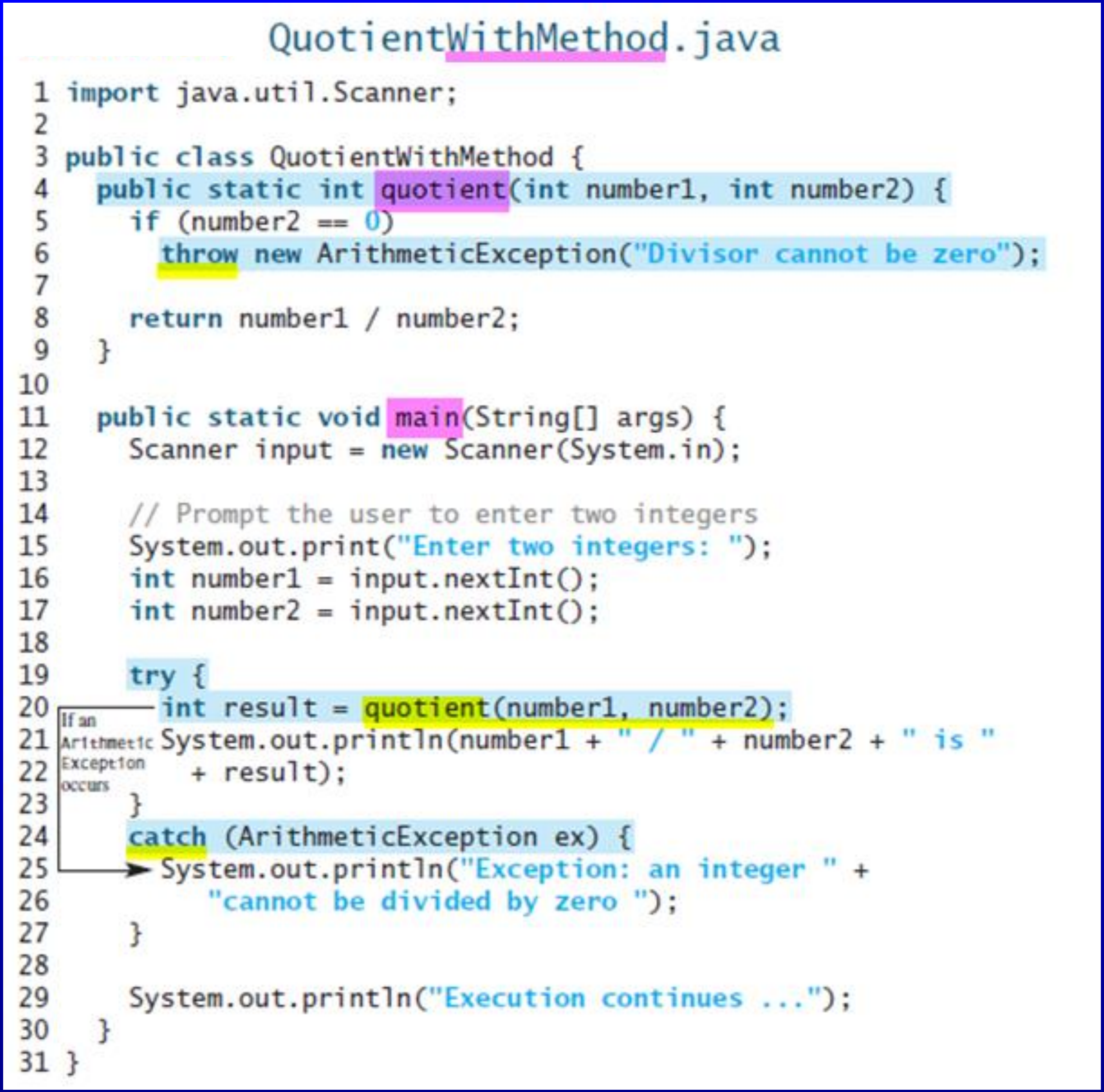
Exception-Handling Overview



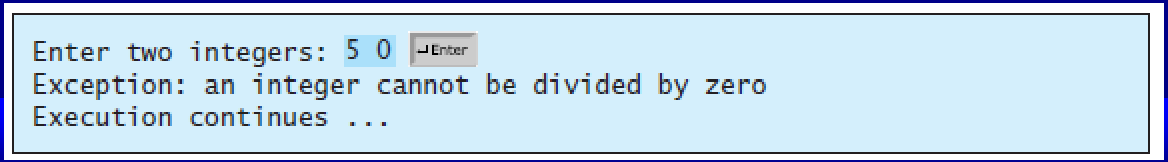
In summary, a template for a try-throw-catch block may look like this:

```
try {
  Code to try;
  Throw an exception with a throw statement or
    from method if necessary;
  More code to try;
}
catch (type ex) {
  Code to process the exception;
}
```

An exception may be thrown directly by using a throw statement in a try block, or by invoking a method that may throw an exception.


QuotientWithMethod.java

```java
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```



```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

- Callee: quotient(), throw exception
- Caller: main(), handle exception

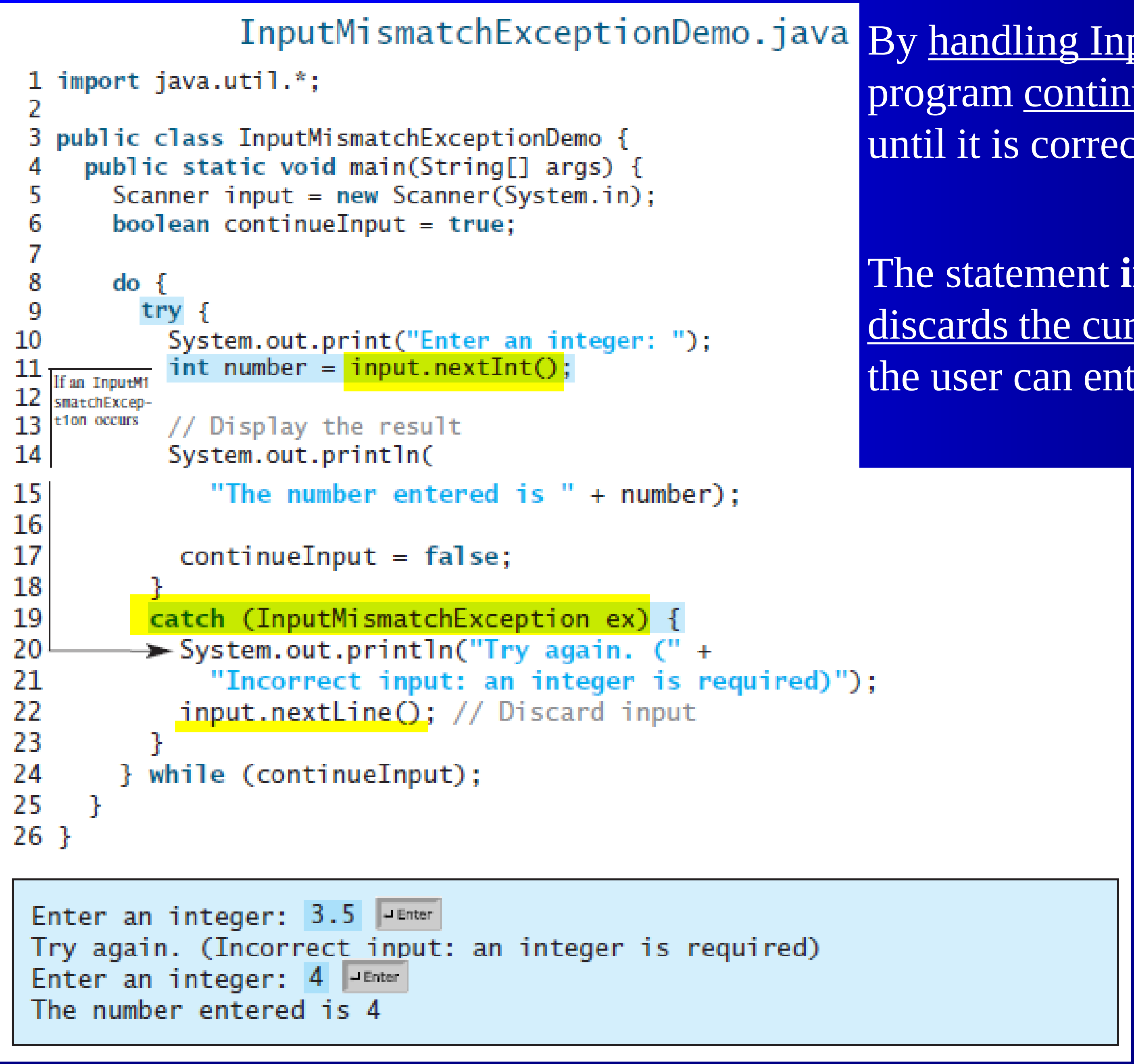
Exception Advantages

Typically, a callee (e.g. library method) can detect the error , but only caller knows how to handle the error.

Advantages of exception handling: **separate** the **detection and handling** of error; enable a callee to throw an exception to its caller

- Callee: (detect error) throw exception
- Caller: (handle error) handle exception

- Without this capability, a callee must handle the exception by itself or terminate the program.


InputMismatchExceptionDemo.java
If an InputMismatchException occurs

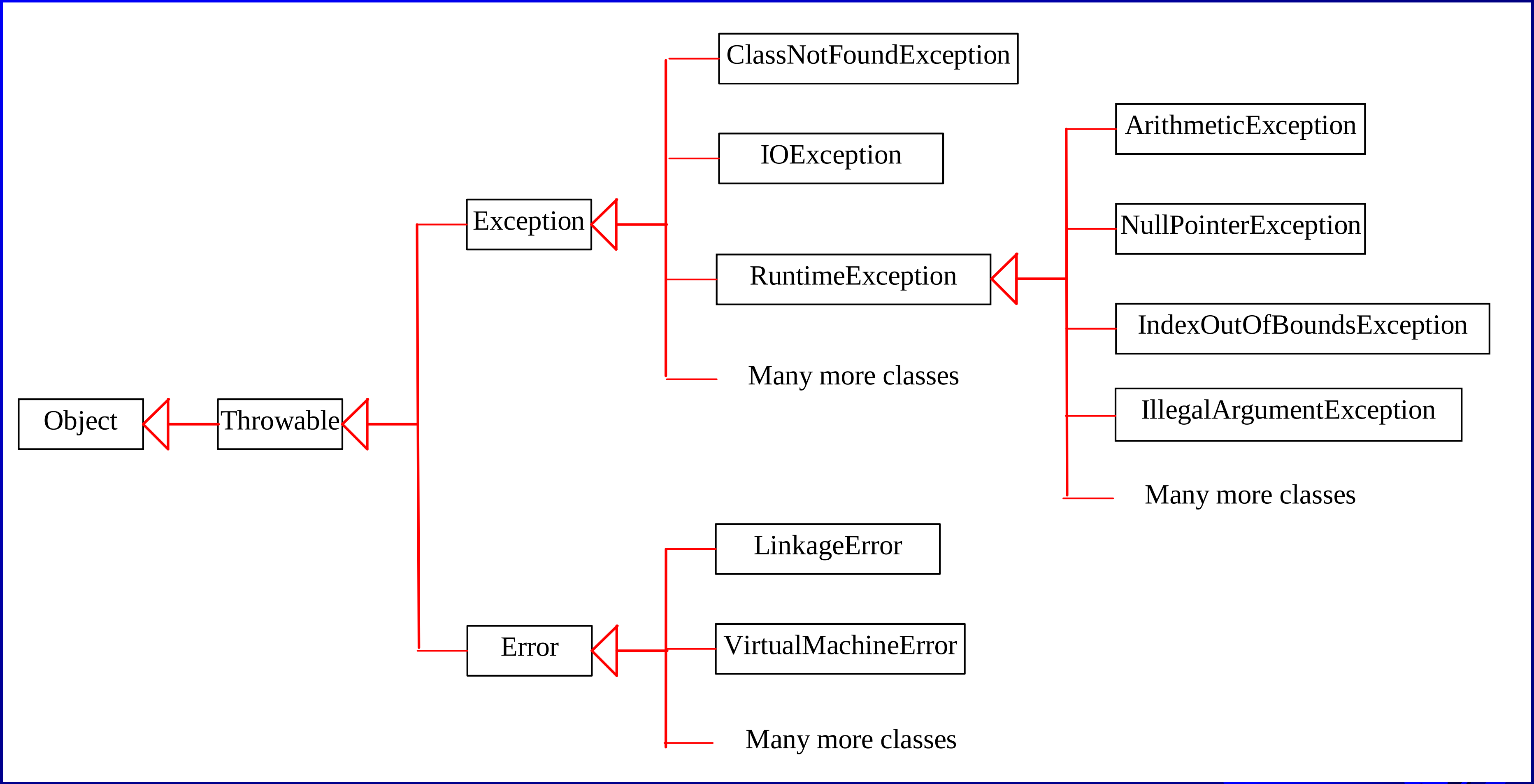
```java
import java.util.*;

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);
  }
}
```

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

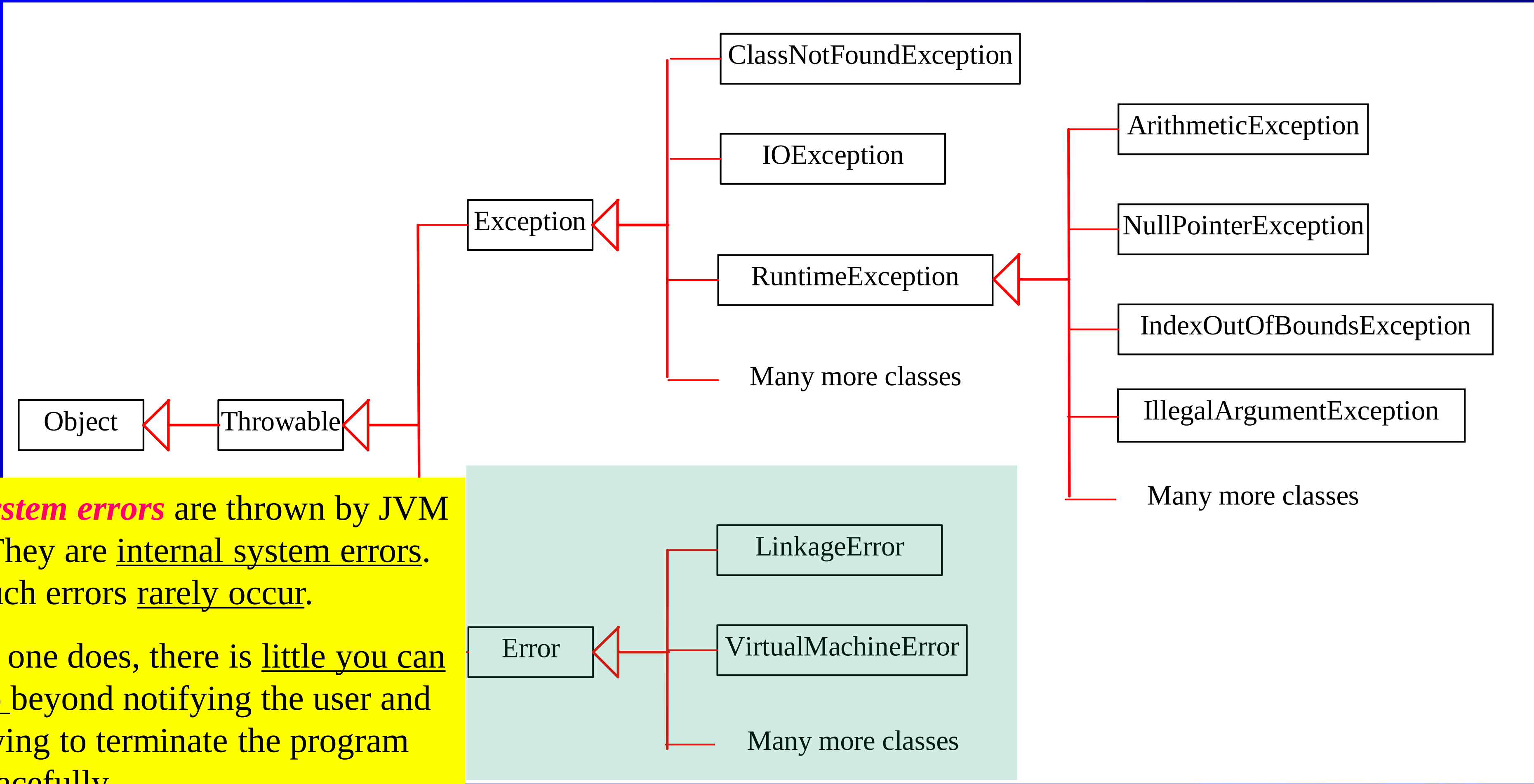
By handling InputMismatchException, program continuously read an input until it is correct.

The statement **input.nextLine()** discards the current input line so that the user can enter a new line of input.

Exception Types



System Errors

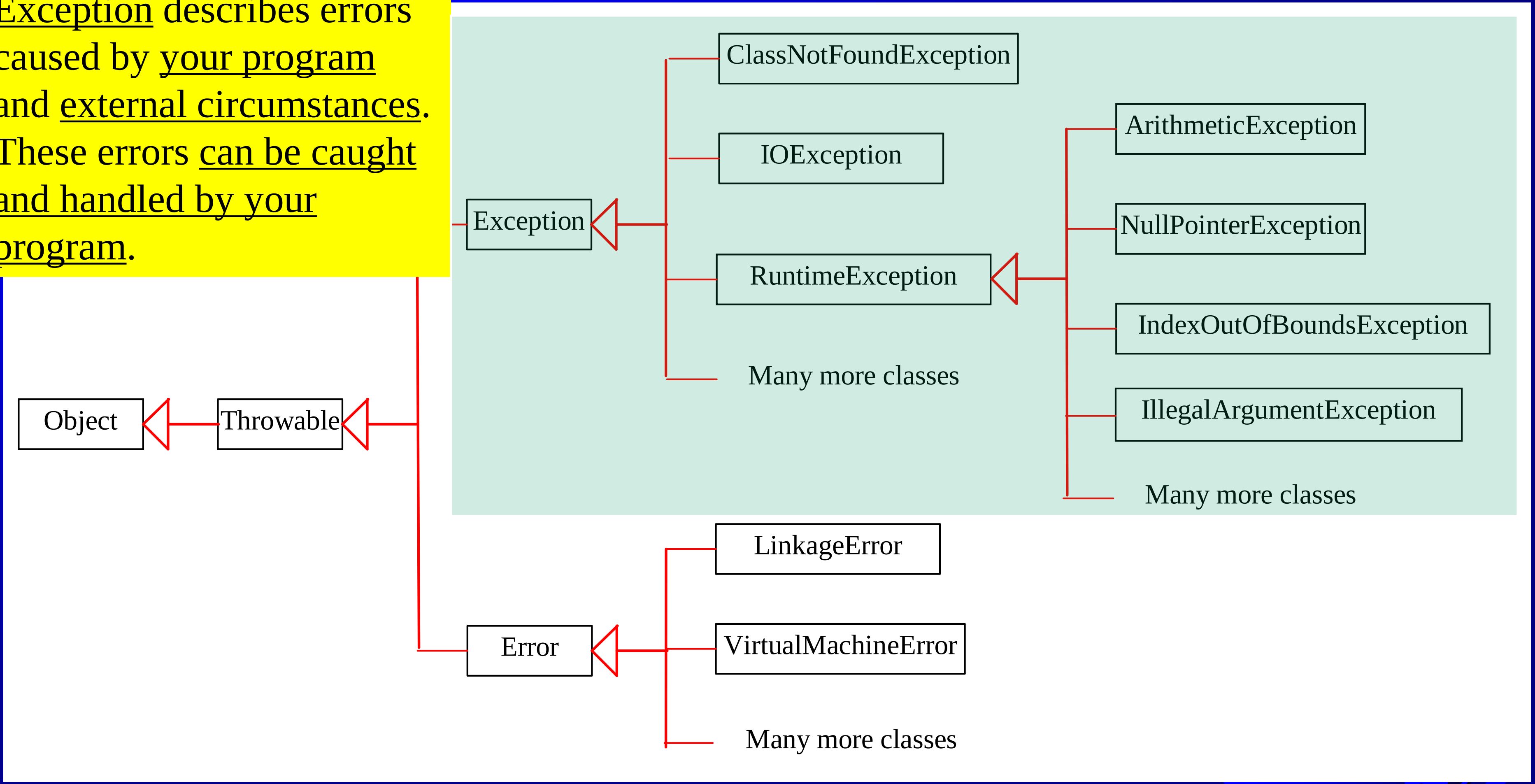


System errors are thrown by JVM . They are internal system errors. Such errors rarely occur.

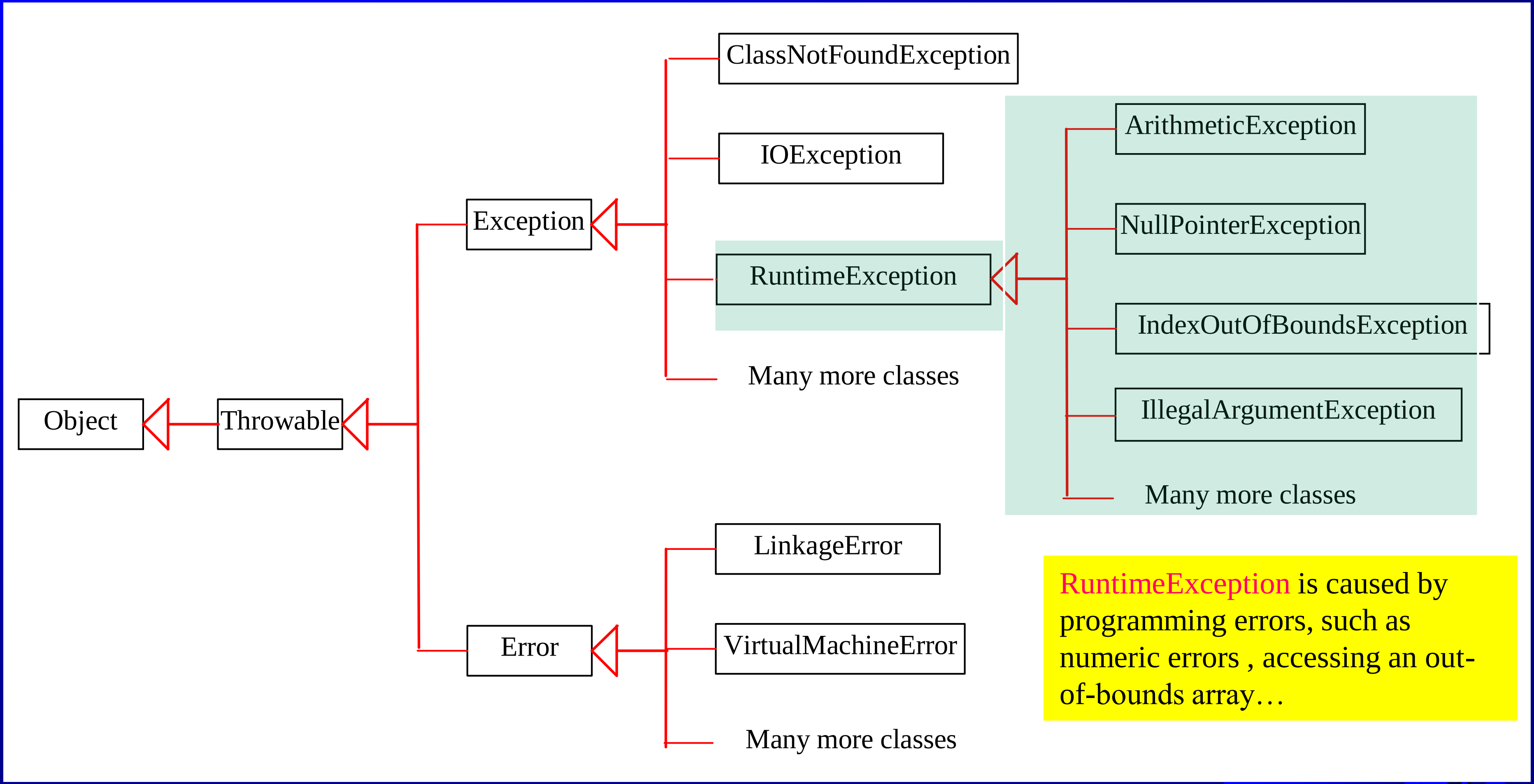
If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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

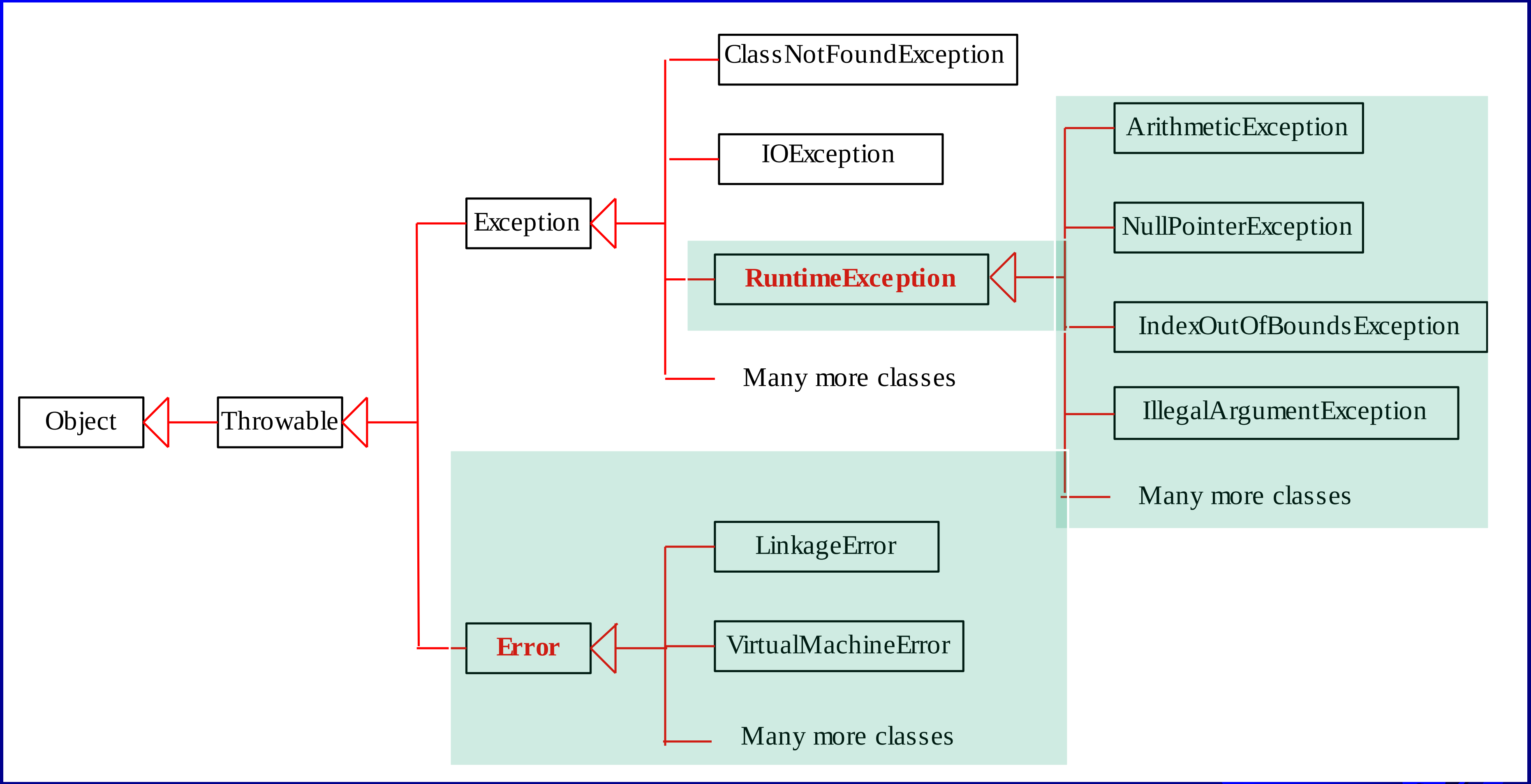


Runtime Exceptions



RuntimeException is caused by programming errors, such as numeric errors , accessing an out-of-bounds array…

Unchecked Exceptions



Unchecked Exceptions

Unchecked exceptions:

RuntimeException, Error and their subclasses

- **No need to declare** such exceptions
- **Un-recoverable**

- In most cases, they are logic errors.
  - e.g. an IndexOutOfBoundsException is thrown if you access an element in an array outside the array bounds.
  - logic errors should be corrected in the program.
- They can occur anywhere in the program. To avoid overuse of try-catch blocks, Java does not mandate you to declare/catch them.

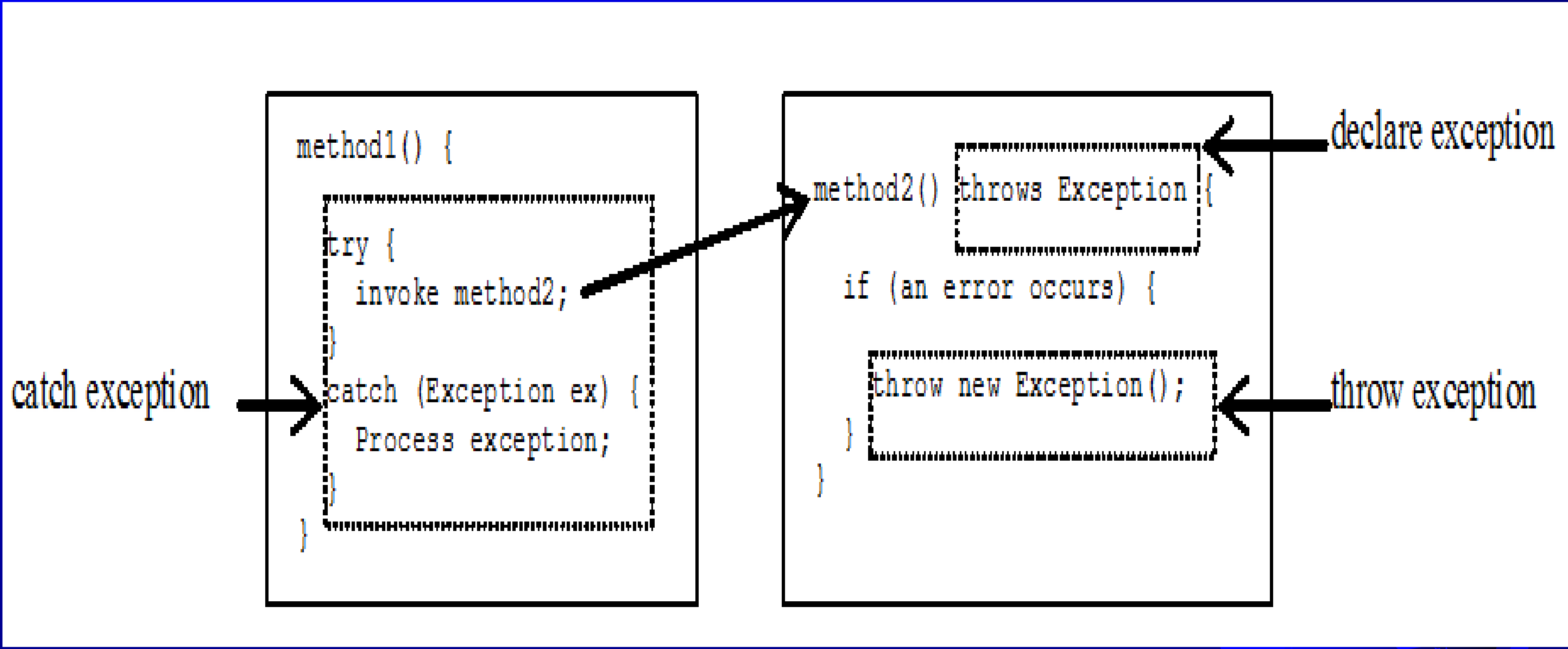
Checked Exceptions

Checked exceptions: all other exceptions

Must be explicitly declared in the method header

- callee : declare and throw the exceptions.
- caller : catch and handle the exceptions.

Declare, Throw, and Catch Exceptions



```
method1() {

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

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Declare Exceptions

A method must declare the types/classes of exceptions it might throw.

```
public void myMethod()  throws IOException
{…}
```

```
//throw multiple exceptions
public void myMethod()
  throws IOException, OtherException
{…}
```

Throw Exceptions

When the program detects an error, the program can create an instance of an appropriate exception class/type and throw it.

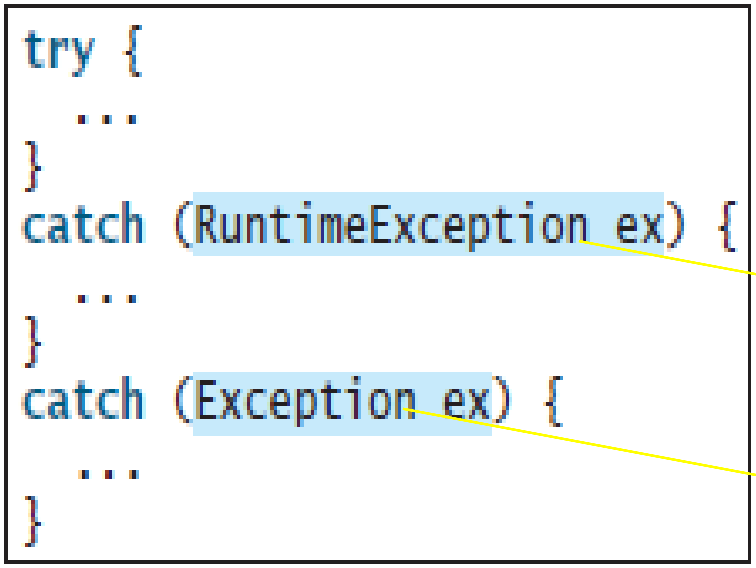
```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

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");
}
```

Catch Exceptions

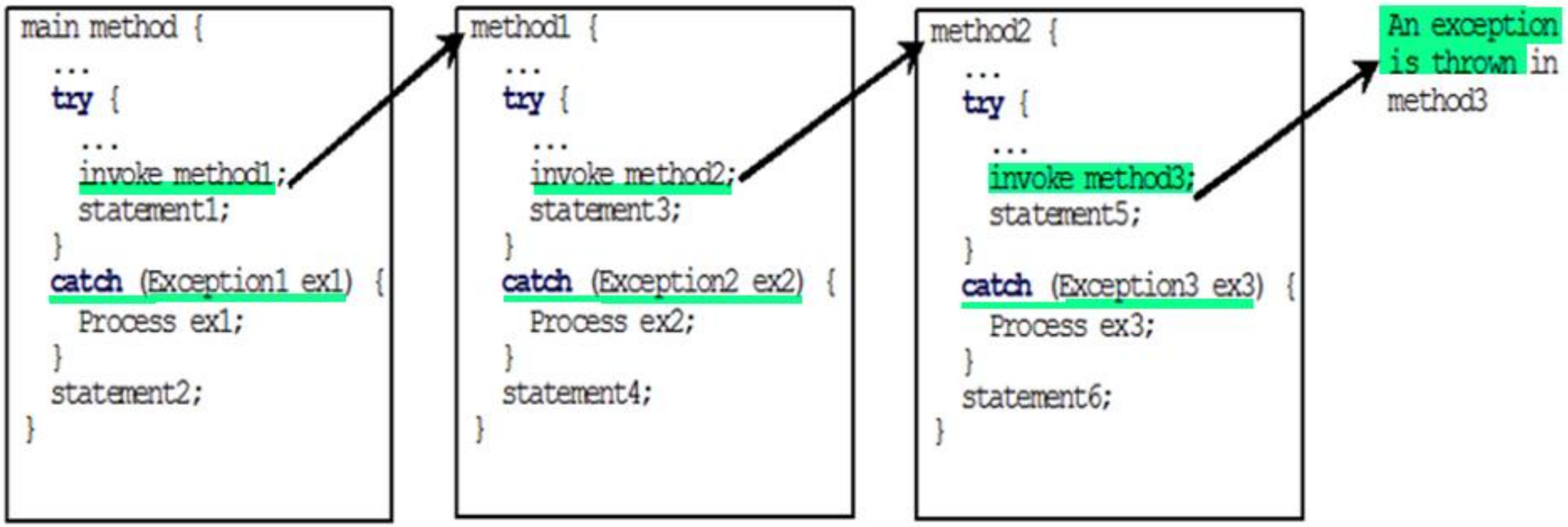


```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

- A catch block of a subclass should appear before that of a superclass
- Incorrect order will cause compile error

Catch Exceptions



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

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

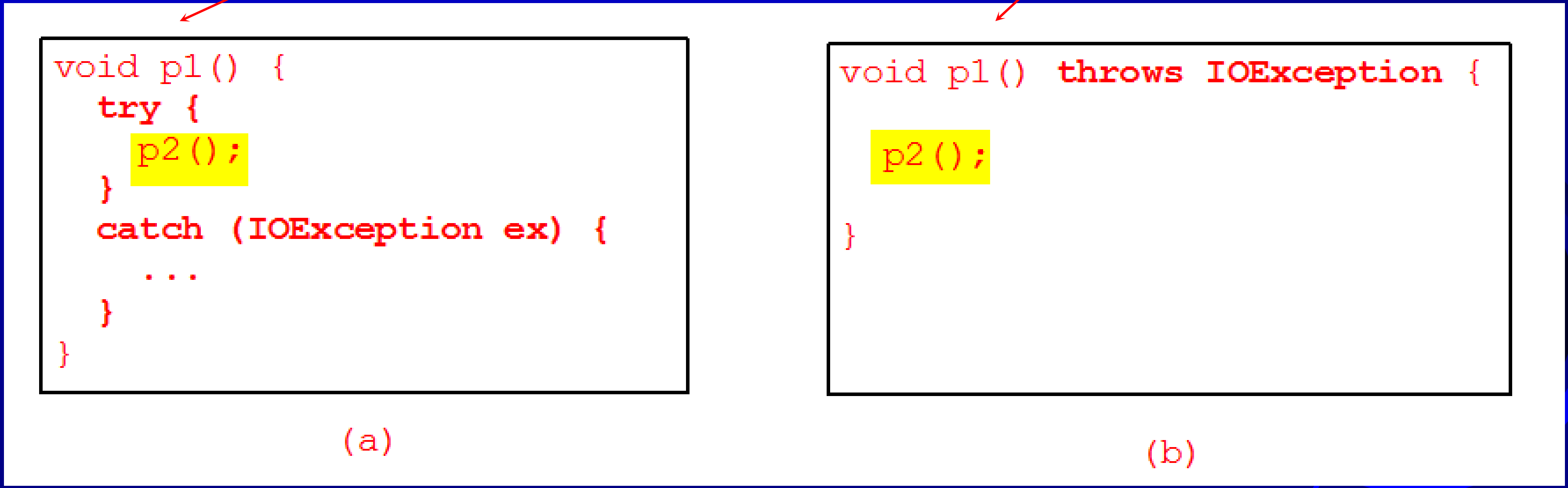
```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

- 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`, and `main`, the program terminates. `statement1` and `statement2` are not executed.

Checked Exceptions

If a method (e.g., **p2()**), declares a checked exception (e.g., IOException),
When calling **p2(),** you must **call it in a try-catch block or declare to throw the exception in the calling method.**

e.g. Suppose that method **p2()** may throw an IOException,



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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Example: Declare, Throw, and Catch Exceptions

- setRadius method throws an exception if radius is negative.

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

```java
public CircleWithException(double newRadius) {
  setRadius(newRadius);
```

```java
    try {
      CircleWithException c1 = new CircleWithException(5);
      CircleWithException c2 = new CircleWithException(-5);
      CircleWithException c3 = new CircleWithException(0);
    }
    catch (IllegalArgumentException ex) {
      System.out.println(ex);
    }

    System.out.println("Number of objects created: " +
      CircleWithException.getNumberOfObjects());
  }
}
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

ex.toString()

toString() :
"Exception class : getMessage()"

Rethrowing Exceptions

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

The `finally` Clause

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

- **finally** block is executed under all circumstances, regardless of whether an exception occurs in the **try** block or is caught

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

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

animation

Trace a Program Execution

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

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed.

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.

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.

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

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

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

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- However, exception handling usually requires more time and resources
  - because it requires instantiating a new exception object, propagating the errors to the calling methods, and rolling back the call stack.

When to Throw Exceptions

- An exception occurs in a method.
  - If a callee want the exception to be processed by its caller, the callee throw an exception to its caller
  - If a callee can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

Use try-catch to deal with unexpected error conditions.

Do not use it to deal with simple, **expected** situations.

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

No need:

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

The File Class

- The File class is a wrapper class for
  directory path + file name

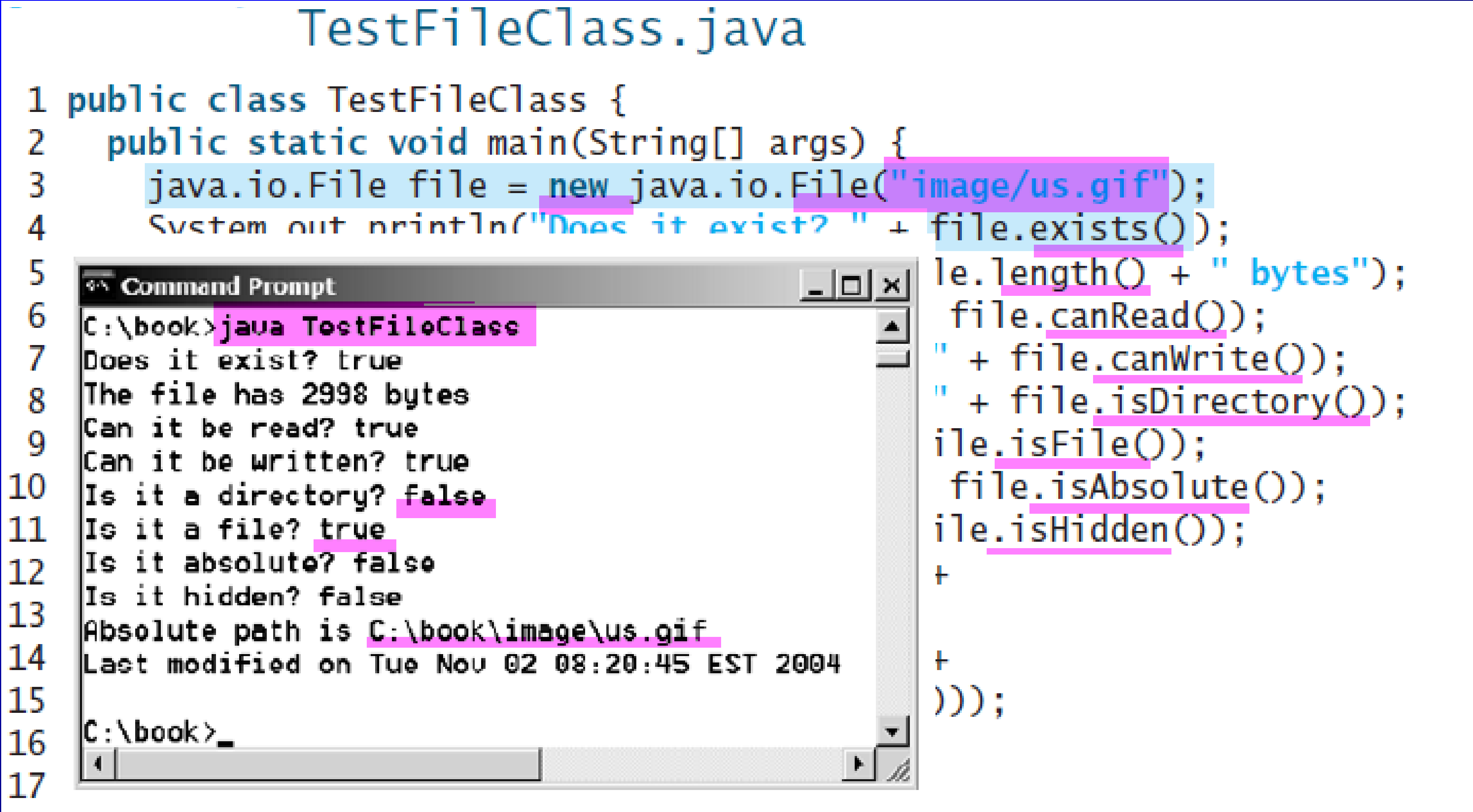
  – *Absolute file name* :
    - for All platforms (Windows, Unix, …), **platform-independent**
      – **“/book/Welcome.java”**
      – Java dirctory seperator: **forward slash (/)**
    - For Windows platform:
      – **“c:\book\Welcome.java” or “c:\\book\\Welcome.java”**

  – *Relative file name* : file name relative to *current directory*
    - **“Welcome.java”**

- Obtaining file properties and manipulating file

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 platforms), and converts drive letters to standard uppercase (on Win32 platforms).
`+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 this file. The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames this file. The method returns true if the operation succeeds.

Create files in a platform-independent way and use the methods in the File class to obtain their properties.



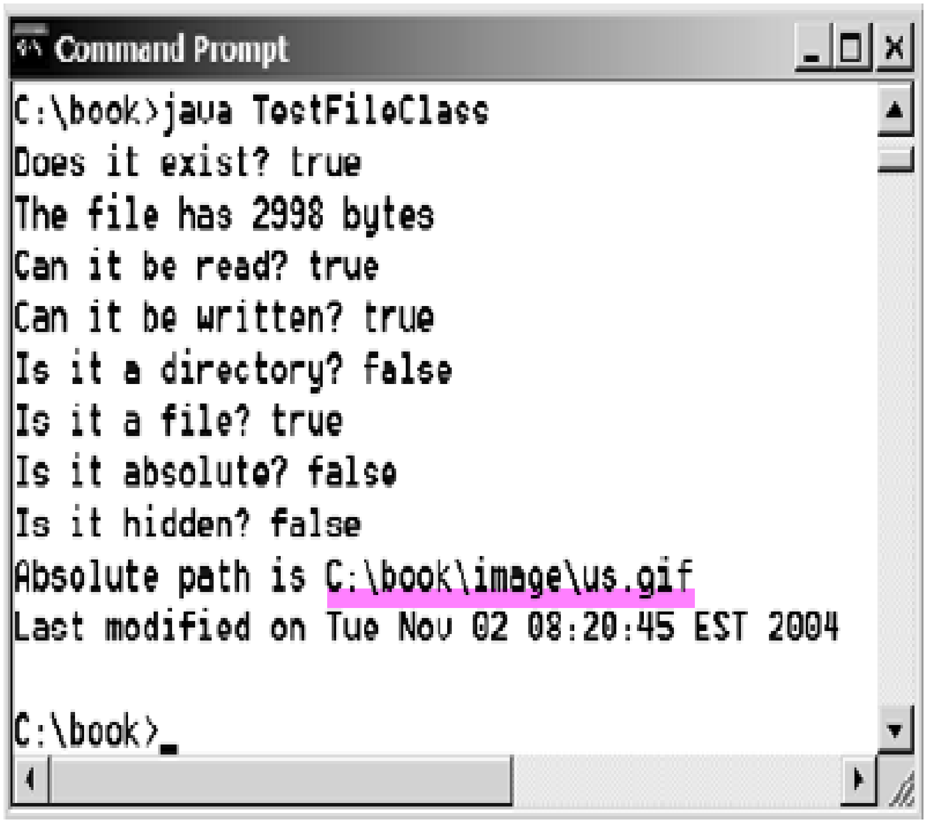
TestFileClass.java

```
 1 public class TestFileClass {
 2   public static void main(String[] args) {
 3     java.io.File file = new java.io.File("image/us.gif");
 4     System.out.println("Does it exist? " + file.exists());
 5                                          le.length() + " bytes");
 6                                           file.canRead());
 7                                          " + file.canWrite());
 8                                          " + file.isDirectory());
 9                                          ile.isFile());
10                                           file.isAbsolute());
11                                          ile.isHidden());
12                                          +
13
14                                          +
15                                          )));
16
17
```

```
C:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\book\image\us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004

C:\book>_
```

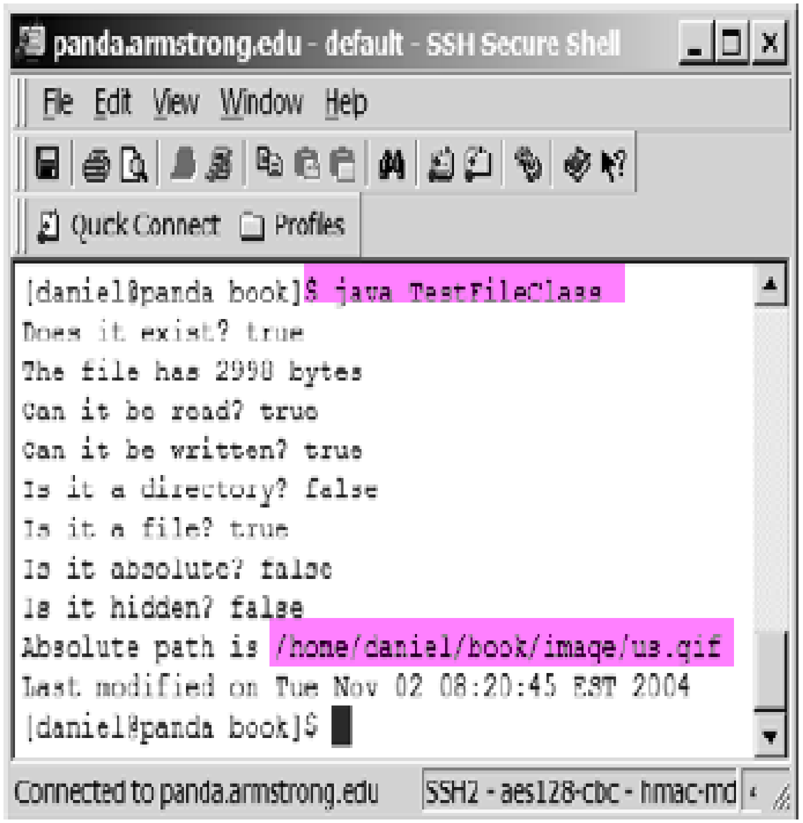
(a) On Windows

The `lastModified()` method returns the date and time when the file was last modified, measured in milliseconds since the beginning of Unix time (00:00:00 GMT, January 1, 1970). The `Date` class is used to display it in a readable format in lines 14–15.



```
C:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\book\image\us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004

C:\book>_
```

(a) On Windows


panda.armstrong.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to panda.armstrong.edu
SSH2 - aes128-cbc - hmac-md

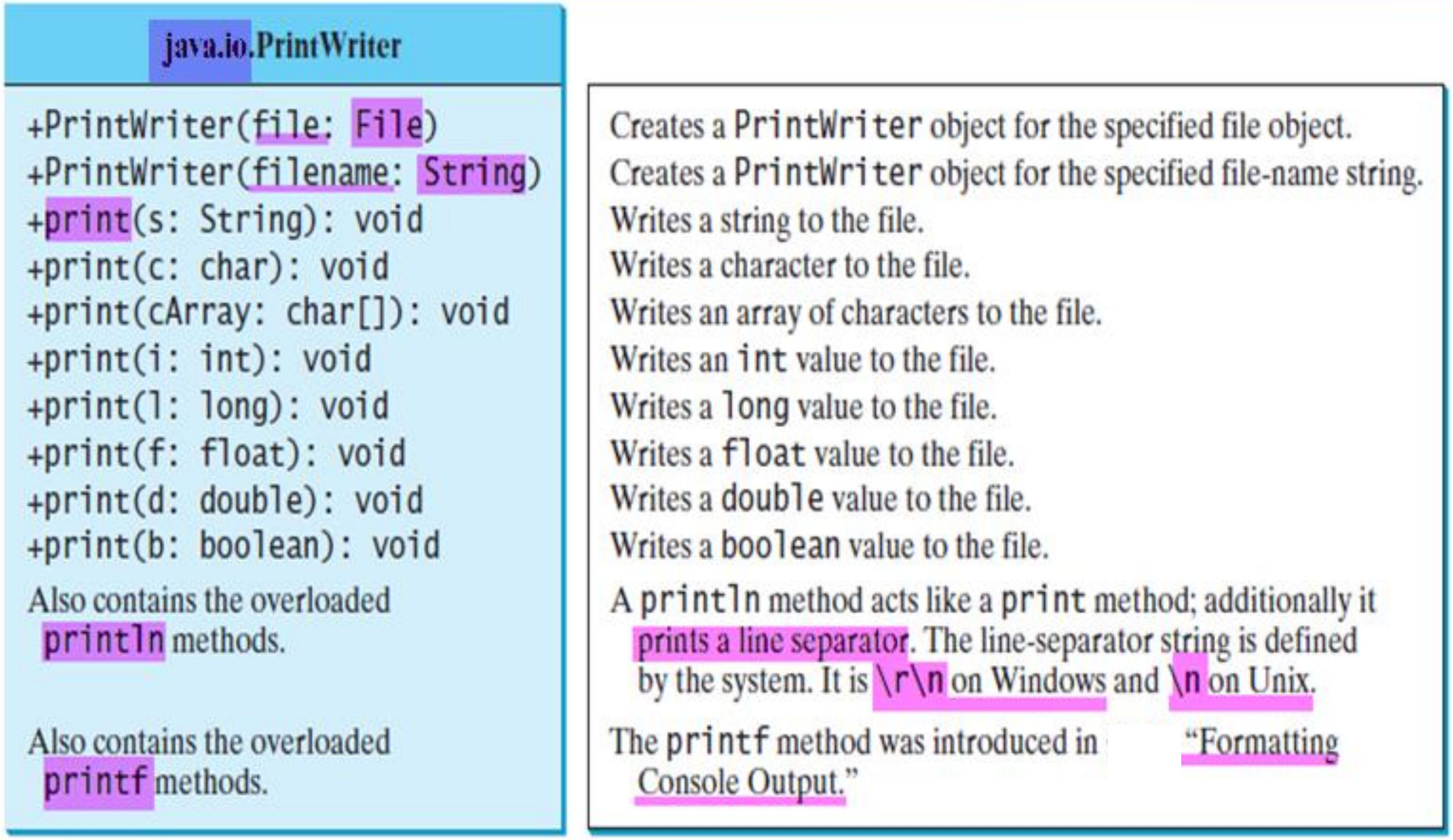
```
[daniel@panda book]$ java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is /home/daniel/book/image/us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004
[daniel@panda book]$
```

(b) On Unix

Text I/O
(File Input & Output)

- Text I/O: reading/writing data from/to a file.
- In order to perform I/O, you need I/O classes.
  –e.g. Scanner and PrintWriter classes.

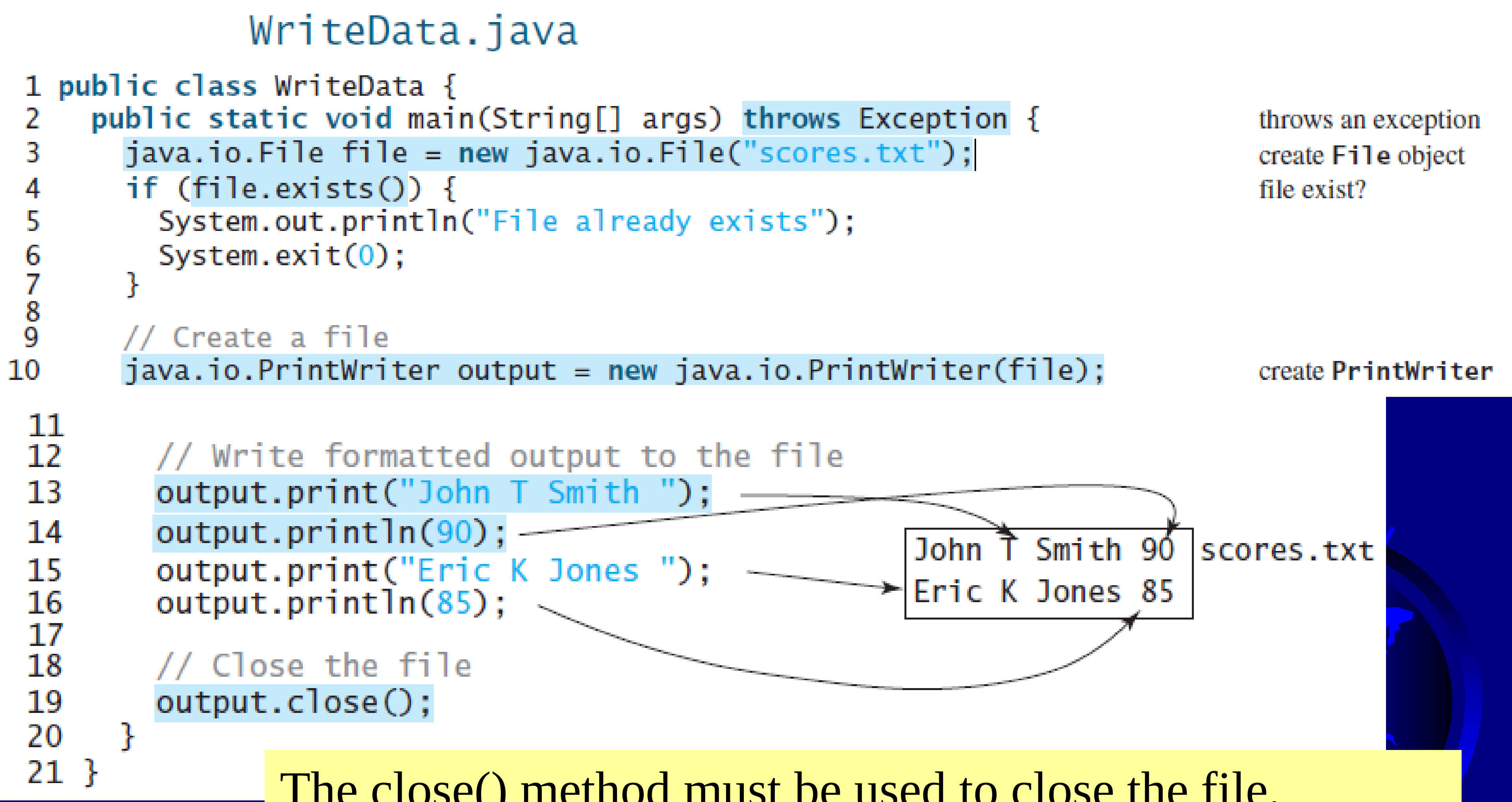
Writing Data Using PrintWriter



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 "Formatting Console Output."

The PrintWriter class contains the methods for writing data to a text file.

Example: creates an instance of **PrintWriter** and writes two lines to the file “scores.txt”.



WriteData.java

```java
public class WriteData {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
    }

    // Create a file
    java.io.PrintWriter output = new java.io.PrintWriter(file);

    // Write formatted output to the file
    output.print("John T Smith ");
    output.println(90);
    output.print("Eric K Jones ");
    output.println(85);

    // Close the file
    output.close();
  }
}
```

throws an exception
create File object
file exist?
create PrintWriter

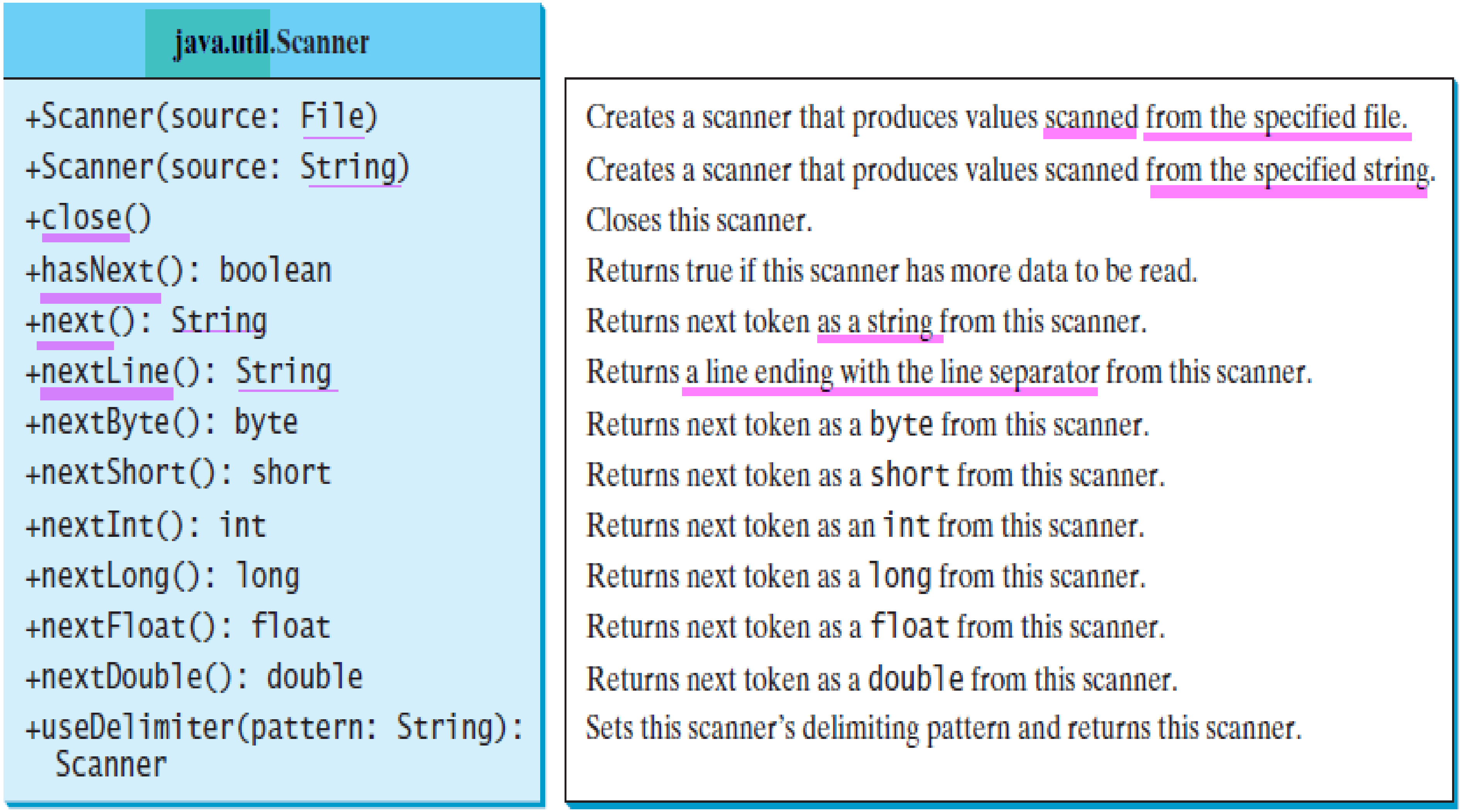
```
John T Smith 90
Eric K Jones 85
```

scores.txt

The close() method must be used to close the file.

Otherwise, the data may not be saved properly in the file.

Reading Data Using Scanner



java.util.Scanner	
+Scanner(source: File)	Creates a scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a scanner that produces values scanned 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.

FIGURE 9.18 The Scanner class contains the methods for scanning data.

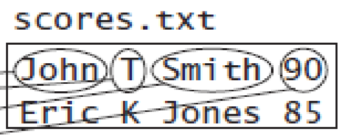
an example that creates an instance of Scanner and reads data from the file “scores.txt”.

ReadData.java

```
 1 import java.util.Scanner;
 2
 3 public class ReadData {
 4   public static void main(String[] args) throws Exception {
 5     // Create a File instance
 6     java.io.File file = new java.io.File("scores.txt");
 7
 8     // Create a Scanner for the file
 9     Scanner input = new Scanner(file);
10
11     // Read data from a file
12     while (input.hasNext()) {
13       String firstName = input.next();
14       String mi = input.next();
15       String lastName = input.next();
16       int score = input.nextInt();
17       System.out.println(
18         firstName + " " + mi + " " + lastName + " " + score);
19     }
20
21     // Close the file
22     input.close();
23   }
24 }
```

create a File

create a Scanner



has next?
read items

close file

Note that 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) (line 6), and use new Scanner(File) to create a Scanner for the file (line 9).

File class

Invoking the constructor new Scanner(File) may throw an I/O exception. So the main method declares throws Exception in line 4.

throws Exception

Each iteration in the while loop reads first name, mi, last name, and score from the text file (lines 12–19). The file is closed in line 22.

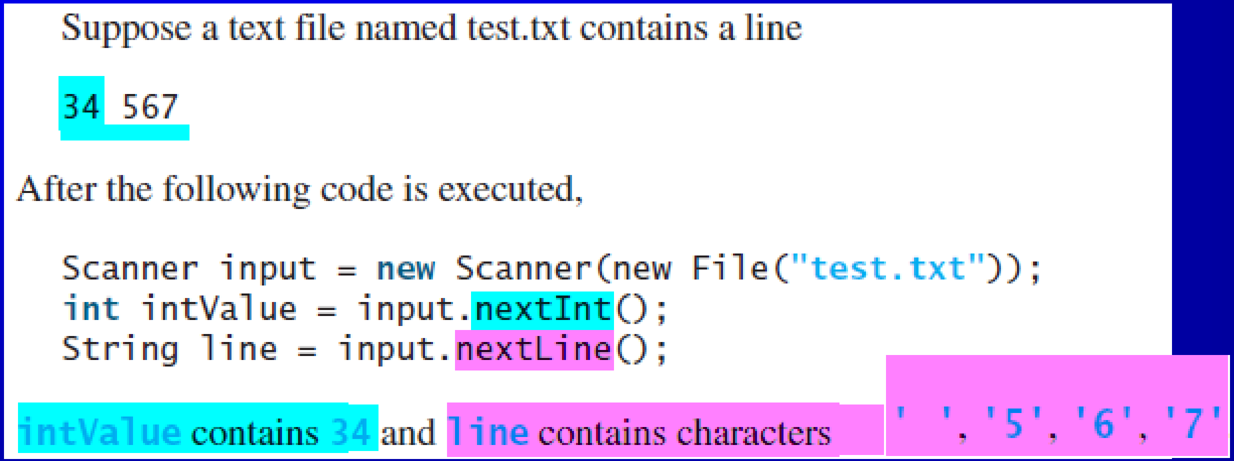
It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file.

close file

Both read a string:

- **next()** reads a string delimited by whitespaces
- **nextLine()** reads a line ending with a line separator

input from file :



Suppose a text file named test.txt contains a line

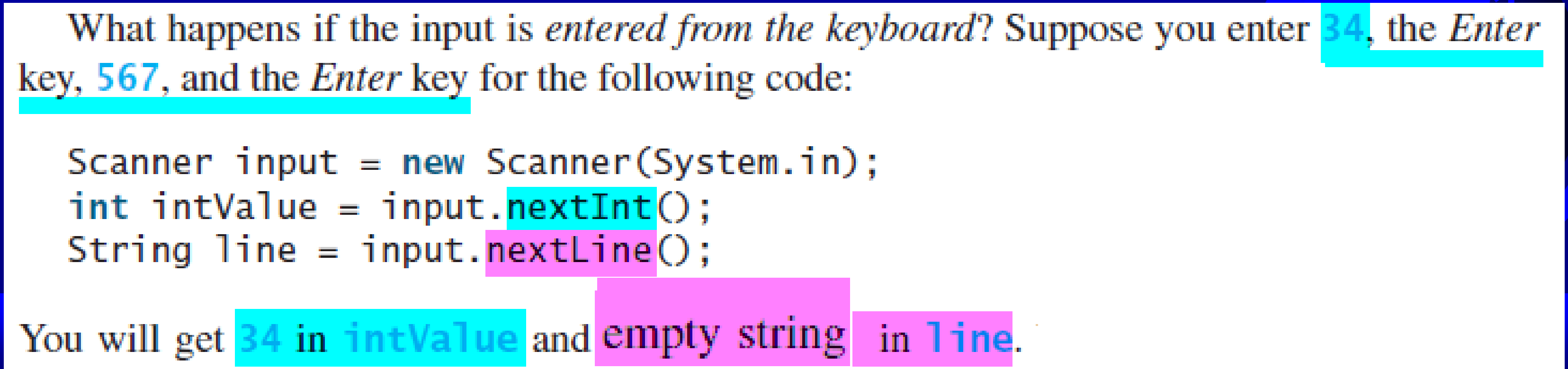
```
34 567
```

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 characters `' '`, `'5'`, `'6'`, `'7'`

input from keyboard :



What happens if the input is *entered from the keyboard*? Suppose you enter `34`, the *Enter* key, `567`, and the *Enter* key for the following code:

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

You will get `34` in `intValue` and empty string in `line`.

Problem: 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 :

java ReplaceText *sourceFile targetFile oldString newString*

For example,

java ReplaceText *oldfile.txt newfile.txt black white*

– replaces all the occurrences of "black" by "white" in *oldfile.txt* and saves the new file in *newfile.txt*.

ReplaceText.java

```java
import java.io.*;
import java.util.*;

public class ReplaceText {
  public static void main(String[] args) throws Exception {
    // Check command-line parameter usage
    if (args.length != 4) {
      System.out.println(
        "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
      System.exit(0);
    }

    // Check if source file exists
    File sourceFile = new File(args[0]);
    if (!sourceFile.exists()) {
       System.out.println("Source file " + args[0] + " does not exist");
       System.exit(0);
    }

    // Check if target file exists
    File targetFile = new File(args[1]);
    if (targetFile.exists()) {
      System.out.println("Target file " + args[1] + " already exists");
      System.exit(0);
    }

    // Create a Scanner for input and a PrintWriter for output
    Scanner input = new Scanner(sourceFile);
    PrintWriter output = new PrintWriter(targetFile);

    while (input.hasNext()) {
      String s1 = input.nextLine();
      String s2 = s1.replaceAll(args[2], args[3]);
      output.println(s2);
    }

    input.close();
    output.close();
  }
}
```

check command usage

source file exists?

target file exists?

create a `Scanner`

create a `PrintWriter`

has next?

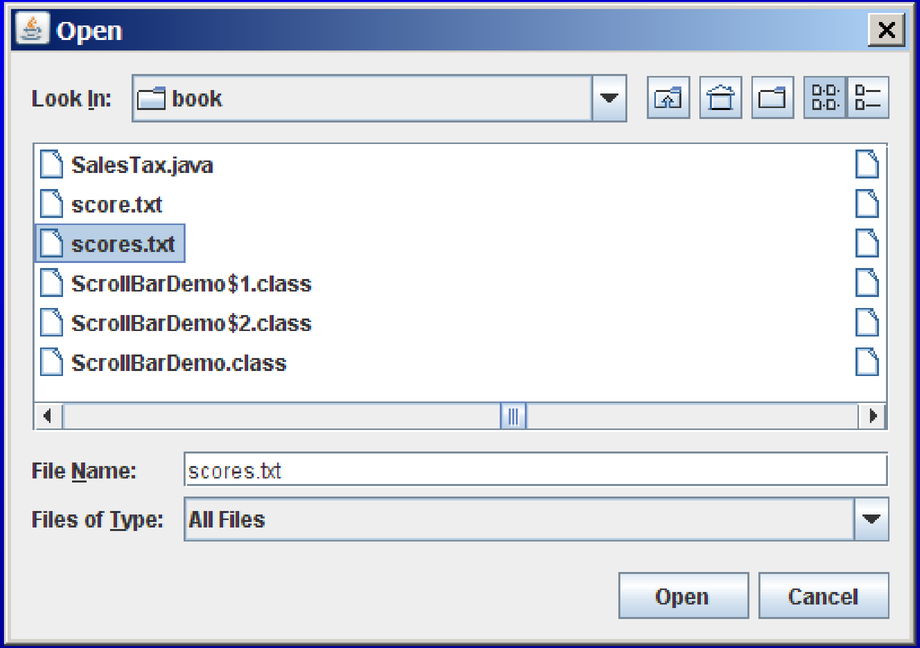
read a line

close file

(GUI) File Dialogs

javax.swing.JFileChooser class : display a file dialog.

From this dialog box, the user can choose a file.



SalesTax.java
score.txt
scores.txt
ScrollBarDemo$1.class
ScrollBarDemo$2.class
ScrollBarDemo.class

File Name: scores.txt

Files of Type: All Files

- Write a program that prompts the user to choose a file and displays its contents on the console.

ReadFileUsingJFileChooser.java

```java
import java.util.Scanner;
import javax.swing.JFileChooser;

public class ReadFileUsingJFileChooser {
  public static void main(String[] args) throws Exception {
    JFileChooser fileChooser = new JFileChooser();
    if (fileChooser.showOpenDialog(null)
        == JFileChooser.APPROVE_OPTION) {
      // Get the selected file
      java.io.File file = fileChooser.getSelectedFile();

      // Create a Scanner for the file
      Scanner input = new Scanner(file);

      // Read text from the file
      while (input.hasNext()) {
        System.out.println(input.nextLine());
      }

      // Close the file
      input.close();
    }
    else {
      System.out.println("No file selected");
    }
  }
}
```

The method returns APPROVE_OPTION or CANCEL_OPTION, which indicates whether the Open button or the Cancel button was clicked.

create a JFileChooser

display file chooser

Read data from the Web

Read from a file (URL) on the Web

☞ first create an instance of the URL class :

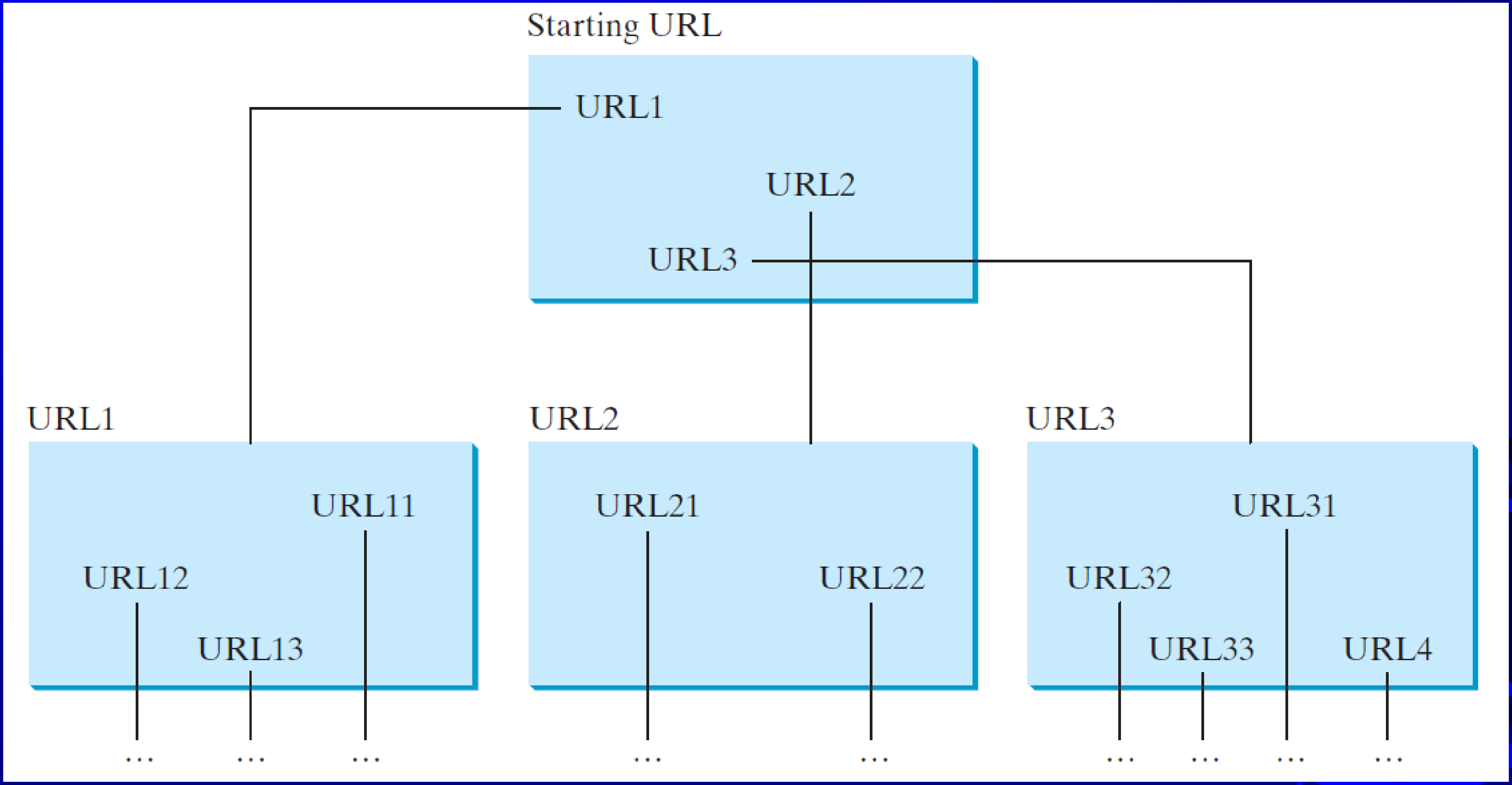
```
import java.net.URL;
URL url = new URL("http://www.google.com/index.html");
```

☞ open an input stream of the instance,
and read from the stream:

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

Case Study: Web Crawler

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



Case Study: Web Crawler

The program **follows the URLs to traverse the Web**.
each URL is traversed only once.

The program maintains two lists of URLs:

- **listOfPendingURLs :** URLs pending for traversing
- **listOfTraversedURLs :** URLs having already been traversed.

Algorithm:

Case Study: Web Crawler

```
Add the starting URL to listOfPendingURLs;
while listOfPendingURLs is not empty {
    Remove a URL from listOfPendingURLs;
    if this URL is not in listOfTraversedURLs {

        Add it to listOfTraversedURLs;
        Display this URL;

        Exit the while loop when the size of S is equal to 100.

        Read the page from this URL;
        for each URL contained in the page {
            if it is not in listOfTraversedURLs
                Add it to listOfPendingURLs
        }
    }
}
```