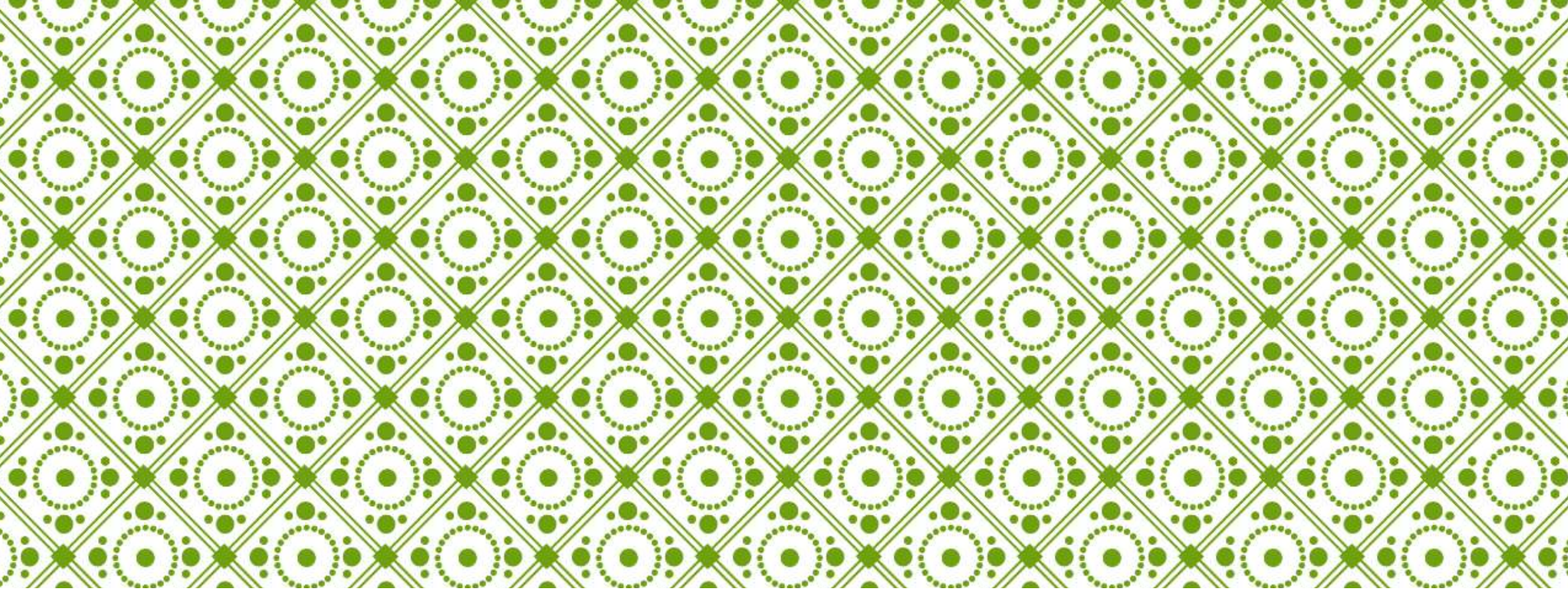


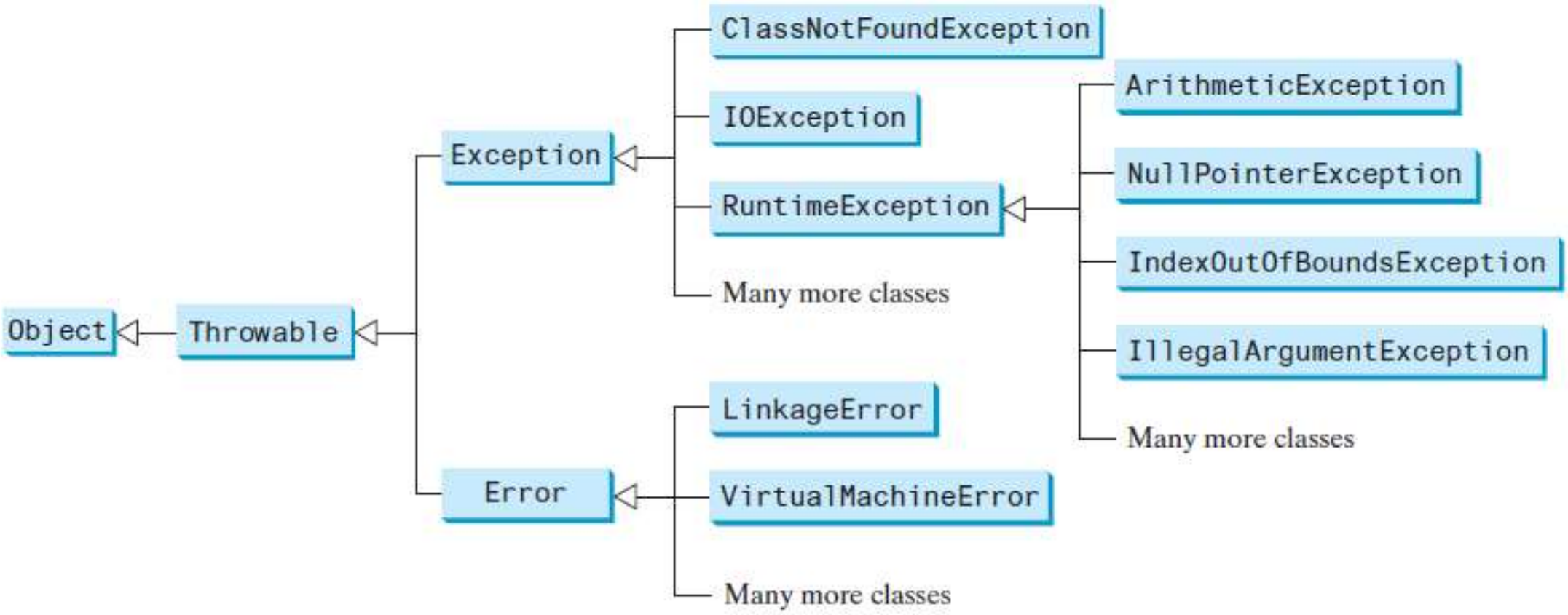
PROGRAMMING 2

Gaza University
Musbah j. Mousa

EXCEPTIONS

- *Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution.*
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- *examples : divide by zero , out of bound of array*

EXCEPTIONS


Object
Throwable
Exception
Error
ClassNotFoundException
IOException
RuntimeException
Many more classes
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Many more classes
LinkageError
VirtualMachineError
Many more classes

EXCEPTIONS

LISTING 12.5 InputMismatchExceptionDemo.java

```
1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = input.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         input.nextLine(); // Discard input
23       }
24     } while (continueInput);
25   }
26 }
```

create a Scanner

try block

If an InputMismatchException occurs

catch block

EXCEPTIONS

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

EXCEPTIONS

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

TABLE 12.1 Examples of Subclasses of `Error`

Class	*Reasons for Exception*
`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.

EXCEPTIONS

- *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.

TABLE 12.2 Examples of Subclasses of `Exception`

Class	*Reasons for Exception*
`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`.

EXCEPTIONS

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

TABLE 12.3 Examples of Subclasses of `RuntimeException`

Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`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.

MORE ON 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.*

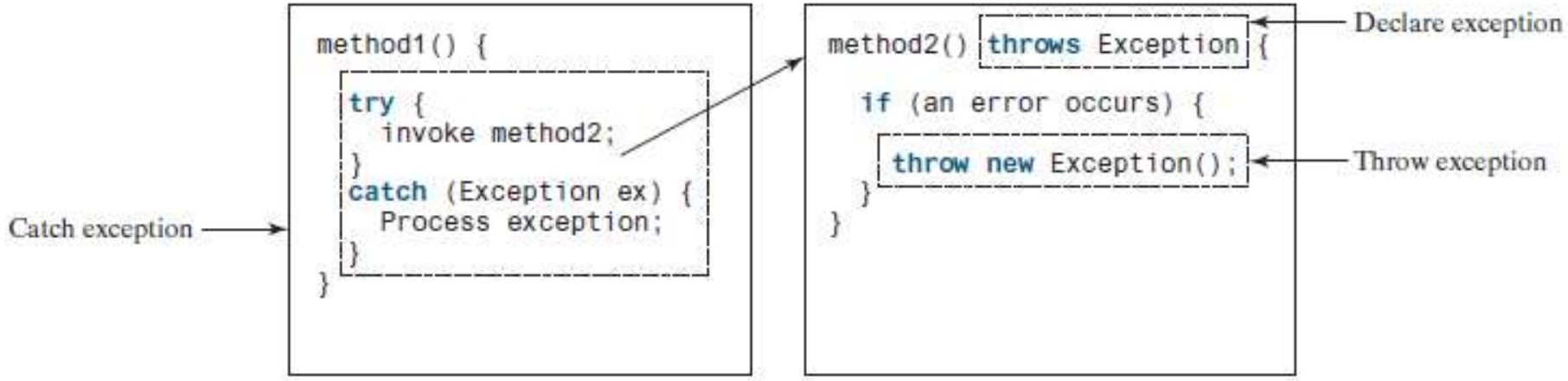


FIGURE 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

MORE ON EXCEPTION HANDLING

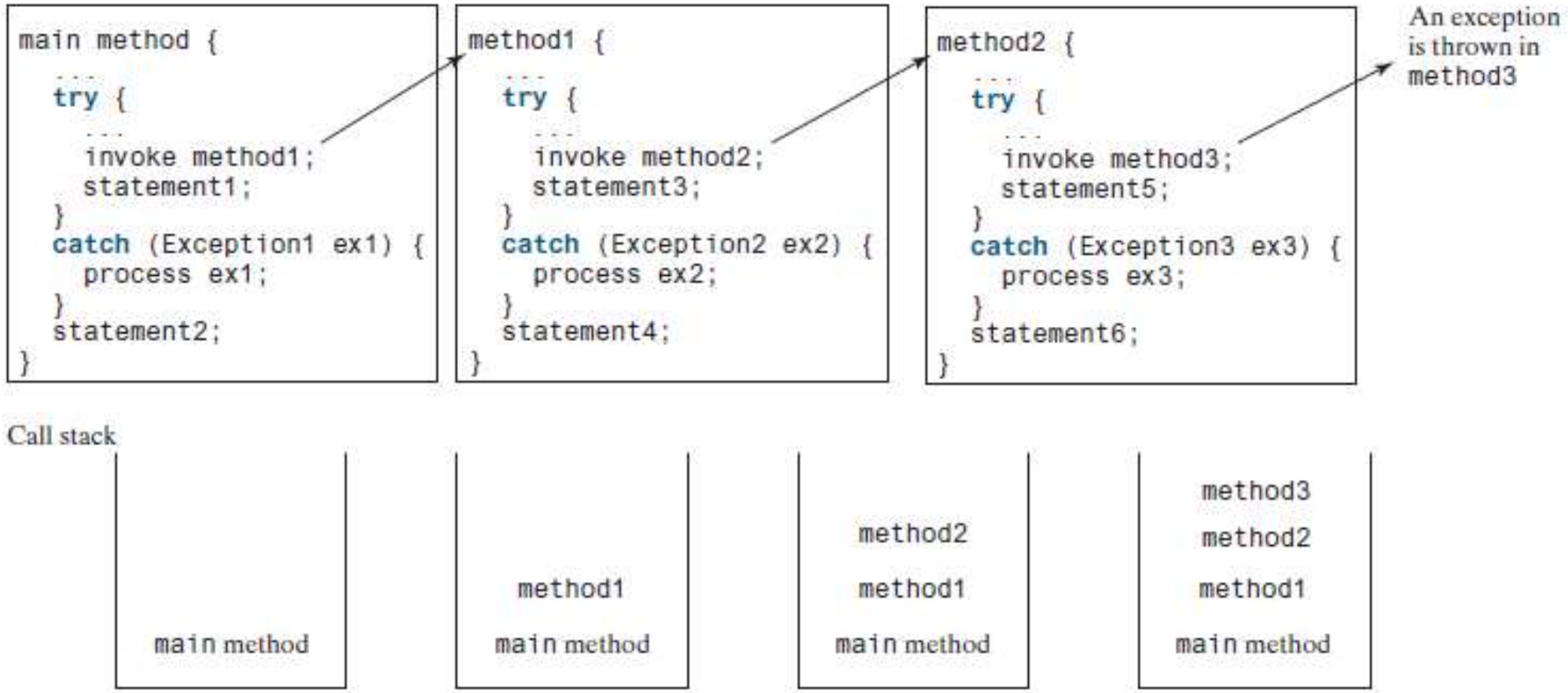


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

MORE ON EXCEPTION HANDLING

12.4.4 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, as shown in Figure 12.4. The `printStackTrace()` method prints stack trace

methods in Throwable

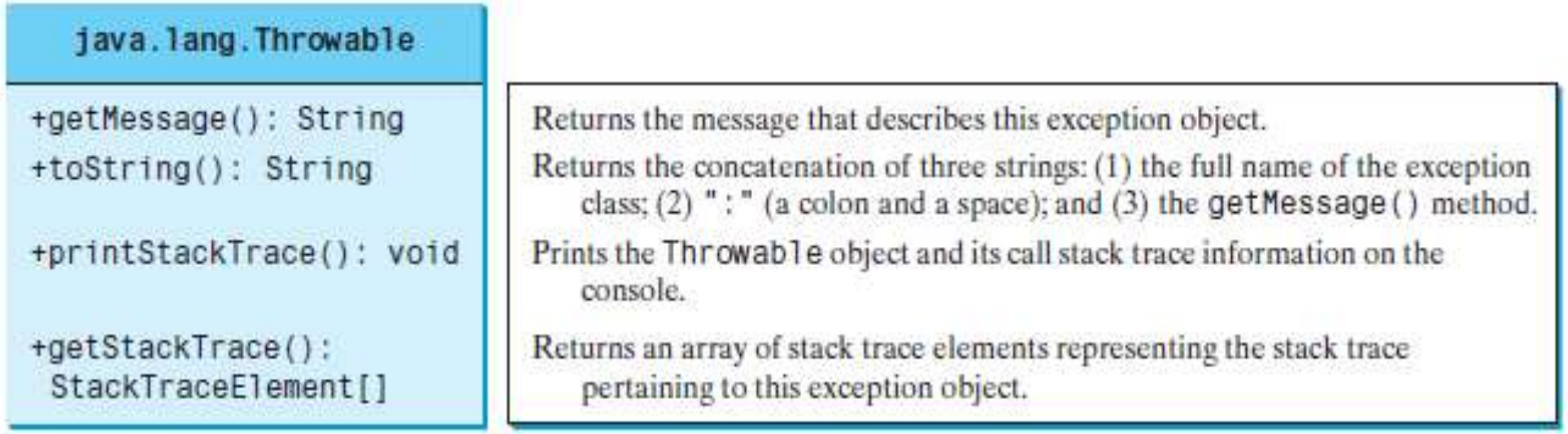


FIGURE 12.4 `Throwable` is the root class for all exception objects.

THE **FINALLY** CLAUSE

- *The* **finally** *clause is always executed regardless of whether an exception occurred or not.*

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.*
- The **File** class contains the methods for obtaining file and directory properties, and for renaming and deleting files and directories.
- *The* ***File*** *class does not contain the methods for reading and writing file contents.*

THE **FILE** CLASS

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 *htdden* 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.

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.*

FILE INPUT AND OUTPUT

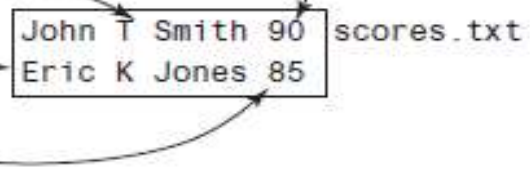
throws an exception
create File object
file exist?

```
1  public class WriteData {
2    public static void main(String[] args) throws java.io.IOException {
3      java.io.File file = new java.io.File("scores.txt");
4      if (file.exists()) {
5        System.out.println("File already exists");
6        System.exit(1);
7      }
8
9      // Create a file
10     java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12     // Write formatted output to the file
13     output.print("John T Smith ");
14     output.println(90);
15     output.print("Eric K Jones ");
16     output.println(85);
17
18     // Close the file
19     output.close();
20   }
21 }
```

create PrintWriter

print data

close file



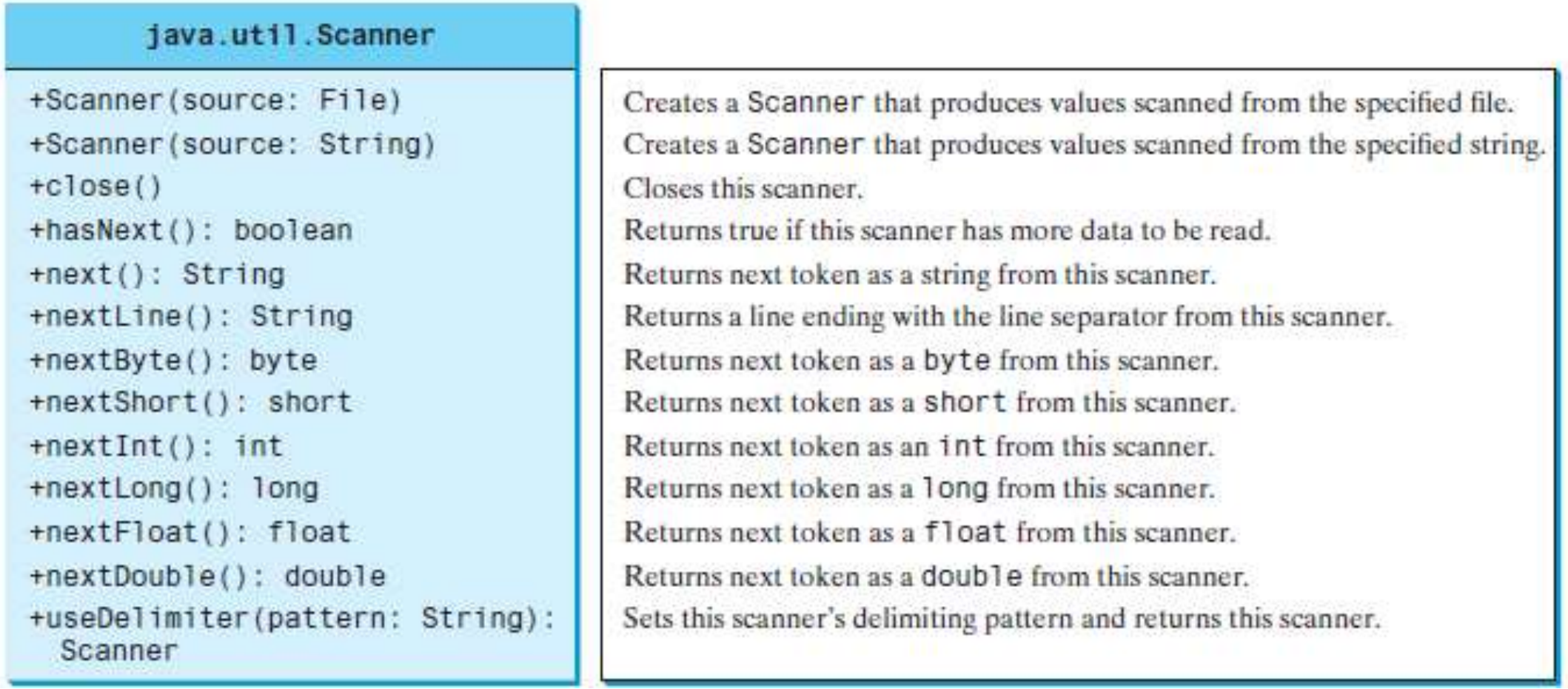
READING DATA USING **SCANNER**

Scanner input = **new** Scanner(System.in);

To read from a file, create a **Scanner** for a file, as follows:

Scanner input = **new** Scanner(**new** File(filename));

READING DATA USING **SCANNER**

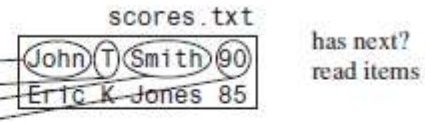


READING DATA USING **SCANNER**

create a File

create a Scanner

```
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 }
```



has next?
read items

close file

QUESTIONS?