**CHAPTER 12: EXCEPTION HANDLING AND TEXT I/O**

* 1. **INTRODUCTION**

Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution.

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

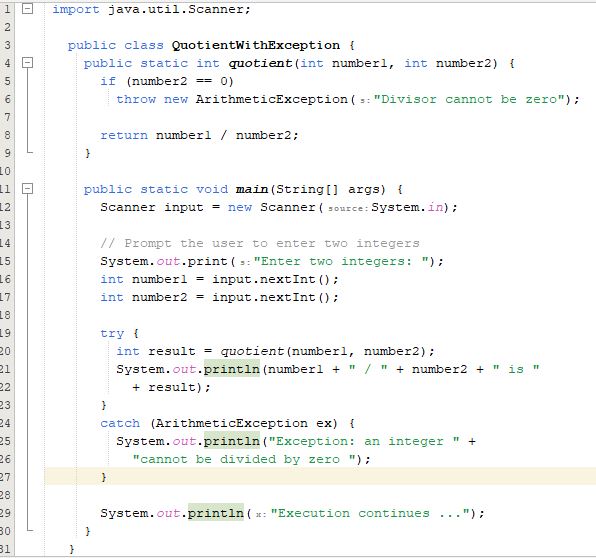
In Java, runtime errors are thrown as exceptions. An exception is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally.

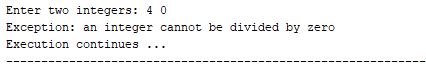
* 1. **EXCEPTION-HANDLING OVERVIEW**

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

Java enables a method to throw an exception that can be caught and handled by the caller.

Let’s check out this the example below:

****

****

If **number2** is **0**, the method throws an exception (line 6) by executing

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

The value thrown, in this case new **ArithmeticException("Divisor cannot be zero")**, is called an **exception**.

The execution of a **throw** statement is called throwing an exception.

The exception is an object created from an exception class. In this case, the exception class is **java.lang.ArithmeticException**.

The constructor **Arithmetic Exception(str)** is invoked to construct an exception object, where **str** is a message that describes the exception.

* 1. **EXCEPTION TYPES**

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

There are many predefined exception classes in the Java API.

IllegalArgumentException

IndexOutOfBoundsException

NullPointerException

ArithmeticException

Object

Throwable

Exception

Error

RuntimeException

IOException

ClassNotFoundException

Many more classes

VirtualMachineError

LinkageError

Many more classes

Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.

**Note:** The class names **Error**, **Exception**, and **RuntimeException** are somewhat confusing. All three of these classes are exceptions and all of the errors occur at runtime.

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**.

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

* System errors are thrown by the JVM and are represented in the **Error** class.

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 are represented in the **Exception** class, which describes errors caused by your program and by external circumstances.

Examples of Subclasses of **Exception**

|  |  |
| --- | --- |
| Class | Reasons for exception |
| ClassNotFoundException | Attempt to use a class that does not exist. This exception would occur, if you tried to run a nonexistent class using the java command or if your program were composed of 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. |

* 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.

Examples of Subclasses of **RuntimeException**

|  |  |
| --- | --- |
| Class | Reasons for exception |
| ClassNotFoundException | Attempt to use a class that does not exist. This exception would occur, if you tried to run a nonexistent class using the java command or if your program were composed of 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. |
| IndexOutOfBoundsException | Index to an array is out of range. |
| IllegalArgumentException | A method has passed an argument that is illegal or inappropriate. |

* 1. **DECLARING, THROWING, AND CATCHING EXCEPTION**

A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

Java’s exception-handling model is based on three operations:

* declaring an exception,
* throwing an exception, and
* catching an exception.
  + 1. **Declaring Exceptions**

In Java, the statement currently being executed belongs to a method. 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**.

To declare an exception in a method, use the throws keyword in the method header, as in this example:

**public void myMethod() throws IOException**

The throws keyword indicates **myMethod** might throw an **IOException**.

* + 1. **Throwing Exceptions**

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**.

If the program detects that an argument passed to the method violates the method contract, the program can create an instance of **IllegalArgumentException** and throw it, as follows:

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

**Note:** IllegalArgumentException is an exception class in the Java API.

* + 1. **Catching Exceptions**

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

**try {**

**statements;**

**}**

**catch (Exception1 exVar1) {**

**handler for exception1;**

**}**

**catch (Exception2 exVar2) {**

**handler for exception2;**

**}**

**...**

**catch (ExceptionN exVarN) {**

**handler for exceptionN;**

**}**

If no exceptions arise during the execution of the try block, the catch blocks are skipped.

* + 1. **Getting Information from Exceptions**

An exception object contains valuable information about the exception.

You may use the fol lowing 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); and (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. |

**Throwable** is the root class for all exception objects.

* 1. **THE FINALLY CLAUSE**

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

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;**

**}**

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 three possible cases:

1. If no exception arises in the **try** block, **finalStatements** is executed and the next statement after the **try** statement is executed.
2. 2. 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.
3. 3. 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.
   1. **WHEN TO USE EXCEPTIONS**

A method should throw an exception if the error needs to be handled by its caller.

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.

Do not use a **try-catch** block to deal with simple, expected situations.

Use it when you have to deal with unexpected error conditions.

For example:

Is better replaced by this code:

This code:

**if (refVar != null)**

**System.out.println(refVar.toString());**

**else**

**System.out.println("refVar is null")**

**try {**

**System.out.println(refVar.toString());**

**}**

**catch (NullPointerException ex) {**

**System.out.println("refVar is null");**

**}**

* 1. **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:

**try {**

**statements;**

**}**

**catch (TheException ex) {**

**perform operations before exits;**

**throw ex;**

**}**

The statement **throw ex** rethrows the exception to the caller so other handlers in the caller get a chance to process the exception **ex**.

* 1. **CHAINED EXCEPTIONS**

Throwing an exception along with another exception forms a chained exception.

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.**

Here’s how:

**public class ChainedExceptionDemo {**

**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");**

**}**

**}**

The **main** method invokes **method1**, **method1** invokes **method2** (line 13), and **method2** throws an exception (line 21). This exception is caught in the **catch** block in method1 and is wrapped in a new exception in line 16. The new exception is thrown and caught in the catch block in the **main** method in line 6. The sample output shows the output from the **printStackTrace()** method in line 7. The new exception thrown from **method1** is displayed first, followed by the original exception thrown from **method2**.

* 1. **DEFINING CUSTOM EXCEPTION CLASSES**

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

Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, 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**.

The **Exception** class contains four constructors.

|  |  |
| --- | --- |
| java.lang.Exception | |
| +Exception() | Constructs an exception with no message. |
| +Exception(message: String) | Constructs an exception with the specified message. |
| +Exception(message: String, cause: Exception) | Constructs an exception with the specified message and a cause. This forms a chained exception. |

* 1. **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.

Data stored in the program are temporary; they are lost when the program terminates.

To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device.

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.

A relative file name is in relation to the current working directory. The complete directory path for a relative file name is omitted.

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 **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 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. |

The **File** class can be used to obtain file and directory properties, to delete and rename files and directories, and to create directories.

**Note:** Constructing a **File** instance does not create a file on the machine. You can create a **File** instance for any file name regardless of whether it exists or not. You can invoke the **exists()** method on a File instance to check whether the file exists.

* 1. **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 writing/reading data to/from a file (referred to as data input and output, or I/O for short).

There are two types of files: text and binary. Text files are essentially characters on disk.

* + 1. **Writing Data Using PrintWriter**

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

First, you have to create a **PrintWriter** object for a text file as follows:

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

Frequently used methods in **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: f loat): 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.” |

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

* + 1. **Closing Resources Automatically Using try-with-resources**

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

**try (declare and create resources) {**

**Use the resource to process the file;**

**}**

* + 1. **Reading Data Using Scanner**

A **Scanner** breaks its input into tokens delimited by whitespace characters.

To read from the keyboard, you create a **Scanner** for **System.in**, as follows:

**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));**

|  |  |
| --- | --- |
| 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. |

The **Scanner** class contains the methods for scanning data.

* 1. **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 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

For example, the following statement creates a URL object for <http://www.google.com/index.html>.

**try {**

**URL url = new URL("http://www.google.com/index.html");**

**}**

**catch (MalformedURLException ex) {**

**ex.printStackTrace();**

**}**

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());**

**THE END!**