



COMP231

Advanced Programming

Chapter 12 Exception Handling and Text IO

Compiled By: Dr. Majdi Mafarja
Fall Semester 2017/2018

Exception-Handling Overview

Show runtime error

Quotient

Run

Fix it using an if statement

QuotientWithIf

Run

With a method

QuotientWithMethod

Run



Exception Advantages

QuotientWithException

Run

Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.



Handling InputMismatchException

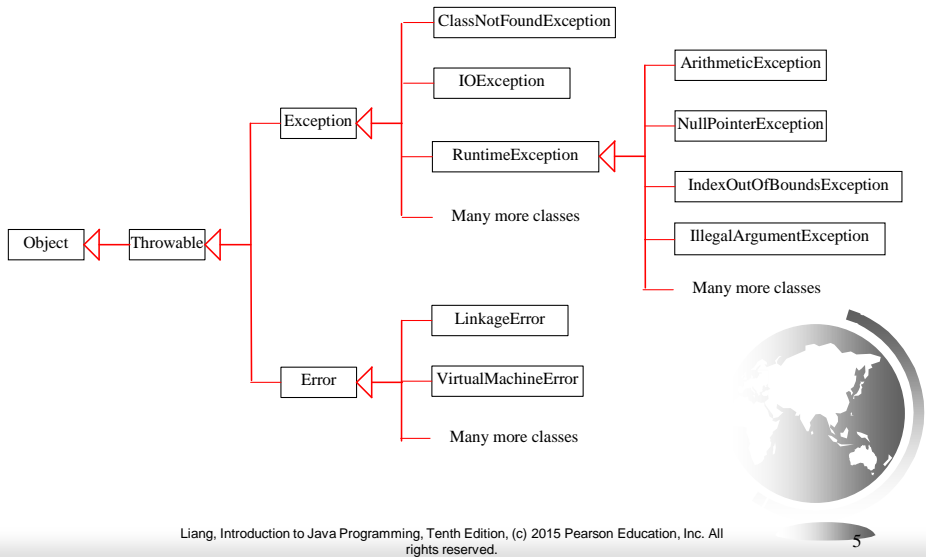
InputMismatchExceptionDemo

Run

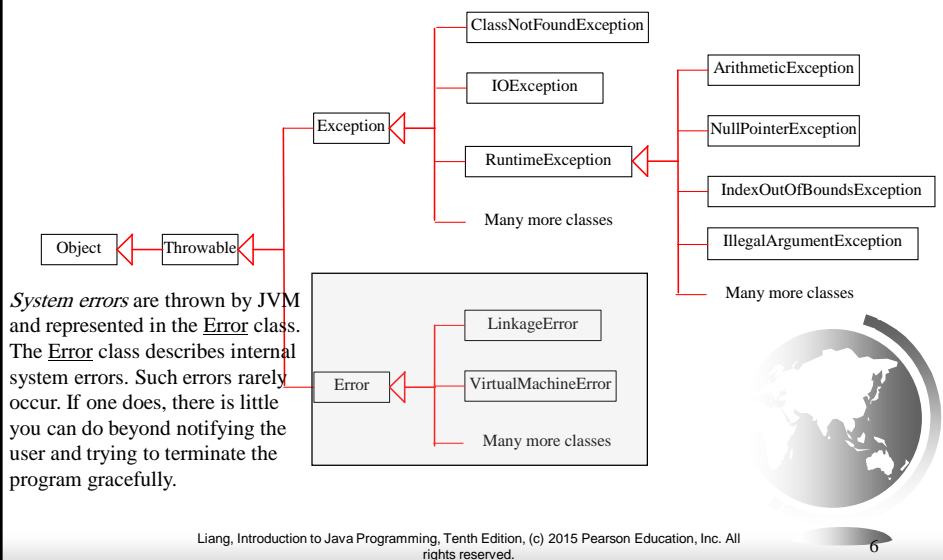
By handling `InputMismatchException`, your program will continuously read an input until it is correct.



Exception Types

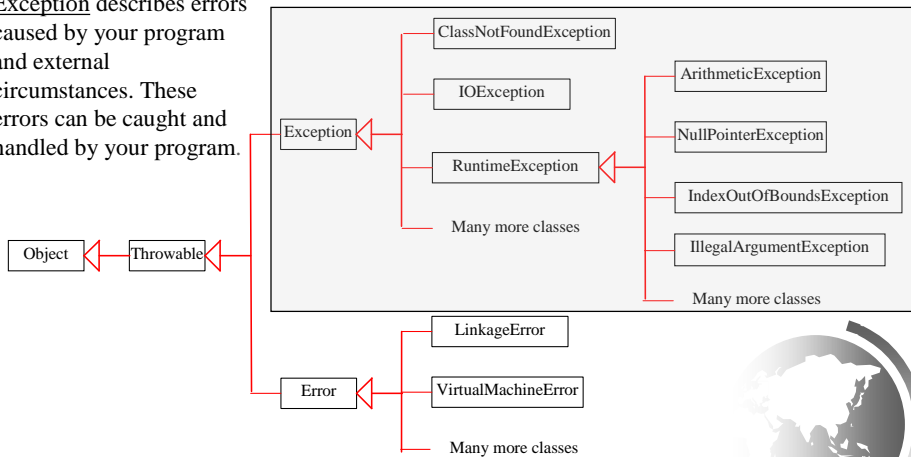


System Errors

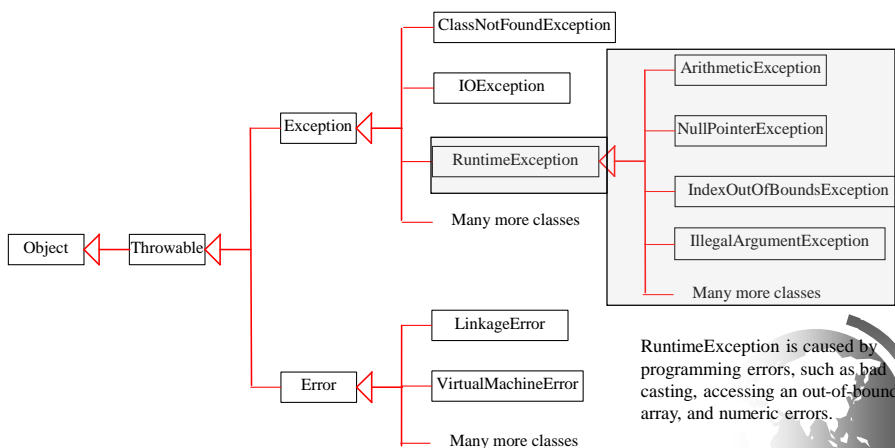


Exceptions

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



Runtime Exceptions



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

Checked Exceptions vs. Unchecked Exceptions

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

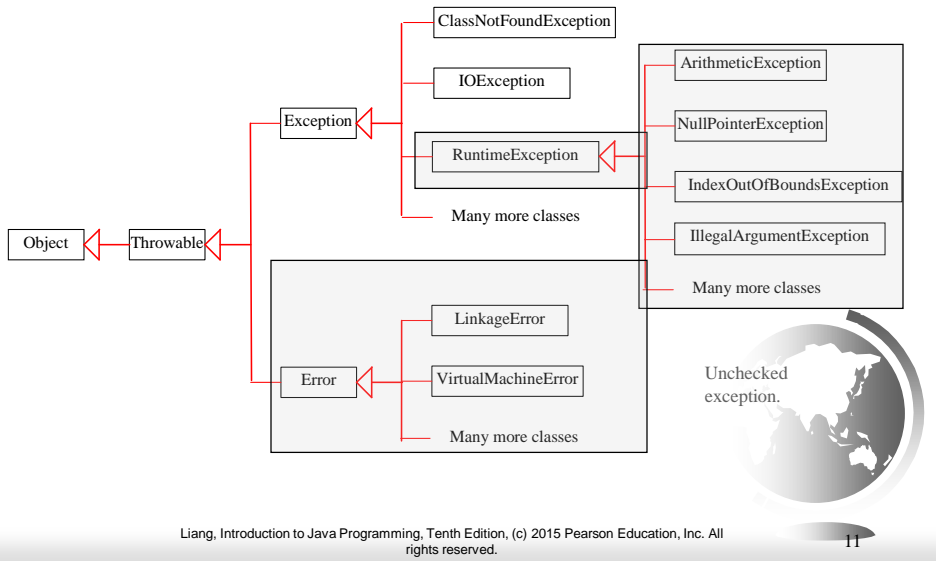


Unchecked Exceptions

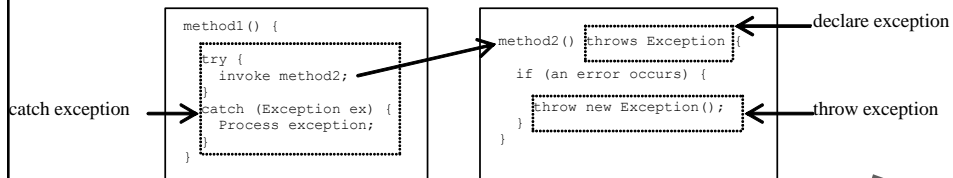
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.



Unchecked Exceptions



Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()  
    throws IOException
```

```
public void myMethod()  
    throws IOException, OtherException
```



Throwing Exceptions

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

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

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



Throwing Exceptions Example

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

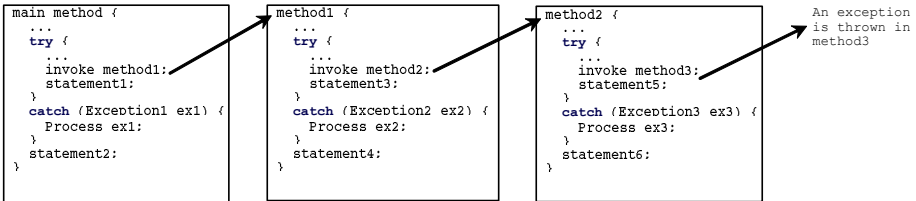


Catching Exceptions

```
try {  
    statements; // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    handler for exception1;  
}  
catch (Exception2 exVar2) {  
    handler for exception2;  
}  
...  
catch (ExceptionN exVar3) {  
    handler for exceptionN;  
}
```



Catching Exceptions



Call Stack



Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
    throw new IOException("File does not exist");
  }
  ...
}
```

Catch or Declare Checked Exceptions

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

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

(a)

```
void p1() throws IOException {  
    p2();  
}
```

(b)

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved.

19

Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 9. The new setRadius method throws an exception if radius is negative.

CircleWithException

TestCircleWithException

Run

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved.

20

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;  
}
```



Trace a Program Execution

Suppose no exceptions in the statements

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



Trace a Program Execution

The final block is always executed

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



Trace a Program Execution

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

Next statement;

Next statement in the method is executed



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



Trace a Program Execution

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

Next statement;

The exception is handled.



Trace a Program Execution

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

Next statement;

The final block is always executed.



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.



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.



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



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



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. Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.



When to Throw Exceptions

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



When to Use Exceptions

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

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



When to Use Exceptions

is better to be replaced by

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



The File Class

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



Obtaining file properties and manipulating file

java.io.File
+File(pathname: String)
+File(parent: String, child: String)
+File(parent: File, child: String)
+exists(): boolean
+canRead(): boolean
+canWrite(): boolean
+isDirectory(): boolean
+isFile(): boolean
+isAbsolute(): boolean
+isHidden(): boolean
+getAbsolutePath(): String
+getCanonicalPath(): String
+getName(): String
+getPath(): String
+getParent(): String
+lastModified(): long
+length(): long
+listFiles(): File[]
+delete(): boolean
+renameTo(dest: File): boolean
+mkdir(): boolean
+mkdirs(): boolean

Creates a File object for the specified path name. The path name may be a directory or a file.

Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.

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.

Returns true if the file or the directory represented by the File object exists.

Returns true if the file represented by the File object exists and can be read.

Returns true if the file represented by the File object exists and can be written.

Returns true if the File object represents a directory.

Returns true if the File object represents a file.

Returns true if the File object is created using an absolute path name.

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.

Returns the complete absolute file or directory name represented by the File object.

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).

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.

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.

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.

Returns the time that the file was last modified.

Returns the size of the file, or 0 if it does not exist or if it is a directory.

Returns the files under the directory for a directory File object.

Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.

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.

Creates a directory represented in this File object. Returns true if the directory is created successfully.

Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

Problem: Explore File Properties

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

```
Command Prompt
C:\book>java TestFileClass
Does it exist? true
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
What is its absolute path? C:\book\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? .\image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? :
What is the name separator? \
C:\book>
```

```
Command Prompt - telnet parida
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
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
What is its absolute path? /home/liang/book/./image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? ./image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? :
What is the name separator? /
$ =
```

TestFileClass

Run

Text I/O

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



Writing Data Using PrintWriter

java.io.PrintWriter
+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void
Also contains the overloaded println methods.
Also contains the overloaded printf methods.

Creates a PrintWriter for the specified file.

Writes a string.

Writes a character.

Writes an array of character.

Writes an int value.

Writes a long value.

Writes a float value.

Writes a double value.

Writes a boolean value.

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.

The printf method was introduced in §4.6, “Formatting Console Output and Strings.”

WriteData

Run

Try-with-resources

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

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

WriteDataWithAutoClose

Run

Reading Data Using Scanner

java.util.Scanner
+Scanner(source: File)
+Scanner(source: String)
+close()
+hasNext(): boolean
+next(): String
+nextByte(): byte
+nextShort(): short
+nextInt(): int
+nextLong(): long
+nextFloat(): float
+nextDouble(): double
+useDelimiter(pattern: String): Scanner

Creates a Scanner object to read data from the specified file.
Creates a Scanner object to read data from the specified string.
Closes this scanner.
Returns true if this scanner has another token in its input.
Returns next token as a string.
Returns next token as a byte.
Returns next token as a short.
Returns next token as an int.
Returns next token as a long.
Returns next token as a float.
Returns next token as a double.
Sets this scanner's delimiting pattern.

ReadData

Run

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 as follows:

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

For example, invoking

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

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

