Advanced Computer Programming

[Lecture 10]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

EXCEPTION HANDLING

There are two aspects to dealing with program errors: **detection** and **handling**.
Exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.

Throwing Exceptions

Usage

To signal an exceptional condition, use the **`throw`** statement to throw an exception object.

Syntax `throw` *exceptionObject*;

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

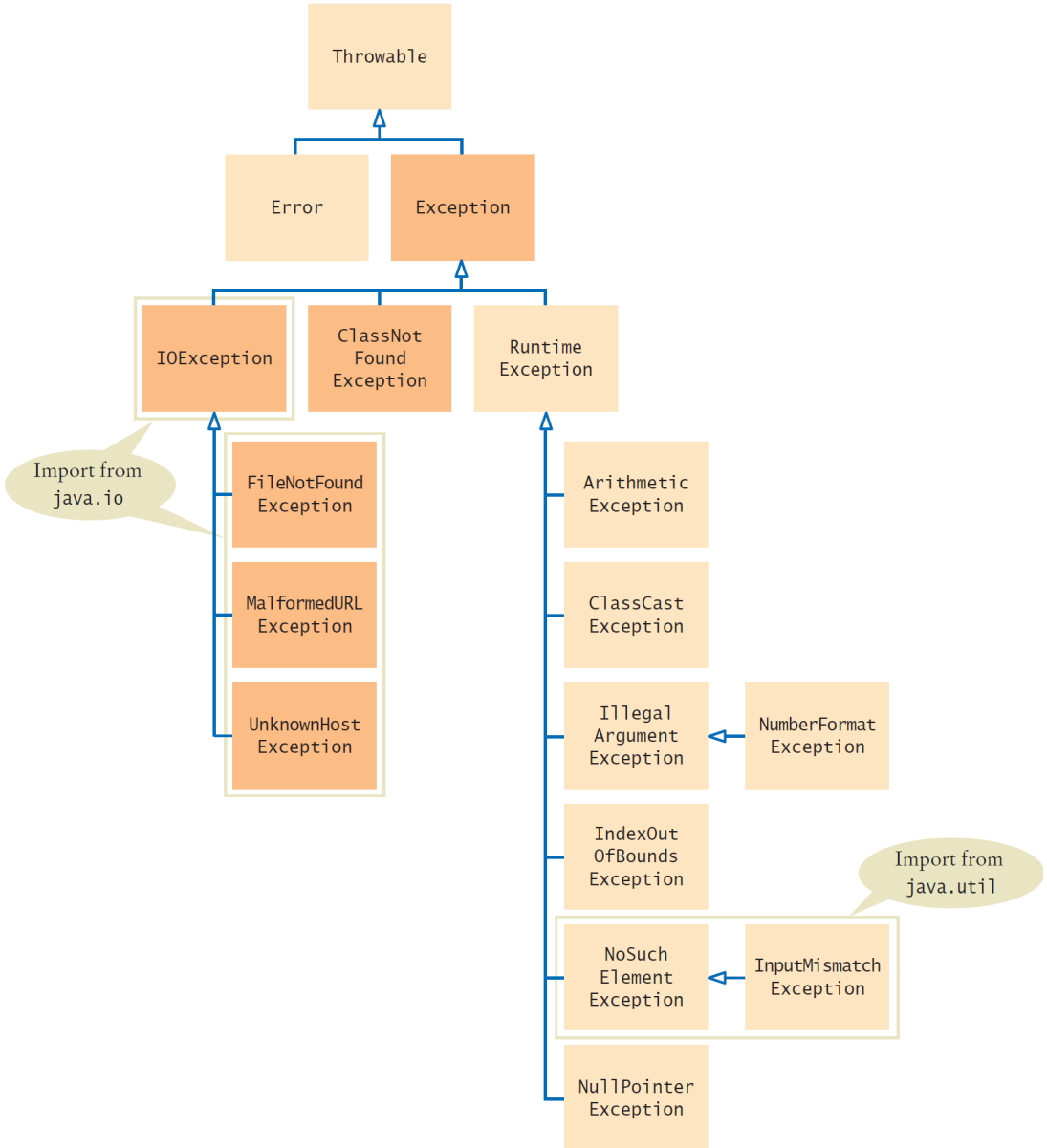
A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

The Java library provides many classes to signal all sorts of exceptional conditions.

EXCEPTION hierarchy



Catching Exceptions

When you throw an exception, processing continues in an exception handler.

Usage

Place the statements that can cause an exception inside a `try` block, and the handler inside a **`catch`** clause.

Catching Exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage);
}
```

This constructor can throw a FileNotFoundException.

When an IOException is thrown, execution resumes here.

This is the exception that was thrown.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

A FileNotFoundException is a special case of an IOException.

Catching Exceptions

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println(exception.getMessage());
}
```

Three exceptions may be thrown in this `try` block:

- The `Scanner` constructor can throw a `FileNotFoundException`.
- `Scanner.next` can throw a `NoSuchElementException`.
- `Integer.parseInt` can throw a `NumberFormatException`.

Catching Exceptions

- If a `FileNotFoundException` is thrown, then the `catch` clause for the `IOException` is executed. (If you look at Figure 2, you will note that `FileNotFoundException` is a descendant of `IOException`.) If you want to show the user a different message for a `FileNotFoundException`, you must place the `catch` clause *before* the clause for an `IOException`.
- If a `NumberFormatException` occurs, then the second `catch` clause is executed.
- A `NoSuchElementException` is *not caught* by any of the `catch` clauses. The exception remains thrown until it is caught by another `try` block.

Checked Exceptions

Definition

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

Checked Exceptions

Definition

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

In Java, the exceptions that you can throw and catch fall into three categories:

- Internal errors are reported by descendants of the type `Error`.
- Descendants of `RuntimeException`, such as as `IndexOutOfBoundsException` or `IllegalArgumentException` indicate errors in your code (Unchecked Exceptions).
- All other exceptions are checked exceptions. These exceptions indicate that something has gone wrong for some external reason beyond your control.

Catching Exceptions

```
try
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile); // Throws FileNotFoundException
   . . .
}
catch (FileNotFoundException exception) // Exception caught here
{
   . . .
}
```

However, it commonly happens that the current method *cannot handle* the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply a method with a `throws` clause.

```
public static String readData(String filename) throws FileNotFoundException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   . . .
}
```

The Finally Clause

Usage

Once a `try` block is entered, the statements in a **`finally`** clause are guaranteed to be executed, whether or not an exception is thrown.

Example:

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

The Finally Clause

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

This variable must be declared outside the `try` block so that the `finally` clause can access it.

This code may throw exceptions.

This code is always executed, even if an exception occurs.

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

Exercise

Add exception handling to the previous exercise.