Chapter 17 Exceptions and Assertions

- CS436/636: subset of slides from Java textbook by Liang with a few added comments

Objectives

- To know what is exception and what is exception handling (§17.2).
- To distinguish exception types: Error (fatal) vs. Exception (non-fatal), and checked vs. uncheck exceptions (§17.2).
- To declare exceptions in the method header (§17.3).
- To throw exceptions out of a method (§17.3).
- To write a try-catch block to handle exceptions (§17.3).
- To explain how an exception is propagated (§17.3).
- To rethrow exceptions in a try-catch block (§17.4).
- To use the finally clause in a try-catch block (§17.5).
- To know when to use exceptions (§17.6).

Syntax Errors, Runtime Errors, and Logic Errors

You learned that there are three categories of errors: syntax errors, runtime errors, and logic errors. *Syntax errors* arise because the rules of the language have not been followed. They are detected by the compiler. *Runtime errors* occur while the program is running if the environment detects an operation that is impossible to carry out. *Logic errors* occur when a program doesn't perform the way it was intended to.

Runtime Errors

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

    // Display the result
    System.out.println(
      "The number entered is " + number);
  }
}
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.

CS436/636: Scanner only throws unchecked exceptions, so we don't have to declare them. However, SQLException of JDBC is a *checked exception*, needs more work by the programmer.

Catch Runtime Errors

Run

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

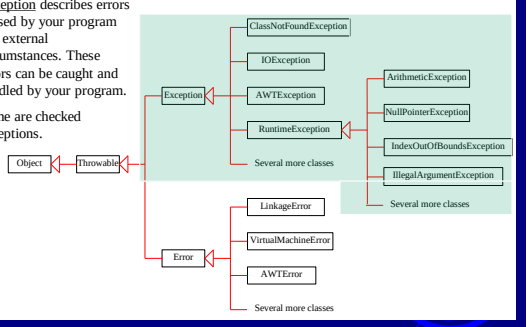
```
import java.util.*;

public class HandleExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = scanner.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)");
        scanner.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

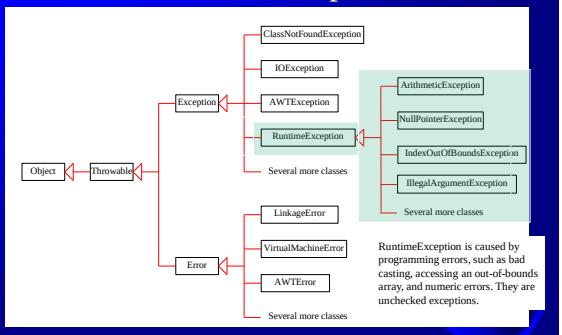
Exceptions

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

Some are checked exceptions.



Runtime Exceptions



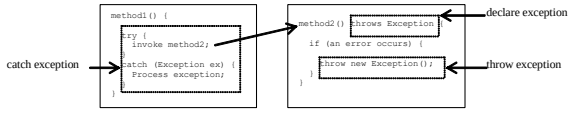
Checked Exceptions vs. Unchecked Exceptions

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.

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;
```

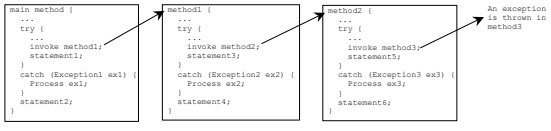
Throwing Exceptions Example

```
/** 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



- If the exception is of type Exception3, it will cause the catch of method2 to execute.
- If the exception is of type Exception2, it will cause the catch of method1 to execute.
- If the exception is of type Exception1, it will cause the catch of method to execute.

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)

Rethrowing Exceptions

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


The `finally` Clause

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

CS436/636: we'll use finally to close our JDBC objects, needed because they involve OS resources.

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

The final block is always executed

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

Next statement;
```

animation

Trace a Program Execution

Next statement in the method is executed

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

Next statement;
```

animation

Trace a Program Execution

Suppose an exception of type Exception1 is thrown in statement2

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

Next statement;
```

animation

Trace a Program Execution

The exception is handled.

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

Next statement;
```

animation

Trace a Program Execution

The final block is always executed.

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

Next statement;
```

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. 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.

However, ignoring an error is a "code smell": always explain in a comment if you choose to do nothing about an exception.