Chapter 14

Exception Handling &

Text Files
Motivations

- When a Java program runs into an unexpected runtime error, the program terminates abnormally.

- How can you handle these events so that the program can (under your control) continue to run or terminate gracefully?

- PAYOFF IDEA The topics in this chapter will allow you to create stronger, more resilient programs that react well to abnormal execution-time situations.
Exception-Handling Overview

**Example** – Scenario 1

The following three code samples directly perform a division operation (CAUTION: Dividing by zero is an undefined operation!)

1. Naive code – no protection
2. Fix it using an *if* statement
3. Fix-it with *Exception* handler.

**Scenario 2.** *What if the runtime error occurs in a called method?*
public class Quotient {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        System.out.println(number1 + " / " + number2 + " is " + 
                            (number1 / number2));
    }
}

CONSOLE

Enter two integers: 100 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at csu.matos.Quotient.main(Quotient.java:15)
    at csu.matos.Driver.main(Driver.java:12)
public class QuotientWithIf {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        if (number2 != 0)
            System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
        else
            System.out.println("Divisor cannot be zero ");
    }
}

CONSOLE

Enter two integers: 100 0
Divisor cannot be zero
public class QuotientWithException {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        try {
            double result = number1 / number2;
            System.out.println(number1 + " / " + number2 + " is " + result);
        } catch (Exception ex) {
            System.out.println("Exception: an integer cannot be divided by zero ");
        }

        System.out.println("Execution continues ...");
    }
}

3. Protect code with Exception

**CONSOLE**

Enter two integers: 100 0
Exception: an integer cannot be divided by zero
Execution continues ...
Exception-Handling Advantage

Exception handling separates error-handling code from normal programming tasks, consequently making programs *easier to read* and to *modify*.

```java
try {
    Code to try;
    Throw an exception with a throw statement or from method if necessary;
    More code to try;
}

// Business logic (this is what needs to be done)

catch (type ex) {
    Code to process the exception;
}

// Deal with troubles here
```
Exception-Handling Overview

```java
public class QuotientWithException {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        try {
            if (number2 == 0)
                throw new ArithmeticException("Divisor cannot be zero");

            System.out.println(number1 + " / " + number2 + " is " +
                                (number1 / number2));
        }
        catch (Exception ex) {
            System.out.println("Exception: an integer " +
                                "cannot be divided by zero ");
        }

        System.out.println("Execution continues ...");
    }
}
```

**Throwing and catching a ‘local’ exception**

**Your turn**

1. Replace with `Exception(...)`
2. What if the try-catch statement is not used?
3. Use `ex.getMessage()`

**CONSOLE**

```
Enter two integers: 100 0
Exception: an integer cannot be divided by zero
Execution continues ...
```
Exception-Handling Overview

Some (of the many) pre-defined Java Exceptions

ArithmeticException
ClassNotFoundException
IllegalAccessException
IOException
EOFException
FileNotFoundException
InputMismatchException
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException

...
Exception Management Advantages

- The Exception-mechanism enables a called method to demand the *strongest attention* from its caller (by performing a *throw-statement*).

- *Without this capability, the called method must handle the problem or terminate the program* (consider the division problem)
Example: Handling InputMismatchException

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

```java
public class InputMismatchExceptionDemo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean continueInput = true;
        while (continueInput) {
            try {
                System.out.print("Enter an integer: ");
                int number = input.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)" );
                input.nextLine(); // discard input
            }
        }
    }
}
```

**CONSOLE**
Enter an integer: 55.55
Try again. (Incorrect input: an integer is required)
Enter an integer: 66
The number entered is 66
Exception Types

System errors are thrown by JVM. There is little you can do beyond notifying the user and trying to terminate the program gracefully.

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

Unchecked

checked
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 (poor logic, bad programming,...) For example
  - `NullPointerException`,
  - `IndexOutOfBoundsException`
- Unchecked exceptions can occur anywhere in the program.
- Java does not mandate you to write code to catch unchecked exceptions (*bad code happens!*).
Declaring Exceptions

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

```java
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,

```java
throw new MyNewException(optionalMsg);

MyNewException ex = new MyNewException();
throw ex(optionalMsg);
```
/** 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

```java
try {
    statements;    // Statements that may throw exceptions
} catch (Exception1 ex1) {
    handler for exception1;
} catch (Exception2 ex2) {
    handler for exception2;
} ...
catch (ExceptionN exn) {
    handler for exceptionN;
}
```

Catching multiple exceptions (one-at-the-time)
Catching Exceptions

```java
main method {
    ... try {
        ... invoke method1;
        statement1;
    } catch (Exception1 ex1) {
        Process ex1;
    }
    statement2;
}

method1 {
    ... try {
        ... invoke method2;
        statement3;
    } catch (Exception2 ex2) {
        Process ex2;
    }
    statement4;
}

method2 {
    ... try {
        ... invoke method3;
        statement5;
    } catch (Exception3 ex3) {
        Process ex3;
    }
    statement6;
}
```

An exception is thrown in method3

Call Stack

```
- main method
- method1
- method2
- method3
```

**Figure 13.3** 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.
Order Matters!

- The order in which exceptions are specified in catch blocks is important.
- A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type.

![Try and Catch Blocks]

**Place exceptions in catch-block from most specialized subclasses first!**

*Note: RunTimeException is a subclass of Exception. See previous chart.*
Java forces you to deal with checked exceptions.

If a method declares a checked exception you must invoke it in a try-catch block or declare to throw the exception in the calling method.
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 8).

- The new `setRadius` method throws an exception if radius is negative.
public class TestCircleWithException {

    public static void main(String[] args) {

        try {
            CircleWithException c1 = new CircleWithException(5);
            CircleWithException c2 = new CircleWithException(-5);
            CircleWithException c3 = new CircleWithException(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }
        System.out.println("Number of objects created: " +
                          CircleWithException.getNumberOfObjects());
    }
}
public class CircleWithException {

    /** The radius of the circle */
    private double radius;

    /** Construct a circle with a specified radius */
    public CircleWithException(double newRadius) {
        setRadius(newRadius);
    }

    /** Construct a circle with radius 1 (Default) */
    public CircleWithException() {
        this(1.0);
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

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

```java
try {
    statements;
}
catch(SomeException ex) {
    perform some operations here;
    throw ex;
}
```
The **finally** Clause

- Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught.

- Java has a **finally** clause that can be used to accomplish this objective.

```java
try {
    statements;
} catch (TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
```
Cautions When Using Exceptions

- Exception handling usually consumes 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.

However (in general) the benefits out-weight the risks!
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.
    A simple *if-statement* should be sufficient.
Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.

- Define custom exception classes if the predefined classes are not sufficient.

- Define custom exception classes by extending Exception or a subclass of Exception.
Example: Defining Custom Exceptions

Defining a custom exception for rejecting a negative radius value (Note: predefined `IllegalArgumentException` could have been used instead)

```java
public class MyInvalidRadiusException extends Exception {

    private String myMsg = "";

    public MyInvalidRadiusException(String userMsg) {
        // user-defined message
        myMsg = userMsg;
    }

    public MyInvalidRadiusException() {
        // default message
        myMsg = "Invalid RADIUS. It must be a positive value";
    }

    @Override
    public String getMessage() {
        return myMsg;
    }
}
```
Example: Defining Custom Exceptions

This is a fragment of the Circle2 class throwing the custom exception

```java
public class Circle2 {
    private double radius;

    public Circle2() throws MyInvalidRadiusException{
        setRadius(0);
    }

    public Circle2(double radius) throws MyInvalidRadiusException{
        setRadius(radius);
    }

    /** Set a new radius - it must be a positive number
     * @throws Exception */
    public void setRadius(double radius) throws MyInvalidRadiusException{
        if ( radius >= 0)
            this.radius = radius;
        else
            throw new MyInvalidRadiusException("Radius must be positive "
                                                            + radius);
    }

    ...
}
```
• An **assertion** is a Java statement that enables you to make working assumptions about your program.

• An assertion contains a Boolean expression that should be true during program execution.

• Assertions can be used to assure program **correctness** and avoid logic errors.

• By default, the assertions are disabled at runtime.
Declaring Assertions

An *assertion* is declared using the Java keyword `assert` (appeared first in JDK 1.4).

Syntax:

```
assert assertion;  or
assert assertion : detailMessage
```

Where

*assertion* is a Boolean expression and
*detailedMessage* is a primitive-type or an Object value.
Executing Assertions

• When an assertion statement is processed, Java evaluates the assertion.
  
  1. If it is false, an `AssertionError` will be thrown.
  2. The `AssertionError` class has
     
     • a no-arg constructor and
     • seven overloaded single-argument constructors of type `int`, `long`, `float`, `double`, `boolean`, `char`, and `Object`.

• For an assert statement with no detail message, the no-arg constructor of `AssertionError` is used.

• For an assert statement with a detail message, an appropriate `AssertionError` constructor is used to match the data type of the message.
public class AssertDemo {
    public static void main(String[] args) {
        int i; int sum = 0;
        for (i = 0; i < 10; i++) {
            sum += i;
        }
        System.out.printf("i= %d  sum= %d \n", i, sum);
        assert (i == 10);
        assert (sum > 10 && sum < 40) : "sum is " + sum;
    }
}

CONSOLE
i= 10  sum= 45
Exception in thread "main" java.lang.AssertionError: sum is 45
at csu.matos.AssertDemo.main(AssertDemo.java:11)
Running Eclipse with Assertions

1. Eclipse Main Menu: Drop-Down Run Button
2. Run Configurations ...
Running Eclipse with Assertions

3. Click on tab: \((X) = \text{Arguments}\)
4. In box “VM Arguments” enter: \(-\text{enableassertions}\) (or just \(-\text{ea}\))
5. Click Run button.
1. Assertion should not be used to replace exception handling.
2. Exception handling deals with unusual circumstances during normal program execution.
3. Assertions are to assure the correctness of the program.
4. Assertions are not used for production-level programs, but for internal consistency and validity checks.
5. JUnit is a similar tool used often in Eclipse to assess validity/consistency of a program.
Text Files: The **File** Class

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’s **memory** 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.
Text Files: The **File** Class

- Every file is placed in a **directory** in the file system.
- An **absolute file name** (or full name) contains a file name, directory path and drive letter.
- Absolute file names are machine dependent. For example in a
  - Windows machine a file name looks like: `c:\book\Welcome.java`
  - UNIX machine a full file name looks like: `/home/liang/book/Welcome.Java`

- The complete directory path for a **relative file** name is omitted. For example, `Welcome.java` is a relative file name. If the current working directory is `c:\book`, the absolute file name would be `c:\book\Welcome.java`. 
Text Files: The **File** Class

- Every file is placed in a **directory** in the file system.
- An **absolute file name** (or full name) contains a file name, directory path and drive letter.
- Absolute file names are machine dependent. For example in a
  - Windows machine a file name looks like: `c:\book\Welcome.java`
  - UNIX machine a full file name looks like: `/home/liang/book/Welcome.Java`

- The complete directory path for a **relative file** name is omitted. For example, `Welcome.java` is a relative file name. If the current working directory is `c:\book`, the absolute file name would be `c:\book\Welcome.java`. 
Text Files: The **File** Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+File(pathname: String)</code></td>
<td>Creates a File object for the specified path name. The path name may be a directory or a file.</td>
</tr>
<tr>
<td><code>+File(parent: String, child: String)</code></td>
<td>Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.</td>
</tr>
<tr>
<td><code>+File(parent: File, child: String)</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>+exists(): boolean</code></td>
<td>Returns true if the file or the directory represented by the File object exists.</td>
</tr>
<tr>
<td><code>+canRead(): boolean</code></td>
<td>Returns true if the file represented by the File object exists and can be read.</td>
</tr>
<tr>
<td><code>+canWrite(): boolean</code></td>
<td>Returns true if the file represented by the File object exists and can be written.</td>
</tr>
<tr>
<td><code>+isDirectory(): boolean</code></td>
<td>Returns true if the File object represents a directory.</td>
</tr>
<tr>
<td><code>+isFile(): boolean</code></td>
<td>Returns true if the File object represents a file.</td>
</tr>
<tr>
<td><code>+isAbsolute(): boolean</code></td>
<td>Returns true if the File object is created using an absolute path name.</td>
</tr>
<tr>
<td><code>+isHidden(): boolean</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>+getAbsolutePath(): String</code></td>
<td>Returns the complete absolute file or directory name represented by the File object.</td>
</tr>
<tr>
<td><code>+getCanonicalPath(): String</code></td>
<td>Returns the same as <code>getAbsolutePath()</code> except that it removes redundant names, such as &quot;.&quot; and &quot;.&quot;, from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).</td>
</tr>
<tr>
<td><code>+getName(): String</code></td>
<td>Returns the last name of the complete directory and file name represented by the File object. For example, new File(&quot;c:\book\test.dat&quot;).getName() returns test.dat.</td>
</tr>
<tr>
<td><code>+getPath(): String</code></td>
<td>Returns the complete directory and file name represented by the File object. For example, new File(&quot;c:\book\test.dat&quot;).getPath() returns c:\book\test.dat.</td>
</tr>
<tr>
<td><code>+getParent(): String</code></td>
<td>Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File(&quot;c:\book\test.dat&quot;).getParent() returns c:\book.</td>
</tr>
<tr>
<td><code>+lastModified(): long</code></td>
<td>Returns the time that the file was last modified.</td>
</tr>
<tr>
<td><code>+length(): long</code></td>
<td>Returns the size of the file, or 0 if it does not exist or if it is a directory.</td>
</tr>
<tr>
<td><code>+listFiles(): File[]</code></td>
<td>Returns the files under the directory for a directory File object.</td>
</tr>
<tr>
<td><code>+delete(): boolean</code></td>
<td>Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.</td>
</tr>
<tr>
<td><code>+renameTo(dest: File): boolean</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>+mkdir(): boolean</code></td>
<td>Creates a directory represented in this File object. Returns true if the the directory is created successfully.</td>
</tr>
<tr>
<td><code>+mkdirs(): boolean</code></td>
<td>Same as <code>mkdir()</code> except that it creates directory along with its parent directories if the parent directories do not exist.</td>
</tr>
</tbody>
</table>
Text Files: The **PrintWriter** Class

To create an output text file define a **PrintWriter** object as follows:

```java
File outputFile = new File("testdata.txt");

PrintWriter printWriter = new PrintWriter(outputFile);
```
Text Files: The **Scanner** Class

To create an input text file define a **Scanner** object as follows:

```java
File inputFile = new File("testdata.txt");
Scanner scanner = new Scanner(inputFile);
```

*System.in* for console input
package csu.matos;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Driver {

    /**
     * Author: V. Matos
     * Date: 7-Feb-2013
     * Goal: Create a small text file on disk using PrintWriter.
     *       Check the file's existence. Read its data using Scanner
     */
    public static void main(String[] args) throws FileNotFoundException {

        File outputFile = new File("testdata.txt");

        if (outputFile.exists()) {
            System.out.println("Error - file already exists");
            System.exit(0);
        }
    }
}
PrintWriter printWriter = new PrintWriter(outputFile);

printWriter.println( "Circle 1" );
printWriter.println( "Circle red true 1" );
printWriter.println( "Rectangle 10 20" );
printWriter.println( "Rectangle blue true 1 5" );
printWriter.close();

File inputFile = new File( "testdata.txt" );

Scanner scanner = new Scanner(inputFile);

while ( scanner.hasNext() ){
    String entireLine = scanner.nextLine();
    System.out.println(entireLine);
}

scanner.close();

}
Using a GUI helper to select a file from the file system.

```java
public static void main(String[] args) throws FileNotFoundException {
    JFileChooser filechooser = new JFileChooser();
    if (filechooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION){
        File inputFile = filechooser.getSelectedFile();
        Scanner scanner = new Scanner(inputFile);
        while (scanner.hasNext()) {
            String line = scanner.nextLine();
            System.out.println(line);
        }
        scanner.close();
    }
}
```
To read a data set from the web you use a **URL** (Uniform Resource Locator). For example the URL to reach **NPR Politics** RSS feed is: 

http://www.npr.org/rss/rss.php?id=1014

```java
public static void main(String[] args) throws IOException {
    URL url = new URL("http://www.npr.org/rss/rss.php?id=1014");
    Scanner scanner = new Scanner(url.openStream());
    while (scanner.hasNext()) {
        String line = scanner.nextLine();
        System.out.println(line);
    }
    scanner.close();
}
```