Exception Handling

Cheng, Wei  COMP110-001  June 6, 2014
Today

• Exception Handling
  – Important in developing Java code. But not a major focus of this course

– This lecture:
  • A lot of concepts
  • Not much code
Recall Program 2

• Program 2: A Simple Console Calculator

Welcome to the simple console calculator!
The following operations are supported: +, -, *, /, %.
Enter your calculation separated by spaces: (Example: 2.5 + 3)
2.5 / 0
Division by 0 is not allowed

If the user try to divide by 0, prints out a message
Recall Program 2

• Program 2: A Simple Console Calculator

Welcome to the simple console calculator!
The following operations are supported: +, -, *, /, %.
Enter your calculation separated by spaces: (Example: 2.5 + 3)
2.5 / 0
Division by 0 is not allowed

• Each of you used an if statement to test whether it is a division and the second operand is 0.
• If it is division by 0, did you still do the division after you print out the message?
Recall Program 2

- If you choose not to do, you have handled this case by skipping the result calculation part

- If you still calculates the result, you will probably get an error like this:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Calculator.main(Calculator.java:XX)
```

The program terminated due to the error.
What is the right thing to do...

• When your code detects a problem?

• In program 2, we printed out a message to indicate a problem. And may choose to skip result calculation.

• Not so much problem for a small program. We have control of everything involved.

• But things quickly become messy when we want to write something slightly bigger.
What if....

- What if you are writing some classes for others to use...
- What do you plan to do when your code detects some problem?

- Do you print out a message?
  - What if the program that uses your class runs in graphical mode?
  - Does the program really want some “uncontrolled” print-outs?

- Do you just let resulting errors terminate the program?
  - Sounds like a terrible idea in most cases
  - But if your class should do something and it is not performed properly, how to inform the program that uses the class?
  - E.g., a method in your class is called and is supposed to return some value. When your code sees error, should it still return any value?
    - If yes, what value?
What if....

• You are using someone’s class for your program.
• E.g., you use the classes provided by Java to read from or write to a file.

• If some problems happen in reading / writing (file not found, cannot read/write), how does your program get notified?
The need of a formal mechanism

• A formal mechanism is needed to handle “problems”

• “Problems” in one class should be reported and handled differently in different programs.

• This mechanism is different from return values in method-calling
Throw and Catch

• In Java, the mechanism is called “Exception Handling”
  – **Throw an exception**: report a problem and asks for some code to handle it properly
  – **Catch an exception**: a piece of code dedicated to handle one or more specific types of problem
Declaring, Throwing, and Catching Exceptions

Overview of Exception handling
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");
}

Step 1: add “throws ExceptionType” in method header
Step 2: when problem occurs, “throw new ExceptionType( .... );”
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();
    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.
Catching Example

```java
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.
The Java Exception Hierarchy

The Java Exception Hierarchy is a classification of exceptions that occur during the execution of Java programs. It is organized in a hierarchical structure, with more specific exceptions inheriting from more general ones.

- **Object**
- **Throwable**
  - **Exception**
    - **NullPointerException**
    - **IndexOutOfBoundsException**
    - **ClassCastException**
    - **IllegalArgumentException**
    - **IllegalAccessException**
    - **ArithmeticException**
    - **NullPointerException**
    - **IndexOutOfBoundsException**
    - **VirtualMachineError**
    - **IOException**
    - **AWTError**
    - **AWTException**
    - **LinkageError**
    - **ClassNotFoundException**
    - **VirtualMachineError**
    - **IOException**
    - **RuntimeException**
    - **Several more classes**
  - **Exception**
    - **NullPointerException**
    - **IndexOutOfBoundsException**
    - **VirtualMachineError**
    - **IOException**
    - **AWTError**
    - **AWTException**
    - **LinkageError**
    - **ClassNotFoundException**
    - **VirtualMachineError**
    - **IOException**
    - **RuntimeException**
    - **Several more classes**
  - **Error**
    - **NullPointerException**
    - **IndexOutOfBoundsException**
    - **VirtualMachineError**
    - **IOException**
    - **AWTError**
    - **AWTException**
    - **LinkageError**
    - **ClassNotFoundException**
    - **VirtualMachineError**
    - **IOException**
    - **RuntimeException**
    - **Several more classes**

The diagram illustrates the inheritance relationships between these exception classes, with unchecked exceptions at the bottom of the hierarchy.
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.
Let’s see more examples with Checked Exceptions

• We want to use Java’s FileInputStream class to read a file. So we do this:

```java
public void methodReadFile( String filename ) {
    FileInputStream fstream = new FileInputStream(filename);
    ...... call fstream’s method to read from file
}
```
Let’s see more examples with Checked Exceptions

- The constructor can throw a checked exception
- So we have to handle it
Let’s see more examples with Checked Exceptions

• So we revise our code to handle this exception:

```java
public void methodReadFile(String filename) {
    try {
        FileInputStream fstream = new FileInputStream(filename);
        ...... call fstream’s method to read from file
    } catch (FileNotFoundException e) {
        System.out.println("I cannot open the file “ + filename);
        System.out.println("This is the error message “ + e.getMessage());
        // do sth more meaningful here
    }
}
```
public void callDivide() {
    Scanner s = new Scanner(System.in);
    try {
        int op1 = s.nextInt();
        int op2 = s.nextInt();
        System.out.println( divide( op1, op2 ) );
    } catch (BadNumberException e) {
        System.out.println( e.getMessage() );
        // do sth smart here
    }
}
The finally Clause

try {
    statements;
}
catch(TheException ex) {
    handling ex;
}
finally {
    finalStatements;
}
Suppose no exceptions in the statements
Trace a Program Execution

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

The final block is always executed.

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

Next statement in the method is executed
Trace a Program Execution

```java
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) {

}
finally {
    finalStatements;
}

Next statement;

The exception is handled.
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements
}
Next statement;

The final block is always executed.
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
finally {
    finalStatements;
}
Trace a Program Execution

```java
try {
    statement1;
    statement3;
} catch (Exception1 ex) {
    handling ex;
} catch (Exception2 ex) {
    handling ex;
    throw ex;
} finally {
    finalStatements;
}

Next statement;

statement2 throws an exception of type Exception2.
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    throw ex;
}
finally {
    finalStatements;
}
Next statement;
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    // Execute the final block
}

Next statement;
try {
    statement1;
    statement2;
    statement3;
} 
catch(Exception1 ex) {
    handling ex;
} 
catch(Exception2 ex) {
    handling ex;
} 
finally {
    finalStatements;
} 
Next statement;
Rethrow the exception and control is transferred to the caller
Extra Reading

http://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html
Next Class

• Read from / Write to Files