Exception Handling and Text I/O (Part 1)

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 12
Announcements

• Assignment 6 is due Nov 18, 11:59 PM
• Quiz 6 is Nov 20
• Assignment 7 will be released Nov 18
  – Due Nov 25, 11:59 PM
• Educational research study
  – Nov 20, weekly reflection
• Degree planning assignment due Dec 2, 11:59 PM
• Reading
  – Chapter 12
Exceptions

• Exceptions are runtime errors caused by your program and external circumstances
  – These errors can be caught and handled by your program
Example: integer divide by zero

```java
import java.util.Scanner;

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));
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
Example: integer divide by zero

- Exception in thread "main" java.lang.ArithmeticException: / by zero
- First approach
  - Mitigate exception with if statement
  - Create a method, so we can reuse it
Example: integer divide by zero

```java
import java.util.Scanner;

public class QuotientWithMethod {
    public static int quotient(int number1, int number2) {
        if (number2 == 0) {
            System.out.println("Divisor cannot be zero");
            System.exit(1);
        }
        return number1 / number2;
    }

    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();

        int result = quotient(number1, number2);
        System.out.println(number1 + " / " + number2 + " is "+ result);
    }
}
```

Problem: a method should never terminate a program.
Example: integer divide by zero

• Exception in thread "main" java.lang.ArithmeticException: / by zero

• First approach
  – Mitigate exception with if statement
  – Create a method, so we can reuse it
    • Problem: a method should never terminate a program

• Second approach
  – Have the method notify the caller
Example: integer divide by zero

```java
import java.util.Scanner;

public class QuotientWithException {
    public static int quotient(int number1, int number2) {
        if (number2 == 0)
            throw new ArithmeticException("Divisor cannot be zero");
        return number1 / number2;
    }

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

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

- **Throw an `ArithmeticException`**
- **Try something that may throw an exception**
- **Catch an `ArithmeticException`**
- **Handle the caught exception in the catch block**
Exception handling

• Exception handling enables a method to throw an exception to its caller
• Without this capability, a method must handle the exception or terminate the program
• Separates
  – The detection of an error
  – The handling of an error
Exception types

- Exceptions are objects
  - Remember, objects are instances of classes
- The root class for exception is `java.lang.Throwable`
  - Three major types
    - System errors
    - Exceptions
    - Runtime Exceptions
Exception types

- Exception
  - ClassNotFound Exception
  - IO Exception
  - RuntimeException
  - LinkageError
  - VirtualMachineError
- Error
  - IllegalArgumentException
- Throwable
  - Many more classes
  - Many more classes

- Object
  - Throwable

CSE 8B, Fall 2020
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.
Exception

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.
RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.
Exception types

• Exceptions are objects
  – Remember, objects are instances of classes

• The root class for exception is `java.lang.Throwable`
  – All Java exception classes inherit directly or indirectly from Throwable

• You can create your own exception classes by extending `Exception` or a subclass of `Exception`
Unchecked exceptions vs. checked exceptions

- **RuntimeException**, **Error**, and their subclasses are known as *unchecked exceptions*
  - Usually programming logic error that are unrecoverable
  - These should be corrected in the program
- All other exceptions are known as *checked exceptions*
  - The compiler forces the programmer to check and deal with these exceptions
Unchecked exceptions

Java does not mandate that you write code to catch or declare unchecked exceptions.
Declaring, throwing, and catching exceptions

```java
method1() {
    try {
        invoke method2;
    } catch (Exception ex) {
        Process exception;
    }
}

method2() throws Exception {
    if (an error occurs) {
        throw new Exception();
    }
}
```

catch exception throw exception declare exception
Declaring exceptions

• Every method must state the types of checked exceptions it might throw
  – This is called *declaring exceptions*

• Examples
  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 called *throwing an exception*

• For 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

• When an exception is thrown, it can be caught and handled in a try-catch block
  – If no exceptions are thrown in the try block, then the catch blocks are skipped
• If an exception is thrown in the try block, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception
  – This is called catching an exception
Catching exceptions

try {
    // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    // Handler for Exception1
}
catch (Exception2 | Exception3 | ... | ExceptionK exVar) {
    // Same code for handling these exceptions
}
...
catch (ExceptionN exVarN) {
    // Handler for ExceptionN
}

The order exceptions are specified is important. A compile error occurs if a catch block for a superclass type appears before a catch block for a subclass type.

If no handler is found, then the program terminates and prints an error message on the console.
Catching exceptions

- The code handling the exception is called the *exception handler*
  - It is found by *propagating the exception* backward through the call stacks, starting from the current method

```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:
}
```

Call Stack

An exception is thrown in method3
Checked exceptions

- Remember, the compiler forces the programmer to check and deal with checked exceptions (i.e., any exception other than Error or RuntimeException)
- 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

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

(a)  

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

(b)
The finally clause

• The finally clause is always executed, regardless of whether an exception occurred.

```java
try {
    // statements
}
catch(TheException ex) {
    // handling statements
}
finally {
    // final statements
}
```
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(TheException ex) {
  // handling statements before rethrowing
  throw ex;
}
```

• You can also throw a new exception along with the original exception
  – This is called *chained exceptions* (see textbook for details)
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}
nextStatement;
try {
    statement1;
    statement2;  // statement2 throws an exception of type Exception2
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}

nextStatement;
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}
nextStatement;
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}

nextStatement;
try {
    statement1;
    statement2;
    statement3;
}
catch(Exception1 ex) {
    handling ex;
}
catch(Exception2 ex) {
    handling ex;
    throw ex;
}
finally {
    finalStatements;
}
nextStatement;

Rethrow the exception and control is transferred to the caller
When to use a try-catch block

• Use a try-catch block to deal with unexpected error conditions
• Do not use it to deal with simple, expected situations
  – For example, use this
    ```java
    if (refVar != null)
        System.out.println(refVar.toString());
    else
        System.out.println("refVar is null");
    ```
  instead of this
    ```java
    try {
        System.out.println(refVar.toString());
    }
    catch (NullPointerException ex) {
        System.out.println("refVar is null");
    }
    ```
When to throw exceptions

• Remember, an exception occurs in a method
• If you want the exception to be processed by its caller, then you should create an exception object and throw it
• If you can handle the exception in the method where it occurs, then there is no need to throw it
Defining custom exception classes

• Use the exception classes in the Java API whenever possible
• If the predefined classes are insufficient, then you can define a custom exception class by extending the java.lang.Exception class
Exception handling

• Exception handling separates error-handling code from normal programming tasks
  – Makes programs easier to read and to modify
• The **try** block contains the code that is executed in **normal** circumstances
• The **catch** block contains the code that is executed in **exceptional** circumstances
• A method should **throw** an exception if the error needs to be handled by its caller
• Warning: 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
Next Lecture

• Programming with assertions
• Exception handling and text I/O
• Reading
  – Chapter 12