HST 952

Computing for Biomedical Scientists

Lecture 7

Outline

- Information hiding revisited
- Intro to exceptions in java
- Programming examples
- Questionnaire

Information Hiding Revisited

- To achieve information hiding goal, need to make a class's instance variables private and provide public accessor methods for retrieving and setting these variables' values
- For instance variables that have a class type, this may not be enough!
- Variables with primitive types in Java are passed or returned *by value*
  - a copy of the variable's value is passed/returned, the original contents of the variable cannot be modified by changing this copy

Information Hiding Revisited

- In general, class variables are passed or returned *by reference*
  - a copy of the memory address the variable **refers** to is passed, the contents of this memory address can be altered once this address is known

    (Exceptions to this rule in Java are variables of the String and StringBuffer classes which act like variables of primitive types when passed/returned)

Information Hiding Revisited

- cloning a variable is one solution to this problem
- a clone has the same contents as the original variable but a different address in memory
- altering the clone does not affect the original
- classes that allow cloning implement the *cloneable interface* (e.g. GregorianCalendar)
- some classes do not allow cloning (do not implement the cloneable interface) e.g. String, StringBuffer

Information Hiding Revisited

- Programming example: Person class

Exceptions: Overview

- Exceptions give us a way of organizing a program into sections for the normal case and the exceptional case
  - exception examples:
    division by zero
    incorrect type of input
- Simplifies development, testing, debugging and maintenance
  - errors are easier to isolate

Exceptions: Some Terminology

- *Throwing an exception*: either Java itself or your code signals that something unusual has happened
- *Handling an exception*: responding to an exception by executing a part of the program specifically written for the exception
  - also called *catching an exception*

Exceptions: Some Terminology

- The normal case is handled in a `try` block
- The exceptional case is handled in a `catch` block
- The catch block takes a parameter of type `Exception`
  - it is called the *`catch`-block parameter*
- Exception is a built-in Java class
- If an exception is *thrown* execution in the `try` block ends and control passes to the `catch` block(s) after the `try` block

try-throw-catch Threesome

Basic code organization:

```
try
{
  <code to try>
  if(test condition)
     throw new Exception("Message to display");
  <more code>
}
catch(Exception e)
{
  <exception handling code>
}
```

Programming example: restricting the length of an input string

try-throw-catch Threesome

Try block

Statements execute up to the conditional `throw` statement

If the condition is `true` the exception is thrown

- control passes immediately to the `catch` block(s) after the `try` block

Else the condition is `false`

- the exception is not thrown
- the remaining statements in the `try` block (those following the conditional throw) are executed

try-throw-catch Threesome

Catch block

Executes if an exception is thrown

- may terminate execution with `exit` statement
- if it does not exit, execution resumes after the `catch` block

Statements after the Catch block

Executed if either the exception is not thrown or if it is thrown but the `catch` block does not exit the program

More about the catch-Block

- Although it may look similar to a method definition

  The `catch`-block is **not** a method definition!

- Every `Exception` has a `getMessage` method
  - it retrieves the string given to the exception object when it was thrown, e.g.

```
throw new Exception("This message is retrieved");
```

- A `catch`-block applies only to an immediately preceding `try` block
  - if no exception is thrown the catch block is ignored

Predefined Exception Classes

- `Exception` is the root class of all exceptions

- Many predefined classes throw exceptions
  - the documentation or interface will tell you
  - the exceptions thrown are often also predefined

- Some common predefined exceptions:
  - `IOException`
  - `ClassNotFoundException`, and
  - `FileNotFoundException`

Documentation for Exception class

Taken from http://java.sun.com/j2se/1.3/docs/api/index.html

java.lang

Class Exception

The package which Exception belongs to

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
```

Ancestor class for all Java classes

Sub-class of Object

Sub-class of Throwable

All Implemented Interfaces:

Serializable

A Java class cannot be a subclass of more than one class, but sometimes we need it to have properties of more than one class. To get around this, Java allows a class to implement more than one interface. An interface is a property of a class that says what methods it must have.

Documentation for Exception class

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BadLocationException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, ExpandVetoException, FontFormatException, GeneralSecurityException, IllegalAccessException, InstantiationException,

InterruptedException, IntrospectionException, InvalidMidiDataException, InvocationTargetException, IOException, LastOwnerException, LineUnavailableException, MidiUnavailableException, MimeTypeParseException,

NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, PrinterException, PrivilegedActionException,

PropertyVetoException, RemarshalException, RuntimeException, ServerNotActiveException, SQLException, TooManyListenersException, UnsupportedAudioFileException, UnsupportedFlavorException,

UnsupportedLookAndFeelException, UserException

Documentation for Exception class

public class Exception
extends Throwable

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

Constructor Summary:

Exception()
    Constructs an Exception with no specified detail message.
Exception(String s)
    Constructs an Exception with the specified detail message.

<u>Documentation for Exception class</u>

Methods inherited from class java.lang.Throwable:
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace,
printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail
public Exception()
  Constructs an Exception with no specified detail message.

public Exception(String s)
  Constructs an Exception with the specified detail message.
  Parameters:
    s - the detail message.

Using an Object that May Throw an Exception

```
Sample object = new SampleClass();
try
{
   <Possibly some code>
   object.doStuff();//may throw IOException
   <Possibly some more code>
}
catch(IOException e)
{
   <Code to handle the IOException, probably
  including this line:>
   System.out.println(e.getMessage());
}
```

- Predefined exceptions usually include a meaningful message that is retrieved with `getMessage`

User-Defined Exception Classes

```
public class DivideByZeroException extends Exception
{
    public DivideByZeroException()
    {
        super("Dividing by Zero!");
    }
    public DivideByZeroException(String message)
    {
        super(message);
    }
}
```

- Must be derived from some already defined exception class
- Often the only method you need to define is the constructor
- Include a constructor that takes a `String` message argument
- Also include a default constructor with a call to `super` and default message string

When to Define Your Own Exception Class

- When you use a `throw`-statement in your code you should usually define your own exception class.
- If you use a predefined, more general exception class, then your `catch`-block will have to be general.
- A general `catch`-block could also catch exceptions that should be handled somewhere else.
- A specific `catch`-block for your own exception class will catch the exceptions it should and pass others on (e.g., DivideByZeroException will only catch divisions by zero and will ignore NumberFormatException*s*)

Example: Using the Divide-ByZero-Exception Class

```
public double divide(int numerator, int denominator)
 {
        double quotient = SENTINEL;
        try
        {
            if (denominator == 0)
                throw new DivideByZeroException();
            quotient = numerator/(double)denominator;
            System.out.println(numerator + "/"
                                    + denominator
                                    + " = " + quotient);
        }
        catch(DivideByZeroException e)
        {
            System.out.println(e.getMessage());
        }
        return(quotient);
 }
```

Catching an Exception in a Method other than the One that Throws It

When defining a method you must include a `throws`-clause to declare any exception that might be thrown but is not caught in the method.

- Use a *`throws`-clause* to "pass the buck" to whatever method calls it (pass the responsibility for the `catch` block to the method that calls it)
  - that method can also pass the buck,
    but eventually some method must catch it

- This tells other methods

  "If you call me, you must handle any exceptions that I throw."

Example: throws-Clause

`divide` method

- May throw a `DivideByZeroException` in another method `normal` that calls it
- But the `catch` block is in `main`
- So `normal` must include a *throws-clause* in the first line of the method definition:

```
public void normal() throws
  DivideByZeroException
{
   <statements to define the normal method>
}
```

More about Passing the Buck

Good programming practice:

Every exception thrown should eventually be caught in some method

- Normally exceptions are either caught in a `catch` block or *deferred* to the calling method in a `throws`-clause
- If a method throws an exception, it expects the catch block to be in that method unless it is deferred by a `throws`-clause
  - if the calling method also defers with a `throws`-clause, its calling program is expected to have the `catch` block, etc., up the line all the way to `main`, until a `catch` block is found

Typical Program Organization for Exception Handling in Real Programs

`MethodA` throws `MyException` but defers catching it (by using a `throws`-clause:

```
public void MethodA() throws MyException
{
   throw new MyException("Bla Bla Bla");
}
```

`MethodB`, which calls `MethodA`, catches `MyException` exceptions:

```
public void MethodB()
{
   try
   {
      MethodA();//May throw MyException exception
   }
   catch(MyException e)
   {
      <statements to handle MyException exceptions>
   }
}
```

Uncaught Exceptions

- In any one method you can catch some exceptions and defer others
- If an exception condition occurs but the exception is not caught in the method that throws it or any of its calling methods, either:
  - the program ends abnormally, or,
  - in the case of a GUI using Swing, the program may become unstable

throws-Clauses in Derived Classes

- You cannot add exceptions to the `throws`-clause of a redefined method in a derived class
  - only exceptions in the `throws` -clause of the parent class's method can be in the `throws` -clause of the redefined method in the derived class
- In other words, you cannot throw any exceptions that are not either caught in a `catch` block or already listed in the `throws` -clause of the same method in the base class
- You can, however, declare fewer exceptions in the `throws` -clause of the redefined method

Multiple Exceptions and catch Blocks in a Method

- Methods can throw more than one exception
- The `catch` blocks immediately following the try block are searched in sequence for one that catches the exception type
  - the first catch block that handles the exception type is the only one that executes
- Specific exceptions are derived from more general types
  - both the specific and general types from which they are derived will handle exceptions of the more specific type
- So put the `catch` blocks for the more specific, derived, exceptions early and the more general ones later

Exception: Reality Check

- Exception handling can be overdone
  - use it sparingly and only in certain ways

- If the way an exceptional condition is handled depends on how and where the method is invoked, then it is better to use exception handling and let the programmer handle the exception (by writing the catch block and choosing where to put it)

- Otherwise it is better to avoid throwing exceptions

The finally Block

At this stage of your programming you may not have much use for the `finally` block, but it is included for completeness - you may find it useful in the future

- You can add a `finally` block after the `try/catch` blocks
- `finally` blocks execute whether or not `catch` block(s) execute
- Code organization using `finally` block:

```
try block
catch block
finally
{
  <Code to be executed whether or not an exception is thrown>
}
```

Three Possibilities for a try-catch-finally Block

- The `try`-block runs to the end and no exception is thrown.
  - The `finally`-block runs after the `try`-block.
- An exception is thrown in the `try`-block and caught in the matching `catch`-block.
  - The `finally`-block runs after the `catch`-block.
- An exception is thrown in the `try`-block and there is no matching `catch`-block.
  - The `finally`-block is executed before the method ends.
  - Code that is after the `catch`-blocks but not in a `finally`-block would not be executed in this situation.

Summary

- An exception is an object descended from the `Exception` class
- Exception handling allows you to design code for the normal case separately from that for the exceptional case
- You can use predefined exception classes or define your own
- Exceptions can be thrown by:
  - certain Java statements
  - methods from class libraries
  - explicit use of the `throw` statement
- An exception can be thrown in either
  - a `try` block, or
  - a method definition without a `try` block, but in this case the call to the method must be placed inside a `try` block

Summary

- An exception is caught in a catch block
- When a method might throw an exception but does not have a `catch` block to catch it, usually the exception class must be listed in the `throws`-clause for the method
- A try block may be followed by more than one catch block
  - more than one catch block may be capable of handling the exception
  - the first catch block that can handle the exception is the only one that executes
  - so put the most specific catch blocks first and the most general last
- Every exception class has a `getMessage` method to retrieve a text message description of the exception caught

Read

- Sections 6.4 - 6.5
- Chapter 7
- Chapter 8

Programming examples

- Exceptions (divide by zero)
- Inheritance (Student & Person classes)

Questionnaire