Lecture 3

Inheritance, Interfaces, Exceptions
Java operators

- // single line comment
- /* */ multiline comment
- /** */ Javadoc comment
- { } block of code
- ; end of statement
Java operators

- [ ] array
- () method
- new create object
- + - * / arithmetic
- >> shift right
- << shift left
- >>> shift no sign
- ! logical not
- && logical and
- && logical or
- == equality
- != inequality
- > greater than
- < less than
- >= greater or equal
- <= less or equal
- & bitwise and
- | bitwise or
- ^ bitwise XOR
- ~ bitwise not
Java control structures

- if (boolean expression) else
- switch() case:
- for(\textit{loop variable};condition;\textit{incrementer})
- while() \textbf{and} do{}while();
- break;
- continue;
- \textbf{NO GOTOs!}
- \textbf{?}: \textbf{- the ternary operator} \textbf{a short hand if-then-else}
package Lecture3;

/**
 * @author logan.poelman
 */
public class ControlStructDemo {
    /** Creates a new instance of ControlStructDemo */
    public ControlStructDemo() {
    }

    public static String testTheCaseSwitch(int messageNumber) {
        String msgToReturn;  //this is just a reference to a string
        switch(messageNumber) {  //this actually allocates a string and assigns it to this reference
            case 1: msgToReturn = "This is case 1 msg";
                break;
            case 2 : msgToReturn = "This is case 2 msg";
                break;
            case 3: msgToReturn = "This is case 3 msg";
                break;
            default: msgToReturn = "This is default msg";
                break;
        }
        return msgToReturn;
    }
}
public static void main(String[] args) {
    int i;
    System.out.println("nFor loop test");
    for(i=7;i<17;i++)
    {
        System.out.println("The value of i is = " + i);
    }

    i=30;
    System.out.println("nWhile loop");
    while(i>0)
    {
        System.out.println("The value of i is = " + i);
        i = i - 5;
    }

    System.out.println("nDo-While loop");
    do
    {
        System.out.println("The value of i is = " + i);
    } while(i>0);

    System.out.println("nSwitch-Case");
    String theStringReturned;
    theStringReturned = testTheCaseSwitch(1);
    System.out.println(theStringReturned);
    theStringReturned = testTheCaseSwitch(3);
    System.out.println(theStringReturned);
    theStringReturned = testTheCaseSwitch(999);
    System.out.println(theStringReturned);
}

}
output

For loop test
The value of i is = 7
The value of i is = 8
The value of i is = 9
The value of i is = 10
The value of i is = 11
The value of i is = 12
The value of i is = 13
The value of i is = 14
The value of i is = 15
The value of i is = 16
While loop
The value of i is = 30
The value of i is = 25
The value of i is = 20
The value of i is = 15
The value of i is = 10
The value of i is = 5
Do-While loop
The value of i is = 0
Switch-Case
This is case 1 msg
This is case 3 msg
This is default msg
Java

• Everything inherits from Object
• Even if you don’t say it!

class MyClassName
{
    //put class fields and methods in here
}

class MyClassName extends Object
{
}
## Except Java Primitive types

<table>
<thead>
<tr>
<th>type</th>
<th>size</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>char</td>
<td>2 bytes</td>
<td>unicode</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32768 to 32,767</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2^{31} to 2^{31}</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>-2^{63} to 2^{63}</td>
</tr>
<tr>
<td>boolean</td>
<td>1 byte</td>
<td>true or false</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>-+3.40282347E+38 (IEEE754)</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>-+1.79769313486231570E+308 (IEEE754)</td>
</tr>
</tbody>
</table>
Java has Primitive Types and Reference Types

• Primitive types are not objects just **data**
• They are passed by being pushed on the stack directly (copied on the stack) therefore *passed by value*
• Reference types are **objects**
• Reference types are manipulated by references. These are like memory handles to the actual object that they reference.
• When passed as parameters to a method, a **copy** of the reference gets passed thus in effect the object is *passed by reference*. (a copy of the object isn’t made, only a copy of the reference to the object).
Java initialization

• All variables are initialized (except local variables but we’ll discuss that later)
• numbers to zero
• boolean to false
• reference types to null
defining a base class

public class PhoneNumber
{
    private int areaCode;
    private int exchange;
    private int line;
    public boolean set(int area, int exch, int newLine)
    {
        if (areaCode<111 || areaCode>999)
            or (exchange<000 || exchange >999)
            or (line<0000 || line >9999) then
            return false;
        areaCode = area;
        exchange = exch;
        line = newLine;
        return true;
    }
}
Create Methods

Add a method to the Methods list, then fill it in below. Click an item's "..." button for its customization.

Methods:
- toString()
- get(int, int, int)
- get()
- set(int, int, int)
- TelephoneNumber(int, int, int)

Name: TelephoneNumber

Access Level: <default>

Return Type: void

Parameters (comma-separated):
int area, int Exchange, int line

Exceptions (comma-separated):

Method Type:
- <default>
- final
- static
- native
- synchronized
- abstract
Constructors

• A method with the same name as the class
• Called when you create an object
• In java all objects are created by the new operation
• Constructors must be declared as void meaning it returns nothing
• A class may have multiple constructors that have different parameter signatures
package Lecture3;

/**
 * @author logan.poelman
 */
public class TelephoneNumber {

    private int area;
    private int exchange;
    private int line;

    /** Creates a new instance of TelephoneNumber */
    public TelephoneNumber() {
    }

    public String toString() {
    }

    public void get(int area, int exchange, int line) {
    }

    public String get() {
    }

    public void set(int area, int exchange, int line) {
    }

    public void TelephoneNumber(int area, int Exchange, int line) {
    }

}
package Lecture3;

public class TelephoneNumber {
    private int area;
    private int exchange;
    private int line;
    /** Creates a new instance of TelephoneNumber */
    public TelephoneNumber() {
        System.out.println("In TelephoneNumber() constructor");
    }

    public String toString() {
        return "TelephoneNumber = " + Integer.toString(area) + " - " + Integer.toString(exchange) + " - " + Integer.toString(line);
    }

    public void get(int Area, int exchange, int line) {
        // This won't work!! Passed by value
        area = this.area;
        exchange = this.exchange;
        line = this.exchange;
    }

    public void set(int area, int exchange, int line) {
        this.area = area;
        this.exchange = exchange;
        this.line = line;
    }

    public TelephoneNumber(int area, int Exchange, int line) {
        // This constructor calls a method
        System.out.println("In TelephoneNumber(int area, int Exchange, int line) constructor");
        set(area, exchange, line);
    }
}
testing the **Telephone class**

```java
public static void main(String[] args) {
    TelephoneNumber myNumber; //allocates a reference NOT the object
    TelephoneNumber yourNumber;
    int area, exchange, line;

    myNumber = new TelephoneNumber(); //create object by calling the constructor
    yourNumber = new TelephoneNumber(212, 217, 2487); //create by parameteric
    // constructor

    myNumber.set(212, 222, 3333); //use a setter method

    System.out.println("My " + myNumber.toString());
    System.out.println("Your " + yourNumber.toString());

    area = 0; exchange=0;line=0;
    myNumber.get(area, exchange, line);
}
```
output

In TelephoneNumber() constructor
In TelephoneNumber(int area, int Exchange, int line) constructor
My TelephoneNumber = 212 - 222 - 3333
Your TelephoneNumber = 212 - 222 - 2487
child class

```java
package Lecture3;

/**
 * @author logan.poelman
 */
public class InternationalPhoneNumber extends TelephoneNumber {
    /** Creates a new instance of InternationalPhoneNumber */
    public InternationalPhoneNumber()
    {
        System.out.println("In InternationalPhoneNumber() constructor");
    }
}
```
testing the
InternationalTelephoneNumber class

class InternationalTelephoneNumber {

    private int areaCode;
    private int exchange;
    private int number;

    public int getAreaCode() {
        return areaCode;
    }

    public void setAreaCode(int areaCode) {
        this.areaCode = areaCode;
    }

    public int getExchange() {
        return exchange;
    }

    public void setExchange(int exchange) {
        this.exchange = exchange;
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String toString() {
        return areaCode + exchange + number;
    }

    public static void main(String[] args) {
        InternationalTelephoneNumber myInternationalNumber = new InternationalTelephoneNumber();
        myInternationalNumber.set(212, 222, 4444); //use a setter method from the base class
        System.out.println("myInternational " + myInternationalNumber.toString());
    }
}
output

In TelephoneNumber() constructor
In InternationalTelephoneNumber() constructor
myInternational TelephoneNumber = 212 - 222 - 4444
Child and Parent Classes

• Child classes *silently* call the default constructor of their parent class as the first thing that happens in the child class constructor.

• Child can “see” *public*, *package scope* and *protected* not *private* in the parent

• Child can use methods defined in the parent even if not defined in the child
Overloading vs. Overriding

• Both are polymorphism in different ways
• Methods can be both
• Variables can only be overridden
• **Overloading** means multiple methods with the same name in the **same class** but having different method “signatures”.
• **Overriding** means a child has a variable or method of the same name (and signature) as the parent.
public class OverloadingVsOverridingDemo extends InternationalTelephoneNumber {

    private int area=-100;  //this is the same name as a variable in TelephoneNumber
    //this is overriding
    private int one,two,three;

    /** Creates a new instance of OverloadingVsOveridingDemo */
    public OverloadingVsOveridingDemo()
    {
        System.out.println("In OverloadingVsOveridingDemo() constructor");
    }

    public void set(int firstInt, int secondInt, int thirdInt)
    {//this overloads the method in the parent of the parent - TelephoneNumber
        System.out.println("\nIn the set() of OverloadingVsOveridingDemo()");
        System.out.println("firstInt = " + Integer.toString(firstInt));
        System.out.println("secondInt = " + Integer.toString(secondInt));
        System.out.println("thirdInt = " + Integer.toString(firstInt));
        System.out.println("total = " +Integer.toString(firstInt+secondInt+thirdInt));
        one=firstInt;
        two=secondInt;
        three=thirdInt;
    }

    public void set(int newValueForOne) //this is an overloaded method name "set"
    {one=newValueForOne;}

    public int getArea()
    {
        System.out.println("\nIn getArea() OverloadingVsOveridingDemo()");
        System.out.println("area = " + Integer.toString(area) + " from class: " + this.getClass().toString() );
        System.out.println("super.area = " + Integer.toString(super.getArea()) + " from class: " + this.getClass().getSuperclass().toString() );
        return area; //returns the overridden area (this area gets returned)
    }
}
public static void main(String[] args) {
    OverloadingVsOveridingDemo myOverloadingVsOveridingDemo = new
    OverloadingVsOveridingDemo();
    TelephoneNumber myTelephoneNumber;

    myOverloadingVsOveridingDemo.set( 333, 444, 555);

    //two reference to the same object
    myTelephoneNumber = myOverloadingVsOveridingDemo;

    myTelephoneNumber.set( 333, 444, 555);

    System.out.println( "\ngetArea() return = " + Integer.toString(
    myOverloadingVsOveridingDemo.getArea() ) ) ;
}
}
a diagram of JVM “memory” for the example (not UML)
Exception & Errors

• Thrown in unexpected errors
• Jumps from the point of error to the nearest matching `catch()`{}
• If no matching catch clause jumps to caller and looks for a match (bubbling up)
• If not catch-ed will cause program to terminate
public class JavaExceptionDemo
{
    /** this is a class that can't be instantiated because the constructor is private */
    private JavaExceptionDemo()
    {
    }

    static int divide(int firstNumber, int secondNumber)
    {
        System.out.println("in divide( " + Integer.toString(firstNumber) + " , " +
                  Integer.toString(secondNumber) + " )");
        return firstNumber/secondNumber;
    }

    static int multiply(int firstNumber, int secondNumber)
    {
        System.out.println("in multiply( " + Integer.toString(firstNumber) + " , " +
                  Integer.toString(secondNumber) + " )");
        return firstNumber*secondNumber;
    }
}
public static void main(String[] args)
{
    //notice we did not create an object here!!
    try{
        System.out.println("JavaExceptionDemo 1\n");
        System.out.println("multiply() = ");
        System.out.println( Integer.toString( multiply(100,200)));
        System.out.println("multiply() = ");
        //this should throw an exception
        System.out.println( Integer.toString( multiply(0x80000000,0x80000000)));
        System.out.println("You should never get to here"); //never gets executed
    }
    catch(Exception e)
    {
        System.out.println("EXCEPTION - The exception thrown was: " + e.getMessage());
    }
    finally
    {
        System.out.println("FINALLY - This will always get executed even if an exception is thrown");
    }
    try{
        System.out.println("\n\nJavaExceptionDemo 2\n");
        System.out.println("divide() = ");
        System.out.println( Integer.toString( divide(100,200)));
        System.out.println("divide() = ");
        //will throw an exception
        System.out.println( Integer.toString( divide(0,0)));
        System.out.println("You should never get to here"); //never gets executed
    }
    catch(Exception e)
    {
        System.out.println("EXCEPTION - The exception thrown was: " + e.getMessage());
    }
    finally
    {
        System.out.println("FINALLY - This will always get executed even if an exception is thrown");
    }
}
output

JavaExceptionDemo 1
multiply() =
in multiply( 100 , 200 )
20000
multiply() =
in multiply( -2147483648 , -2147483648 )
0
You should never get to here
FINALLY - This will always get executed even if an exception is thrown
JavaExceptionDemo 2
divide() =
in divide( 100 , 200 )
0
divide() =
in divide( 0 , 0 )
EXCEPTION - The exception thrown was: / by zero
FINALLY - This will always get executed even if an exception is thrown
try, catch and finally

- **try** enclosed the block of code to “watch” for exceptions
- **catch** encloses a block of code to handle the exception or re-throw them. May have multiple but only one gets executed (first one matched)
- **finally** always gets executed (whether an exception has happened or not) and any exceptions that might get thrown in it get *ignored*
why use exceptions?

• avoids having to check method returns for errors, just assume the method did what it was supposed to

• supports handling unforeseen problems (low memory, devices removed, bugs like divide by zero).

• Java has no GOTOs!
multiple catch clauses

catch(FileNotFoundException e)
{
    //handle file IO exceptions here
}
catch(IOException e)  /or match this exception
{
    //handle file IO exceptions here
}
catch(Exception e)  //otherwise this matches all
{
    //all the rest of the exceptions
}
finally
{
    return someValue; //if appropriate to your method design
}
Exceptions

• You must specify any exceptions that your code throws or doesn’t catch – using the `throws` keyword
• This insures that code is written defensively
• `use printStackTrace()` to see what methods were called that lead to the exception being thrown.
• you can define your own exceptions and then throw them
• two categories of exceptions: checked and unchecked
  – checked – application level – declared in the `throws` clause of a class definition
  – unchecked – system level – your code usually can’t recover from these, you just propagate them up and the program terminates
CustomException class

package Lecture3;
import java.lang.*;
public class CustomException extends java.lang.Exception {

    /**
     * Creates a new instance of <code>MyDemoException</code> without detail message.
     */
    public CustomException()
    {
        super(); //just call the constructor of Exception
    }

    /**
     * Constructs an instance of <code>MyDemoException</code> with the specified detail message.
     * @param msg the detail message.
     */
    public CustomException(String msg)
    {
        super(msg);
    }
}
package Lecture3;
import Lecture3.CustomException;
public class TestCustomException {
    public TestCustomException()
    {
    }
    public static void ThisThrowsAnException() throws CustomException
    {
        throw new CustomException("This is a custom exception thrown by thisThrowsAnException");
    }
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Testing a custom defined exception");
            System.out.println("About to invoke: ThisThrowsAnException()");
            ThisThrowsAnException(); //should throw an exception and jump to the catch
            System.out.println("After the ThisThrowsAnException() invocation");
        }
        catch(CustomException e)
        {
            System.out.println("In the catch clause");
            System.out.println("EXCEPTION: " + e.toString());
        }
        finally
        {
            System.out.println("Good bye");
        }
    }
}
output

Testing a custom defined exception
About to invoke: ThisThrowsAnException()
In the catch clause
EXCEPTION: Lecture3.CustomException: This is a custom exception thrown by thisThrowsAnException
Good bye
Summary

• Java has Primitive (data only) and Reference Types (objects).
• Pass by value vs. pass by reference
• All classes are derived from the object base class directly or indirectly (via a parent or parent of a parent, etc.)
• Children inherit non-private methods and variables from their parents
• new & Constructors create instances of classes (objects)
• Child constructors call the default parent constructor …
• Overloading vs. Overriding
• Exceptions are objects that get thrown when a problem occurs. Catch them for robust exception handling.
• Exceptions can propagate (bubble up).