Object Oriented Programming

Administrivia and introduction

Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

HÖGSKOLAN HALMSTAD

CERES

March 25th, 2008

Resources

Teachers

- Nicolina Månsson, lectures and project supervision
- Yan Wang, exercises
- Verónica Gaspes, lectures and project supervision

Web page

www.hh.se/staff/vero/oop

Book

Object Oriented Software Development using Java (2nd ed), Xiaoping Jia.

Plan

What you get

- 8 lectures
- 8 exercises with supervision
- project specification and supervision

What you produce

- questions and solutions to exercises
- a project implementation and presentation (1/2 your grade)
- a written exam (1/2 your grade)

Progression

Programming

Primitive types, variables, program control structures, basic I/O, functions, classes etc.

Algorithms and data structures

recursion, efficient algorithms for searching & sorting etc, abstract data types and structures like trees, lists etc.

Low level programming

Register programming, memory management, implementing hardware drivers, C- /assembly programming

Progression

Larger applications

This course shows how to use object oriented techniques to program reusable modules.
You will also learn how to use well known designs for organizing the modules: Design Patterns

Bottom up

When programming in an application domain it might be useful to think bottom up: first design a library of useful classes and objects that help in many programs. In OOP we cal this a framework. The Java library includes a number of these: IO, GUI, etc.
The course will also show you the internals of some frameworks.

Classes

A Java program is a collection of classes

Uses of classes

- Define auxiliary functions to be used in the `main` method,
- Build a library of useful functions,
- Define an abstract data type,
- Define a template for creating objects.

Globals and auxiliary functions

```
class CommandInterpreter{
    private static int    size;
    private static int[]  a;
    private static int    x;
    private static java.util.Random r;
    private static java.io.BufferedReader in;

    private static void execute(Command c){ ...
    public static void main(String[] cmdLn){
       while(true)
           execute(Command.valueOf(in.readLine()));
    }
}
```


What Commands?

```
class CommandInterpreter{

   enum Command {SORT,SEARCH,RANDOM,QUIT}

   static void execute(Command c){
       switch (c){
       case SORT : ...; break;
       case SEARCH : ...; break;
       case RANDOM : ...; break;
       case QUIT : System.exit(0);
       default :
           System.out.println("No command:"+c);
       }
   }
}
```

More on enum

```
class CommandInterpreter{

   enum Command {SORT,SEARCH,RANDOM,QUIT}

   public static void main(String[] cmdLn){
      System.out.println("available commands:");
      for ( Command c : Command.values() ){
         System.out.println(c);
      }
      String line = null;
      while(true){
         line = in.readLine().toUpperCase();
         execute( Command.valueOf(line) );
      }
}
```


Classes for libraries

In the Java API

The classes `Math` and `Arrays` are libraries of useful constants and functions.

These classes define `public static` constants and functions that can be used in other classes as in

```
Math.PI
Math.sin(Math.PI/2)
Math.floor(Math.PI)


Arrays.fill(a,3)
Arrays.sort(a)
```


Classes for Abstract Data Types

Räkna med bråk!

Java provides us with primitive types for integer numbers (`int`, `long`, `short`, `char`) and real numbers (`float`, `double`). Other number classes might be needed! For example Rational numbers (bråk).

Introductory examples

$$\frac{1}{2} + \frac{3}{5} = ?$$

$$\frac{1}{2} = \frac{2}{4} = \frac{10}{20} = \frac{-2}{-4} \ldots$$

Abstract data type: rational numbers

```
public class Rational implements Comparable<Rational>{
    private int num, den;
    /*
    den > 0;
    gcd(Math.abs(num),den)==1;
    */
    ...}
```

Representation and invariants

We use fields to represent the different rational numbers. The types of Java are not enough to express what we want

- The denominator should not be zero
- We use the sign of the numerator for the sign of the rational
- Numerator and denominator should be mutually prime (no common divisors!)

Establishing the invariant

```
Rational(int n, int d) throws ArithmeticException{
  if(d==0)
    throw new ArithmeticException("0 den in rat");
  num = (d<0)?-n:n;
  den = Math.abs(d);
  simplify();
}

Rational(int n){this(n,1);}

Rational(){this(0);}
```

simplify divides numerator and denominator by their greatest common divisor.

Preserving the invariant

```
public static Rational plus (Rational a, Rational b){
   return rational(a.num*b.den+b.num*a.den,a.den*b.den);
}

public static Rational times (Rational a, Rational b){
   return rational(a.num*b.num,a.den*b.den);
}
```

What is rational and why do we need it?

Values and Equality in Java

When we declare

```
Rational a
```

place is reserved in memory for an address!

When we use the statement

```
a = new Rational(1, 2);
```

place is reserved in memory for the two integers and the address of this memory chunck is what is strored in a.

Consequence

```
a = new Rational(1, 2);
b = new Rational(1, 2);
```

a is not equal to b!

Dealing with values

In the implementation of rationals we can keep track of what values have been created so that we do not need to create them again!

```
private static java.util.Map<String,Rational> values
          new java.util.HashMap<String,Rational>();
public static Rational rational(int n, int d){
  Rational r = new Rational(n,d);
  Rational inValues = values.get(r.toString());
    if(inValues == null){
        values.put(r.toString(),r);
        return r;
    }
    else return inValues;
}
```

Remember to make the constructors private!

Classes as templates for objects

- Rational numbers are created using the constructors and there are no operations that can change the value of a rational number! We think of rational numbers as mathematical values.
- Rational numbers are examples of inmutable objects.
- Classes can also be used to define so called mutable objects
- Think of defining a class that captures the notion of a ball:

having a position, a size and a color.

- We can very well think of a ball that moves, changes position.

Balls

```
import java.awt.*;
class Ball{
    private int centerX, centerY, radius;
    private Color color;

    public Ball(int x, int y, int r, Color c){
        centerX = x;
        centerY = y;
        radius = r;
        color = c;
    }
    public void move(){
        centerX = centerX + 10;
        centerY = centerY + 10;
    }
}
```


Mutators and equality

State

We say that the coordinates of the center, the radius and the color are the state of a ball.

Mutators

Some methods in the class can be used to change the state of an object, they are called mutators. Methods that deal with a particular instance are called instance methods and they are the ones that are not startic.

Equality

For this mutable objects (like Balls) it is important that equality doesn't refer to the state, we do not want to think of a ball as being another ball because it has moved!