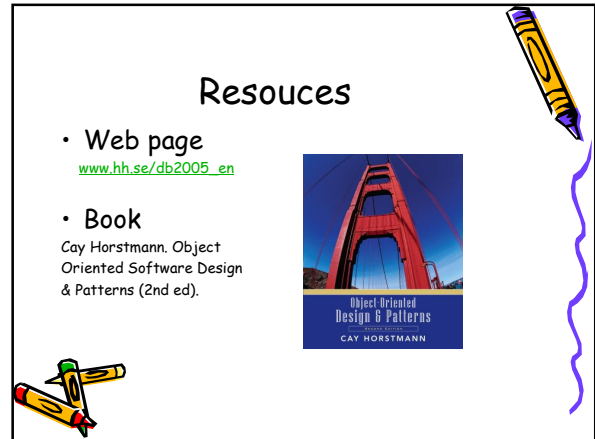


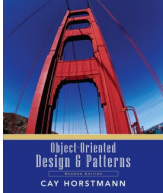

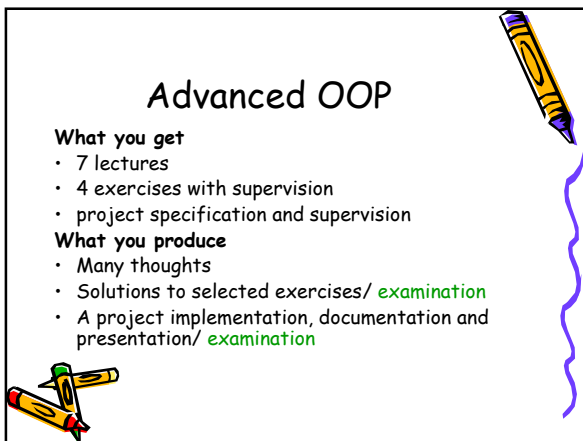
## Advanced Object Oriented Programming

Administrative and introduction



## Resources

- Web page  
[www.hh.se/db2005\\_en](http://www.hh.se/db2005_en)
- Book  
Cay Horstmann. Object Oriented Software Design & Patterns (2nd ed).


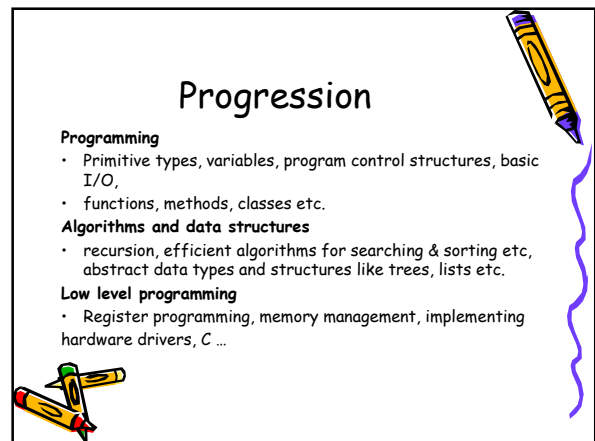
## Advanced OOP

**What you get**

- 7 lectures
- 4 exercises with supervision
- project specification and supervision

**What you produce**

- Many thoughts
- Solutions to selected exercises/ **examination**
- A project implementation, documentation and presentation/ **examination**

## Progression

**Programming**


- Primitive types, variables, program control structures, basic I/O,
- functions, methods, 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 ...



## Advanced OOP

### 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 call this a **framework**. The Java library includes a number of these: IO, GUI, etc.

## ...foundation of java...classes



## Java and classes

A Java program is a collection of classes

- **Uses of classes:**


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

Define **auxiliary functions** to be used in the main method,

```
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())); // call of execute method
    }
}
```



```

class CommandInterpreter1{
    private enum Command {SORT,SEARCH,RANDOM,QUIT}
    private static void execute( Command c){
    switch (c){
        case SORT : // do something ;
            break;
        case SEARCH : // do something ;
            break;
        case RANDOM : // do something ;
            break;
        case QUIT : System.exit(0);
        default :
            System.out.println("No command-"+c);
    }
}
}

```





## Enum- values(), valueOf()

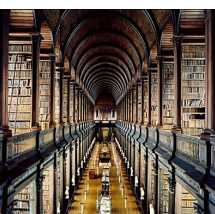
```

public static void main(String[] cmdLn) {
    for (Command c : Command.values() ){
        System.out.println(c);
    }
    String line = null;
    while(true){
        try{
            line = in.readLine();
            execute (Command.valueOf( line.toUpperCase() ));
        } catch (IllegalArgumentException e) {
            System.out.println("Sorry! no Command " + line);
        }
    }
}
}

```

## Classes as 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:

```

Math.PI
Math.sin(Math.PI/2)
Math.floor(Math.PI)
Arrays.fill(a,3)
Arrays.sort(a)
JOptionPane.showMessageDialog()

```

## Static methods



- Don't operate on objects
- Example: `Math.sqrt`
- Example: *factory method*

```

public static Greeter getRandomInstance()
{
    if (generator.nextBoolean()) // note: generator is static field
        return new Greeter("Mars");
    else
        return new Greeter("Venus");
}

```

- Invoke through class:  
`Greeter g = Greeter.getRandomInstance();`
- Static fields and methods should be rare in OO programs

## Classes for Abstract Data Types

- Same exempel: Queue, Process, Number
- 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).

$$\frac{1}{2} + \frac{3}{4} = ?$$



## ADT- rational numbers

```
public class Rational implements Comparable<Rational> {  
    private int num, den;  
  
    // num  
    den > 0;  
    god (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 denom
- Numerator and denominator should not have 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 (keep it) 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);  
}
```

## Values and equality

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 is stored in a.

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



## Values and equality

```
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;
}
```



## Immutable 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 **immutable** objects.

Rules:

- Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
- Make all fields **final** and **private**, make all methods **final** (avoid overriding). Make the constructor **private**, let factory method create objects.
- If the instance fields include references to mutable objects, don't allow those objects to be changed or allow methods to return references (create copies).



## Classes as templates for objects

- Classes are templates, describe how objects are going to be like...properties, behaviour and constructor
- Object are instance of class (concrete, real)

```
- name
- age
- other
-----
- kan_sing()
- kan_talk()
```



```
name: Bobby
age: 12
-----
kan_sing()
kan_talk()
```



```
Name: Koky
age: 3
-----
kan_sing()
kan_talk()
```



## Classes as templates for 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.



....and we can very well think of a ball that moves, change position.



## Classes as templates for objects

```
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;
  }
}
```



## Classes as templates for objects

- **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.
- **Accessors**,  
Methods that read and return information from the object
- **Equality problem**  
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!

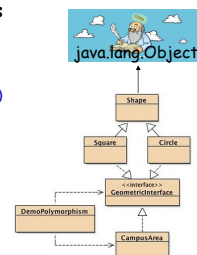


## Class Object at the top

**Object: The Cosmic Superclass**

All classes extend Object

- Most useful methods:
  - String toString()
  - boolean equals(Object otherObject)
  - Object clone()
  - int hashCode()



## toString() method

Returns a string representation of the object  
• Useful for debugging

• **Overriding the toString Method**

```
public class Employee
{
    public String toString()
    {
        return getClass().getName()
            + "[name=" + name + ", salary=" + salary + "]";
    }
}
```

Class c = e.getClass();  
System.out.println(c.getName());

Typical string:  
Employee[name=Harry Hacker,salary=35000]

## toString() method

• **Overriding toString in Subclass**

```
public class Manager extends Employee
{
    public String toString()
    {
        return super.toString() + "[department=" + department + "];" ...
    }
}
```

Typical string  
Manager[name=Dolly Dollar,salary=100000][department=Finance]

! Note that superclass reports actual class name : Manager

## The equals() Method

• equals tests for equal *contents*  
• == tests for equal *location*  
Used in many standard library methods

```
public int arraySearch(Object elem)
{
    if (elem != null)
    {
        for (int i = 0; i < size; i++)
            if (elementData[i] == elem)
                return i;
    }
    return -1;
}
```

## The equals() Method

• **Overriding the equals Method**

• Notion of equality depends on class  
• Common definition: compare all fields

```
public class Employee
{
    // fields name and salary
    public boolean equals(Object otherObject)
    {
        Employee other = (Employee) otherObject;
        return name.equals(other.name)
            && salary == other.salary;
    }
}
```

Type casting from  
Object to Employee

## The Object.equals Method

Object.equals tests for identity:

```
public class Object
{
    ...
    public boolean equals(Object obj)
    {
        return this == obj;
    }
    ...
}
```

- Override equals if you don't want to inherit that behavior



## The Perfect equals Method

- Start with these three tests:

```
public boolean equals(Object otherObject)
{
    if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    ...
}
```

- 1) Add test for null:

```
if (otherObject == null) return false
What happens if otherObject not an Employee
```

- 2) Test for class equality

```
if (getClass() != otherObject.getClass()) return false;
```



## Hashing

- hashCode method used in HashMap, HashSet
- Computes an int from an object
- Example: hash code of String

```
int h = 0;
for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
```

- Hash code of "eat" is 100184
- Hash code of "tea" is 114704



## Hashing for objects

- Must be compatible with equals: Be sure!

```
if x.equals(y),
then
x.hashCode() == y.hashCode()
```

Object.hashCode hashes memory address,

```
public class Employee {
    public int hashCode()
    {
        return name.hashCode()
            + new Double(salary).hashCode();
    }
    ...
}
```



## Shallow and Deep Copy

- Clone to make copy
- `Employee cloned = (Employee)e.clone();`
- `Object.clone` makes new object and copies all fields
- `Object.clone` is protected
- Subclass *must* redefine clone to be public



## The clone Method

### • Shallow Cloning

```
public class Employee implements Cloneable
{
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}
```



## The clone Method

- **Deep Cloning**
- Why doesn't clone make a deep copy?
- Not a problem for immutable fields
- You must clone mutable fields

```
public class Employee implements Cloneable
{
    // Date hireDate= ...
    public Object clone()
    {
        try
        {
            Employee cloned = (Employee) super.clone();
            cloned.hireDate = (Date)hiredate.clone();
            return cloned;
        }
        catch (CloneNotSupportedException e)
        {
            return null; }
    }
}
```

Clone all  
immutable  
fields



## Java documentation

- **Documentation Comments**
- Delimited by `/** ... */`
- First sentence = summary
- `@param` *parameter explanation*
- `@return` *explanation*
- Easy to keep documentation in sync with code
- You must document *all* classes and methods



## Java documentation

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute (@link URL). The name argument is a specific
 * that is relative to the url argument.
 * @param * This method always returns immediately, whether or not the image exists.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argumen
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url , String name)
{ try {
  return getImage(new URL(url, name));
} catch (MalformedURLException e)
{ return null; }
}
```

