Classes and Objects

Objective

- Objective: Shows how to write and use classes

Ingredients of a Class

class is a template or blueprint from which objects are created.

```
class NameOfClass
{
constructor1 // construction of object
constructor2
∘ ∘ ∘
method1 // behavior of object
method2
∘ ∘ ∘
field1 // state of object
field2
```

Outline

- Ingredients of a class
  - Instance fields
  - Initialization and constructors
  - Methods
  - Class modifiers
- Packages: How classes fit together

Instance Fields

```
class Employee
{ ...
private String name;
private double salary;
private Date hireDay;
}
```

- Various types of fields
  - Classification according to data type
    - A field can be of any primitive type or an object
  - Classification according to accessibility
    - public, default, protected, private

Instance Fields

- Access modifiers:
  - private: visible only within this class

```
class Employee
{ ...
public void raiseSalary(double byPercent)
{ double raise = salary * byPercent / 100;
salary += raise; }

private double salary;
```

  - Default (no modifier): visible in package
  - protected: visible in package and subclasses
  - public: visible everywhere

Instance Fields

- It is never a good idea to have public instance fields because everyone can modify it. Normally, we want to make fields private. OOP principle.

```
class Employee
{ ...
// accessor method
public double getSalary()
{ return salary;}
// mutator method
public void raiseSalary(double byPercent)
{ double raise = salary * byPercent / 100;
salary += raise; }

// private field
private double salary;
```

Instance Fields

- **Static** fields belong to class, not object

```
class Employee
{ ...
public Employee()
{
id = nextID;
nextID++;
}
private int id;
public static int nextID =1; // public for convenience
// of example
}
Employee harry = new Employee(); // harry.id =1
Employee jack = new Employee(); // jack.id = 2
```

Instance Fields

- What if static is removed?

```
class Employee
{ ...
public Employee()
{
id = nextID;
nextID++;
}
private int id;
private int nextID =1;
}

Employee harry = new Employee();
Employee jack = new Employee();

They both have id 1.
```

Instance Fields

- Constants:
  - Declared with **static final.**
  - Initialized at declaration and cannot be modified.
- Example:

```
Public class Math
{ …
public static final double PI = 3.141592;
…
}//Called with Math.PI
```

- Notes:
  - Static fields are rare, static constants are more common.

Outline

- Ingredients of a class
  - Instance fields
  - Initialization and constructors
  - Methods
  - Class modifiers
- Packages: How classes fit together

Initialization of Instance Fields

- Several ways:

1. Explicit initialization
2. Initialization block
3. Constructors

Initialization of Instance Fields

- Explicit initialization: initialization at declaration.

```
private double salary = 0.0;
private String name = “”;
```

- Initialization value does not have to be a constant value.

```
class Employee
{ …
…
private int id = assignId();
private static int nextId=1;

static int assignId()
{ int r = nextId;
nextId++;
```

Initialization of Instance Fields

- Initialization block:
  - Class declaration can contain arbitrary blocks of codes.

```
class Employee
{ ...
private int id;
private static int nextId=1;
// object initialization block
{ id = nextId;
nextId++;
}
```

Initialization of Instance Fields

- Initialization by constructors

```
class Employee
{
public Employee(String n, double s,
int year, int month, int day)
{
name = n;
salary = s;
GregorianCalendar calendar
= new GregorianCalendar(year, month - 1, day);
// GregorianCalendar uses 0 for January
hireDay = calendar.getTime();
}

private String name;
private double salary;
```

Initialization of Instance Fields

- What happens when a constructor is called

- Object created
- All data fields initialized to their default value (0, false, null)
- Field initializers and initialization blocks are executed
- Body of the constructor is executed
  - Note that a constructor might call another constructor at line 1.

Initialization of Instance Fields

- Default constructor:
  - Constructor with no parameters

```
class Employee
{ ...
public Employee()
{
id = nextID;
nextID++;
}
}
```

Initialization of Instance Fields

- If programmer provides no constructors, Java provides an a default constructor that set all fields to default values
  - Numeric fields, 0
  - Boolean fields, false
  - Object variables, null

- Note: If a programmer supplies at least one constructor but does not supply a default constructor, it is illegal to call the default constructor.

  In our example, the following should be wrong if we have only the

Constructors

- Constructors define initial state of objects

```
class Employee
{
public Employee(String n, double s,
int year, int month, int day)
{
name = n;
salary = s;
GregorianCalendar calendar
= new GregorianCalendar(year, month - 1, day);
// GregorianCalendar uses 0 for January
hireDay = calendar.getTime();
}
private String name; private double salary;
private Date hireDay; }
```

```
Employee hacker = new Employee("Harry Hacker", 35000, 1989,10,1);
```

Constructors

- A class can have one or more constructors.
- A constructor
  - Has the same name as the class
  - May take zero, one, or more parameters
  - Has no return value
  - Almost always public, although can be others
  - Always called with the **new** operator

Using this in Constructors

- **this** refers to the current object.
- More meaningful parameter names for constructors

```
public Employee(String name, double salary, int year, int month, int day) {
this.name = name;
this.salary = salary;

…
}
```

- Can also be used in other methods

```
public void setName( String name) {this.name= name;}
```

- No copy constructor in Java. To copy objects, use the **clone** method, which will be

Calling another constructor

- Can call another constructor at line 1:

```
class Employee
{
public Employee(String name, double salary){...}
public Employee(double salary)
{
this(``Employee #'' + nextID, salary );
nextID++;
}
}
```

Object Creation

- Must use **new** to create an object instance

```
Employee hacker = new Employee("Harry Hacker", 35000, 1989,10,1);
```

This is illegal:

```
Employee number007(“Bond”, 1000, 2002, 2, 7);
```

Object Destruction

- No delete operator. Objects are destroyed automatically by garbage collector
- Garbage collector destroy objects not referenced periodically
  - To force garbage collection, call System.gc();
- To reclaim non-memory resources (IO connection), add a **finalize** method to your class.
  - This method is called usually before garbage collect sweep away your object. But you never know when.

ConstructorTest.java

- A better way is to add a **dispose** method to your class and call it manually in

Outline

- Ingredients of a class
  - Instance fields
  - Initialization and constructors
  - Methods
  - Class modifiers
- Packages: How classes fit together

Methods

- Three key characteristics of objects
  - Identity
  - State
    - Values of fields
  - Behavior
    - What we can do with them?

- Methods determine behavior of objects

Methods

```
class Employee
{
public String getName()
{...}
public double getSalary()
{...}

public Date getHireDay()
{...}

public void raiseSalary(double byPercent)
{...}
}
```

Methods

- Plan:
  - Types of methods
  - Parameters of methods (pass by value)
  - Function overloading

Methods

- Types of methods
  - Classification according to functionality
    - Accessor, mutator, factory
  - Classification according to accessibility
    - public, protected, default, private
  - Classification according to host
    - Static vs non-static

Types of Methods

- Accessor methods:

```
public String getName()
{
return name;
}
```

- Mutator methods:

```
Public void setSalary(double newSalary)
{
```

Types of Methods

Factory methods

- Produce objects of the class: usually static

  NumberFormat.getNumberInstance() // for numbers

  NumberFormat.getCurrencyInstance()// for currency values

  NumberFormat.getPercentInstance()//

- Why useful? (We already have constructors)
  - More flexibility in name
    - Constructors must have the same name as class. But sometimes other names make sense.
  - Factory methods can generate object of subclass, but constructors cannot.

Methods

- Types of methods
  - Classification according to functionality
    - Accessor, mutator, factory
  - Classification according to accessibility
    - public, protected, default, private
  - Classification according to host
    - Static vs non-static

Accessibility of Methods

- **public**: visible everywhere
- **protected**: visible in package and in subclasses
- Default (no modifier): visible in package
- **private**: visible only inside class (more on this later)

- A method can access fields and methods that are visible to it.
  - Public method and fields of any class
  - Protected fields and methods of superclasses and classes in the same package
  - Fields and methods without modifiers of classes in the same packages
  - Private fields and methods of the same class.

Accessibility of Methods

- A method can access the private fields of all objects of its class

```
class Employee
{ ...
public boolean equals(Employee another)
{
retun name.equals(another.name);
}
private String name;
}
```

Public method and private field

- Bad idea for a method to return an object

```
class Employee
{
public Date getHireDay() { return hireDay;}

private String name;
private double salary;
private Date hireDay;
}
```

Other classes can get the object and modify, although it is supposed to be private to Employee.

Better solution:

```
public Date getHireDay() { return hireDay.clone();}
```

Methods

- Types of methods
  - Classification according to functionality
    - Accessor, mutator, factory
  - Classification according to accessibility
    - public, protected, default, private
  - Classification according to host
    - Static vs non-static

Static Methods

- Declared with modifier **static**.
- It belongs to class rather than any individual object. Sometimes called class method.
- Usage: **className.staticMethod()** NOT **objectName.staticMethod()**

```
class Employee
{ public static int getNumOfEmployees()
{
return numOfEmpolyees;
}
private static int numOfEmployees = 0;
…
}
Employee.getNumOfEmployee(); // ok
Harry.getNumOfEmployee(); // not this one
```

Static methods

- Explicit and implicit parameters:

```
class Employee
{
public void raiseSalary(double byPercent)
{...}

}
```

Explicit parameters: byPercent

Implicit parameters: this object

```
public class Math{
public static double pow(double x, double y) {...}
}
Math.pow(2, 3);
```

- Static methods do not have the implicit parameter

Static Methods

- The main method

```
class EmployeeTest
{ public static void main(String[] args)
{
}
}
```

is always static because when it is called, there are not objects yet

Static Methods

- A static method cannot access non-static fields

```
class Employee
{ public static int getNumOfEmployees()
{
return id; // does not compile
// id == this.id
}

private static int numOfEmployees = 0;
private int id = 0;
}
```

Methods

- Plan:
  - Types of methods
  - Parameters of methods (pass by value)
  - Function overloading

Parameters of Method

- Parameter (argument) syntax same as in C

- Parameters are all passed by value, not by reference
  - Value of parameter copied in function call

```
public static void doubleValue( double x)
{ x = 2 * x; }

double A = 1.0;
doubleValue( A );
// A is still 1.0
```

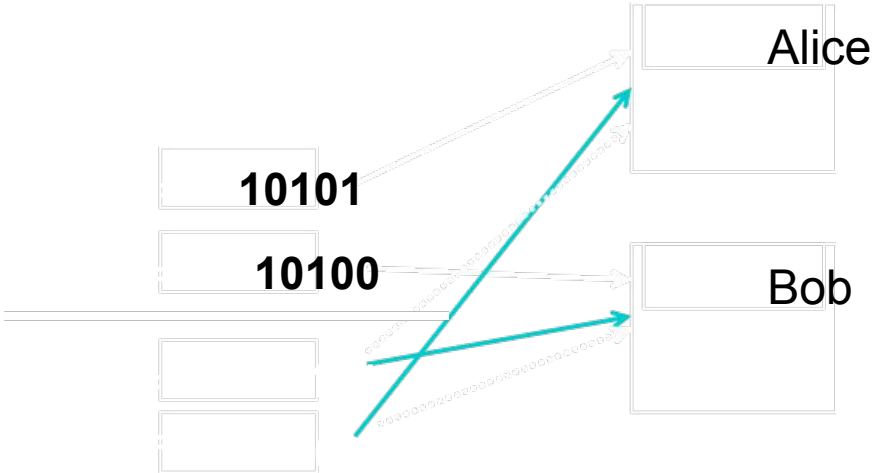
A= 2.0

Parameters of Method

- When parameters are object references
  - Parameter values, i.e. object references are copied

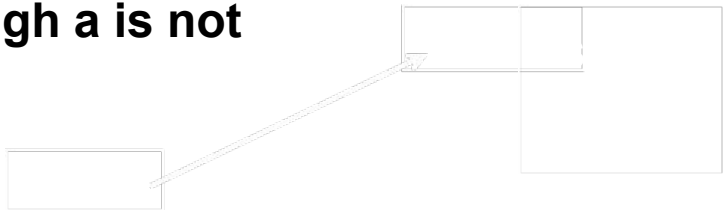
```
public void swap( Employee x, Employee y)
{ Employee tmp = x;
x=y;
y=tmp;
}

Employee A = new Employee(“Alice”,..
Employee B = new Employee(“Bob”,..
Swap(A, B
```



Parameters of Method

```
// a function that modify content of object,
// but not object reference
    void bonus(Employee A, double x)
    {
    A.raiseSalary(x);//a.salary modified although a is not
    }
    //ParamTest.java
```



Command-Line arguments
Parameters of the main Method

```
public static void main(String args[])
    {
        for (int i=0; i<args.length; i++)
        System.out.print(args[i]+“ ”);
        System.out.print(“\n”);
    }

    // note that the first element args[0] is not
    // the name of the class, but the first
    // argument

    //CommandLine.java
```

Methods

- Plan:
  - Types of methods
  - Parameters of methods (pass by value)
  - Function overloading

Function Overloading

- Can re-use names for functions with different parameter types

```
void sort (int[] array);
void sort (double[] array);
```

- Can have different numbers of arguments

```
void indexof (char ch);
void indexof (String s, int startPosition);
```

- Cannot overload solely on return type

```
void sort (int[] array);
boolean sort (int[] array); // not ok
```

Resolution of Overloading

- Compiler finds best match
  - Prefers exact type match over all others
  - Finds “closest” approximation
    - Only considers widening conversions, not narrowing

- Process is called “resolution”

```
void binky (int i, int j);
void binky (double d, double e);

binky(10, 8) //will use (int, int)
binky(3.5, 4) //will use (double, double)
```

Outline

- Ingredients of a class
  - Instance fields
  - Initialization and constructors
  - Methods
  - Class modifiers
- Packages: How classes fit together

Class Modifiers

- **public**: visible everywhere

```
public class EmployeeTest { ...
}
```

- Default (no modifier): visible in package

```
class Employee { ...
}
```

- **private**: only for inner classes, visible in the outer class (more on this later)

```
public class Tree { ...
private class TreeNode{...}
```

Outline

- Ingredients of a class
  - Instance fields
  - Initialization and constructors
  - Methods
  - Class modifiers

- Packages: How classes fit together

Packages

Plan

- What are packages
- Creating packages
- Using packages

Packages

- A package consists of a collection of classes and interfaces
- Information about packages in JSDK 1.4.1 can be found http://java.sun.com/j2se/1.4.1/docs/api/index.html
- Example:
  - Package java.lang consists of the following classes
  - Boolean Byte Character Class ClassLoader Compiler Double Float Integer Long Math Number Object SecurityManager Short StackTraceElement StrictMath String StringBuffer System Thread

Packages

- Packages are convenient for organizing your work

- Guarantee uniqueness of class names
  - Complete name of class: package name + class name
  - Avoids name conflict. Example
    - java.sql.Date
    - java.util.Date

- Packages are organized hierarchically.
  - Example
    - java.security
    - java.security.acl java.security.cert java.security.interfaces java.security.spec

Packages

- Plan
  - What are packages
  - Creating packages
  - Using packages

Creating Packages

- To add to class to a package, say foo
  - Begin the class with the line

    package foo;
  - This way we can add as many classes to foo as you wish

- Where should one keep the class files in the foo package?
  - In a directory name foo :
    - .../foo/{first.class, second.class, ...}

- Subpackages and subdirectories must match
  - All classes of foo.bar must be placed under .../foo/bar/

Creating Packages

- Classes that do not begin with “package …” belongs to the default package:
  - The package located at the current directory, which has no name
  - Consider a class under …/foo/bar/
    - If it starts with “package foo.bar ”, it belongs to the package “foo.bar”
    - Else it belong to the default package

Packages

- Plan
  - What are packages
  - Creating packages
  - Using packages

Using Packages

- Use full name for a class: packageName.className
  java.util.Date today = new java.util.Date();

- Use **import** so as to to use shorthand reference
  import java.util.Date;
  Date today = new Date();

Differ from “include” directive in C++. Merely a convenience.

- Can import all classes in a package with wildcard
  - import java.util.*;
  - Makes everything in the java.util package accessible by shorthand name: Date, Hashtable, etc.

- Everything in java.lang already available by short name, no import

Using packages

- Resolving Name Conflict
  - Both java.util and java.sql contain a Date class

```
import java.util.*;
import java.sql.*;
Date today; //ERROR--java.util.Date or java.sql.Date?
```

  - Solution:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

  - What if we need both? Use full name

Using packages

- Informing java compiler and JVM location of packages
  - Set the class path environment variable:
    - On UNIX/Linux: Add a line such as the following to **.cshrc**

      setenv CLASSPATH /home/user/classDir1:/home/user/classDir2:.

      - The separator ":" allows you to indicate several base directories where packages are located.

      - Java compiler and JVM will search for packages under both of the following two directories
        - /home/user/classDir1/
        - /home/user/classDir2/

UsingPackages

Set the CLASSPATH environment variable:

- On Windows 95/98: Add a line such as the following to the autoexec.bat file

  SET CLASSPATH=c:\user\classDir1;\user\classDir2;.

  - **Now, the separator is “;”.**

- On Windows NT/2000/XP: Do the above from control panel

Using Packages

- Example:

setenv CLASSPATH /homes/lzhang/DOS/teach/201/code/:/appl/Web/HomePages/faculty/lzhang/teach/201/codes/servlet/jswdk/lib/servlet.jar:/appl/Web/HomePages/faculty/lzhang/teach/201/codes/servlet/jswdk/webserver.jar:.

- jar files: archive files that contain packages. Will discuss later.