

Packages and Information Hiding

- Packages as a structuring concept
- Information hiding through access specifiers

OOAD 1999/2000

Claudia Niederée, Joachim W. Schmidt
Software Systems Institute

c.niederee@tu-harburg.de <http://www.sts.tu-harburg.de>

Package: the Library Unit



- Managing “name spaces”
 - Class members are already hidden inside class
 - Class names could clash
 - Need completely unique name
- Packages
 - organize classes into libraries
 - structure name space for classes
 - restrict visibility
 - may be nested

Creating a Library of Classes

```
package mypackage;
```

```
public class Class1{ ... }
```

Class1.java

```
package mypackage.mysubpackage1;
```

```
public class Class5{ ... }
```

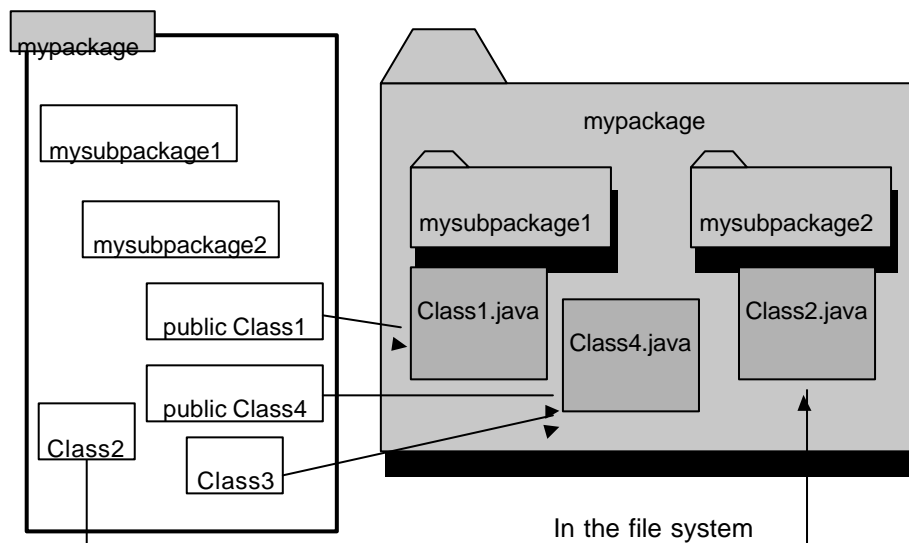
Class5.java

- **public** class is under the umbrella **mypackage**
- Client programmer must import the package

```
import mypackage.*;
```

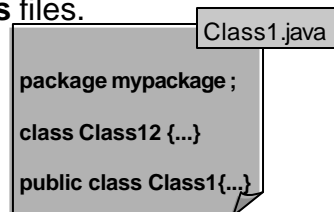
```
import java.util.Vector;
```

Organizing Your Packages



Compilation Units

- Compilation units (**.java** files)
 - Name of **.java** file == name of single **public** class
 - Other non-**public** classes are not visible outside the package.
 - Each class in file gets its own **.class** file.
 - Program is a bunch of **.class** files.



```
Class1.java
package mypackage ;
class Class12 {...}
public class Class1{...}
```

How to create a package step by step

- Create a directory having the same name as the new package:
`mypackage:> mkdir mysubpackage1`
- Class files belonging to the package have to be put into the package's directory:
`mysubpackage:> xemacs Class5.java &`
- The emerging files have to contain an adequate package-clause and a public class with the same name than the file name:

```
package mypackage.mysubpackage1;
public class Class5 { ... }
```

How to use a package step by step

- Import the package into the source file in which it should be used:
`import mypackage.mysubpackage.*;`
`public class ...`
- Before you can compile your code the package's path has to be included in the class path:

Unix:

```
mydir:> setenv CLASSPATH $CLASSPATH"/users/sts/kurs/"
```

Dos:

```
mydir:> set CLASSPATH=%CLASSPATH%;c:\users\sts\kurs\
```

- Compile your file like you always do.

CLASSPATH and Packages

- The Java interpreter
 - uses the CLASSPATH environment variable as starting point for search and
 - looks for package x.y.z in a folder on the path x/y/z.
 - CLASSPATH takes care of first part:

```
CLASSPATH=../local/jdk12:/users/sts/kurs
```

CLASSPATH and Package Programs

- Sometimes confusing:
 - When your main-method is inside a class with a package statement, you must **always** give the full package name before the program name in order to run the program:

```
mydir:> java mypackage.mysubpackage.Test
```

- To run this program from within the package's directory, your classpath has to include the root directory of the package hierarchy:

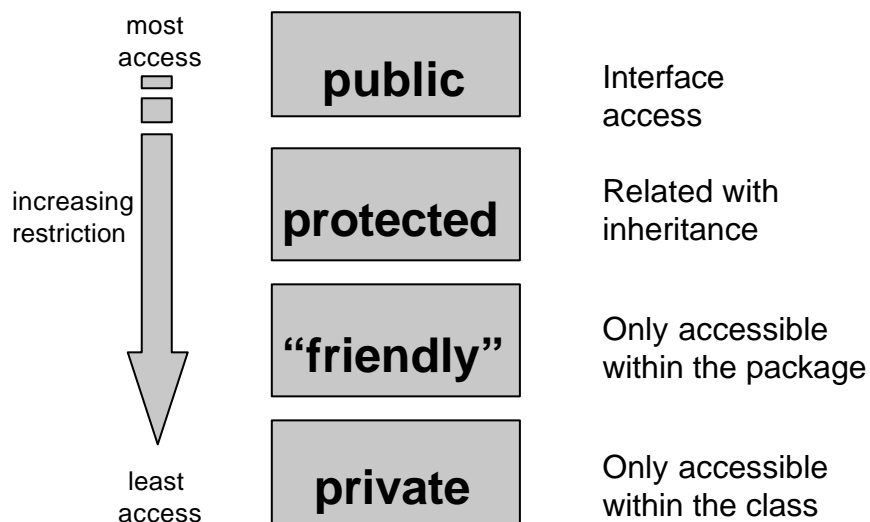
```
mydir:> setenv CLASSPATH $CLASSPATH"../.."
```

or

```
mydir:> set CLASSPATH=%CLASSPATH%;..\..\
```

```
mysubpackage:> java mypackage.mysubpackage.Test
```

Java Access Control



public : Interface Access

- Any class, in any package, has access to a class's public members.
- Declare public members only, if such access cannot produce undesirable results, if an outsider uses them.
- Signature of public members should be relatively stable.
- use the keyword `public`.

“Friendly”

- Default access, has no keyword
- public to other members of the same package, private to anyone outside the package.
- Easy interaction for related classes (that you place in the same package)
- Also referred to as “package access”

public and „friendly“

```
package food.dessert;
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void decorate() { System.out.println("Cookie
decorated"); }
}

// Separate file:
import food.dessert.*;
public class Dinner {

    public static void main(String args[]) {
        Cookie c = new Cookie();
        //! c.decorate(); // can't access
    }
}
```

protected: subclass access

- Inheritors (and the package) can access protected members
BUT
- They are then vulnerable to changes in the base-class implementation
- Class users are prevented from accessing protected members

```

public class Counter {
    protected int value;
    public int getValue() { return value; }
    public void increment() {value ++;}
    public Counter( int startValue)
        { value = startValue; }
}

public class ResetableCounter extends Counter {
    private int startValue;
    public ResetableCounter( int startValue) {
        super(startValue);
        this.startValue = startValue;
    }
    public void reset() {value = startValue; }
}

```

private: Can't Touch That!

- Most restrictive access level
- A private member is accessible only to the class in which it is defined
- For example useful for:
 - variables that contain information that if accessed by an outsider could put the object in an inconsistent state
 - methods that, if invoked by an outsider, could jeopardize the state of the object or the program in which it's running
- To declare a private member, use the keyword **private**.

private: Can't Touch That!

```
class Resource {
    private static count = 5;
    private Resource() {}
    static Resource makeAResource() {
        if (count > 0) {
            count --; return new Resource();
        } else return null;
    }
}

public class ResourceUser {
    public static void main(String args[]) {
        //! Resource r = new Resource();
        Resource r = Resource.makeAResource();
    }
}
```

Class Access

- Classes as a whole can be **public** or “friendly”
- Only one **public** class per file, usable outside the library
- All other classes “friendly,” only usable within the library