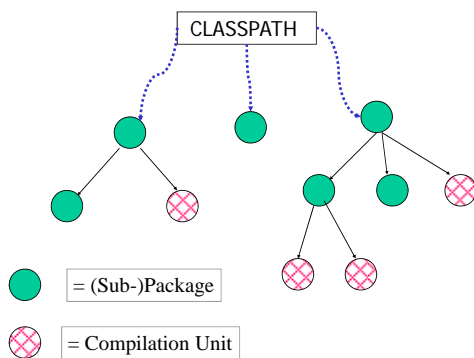


# Packages

Partitioning class name space  
Access control of names

## Organization of a Java Program

- Java Program
  - Forest of packages
    - E.g., \$PWD, java, javax, etc.
- Package (Tree)
  - Directory of sub-packages and compilation units
    - E.g., java.io, java.util, java.lang.Thread, java.applet.Applet;
- Compilation Unit (Leaf)
  - File containing class and interface definitions with at most one public class
- CLASSPATH environment variable
  - Set to full-path names of roots of trees in the forest



### package statement

- Omitted : *anonymous* package
- For \*.class byte code files to be accessible to the JVM, ensure “.” (\$PWD) is contained in the environment variable CLASSPATH.
- package Pk ;
- If a class is declared inside package Pk (or subpackage Pk.Sk), then the \*.java source files must be stored in the directory Pk (or Pk/Sk), and the CLASSPATH must contain at least ../Pk.

### import statement (for programmer convenience)

- importing a type explicitly
  - import java.util.StringTokenizer;
- importing type-on-demand
  - import java.io.\*;

### Access Control

- private : inaccessible from outside the class
- default : accessible in the package
- protected : accessible in the subclasses and the package
- public : no restriction on accessibility

|                                | private | default | protected | public |
|--------------------------------|---------|---------|-----------|--------|
| same class                     | Y       | Y       | Y         | Y      |
| same package subclass          | N       | Y       | Y         | Y      |
| same package non-subclass      | N       | Y       | Y         | Y      |
| different package subclass     | N       | N       | Y         | Y      |
| different package non-subclass | N       | N       | N         | Y      |

### Access Control : C and S in different packages

```
class C {
    public int x;
    protected int y;
    private int z;
    int p(C c) {
        return c.x + c.y +
            c.z ;
    }
    int q(S s) {
        return s.x + s.y
        + ((C) s).y + ((C) s).z
        // + s.z
        // + s.j + s.k ;
    }
}
```

```
class S extends C {
    public int i;
    protected int j;
    private int k;
    int p(C c) {
        return c.x ;
        // + c.y + c.z ;
    }
    int q(S s) {
        return s.x + s.y +
        // s.z +
        s.i + s.j +
        s.k ;
    }
}
```

### Notes

- Public, protected and private fields of a direct instance (e.g., C.x, C.y, C.z) of a class are accessible in the class (C's text), while the private fields of a subclass instance (e.g., S.z) are accessible in the class (C's text) only via a static cast (e.g., (C) s.z).
- **The inherited protected field due to parent class (C) of a direct instance of a subclass (e.g., s.y) is available in subclass (S's text).**
  - Note that the protected field of a direct class instance (e.g., C.y) is not available in the subclass (S's text) and the protected field of a subclass instance is not available in the parent class (C's text) (e.g., s.j).

### Other Constraints

- *Access modifier* of a hiding/overriding method must provide *at least as much access as* the hidden/overridden method.
  - **Otherwise, access barrier beaten by casting.**

```
class C {
    public void p() {}
}
class S extends C {
    private void p() {}
}
C x = new S();
x.p();
```

*Illegal Code*

### Hiding and Overriding : Static and Dynamic Binding

```
class C {
    int a = 84;
    static int q() {}
    int p() {...}
}
class S extends C {
    int a = 77;
    static int q() {
        super.q() + 1;
    }
    int p() {...
        super.p();
    }
}
```

```
S x = new S();
C y = x;

(x.a == 77)
(y.a == 84)
(((C) x).a == 84)

(x.p() == y.p())

(x.q() != y.q())
(((C) x).q() == y.q())
```

### Notes

- Static methods, static fields and instance fields are *statically* bound, while instance methods are *dynamically* bound.
- Static methods, static fields, and instance fields can be *hidden*, while instance methods can be *overridden*.
- Hidden members of a subclass instance can be accessed outside the class (text) using *static cast* and within the subclass (text) using **super** prefix.

### Defining a Method with the Same Signature as a Superclass's Method

|                          | Superclass Instance Method     | Superclass Static Method       |
|--------------------------|--------------------------------|--------------------------------|
| Subclass Instance Method | Overrides                      | Generates a compile-time error |
| Subclass Static Method   | Generates a compile-time error | Hides                          |

## Accessibility and Overriding

A method can be *overridden* in a subclass only if the method in the superclass is accessible.

If a method in the superclass is *not accessible* then method in the subclass *does not override* it *even if they have the same signature*.

```
package P1;

public abstract class AbstractBase {

    private void pri() {    System.out.print("AbstractBase.pri()"); }

    void pac() {    System.out("AbstractBase.pac()"); }

    protected void pro() {    System.out("AbstractBase.pro()"); }

    public void pub() {    System.out("AbstractBase.pub()"); }

    public final void show() {

        pri();

        pac();

        pro();

        pub();

    }
}
```

```
package P2; // Different package from AbstractBase
import P1.AbstractBase;
public class Concrete1 extends AbstractBase {
    public void pri() { print("Concrete1.pri()"); }
    public void pac() { print("Concrete1.pac()"); }
    public void pro() { print("Concrete1.pro()"); }
    public void pub() { print("Concrete1.pub()"); }
}
new Concrete1().show();
// only protected and public can be overridden
OUTPUT:
    AbstractBase.pri()
    AbstractBase.pac()
    Concrete1.pro()
    Concrete1.pub()
```

```
package P1; // Same package as AbstractBase
import P2.Concrete1;
public class Concrete2 extends Concrete1 {
    public void pri() { print("Concrete2.pri()"); }
    public void pac() { print("Concrete2.pac()"); }
    public void pro() { print("Concrete2.pro()"); }
    public void pub() { print("Concrete2.pub()"); }
}
new Concrete2().show();
// default (package), protected and public can be overridden
OUTPUT:
    AbstractBase.pri()
    Concrete2.pac()
    Concrete2.pro()
    Concrete2.pub()
```

```

package P3; // Different package from AbstractBase
import P1.Concrete2;
public class Concrete3 extends Concrete2 {
    public void pri() { print("Concrete3.pri()"); }
    public void pac() { print("Concrete3.pac()"); }
    public void pro() { print("Concrete3.pro()"); }
    public void pub() { print("Concrete3.pub()"); }
}
new Concrete3().show();
// Concrete3.pac() overrides Concrete2.pac()
// which in turn overrides AbstractBase.pac()
OUTPUT:
    AbstractBase.pri()
    Concrete3.pac()
    Concrete3.pro()
    Concrete3.pub()

```

## Java Development Kit

- `java.lang` (contains `Threads`)
- `java.io` (contains `StreamTokenizer`)
- `java.awt` (Abstract Windowing Toolkit)
- `java.net` (Support for TCP/IP, HTTP)
- `java.applet` (To run code via Browser)
- `java.util` (Standard Utilities)
- `javac`, `java`, `javadoc`

## Scope and Lifetime

- The *scope* of an identifier declaration is *the region of text* in which the declaration is effective.
  - E.g., class scope, block scope, global, etc.
- The *lifetime* of a variable (or an object) bound to an identifier is the period during which it exists in the memory.
  - E.g., Static variables and objects allocated via `new` on *heap* have “infinite” lifetime. (garbage collection)
  - E.g., Formal parameters and local variables of a method are allocated on *stack*. (LIFO discipline)

```

Enumeration myEnumerate(final Object[] arr) {

    class E implements Enumeration {
        int count = 0;
        public boolean hasMoreElements()
            { return count < arr.length; }
        public Object nextElement()
            { return arr[count++]; }
    }
    return new E();
}

```

- *Scope* of `arr` is the body of `myEnumerate` function.
- *Lifetime* of the array instance referenced by `arr` is “infinite”. It survives as long as the `Enumeration` object returned is “alive”.