Organization of a Java Program

- Java Program
  - Forest of packages
    - E.g., SPWD, 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 “.” (SPWD) 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

<table>
<thead>
<tr>
<th>Access Control</th>
<th>private</th>
<th>default</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>same class</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>same package</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>subclass</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>different class</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>non-subclass</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

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.

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

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.

Hiding and Overriding: Static and Dynamic Binding

```java
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() + 1;
        ...
    }
}
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())
```

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

<table>
<thead>
<tr>
<th>Subclass Instance Method</th>
<th>Superclass Instance Method</th>
<th>Superclass Static Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overrides</td>
<td>Generates a compile-time error</td>
<td></td>
</tr>
<tr>
<td>Generates a compile-time error</td>
<td>Hides</td>
<td></td>
</tr>
</tbody>
</table>
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.**

```java
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()
Concrete1.pac()
Concrete1.pro()
Concrete1.pub()
```

```java
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()