Access Modifiers

Default permission on classes, variables, and methods is *package protection*.

`private` – Only this class can access.

`protected` – Only this class or subclasses can access.

`public` – Any class can access.

Other Mofifiers

`final` class – subclassing is not allowed.

`final` method – overriding is not allowed.

`final` variable – can not be changed (like C's `const`).

Recall: `abstract`, `static`, and `synchronized`.

There are a few others.

Java Packages

Java classes can be grouped into packages.

Full name of class is: `<package-name>.<class-name>`

```
package my.geometry;

public class Point
{
  ...
}
```

Full name of `Point` is `my.geometry.Point`.

Can, import `my.geometry.*` or `my.geometry.Point` or just use `my.geometry.Point` in code.

Compiled class will be in `my/geometry/Point.class`

Packages provide management of name space – very useful in large software projects.

Object Oriented Metaphor

Objects have state.

Methods are messages sent to objects.

Objects interpret messages as they choose.

Object Oriented Design

- Subclassing
- Code Inheritance
- Polymorphism
- Information Hiding
- Encapsulation
- Frameworks
- Patterns

Subclassing

Many reasons to subclass:

specialization – child is special case of parent.
E.g. `Number`, `Integer`, `Double`.

specification – behavior is implemented by child.
E.g. `InputStream`, `FileInputStream`; Event `Adapter`s.

extension – child provides more functionality.
E.g. `Panel`, `Applet`.

Code Inheritance

Subclasses inherit methods from their parents.

Can make small changes to the behavior of a class by

- subclassing, and then
- overriding a few methods.

Protected interface can provide easier access to superclass.

Example: Hashtable resizing algorithm.

Polymorphism

Can make functions that take many types of objects.

Good for code reuse.

Provides *extensibility*: a subclass with different behavior can be substituted for parent class.

Example: FilterMain and Filters

Information Hiding

Provide *public interface* to object, but hide implementation.

- Clients do not need to know implementation details.
- Implementers do not need to know how object will be used.

Example: Dictionary

Languages can enforce information hiding via access permissions: `public`, `private`, etc.

Encapsulation

Encapsulate simple objects inside composite object.

Example: `ValueSlider`.

Encapsulation and Information Hiding often go hand in hand: Access to encapsulated objects is not allowed, access is only allowed through composite object.

Frameworks

Frameworks provide implementation of high-level interaction of objects in the system.

Applications implement details.

Example, Smalltalk's Model-View-Controller (MVC).

Can use abstract classes to specify framework. Subclasses would implement the details.

Model-View-Controller

Model knows about data and how to manipulate data.

View knows how to display the data. Queries model to get data.

Controller provides interface for user to manipulate the model and the view.

Advantages:

- Different controllers can provide different interfaces for manipulating the same data.
- Multiple views can be provided for the same model.

Example: Clock

Patterns

A *pattern* is a "well-proven generic scheme for solving a recurring design problem."

- Language Independent
- Come from examples in existing systems.
- Like high-level control structures in languages.

Examples: Flyweight, Adapter, Observer

Flyweight

Reduces memory usage for many objects with same state.

Solution: let objects share state.

Example: Scheme's Symbols, Playing Cards

Adapter

Client needs one interface, existing object provides another interface.

Solution: Wrap existing object in an adapter that has desired interface.

Client communicates with adapter, adapter communicates with object.

Observer

Several objects need to stay consistent with data.

Solution: Have list of objects observing data, when data is changed, all observing objects can be notified.

Similar to `Listeners` for `Events` in Java's AWT.

Subtypes in OO

(not Java specific)

Recall, variables can hold "more specific" instances:
`Bird b = new Owl()`

Everything that can be done to/with a `Bird` can be done to/with an `Owl`.

This defines a *subtype* relationship (similar to *subclass*).

type (in OO language) – A collection of operation signatures.

operation signatures – the operation name, the types of the arguments, and the type of the result.

Several sub-type rules: *contravariant rule* and *covariant rule*

Consider `Bird[]` and `Owl[]`. What is subtype relationship?

Subtype Rules

(not Java specific – Java notation abused)

Arrays have two operations, *assignment* and *lookup* (`array[i] = 10` and `x = array[i]`.

Lets view these as methods on array objects:

```
Value Array.elementAt(int index);
void Array.setElementAt(Value v, int index);
```

Recall, that you should be able to do anything with a subtype that you can do with the original type.

Again consider `Bird[]` and `Owl[]`. We have:

```
Bird Bird[].elementAt(int index);
void Bird[].setElementAt(Bird b, int index);
```

and

```
Owl Owl[].elementAt(int index);
void Owl[].setElementAt(Owl o, int index);
```


Owl[] and Bird[]

Suppose `Owl[]` is a subtype of `Bird[]`

```
Bird[] owls = new Owl[10];
Bird[] birds = new Birds[10];

birds.setElementAt(new Bird(),0);     // good
owls.setElementAt(new Bird(),0);      // not good
```

Suppose `Bird[]` is a subtype of `Owl[]`

```
Owl[] owls = new Owl[10];
Owl[] birds = new Birds[10];

owls.elementAt(0).bob();              // good
birds.elementAt(0).bob();             // not good
```


Co[ntra]variance

Possible type relationships between S and T:

- S is a subtype of T.
- T is a subtype of S.
- S is not related to T.

Given operation signatures: R_s S.foo(P_s) and R_t T.foo(P_t)

Contravariant rule: S is subtype of T if R_s is subtype of R_t and P_t is subtype of P_s.

Covariant rule: S is subtype of T if R_s is subtype of R_t and P_s is subtype of P_t.

What is type relationship of `Bird[]` and `Owl[]`?