## 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 Modifiers

`final` class – subclassing is not allowed.

`final` method – overriding is not allowed.

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

Recall: `abstract` and `static`.

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. `SchemeNumber`, `SchemeInteger`, `SchemeDouble`.

*specification* – behavior is implemented by child.
E.g. `InputStream`, `FileInputStream`, `StringInputStream`.

*extension* – child provides more functionality.
E.g. `InputStream`, `DataInputStream`, `PushbackInputStream`



## 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 `Filter`s

## 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.

Do not expose *anything* but the public interface!



## 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.



## 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

See: *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma, R. Helm, R. Johnson, J. Vlissides.



## Flyweight

Reduces memory usage for many objects with same state.

Solution: let objects share state.

Example: Scheme's Symbols (`SchemeSymbol`), Playing Cards

## SchemeSymbol.java

```java
import java.util.Hashtable;

public class SchemeSymbol extends SchemeObject
{
    static Hashtable symbolTable = new Hashtable();
    String value;

    private SchemeSymbol(String str) { value = str; }

    public static SchemeSymbol getSymbol(String str)
    {
        // if in table, return it.
        if (symbolTable.containsKey(str))
            return (SchemeSymbol)symbolTable.get(str);

        // if not add it and return it.
        SchemeSymbol sym = new SchemeSymbol(str);
        symbolTable.put(str,sym);
        return sym;
    }

    public String toString() { return value; }
}
```



## 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.



## Threads

A modern operating system runs many processes (running programs) at once.

Processes can communicate through IPC (interprocess communication).

These "heavy weight" processes are protected from each other by the operating system.

Often times it is useful to write programs as if they were multiple processes. But protection of operating system is not needed and overhead is too much.

*Threads* are *light weight processes*.

A process can have many running threads. Threads share the process's memory. Each thread has its own call stack.

## Threads vs State-Machines

Consider animation of unfriendly creatures in Doom-like game.

**state-machine:** At each clock tick animate one step of creature movement. Must have kept state to determine where creature currently is. Then must generate next movement.

```
Creature[] creatures = ...;

while(alive())
{
   for(int i=0; i < creatures.length; i++)
      creature[i].animate();
}
```

This works, but gets complicated very quickly as creatures need more diverse sequences of moves.



## Threads vs State-Machines (cont)

**threads:** Each creature is run by its own thread. At each clock tick creature thread gets awakened. Creature thread can do whatever it wants. Creature thread puts itself to sleep when it has finished its turn. Creature's state is stored on the creature thread's call stack.

```
Creature[] creatures = ...;
for(int i=0; i < creatures.length i++)
   creatures.start();  // start creature threads.

while(alive()) {
   nofityAll();  // this notifies all threads waiting
}                // on 'this' object.
```

This assumes that each creature puts itself to sleep by *waiting* on `this` object.



## Threads: Synchronization

When there are multiple threads in the system running at once, sequentially written code might not work as it originally did.

Threads can trample on each other. Given a code block:

```
x = x + 1;
```

Say $x = 10$ and two threads run this block at the same time.

| x | Thread 1 | Thread 2 |
|---|---|---|
| 10 | `x+1` ⇒ 11 | |
| 10 | | `x+1` ⇒ 11 |
| 10 | | `x = 11` |
| 11 | | |
| 11 | `x = 11` | |
| 11 | | |

The code executed twice and the value of `x` is only `11`. It should be `12`.



## Synchronization (cont)

(learn more about synchronization in Operating Systems)

A method can be synchronized on `this`. A block of code can be synchronized on any object.

The object that a block of code or method is synchronized on holds the *lock* on that block of code. No other thread can enter a synchronized block that uses the same object as its lock.

```
public synchronized int incX()
{
   x = x + 1;
   return x;
}
```

or

```
synchronized (<expression>)
{
   <code>
}
```

## Threads and Methods

Threads of control are an orthogonal issues to classes. Threads make method invocations on objects.

Any thread can invoke any method on any object.



## Where to use Threads

Good places to use Threads include:

- GUI Animation.
- Server Applications.
- IO intensive Applications.

Why? Because threads allow you to easily do multiple things roughly at the same time.