Chapter 6 : Reusing Classes

The first way is to create objects of your existing classes inside the new class. ---- *composition*

You're simply reusing the functionality of the code, not its form.

The second approach is to create a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. ---- *inheritance*

6.1 Composition syntax

You simply place object references inside new classes.

Sample : SprinklerSystem.java

Every nonprimitive object has a ***toString( )*** method, and it's called in special situations when the compiler wants a **String**.

Primitives that are fields in a class are automatically initialized to zero. But the object references are initialized to **null**, and if you try to call methods for any of them you'll get an exception.

If you want the references initialized, you can do it:

1. At the point the objects are defined. (initialized before the constructor is called .)

2. In the constructor.

3. Right before you actually need to use the object. ---- *lazy initialization*

Sample : Bath.java

6.2 Inheritance syntax

You're always doing inheritance when you create a class ---- Java's standard root class **Object** .

```
class  derived_class_name  extends  base_class_name {
}
```

The derived class will automatically get all the data members and methods in the base class.

Sample : Detergent.java

Both ***Cleanser*** and ***Detergent*** contain a **main( )** method.

You can create a **main( )** for each class, but only the **main( )** for the class invoked on the command line will be called. ---- **main( )** is **public**, and the class doesn't need to be **public**.

This technique of putting a **main( )** in each class allows easy unit testing for each class.

Summary :

1. To plan for inheritance, as a general rule make all fields private and all methods public or protected.

2. You can take a method defined in the base class and modify it.

3. You can also add new methods to the derived class.

inheritance ---- *reusing the interface*

Java has the keyword **super** that refers to the “superclass”. Thus the expression **super.scrub( )** calls the base-class version of the method **scrub( )**.

6.2.1 Initializing the base class

When you create an object of the derived class, it contains a *subobject* of the base class.

The base-class constructor has all the appropriate knowledge and privileges to perform the base-class initialization.

Java automatically inserts calls to the base-class constructor in the derived-class constructor.

Sample : Cartoon.java   Chess.java

The construction happens from the base “outward”, so the base class is initialized before the derived-class constructors can access it.

If your class doesn’t have default arguments, or if you want to call a base-class constructor that has an argument, you must explicitly write the calls to the base-class constructor using **super** and the appropriate argument list .

Note : The call to the base-class constructor *must* be the first thing you do in the derived-class constructor .

6.3 Combining composition and inheritance

Sample : PlaceSetting .java

Note : While the compiler forces you to initialize the base classes, it doesn’t watch over you to initialize the member objects.

6.3.1 Guaranteeing proper cleanup

If you want something cleaned up for a class, you must explicitly write a special method to do it, and make sure that the client programmer knows that they must call this method.

Sample : CADSystem .java

The **finally** clause means “always call **cleanup( )** for **x**, no matter what happens.”

Summary :

Follow the same form imposed by a C++ compiler on its destructors:
1. Perform all of the cleanup work specific to your class, in the reverse order of creation.
2. Call the base-class cleanup method .

6.3.2 Name Hiding

If a Java base class has a method name that's overloaded several times, redefining that method name in the derived class will *not* hide any of the base-class versions.

Sample : Hide.java

6.4 Choosing composition vs. inheritance

Composition is generally used when you want the features of an existing class, but not its interface. ---- private member

However, making the members **public** assists the client programmers to use the class and requires less code complexity. (Note :This is a special case.)

Sample : Car.java

When you inherit, you take an existing class and make a special version of it.

is-a ---- inheritance *has-a* ---- composition

6.5 protected

This is **private** as far as the class user is concerned, but available to anyone who inherits from this class or anyone else in the same **package**. ---- protected

protected in Java is automatically “friendly”.

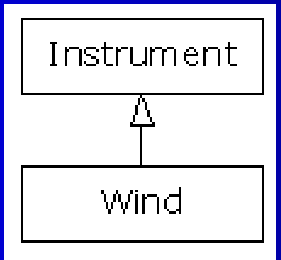
Rule : Leave the data members **private**, then allow controlled access to inheritors of the class through **protected** methods.

Sample : Orc.java

6.6 Upcasting

Sample : Wind.java

The **tune( )** method accepts an **Instrument** reference. However, in **Wind**.**main( )** the **tune( )** is given a **Wind** reference.



Upcasting is always safe because you're going from a more specific type to a more general type. The only thing that can occur to the class interface is losing of methods.

Summary :

- We’ll use existing classes to build new classes with composition. Less frequently, we’ll use inheritance.

- If we must upcast, then inheritance is necessary.

6.7 *final*

It means “This cannot be changed”.

6.7.1 Final data

1.It can be a *compile-time constant* that won’t ever change.

2.It can be a value initialized at run-time that you don’t want changed.

A field that is both **static** and **final** has only one piece of storage that cannot be changed.

With a primitive, **final** makes the *value* a constant, but with an object reference, **final** makes the *reference* a constant. However, the object itself can be modified. This restriction includes arrays.

Sample : FinalData.java

```
public static final int VAL_THREE = 39;
```

final static primitives with constant initial values are named with all capitals by convention, with words separated by underscores.

The values of **i4** for **fd1** and **fd2** are unique, but the value for **i5** is not changed by creating the second **FinalData** object. ---- static vs non-static

Blank finals:

Blank finals are fields that are declared as **final** but are not given an initialization value.

In all cases, the blank final *must* be initialized before it is used.

A **final** field inside a class can now be different for each object and yet retains its immutable quality.

Sample : BlankFinal.java

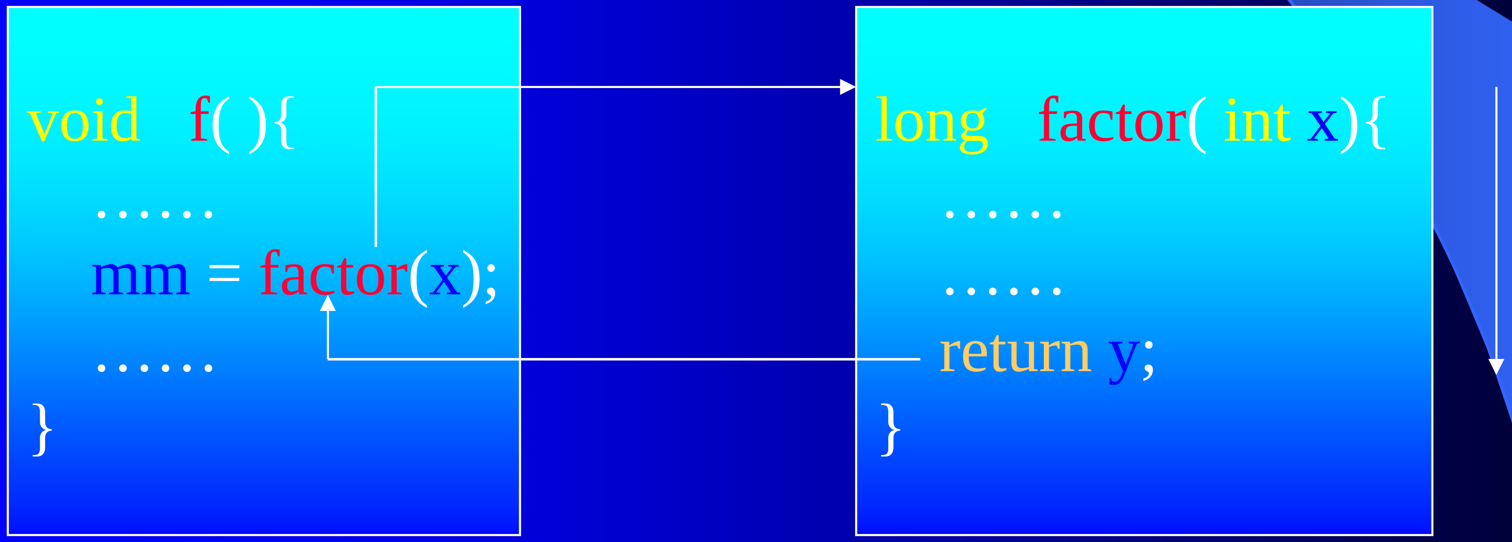
Final arguments:

Sample : FinalArguments.java

6.7.2 Final methods

The first reason is to put a “lock” on the method to prevent any inheriting class from changing its meaning.

The second reason for **final** methods is efficiency. ---- inline



```
void   f( ){
    ……
    mm = factor(x);
    ……
}
```

```
long   factor( int x){
    ……
    ……
    return y;
}
```

Function Invoking

It’s better to make a method **final** only if it’s quite small or if you want to explicitly prevent overriding.

final and private :

Any **private** methods in a class are implicitly **final**.

If you try to override a private method, you’ve just created a new method.

Sample : FinalOverridingIllusion.java

“Overriding” can only occur if the method is part of the base-class interface.

Since a **private** method is unreachable and effectively invisible, it doesn’t factor into anything except for the code organization of the class.

6.7.3 Final classes

Defining the class as **final** simply prevents inheritance—nothing more.

However, the data members can be **final** or not, as you choose.

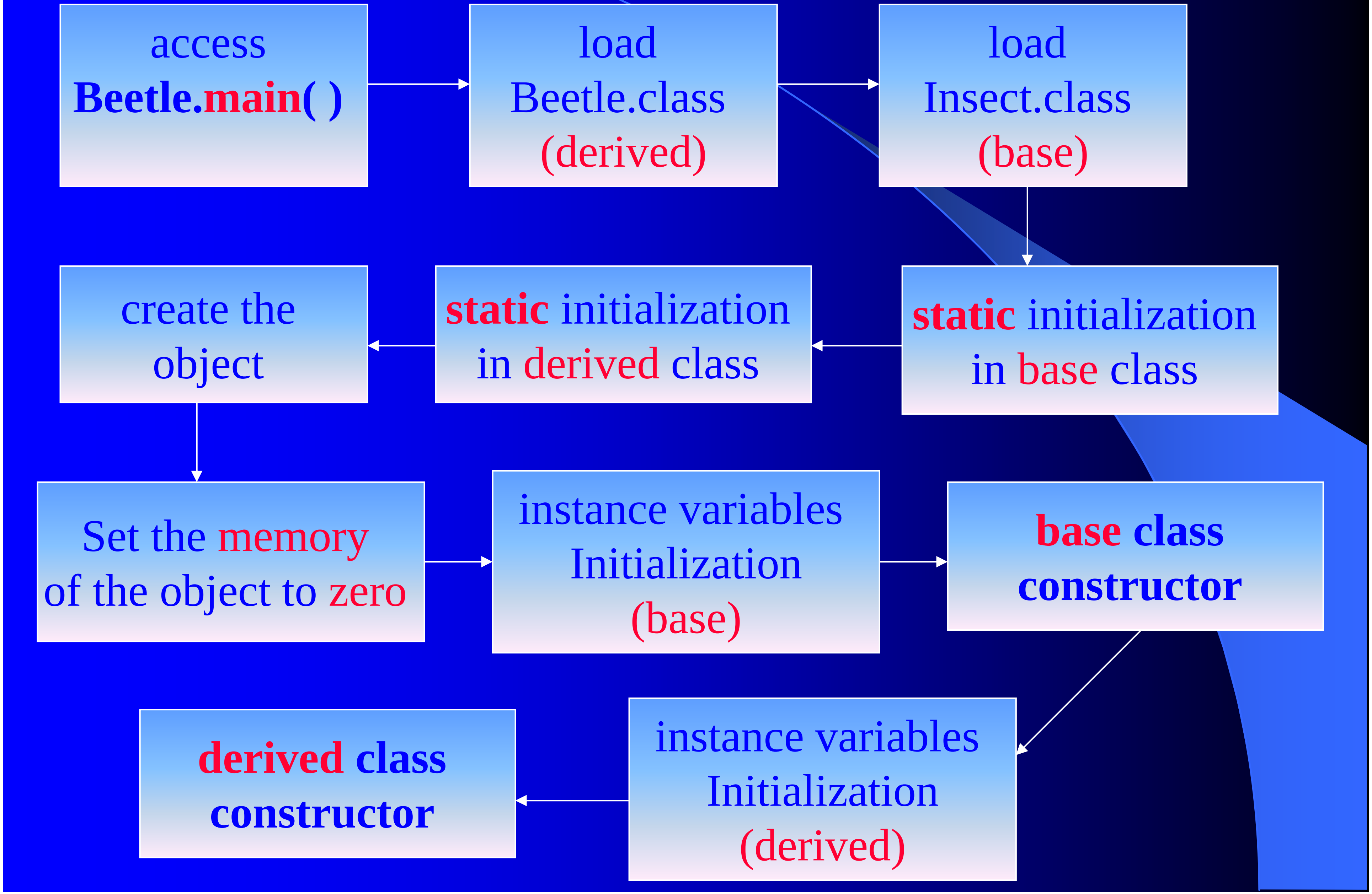
All methods in a **final** class are implicitly **final.**

Sample : Jurassic.java

6.7 Initialization and class loading

Class code is loaded at the point of first use. ---- the first object of that class is constructed or a **static** field or **static** method is accessed.

Sample : Beetle.java



access **Beetle.main( )**

load Beetle.class (derived)

load Insect.class (base)

static initialization in base class

static initialization in derived class

create the object

Set the memory of the object to zero

instance variables Initialization (base)

base class constructor

instance variables Initialization (derived)

derived class constructor

When you start a design you should generally prefer composition during the first cut and use inheritance only when it is clearly necessary.

Our goal is a hierarchy in which each class has a specific use and is neither too big nor annoyingly small.