13 面向对象编程V：可修改性.

一. 可修改性.

1. (狭义) 可修改性：对已有实现的修改.

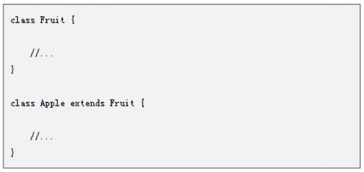
2. 可扩展性：对新的实现的扩展.

3. 灵活性：对实现的动态配置.

二. 继承和组合的选择

- 组合和继承都允许你在新的类中设置子对象（subobject），组合是显式地这样做的，而继承则是隐式的。

继承：



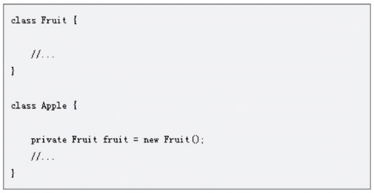
```
class Fruit {

   //...
}

class Apple extends Fruit {

   //...
}
```

组合：



```
class Fruit {

   //...
}

class Apple {

   private Fruit fruit = new Fruit();
   //...
}
```

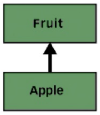


Figure 1. The inheritance relationship

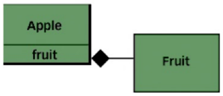
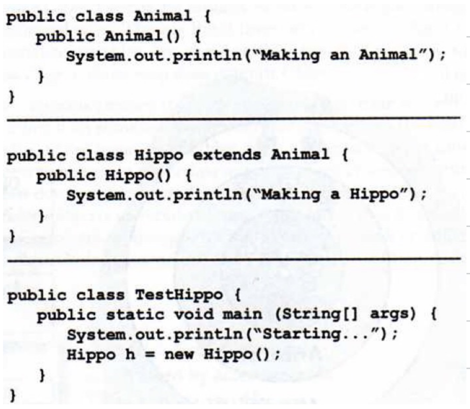


Figure 2. The composition relationship

- 组合技术通常用于你想要在新类中使用现有类的功能而非它的接口的情形。即，你在新类中嵌入某个对象，借其实现你所需要的功能，但新类的用户看到的只是你为新类所定义的接口，而非嵌入对象的接口。为取得此效果，你需要在新类中嵌入一个 private 的现有类的对象。

- 有时，允许类的用户直接访问新类中的组合成份是极具意义的；也就是说，将成员对象声明为 public。如果成员对象自身都实现了具体实现的隐藏，那么这种做法就是安全的。当用户能够了解到你在组装一组部件时，会使得端口更加易于理解。

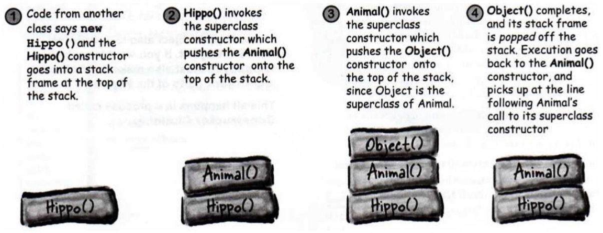
三.继承和构造方法



```
public class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main (String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```



① Code from another class says **new Hippo ()** and the **Hippo()** constructor goes into a stack frame at the top of the stack.

② **Hippo()** invokes the superclass constructor which pushes the **Animal()** constructor onto the top of the stack.

③ **Animal()** invokes the superclass constructor which pushes the **Object()** constructor onto the top of the stack, since Object is the superclass of Animal.

④ **Object()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **Animal()** constructor, and picks up at the line following Animal's call to its superclass constructor

带继承的初始化:

1. Access Main(), load base class, Load until the root base class
2. Static Initialization in the root base class then the next derived class, and so on
3. All the instance variable(primitives and the object reference are set to 0 or 0.0 false null)
4. The base-class constructor will be called
5. The instance variables are initialized in textual order
6. The rest of the body of the constructor is executed

- 事实上，一个类的初始化包括3个步骤：
  - 加载（Loading），由类加载器执行，查找字节码，并创建一个Class对象（只是创建）；
  - 链接（Linking），验证字节码，为静态域分配存储空间（只是分配，并不初始化该存储空间），解析该类创建所需要的对其它类的应用；
  - 初始化（Initialization），首先执行静态初始化块static{}，初始化静态变量，执行静态方法（如构造方法）。

常量在编译阶段会存入调用它的类的常量池中，本质上没有直接引用到定义该常量的类，因此不会触发定义常量的类的初始化.

Example:

```
class StaticBlock {
	static final int c = 3;

	static final int d;

	static int e = 5;
	static {
		d = 5;
		e = 10;
		System.out.println("Initializing");
	}
	StaticBlock() {
		System.out.println("Building");
	}
}

public class StaticBlockTest {
	public static void main(String[] args) {
		System.out.println(StaticBlock.c);
		System.out.println(StaticBlock.d);
		System.out.println(StaticBlock.e);
	}
}
```

输出:

```
3
Initializing
5
10
```

```
class Const{
  public static final String NAME = "我是常量";
  static{
    System.out.println("初始化Const类");
  }
}

public class FinalTest{
  public static void main(String[] args){
    System.out.println(Const.NAME);
  }
}
```

输出:

我是常量.

```
class Const{
  static{
    System.out.println("初始化Const类");
  }
}

public class ArrayTest{
  public static void main(String[] args){
    Const[] con = new Const[5];
  }
}
```

无输出.