可修改性

刘 钦

南京大学软件学院

reference

- http://bobah.net/book/export/html/55
- http://stackoverflow.com/questions/1504633/what-is-the-point-of-invokeinterface
- http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html
- http://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html

Outline

- 可修改性
- 继承vs组合
- 继承和构造方法

可修改性

动物园程序

每一种动物用一个对象表示

Client代码

创建类的
代码

```
Animal[] animals = new Animal[5];
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
animals [3] = new Hippo();
animals [4] = new Lion();

for (int i = 0; i < animals.length; i++) {

    animals[i].eat();

    animals[i].roam();
}
```

使用类的
代码

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

But look what you get to do... you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example), you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

When 'i' is 0, a Dog is at index 0 in the array, so you get the Dog's eat() method. When 'i' is 1, you get the Cat's eat() method

Same with roam().

关于Client代码默认的知识

- 大量的
- 分散的
- 如果发生修改重新编译的话，是需要大量的时间的

需求的变更

- 狗eat的行为发生改变
- 新的一种动物加入进来
- 创建一个能放任何动物的list

可修改性

- （狭义）可修改性
  - 对已有实现的修改
- 可扩展性
  - 对新的实现的扩展
- 灵活性
  - 对实现的动态配置

实现可修改性对Client代码的影响

- （狭义）可修改性
  - 对已有实现的修改
  - 希望不影响Client代码
- 可扩展性
  - 对新的实现的扩展
  - 希望不影响Client使用类的代码
- 灵活性
  - 对实现的动态配置
  - 希望不影响Client使用类的代码

实现的修改—狗eat行为发生修改

- class Dog extends Animal{
- public void eat(){
  - 。。。
  - xxx
- }
- }

可扩展性—新的动物

- animal[4] = new Panda();

灵活性-//可以放指向任何继承Animal的动物

- class Zoo{
  - List<Animal> petlist;
- }

继承vs组合

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。

Inheritance

```
class Fruit {

    //...
}

class Apple extends Fruit {

    //...
}
```

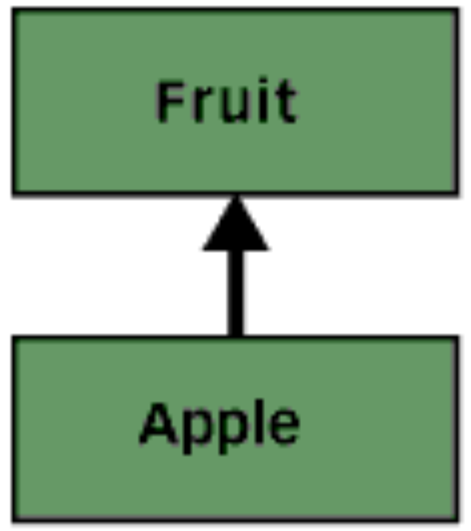


Figure 1. The inheritance relationship

Composition

```
class Fruit {

    //...
}

class Apple {

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

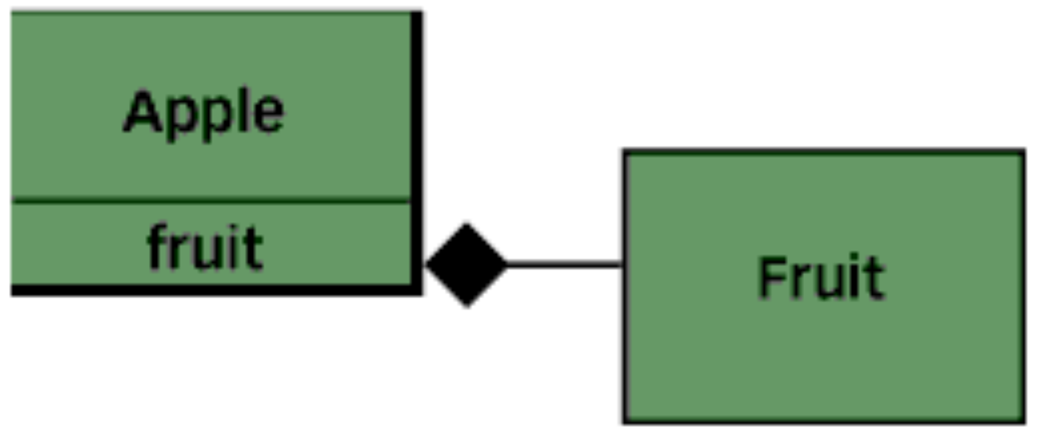


Figure 2. The composition relationship

The change

继承使得添加一个新的子类很容易
但是对父类做改动，会使得所有子类都要做改动

- Adding a new subclass
  - Inheritance helps make code easier to change
- Changing the superclass interface
  - one little change to a superclass can ripple out and require changes in many other places in the application's code.

Code reuse via inheritance I

```
class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {

        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple extends Fruit {
}

class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Code reuse via inheritance II

```
class Peel {

    private int peelCount;

    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }

    public int getPeelCount() {

        return peelCount;
    }
    //...
}

class Fruit {

    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel() {

        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}
```

```
// Apple still compiles and works fine
class Apple extends Fruit {
}

// This old implementation of Example1
// is broken and won't compile.
class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Code reuse via composition I

```
class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {

        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple {

    private Fruit fruit = new Fruit();

    public int peel() {
        return fruit.peel();
    }
}

class Example2 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Code reuse via composition II

```
class Peel {

    private int peelCount;

    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }

    public int getPeelCount() {

        return peelCount;
    }
    //...
}

class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public Peel peel() {

        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}
```

```
// Apple must be changed to accomodate
// the change to Fruit
class Apple {

    private Fruit fruit = new Fruit();

    public int peel() {

        Peel peel = fruit.peel();
        return peel.getPeelCount();
    }
}

// This old implementation of Example2
// still works fine.
class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Comparing composition and inheritance

- It is easier to change the interface of a back-end class (composition) than a superclass (inheritance). As the previous example illustrated, a change to the interface of a back-end class necessitates a change to the front-end class implementation, but not necessarily the front-end interface. Code that depends only on the front-end interface still works, so long as the front-end interface remains the same. By contrast, a change to a superclass's interface can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclass's interface.

- It is easier to change the interface of a front-end class (composition) than a subclass (inheritance). Just as superclasses can be fragile, subclasses can be rigid. You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its supertypes. For example, you can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.

- Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object. With inheritance, you get the image of the superclass in your subclass object image as soon as the subclass is created, and it remains part of the subclass object throughout the lifetime of the subclass.

- It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition, unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.

- The explicit method-invocation forwarding (or delegation) approach of composition will often have a performance cost as compared to inheritance's single invocation of an inherited superclass method implementation. I say "often" here because the performance really depends on many factors, including how the JVM optimizes the program as it executes it.

- With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remain inside the same class.

Choosing between inheritance and composition

- Make sure inheritance models the is-a relationship
- Don't use inheritance just to get code reuse
- Don't use inheritance just to get at polymorphism

组合的选择

- 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。

  - Car对象即为一个好例子：

Car

- //: c06:Car.java
- // Composition with public objects.

- class Engine {
-   public void start() {}
-   public void rev() {}
-   public void stop() {}
- }

- class Wheel {
-   public void inflate(int psi) {}
- }

- class Window {
-   public void rollup() {}
-   public void rolldown() {}
- }

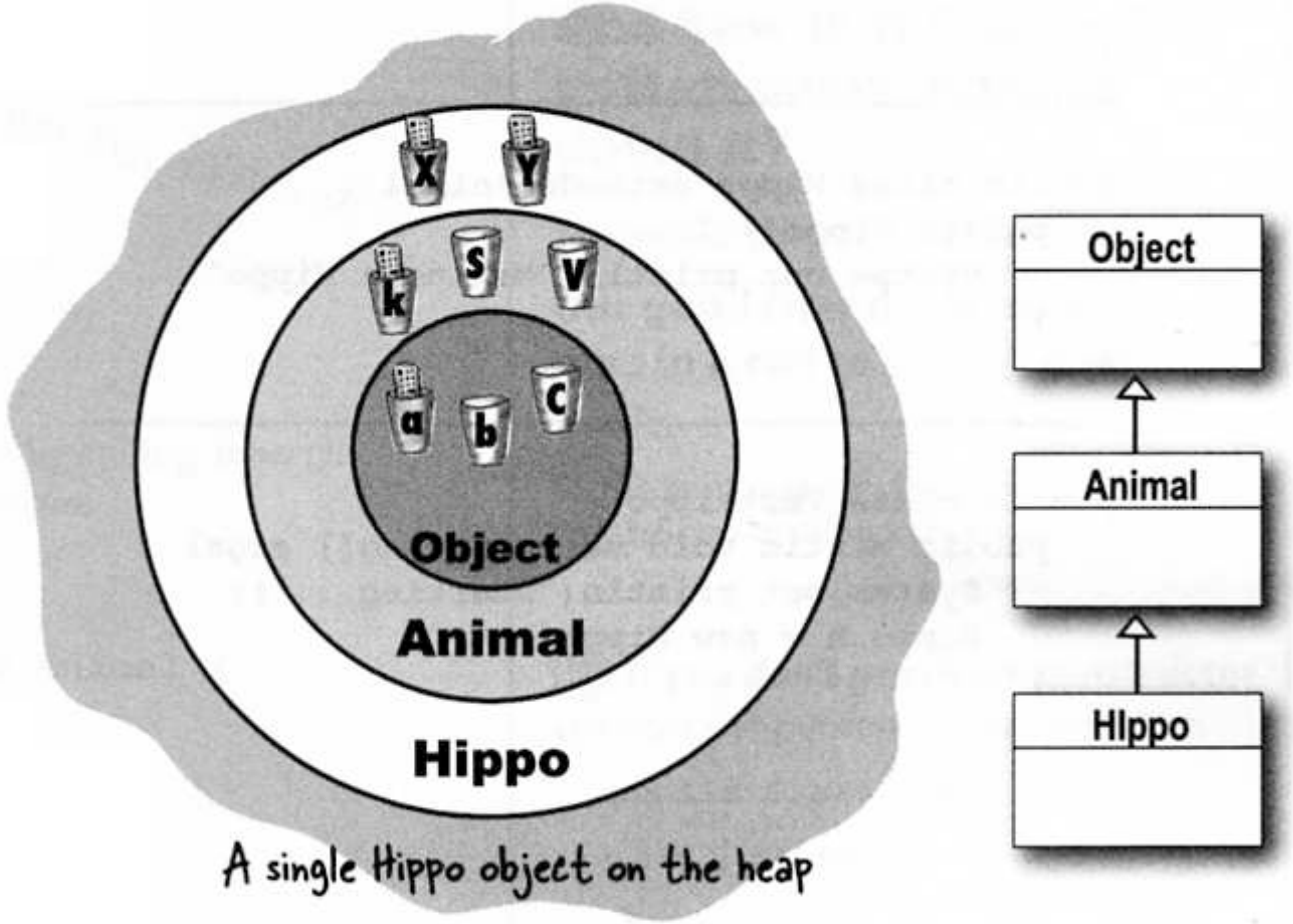
- class Door {
-   public Window window = new Window();
-   public void open() {}
-   public void close() {}
- }

```
public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

继承和构造方法

Inheritance and constructors

- `



A new **Hippo** object also IS-A **Animal** and IS-A **Object**. If you want to make a **Hippo**, you must also make the **Animal** and **Object** parts of the **Hippo**.

This all happens in a process called **Constructor Chaining**.

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

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

```java
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

How to invoke a superclass constructor?

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
BAD! →   Animal(); ← NO! This is not legal!
         size = newSize;
   }
}
```

```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      super(); ← you just say super()
      size = newSize;
   }
}
```

And how is it that we've gotten away without doing it?

You probably figured that out.

Our good friend the compiler puts in a call to *super()* if you don't.

So the compiler gets involved in constructor-making in *two* ways:

1. **If you *don't* provide a constructor**

The compiler puts one in that looks like:

```
public ClassName() {
        super();
}
```

2. **If you *do* provide a constructor but you do *not* put in the call to super()**

The compiler will put a call to super() in each of your overloaded constructors.* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to *super()* is always a no-arg call. If the superclass has overloaded constructors, only the no-arg one is called.

Possible constructors for class Boop

☑
```java
public Boop() {
    super();
}
```

☑
```java
public Boop(int i) {
    super();
    size = i;
}
```

These are OK because the programmer explicitly coded the call to super(), as the first statement.

☑
```java
public Boop() {
}
```

☑
```java
public Boop(int i) {
    size = i;
}
```

These are OK because the compiler will put a call to super() in as the first statement.

⊘
```java
public Boop(int i) {
    size = i;
    super();
}
```

BAD!! This won't compile! You can't explicitly put the call to super() below anything else.

Superclass constructor with arguments

```
public abstract class Animal {
   private String name;          <- All animals (including
                                    subclasses) have a name

   public String getName() {     <- A getter method that
      return name;                  Hippo inherits
   }

   public Animal(String theName) {
      name = theName;            <- The constructor that
   }                                takes the name and assigns
}                                   it the name instance
                                    variable
```

```
public class Hippo extends Animal {

   public Hippo(String name) {
      super(name);               <- Hippo constructor takes a name
   }
}                                   it sends the name up the Stack to
                                    the Animal constructor
```

```
public class MakeHippo {
   public static void main(String[] args) {  Make a Hippo, passing the
      Hippo h = new Hippo("Buffy");        <- name "Buffy" to the Hippo
      System.out.println(h.getName());        constructor. Then call the
   }                                          Hippo's inherited getName()
}
```

Invoking one overloaded constructor from another

Use this() to call a constructor from another overloaded constructor in the same class.

The call to this() can be used only in a constructor, and must be the first statement in a constructor.

A constructor can have a call to super() OR this(), but never both!

```java
class Mini extends Car {

    Color color;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // more initialization
    }

    public Mini(int size) {
        this(Color.Red);
        super(size);

    }
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls super()).

This is The Real Constructor that does The Real Work of initializing the object (including the call to super())

Won't work!! Can't have super() and this() in the same constructor, because they each must be the first statement!

File Edit Window Help Drive

```
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor

      super();
        ^
```

```
$ javap -verbose -c -private HelloWorld
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object
	SourceFile: "HelloWorld.java"
	minor version: 0
	major version: 50
	Constant pool:
const #1 = Method #6.#15; // java/lang/Object."<init>":()V
const #2 = Field #16.#17; // java/lang/System.out:Ljava/io/
PrintStream;
const #3 = String #18; // Hello, world!
const #4 = Method #19.#20; // java/io/PrintStream.println:
(Ljava/lang/String;)V
const #5 = class #21; // HelloWorld
const #6 = class #22; // java/lang/Object
const #7 = Asciz <init>;
const #8 = Asciz ()V;
const #9 = Asciz Code;
const #10 = Asciz LineNumberTable;
const #11 = Asciz main;
const #12 = Asciz ([Ljava/lang/String;)V;
const #13 = Asciz SourceFile;
const #14 = Asciz HelloWorld.java;
const #15 = NameAndType #7:#8;// "<init>":()V
const #16 = class #23; // java/lang/System
const #17 = NameAndType #24:#25;// out:Ljava/io/
PrintStream;
const #18 = Asciz Hello, world!;
const #19 = class #26; // java/io/PrintStream
const #20 = NameAndType #27:#28;// println:(Ljava/lang/
String;)V
const #21 = Asciz HelloWorld;
const #22 = Asciz java/lang/Object;
const #23 = Asciz java/lang/System;
const #24 = Asciz out;
const #25 = Asciz Ljava/io/PrintStream;;
const #26 = Asciz java/io/PrintStream;
const #27 = Asciz println;
const #28 = Asciz (Ljava/lang/String;)V;

{
public HelloWorld();
	Code:
		Stack=1, Locals=1, Args_size=1
		0: aload_0
		1: invokespecial #1; //Method java/lang/
Object."<init>":()V
		4: return

	LineNumberTable:
		line 1: 0

public static void main(java.lang.String[]);
	Code:
		Stack=2, Locals=1, Args_size=1
		0: getstatic #2; //Field java/lang/
System.out:Ljava/io/PrintStream;
		3: ldc #3; //String Hello, world!
		5: invokevirtual #4; //Method java/io/
PrintStream.println:(Ljava/lang/String;)V
		8: return
	LineNumberTable:
		line 5: 0
		line 6: 8
}
```

Initialization with inheritance

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 primitives and the object reference are set to 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

```
class Characteristic {
  private String s;
  Characteristic(String s) {
    this.s = s;
    System.out.println("Creating Characteristic " + s);
  }
  protected void dispose() {
    System.out.println("finalizing Characteristic " + s);
  }
}

class Description {
  private String s;
  Description(String s) {
    this.s = s;
    System.out.println("Creating Description " + s);
  }
  protected void dispose() {
    System.out.println("finalizing Description " + s);
  }
}
```

```
class LivingCreature {
  private Characteristic p = new Characteristic("is alive");
  private Description t =
    new Description("Basic Living Creature");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void dispose() {
    System.out.println("LivingCreature dispose");
    t.dispose();
    p.dispose();
  }
}
```

```
class Animal extends LivingCreature {
  private Characteristic p= new Characteristic("has heart");
  private Description t =
    new Description("Animal not Vegetable");
  Animal() {
    System.out.println("Animal()");
  }
  protected void dispose() {
    System.out.println("Animal dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}
```

- class Amphibian extends Animal {
-   private Characteristic p =
-     new Characteristic("can live in water");
-   private Description t =
-     new Description("Both water and land");
-   Amphibian() {
-     System.out.println("Amphibian()");
-   }
-   protected void dispose() {
-     System.out.println("Amphibian dispose");
-     t.dispose();
-     p.dispose();
-     super.dispose();
-   }
- }

- public class Frog extends Amphibian {
- private static Test monitor = new Test();
- private Characteristic p = new Characteristic("Croaks");
- private Description t = new Description("Eats Bugs");
- public Frog() {
- System.out.println("Frog()");
- }
- protected void dispose() {
- System.out.println("Frog dispose");
- t.dispose();
- p.dispose();
- super.dispose();
- }
- public static void main(String[] args) {
- Frog frog = new Frog();
- System.out.println("Bye!");
- frog.dispose();
- }

- monitor.expect(new String[] {
- "Creating Characteristic is alive",
- "Creating Description Basic Living Creature",
- "LivingCreature()",
- "Creating Characteristic has heart",
- "Creating Description Animal not Vegetable",
- "Animal()",
- "Creating Characteristic can live in water",
- "Creating Description Both water and land",
- "Amphibian()",
- "Creating Characteristic Croaks",
- "Creating Description Eats Bugs",
- "Frog()",
- "Bye!",
- "Frog dispose",
- "finalizing Description Eats Bugs",
- "finalizing Characteristic Croaks",
- "Amphibian dispose",
- "finalizing Description Both water and land",
- "finalizing Characteristic can live in water",
- "Animal dispose",
- "finalizing Description Animal not Vegetable",
- "finalizing Characteristic has heart",
- "LivingCreature dispose",
- "finalizing Description Basic Living Creature",
- "finalizing Characteristic is alive"
- });

```java

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  private int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = " + radius);
  }
  void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}
```

-
- public class PolyConstructors {
- private static Test monitor = new Test();
- public static void main(String[] args) {
- new RoundGlyph(5);
- }
- } ///:~

- monitor.expect(new String[] {
- "Glyph() before draw()",
- "RoundGlyph.draw(), radius = 0",
- "Glyph() after draw()",
- "RoundGlyph.RoundGlyph(), radius = 5"
- });

Behavior of polymorphic methods inside constructors

- What happens if you're inside a constructor and you call a dynamically-bound method of the object being constructed?
- The overridden method will be called before the object is fully constructed.

类的初始化

- 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）。

下面列出了可能造成类被初始化的操作

- 创建一个Java类的实例对象
- 调用一个Java类的静态方法
- 为类或接口中的静态域赋值
- 访问类或接口中声明的静态域，并且该域的值不是常值变量
- 在一个顶层Java类中执行assert语句
- 调用Class类和反射API中进行反射操作

Example - 类的初始化

- class A{
-     static int value = 100;
-     static{
-         System.out.println("类A初始化");
-     }
- }
- class B extends A{
-     static{
-         System.out.println("类B初始化");
-     }
- }
- public   class StaticFieldInit{
-     public static void main(String[] args){
-         System.out.println(B.value);
-     }
- }

- 结果:

- 类A初始化
- 100

- //当访问一个Java类或接口的静态域时，只有真正声明这个域的类或接口才会被初始化

- 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);
- }
- }

- 结果：
- 3
- Initializing
- 5
- 10

- 原因是这样的：
- 输出c时，由于c是编译时常量，不会引起类初始化，因此直接输出，输出d时，d不是编译时常量，所以会引起初始化操作，即static块的执行，于是d被赋值为5，e被赋值为10，然后输出Initializing，之后输出d为5，e为10。

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：

```
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);
    }
}
```

- 执行后输出的结果如下：

- 我是常量

- 虽然程序中引用了 const 类的常量 NAME，但是在编译阶段将此常量的值“我是常量”存储到了调用它的类 FinalTest 的常量池中，对常量 Const.NAME 的引用实际上转化为了 FinalTest 类对自身常量池的引用。也就是说，实际上 FinalTest 的 Class 文件之中并没有 Const 类的符号引用入口，这两个类在编译成 Class 文件后就不存在任何联系了。

通过数组定义来引用类，不会触发类的初始化：

```
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];
    }
}
```

- 执行后不输出任何信息，说明 Const 类并没有被初始化。

- 但这段代码里触发了另一个名为“LLConst”的类的初始化，它是一个由虚拟机自动生成的、直接继承于java.lang.Object 的子类，创建动作由字节码指令 newarray 触发，很明显，这是一个对数组引用类型的初初始化，而该数组中的元素仅仅包含一个对 Const 类的引用，并没有对其进行初始化。

- 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];
-         for(Const a:con)
-             a = new Const();
-     }
- }

- 这样便会得到如下输出结果：

- 初始化Const类

- 根据四条规则的第一条，这里的 new 触发了 Const 类。

接口的初始化

- 接口也有初始化过程，上面的代码中我们都是用静态语句块来输出初始化信息的，而在接口中不能使用“static{}”语句块，但编译器仍然会为接口生成类构造器，用于初始化接口中定义的成员变量（实际上是 static final 修饰的全局常量）。

- 二者在初始化时最主要的区别是：当一个类在初始化时，要求其父类全部已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量），才会初始化该父接口。这点也与类初始化的情况很不同，回过头来看第 2 个例子就知道，调用类中的 static final 常量时并不会 触发该类的初始化，但是调用接口中的 static final 常量时便会触发该接口的初始化。

```
LIUmatoMacBook-Air:Downloads liuqin$ javap -verbose StaticBlockTest
Classfile /Users/liuqin/Downloads/StaticBlockTest.class
  Last modified 2016-5-12; size 475 bytes
  MD5 checksum 583932c37f5d93c1bc0ded82d2dfd54e
  Compiled from "StaticBlockTest.java"
public class StaticBlockTest
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #8.#17         // java/lang/Object."<init>":()V
   #2 = Fieldref           #18.#19        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = Class              #20            // StaticBlock
   #4 = Methodref          #21.#22        // java/io/PrintStream.println:(I)V
   #5 = Fieldref           #3.#23         // StaticBlock.d:I
   #6 = Fieldref           #3.#24         // StaticBlock.e:I
   #7 = Class              #25            // StaticBlockTest
   #8 = Class              #26            // java/lang/Object
   #9 = Utf8               <init>
  #10 = Utf8               ()V
  #11 = Utf8               Code
  #12 = Utf8               LineNumberTable
  #13 = Utf8               main
  #14 = Utf8               ([Ljava/lang/String;)V
  #15 = Utf8               SourceFile
  #16 = Utf8               StaticBlockTest.java
  #17 = NameAndType        #9:#10         // "<init>":()V
  #18 = Class              #27            // java/lang/System
  #19 = NameAndType        #28:#29        // out:Ljava/io/PrintStream;
  #20 = Utf8               StaticBlock
  #21 = Class              #30            // java/io/PrintStream
  #22 = NameAndType        #31:#32        // println:(I)V
  #23 = NameAndType        #33:#34        // d:I
  #24 = NameAndType        #35:#34        // e:I
  #25 = Utf8               StaticBlockTest
  #26 = Utf8               java/lang/Object
  #27 = Utf8               java/lang/System
  #28 = Utf8               out
  #29 = Utf8               Ljava/io/PrintStream;
  #30 = Utf8               java/io/PrintStream
  #31 = Utf8               println
  #32 = Utf8               (I)V
  #33 = Utf8               d
  #34 = Utf8               I
  #35 = Utf8               e
```

```
{
  public StaticBlockTest();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 18: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: iconst_3
         4: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
         7: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        10: getstatic     #5                  // Field StaticBlock.d:I
        13: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
        16: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
        19: getstatic     #6                  // Field StaticBlock.e:I
        22: invokevirtual #4                  // Method java/io/PrintStream.println:(I)V
        25: return
      LineNumberTable:
        line 20: 0
        line 21: 7
        line 22: 16
        line 23: 25
}
SourceFile: "StaticBlockTest.java"
```

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

-     static final int d;

-     static int e = 5;

-     StaticBlock() {
-         System.out.println("Building");
-     }
- }

- 在这段代码中，对e的声明被放到static块后面，于是，e会先被初始化为10，再被初始化为5，所以这段代码中e会输出为5。