



# 第7章 复用类

# 导入

- 面向对象三大特征之二:继承
- 如何重复使用类
  - 复制粘贴
  - 组合和继承

## 7.1 组合

- 注意toString的用法

```
class WaterSource {  
    private String s;  
    WaterSource() {  
        System.out.println("WaterSource()");  
        s = "Constructed";  
    }  
    @Override  
    public String toString() { return s; }  
}
```

```
public class SprinklerSystem {  
    private String valve1;  
    private WaterSource source = new WaterSource();  
    @Override  
    public String toString() {  
        return "valve1 = " + valve1 + " " +  
            "source = " + source; // [1]  
    }  
    public static void main(String[] args) {  
        SprinklerSystem sprinklers = new SprinklerSystem();  
        System.out.println(sprinklers);  
    }  
}
```

# 回顾初始化

- 定义对象的地方
- 构造器初始化
- 惰性初始化
- 实例初始化
- 练习1

```

class Soap {
    private String s;
    Soap() {
        System.out.println("Soap()");
        s = "Constructed"; // 构造器初始化
    }
}

```

```

public class Bath {
    .....
    private int i;
    // 实例初始化
    {
        i = 47;
    }
}

```

```

public class Bath {
    private String s1; // 定义处初始化
    s1 = "Happy",
}

```

```

public class Bath {
    private String s4;
}

```

```

    s2 = "Happy",
    s3, s4;
    .....
}

```

```

    public String toString() {
        if(s4 == null) // 惰性初始化
            s4 = "Joy";
        .....
    }
}

```



## 7.2 继承语法

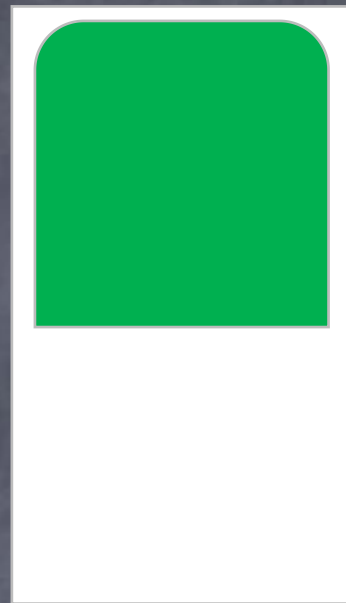
- 除非显式地继承其他类, 否则就隐式地继承 Java 的标准根类对象 (Object)
- 使用 extends 关键字
- `class SubClass extends SuperClass{}`
- 练习<sub>2</sub>

```
class Cleanser {  
    private String s = "Cleanser";  
    public void append(String a) { s += a; }  
    public void apply() {  
        append(" apply()");  
    }  
    public void scrub() {  
        append(" scrub()");  
    }  
    @Override  
    public String toString() { return s; }  
    public static void main(String[] args) {  
        Cleanser X = new Cleanser();  
        X.apply();  
        System.out.println(X);  
    }  
}
```

```
public class Detergent extends Cleanser {  
    // 覆盖一个方法  
    @Override  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub(); // Call base-class version  
    }  
    // 给接口添加新的方法  
    public void foam() { append(" foam()"); }  
    // Test the new class:  
    public static void main(String[] args) {  
        Detergent X = new Detergent();  
        X.apply(); // 通过继承获得  
        X.foam();  
        System.out.println(X);  
    }  
}
```

# 初始化基类

- 导出类: 基类的扩展
- 导出类对象: 包含基类对象
- 调用基类构造器
  - 编译器自动添加调用基类默认构造器
  - 初始化过程是由内向外的





```
class Art {  
    Art() {  
        System.out.println("Art constructor");  
    }  
}
```

```
class Drawing extends Art {  
    Drawing() {  
        System.out.println("Drawing  
constructor");  
    }  
}
```

• 练习 3, 4, 5

```
public class Cartoon extends Drawing {  
    public Cartoon() {  
        System.out.println("Cartoon constructor");  
    }  
    public static void main(String[] args) {  
        Cartoon X = new Cartoon();  
    }  
}
```

# 带参数的构造器

- 显式调用

- super

- 练习 6, 7, 8, 9

```
class Game {
```


```
    Game(int i) {
```

```
        System.out.println("Game constructor");
```

```
    }
```

```
}
```

```
class BoardGame extends Game {
```

```
    BoardGame(int i) { 
```

```
        super(i); // 如果删除会有编译错误
```

```
        System.out.println("BoardGame constructor");
```

```
    }
```

```
}
```

## 7.3 代理

组合、继承与代理

暴露全部/部分接口

采用组合的方式

具有非常强大的可控性

# 示例

```
public class SpaceShipControls {  
  
    void up(int velocity) {}  
  
    void down(int velocity) {}  
  
    void left(int velocity) {}  
  
    .....  
}
```

```
public class SpaceShipDelegation {  
  
    private SpaceShipControls controls = new  
    SpaceShipControls();  
  
    public void back(int velocity) {  
  
        controls.back(velocity);  
  
    }  
  
    public void down(int velocity) {  
  
        controls.down(velocity);  
  
    } .....  
}
```

## 7.4 结合使用组合与继承

```
class Plate {  
    Plate(int i) { ..... }  
}
```

```
class DinnerPlate extends Plate {  
    DinnerPlate(int i) {  
        super(i);  
    }  
}
```

```
public class PlaceSetting extends Custom {  
    private DinnerPlate pl;  
    public PlaceSetting(int i) {  
        super(i + 1);  
        pl = new DinnerPlate(i + 5);  
    }  
    public static void main(String[] args) {  
        PlaceSetting X = new PlaceSetting(9);  
    }  
}
```



## 7.5 如何选择组合与继承

### 组合: 更看重 **功能**

- 新类中添加一个私有字段
- “has-a” 关系

### 继承: 更看重 **接口**

- 导出类更加 **具体化**
- “is-a” 关系



## 7.6 protected 关键字



类的使用者

私有的



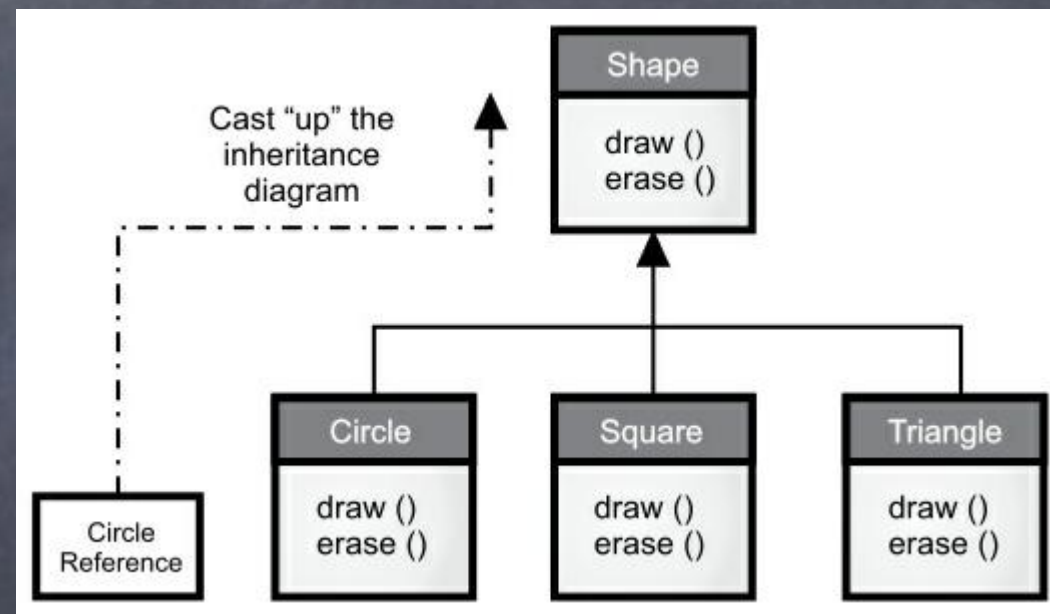
导出类

可访问



## 7.7 向上转型

- 继承并扩展新的方法
- 继承并修改旧的行为
  - 新类是已有类的一种类型
  - 把新类当做已有类来使用



- 能组合不继承

- 练习

# 继承的强大

```
class Instrument {  
    public void play() {}  
  
    static void tune(Instrument i) {  
        // ...  
  
        i.play();  
    }  
}
```

```
public class Wind eXtends Instrument {  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        Instrument.tune(flute);  
    }  
}
```

```
}
```

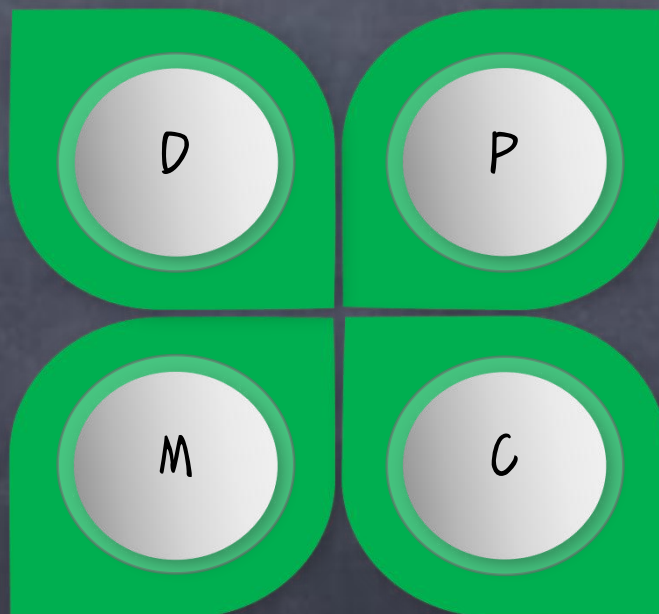
## 7.8 final 关键字

### 修饰数据时

- 基本类型,永不改变
- 引用,不能指向另一个对象
- 空白final,初始化后不再改变

### 修饰方法时

- 导出类不能覆盖
- 内嵌提高效率



### 修饰参数时

无法改变参数引用的指向

### 修饰类时

不允许继承



## 7.9 初始化与类的加载

- 每一个类都存在一个.class文件
- 使用时加载class文件
  - 创建第一个对象
  - 访问类中的static字段或方法
- **加载顺序**: 导出类——>基类
- 初始化顺序: 基类——>导出类

# 类加载的步骤

加载器classloader寻找class文件

如果发现有基类,继续寻找并加载

根基类static初始化

导出类static初始化

在内存中划分合适的空间

空间清零

字段初始化

构造器初始化



# 课堂练习

**练习1:** (2) 创建一个简单的类。在第二个类中，将一个引用定义为第一个类的对象。运用惰性初始化来实例化这个对象。

**练习2:** (2) 从**Detergent**中继承产生一个新的类。覆盖**scrub()**并添加一个名为**sterilize()**的新方法。

读者会发现，构建过程是从基类“向外”扩散的，所以基类在导出类构造器可以访问它之前，就已经完成了初始化。即使你不为**Cartoon()**创建构造器，编译器也会为你合成一个默认的构造器，该构造器将调用基类的构造器。

**练习3:** (2) 证明前面这句话。

**练习4:** (2) 证明基类构造器：(a) 总是会被调用；(b) 在导出类构造器之前被调用。

**练习5:** (1) 创建两个带有默认构造器（空参数列表）的类**A**和类**B**。从**A**中继承产生一个名为**C**的新类，并在**C**内创建一个**B**类的成员。不要给**C**编写构造器。创建一个**C**类的对象并观察其结果。

**练习6:** (1) 用**Chess.java**来证明前一段话。

**练习7:** (1) 修改练习5，使**A**和**B**以带参数的构造器取代默认的构造器。为**C**写一个构造器，并在其中执行所有的初始化。

**练习8:** (1) 创建一个基类，它仅有一个非默认构造器；再创建一个导出类，它带有默认构造器和非默认构造器。在导出类的构造器中调用基类的构造器。

**练习9:** (2) 创建一个**Root**类，令其含有名为**Component 1**、**Component 2**、**Component 3**的类的各一个实例（这些也由你写）。从**Root**中派生一个类**Stem**，也含有上述各“组成部分”。所有的类都应带有可打印出类的相关信息的默认构造器。



# 课堂练习

用该**protected**方法。

练习16: (2) 创建一个名为**Amphibian**的类。由此继承产生一个称为**Frog**的类。在基类中设置适当的方法。在**main()**中, 创建一个**Frog**并向上转型至**Amphibian**, 然后说明所有方法都可工作。

练习17: (1) 修改练习16, 使**Frog**覆盖基类中方法的定义 (令新定义使用相同的方法特征签名)。请留心**main()**中都发生了什么。

练习18: (2) 创建一个含有**static final**域和**final**域的类, 说明二者间的区别。

3, 试着调用该

练习19: (2) 创建一个含有指向某对象的空白**final**引用的类。在所有构造器内部都执行空白类的方法内部调**final**的初始化动作。说明Java确保**final**在使用前必须被初始化, 且一旦被初始化即无法改变。

练习20: (1) 展示**@Override**注解可以解决本节中的问题。

练习21: (1) 创建一个带**final**方法的类。由此继承产生一个类并尝试覆盖该方法。

练习22: (1) 创建一个**final**类并试着继承它。

练习23: (2) 请证明加载类的动作仅发生一次。证明该类的第一个实体的创建或者对**static**成员的访问都有可能引起加载。

练习24: (2) 在**Beetle.java**中, 从**Beetle**类继承产生一个具体类型的“甲壳虫”。其形式与现有类相同, 跟踪并解释其输出结果。



提问