# Java面向对象

# 类

- 类由数据属性与函数属性封装而成。
  - Java语言把数据属性称为域变量（field varaible）、属性、成员变量等；
  - 而把函数属性称为成员方法，简称为方法。

# 类声明

The items between [ and ] are optional.

[modifiers] class *ClassName* [**extends** *SuperClassName*] [**implements** *InterfaceNames*] {

. . .

}

- 例子：

```
class NameOfClass {
    . . .
}
```

```
class ImaginaryNumber extends Number implements Arithmetic {
    . . .
}
```

```
public class NameOfClass extends SuperClassName {
    . . .
}
```

```
abstract class ImaginaryNumber implements Arithmetic, Collection{
    . . .
}
```

# 类声明

[modifiers] class *ClassName* [**extends** *SuperClassName*] [**implements** *InterfaceNames*] {

. . .

}

- modifiers 可以声明这个类是abstract, final还是public
- ClassName 是当前定义的类的名字
- SuperClassName 是当前定义类ClassName的超类（父类）（只能有一个，即Java是单继承）
- InterfaceNames 是一个接口的名字的集合（以逗号隔开），当前定义的类会实现这些接口

# 类声明

[modifiers] class *ClassName* [extends *SuperClassName*] [implements *InterfaceNames*] {

. . .

}

很多时候类创建之后并不想要被继承：

1. 继承本身会打破封装性，会泄露敏感信息，并且可能会带来低效存储的问题
2. final修饰的class可以使的编译器进行内联函数的优化，因为method不会再改变了

如果modifiers是final，那么其不可以有任何子类

如果modifiers是public，那么其可以被包外的类访问

如果modifiers为空（即没有修饰符），那么就是一个缺省类（the default, also known as package-private），只可以被当前包里的类所访问

如果modifiers是abstract，那么其是一个虚类（或抽象类），不可以实例化对象，是用来继承的。

# 包（package）结构

- 包为Java的类提供了一种类似文件系统的组织形式，可以有效避免命名冲突。将相关的类组织到相关的包中是很好的模块化设计。
- 每一个Java的类都隶属于一个包
- 每个Java的源文件的第一句就是包声明

```
package pkg1[.pkg2[.pkg3...]];
```

- 如果源文件没有包声明语句，那么其隶属于一个无名的默认包中

# 包结构

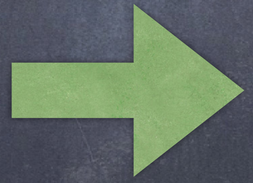
- 例子

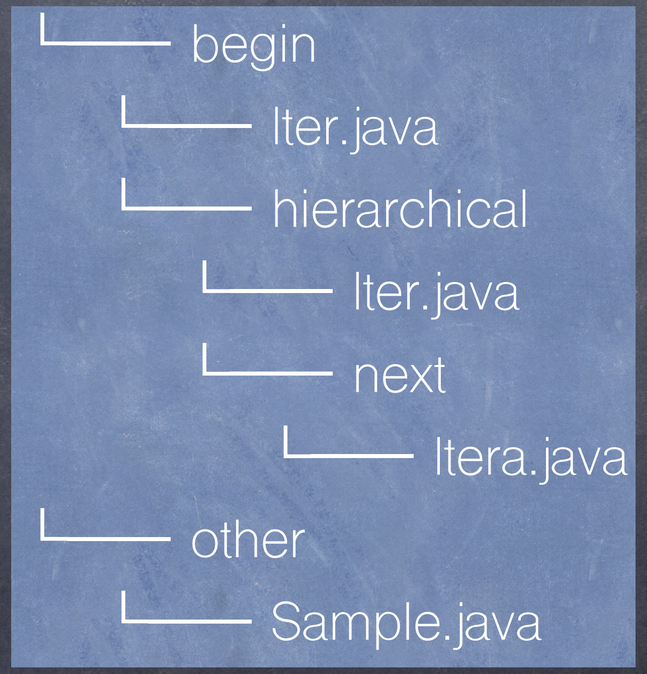
```
package begin;
public class Itera {
}
```

```
package begin.hierarchical;
public class Itera {
}
```

```
package begin.hierarchical.next;
public class Itera {
}
```

```
package other;
public class Sample {
}
```






具体解释执行class文件时，需要添加上包，比如： java begin.hierarchical.next.Itera

# 包结构

- 使用时，利用import语句可以将包的具体类引入

```
package begin;
public class Itera {

}
```

```
package other;
public class Sample {

}
```

```
import begin.Itera;
import other.*;
public class ImportSample {
    Itera i;
    Sample s;
}
```

等价于

```
public class ImportSample {
    begin.Itera i;
    other.Sample s;
}
```

# 包结构

- 例子：使用built-in的包

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;

class MyClass {
  public static void main(String[] args) {
    try{
        String fileInput = "file.txt";
        String fileOutput = "out.txt";
        Scanner myObj = new Scanner(new File(fileInput));
        PrintWriter fileOut = new PrintWriter(new File(fileOutput));
        while(myObj.hasNextLine()){
            String line = myObj.nextLine();
            fileOut.println(line);
        }
        myObj.close();
        fileOut.close();
    }
    catch(Exception e){
        throw new RuntimeException(e);
    }
  }
}
```

# Python的包结构

```
sound/
    __init__.py
    formats/
            __init__.py
            wavread.py
            aiffread.py
            ...
    effects/
            __init__.py
            echo.py
            surround.py
            ...
    filters/
            __init__.py
            equalizer.py
            vocoder.py
            ...
```

A directory must contain a file named `__init__.py` in order for Python to consider it as a package.

This file can be left empty and can also be placed by some the initialization code (These code will be implicitly executed when the package is imported)

- 用法：

```
from sound.formats import aiffread

aiffread.XXX()
```

```
import sound.formats

sound.aiffread.XXX()
```

# 成员变量

- 类的成员变量描述了该类的对象的内部信息，一个成员变量可以是简单变量，也可以是对象、数组等其他结构型数据。成员变量的格式如下：

  [accessSpecifier] [final] type varaibleName [=initial_value];

  其中accessSpecifier 一般为public、protected、private、和缺省（package-private）

  当final被声明时，该成员变量是一个常量（不可改变）

- 在定义类的成员变量时，可以同时赋初值，但对成员变量的操作只能放在方法中。

# 访问控制（public, protected, default, private）

| Modifier | Class<br>当前类 | Package<br>当前类所在的包 | Subclass<br>当前类的子类 | All the others<br>所有其他 |
|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | ✓ | ✗ |
| (Default) | ✓ | ✓ | ✗ | ✗ |
| private | ✓ | ✗ | ✗ | ✗ |

# 访问控制例子

```
class A {
  public int x;
  public void print() { ... }
}

class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

```
package  abc;
class A {
  public int x;
  public void print() { ... }
}

package xyz;
import abc.A;
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

Public允许全权访问，无任何限制，访问成员变量（或者成员方法），需要先实例化对象。

# 访问控制例子

```
class A {
   private int x;
   private void print() { ... }
}
class B {
   void test() {
   A a = new A();
   a.x = 100;
   a.print();
   }
}
```

编译不通过!

```
x has private access in A
                a.x = 100;
                  ^
print() has private access in A
                a.print();
                  ^
2 errors
```

```
class A {
   private int x;
   private void print() { ... }
   boolean comare(A other){
      return this.x > other.x
   }
}
```

The same class can access!

# 访问控制例子

```
//same package
class A {
  protected int x;
  protected void print() { ... }
}
//same package
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

```
package  abc;
class A {
  protected int x;
  protected void print() { ... }
}
package xyz;
import abc.A;
class B extends A {
  void test() {
  A a = new A();
  B b = new B();
  a.x = 100;
  a.print();
  b.x = 100;
  b.print();
  }
}
```

编译不通过！

正确

Protected 允许类本身、其子类（⚠️有一定限制）以及同一个包中所有类访问

# 访问控制例子

```
package abc;
class A {
  int x;
  void print() { ... }
}

package abc;
class B {
  void test() {
  A a = new A();
  a.x = 100;
  a.print();
  }
}
```

缺省的变量允许类本身以及同一个包（其他包则不行，即使是其子类）中所有类访问

# 访问控制

- 一般而言，对于"敏感"的数据应该设为私有变量，从而对外界进行隔离
- 通过公共方法（get和set）来进行访问和修改这些变量

这就是JAVA的封装性

# 封装性例子

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

方法名一般以get开头，接相应的变量名，其中变量名的第一个字母大写，这个方法其实就是选择子，或者访问子，也叫观察子（observer）

变异子方法名一般以set开头，接相应的变量名，这个方法其实就是变异子

# 方法

- 类的成员方法是类的对象的一些行为的描述。成员方法的格式如下：

[accessSpecifier] [final] returnType methodName ([paramlist]) [throws exceptionsList]

同样，其中accessSpecifier 一般为public、protected、private、和缺省（package-private），刻画了和成员变量一样的访问规则。

当final被声明时，该成员方法不可被重写（overriding）。

throws exceptionsList 用来声明方法可能会在运行时出现哪些异常（如IO异常）

javadoc:
用来提供了类或者方法的一些说明
如参数、返回值，
这是java注释的一种。
其他类型的注释：
1. // 单行注释
2. /* 多行注释 */

# 例子

```
public class Hailstone {
    /**
     * Compute a hailstone sequence.
     * @param n  starting number for sequence; assumes n > 0.
     * @return hailstone sequence starting with n and ending with 1.
     */
    public List<Integer> hailstoneSequence(int n) {
        List<Integer> list = new ArrayList<Integer>();
        while (n != 1) {
            list.add(n);
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
        }
        list.add(n);
        return list;
    }
}
```

method signature

method name

parameter declaration

return type

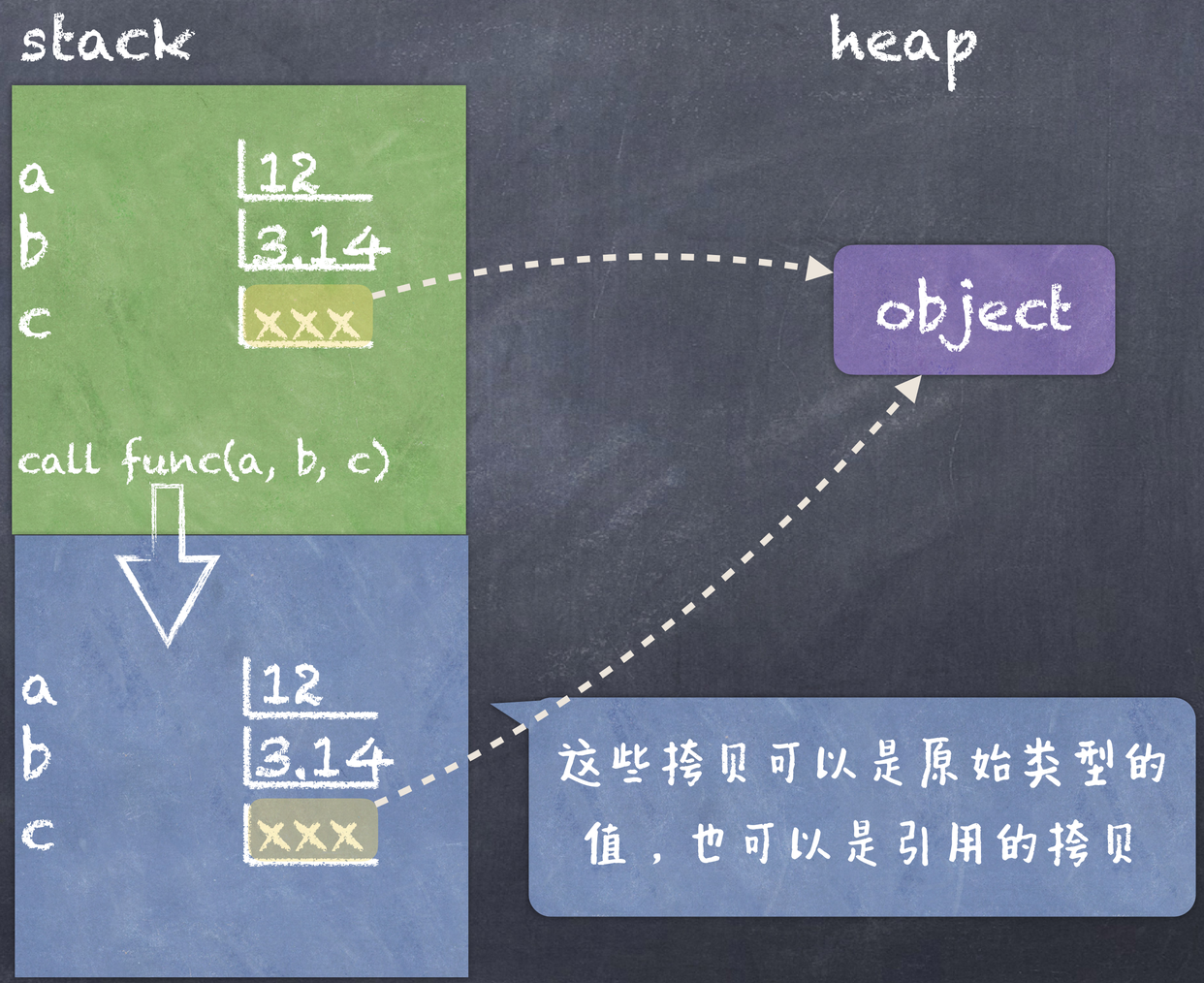
method body

return statement

为什么函数签名不包含返回值?
避免调用时的不确定性

# 值传递（Pass by value）

- 在Java中，方法的实参是通过值传递的。当方法调用时，实参的值的拷贝会赋给方法中的参数变量.
- 方法中可以对该拷贝值做出改变，但该改变不会影响到原来的变量。



```
public class PassBy {
  public void tryPrimitives(int i, double f, char c, boolean test)
  {
    i += 10;     //This is legal, but the new values
    c = 'z';     //won't be seen outside tryPrimitives.
    f -= 3.0;
    if(test)
      test = false;
    else
      test = true;
  }

  public static void main(String[] args) {
        PassBy p = new PassBy();
        int ii = 1;
        double ff = 1.0;
        char cc = 'a';
        boolean bb = false;
        p.tryPrimitives(ii, ff, cc, bb);
        System.out.println("ii = " + ii + ", ff = " + ff +
                           ", cc = " + cc + ", bb = " + bb);

  }
}
```

原始数据的值的拷贝传进被调用方法内（callee），callee对其改变不会影响调用方法（caller）中变量的值

```
class Record
{
  int num;
  String name;
}

public class PassBy {
  public void tryObject(Record r)
  {
    r = new Record();
    r.num = 100;
    r.name = "Fred";
  }
  public static void main(String[] args) {
        PassBy p = new PassBy();
        Record id = new Record();
        id.num = 2;
        id.name = "Barney";
        p.tryObject(id);
        System.out.println(id.name + "  " + id.num);
  }
}
```

对象的引用的拷贝（可以认为就是该对象的内存中地址的拷贝）传进被调用方法内（callee），callee对其重新赋值不会影响调用方法（caller）中变量的值

```
class Record
{
  int num;
  String name;
}

public class PassBy {
  public void tryObject(Record r)
  {
    r.num = 100;
    r.name = "Fred";
  }
  public static void main(String[] args) {
        PassBy p = new PassBy();
        Record id = new Record();
        id.num = 2;
        id.name = "Barney";
        p.tryObject(id);
        System.out.println(id.name + "  " + id.num);
  }
}
```

但是通过传递的引用可以更改其指向的对象的内容，该改变会反映到原来的引用上

It is often not good programming style to change the values of instance variables outside the object.

Normally, the object should have a method (mutator) to set the values of its instance variables !

# 参数传递

- Python中的方法中参数是什么传递？

官方文档：赋值传递（Pass by assignment）

实际上，和Java的值传递语义一致，只不过python中没有原始类型而已

# 构造方法

- 对象的实例化通过构造方法（即构造子Constructor）来实现（当使用 new 语句时自动调用，不能显式地调用）
- 构造方法的名字与类名相同
- 构造方法没有返回值
- 构造方法可以有多个，构成方法可以重载
- 当没有声明构造函数时，默认含有一个无参数的构造函数，但当显示地声明了构造函数之后，该默认的构造函数就不存在。

# 构造方法例子

```
class Main {

  int a;
  boolean b;
  public static void main(String[] args) {

    // A default constructor is called
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

等价于

```
class Main {

  int a;
  boolean b;

  Main() {
    a = 0;
    b = false;
  }

  public static void main(String[] args) {
    // call the constructor
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

与类名相同，没有返回值

通过new来隐式地调用构造子

# 构造方法例子

```java
class Main {

  String language;

  // constructor with no parameter
  Main() {
    this.language = "Java";
  }

  // constructor with a single parameter
  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    // call constructor with no parameter
    Main obj1 = new Main();

    // call constructor with a single parameter
    Main obj2 = new Main("Python");

    obj1.getName();
    obj2.getName();
  }
}
```

多个构造方法即重载了构造方法）

会根据参数形式来选择具体的构造方法

```java
class Main {

  String language;

  Main(String language) {
    this.language = language;
  }

  public void getName() {
    System.out.println("Programming Langauage: " + this.language);
  }

  public static void main(String[] args) {

    Main obj1 = new Main();
    obj1.getName();
  }
}
```

⚠error! 声明了构造方法后，默认的无参数构造子就不存在了

# 构造函数调用其他构造函数

```
int sum;
// first constructor
Main() {
  // calling the second constructor
  this(5, 2);
}
// second constructor
Main(int arg1, int arg2) {
  // add two value
  this.sum = arg1 + arg2;
}
void display() {
  System.out.println("Sum is: " + sum);
}
// main class
public static void main(String[] args) {
  // call the first constructor
  Main obj = new Main();
  // call display method
  obj.display();
}
}
```

利用this(...)语句，⚠️注意这里和调用普通函数或者数据属性的dot expression不同（即this.XXX）

隐式地调用了第二个构造函数

# 标识符（Identifier）名字规则和约定

- 标识符：标识类、变量、常量和方法的名字，由字母（A~Z、a~z）、特殊符号（$、_）和数字（0~9）构成，区分大小写，且名字的第一个符号不能为数字，标识符不能是Java关键字
  - 类：一般首字母大写
  - 变量：一般小写开始（中间不同的单词的开头大写，如getCurrentValue）
  - 常量：一般全部大写
  - 方法：和变量一致

# static关键字

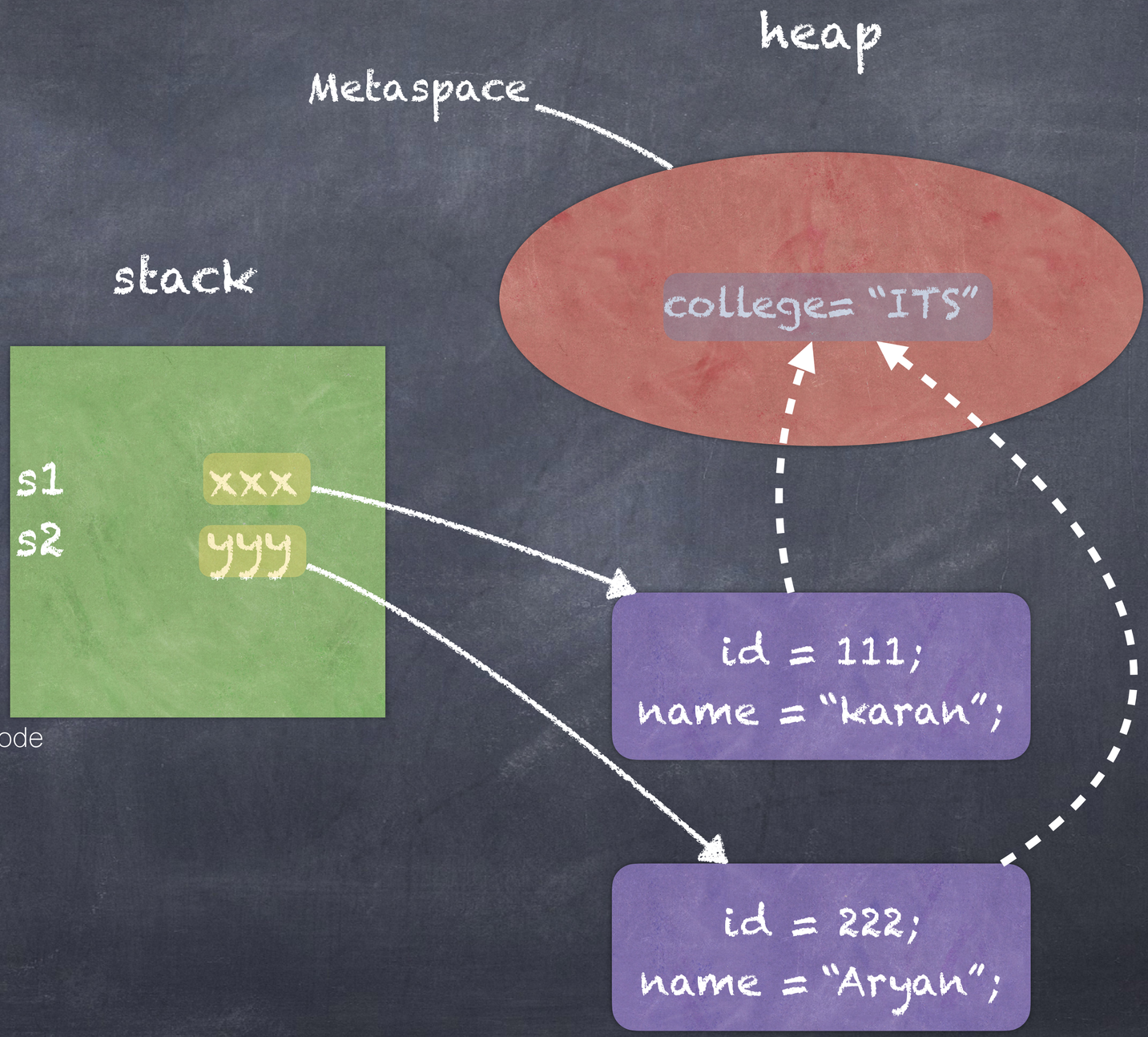
- static是Java中用来表达隶属于"类"本身，而不隶属于类的对象的一个关键字
- static可以修饰：
  - 变量（该变量即为类变量或静态变量）
  - 方法（该方法即为类方法或静态方法）
  - 语句块（用来初始化类变量）

# 静态变量

- 静态变量用来表达**所有**对象的共有属性
- 静态变量在JVM对类加载之后就得到了相应的内存，其只有一份，不会随着对象的创建增多。

```
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}

public class TestStaticVariable1{
    public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //we can change the college of all objects by the single line of code
    //Student.college="BBDIT";
    s1.display();
    s2.display();
  }
}
```



# 静态方法

- 静态方法隶属于类，可以直接用类名来调用，而无需实例化一个对象来调用
- 静态方法可以访问和更改静态变量，但不能直接访问非静态的成员（变量和方法）
- 在静态方法中，this（表示当前对象的引用）和super（表示当前对象的相应的父类对象引用）无法使用

# 靜態方法

```
class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Compile Time Error

# 静态语句块

- 静态语句块用来初始化静态变量
- 静态语句块不能访问非静态成员
- 静态语句块在 main方法之前调用

```
class A1{
    static int a;
    static int b;
    static{
        a = 1;
        b = 2;
    }
    public static void main(String args[]){
        System.out.println("hello");
    }
}
```

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
```

Output: static block is invoked
Hello main

# 继承

# 继承

- Java通过关键字extends来表达继承关系

```
class SubClass extends SuperClassName {
  ...
}
```

- 子类可调用父类的方法和变量（所以继承破坏了封装性），子类可增加父类中没有的方法和变量
- Java只支持单继承，即只有一个"直接"父类。父类的父类也是该子类的父类，但不是直接父类。Java中所有类都是java.lang.Object 的子类。

# 例子

```
class Vehicle {
  String brand;
  public void setB(String s) { brand = s; }
  public void showB() { System.out.println(brand); }
}
class Bus extends Vehicle {
  int gas;
  public void setG(int g) { gas = g; }
  public void showG() { System.out.println(gas); }
  public static void main(String[] args){
      Bus b = new Bus();
      b.setB("audi");
      b.setG(100);
      b.showB();
      b.showG();
  }
}
```

继承了相应的属性，所以可以直接使用

# 重新定义变量例子

```
class A {
    private int m = 0;
    int i=256, j =64;
    static int k = 32;
    final float e = 2.718f;
}
public class B extends A {
    public char j='x';
    final double k =5;
    static int e =321;
    void show() { System.out.println(i + " " + j + " " +  k + " " + e );  }
    void showA() { System.out.println(super.j + " " + super.k + " " + super.e); }
    public static void main(String[] args) {
        B b = new B();
        b.show();
        b.showA();
    }
}
```

Private不能继承

重新定义父类中的变量，父类中相应的变量被隐藏（hidden）

可以通过super关键词来访问这些隐藏的变量，当然static的可以通过类名来访问

# 重新定义方法例子

```
class Vehicle {
    String brand;
    public void setB(String s) { brand = s; }
    public void showB() { System.out.println(brand); }
}

class Bus extends Vehicle {
    public void setB(String g) {System.out.println("sub set"); brand = g; }
    public void showB() {System.out.println("sub show");  System.out.println(brand); }
    public static void main(String[] args){
        Bus b = new Bus();
        b.setB("audi");
        b.showB();
        }
}
```

签名相同的方法，重新定义就是方法重写 (Overriding)

# 隐藏和重写（Override）的区别

- 重写对应运行时，Java会在运行时判断哪个方法会被调用（It is for non-static methods.）
- 隐藏对应编译时，Java在编译阶段就已经确定好了调用的对象（即静态和实例变量、静态方法）

# 隐藏和重写的区别

```
public class Animal {
    public String name = "animal";
    public static void something() {
        System.out.println("animal.something");
    }
    public void eat() {
        System.out.println("animal.eat");
    }
    public Animal() {
    }
}
```

```
public class Dog extends Animal {
    public String name = "dog";
    public static void something() {
        // This method merely hides Animal.something(),
        // but does not override it
        System.out.println("dog.something");
    }
    public void eat() {
        // This method overrides eat(),
        // and will affect calls to eat()
        System.out.println("dog.eat");
    }
    public Dog() {
    }
    public static void main(String[] args) {
        Animal  animal = new Dog();
        System.out.println(animal.name);
        animal.something();
        animal.eat();
    }
}
```

父子类中签名相同的函数，Static 和non-static必须一致

Upcasting

子类型多态

Output:

animal

animal.something

dog.eat

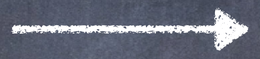
# 父类的构造方法

- 在Java中，任何类的构造方法，第一行语句必须是调用父类的构造方法。
- 如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句super();
- 构造方法无法继承，它隶属于特定的类。因此，如果即使类没有写构造方法，那么其也会有一个默认的构造方法，而不是继承于父类的构造方法

# 例子

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

隐式地调用super()



Output:

Art Constructor
Drawing Constructor
Cartoon Constructor

子类的构造法的第一句必须调用父类的构造方法

# 例子

```
class Game {
    Game(int i) {
        System.out.println("Game Constructor");
    }
}
class BoardGame extends Game {
    BoardGame(int i) {
        System.out.println("BoardGame Constructor");
    }
}
public class Chess extends BoardGame {
    Chess() {
        super(3);
            System.out.println("Cartoon Constructor");
    }
    public static void main(String args[]) {
        Chess c = new Chess();
    }
}
```

Compile error! Why?

# 类成员访问修饰符与继承的关系

- 私有的（private）类成员不能被子类继承
- 构造方法不被继承
- 公共的（public）和保护性的（protected）类成员能被子类继承，且子类和父类可以属于不同的包
- 无修饰的父类成员，仅在同一个包中才能被子类继承

# 类成员访问修饰符与继承的关系

- 一些注意点⚠️：
  - 重写可继承的函数时，其访问权限不能比父类中被重写的方法访问权限更低
    - 比如，如果父类的一个方法访问修饰符是protected，那么子类重写时可变为public、也可不变，但不能为private 或缺省。
  - 重写可继承的函数时，返回值类型如果是原始类型，必须一致，如果是对象类型，那么可以不一致（但必须是原函数返回值的子类）
  - 对于隐藏而言，则没有限制（你可以认为隐藏本质上其实和原来的是两种不同的成员，因为在编译阶段就已经分开了）

# 函数重载（Overloading）

- 方法的重载是指方法带有不同的参数，但使用相同的名字。
  - 一般方法的参数不同则表示实现不同的功能，但功能相似。
- 所谓参数不同是指：参数个数不同 或 参数类型不同 或参数的顺序不同。
- 函数重载即特设多态（Ad hoc polymorphism）

# 例子

```
double sum(double a, double b){
    ...
}

double sum(double a, double b, double c){
    ...
}

float sum(float a, float b){
    ...
}
```

Same method name

different types or numbers of parameters

⚠重载是定义多个函数，这些函数名一样，参数不同，而重写(也叫覆盖)是重新定义父类中签名相同的函数。重载是特设多态，重写是子类型多态。重载在函数调用时所调用的具体函数（函数绑定）在编译时确定（early binding），重写的函数绑定在运行时（late binding）

# instanceof

- 子类的每个对象也是父类的对象
- 可以直接用子类对象赋给一个父类变量

```
SuperClass variable = new SubClass();
```

- 这种转化叫作：向上转型（upcasting）

# instanceof

- 但父类对象不能直接赋给子类

```
public class Animal {
   ...
}

public class Dog extends Animal {
      ...
      public static void main(String[] args) {
         Dog   animal = new Animal();
      }
   }
```

incompatible types

向下转型

downcasting

```
public class Animal {
   ...
}

public class Dog extends Animal {
      ...
      public static void main(String[] args) {
         Dog   animal = (Dog) new Animal();
      }
   }
```

explicit conversion

class Animal cannot be cast to class Dog

# instanceof

```
public class Dog extends Animal {
    ...
    public static void main(String[] args) {
      Animal animal = new Dog();
      Dog  dog = (Dog) animal;
    }
}
```

Correct

- 想要安全地进行向下转型时，一般需要用运算符instanceof来进行判断

```
if (animal instanceof Dog)
    Dog  dog = (Dog) animal;
```

dog instanceof Animal 也为true， 即子类的对象即是其父类的对象

# 抽象类

# 抽象类（abstract class）

- 如果一个类中没有包含足够的信息来描绘一个具体的对象（方法没实现），这样的类就是抽象类。
- 并不能直接由抽象类创建对象，只能通过抽象类派生出新的子类，再由其子类来创建对象。
  - 也就是说，抽象类就是不能用new运算符来创建实例对象的类

# 抽象类

- 抽象类的定义：

```
abstract class classname
{
    ... // other declarations
    abstract  type methodname (parameters);
 }
```

抽象方法，在抽象方法里，不能定义方法体。只需声明不需实现

# 继承抽象类

- 抽象类的子类可以是抽象类
- 如果子类不是抽象类，那么其必须实现父类中的所有抽象方法，即抽象方法必须被子类的方法所覆盖。
  - 抽象类某种意义上限制了子类的行为，因此，抽象类有点类似"模板"的作用，其目的是根据它的格式来创建和修改新的类。

# 例子

```
abstract public class Shape {
    protected String name;
    public Shape(String xm){
        name = xm;
        System.out.println("名称 ： " + name);
    }
    abstract public double getArea();  //面积
    abstract public double getLength(); //周长
}
```

```
public class Circle extends Shape{
    private final static double PI = 3.14;
    private double radius;

    public Circle(String xm, double r) {
        super(xm);
        this.radius = r;
    }

    public double getArea() {
        return PI*radius*radius;
    }

    public double getLength() {
        return 2*PI*radius;
    }
}
```

必须实现，否则Circle必须要声明为Abstract 类

# 一些抽象类注意点

- 由于抽象类是需要被继承的，所以abstract类不能用final来修饰。也就说，一个类不能既是最终类，又是抽象类，即关键字abstract与final不能合用。
- 抽象类中不一定包含抽象方法，但包含抽象方法的类一定要声明为抽象类。

# 接口

# 接口（Interface）

- 抽象类可以作为子类的模版。但由于Java的单继承特性，使得一个子类只有一个直接父类，无法具备多个模版
- 此外，继承也会有很多问题，如果滥用继承，会导致继承层次过深。而复杂的继承关系会导致维护性降低。
- Java中的接口提供了多个模版的可能
  - 可以看成抽象方法的集合。一个类可以实现多个接口

# 接口

```
[public] interface interfaceName [extends List-of-super-interfaceNames]
  {
     [public][static][final] type variableName = constantValue;
     ......
     [public][abstract] type methodName (parameterList);
     ......
    [public] static type methodName(parameterList)
    {
        //method body
    }
    ......
   [public] default type methodName(parameterList)
    {
        //method body
   }
   ......
}
```

# 接口的一些注意点

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口只有三种类型的方法（抽象方法（缺省）、静态方法、default 方法）。
- 接口的变量都是 static final 修饰的（静态常量），缺省的也是静态常量。
- 接口的成员可访问性都是public（缺省也是public）。

# 例子

```
public interface IShape {
    public static final double PI = 3.14;
    double getArea();
    public abstract double getLength();
    public static void showPI(){
        System.out.println(PI);
    }
    public default void getInfo(){
        System.out.println("这是一个图形");
    }
}
```

# 接口的实现与引用

- 接口的实现：利用接口的特性来建造类的过程，类似继承，但是关键词为`implements`
- 一个类可以实现多个接口，每个接口用逗号隔开

```
class  className  implements  List-of-interfaceNames
 {
    ......
 }
```

# 接口的实现与引用

- 一个类必须实现接口的所有抽象方法（除非该类是抽象类）
- 类实现的所有抽象方法都必须声明public
- 接口可以作为一种引用类型使用，可以声明接口类型的变量或数组，并用它来访问实现该接口的类的对象。

why

# 例子

```
public interface IShape {
    public static final double PI = 3.14;
    double getArea();
    public abstract double getLength();
    public static void showPI(){
        System.out.println(PI);
    }
    public default void getInfo(){
        System.out.println("这是一个图形");
    }
}
```

```
public class CireCleForI implements IShape{
    double radius;
    public CireCleForI(double r){
        this.radius = r;
    }
    public double getArea() {
        return PI*radius*radius;
    }
    public double getLength() {
        return 2*PI*radius;
    }
}
```

接口作为变量类型

```
public class TestIShape {

    public static void main(String[] args){
        IShape cir = new CireCleForI(2.2);
        System.out.println("面积 :  "+ cir.getArea());
        System.out.println("周长 :  "+ cir.getLength());
        cir.getInfo();
        IShape.showPI();
    }

}
```

对象调用方法（默认方法、抽象方法）

静态方法必须接口名访问

# 接口继承

- 接口可通过extends关键字声明该新接口是某个已存在的父接口的子接口，它将继承父接口的所有变量与方法（静态方法除外，静态方法只能通过接口名来访问）。
- 接口支持多继承（一个接口可以继承多个接口，接口不可以继承类，多个父接口用逗号隔开）。
- 如果接口中定义了与父接口同名的常量或相同的方法，则父接口中的常量和静态方法被隐藏，默认方法和抽象方法被重写。

# 例子

```
public interface Face1 {
    static final double PI = 3.14;
    abstract double area();
}
public interface Face2 {
    default void setColor(String c){
        System.out.println("颜色是 ：  "+ c);
    }
    abstract void volume();
}
public interface Face3 extends Face1, Face2 {
    static final double PI = 3.1415;
    public default void setColor(String c){
        System.out.println("颜色是 ：  "+ c + "3");
    }
}
```

多继承

隐藏父接口变量

重写默认方法

# 接口实现混入（Mixin）

- 利用default方法实现混入

```
public interface Flyable {
  default void fly() {
    System.out.println("I can fly!");
  }
}

public interface Swimmable {
  default void swim() {
    System.out.println("I can swim!");
  }
}
```

Cannot instantiate

It is able to fly and swim

```
public class Duck implements Flyable, Swimmable {

}
```

# 接口多继承中的名字冲突问题

- 接口的多重继承中可能存在常量名或方法名重复的问题，即名字冲突问题
- 对于常量，若名称不冲突，子接口可以继承多个父接口中的常量，但如果多个父接口中有同名的常量，则必须通过 接口名.常量名 区分。
- 对于多个父接口中存在同名的方法包含默认方法时，也会发生命名冲突，这时不能通过 接口名.默认方法名 来解决
  - 必须通过在当前类中定义一个同名的方法才行

# 例子

```
public interface Face1 {
    static final double PI = 3.14;
    abstract double area();
    default void setColor(String c) {}
}

public interface Face2 {
    static final double PI = 3.1415;
    default void setColor(String c){
        System.out.println("颜色是 ： "+ c);
    }
    abstract void volume();
}
```

Face3 inherits unrelated defaults for setColor(String) from types Face1 and Face2

```
/** error */
public interface Face3 extends Face1, Face2 {
}
```

重新定义该方法，可以是默认方法，也可以是抽象方法

```
/** correct */
public interface Face3 extends Face1, Face2 {
    default void setColor(String c){
        System.out.println(Face2.PI);
        System.out.println("颜色是 ： "+ c);
    }
}
```

变量名只要加接口名即可

```
/** correct */
public interface Face3 extends Face1, Face2 {
    default void setColor(String c){
        Face2.super.setColor(c);
    }
}
```

通过 接口名.super.方法名 访问父接口的方法

# 如果发生了继承类和实现接口的命名冲突

- 那么"类"优先，继承的父方法中的同名方法

```
public class SimpleSetColor {
    protected double x;
    public void setColor(String c){
        System.out.println("颜色是Simple：  "+ c);
    }
    protected String set(){return "0";}
}

public interface Face2 {
    static final double PI = 3.12;
    default void setColor(String c){
        System.out.println("颜色是：  "+ c);
        System.out.println(this.PI);
    }
    abstract void volume();
}
```

```
public class Cylinder extends SimpleSetColor implements Face2 {

    private double radius;
    private int height;
    protected String color;
    private boolean x;

    public Cylinder( double r,int  h){
        this.radius = r;
        this.height = h;
    }
    public double area() {
        return Face1.PI*radius*radius;
    }
    public void volume() {
        System.out.println("体积为：  " + area()*height);
    }
    public static void main(String[] args){
        Cylinder c = new Cylinder(3.0, 2);
        c.setColor("红色");
        c.volume();
    }
    public String set(){return "1.0";}
}
```

这里会调用SimpleSetColor的方法

# Any questions ?