10 面向对象编程II：封装

一、封装.

类有成员变量和成员方法.

Example:

- Part of GPS system
- Latitude and Longitude represent a Position
- Wanna get the distance and direction between two position

- How to design one or two class to implement this use case?

设计1A:

```
public class Position
{
  public double latitude;
  public double longitude;
}
```

```
public class PositionUtility
{
  public static double distance( Position position1, Position position2 )
  {
    // Calculate and return the distance between the specified positions.
  }
  public static double heading( Position position1, Position position2 )
  {
    // Calculate and return the heading from position1 to position2.
  }
}
```

设计1B:

```
public class Position {
    double latitude;
    double longitude;

    public static double calculateDistance(double x1, double y1, double x2, double y2) {

    }
    public static double calculateDirection(double x1, double y1, double x2, double y2){

    }
}
```

设计1C:

```
public class Position {
    double latitude;
    double longitude;

    public double getDistance(double x2, double y2){

    }
    public double getDirection(double x2, double y2){

    }
}
```

设计1D:

```
public class Position {
    double x1,x2,y1,y2;

    public double calculateDistance(){

    }
    public double calculateDirection(){

    }
}
```

设计 2：

```
public class Position
{
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to
    // position.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to
    // position.
  }
  public double latitude;
  public double longitude;
}
```

优劣：

- Pros
  - The call's semantics clearly indicate that the direction proceeds from my house to the coffee shop.
  - Currying effectively specializes the function on its first argument, resulting in clearer semantics.
- Cons

设计 3：成员变量私有化（提供 getter 和 setter）（防御式编程）

```
public class Position
{
  public Position( double latitude, double longitude )
  {
    setLatitude( latitude );
    setLongitude( longitude );
  }
  public void setLatitude( double latitude )
  {
    // Ensure -90 <= latitude <= 90 using modulo arithmetic.
    // Code not shown.
    // Then set instance variable.
    this.latitude = latitude;
  }
  public void setLongitude( double longitude )
  {
    // Ensure -180 < longitude <= 180 using modulo arithmeti
    // Code not shown.
    // Then set instance variable.
    this.longitude = longitude;
  }
```

```
  public double getLatitude()
  {
    return latitude;
  }
  public double getLongitude()
  {
    return longitude;
  }
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to
    // position.
    // Code not shown.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to
    // position.
  }
  private double latitude;
  private double longitude;
}
```

优劣：

- Pros
  - Defensive Programming
  - Isolating the decision
- Cons

设计4：

```
public class Position
{
  public Position( double latitude, double longitude )
  {
    setLatitude( latitude );
    setLongitude( longitude );
    // Default to plane geometry and kilometers
    geometry = new PlaneGeometry();
    units = new Kilometers();
  }
  public void setLatitude( double latitude )
  {
    setPhi( Math.toRadians( latitude ) );
  }
  public void setLongitude( double longitude )
  {
    setTheta( Math.toRadians( longitude ) );
  }
  public void setPhi( double phi )
  {
    // Ensure -pi/2 <= phi <= pi/2 using modulo arithmetic.
    // Code not shown.
    this.phi = phi;
  }
  public void setTheta( double theta )
  {
    // Ensure -pi < theta <= pi using modulo arithmetic.
    // Code not shown.
    this.theta = theta;
  }
  // Setters for geometry and units not shown
  public double getLatitude()
  {
    return( Math.toDegrees( phi ) );
  }
  public double getLongitude()
  {
    return( Math.toDegrees( theta ) );
  }
  // Getters for geometry and units not shown
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to
    // position using the current geometry and units.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to
    // position using the current geometry and units.
  }
  private double phi;
  private double theta;
  private Geometry geometry;
  private Units units;
}
```

优劣：

- Pros
  - Isolating potential change

设计的完备性：

Example：一个只能加水而不能倒水的杯子

封装规则：

- **Encapsulation rule 1:**
  - **Place data and the operations that perform on that data in the same class**
- **Encapsulation rule 2:**
  - **Use responsibility-driven design to determine the grouping of data and operations into classes**
- **Encapsulation rule 3:**
  - **The responsibility should be complete**

二. 类的职责与封装

1. 数据职责

- 表征对象的本质特征
- 行为（计算）所需要的数据
  - 教务系统中学生对象：计算年龄
  - 税务系统中纳税人：计算所得税

2. 行为职责

- 表征对象的本质行为
- 拥有数据所应该体现的行为
  - 出生年月
  - 个人收入

数据职责和行为职责"在一起".

3. static 关键字.

- Java is object-oriented, but there is a special case where there is no need to have an instance of the class.
- The keyword static let a method run without any instance of the class.
- A static method means "behavior not dependent on an instance variable, so no instance/object is required. Just the class."

static 方法中不能调用非静态变量和方法.

Static variable:

value is the same for ALL instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
   int duckCount = 0;
    public Duck() {
       duckCount++;
   }
}
```

this would always set duckCount to 1 each time a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgey. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance*.

```
public class Duck {

   private int size;
   private static int duckCount = 0;

   public Duck() {
       duckCount++;
   }

   public void setSize(int s) {
      size = s;
   }
   public int getSize() {
      return size;
   }
}
```

The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.

Now it will keep incrementing each time the Duck constructor runs, because duckCount is static and won't be reset to 0.

static 变量只有一份拷贝

初始化静态变量：

- Static variables are initialized when a class is loaded.
- A class is loaded because the JVM decides it's time to load it.
- And there are two guarantees about static initialization:
  - Static variables in a class are initialized before any object of that class can be created.
  - Static variables in a class are initialized before any static method of the class runs.

4. final关键字：一旦初始化就不再改变。(声明时必须初始化)

```
public static final double PI = 3.141592653589793;
```

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one.

Constant variable names should be in all caps!

- A variable marked final means that – once initialized - it can never change.

non-static final variables

```
class Foof {
    final int size = 3; <-- now you can't change size
    final int whuffie;

    Foof() {
        whuffie = 42; <-- now you can't change whuffie
    }

    void doStuff(final int x) {
        // you can't change x
    }

    void doMore() {
        final int z = 7;
        // you can't change z
    }
}
```

final method

```
class Poof {
    final void calcWhuffie() {
        // important things
        // that must never be overridden
    }
}
```

final class

```
final class MyMostPerfectClass {
    // cannot be extended
}
```

A final **variable** means you can't **change** its value.

A final **method** means you can't **override** the method.

A final **class** means you can't **extend** the class (i.e. you can't make a subclass).

空白final：

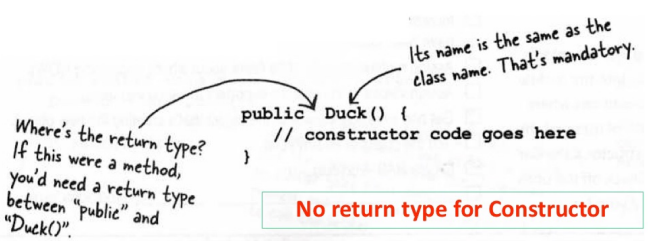
```
class Poppet {
  private int i;
  Poppet(int ii) {
    i = ii;
  }
}
public class BlankFinal {
  private final int i = 0; // Initialized final
  private final int j; // Blank final
  private final Poppet p; // Blank final reference

  // Blank finals MUST be initialized in the constructor:
  public BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet(1); // Initialize blank final reference
  }
  public BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet(x); // Initialize blank final reference
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
} ///:~
```

- 使用**final** 方法的原因有两个。
  - 第一个原因是把方法锁定，以预防任何继承类修改它的意义。这是出于设计的考虑：你想要确保在继承中方法行为保持不变，并且不会被重载。
  - 使用**final**方法的第二个原因是效率。如果你将一个方法指明为**final**，就是同意编译器将针对该方法的所有调用都转为内嵌（inline）调用。

5. 类的构造方法.

- The key feature of a constructor is that it runs before the object can be assigned to a reference.
- The constructor gives you a chance to step into the middle of new.



三.对象

1.对象初始化

- The variable are initialized before any methods can be called, even the constructor;
- Static data initialized first, then the non-static;
- Static data initialized and block will be executed in text order.

初始化顺序示例：

```
public class StaticOrder{
    public static int X = 20;
    public static int Y = 2 * X;
    static{
        X = 30;
    }
    public static void main(String[] args){
        System.out.println(Y);          //输出40；
    }
```

2.垃圾回收机制

- Java的垃圾回收器要负责完成3件任务：
  - 分配内存
  - 确保被引用的对象的内存不被错误回收
  - 回收不再被引用的对象的内存空间。
- 垃圾回收是一个复杂而且耗时的操作。如果JVM花费过多的时间在垃圾回收上，则势必会影响应用的运行性能。一般情况下，当垃圾回收器在进行回收操作的时候，整个应用的执行是被暂时中止（stop-the-world）的。这是因为垃圾回收器需要更新应用中所有对象引用的实际内存地址。不同的硬件平台所能支持的垃圾回收方式也不同。比如在多CPU的平台上，就可以通过并行的方式来回收垃圾。而单CPU平台则只能串行进行。不同的应用所期望的垃圾回收方式也会有所不同。服务器端应用可能希望在应用的整个运行时间中，花在垃圾回收上的时间总数越小越好。而对于与用户交互的应用来说，则可能希望所垃圾回收所带来的应用停顿的时间间隔越小越好。对于这种情况，JVM中提供了多种垃圾回收方法以及对应的性能调优参数，应用可以根据需要来进行定制。