10 - 面向对象编程 II
封装

刘 钦

南京大学软件学院

知识点

- 封装的原则
- static变量
- static方法
- static变量的初始化
- final关键字
- 构造方法的重载
- 对象的初步初始化
- 垃圾回收机制基本原理

Outline

- 封装
- 类
- 对象

封装

Use case

- 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?

```java
public class PositionA {
    public double latitude;
    public double longitude;
}
class PositionUtility{
    public static double distance(PositionA position1, PositionA position2){
        //Calculate and return the distance between the specified positions.
        return 0.0;
    }
    public static double heading(PositionA position1, PositionA position2){
        //Calculate and return the heading from position1 to position2
        return 0.0;
    }
}
```

Design A

```java
public static void main(String[] args){
    PositionA p1 = new PositionA();
    p1.latitude = 111;
    p1.longitude = 35;
    PositionA p2 = new PositionA();
    p2.latitude = 108;
    p2.longitude = 36;
    double d = PositionUtility.distance(p1,p2);
    double h = PositionUtility.heading(p1,p2);
}
```

Design A

Problems for Design A

- Welcome to 1972!
- Fortran excitedly used the new International Mathematics and Statistics Library (IMSL) in just this manner

Design B

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

    public static double calculateDistance(double x1, double y1, double x2, double y2) {
        return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));

    }
    public static double calculateDirection(double x1, double y1, double x2, double y2){
        return Math.atan2(y2-y1,x2-x1);
    }

    public static void main(String[] args){
        double d = calculateDistance(111,35,108,36);
        double h = calculateDirection(111,35,108,36);
    }
}
```

Design C

```java
public class PositionC {
    double latitude;
    double longitude;

    public double getDistance(double x2, double y2){
        return Math.sqrt(Math.pow(x2 - latitude, 2) + Math.pow(y2 - longitude, 2));

    }
    public double getDirection(double x2, double y2){
        return Math.atan2(y2-longitude,x2-latitude);
    }

    public static void main(String[] args){
        PositionC p1 = new PositionC();
        p1.latitude = 111;
        p1.longitude = 35;
        double d = p1.getDistance(108,36);
        double h = p1.getDirection(108,36);
    }
}
```

Design D

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

    public double calculateDistance(){
        return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));
    }
    public double calculateDirection(){
        return Math.atan2(y2-y1,x2-x1);
    }
    public static void main(String[] args){
        PositionD p1 = new PositionD();
        p1.x1 = 111;
        p1.y1 = 35;
        p1.x2 = 108;
        p1.y2 = 36;
        double d = p1.calculateDistance();
        double h = p1.calculateDirection();
    }
}
```

Design I

```java
public class Position{

    public double longitude;
    public double latitude;

    public Position(double lo, double la){
        longitude = lo;
        latitude = la;
    }

    public double distance(Position p){
       return 0.0;
    }
    public double direction(Position p){
        return 0.0;
    }

    public static void main(String[] args){
        Position p1 = new Position(111,35);
        Position p2 = new Position(108,36);
        double d = p1.distance(p2);
        double h = p1.direction(p2);
    }
}
```

Pros and Cons

- 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

Reuse（重用）

- Position类的重用
    - GPS系统中，各个点会构成一个图的拓扑结构，表达交通路线。

```java
public class PositionII{
    private double longitude;
    private double latitude;

    public PositionII(double longitude, double
latitude){
        this.longitude = longitude;
        this.latitude = latitude;
    }
    public double getLongtitude(){
        return longitude;
    }
    public double getlatitude(){
        return latitude;
    }
    public void setLongtitude(double d){
        if(d < 0)
            longitude = 0;
        else if(d > 180)
            longitude = 180;
        else
            longitude = d;
    }
    public void setlatitude(double d){
        if(d < 0)
            latitude = 0;
        else if(d > 90)
            latitude = 90;
        else
            latitude = d;
    }

    public double distance(PositionII p){
        return 0.0;
    }
    public double direction(PositionII p){
        return 0.0;
    }

    public static void main(String[] args){
        PositionII p1 = new PositionII(111,35);
        PositionII p2 = new PositionII(108,36);


        double d = p1.distance(p2);
        double h = p1.direction(p2);
    }
}
```

Design II

Pros and Cons

- Pros
  - Defensive Programming
  - Isolating the decision
- Cons

Requirement Change（变更）

- Unit
  - Kilometers
  - NauticalMiles
  - StatueMiles
  - Radians
- Three implementations for geometry
  - PlaneGeometry
  - SphericalGeometry
  - EllipticalGeometry

Design III

```
public class PositionIII {
    private double phi;
    private double theta;
    private Geometry geometry;
    private Units units;

    public PositionIII(double latitude, double longitude){
        this.phi = latitude;
        this.theta = longitude;
        this.geometry = new Geometry();
        this.units = new Units();
    }

    public void setLatitude(double latitude){
        setPhi(Math.toRadians(latitude));
    }
    public void setLongitude(double longitude){
        setTheta(Math.toRadians(longitude));
    }
    public void setPhi(double phi){
        if(phi < -0.5*Math.PI)
            this.phi = -0.5*Math.PI;
        else if(phi > 0.5*Math.PI)
            this.phi = 0.5*Math.PI;
        else
            this.phi = phi;
    }

    public void setTheta(double theta){
        if(theta < -Math.PI)
            this.theta = -Math.PI;
        else if(theta > Math.PI)
            this.theta = Math.PI;
        else
            this.theta = theta;
    }
    public double getLatitude(){
        return Math.toDegrees(this.phi);
    }
    public double getLongitude(){
        return Math.toDegrees(this.theta);
    }
    public double getDistance(PositionIII p){
        return units.toKilos(geometry.getDistance(this.phi, this.theta,
p.phi, p.theta));
    }
    public double getDirection(PositionIII p){
        return geometry.getDirection(this.phi, this.theta, p.phi,
p.theta);
    }
    public static void main(String[] args){
        PositionIII p1 = new PositionIII(111,35);
        PositionIII p2 = new PositionIII(108,36);
        double d = p1.getDistance(p2);
        double h = p1.getDirection(p2);
    }

}
class Geometry{
    public double getDistance(double x1, double y1,double x2, double y2){
        return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));
    }
    public double getDirection(double x1, double y1,double x2, double y2)
{
        return Math.atan2(y2-y1,x2-x1);
    }
}
class Units{
    public double toMiles(double d){
        return d*0.621371;
    }
    public double toKilos(double d){
        return d*1.60934;
    }
}
```

Pros and Cons

- Pros
    - Isolating potential change

完备性

- 例子
  - 一个只能加水而不能倒水的杯子
  - 只能入库，不能出库的仓库

Encapsulation

- 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

类的职责与封装

- 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
- 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

数据职责与行为职责“在一起”

类

Outline

- 封装
- 类
    - static
    - final
    - 类的构造方法
- 对象

Class

- 类
    - 成员变量
    - 成员方法
- class Dog{
    - int size;
    - Dog();
    - Dog(int s);
    - void bark();
- }

Class

- 对象的创建
  - Dog one ＝ new Dog(70);
- 属性的访问
  - one.size = 70;
- 方法的访问
  - one.bark();

类三部曲

- 3 steps of object declaration, creation and assignment
- Duck myDuck = new Duck();

Math methods

- Math mathObject = new Math();

- Error : Math() has private access in java.lang.Math

- There's no global thing in Java.
- Math methods never use instance variables.

- int x = Math.round(42.2f);
- int y = Math.min(56,12);
- int z = Math.abs(-343);

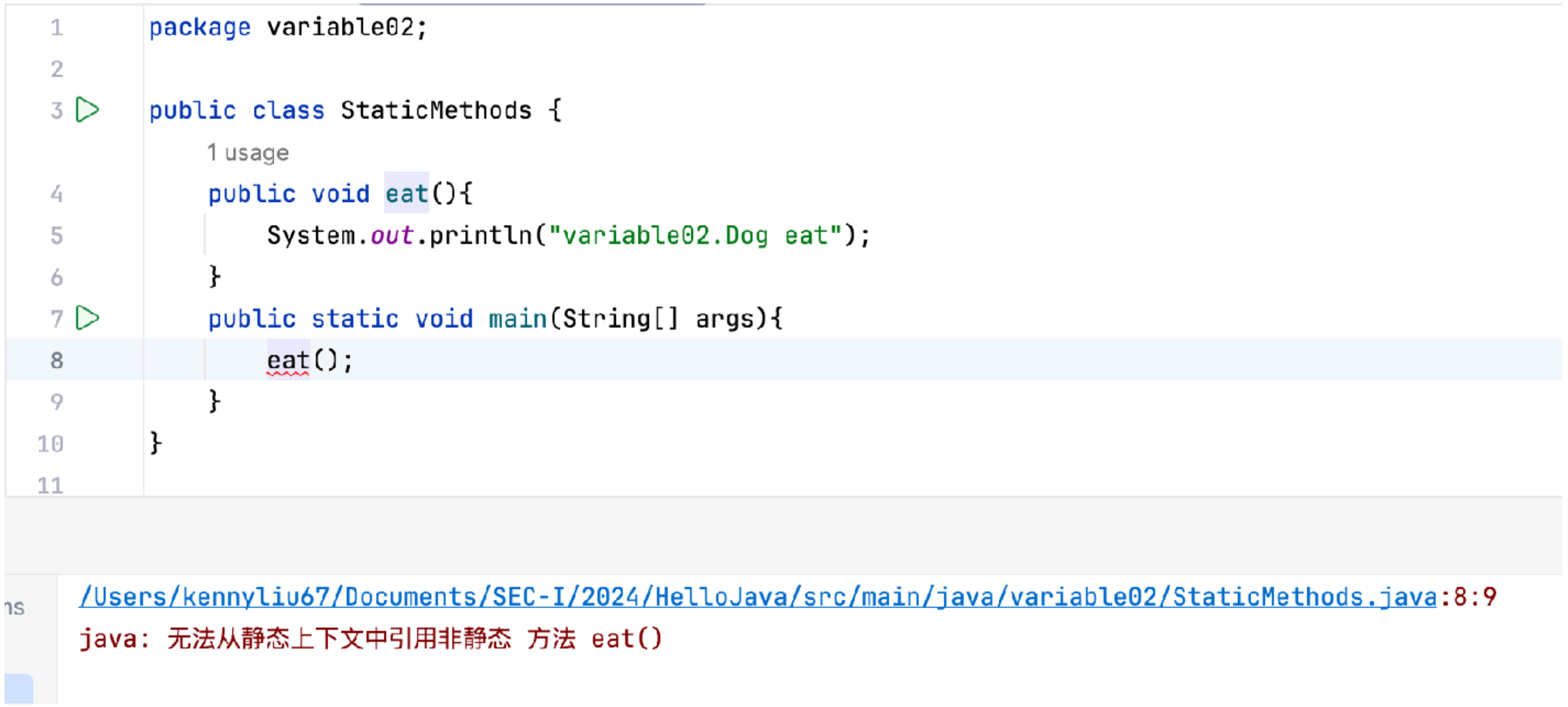
Static methods

- 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.”

Non-static method vs static method

- SoundPlayer player = new SoundPlayer();
- player.playSound(title); //引用变量
- Vs
- Math.min(42,36)//类

Static methods can't use none-static(instance) variable.



```
package variable02;

public class StaticMethods {
    1 usage
    public void eat(){
        System.out.println("variable02.Dog eat");
    }
    public static void main(String[] args){
        eat();
    }
}
```

```
/Users/kennyliu67/Documents/SEC-I/2024/HelloJava/src/main/java/variable02/StaticMethods.java:8:9
java: 无法从静态上下文中引用非静态 方法 eat()
```

Static Variable: Value is the same for all instance of the class.

```java
package oo08;

import java.util.ArrayList;

public class Dog{
    public String name;
    public int age;
    public static int numOfDogs;

    public Dog(){
        name="Puppy";
        age=0;
        numOfDogs++;
    }
    public Dog(String s, int a){
        name=s;
        age=a;
        numOfDogs++;
    }

    public void bark(){

        if(age>=3) System.out.println(name+":WoWoWo!");
        else System.out.println(name+":Wo!");

    }

    public static void main(String[] args){
        Dog d1 = new Dog("oo.Teddy",3);
        Dog d2 = new Dog();
        Dog d3 = d1;
        Dog d4 = new Dog("oo.Teddy",3);

        System.out.println(d1.name);

        d3.bark();
        d2.bark();

        if(d1==d4) System.out.println("yes");
        else System.out.println("no");

        String s1 = new String("abc");
        String s2 = new String("abc");

        if(s1==s2) System.out.println("yes");
        else System.out.println("no");

        if(s1.equals(s2)) System.out.println("yes");
        else System.out.println("no");

        String s3 = "abc";
        String s4 = "abc";

        if(s3==s4) System.out.println("yes");
        else System.out.println("no");

        d2 = d1;
        //d2 = null;

        d2.bark();

        int i =100;
        Integer integer = new Integer(i);
        System.out.print(integer.intValue());

        ArrayList<Integer> scoresList = new ArrayList<Integer>();

        //java.lang.Object is  root class

    }
}
```

Initializing a static variable

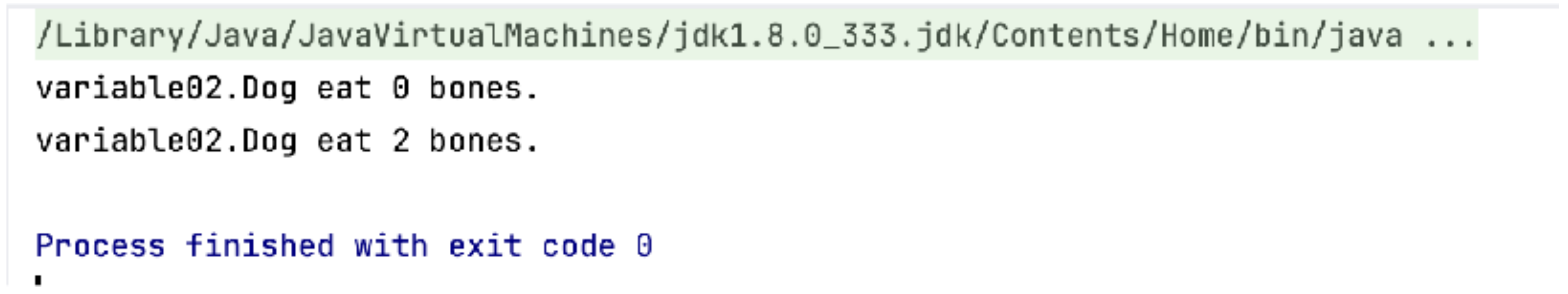
- Static variables are initialized when a class is loaded.
- A class is loaded because the JVM decides it's time to load it.
- And there are two guarantees about static initialization:
  - Static variables in a class are initialized before any object of that class can be created.
  - Static variables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

静态变量初始化

```java
public class StaticMethods {

    static int amount;

    public StaticMethods(int amount){
        this.amount = amount;
    }
    public static void  eat(){
        System.out.println("variable02.Dog eat "+amount+" bones.");
    }
    public static void main(String[] args){
        eat();
        StaticMethods staticMethods = new StaticMethods(2);
        eat();
    }
}
```



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_333.jdk/Contents/Home/bin/java ...
variable02.Dog eat 0 bones.
variable02.Dog eat 2 bones.

Process finished with exit code 0
```

Static final variables are constants

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

- public static final double PI = 3.141592653589793

常量初始化

- public class Foo{
    - public static final int FOO_X = 3;
- }
- 或者
- public class BAR{
    - public static final double BAR_X;
    - static{
        - BAR_X = (double) Math.random();
    - }
- }

final关键字

- 修饰变量
    - 不能改变值
- 修饰方法
    - 不能override
- 修饰类
    - 不能有子类

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);
    - }
- } ///:~

final 参数

- class Gizmo {
- public void spin() {}
- }
- public class FinalArguments {
  - void with(final Gizmo g) {
  - //! g = new Gizmo(); // Illegal -- g is final
  - }
  - void without(Gizmo g) {
    - g = new Gizmo(); // OK -- g not final
    - g.spin();
  - }
  - // void f(final int i) { i++; } // Can't change
  - // You can only read from a final primitive:
  - int g(final int i) { return i + 1; }
  - public static void main(String[] args) {
    - FinalArguments bf = new FinalArguments();
    - bf.without(null);
    - bf.with(null);
  - }
- } ///:~

final 方法

- 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。

Constructor

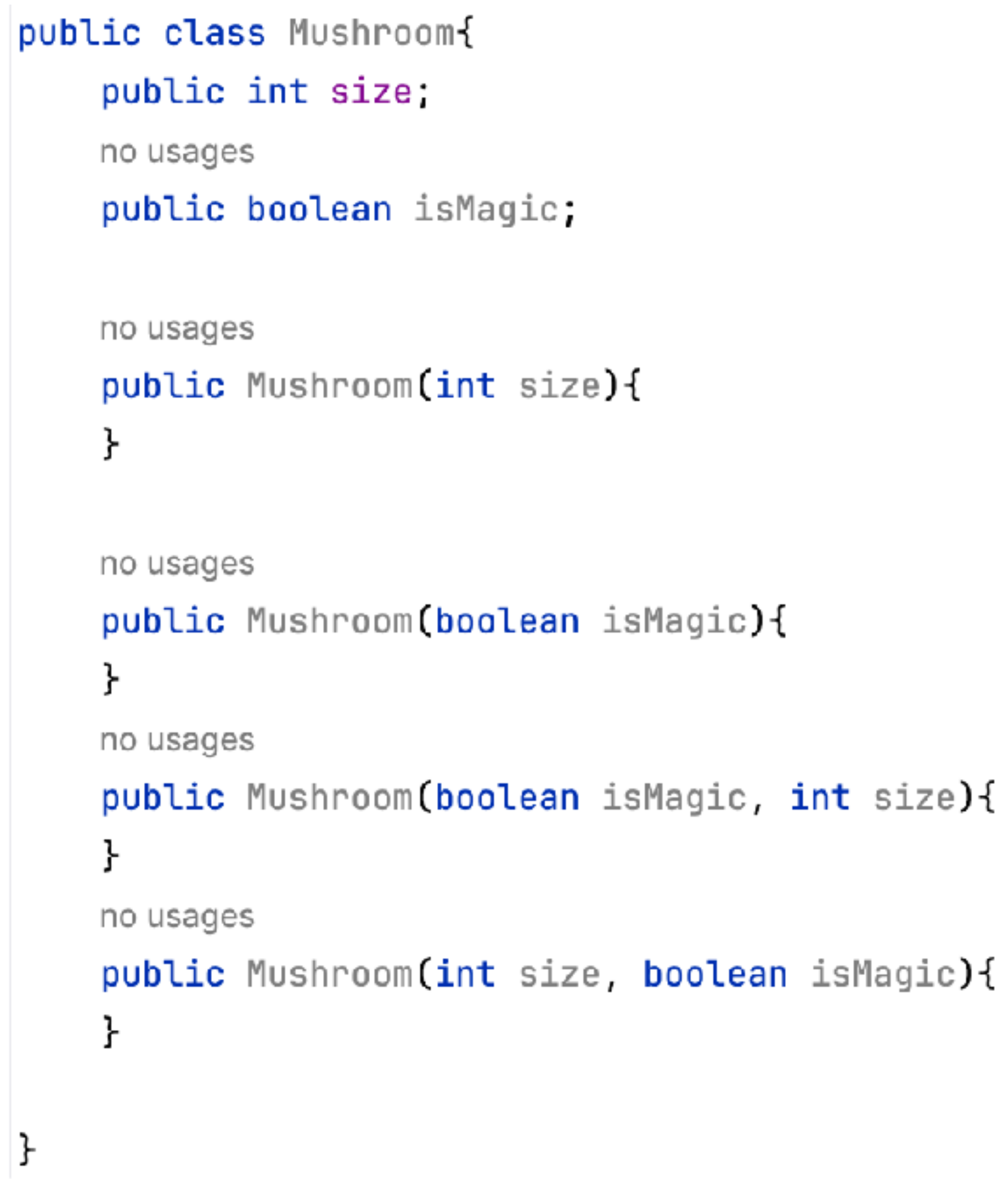
- The key feature of a constructor is that it runs before the object can be assigned to a reference.
- The constructor gives you a chance to step into the middle of new.
- Initializing the state of a class.

- public class Mushroom{
  - public Mushroom()
- }

No return type for Constructor

重载构造方法

- 如果你没写构造方法，编译器会帮你创建构造方法。
- 编译器只能帮你创建无参数的构造方法。
- 如果你写了有参数的构造方法，你必须自己写无参数的构造方法。
- public class Mushroom{
    - public Mushroom(int size)
    - public Mushroom()
    - public Mushroom(boolean isMagic)
    - public Mushroom(boolean isMagic, int size)
    - public Mushroom(int size, boolean isMagic)
- }

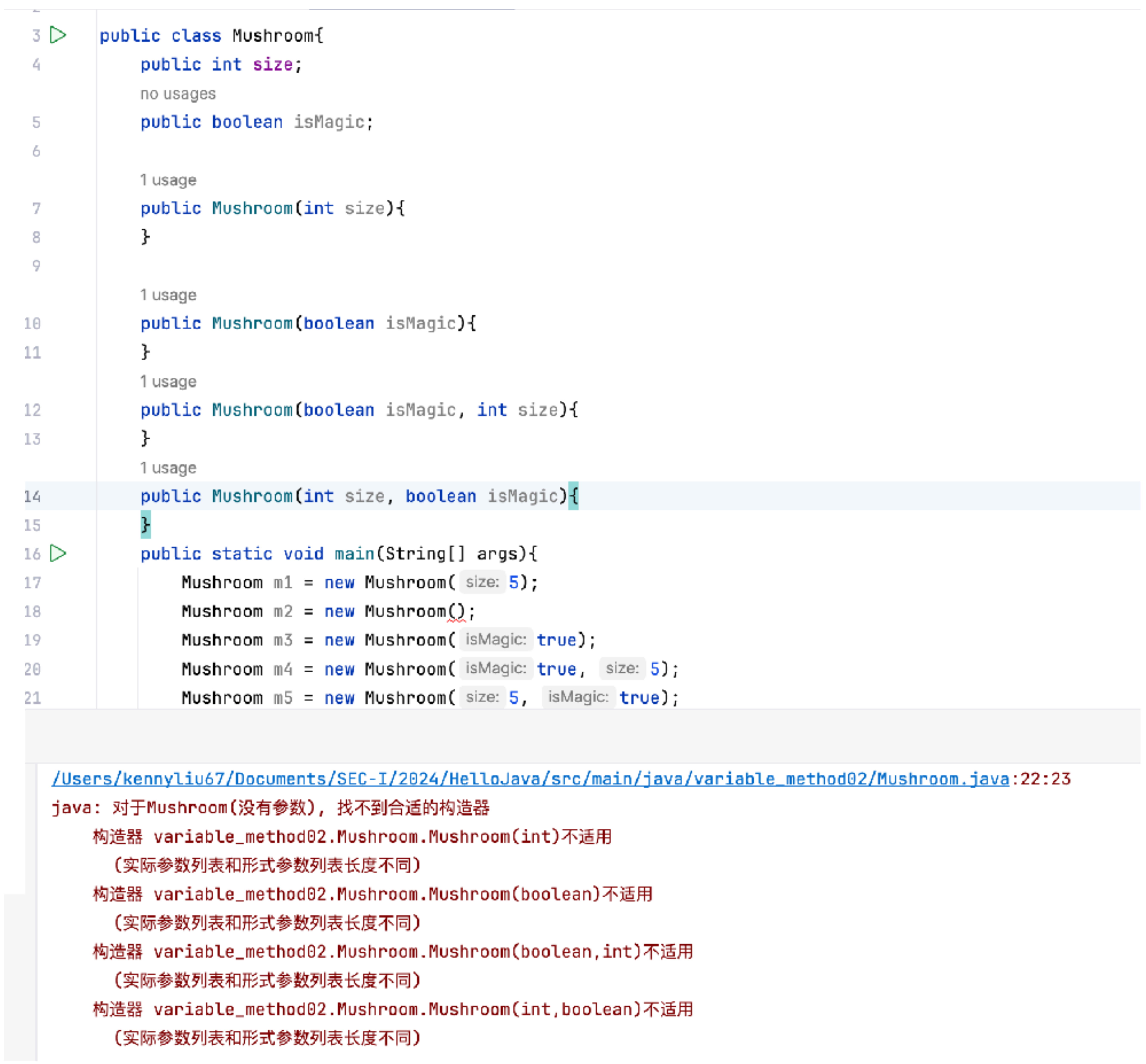


```java
public class Mushroom{
    public int size;
    no usages
    public boolean isMagic;

    no usages
    public Mushroom(int size){
    }

    no usages
    public Mushroom(boolean isMagic){
    }
    no usages
    public Mushroom(boolean isMagic, int size){
    }
    no usages
    public Mushroom(int size, boolean isMagic){
    }

}
```



```java
public class Mushroom{
    public int size;
    no usages
    public boolean isMagic;

    1 usage
    public Mushroom(int size){
    }

    1 usage
    public Mushroom(boolean isMagic){
    }
    1 usage
    public Mushroom(boolean isMagic, int size){
    }
    1 usage
    public Mushroom(int size, boolean isMagic){
    }
    public static void main(String[] args){
        Mushroom m1 = new Mushroom( size: 5);
        Mushroom m2 = new Mushroom();
        Mushroom m3 = new Mushroom( isMagic: true);
        Mushroom m4 = new Mushroom( isMagic: true,  size: 5);
        Mushroom m5 = new Mushroom( size: 5,  isMagic: true);
```

```
/Users/kennyliu67/Documents/SEC-I/2024/HelloJava/src/main/java/variable_method02/Mushroom.java:22:23
java: 对于Mushroom(没有参数), 找不到合适的构造器
    构造器 variable_method02.Mushroom.Mushroom(int)不适用
      (实际参数列表和形式参数列表长度不同)
    构造器 variable_method02.Mushroom.Mushroom(boolean)不适用
      (实际参数列表和形式参数列表长度不同)
    构造器 variable_method02.Mushroom.Mushroom(boolean,int)不适用
      (实际参数列表和形式参数列表长度不同)
    构造器 variable_method02.Mushroom.Mushroom(int,boolean)不适用
      (实际参数列表和形式参数列表长度不同)
```

对象

Outline

- 封装
- 类
- 对象
    - 对象初始化
    - 垃圾回收机制

对象初始化初步

- 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.

静态域的初始化顺序的示例

- 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；
  - }

```
•    public class StaticTest {
•    public static int X = 10;
•    static {

•        X = 30;

•    }
•
•    public static int Y = X * 2;

•    public static void main(String[] args) {

•        System.out.println(Y); // 输出60
•    }
• }
```

```
public class StaticTest {

    //这里和上一段代码的不同在于把静态代码块和静态变量X的赋值这两句代码交换了位置。



    static {

        X = 30;

    }
    public static int X = 10;

    public static int Y = X * 2;

    public static void main(String[] args) {

        System.out.println(Y); // 输出20

    }

}
```

生存期

- 局部变量
  - 方法执行的时间段
- 成员变量
  - 对象存活的时间段

垃圾回收

- 当对象指向自己的最后的引用消失，对象就可以被GC回收。
- void go(){
    - Life z = new Life();
- }// 1. 方法执行完毕，局部变量消失，源对象可以回收

- void go(){
    - Life z = new Life();
    - z = new Life(); // 2. 指向另一个对象，原对象可以回收
- }

- void go(){
    - Life z = new Life();
    - Z = null；// 3. 指向null，原对象可以回收
- }

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。

垃圾回收机制

- Java 垃圾回收机制最基本的做法是分代回收。
  - 内存中的区域被划分成不同的世代，对象根据其存活的时间被保存在对应世代的区域中。
  - 一般的实现是划分成3个世代：年轻、年老和永久。
- 内存的分配是发生在年轻世代中的。当一个对象存活时间足够长的时候，它就会被复制到年老世代中。对于不同的世代可以使用不同的垃圾回收算法。进行世代划分的出发点是对应用中对象存活时间进行研究之后得出的统计规律。一般来说，一个应用中的大部分对象的存活时间都很短。比如局部变量的存活时间就只在方法的执行过程中。基于这一点，对于年轻世代的垃圾回收算法就可以很有针对性。

垃圾回收机制

- 年轻世代的内存区域被进一步划分成
    - 伊甸园（Eden）
    - 两个存活区（survivor space）。
- 伊甸园是进行内存分配的地方，是一块连续的空闲内存区域。在上面进行内存分配速度非常快，因为不需要进行可用内存块的查找。
- 两个存活区中始终有一个是空白的。在进行垃圾回收的时候，伊甸园和其中一个非空存活区中还存活的对象根据其存活时间被复制到当前空白的存活区或年老世代中。经过这一次的复制之后，之前非空的存活区中包含了当前还存活的对象，而伊甸园和另一个存活区中的内容已经不再需要了，只需要简单地把这两个区域清空即可。下一次垃圾回收的时候，这两个存活区的角色就发生了交换。一般来说，年轻世代区域较小，而且大部分对象都已经不再存活，因此在其中查找存活对象的效率较高。

垃圾回收机制

- 而对于年老和永久世代的内存区域，则采用的是不同的回收算法，称为“标记-清除-压缩（Mark-Sweep-Compact）”。

- 标记的过程是找出当前还存活的对象，并进行标记；清除则遍历整个内存区域，找出其中需要进行回收的区域；而压缩则把存活对象的内存移动到整个内存区域的一端，使得另一端是一块连续的空闲区域，方便进行内存分配和复制。