

OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
 - Project 3 提交时间 4月10日晚9点

- 复习

- 操作符

- 关系操作: 布尔表达式, `==`, `.equals()`
 - 逻辑操作: 布尔表达式
 - 字符串连接: `s+t`; `s+= t`; `.toString()`;

- 控制结构

- `foreach`: `for(int i : a)`

- 静态方法

- 属于类的方法 **vs** 属于对象的方法
 - 不需要创建对象就可以调用
 - `main`

- 静态数据

- 属于类的数据 **vs** 属于对象的方法
 - 不需要创建对象就可以访问

- 复习
 - Java 静态方法与静态数据

```
public class StaticTest {  
    double d;  
    static void display() {  
        System.out.println("Hello");  
    }  
  
    public static void main(String [ ]args) {  
        display();  
        StaticTest.display();  
        StaticTest s = new StaticTest();  
        s.display();  
    }  
}
```

```
public class StaticTest {  
    static int i = 1;  
    static void display() {  
        System.out.println("Hello");  
    }  
  
    public static void main(String [ ]args) {  
        display();  
        StaticTest.display();  
        int a = StaticTest.i;  
    }  
}
```

OOP with Java

- 创建对象
- `this` 关键字
- 销毁对象
- 成员初始化

OOP with Java

- 创建对象
- this 关键字
- 销毁对象
- 成员初始化

创建对象

- 构造函数(Constructor)

- 一个函数
- 名称与类名相同
 - MyType
- 无返回值
- Let's try it

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        MyType n = new MyType();  
        m.set(1);  
        n.set(2);  
    }  
}
```

创建对象

- 构造函数
 - 带参数的构造函数
 - Let's try it
- 例子
 - String
 - Integer
- 默认构造函数 (Default Constructor)

创建对象

- 构造函数
 - 作用: 初始化对象
- 无返回值 vs 返回值为void
 - `MyType()`
 - `void MyType()`
- `new` 操作符的返回值
 - `new MyType();`

创建对象

- 问题:

```
Integer i1 = new Integer(11235);  
Integer i2 = new Integer("11235");
```

- 多个构造函数?

创建对象

- 函数重载(Method Overloading)
 - 问题: C语言编写函数实现以下功能:
 - 给定一个整数参数, 输出这个参数
 - 给定一个字符串参数, 输出这个字符串
 - 给定一个结构体, 输出结构体的内容

```
int print_int(int *a);  
int print_str(const char* s);  
int print_struct_A(const struct A *t);  
int print_struct_B(const struct B *t);  
int print_struct_C(const struct C *t);
```

创建对象

- 观察1

print

```
int print_int(int *a);  
int print_str(const char* s);  
  
int print_struct_A(const struct A *t);  
int print_struct_B(const struct B *t);  
int print_struct_C(const struct C *t);  
.....
```

对比人类语言:

- we will say: wash the shirt, wash the car, wash the dog
- we won't say: shirtWash the shirt, carWash the car, dogWash the dog

创建对象

- 观察2

```
int print_int(int *a);  
int print_str(const char* s);  
  
int print_struct_A(const struct A *t);  
int print_struct_B(const struct B *t);  
int print_struct_C(const struct C *t);  
.....
```

- 函数名随参数的不同而不同
- 冗余信息

创建对象

- C语言不支持
 - 函数由函数名唯一确定

```
int print(int *a);  
int print(const char* s);  
  
int print(const struct A *t);  
int print(const struct B *t);  
int print(const struct C *t);  
.....
```

Java 是否支持?

创建对象

- 函数重载

- 函数名相同, 参数类型/数量不同

- 优点

- 接口简洁, 统一

```
public class Printer {  
  
    void print(int x) {  
        System.out.println("print an integer: " + x);  
    }  
    void print(MyType m) {  
        System.out.println("print a MyType: " + m.get());  
    }  
    public static void main(String [ ]args) {  
        Printer p = new Printer();  
        p.print(3);  
        p.print(new MyType());  
    }  
}
```

创建对象

- 函数重载
 - 重载构造函数
 - Let's try it

```
Integer i1 = new Integer(11235);  
Integer i2 = new Integer("11235");
```


创建对象

- 函数重载

- 如何区分不同的函数？函数名+参数列表
- 参数列表：参数类型，参数顺序
 - `print(int i, String s) { ... }`
 - `print(String s, int l) { ... }`
- 返回值？
 - 返回值不能重载

```
void f() { }  
int f() {return 1;}
```

```
int x = f();  
f();
```

创建对象

- 函数重载与基本类型的转换

```
public class PrimitiveOverloading {  
    void f(char x) { System.out.println("f(char)"); }  
    void f(byte x) { System.out.println("f(byte)"); }  
    void f(short x) { System.out.println("f(short)"); }  
    void f(int x) { System.out.println("f(int)"); }  
    void f(long x) { System.out.println("f(long)"); }  
    void f(float x) { System.out.println("f(float)"); }  
    void f(double x) { System.out.println("f(double)"); }  
    public static void main(String [ ]args) {  
        PrimitiveOverloading p = new PrimitiveOverloading();  
        p.f('a');  
        byte x1 = 0; p.f(x1);  
        short x2 = 0; p.f(x2);  
        p.f(3); p.f(3L); p.f(3.0f); p.f(3.0d);  
    }  
}
```

创建对象

- 函数重载与基本类型的转换

```
public class PrimitiveOverloading {  
    // void f(char x) { System.out.println("f(char)"); }  
    void f(byte x) { System.out.println("f(byte)"); }  
    void f(short x) { System.out.println("f(short)"); }  
    void f(int x) { System.out.println("f(int)"); }  
    void f(long x) { System.out.println("f(long)"); }  
    void f(float x) { System.out.println("f(float)"); }  
    void f(double x) { System.out.println("f(double)"); }  
    public static void main(String [ ]args) {  
        PrimitiveOverloading p = new PrimitiveOverloading();  
        p.f('a');  
        byte x1 = 0; p.f(x1);  
        short x2 = 0; p.f(x2);  
        p.f(3); p.f(3L); p.f(3.0f); p.f(3.0d)  
    }  
}
```

创建对象

- 函数重载与基本类型的转换

```
public class PrimitiveOverloading {  
    // void f(char x) { System.out.println("f(char)"); }  
    // void f(byte x) { System.out.println("f(byte)"); }  
    void f(short x) { System.out.println("f(short)"); }  
    void f(int x) { System.out.println("f(int)"); }  
    void f(long x) { System.out.println("f(long)"); }  
    void f(float x) { System.out.println("f(float)"); }  
    void f(double x) { System.out.println("f(double)"); }  
    public static void main(String [ ]args) {  
        PrimitiveOverloading p = new PrimitiveOverloading();  
        p.f('a');  
        byte x1 = 0; p.f(x1);  
        short x2 = 0; p.f(x2);  
        p.f(3); p.f(3L); p.f(3.0f); p.f(3.0d)  
    }  
}
```

创建对象

- 函数重载与基本类型的转换

```
public class PrimitiveOverloading {  
    // void f(char x) { System.out.println("f(char)"); }  
    // void f(byte x) { System.out.println("f(byte)"); }  
    // void f(short x) { System.out.println("f(short)"); }  
    // void f(int x) { System.out.println("f(int)"); }  
    void f(long x) { System.out.println("f(long)"); }  
    void f(float x) { System.out.println("f(float)"); }  
    void f(double x) { System.out.println("f(double)"); }  
    public static void main(String [ ]args) {  
        PrimitiveOverloading p = new PrimitiveOverloading();  
        p.f('a');  
        byte x1 = 0; p.f(x1);  
        short x2 = 0; p.f(x2);  
        p.f(3); p.f(3L); p.f(3.0f); p.f(3.0d)  
    }  
}
```

创建对象

- 函数重载与基本类型的转换
 - 当转换不损失精度 (up-casting)
 - 调用参数类型"最近"的函数
 - 例如: char → int, byte → short, short → int, int → long, long → float, float → double
 - 当转换损失精度 (down-casting)
 - 需要强制转换
- 函数重载与toString()方法
 - System.out.println();
- 函数重载与autoboxing/unboxing
 - int and Integer

创建对象

- 构造函数

- 默认构造函数

- 当没有构造函数时, 系统默认给定
 - 当给定构造函数时, 没有默认构造函数

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    MyType(double x) { set(x); }  
  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
    }  
}
```

创建对象

- 总结
 - 构造函数
 - 默认构造函数
 - 函数重载

OOP with Java

- 创建对象
- **this** 关键字
- 销毁对象
- 成员初始化

this 关键字

- 含义

- 在类的非静态方法中, 返回调用该方法的对象的引用

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) {  
        d = x;  
    }  
    double get() { return d; }  
    void display(){  
        System.out.println(this.toString());  
    }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.display();  
        MyType n = new MyType();  
        n.display();  
    }  
}
```

this 关键字

- 例1: 区分参数名称与数据成员名称

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double d) {  
        d = d;  
    }  
    double get() { return d; }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.set(1);  
    }  
}
```

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double d) {  
        this.d = d;  
    }  
    double get() { return d; }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.set(1);  
    }  
}
```

this 关键字

- 例2: 返回当前对象

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    MyType increase() {  
        d++; // this.d++;  
        return this;  
    }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        MyType n = m.increase();  
    }  
}
```

this 关键字

- 例3:作为其他方法的参数

```
public class MyType {
    int i;
    double d;
    char c;
    void set(double x) { d = x; }
    double get() { return d; }
    MyType increase() {
        return Worker.increase(this);
    }
    public static void main(String [ ]args) {
        MyType m = new MyType();
        MyType n = m.increase();
    }
}
class Worker {
    public static MyType increase(MyType m){
        m.set(m.get() + 1)
        return m;
    }
}
```

this 关键字

- 例4: 在构造函数中调用构造函数

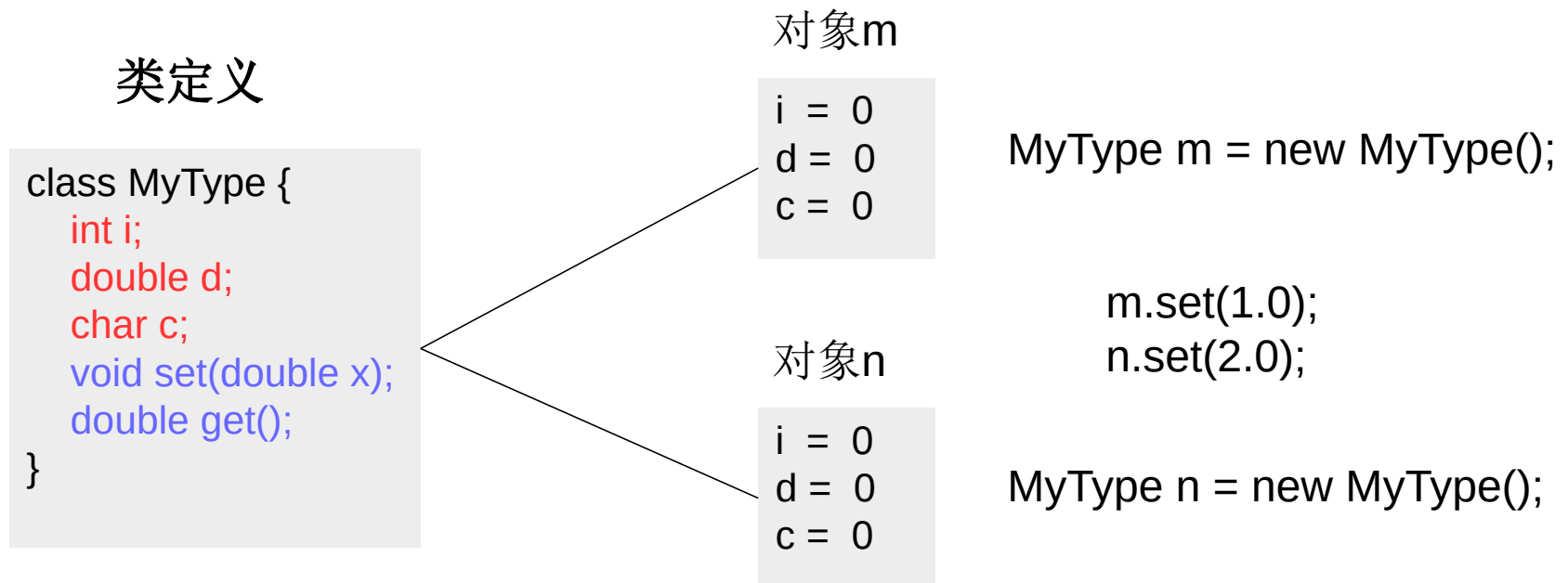
- this(...)
- 出现在构造函数第一行
- 只能调用一个构造函数

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    MyType(double d) {this.d = d;}  
    MyType(int i) {this.i = i;}  
    MyType(int i, double d, char c){  
        this(d);  
        this.i = i; // can not use this(i) again  
        this.c = c;  
    }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.set(1);  
    }  
}
```

this 关键字

- 静态方法中是否可以使用this 关键字？
- Let's try
-
- 在类的非静态方法中, 返回调用该方法的对象的引用

this 关键字



问: **set()** 方法如何区分不同的对象?

答: 编译器在调用**set()**方法时, 隐含的增加了参数 **MyType this**, 说明调用**set()**方法的对象。换句话说, 在**set()**内部, **this**指代哪一个对象调用了它。

问: 静态环境与非静态环境的区别?

答: 是否与具体对象绑定 ↔ 是否可以使用**this**关键字

this 关键字

- 如何使用静态方法实现对象的方法？

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.set(1);  
    }  
}
```

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    public static void set(MyType t, double x) {  
        t.d = x;  
    }  
    public static double get(MyType t) {  
        return t.d;  
    }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        MyType.set(m, 1);  
        m.set(m, 1);  
    }  
}
```

this 关键字

- static 函数
 - 编译器不为其添加新的参数
 - 无法使用this

this 关键字

- 总结
 - 在类的非静态方法中, 返回调用该方法的对象的引用

- 复习: 对象的创建
 - new操作符+构造函数

- 构造函数

- 一些特殊的方法
 - 函数名与类名一样
 - 没有返回值

- new 操作符

- 返回一个引用, 指向调用构造函数所创建的对象
 - 类似于malloc

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
  
    MyType() { System.out.println("HI"); }  
  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
    }  
}
```

• 复习

– 函数重载: 函数 = 函数名 + 参数列表

```
public class Printer {  
    void print(int x) {  
        System.out.println("print an integer: " + x);  
    }  
    void print(MyType m) {  
        System.out.println("print a MyType: " + m.get());  
    }  
}  
  
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
  
    MyType(double x) { set(x); };  
    MyType() { System.out.println("mytype create"); };  
  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        MyType n = new MyType(2.0);  
    }  
}
```

- `System.out.println()`发生了什么？

<code>void</code>	<code>println(boolean x)</code> Prints a boolean value and then terminates the line.
<code>void</code>	<code>println(char x)</code> Prints a character and then terminates the line.
<code>void</code>	<code>println(char[] x)</code> Prints an array of characters and then terminates the line.
<code>void</code>	<code>println(double x)</code> Prints a double-precision floating-point number and then terminates the line.
<code>void</code>	<code>println(float x)</code> Prints a floating-point number and then terminates the line.
<code>void</code>	<code>println(int x)</code> Prints an integer and then terminates the line.
<code>void</code>	<code>println(long x)</code> Prints a long integer and then terminates the line.
<code>void</code>	<code>println(Object x)</code> Prints an Object and then terminates the line.
<code>void</code>	<code>println(String x)</code> Prints a String and then terminates the line.

- 复习

- this 关键字

- 在类的非静态方法中, 返回调用该方法的对象的引用

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) {  
        this.d = x;  
    }  
    double get() { return d; }  
    public static void main(String [ ]args) {  
        MyType m = new MyType();  
        m.set(1);  
    }  
}
```

OOP with Java

- 创建对象
- this 关键字
- 销毁对象
- 成员初始化

销毁对象

- 创建类的对象

```
MyType a = new MyType();
```

- 使用对象的方法

```
int b = a.i; a.set(); a.get();
```

- 销毁对象

- Java 自动销毁
- 垃圾回收机制 (Garbage Collection)

- new \approx malloc

问题: 垃圾回收能否保证对象被"正确"的销毁?

- 不需要调用free()

销毁对象

- 对象占有的资源

- new 操作时系统分配的内存
- 其他资源
 - 例如:文件描述符, 锁..

系统自动分配

程序显式申请

销毁对象

```
public class Resource throw IOException{
    int i;
    BufferedReader f;
    Resource() { i = 0;}
    void open() {
        f= new BufferedReader(new FileReader(new File("a.txt")));
    }
    String readLine() { return f.readLine();}

    public static void main(String [ ]args) {
        Resource r = new Resource();
        r.open();
        ....
    }
}
```

系统自动分配

程序显式申请

销毁对象

- 对象占有的资源

- new 操作时系统分配的内存
- 其他资源

系统自动分配

程序显式分配

- 销毁对象占有的资源

- new 操作时系统分配的内存
- 其他资源

系统自动回收(垃圾回收)

程序显式回收

垃圾回收不回收程序显式申请的资源!

销毁对象

```
public class Resource {  
    int i;  
    BufferedReader f;  
    Resource() { i = 0;}  
    void open() {  
        f= new BufferedReader(new FileReader(new File("a.txt")));  
    }  
    String readLine() { return f.readLine();}  
    void close() {  
        f.close();  
    }  
  
    public static void main(String [ ]args) {  
        Resource r = new Resource();  
        r.open();  
        ....  
        r.close();  
    }  
}
```

系统自动分配/回收(垃圾回收)

程序显式申请

程序显式回收

销毁对象

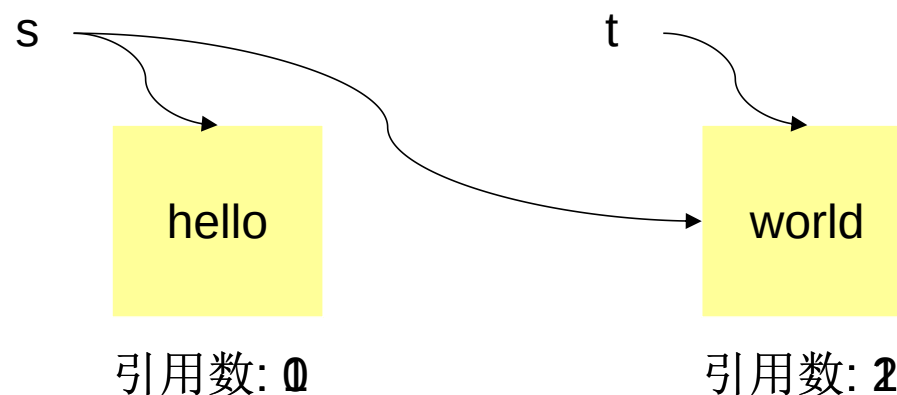
- 垃圾回收(要点1)

1.仅回收new创建的内存.

销毁对象

- 垃圾回收(Garbage Collection, GC)
 - 对象的引用数
 - 多少个引用指向该对象
 - 引用数为0时Java虚拟机将对象标记为可回收

```
String s = new String("hello");  
String t = new String("world");  
s = t;
```



1. 内容为"hello" 的对象可回收
2. 是否立即被回收?

否!

销毁对象

- 垃圾回收
 - JVM的一个进程
 - 何时进行垃圾回收?
 - 当Java虚拟机 (JVM)发现内存不够时尝试进行回收
 - 由JVM决定是否回收, 何时回收 (并非实时进行)
 - `Garbage collection != free()`
 - 为何如此设计?
 - 垃圾回收将占用系统资源, 影响用户程序
 - 减少虚拟机进行垃圾回收的频率

销毁对象

- 垃圾回收
 - `System.gc()` 调用
 - 通知JVM可以进行垃圾回收
 - 类的`finalize()`方法
 - 对该对象进行垃圾回收之前, 调用`finalize()`方法
 - 下一次垃圾回收时, 再正式回收内存
 - 并非引用数为0时调用!
 - 不等于C++析构函数!
 - 避免使用`finalize()`方法

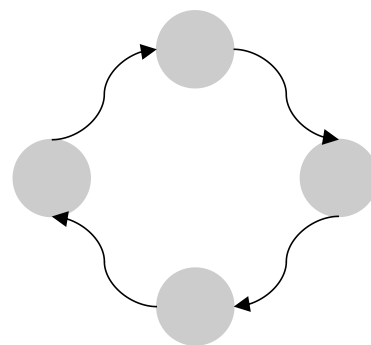
销毁对象

- 垃圾回收(要点2)

2.是否回收, 何时回收由Java虚拟机控制.

销毁对象

- JVM如何实现垃圾回收
 - 算法1: 引用计数
 - 每个对象包含一个计数器, 记录指向该对象的引用数
 - 垃圾收集器检查所有的对象, 若引用数为0则删除
 - 问题
 - 循环引用
 - 基本没有JVM使用该方法



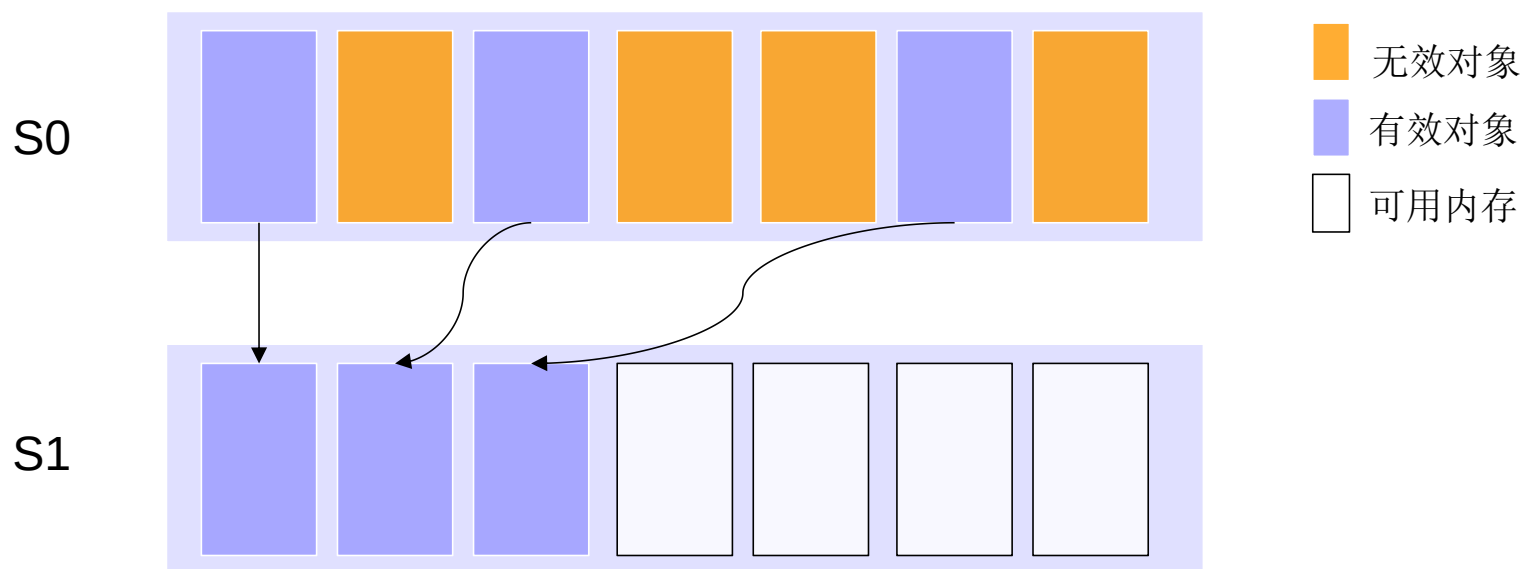
• JVM如何实现垃圾回收

– 算法2: stop-and-copy

- 找出所有有效的对象
 - 从栈上的引用出发可以找到所有的有效对象
- 每找到一个有效对象, 将其拷贝到另外一块内存区域 (copy)
- 修改所有引用的值
- 被垃圾搜集的程序将被停止 (stop)

– 特点

- 两倍空间
- 当程序稳定(不容易产生垃圾)时, copy动作多余



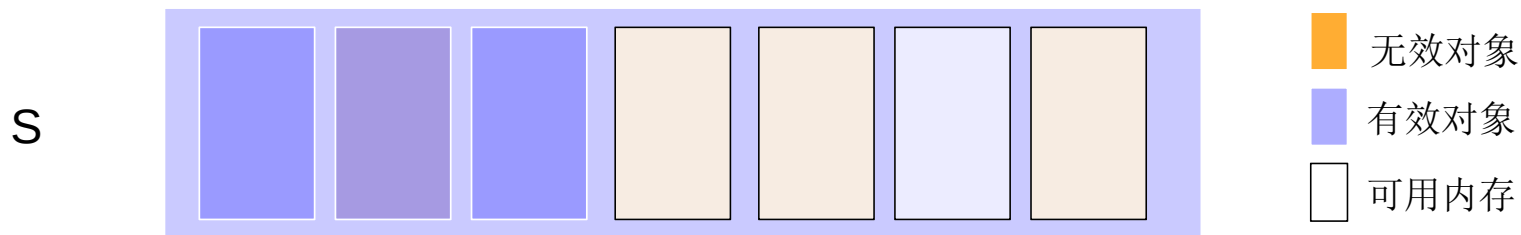
• JVM如何实现垃圾回收

– 算法3: mark-and-sweep

- 找出所有有效的对象
 - 从栈上的引用出发可以找到所有的有效对象
- 将有效对象标记为1 (mark)
- 在当前内存区域重新放置有效对象(sweep)
- 修改所有引用的值

– 特点

- 当程序相对稳定(不容易产生垃圾)时, 能快速完成收集
- 当程序容易产生垃圾时, 效率较低



- JVM如何实现垃圾回收

- 算法4:

- 结合stop-and-copy, mark-and-sweep
 - 当较容易产生垃圾时:使用stop-and-copy
 - 当不容易产生垃圾时:使用mark-and-sweep

- 如何判断是否容易产生垃圾

- 大多数对象生存周期较短
 - 每个对象记录其生存周期(generation)
 - 经历了多少次垃圾搜集
 - 根据生存周期归类
 - 生存周期较小的对象: 使用stop-and-copy
 - 生存周期较大的对象: 使用mark-and-sweep

adaptive generational stop-and-copy mark-and-sweep
garbage collection

销毁对象

- 总结

- 垃圾回收

- 1.仅回收new创建的内存.**

- 2.是否回收, 何时回收由Java虚拟机控制.**





C酱
接着用goto语句
飞起来了~!



可是飞到了意料之外的地方—!!
C酱闯入了未定义区域—!
华丽地脱离了跑道~!!

获胜者会是Java酱吗!?

哎呀、
我要赢了!



哦！
Java酱，难道在终点线前
触发了垃圾收集？

.....

Full GC！
是Full GC！

Java酱
为了回收内存
而停下了！

ヒッ！

Swift酱超过了停下的Java酱、
从容地冲过了终点！！

JS酱也要加油啊！

大家、
都好快啊~

ヒッ！...

EOF

真美...

OOP with Java

- 创建对象
- this 关键字
- 销毁对象
- 成员初始化

成员初始化

- 类数据成员的默认初始化

```
public class InitialValues {  
    boolean t;          // false  
    char c;             // '\u0000'  
    byte b;             // 0  
    short s;            // 0  
    int i;              // 0  
    long l;             // 0  
    float f;            // 0.0  
    double d;           // 0.0  
    InitialValues reference;  
  
    public static void main(String [ ]args) {  
        InitialValues iv = new InitialValues();  
    }  
}
```

但方法的局部变量并无默认初始化！

成员初始化

- 类数据成员的初始化

```
public class InitialValues {  
    boolean t = true;  
    char c = 'x';  
    byte b = 47;  
    short s = 0xff;  
    int i = 999;  
    long l = 1;  
    float t = 3.14f;  
    double d = 3.14159;  
    InitialValues reference = new InitialValues();  
  
    public static void main(String [ ]args) {  
        InitialValues iv = new InitialValues();  
    }  
}
```


成员初始化

- 使用构造函数初始化

```
public class MyType {  
    int i;  
    double d;  
    char c;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    MyType (int i, double d, char c){  
        this.i = i;  
        this.d = d;  
        this.c = c;  
    }  
    public static void main(String [ ]args) {  
        MyType m = new MyType(3, 3.14, 'x');  
        System.out.println(m.get());  
    }  
}
```

成员初始化

- 初始化的顺序
 - 所有数据成员初始化在构造函数调用前完成
 - 按照成员定义的顺序初始化

```
public class MyType {  
    int i = 1;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    double d = 1.0;  
    MyType (int i, double d, char c){  
        this.i = i;  
        this.d = d;  
        this.c = c;  
    }  
    char c = 'a';  
    public static void main(String [ ]args) {  
        MyType m = new MyType(3, 3.14, 'x');  
    }  
}
```

成员初始化

- 静态数据成员初始化
 - 默认初始化与成员数据默认初始化相同

```
public class InitialValues {  
    static boolean t = true;  
    static char c = 'x';  
    static byte b = 47;  
    static short s = 0xff;  
    static int i = 999;  
    static long l = 1;  
    static float f = 3.14f;  
    static double d = 3.14159;  
    static InitialValues reference = new InitialValues();  
  
    public static void main(String [ ]args) {  
        InitialValues iv = new InitialValues();  
    }  
}
```


成员初始化

- 静态成员初始化

- 当第一个该类型的对象被创建时初始化
- 后续创建该类型的对象时不初始化
- 先初始化静态数据成员, 后初始化对象数据成员

```
public class MyType {  
    int i = 1;  
    void set(double x) { d = x; }  
    double get() { return d; }  
    double d = 1.0;  
    MyType (int i, double d, char c){  
        this.i = i; this.d = d; this.c = c;  
    }  
    char c = 'a';  
    static int si = 10;  
    public static void main(String [ ]args) {  
        MyType m = new MyType(3, 3.14, 'x');  
    }  
}
```

成员初始化

- 总结
 - 对象的数据成员初始化
 - 类的静态成员初始化
 - 初始化的顺序

OOP with Java

- 创建对象
 - 构造函数概述
 - 重写方法
 - 默认构造函数
- this 关键字
- 销毁对象
 - 垃圾回收
- 成员初始化