

✧ 1 ✧ Java 应用与开发课程教学体系

很高兴同学们能够选修 Java 应用与开发课程。

希望我们一起通过这门课程的学习，建立 Java 语言编程的初步知识体系，掌握 Java 应用系统开发的方式、方法。更重要的，能够对编程这个事情、这项技能有更加深刻的认知，对未来的职业化发展有所促进。

Java 应用与开发课程的教学体系如图1.1所示，包括了 Java SE 和 Java EE 两个部分，每部分都涉及一些验证性实验，另外，会开展两次稍微大一点的集成开发项目。同时，在学习的过程中会穿插一些开发工具、设计模式、应用服务器和数据库的基本应用。

在课程学习的过程中，希望同学们要有足够的求知欲，养成良好的学习态度，具备不断探索的精神，多尝新、多实践、多总结。我想这是计算机专业人士应该具备的基本素养。

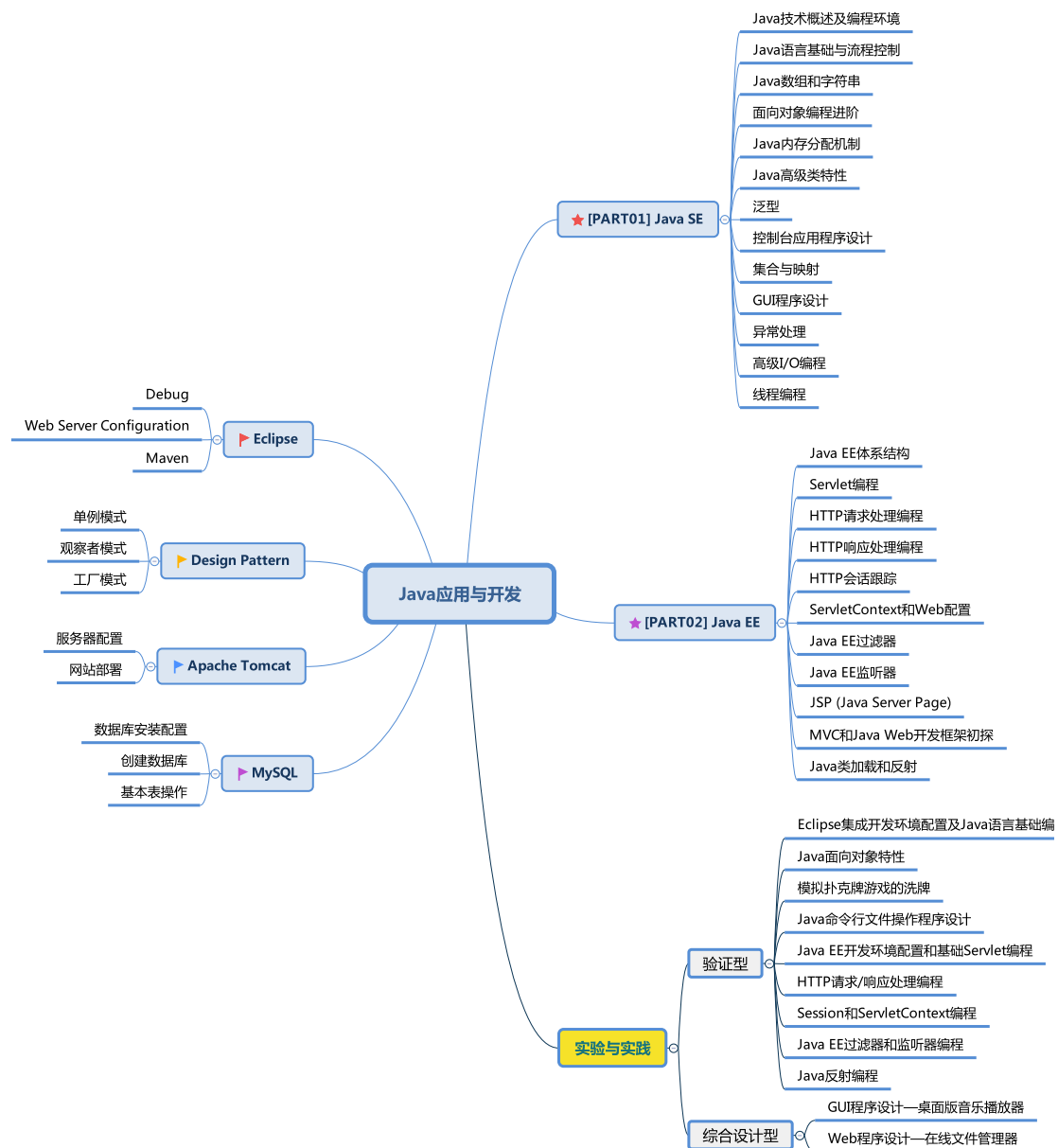


图 1.1 Java 应用与开发课程教学体系

✧ 2 ✧ Java 面向对象编程进阶

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第二周（根据校历，本周有两次课）

参考教材： 本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解多态和虚方法调用的概念，掌握其用法
2. 掌握方法重载的方法
3. 掌握 static 属性、方法和初始化块的用法
4. 了解设计模式，掌握单例设计模式
5. 掌握 final 关键字的概念和使用方法

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

2.1 多态性

2.1.1 多态的概念

在 Java 中，子类的对象可以替代父类的对象使用称为**多态**。Java 引用变量与所引用对象间的类型匹配关系如下：

- 一个对象只能属于一种确定的数据类型，该类型自对象创建直至销毁不能改变。
- 一个引用类型变量可能引用（指向）多种不同类型的对象——既可以引用其声明类型的对象，也可以引用其声明类型的子类的对象。

```
1 Person p = new Student(); //Student 是 Person 的子类
```

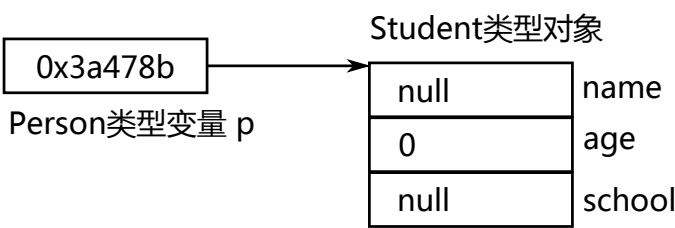


图 2.1 Java 多态

多态性同样适用与引用类型数组元素。

```
1 Person[] p = new Person[3];
2 p[0] = new Student(); // 假设 Student 类继承了 Person 类
3 p[1] = new Person();
4 p[2] = new Graduate(); //假设 Graduate 类继承了 Student 类
```

一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，该变量则不能再访问子类中添加的属性和方法，这体现了父类引用对子类对象的能力屏蔽性。

```
1 Student m = new Student();
2 m.setSchool("ouc"); // 合法
3 Person e = new Student();
```

```
4 e.setSchool("ouc"); // 非法
```

2.1.2 多态用法示例

Code: Person.java

```
1 public class Person {...}
```

Code: Student.java

```
1 public class Student extends Person {
2     private String school;

4     public void setSchool(String school) {
5         this.school = school;
6     }

8     public String getSchool() {
9         return school;
10    }

12    @Override
13    public String getInfo() {
14        return super.getInfo() + "\tSchool: " + school;
15    }
16 }
```

Code: PolymorphismSample.java

```
1 public class PolymorphismSample {
2     public void show(Person p) {
3         System.out.println(p.getInfo());
4     }

6     public static void main(String[] args) {
7         PolymorphismSample ps = new PolymorphismSample();
8         Person p = new Person();
9         ps.show(p);
10        Student s = new Student();
11        ps.show(s);
12    }
```

```
13 | }
```

多态提升方法通用性

以上代码中，`show()` 方法既可以处理 `Person` 类型的数据，又可以处理 `Student` 类型的数据，乃至未来定义的任何 `Person` 子类类型的数据，即不必为相关的每种类型单独声明一个处理方法，提高了代码的通用性。

2.1.3 虚方法调用

思考：一个引用类型的变量如果声明为父类的类型，但实际引用的是子类对象，则该变量就不能再访问子类中添加的属性和方法。但如果此时调用的是父类中声明过、且在子类中重写过的方法，情况如何？

补充代码。

2.1.4 对象造型

引用类型数据值之间的强制类型转换称为**造型**（Casting）。造型以下几种情况需要注意：

1. 从子类到父类的类型转换可以自动进行。

```
1 Person p = new Student();
```

2. 在多态的情况下，有时我们可能需要恢复一个对象的本来面目，以发挥其全部潜力。从父类到子类的类型转换必须通过造型实现。

```
1 Person p1 = new Student();  
2 Student s1 = (Student)p1; // 合法  
3 Person p2 = new Person();  
4 Student s2 = (Student)p2; // 非法
```

3. 无继承关系的引用类型间的转换是非法的。

```
1 String s = "Hello World!";  
2 Person p = (Person)s; // 非法
```

2.1.5 instanceof 运算符

如果运算符 `instanceof` 左侧的变量当前时刻所引用的对象的**真正类型**是其右侧给出的类型**或者是其子类**，则整个表达式的结果为 `true`。

```

1 class Person { --- }
2 class Student extends Person { --- }

4 public class Tool {
5     public void distribute(Person p) {
6         if (p instanceof Student) {
7             System.out.println("处理 Student 类型及其子类类型对象");
8         } else {
9             System.out.println("处理 Person 类型及其子类类型对象");
10        }
11    }
12 }


```

```

1 public class Test() {
2     public static void main(String[] args) {
3         Tool t = new Tool();
4         Student s = new Student();
5         t.distribute(t);
6     }
7 }

```

2.1.6 虚方法调用和造型

课程配套代码  package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

强调以下与虚方法调用和造型相关的重点：

- 系统根据运行时对象的真正类型来确定具体调用哪一个方法，这一机制被称为**虚方法调用**。
- 造型是引用类型数据值之间的强制类型转换。
- `instanceof` 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

2.2 方法重载

2.2.1 方法重载的概念

在一个类中存在多个同名方法的情况称为**方法重载**（Overload）。Java 对方法重载有以下要求：

- 重载方法参数列表必须不同。
- 重载既可以用于普通方法，也可以用于构造方法。

课程配套代码  sample.oop.MethodOverloadSample.java

2.2.2 调用重载的构造方法

使用 `this` 调用当前类中重载构造方法

可以在构造方法的第一行使用关键字 `this` 调用其它（重载的）构造方法。

```
1 public class Person {  
2     ...  
3     public Person(String name,int age) {  
4         this.name = name;  
5         this.age = age;  
6     }  
7     public Person(String name) {  
8         this(name, 18);  
9     }  
10    ...  
11 }
```

注意

关键字 `this` 的此种用法只能用在构造方法中，且 `this()` 语句如果出现必须位于方法体中代码的第一行。

使用 `super` 调用父类构造方法

Code: [Person.java](#)


```

1 public class Person {
2     ... (此处没有无参构造方法)
3     public Person(String name, int age) {
4         this.name = name;
5         this.age = age;
6     }
7     ...
8 }

```

Code: Student.java

```

1 public class Student extends Person {
2     private String school;
3     public Student(String name, int age, String school) {
4         super(name, age); // 显式调用父类有参构造方法
5         this.school = school;
6     }
7     public Student(String school) { //编译出错
8         // super(); // 隐式调用父类有参构造方法，则自动调用父类无参构造方法
9         this.school = school;
10    }
11 }

```

上述代码为什么会编译出错？

在 Java 类的构造方法中一定直接或间接地调用了其父类的构造方法（Object 类除外）。

1. 在子类的构造方法中可使用 `super` 语句调用父类的构造方法，其格式为 `super(< 实参列表 >)`。
2. 如果子类的构造方法中既没有显式地调用父类构造方法，也没有使用 `this` 关键字调用同一个类的其他重载构造方法，则系统会默认调用父类无参数的构造方法，其格式为 `super()`。
3. 如果子类构造方法中既未显式调用父类构造方法，而父类中又没有无参的构造方法，则编译出错。

课程配套代码  sample.oop.ConstructorOverloadSample.java

2.2.3 对象构造/初始化细节

第一阶段 为新建对象的实例变量分配存储空间并进行默认初始化。

第二阶段 按下述步骤继续初始化实例变量：

1. 绑定构造方法参数；
2. 如有 `this()` 调用，则调用相应的重载构造方法然后跳转到步骤 5；
3. 显式或隐式追溯调用父类的构造方法（`Object` 类除外）；
4. 进行实例变量的显式初始化操作；
5. 执行当前构造方法的方法体中其余的语句。

2.3 关键字 `static`

在 Java 类中声明**属性、方法和内部类**时，可使用关键字 `static` 作为修饰符。

- `static` 标记的属性或方法由整个类（所有实例）共享，如访问控制权限允许，可不必创建该类对象而直接用类名加“.”调用。
- `static` 成员也称“**类成员**”或“**静态成员**”，如“类属性”、“类变量”、“类方法”和“静态方法”等。

2.3.1 `static` 属性和方法

`static` 属性

- `static` 属性由其所在类（包括该类所有的实例）共享。
- 非 `static` 属性则必须依赖具体/特定的对象（实例）而存在。

`static` 方法

要在 `static` 方法中调用其所在类的非 `static` 成员，应首先创建一个该类的对象，通过该对象来访问其非 `static` 成员。

课程配套代码 ▶ `sample.oop.StaticMemberAndMethodSample.java`

2.3.2 初始化块

`static` 初始化块

在类的定义体中，方法的外部可包含 `static` 语句块，**`static` 块仅在其所属的类被载入时执行一次**，通常用于初始化 `static`（类）属性。

非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

课程配套代码 [sample.oop.StaticInitBlockSample.java](#)

2.3.3 静态导入

静态导入用于在一个类中导入其他类或接口中的 static 成员，语法格式如下：

```
1 import static <包路径>.<类名>.*
3 import static <包路径>.<类名>.<静态成员名>
```

Code: 静态导入应用示例

```
1 import static java.lang.Math.*;
2 public class Test {
3     public static void main(String[] args) {
4         double d = sin(PI * 0.45);
5         System.out.println(d);
6     }
7 }
```

2.3.4 Singleton 设计模式

所谓“模式”就是被验证为有效的常规问题的典型解决方案。设计模式（Design Pattern）在面向对象分析设计和软件开发中占有重要地位。好的设计模式可以使我们的更加方便的重用已有的成功设计和体系结构，极大的提高代码的重用性和可维护性。

经典设计模式分类主要分为以下三大类：

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

Singleton 设计模式也称“单子模式”或“单例模式”。

采用调试方式讲解示例代码

Singleton 代码的特点包括以下几个方面：

1. 使用静态属性 `onlyone` 来引用一个“全局性”的 Single 实例。
2. 将构造方法设置为 `private` 的，这样在外界将不能再使用 `new` 关键字来创建该类的新实例。
3. 提供 `public static` 的方法 `getSingle()` 以使外界能够获取该类的实例，达到全局可见的效果。

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 `ServletContext` 全局上下文对象），则使用该模式。

2.4 关键字 final

在声明 Java 类、变量和方法时可以使用关键字 `final` 来修饰，以使其具有“终态”的特性：

1. `final` 标记的类不能被继承；
2. `final` 标记的方法不能被子类重写；
3. `final` 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次；
4. `final` 标记的成员变量必须在声明的同时或在每个构造方法中显式赋值，然后才能使用；
5. `final` 不允许用于修饰构造方法、抽象类以及抽象方法。

关键字 `final` 应用举例如下：

```
1 public final class Test {
2     public static int totalNumber = 5;
3     public final int id;
4     public Test() {
5         id = ++totalNumber; // 赋值一次
6     }
7     public static void main(String[] args) {
8         Test t = new Test();
9         System.out.println(t.id);
10        final int i = 10;
11        final int j;
12        j = 20;
13        j = 30; // 非法
14    }
15 }
```

实验设计

✚ 3 ✚ Java 内存模型与分配机制

基本信息

课程名称：Java 应用与开发

授课教师：王晓东

授课时间：第二周（根据校历，本周有两次课）

参考教材：本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解 JVM 内存模型，掌握 JVM 内存构成
2. 理解 Java 程序的运行过程，学会通过调试模式观察内存的变化
3. 了解 Java 内存管理，认识垃圾回收
4. 建立编程时高效利用内存、避免内存溢出的理念

授课方式

理论课：多媒体教学、程序演示

实验课：上机编程

教学内容

3.1 Java 内存模型

3.1.1 Java 虚拟机（Java Virtual Machine, JVM）

Java 虚拟机的架构如图3.1所示。

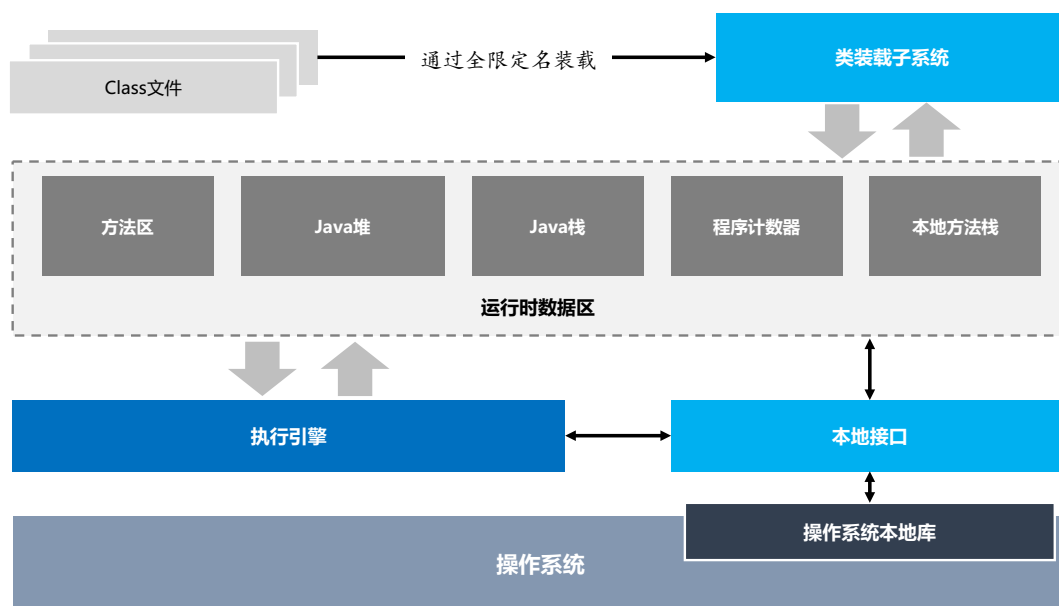


图 3.1 Java 虚拟机架构

- Java 程序运行在 JVM 上，JVM 是程序与操作系统之间的桥梁。
- JVM 实现了 Java 的平台无关性。
- JVM 是内存分配的前提。

3.1.2 JVM 内存模型

Java 程序运行过程会涉及的内存区域包括：

程序计数器 当前线程执行的字节码的行号指示器。

栈 保存局部变量的值，包括：用来保存基本数据类型的值；保存类的实例，即堆区对象的引用（指针），也可以用来保存加载方法时的帧。（Stack）

JVM内存模型

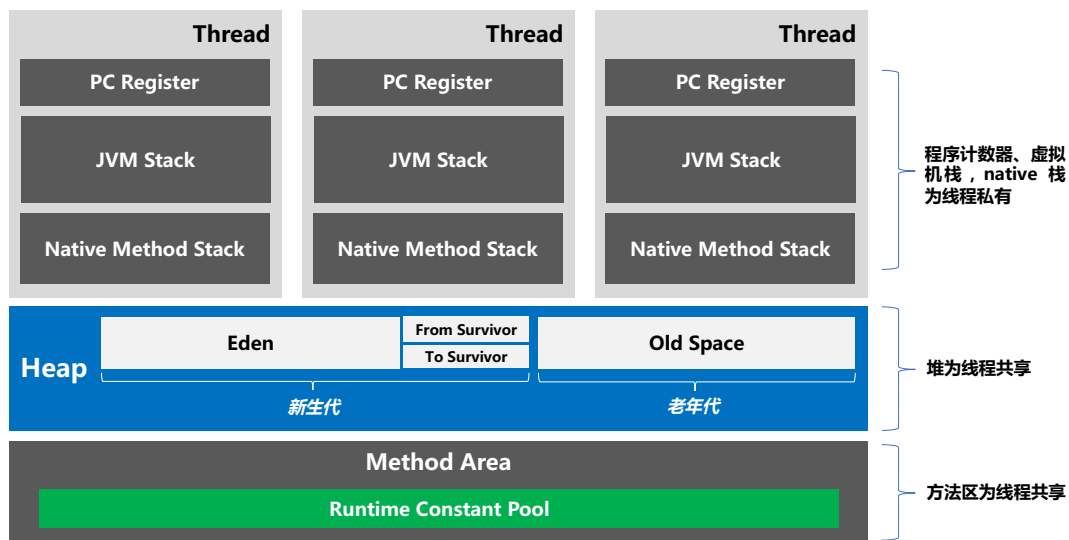


图 3.2 JVM 内存模型

堆 用来存放动态产生的数据，如 `new` 出来的对象和数组¹。（Heap）

常量池 JVM 为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量（基本类型、`String`）和对其他类型、方法、字段的符号引用。池中的数据和数组一样通过索引访问，常量池在 Java 程序的动态链接中起了核心作用。（Perm）

代码段 存放从硬盘上读取的源程序代码。（Perm）

数据段 存放 `static` 定义的静态成员。（Perm）

3.2 Java 程序内存运行分析

3.2.1 预备知识和所用讲解程序

1. 一个 Java 文件，只要有 `main` 入口方法，即可认为这是一个 Java 程序，可以单独编译运行。

¹注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆内存中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

2. 无论是普通类型的变量还是引用类型的变量（俗称实例），都可以作为局部变量，他们都可以出现在栈中。
3. 普通类型的变量在栈中直接保存它所对应的值，而引用类型的变量保存的是一个指向堆区的指针。通过这个指针，就可以找到这个实例在堆区对应的对象。因此，普通类型变量只在栈区占用一块内存，而引用类型变量要在栈区和堆区各占一块内存。

Code: Test.java

```
1 public class Test {
2     public static void main(String[] args) {
3         Test test = new Test(); //1
4         int data = 9; //2
5         BirthDate d1 = new BirthDate(22, 12, 1982); //3
6         BirthDate d2 = new BirthDate(10, 10, 1958); //4
7         test.m1(data); //5
8         test.m2(d1); //7
9         test.m3(d2);
10    }
11
12    public void m1(int i) {
13        i = 1234; //6
14    }
15    public void m2(BirthDate b) {
16        b = new BirthDate(15, 6, 2010); //8
17    }
18    public void m3(BirthDate b) {
19        b.setDay(18);
20    }
21 }
```

3.2.2 程序调用过程

程序调用过程（一）

- JVM 自动寻找 main 方法，执行第一句代码，创建一个 Test 类的实例，在栈中分配一块内存，存放一个指向堆区对象的指针 110925。
- 创建一个 int 型的变量 data，由于是基本类型，直接在栈中存放 data 对应的值 9。

- 创建两个 BirthDate 类的实例 d1、d2，在栈中分别存放了对应的指针指向各自的对象。它们在实例化时调用了有参数的构造方法，因此对象中有自定义初始值。

程序调用过程（二）

- 调用 test 对象的 m1 方法，以 data 为参数。JVM 读取这段代码时，检测到 i 是局部变量，则会把 i 放在栈中，并且把 data 的值赋给 i。

程序调用过程（三）

- 把 1234 赋值给 i。

程序调用过程（四）

- m1 方法执行完毕，立即释放局部变量 i 所占用的栈空间。

程序调用过程（五）

- 调用 test 对象的 m2 方法，以实例 d1 为参数。JVM 检测到 m2 方法中的 b 参数为局部变量，立即加入到栈中，由于是引用类型的变量，所以 b 中保存的是 d1 中的指针，此时 b 和 d1 指向同一个堆中的对象。在 b 和 d1 之间传递是指针。

程序调用过程（六）

- m2 方法中又实例化了一个 BirthDate 对象，并且赋给 b。在内部执行过程是：在堆区 new 了一个对象，并且把该对象的指针保存在栈中 b 对应空间，此时实例 b 不再指向实例 d1 所指向的对象，但是实例 d1 所指向的对象并无变化，未对 d1 造成任何影响。

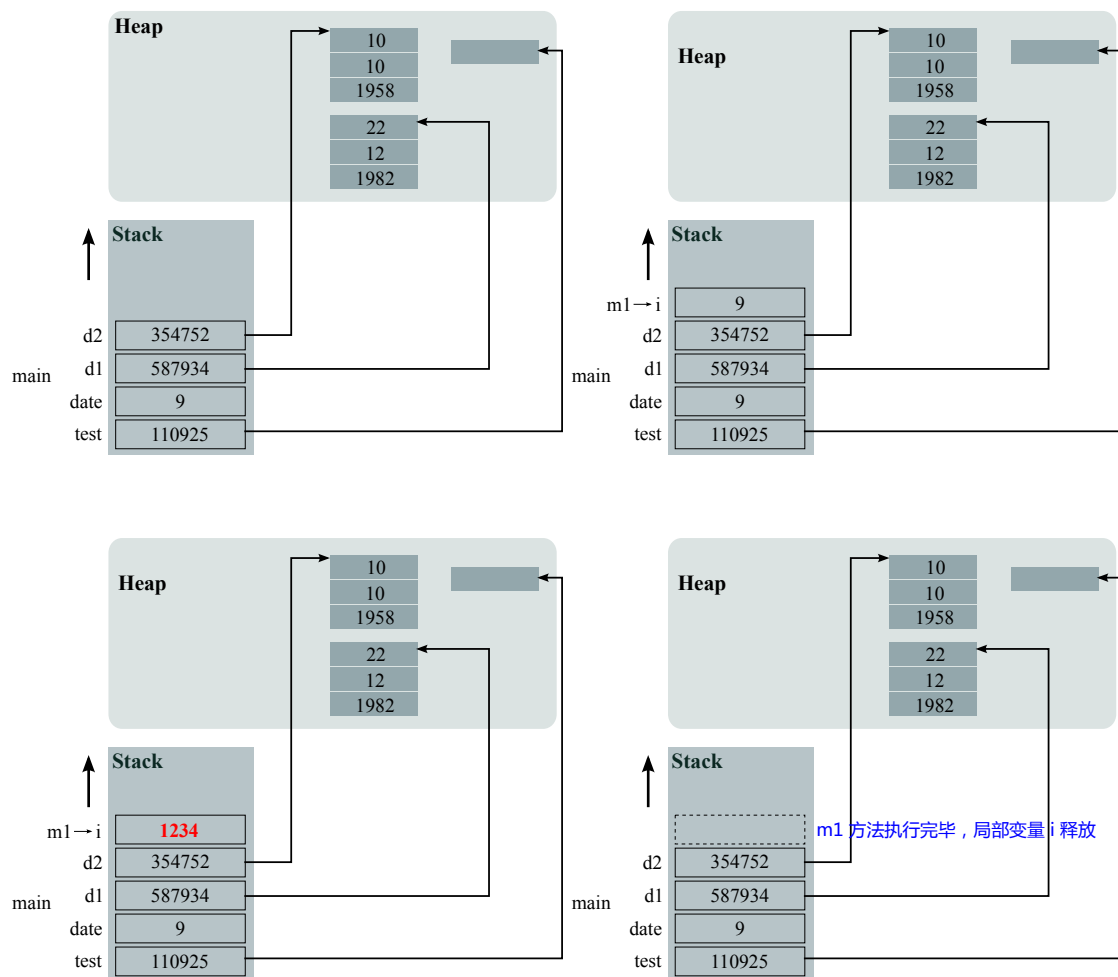
程序调用过程（七）

- m2 方法执行完毕，立即释放局部引用变量 b 所占的栈空间，注意只是释放了栈空间，堆空间要等待自动回收。

程序调用过程（八）

- 调用 test 实例的 m3 方法，以实例 d2 为参数。JVM 会在栈中为局部引用变量 b 分配空间，并且把 d2 中的指针存放在 b 中，此时 d2 和 b 指向同一个对象。再调用实例 b 的 setDay 方法，其实就是调用 d2 指向的对象的 setDay 方法。

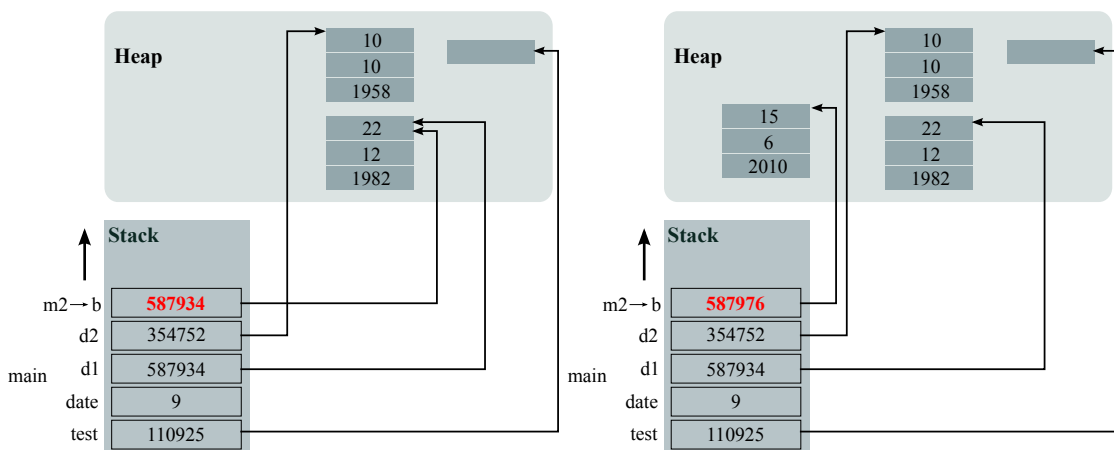
- 调用实例 b 的 setDay 方法会影响 d2，因为二者指向的是同一个对象。
- m3 方法执行完毕，立即释放局部引用变量 b。



3.2.3 Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中保存一个指向堆区的指针，真正的对象存放在堆中。
- 作为参数时基本类型就直接传值，引用类型传指针。

注意什么是对象



```
1 MyClass a = new MyClass();
```

此时 `a` 是指向对象的指针，而不能说 `a` 是对象。指针存储在栈中，对象存储在堆中，操作实例实际上是通过指针间接操作对象。多个指针可以指向同一个对象。

- **栈中的数据和堆中的数据销毁并不是同步的。**方法一旦执行结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁；而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。
- **栈、堆、代码段、数据段等都是相对于应用程序而言的。**每一个应用程序都对应唯一的一个 JVM 实例，每一个 JVM 实例都有自己的内存区域，互不影响，并且这些内存区域是该 JVM 实例所有线程共享的。

3.3 Java 内存管理建议

3.3.1 Java 垃圾回收机制

JVM 的垃圾回收机制（GC）决定对象是否是垃圾对象，并进行回收。垃圾回收机制的特点包括：

- 垃圾内存并不是用完了马上就被释放，所以会产生内存释放不及时的现象，从而降低内存的使用效率。而当程序庞大的时候，这种现象更为明显。
- 垃圾回收工作本身需要消耗资源，同样会产生内存浪费。

JVM 中的对象生命周期一般分为 7 个阶段：①创建阶段、②应用阶段、③不可视阶段、④不可到达阶段、⑤可收集阶段、⑥终结阶段、⑦释放阶段。

Java 需要内存管理，在 JVM 中运行的对象的整个生命周期中，进行人为的内存管理是必要的，主要原因体现在：

- 虽然 JVM 已经代替开发者完成了对内存的管理，但是硬件本身的资源是有限的。
- 如果 Java 的开发人员不注意内存的使用依然会造成较高的内存消耗，导致性能的降低。

3.3.2 JVM 内存溢出和参数调优

当遇到 `OutOfMemoryError` 时该如何做？

- 常见的 OOM（Out Of Memory）内存溢出异常，就是堆内存空间不足以存放新对象实例时导致。
- 永久区内存溢出相对少见，一般是由于需要加载海量的 Class 数据，超过了非堆内存的容量导致。通常出现在 Web 应用刚刚启动时。因此 Web 应用推荐使用预加载机制，方便在部署时就发现并解决该问题。
- 栈内存也会溢出，但是更加少见。

对内存溢出的处理方法不外乎这两种：❶ 调整 JVM 内存配置；❷ 优化代码。
创建阶段的 JVM 内存配置优化需要关注以下项：

堆内存优化 调整 JVM 启动参数 `-Xms -Xmx -XX:newSize -XX:MaxNewSize`，如调整初始堆内存和最大对内存 `-Xms256M -Xmx512M`。或者调整初始 New Generation 的初始内存和最大内存 `-XX:newSize=128M -XX:MaxNewSize=128M`。

永久区内存优化 调整 `PermSize` 参数，如 `-XX:PermSize=256M -XX:MaxPermSize=512M`。

栈内存优化 调整每个线程的栈内存容量，如 `-Xss2048K`。

3.3.3 内存优化的小示例

减少无谓的对象引用创建

[Code: Test 1](#)

```
1  for(int i=0; i<10000; i++) {  
2      Object obj = new Object();  
3  }
```

Code: Test 2

```
1  Object obj = null;  
2  for( int i=0; i<10000; i++) {  
3      obj = new Object();  
4  }
```

内存性能分析

Test 2 比 Test 1 的性能要好。两段程序每次执行 for 循环都要创建一个 Object 的临时对象，JVM 的垃圾回收不会马上销毁但这些临时对象。相对于 Test 1，Test 2 则只在栈内存中保存一份对象的引用，而不必创建大量新临时变量，从而降低了内存的使用。

不要对同一对象初始化多次

```
1  public class A {  
2      private Hashtable table = new Hashtable();  
3      public A() {  
4          table = new Hashtable();  
5      }  
6  }
```

内存性能分析

上述代码 new 了两个 Hashtable，但是却只使用了一个，另外一个则没有被引用而被忽略掉，浪费了内存。并且由于进行了两次 new 操作，也影响了代码的执行速度。另外，不要提前创建对象，尽量在需要的时候创建对象。

3.3.4 对象其他生命周期阶段内存管理

应用 即该对象至少有一个引用在维护它。

不可视 即超出该变量的作用域。

因为 JVM GC 并不是马上进行回收，而是要判断对象是否被其他引用维护。所以，如果我们在使用完一个对象以后对其进行 `obj = null` 或者

obj.doSomething() 操作，将其标记为空，则帮助 JVM 及时发现这个垃圾对象。

不可到达 即在 JVM 中找不到对该对象的直接或者间接的引用。

可收集，终结，释放 垃圾回收器发现该对象不可到达，finalize 方法已经被执行，或者对象空间已被重用的时候。

Java 的 finalize() 方法

Java 所有类都继承自 Object 类，而 finalize() 是 Object 类的一个函数，该函数在 Java 中类似于 C++ 的析构函数（仅仅是类似）。一般来说可以通过重载 finalize() 的形式来释放类中对象。

```
1 public class A {  
2     public Object a;  
  
4     public A() {  
5         a = new Object();  
6     }  
  
8     protected void finalize () throws java.lang.Throwable {  
9         a = null; // 标记为空，释放对象  
10        super. finalize (); // 递归调用超类中的 finalize 方法  
11    }  
12 }
```

什么时候 finalize() 被调用由 JVM 来决定。尽量少用 finalize() 函数，finalize() 函数是 Java 提供给程序员一个释放对象或资源的机会。但它会加大 GC 的工作量，因此尽量少采用 finalize 方式回收资源。

- 一般的，纯 Java 编写的 Class 不需要重写 finalize() 方法，因为 Object 已经实现了一个默认的，除非我们要实现特殊的功能。
- 用 Java 以外的代码编写的 Class(比如 JNI、C++ 的 new 方法分配的内存)，垃圾回收器并不能对这些部分进行正确的回收，这就需要我们覆盖默认的方法来实现对这部分内存的正确释放和回收。

实验设计
