innovatis

Java OOP

Part 1

First Object

1 Write your class

```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
}
```

instance variables

a method

2 Write a tester (TestDrive) class

just a main method (we're gonna put code in it in the next step)

```
class DogTestDrive {
  public static void main (String[] args) {
    // Dog test code goes  here
  }
}
```

3 In your tester, make an object and access the object's variables and methods

```
class DogTestDrive {
   public static void main (String[] args) {
     Dog d = new Dog();
     d.size = 40;
     d.bark();
   }
}
```

make a Dog object

use the dot operator (.) to set the size of the Dog

and to call its bark() method

dot operator

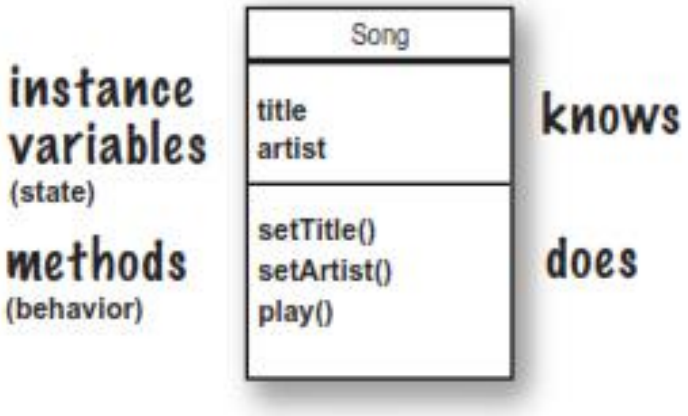
If you already have some OO savvy, you'll know we're not using encapsulation. We'll get there in chapter 4.

Class vs Object

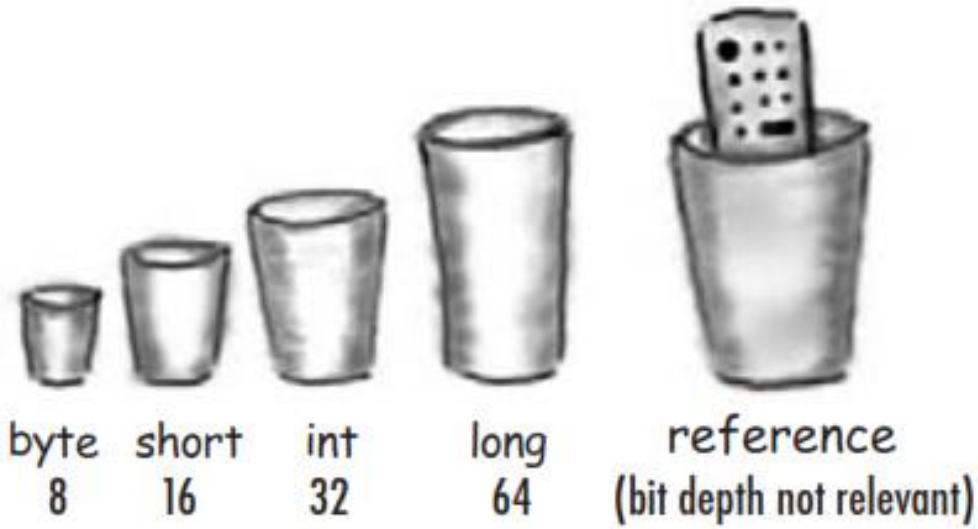
Remember: a class describes what an object knows and what an object does

A class is the blueprint for an object. When you write a class, you're describing how the JVM should make an object of that type. You already know that every object of that type can have different *instance variable* values. But what about the methods?

Can every object of that type have different method behavior?



Primitive VS Object



An object reference is just another variable value.

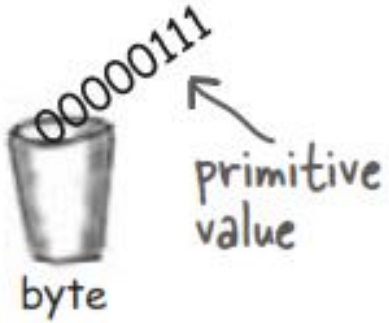
Something that goes in a cup.
Only this time, the value is a remote control.

Continue...

Primitive Variable

```
byte x = 7;
```

The bits representing 7 go into the variable. (00000111).

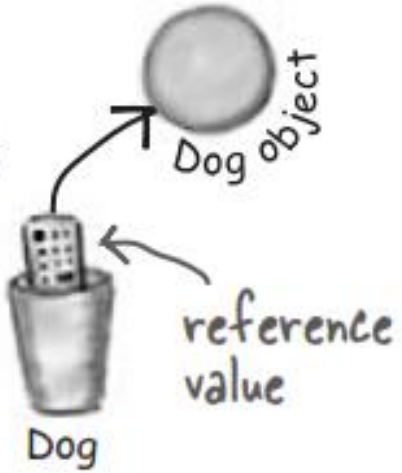


Reference Variable

```
Dog myDog = new Dog();
```

The bits representing a way to get to the Dog object go into the variable.

The Dog object itself does not go into the variable!



3 Steps

The 3 steps of object declaration, creation and assignment

```
1          3   2
Dog myDog = new Dog();
```

1 Declare a reference variable

```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a reference variable, and names that variable *myDog*. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

Dog

2 Create an object

```
Dog myDog = new Dog();
```

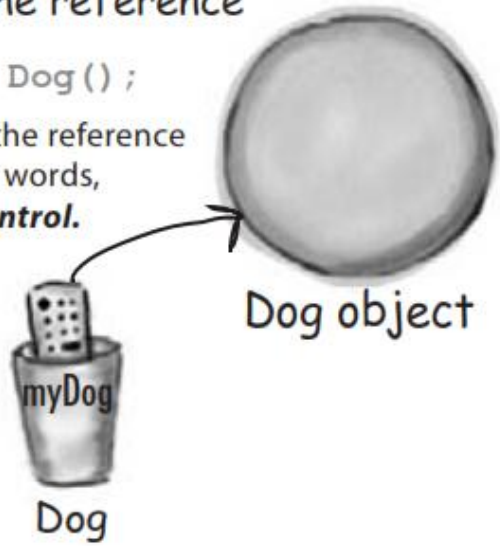
Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in chapter 9.)

Dog object

3 Link the object and the reference

```
Dog myDog = new Dog();
```

Assigns the new Dog to the reference variable myDog. In other words, ***programs the remote control.***



Dog object

Dog

Object Life Cycle

Life and death on the heap

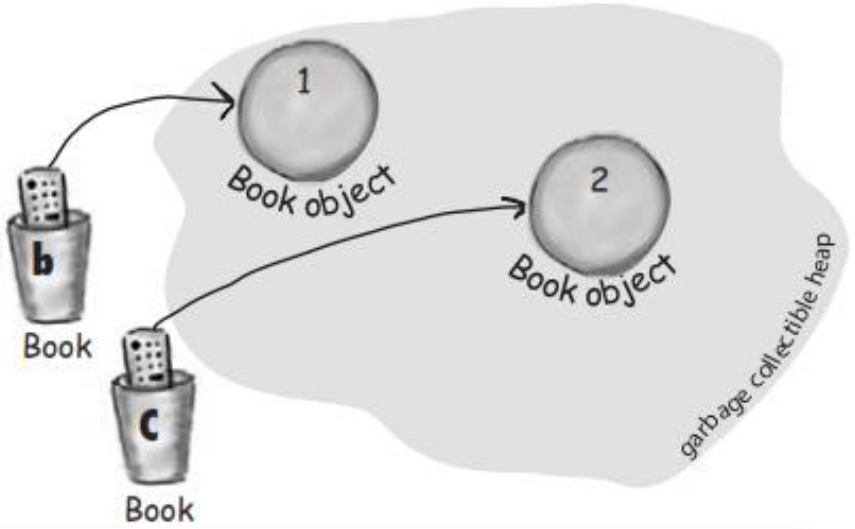
```
Book b = new Book();
Book c = new Book();
```

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2




```
b = c;
```

Assign the value of variable ***c*** to variable ***b***. The bits inside variable ***c*** are copied, and that new copy is stuffed into variable ***b***. Both variables hold identical values.

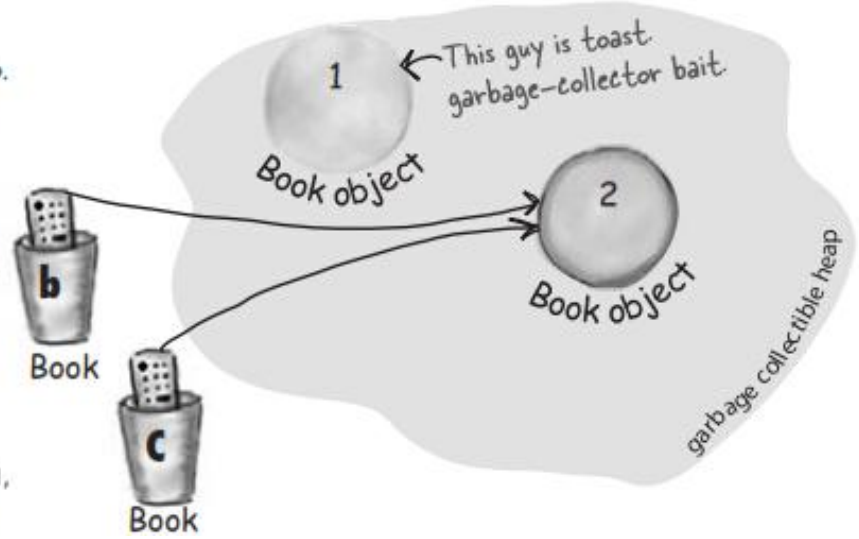
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The first object that ***b*** referenced, Object 1, has no more references. It's *unreachable*.



Continue...

```
c = null;
```

Assign the value `null` to variable **c**. This makes **c** a *null reference*, meaning it doesn't refer to anything. But it's still a reference variable, and another Book object can still be assigned to it.

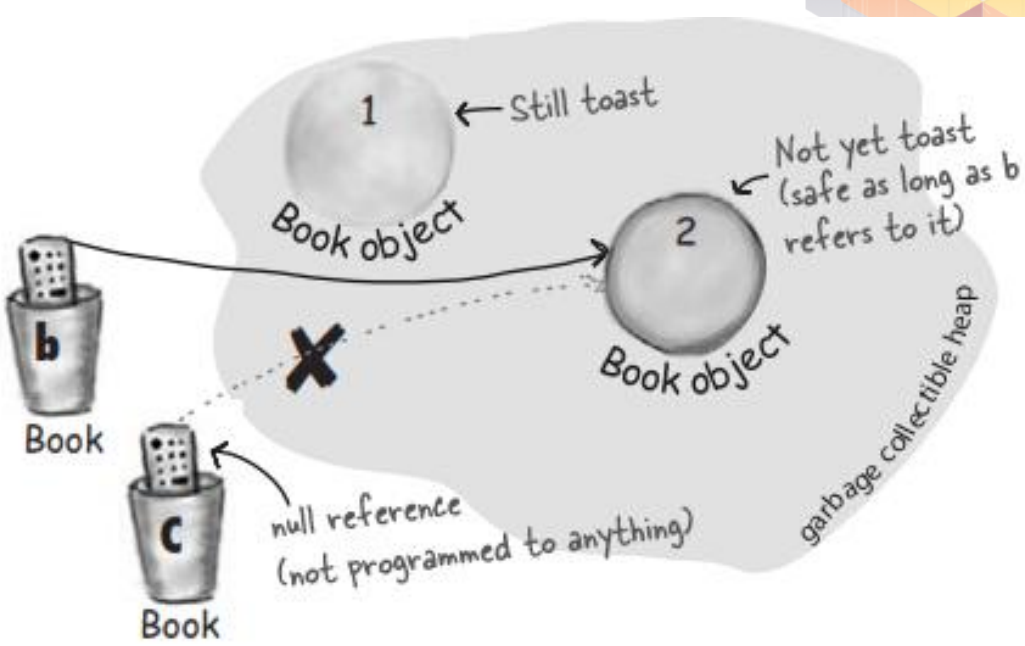
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

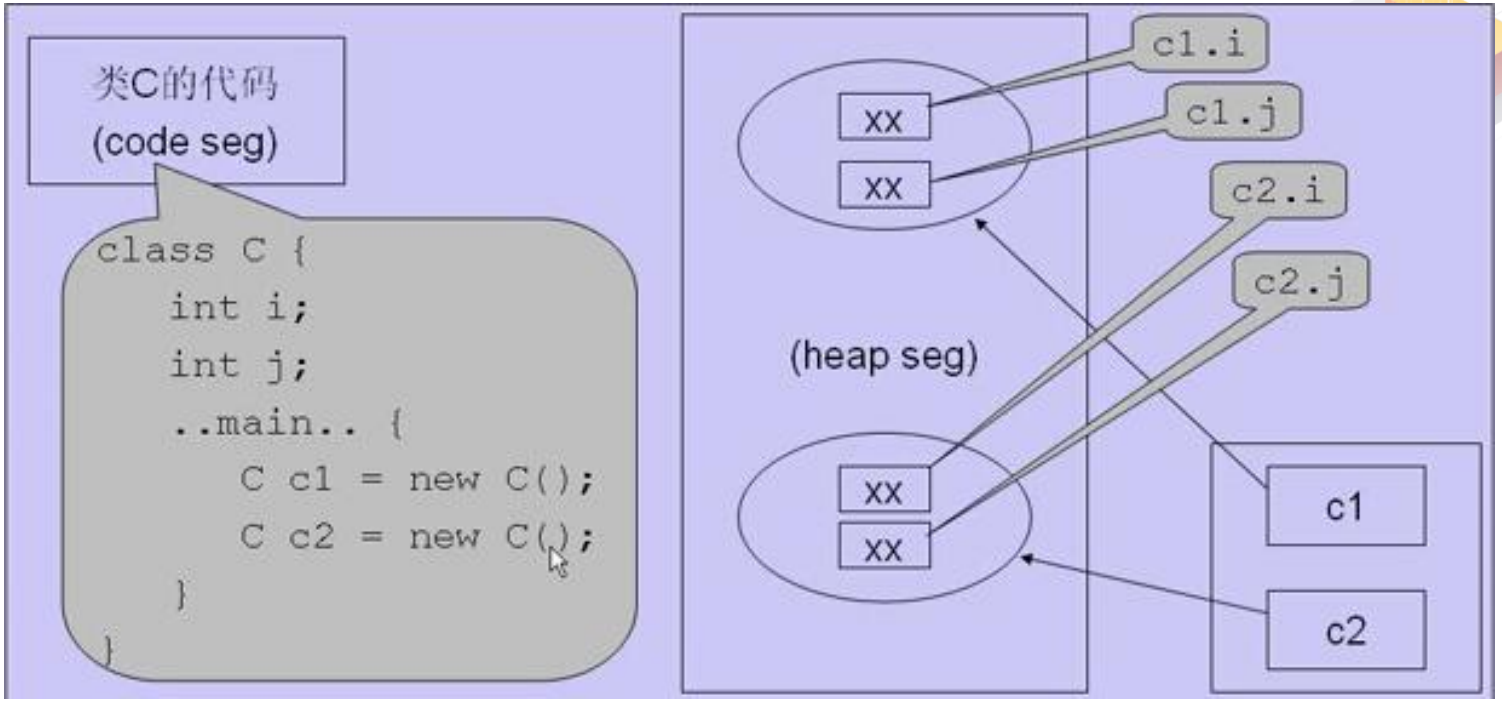
null References: 1

Reachable Objects: 1

Abandoned Objects: 1

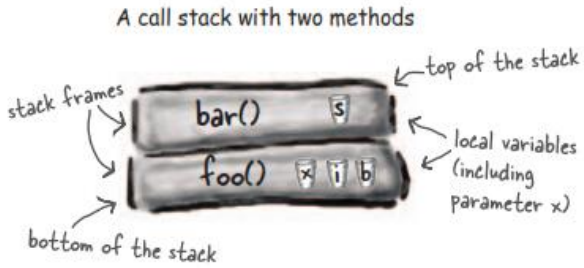


Continue...


类C的代码
(code seg)
class C {
int i;
int j;
..main.. {
C c1 = new C();
C c2 = new C();
}
}
c1.i
c1.j
c2.i
c2.j
xx
xx
xx
xx
(heap seg)
c1
c2

Stack VS Heap

- **Heap: Where objects live**
- **Stack: Where methods and local variables live**



The method on the top of the stack is always the currently-executing method.

Stack Scenario

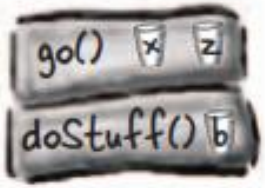
```
public void doStuff() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}
public void crazy() {
   char c = 'a';
}
```

A stack scenario

The code on the left is a snippet (we don't care what the rest of the class looks like) with three methods. The first method (*doStuff()*) calls the second method (*go()*), and the second method calls the third (*crazy()*). Each method declares one local variable within the body of the method, and method *go()* also declares a parameter variable (which means *go()* has two local variables).

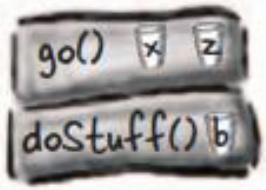
1. Code from another class calls **doStuff()**, and **doStuff()** goes into a stack frame at the top of the stack. The boolean variable named 'b' goes on the **doStuff()** stack frame.

2. **doStuff()** calls **go()**. **go()** is *pushed* on top of the stack. Variables 'x' and 'z' are in the **go()** stack frame.



3. **go()** calls **crazy()**. **crazy()** is now on the top of the stack, with variable 'c' in the frame.

4. **crazy()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **go()** method, and picks up at the line following the call to **crazy()**.



Instance vs local variable

Instance Variables

Instance variables are declared inside a *class* but *not* inside a method. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {
    int size;
}
```

Every Duck has a "size" instance variable.

Local Variables

Local variables are declared inside a *method*, including method parameters. They're temporary, and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

```
public void foo(int x) {
    int i = x + 3;
    boolean b = true;
}
```

The parameter x and the variables i and b are all local variables.

- **Instance variable has default value**
- **Local variable need to be initialized before using**

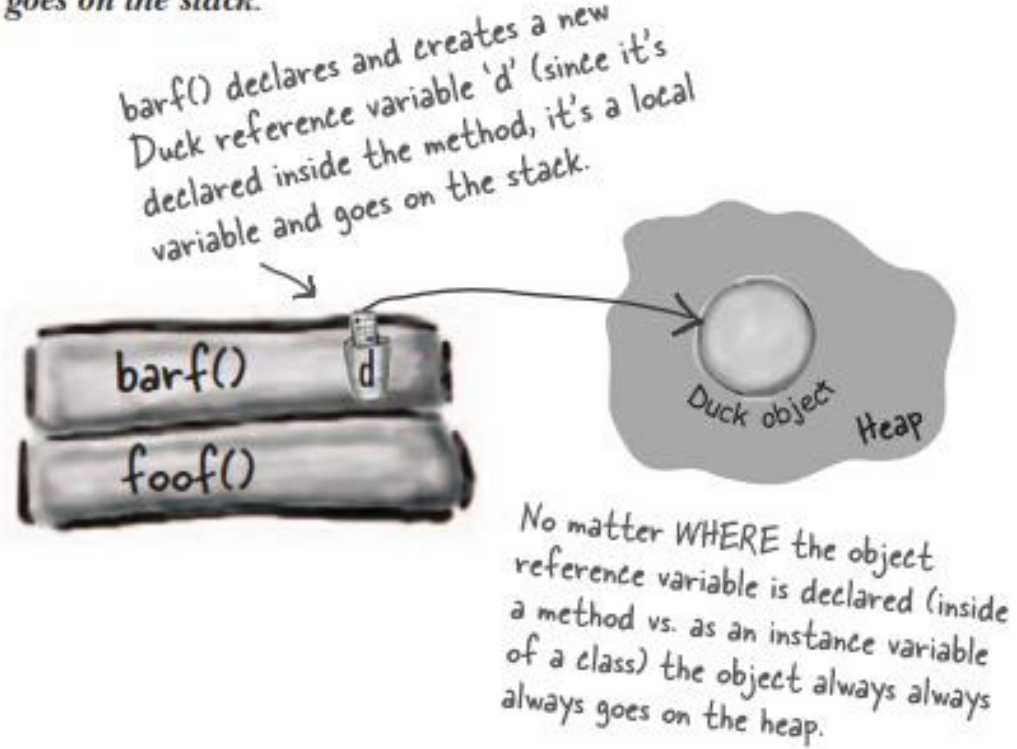
Local variable that is a object

Remember, a non-primitive variable holds a *reference* to an object, not the object itself. You already know where objects live—on the heap. It doesn't matter where they're declared or created. ***If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack.***

The object itself still goes in the heap.

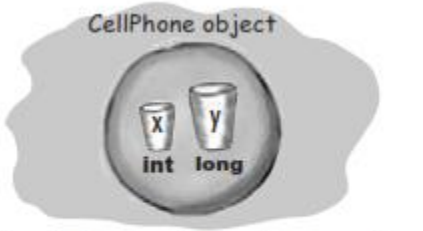
```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck(24);
   }
}
```

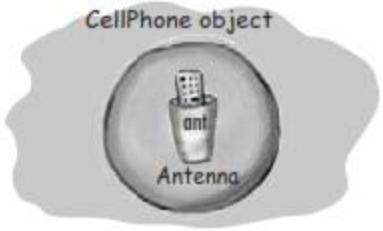


Where is instance variable?

- It's on the heap, inside the object
- What if instance variables are objects?

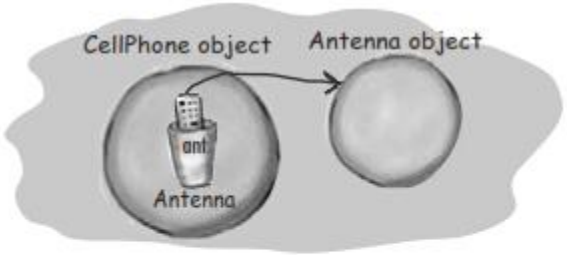


Object with two primitive instance variables. Space for the variables lives in the object.



Object with one non-primitive instance variable—a reference to an Antenna object, but no actual Antenna object. This is what you get if you declare the variable but don't initialize it with an actual Antenna object.

```
public class CellPhone {
  private Antenna ant;
}
```



Object with one non-primitive instance variable, and the Antenna variable is assigned a new Antenna object.

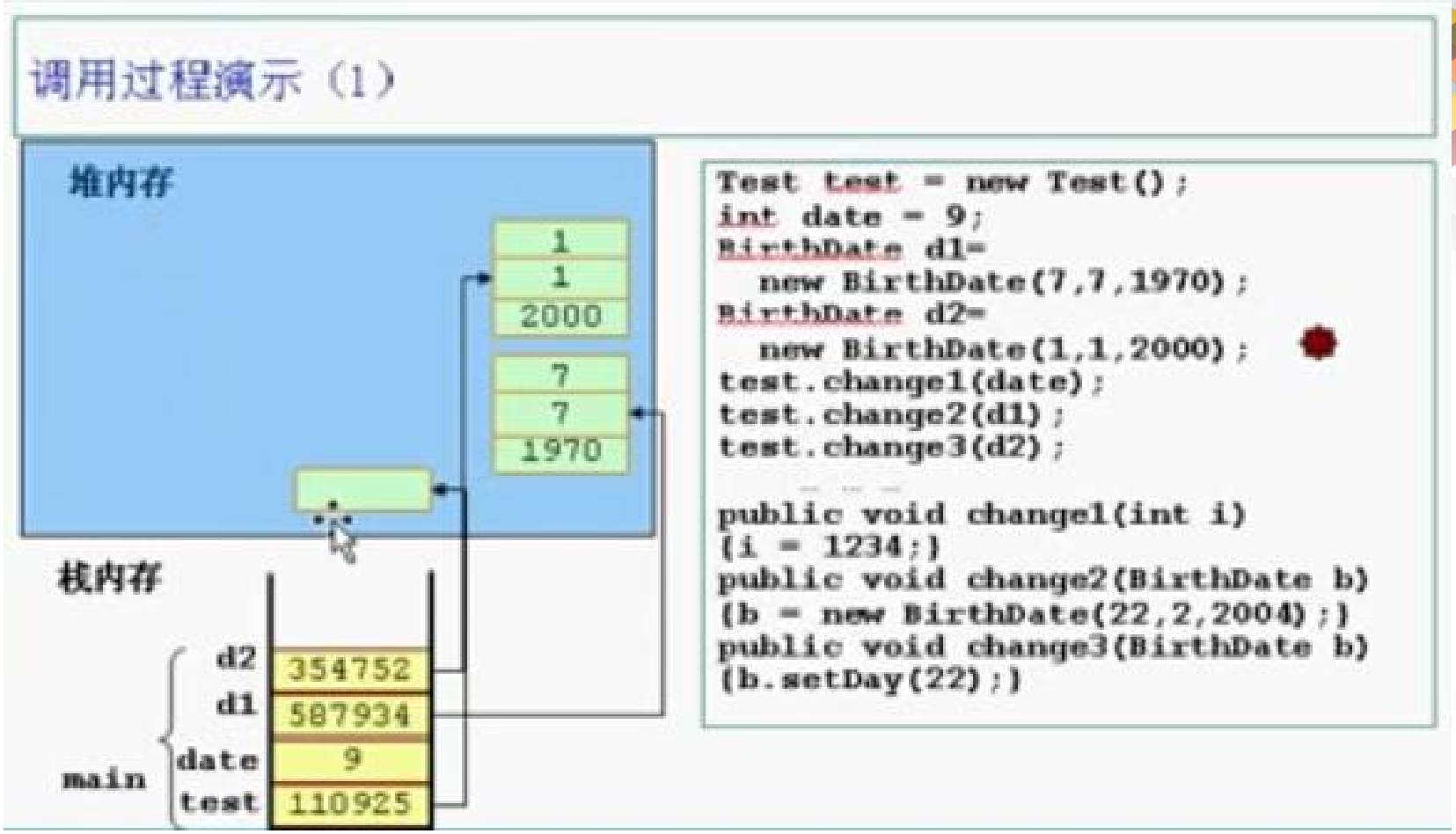
```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

Code example

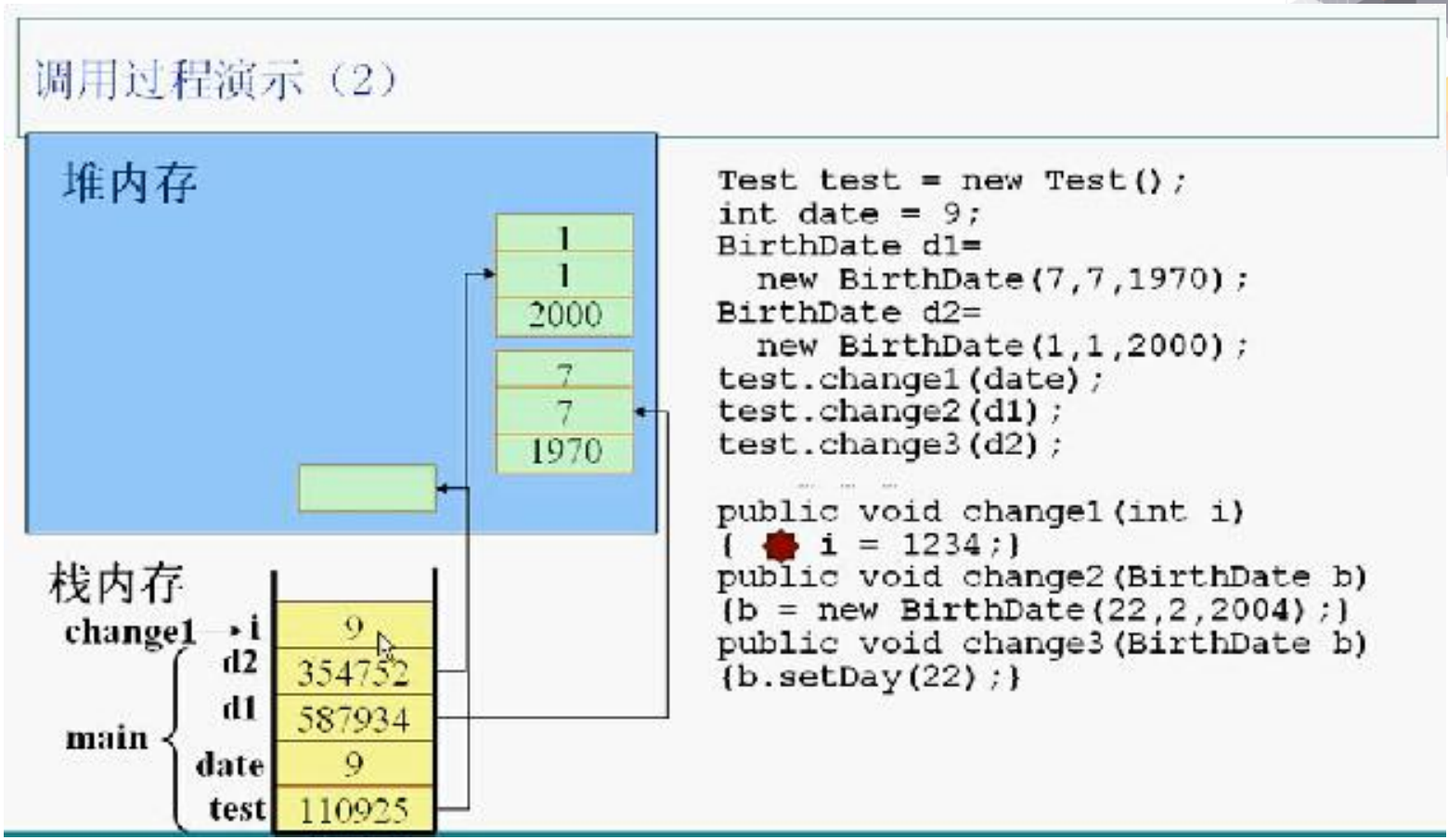
```
//定义如下类:
class BirthDate {
    private int day;
    private int month;
    private int year;
    public BirthDate(int d, int m, int y) {
        day = d; month = m; year = y;
    }
    public void setDay(int d) {day = d;}
    public void setMonth(int m) {month = m;}
    public void setYear(int y) {year = y;}
    public int getDay() {return day;}
    public int getMonth() {return month;}
    public int getYear() {return year;}
    public void display() {
       System.out.println
        (day + " - " + month + " - " + year);
    }
}
```

```
public class Test{
    public static void main(String args[]){
        Test test = new Test();
        int date = 9;
        BirthDate d1= new BirthDate(7,7,1970);
        BirthDate d2= new BirthDate(1,1,2000);
        test.change1(date);
        test.change2(d1);
        test.change3(d2);
        System.out.println("date=" + date);
        d1.display();
        d2.display();
    }
    public void change1(int i){i = 1234;}
    public void change2(BirthDate b)
    {b = new BirthDate(22,2,2004);}
    public void change3(BirthDate b)
    {b.setDay(22);}
}
```

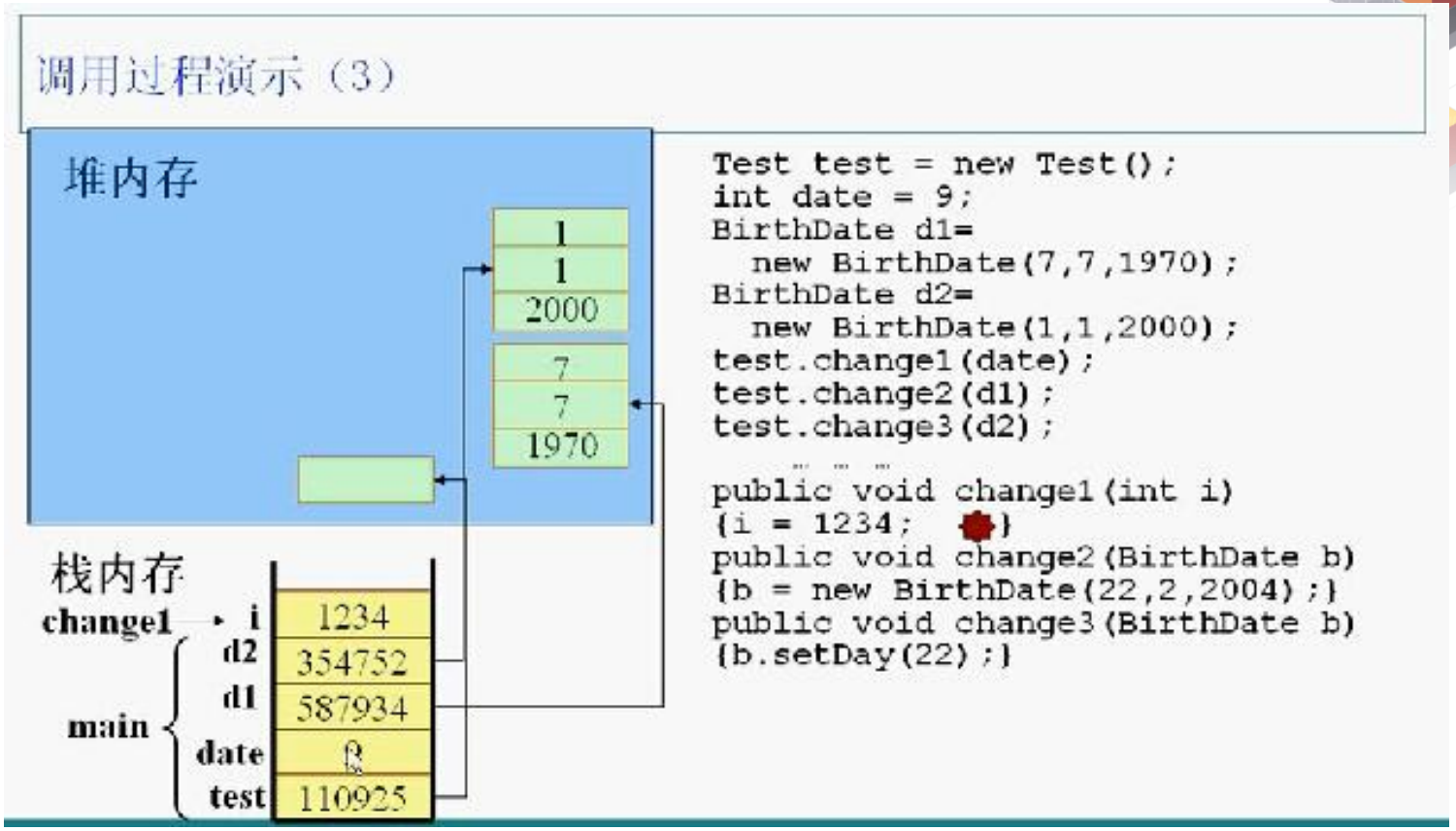
调用过程


调用过程演示（1）
堆内存
1
1
2000
7
7
1970
栈内存
d2
354752
d1
587934
date
9
main
test
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

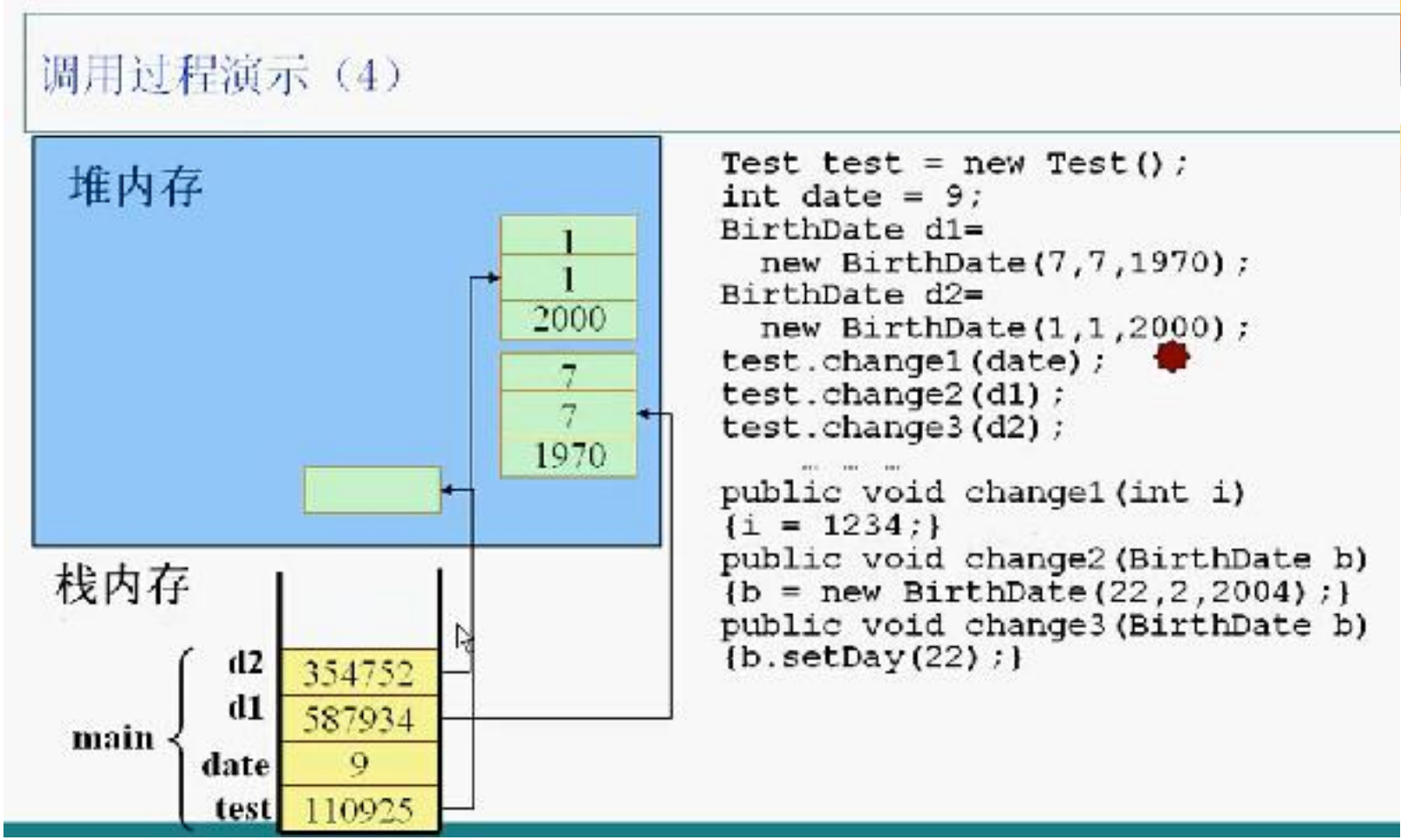
Continue...


调用过程演示（2）
堆内存
1
1
2000
7
7
1970
栈内存
change1
i
d2
d1
main
date
test
9
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{ i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

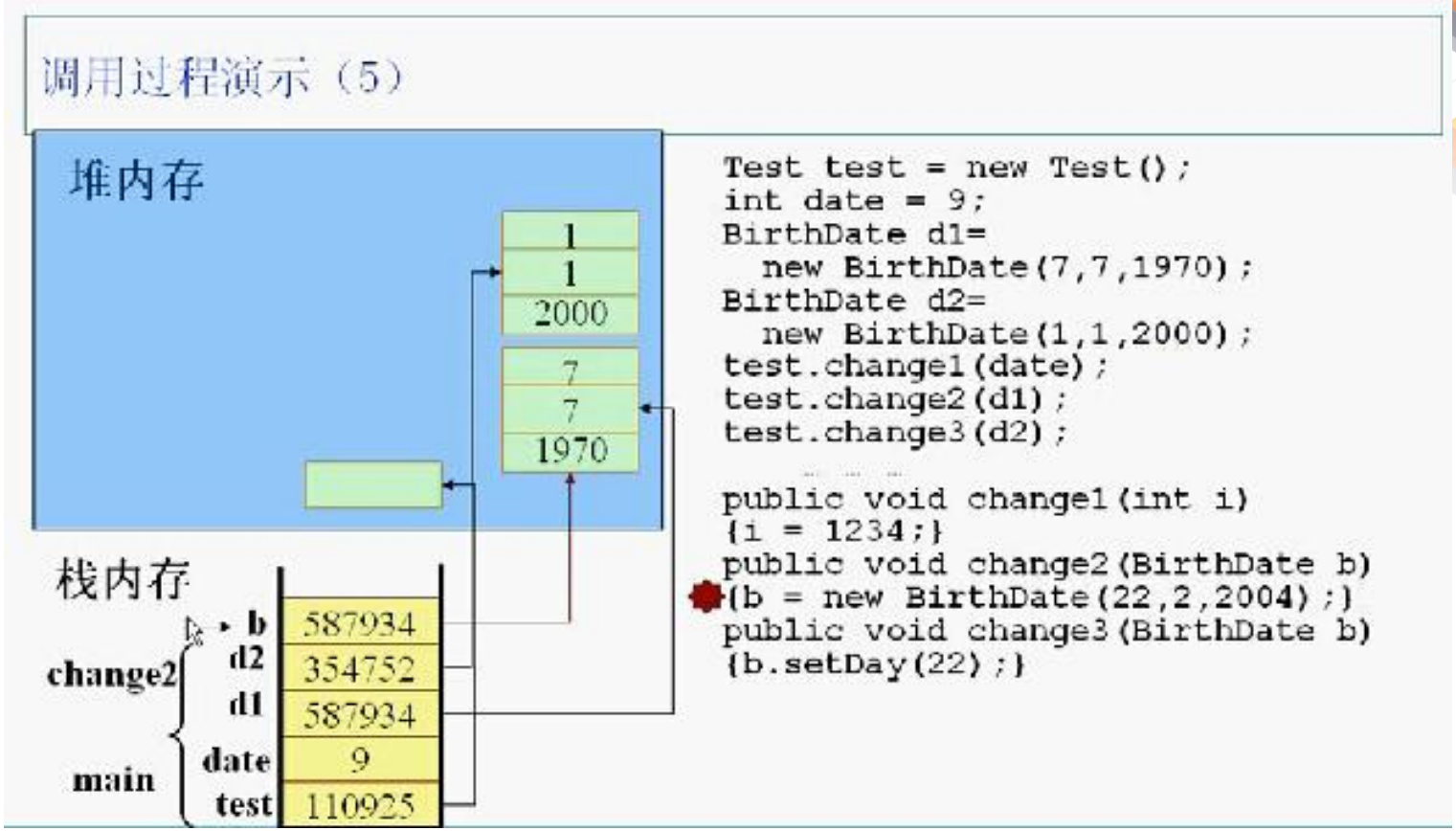
Continue...


调用过程演示（3）
堆内存
1
1
2000
7
7
1970
栈内存
change1
i
d2
d1
main
date
test
1234
354752
587934
8
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

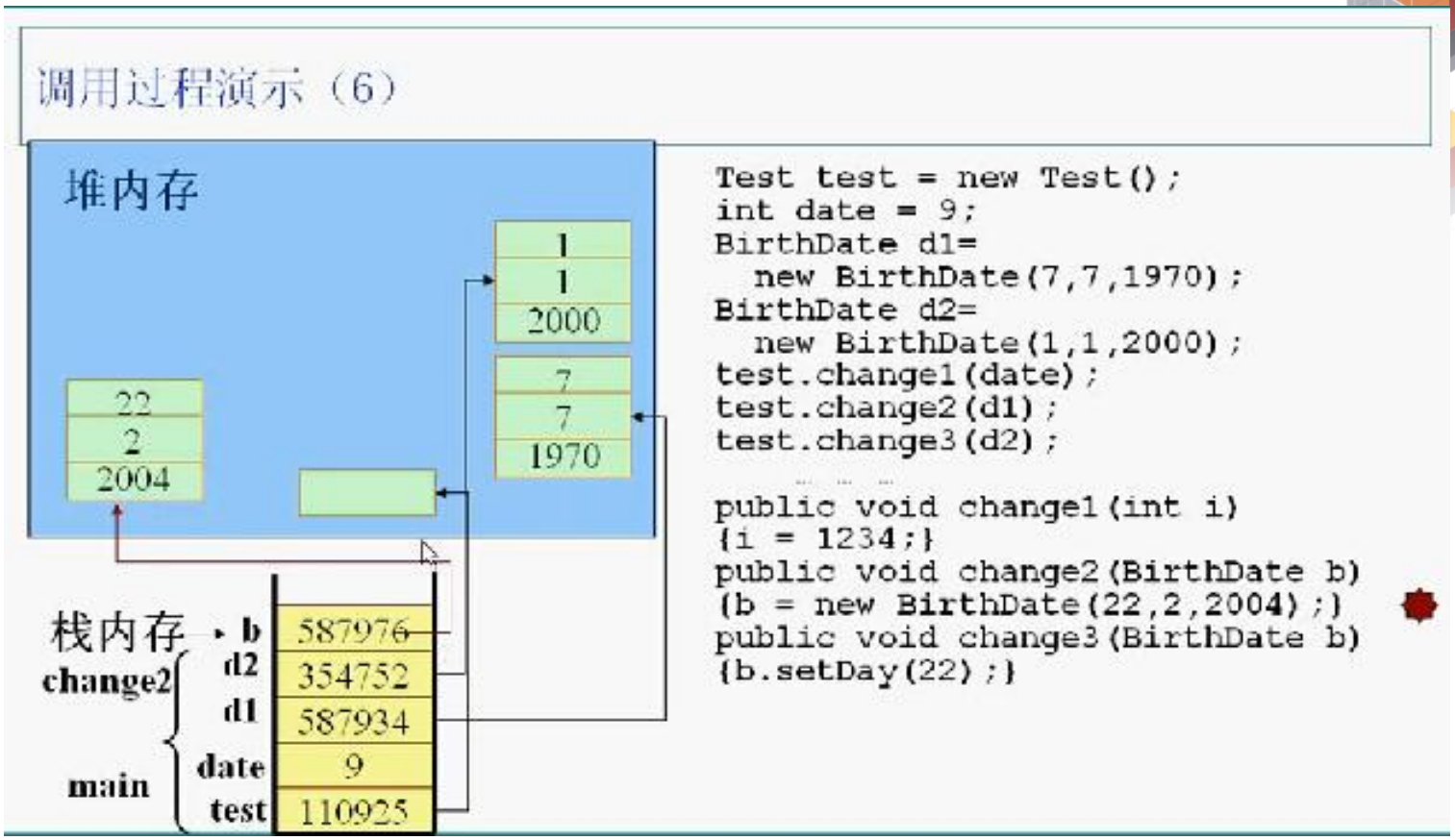
Continue...


调用过程演示（4）
堆内存
1
1
2000
7
7
1970
栈内存
main
d2
d1
date
test
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

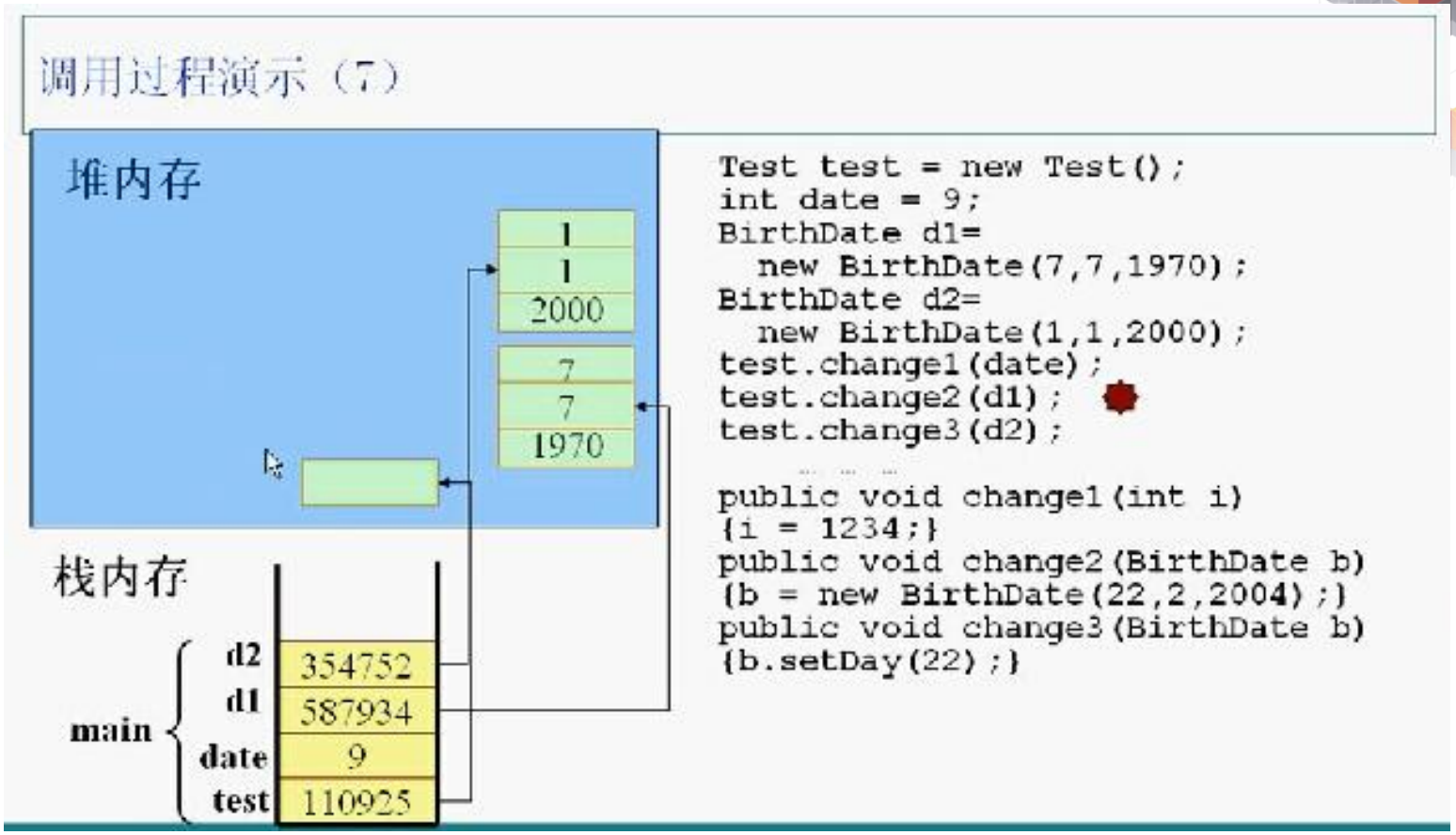
Continue...


调用过程演示（5）
堆内存
1
1
2000
7
7
1970
栈内存
b
d2
d1
date
test
change2
main
587934
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

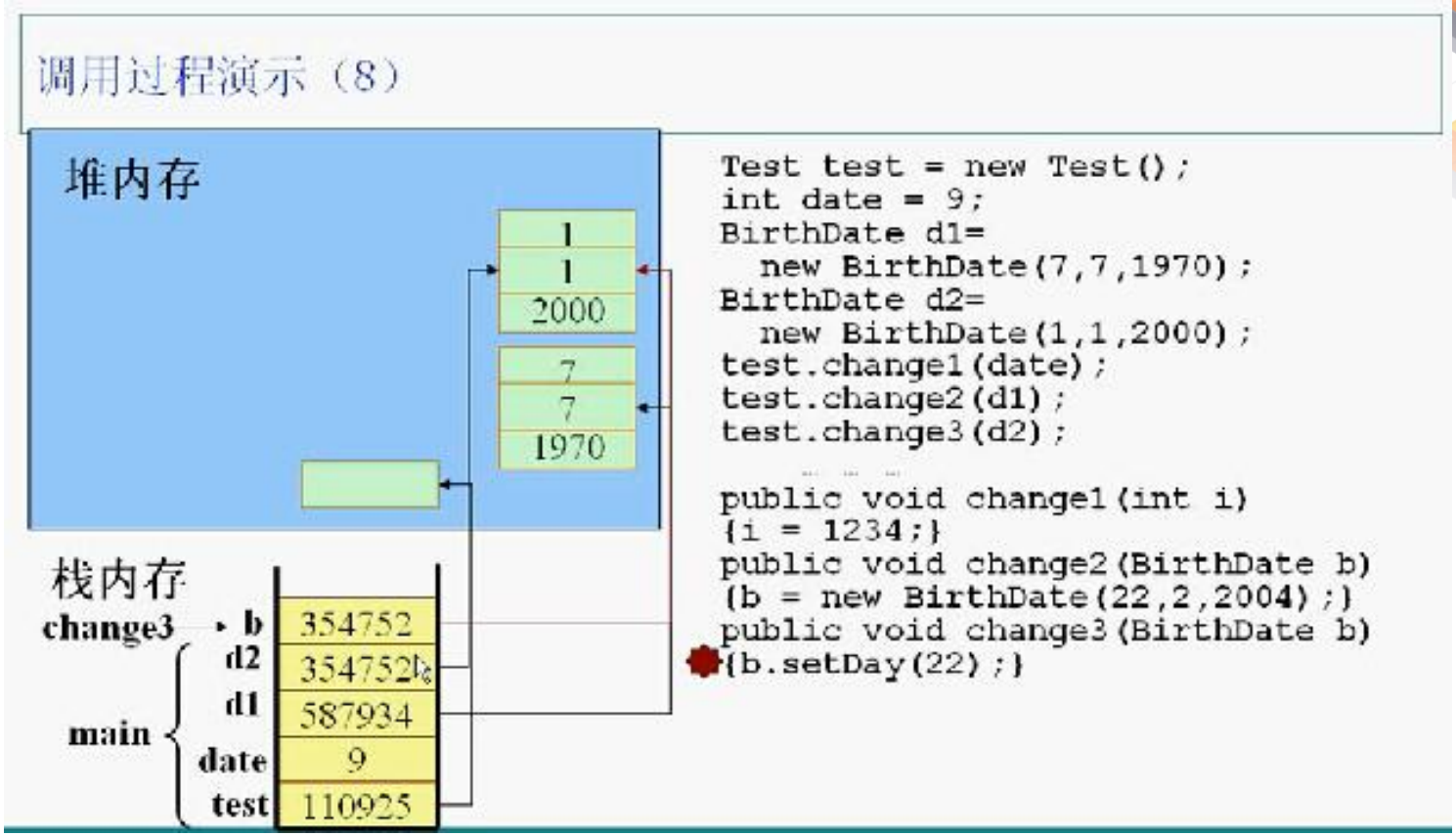
Continue...


调用过程演示（6）
堆内存
1
1
2000
7
7
1970
22
2
2004
栈内存
b
d2
d1
date
test
change2
main
587976
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

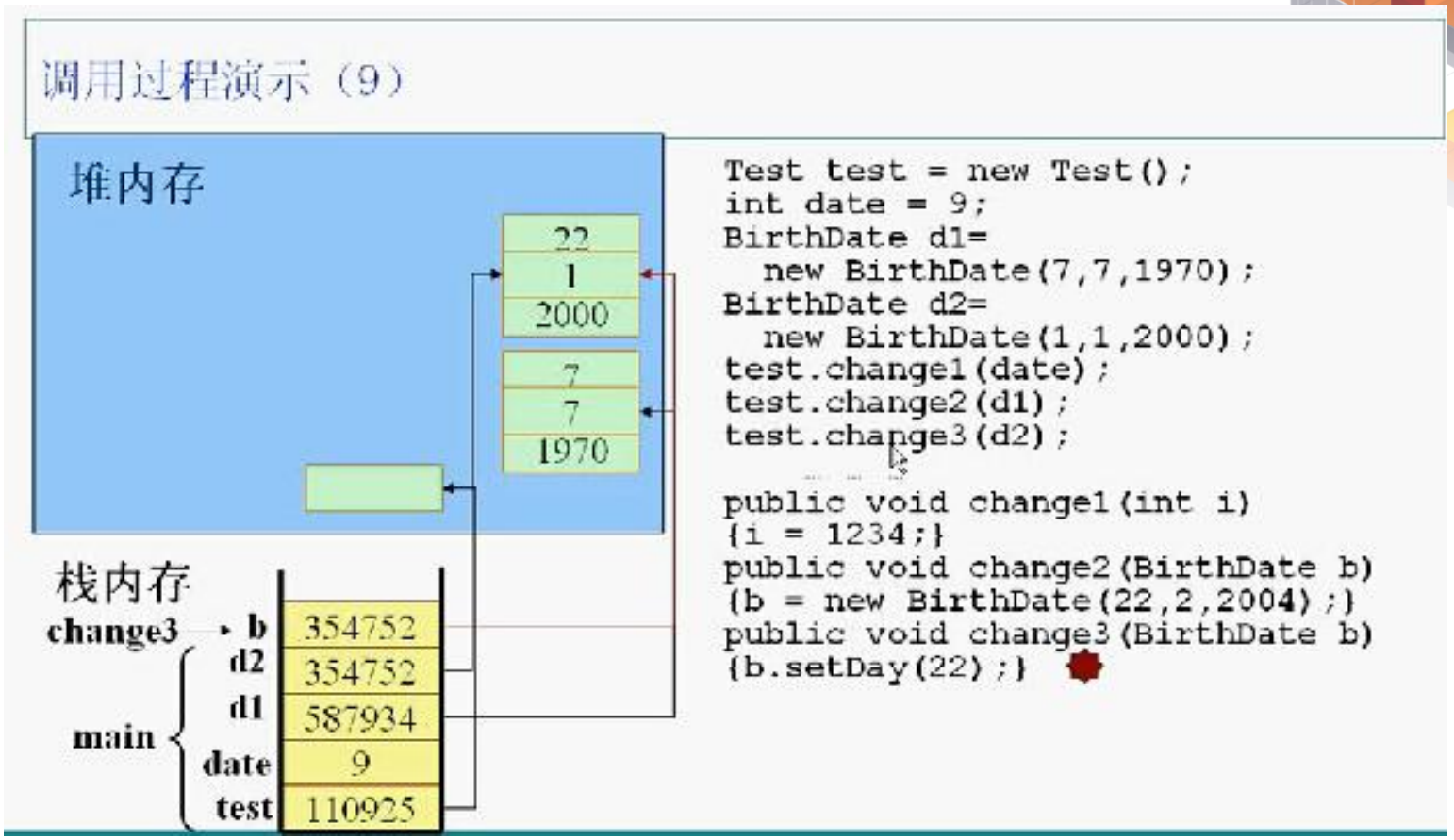
Continue...


调用过程演示（7）
堆内存
1
1
2000
7
7
1970
栈内存
main
d2
d1
date
test
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

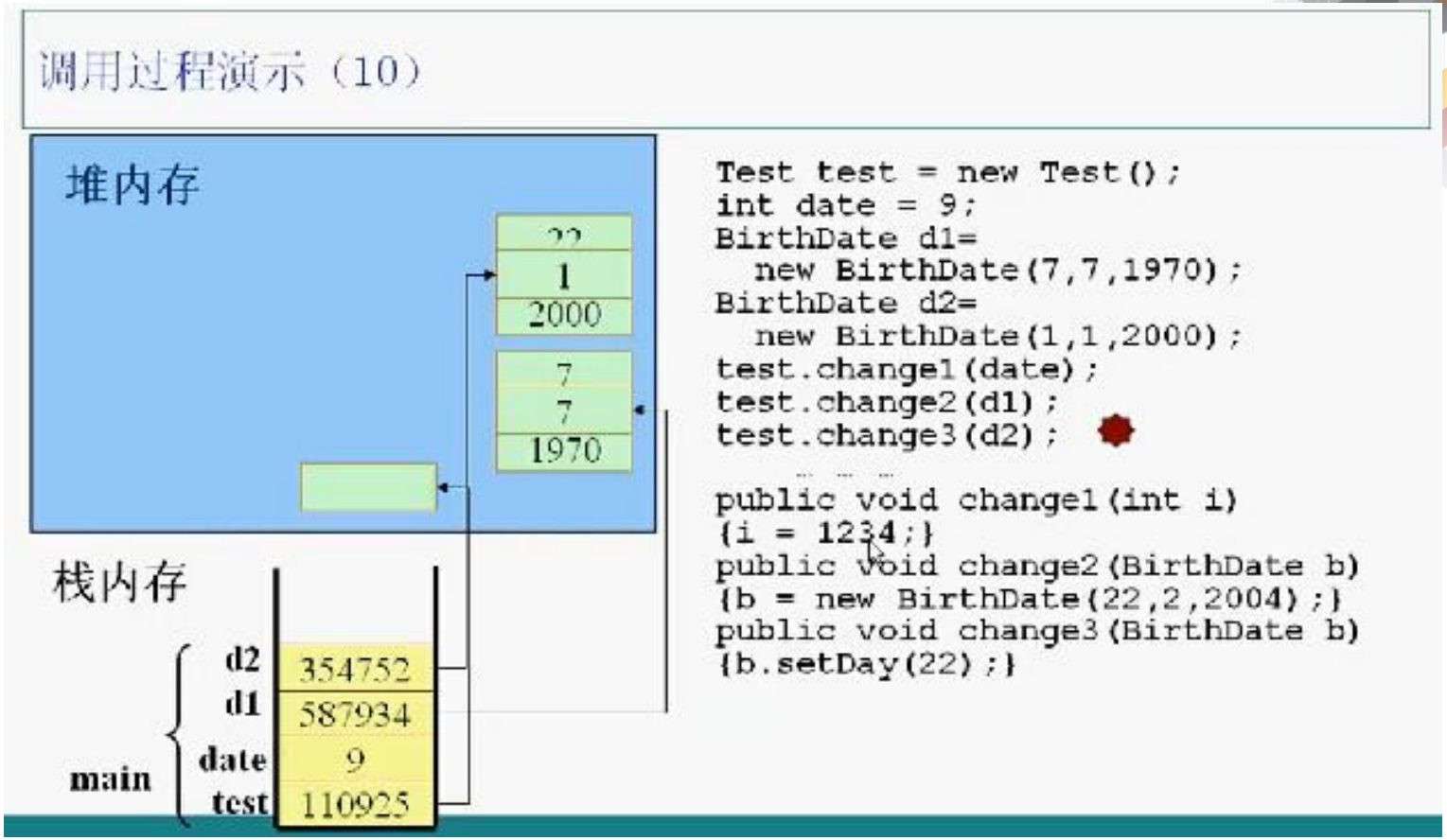
Continue...


调用过程演示（8）
堆内存
1
1
2000
7
7
1970
栈内存
change3
b
d2
main
d1
date
test
354752
354752h
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

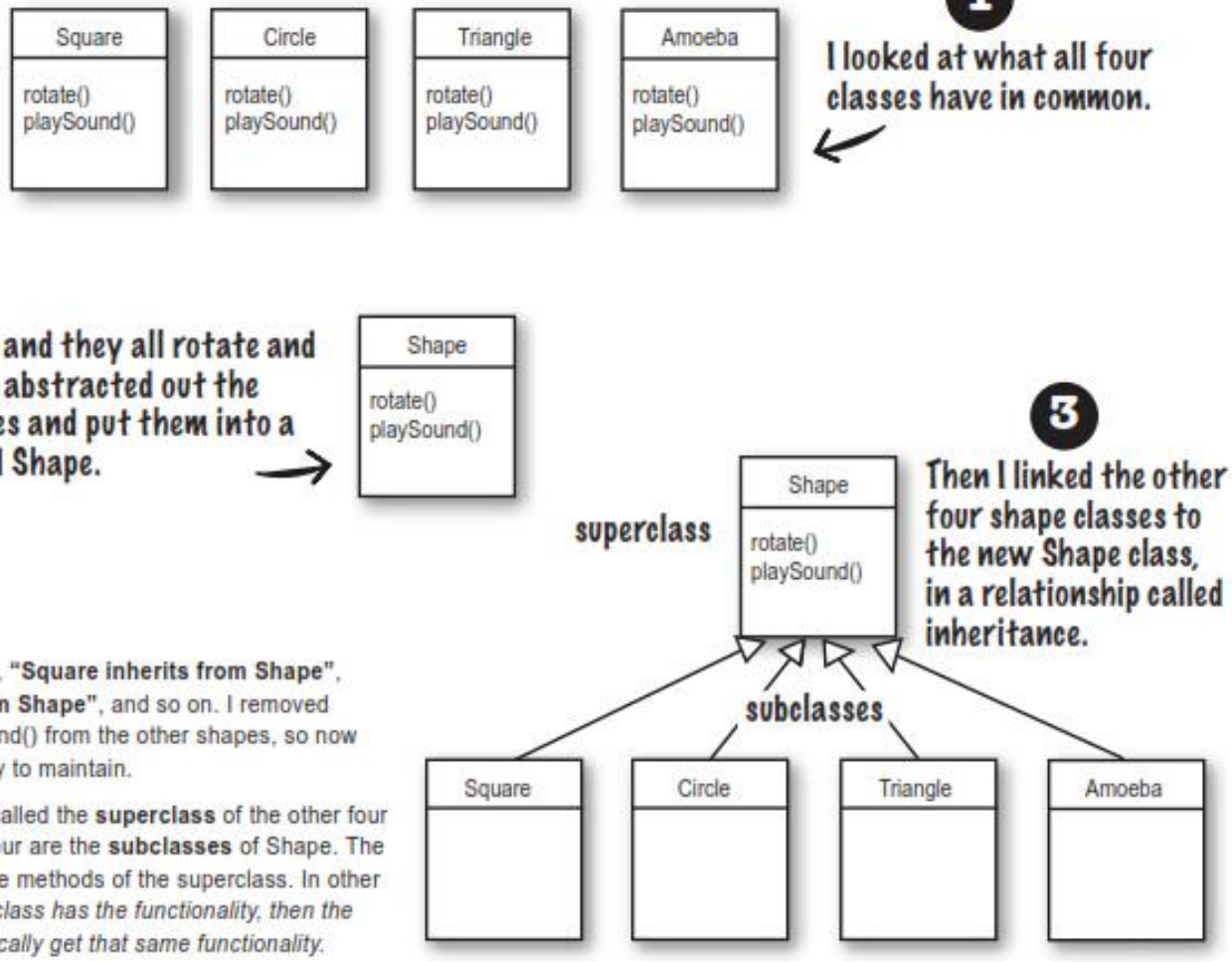
Continue...


调用过程演示（9）
堆内存
22
1
2000
7
7
1970
栈内存
change3
b
354752
d2
354752
d1
587934
main
date
9
test
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

Continue...


调用过程演示（10）
堆内存
22
1
2000
7
7
1970
栈内存
d2
d1
date
test
main
354752
587934
9
110925
Test test = new Test();
int date = 9;
BirthDate d1=
new BirthDate(7,7,1970);
BirthDate d2=
new BirthDate(1,1,2000);
test.change1(date);
test.change2(d1);
test.change3(d2);
public void change1(int i)
{i = 1234;}
public void change2(BirthDate b)
{b = new BirthDate(22,2,2004);}
public void change3(BirthDate b)
{b.setDay(22);}

Inheritance

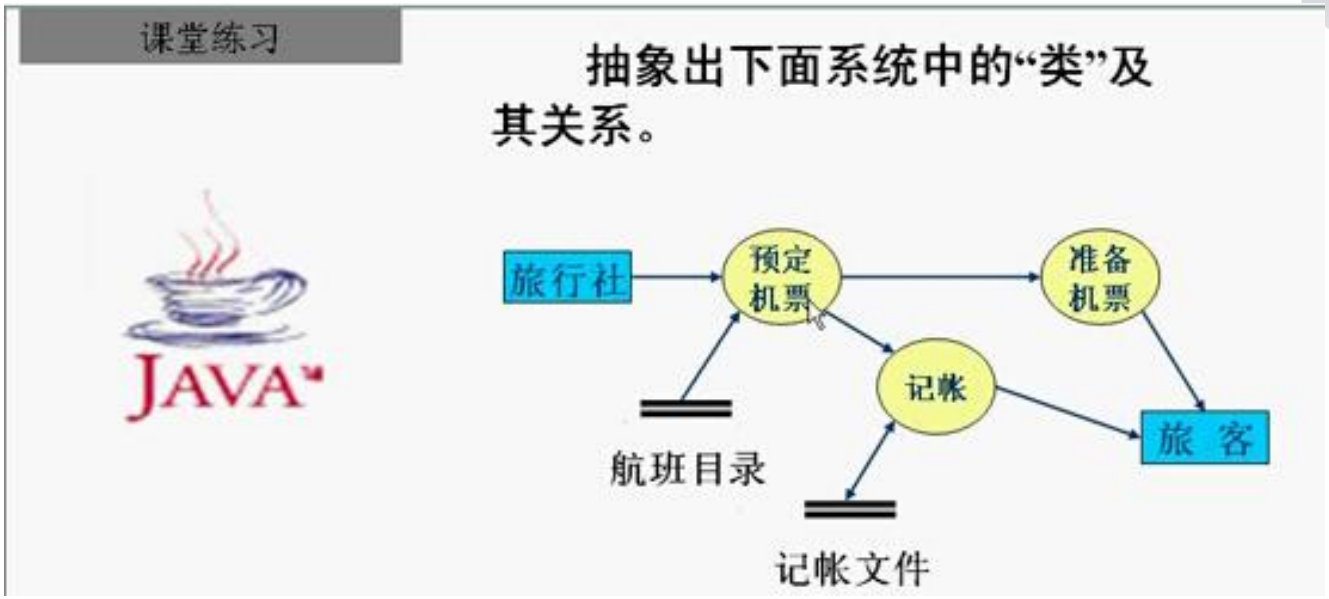


You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

Classroom Exercise

Find Classes and methods



Homework

课 堂 练 习

课堂练习

定义一个“点”（Point）类用来表示三维空间中的点（有三个坐标）。要求如下：

1. 可以生成具有特定坐标的点对象。
2. 提供可以设置三个坐标的方法。
3. 提供可以计算该“点”距原点距离平方的方法。
4. 编写程序验证上述三条。

TestPoint.java