Chapter 7

Constructing Objects Part 2

Initializing Instance Variables

```
 1 class Initialize1
 2 {
 3    private int x, y;
 4    //-------------------------------------
 5    public Initialize1()
 6    {
 7       x = 0;
 8       y = 1;
 9    }
10    //-------------------------------------
11    public void displaySum()
12    {
13       System.out.println(x + y);
14    }
15 }
```

Default Values for Instance Variables

Type	Default value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0f`
`double`	`0.0`
`char`	space
`boolean`	`false`
any class	`null`

Explicit initialization better than using default

1) Not everyone knows that instance variables not explicitly initialized get default values. Explicit initialization makes clear what the initial values will be.

2) If Java were changed so that instance variables were given different default values or not given default values at all, classes that relied on the current default values would no longer compile to the correct bytecode.

Local variables do not have default value

```
public void f()
{
    int z;
    System.out.println(z); // error
}
```

Initializing in declaration

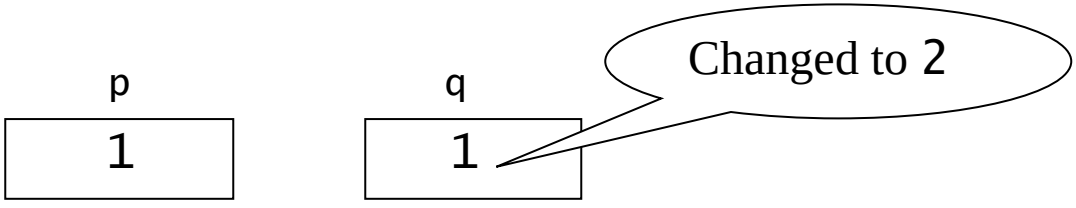
```
1 class Initialize2
2 {
3    private int x, y = 1;
4    //--------------------------------
5    public void displaySum()
6    {
7       System.out.println(x + y);
8    }
9 }
```

More flexible to initialize in constructor

```
 1 class Initialize3
 2 {
 3    private int x, y;
 9    //-----------------------------------
 4    public Initialize3(int xx, int yy)
 5    {
 6       x = xx;
 7       y = yy;
 8    }
 9    //-----------------------------------
 9    public void displaySum()
10    {
11       System.out.println(x + y);
12    }
13 }
```

Passing Primitive Types Vs Passing References

```
int p = 1;                          // p is 1 here
change(p);
System.out.println(p);              // p is still 1
```



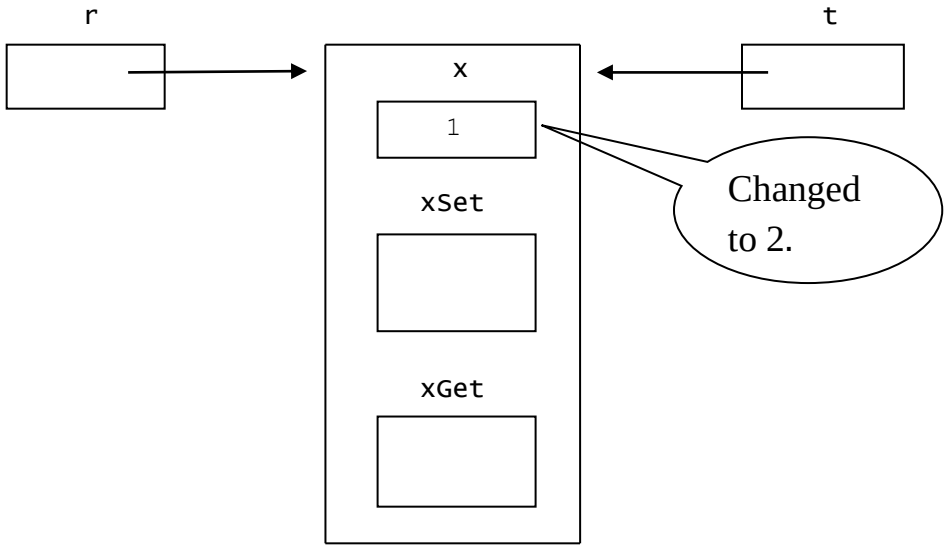

```
public static void change(int q)
{
   q =  2;                          // does not affect p
}
```

```
 1 class SideEffect
 2 {
 3    private int x = 1;
 4    //-------------------------------------
 5    public void xSet(int xx)
 6    {
 7       x = xx;
 8    }
 9    //-------------------------------------
10    public int xGet()
12    {
13       return x;
14    }
15 }
```

r and t point to the same object

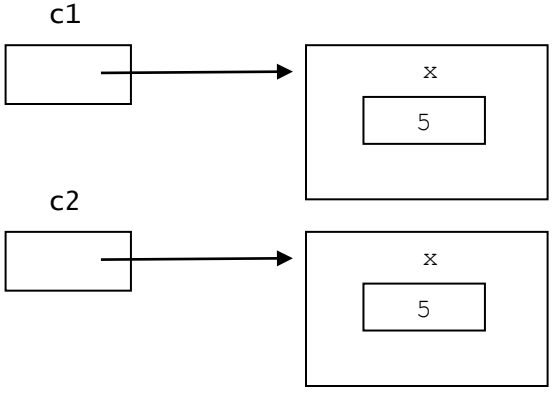
```
SideEffect r = new SideEffect(); // x is 1 here
change(r);
System.out.println(r.xGet());    // x is 2 here


public static void change(SideEffect t)
{
   t.xSet(2);                    // assigns 2 to x
}
```

r is not changed. What r points to is changed.


r
t
x
1
xSet
xGet
Changed
to 2.

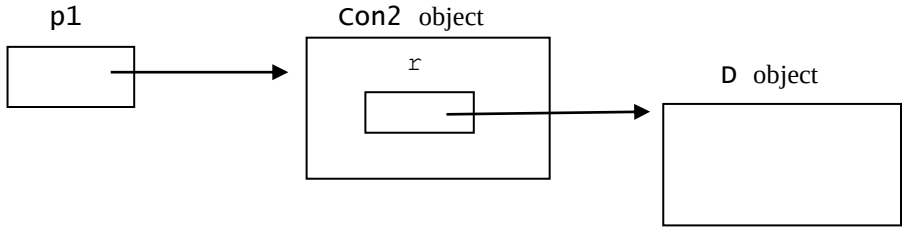
Copy Constructor

```
 1 class Con1
 2 {
 3    private int x;
 4    //---------------------------------
 5    public Con1(int xx)         // first constructor
 6    {
 7       x = xx;
 8    }
 9    //---------------------------------
10    public Con1(Con1 original) // copy constructor
11    {
12       x = original.x;
13    }
14 }
15 //================================================
16 class TestCon1
17 {
18    public static void main(String[] args)
19    {
20       Con1 c1 = new Con1(5); // calls first constructor
21       Con1 c2 = new Con1(c1);// calls second constructor
22    }
23
```

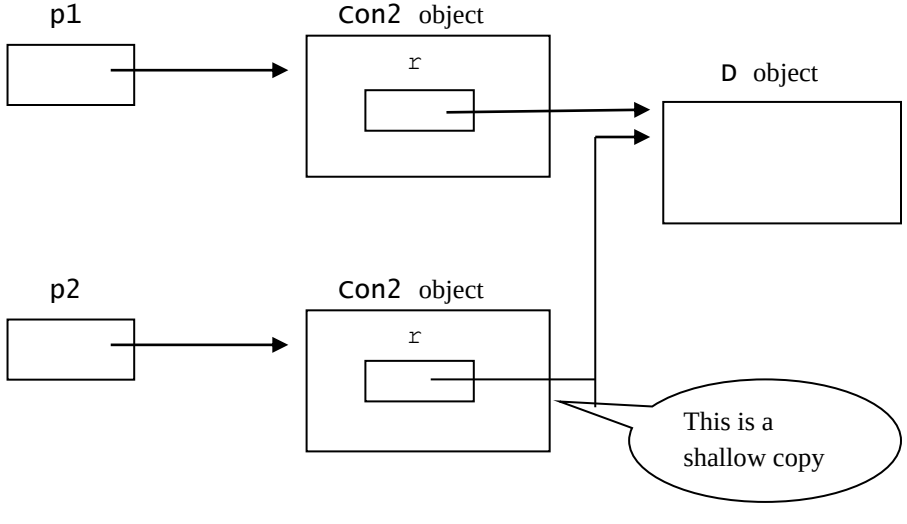

c1
x
5
c2
x
5

```
 1 class Con2
 2 {
 3    private D r;
 4    //-----------------------------------
 5    public Con2()
 6    {
 7       r = new D();
 8    }
 9    //-----------------------------------
10    public Con2(Con2 original)
11    {
12       r = original.r;    // bad
13    }
14 }
```

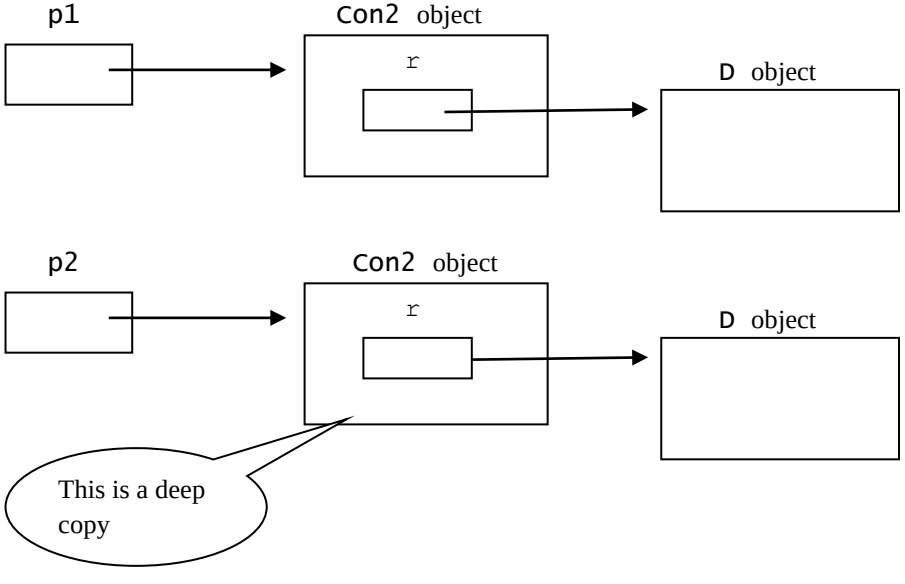
a)


p1
Con2 object
r
D object

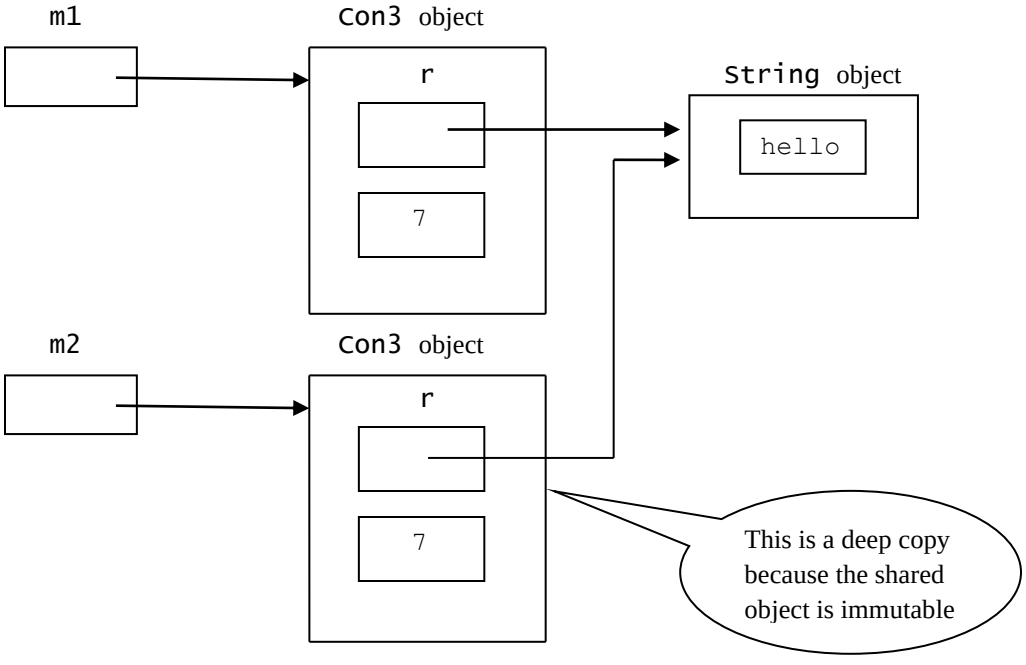
b)


p1
Con2 object
r
D object
p2
Con2 object
r
This is a
shallow copy

c)


p1
Con2 object
r
D object
p2
Con2 object
r
D object
This is a deep copy

```
 1 class Con3
 2 {
 3    private String r;
 4    private int x;
 5    //-----------------------------------
 6    public Con3(String rr, int xx)
 7    {
 8       r = rr;
 9       x = xx;
10    }
11    //-----------------------------------
12    public Con3(Con3 original)       // copy constructor
13    {
14       r = original.r; // simple copy ok
15       x = original.x; // simple copy ok for primitives
16    }
17 }
18 //================================================
19 class TestCon3
20 {
21    public static void main(String[] args)
22    {
23       Con3 m1 = new Con3("hello", 7);
24       Con3 m2 = new Con3(m1);
25    }
26 }
```


m1
Con3 object
r
String object
hello
7
m2
Con3 object
r
7
This is a deep copy
because the shared
object is immutable

Fixing Privacy Leaks Using Copy Constructors

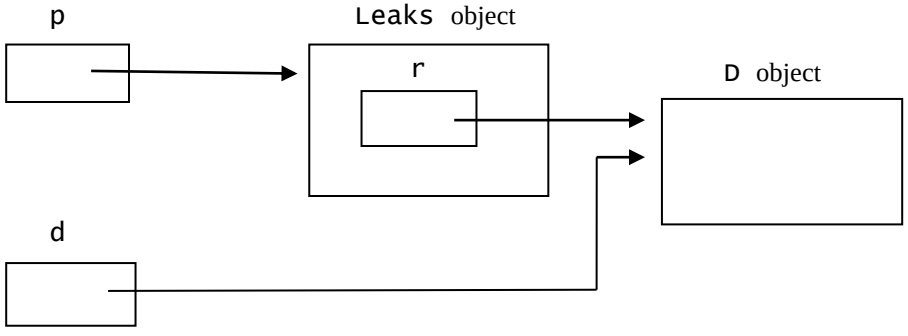
```
 1 class Leaks
 2 {
 3    private D r;
 4    //-----------------------------------
 5    public Leaks(D rr)
 6    {
 7       r = new D();
 8    }
 9    //-----------------------------------
10    pubic Leaks(Leaks original)
11    {
12       r = original.r;    // bad: creates shallow copy
13    }
14    //-----------------------------------
15    public D rGet()
16    {
17       return r;  // bad: gives direct access to r object
18    }
19    //-----------------------------------
20    public void rSet(D rr)
21    {
22       r = rr;   // bad: gives direct access to r object
23    }
24 }
```

```
D d = p.rGet();
```

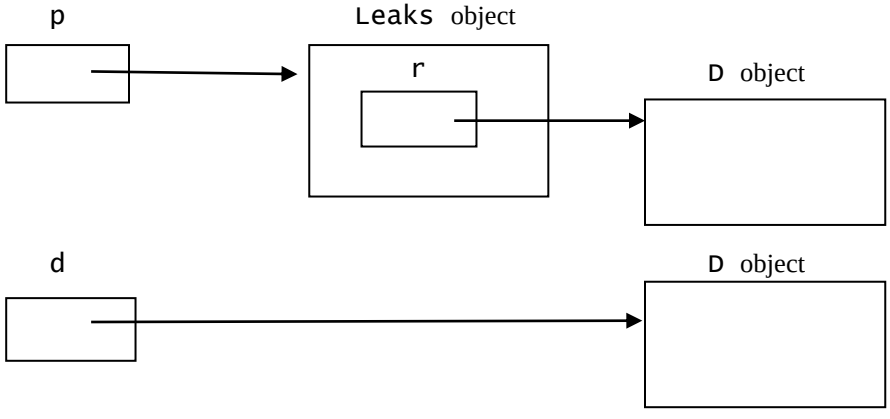
or

```
p.rSet(d);
```

Bad:



Good:



```
 1 class NoLeaks
 2 {
 3    private D r;
 4    //-----------------------------------
 5    public NoLeaks(D rr)
 6    {
 7       r = new D();
 8    }
 9    //-----------------------------------
10    public NoLeaks(NoLeaks original)
11    {
12       r = new D(original.r);          // now ok
13    }
14    //-----------------------------------
15    public D rGet()
16    {
17       return new D(r);                // now ok
18    }
19    //-----------------------------------
20    public void rSet(D rr)
21    {
22       r = new D(rr);                  // now ok
23    }
24 }
```