Objects and Classes

Part 4 – More on Methods & Encapsulation

Chapter 4, Core Java, Volume I

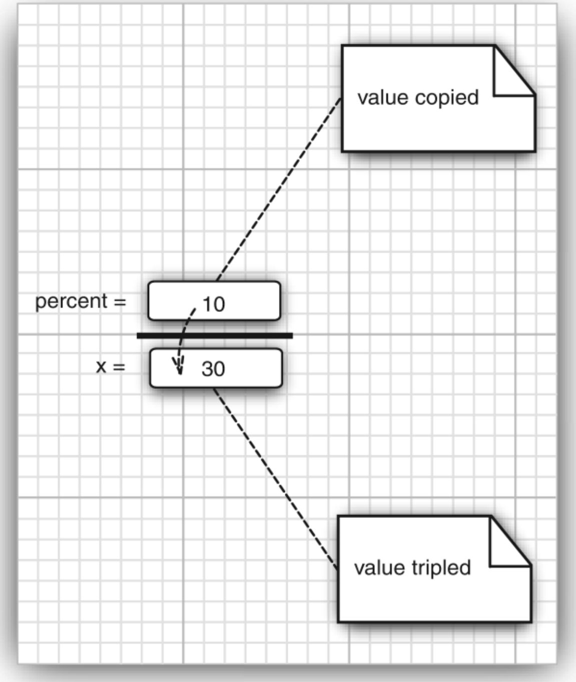
Contents

- Parameter Passing
- Method Overloading
- Access Modifiers: public and private
- Accessing and Modifying Private Data
- More on Encapsulation

Parameter Passing (review)

- Call by value: The method gets copies of the argument values.
- A method cannot change the contents of variables passed to it.
- Example:

```
public static void tripleValue(double x) // doesn't work
{
   x = 3 * x;
}
```



- In the following call, the percent variable is not changed:

```
double percent = 10;
tripleValue(percent);
```

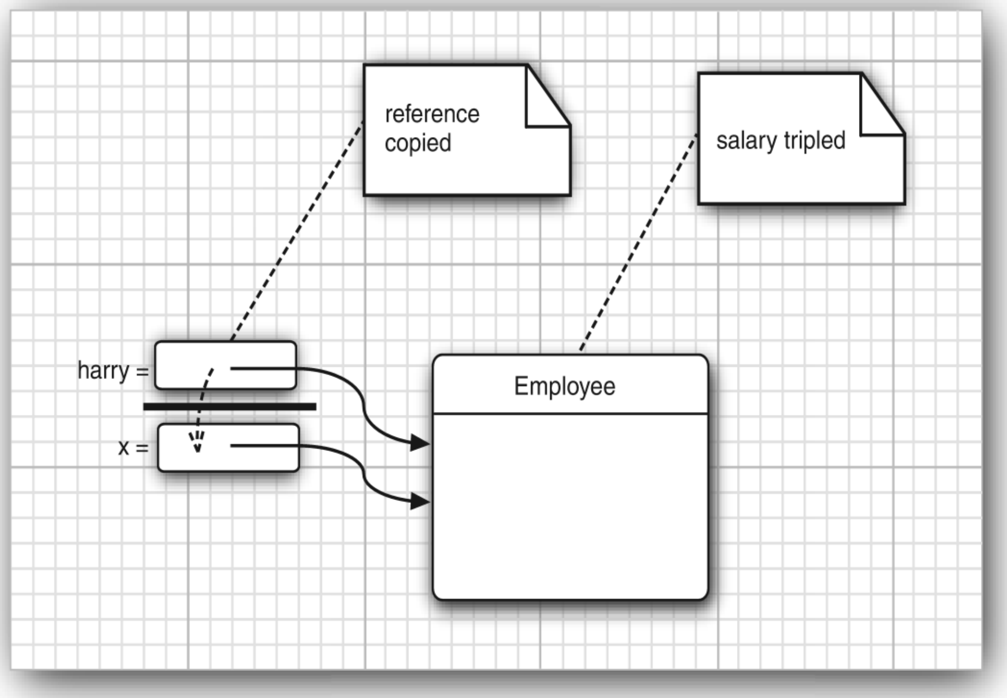
Parameter Passing

- Call with Object References : a method can mutate objects:

```
public static void tripleSalary( Employee x ) // works
{
   double s = x.getSalary();
   x.raiseSalary( s * 3.0 );
}
```

- In the following call, the salary is changed:

```
harry = new Employee(. . .);
tripleSalary( harry );
```



Parameter Passing

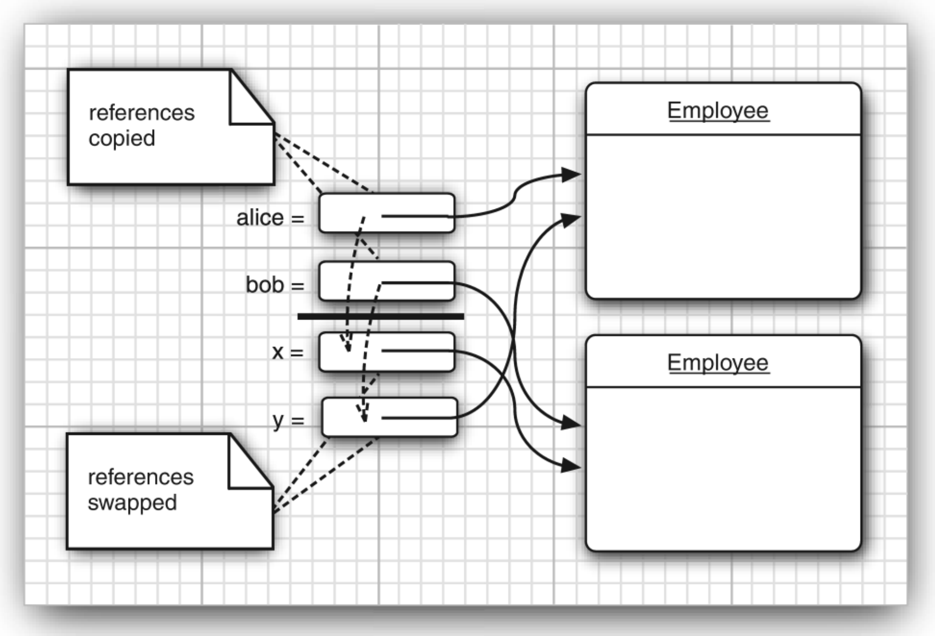
- Some people say: "Java uses "call by value" in passing numbers and "call by reference" in passing objects.
- That's nonsense. In Java, everything is passed by value.
- Object reference variables are passed by value.

- If variables for objects were passed by reference, you could swap them:

```
public static void swap(Employee x, Employee y) // does
{
   Employee temp = x;
   x = y;
   y = temp;
}
```

- But in the following call, a and b are not swapped:

```
Employee alice = new Employee("Alice", . . .);
Employee bob = new Employee("Bob", . . .);
swap(alice, bob);
```



Parameter Passing Test : 3 Cases

```
public class ParamTest
{
  public static void main(String[] args)
  {
     // Test 1: Methods can't modify numeric parameters
    System.out.println("Testing tripleValue:");
    double percent = 10;
    System.out.println("Before: percent=" + percent);
    tripleValue(percent);
    System.out.println("After: percent=" + percent);

     // Test 2: Methods can change the state of object parameters
    System.out.println("\nTesting tripleSalary:");
    Employee harry = new Employee("Harry", 50000);
    System.out.println("Before: salary=" + harry.getSalary());
    tripleSalary(harry);
    System.out.println("After: salary=" + harry.getSalary());

      // Test 3: Methods can't attach new objects to object parameters
     System.out.println("\nTesting swap:");
     Employee a = new Employee("Alice", 70000);
     Employee b = new Employee("Bob", 60000);
```

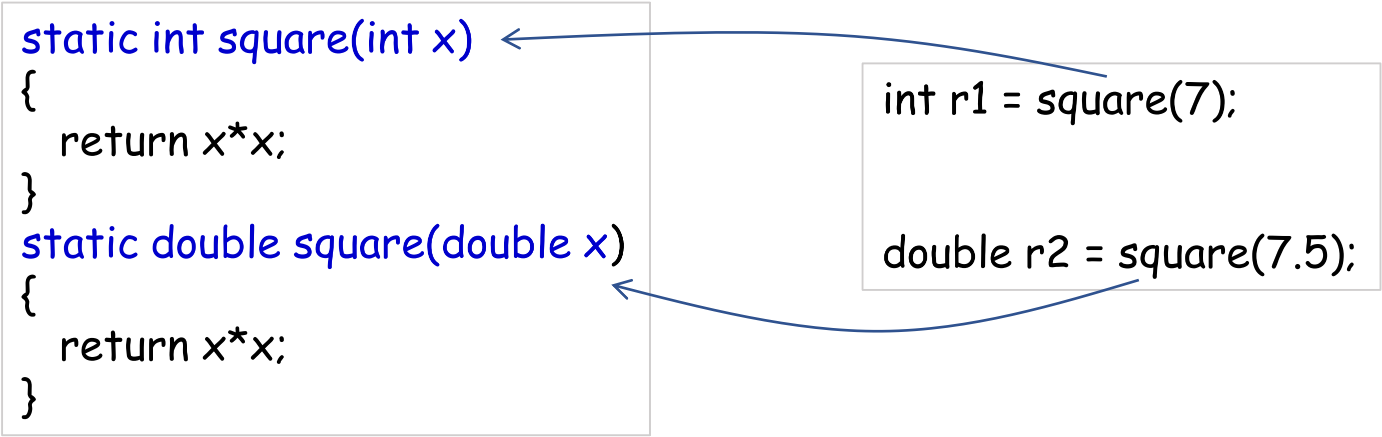
```
     System.out.println("Before: a=" + a.getName());
    System.out.println("Before: b=" + b.getName());
    swap(a, b);
    System.out.println("After: a=" + a.getName());
    System.out.println("After: b=" + b.getName());
  }
  public static void tripleValue(double x) // doesn't work
  {  x = 3 * x;
    System.out.println("End of method: x=" + x);
  }
  public static void tripleSalary(Employee x) // works
  {   x.raiseSalary(x.getSalary()*3.0);
    System.out.println("End of method: salary=" + x.getSalary());
  }
  public static void swap(Employee x, Employee y)
  {   Employee temp = x;
    x = y;
    y = temp;
    System.out.println("End of method: x=" + x.getName());
    System.out.println("End of method: y=" + y.getName());
  }
}
```

Method Overloading

- Method overloading
  - Methods with the same name declared in the same class
  - Must have different sets of parameters
- Example: String class has four pubic methods called indexOf:

```
indexOf(int)  // int parameter receives a character
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

- The compiler distinguishes overloaded methods by their signatures—
  - the *number*, *types* and *order* of its parameters.



```
static int square(int x)
{
   return x*x;
}
static double square(double x)
{
   return x*x;
}
```

```
int r1 = square(7);


double r2 = square(7.5);
```

Method Overloading

- Overloaded methods cannot be distinguished by *return type.*

```
static int square(int x)
{
   return x*x;
}
static double square(int x)
{
   return (double) x*x;
}
```

not method overding!
but, multiple definition error!
Think why?

- Method overloading can be used in defining static methods, non-static methods and constructors

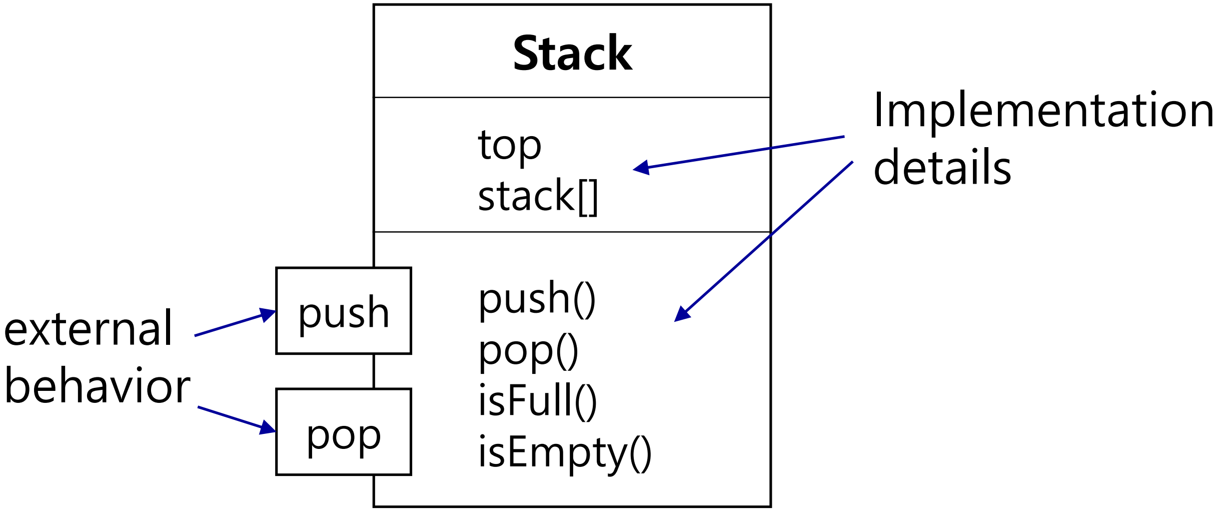
```
public Employee(String n, double s, int year, int month, int day)
{
    name = n;
    salary = s;
    hireDay = LocalDate.of(year, month, day);
}
```

```
public Employee(String n, int year, int month,  int day)
{
    name = n;
    salary = 0.0;
    hireDay = LocalDate.of(year, month, day);
}
```

Access Modifiers : Public and Private

- 4 Types of access modifiers can be specified on methods and variables
  - Public : can be accessed anywhere
  - Private : can only be accessed inside its own class
  - Package : can be accessed inside the package inn which the class is included (ref. package)
  - Protected : can be accessed by its package and subclasses (ref. inheritance)

- General Guidelines : Data Abstraction and Encapsulation
  - Hides implementation details
    - Instance fields
    - Auxiliary methods (helper methods)
  - Exposes external behavior
    - Methods



Accessing and Modifying Private Data

- Using getXXX()/setXXX() Methods
  - Getting and setting private instance variables
    .e.g getSalary(), setSalary()
- Benefit 1: The internal representation can evolve:

```
private String name;
public string getName() { return name; }
```

⇩

```
private String firstName;
private String lastName;

public String getName() { return firstName + " " + lastName; }
public String getFirstName() { return firstName; }
```

Cilents are not affected!

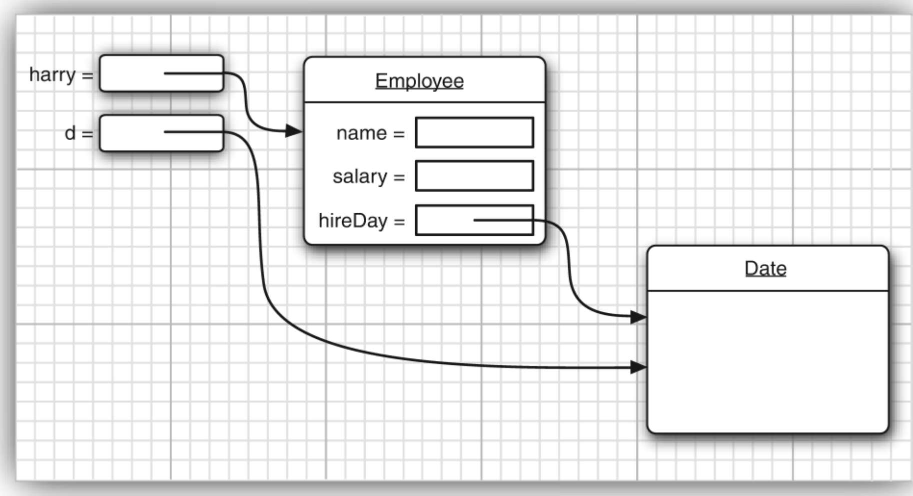
- Benefit 2: The field can be easily validated
  - e.g. setSalary method might check that the salary is never less than 0

More on Encapsulation : Returning Mutable Objects

- Be careful not to write accessor methods that return a reference to mutable objects

```
class Employee
{
   private Date hireDay;
   ...
   public Date getHireDay()
   {
       return hireDay;
   }
   ...
}
```

```
Employee harry = ...;
Date d = harry.getHireday();
doule tenYearsInMillisSeconds = 10*365.25*24*60*1000;
d.setTime(d.getTime() – (long)tenYearsInMilliSeconds);
```



Violate the principle of encapsulation!

- No problem in

```
public LocalDate getHireDay()
{
      return hireDay;
 }
```

  - Why?

More on Encapsulation : Class-based Access Privileges

- You know that a method can access the private data of the object on which it is invoked.

```
public getName()
{
       return name;
 }
```

- A method can access in private data of all objects of its class.

```
Class Empoyee
{
      …
      public boolean equals(Employee other)
      {
          return name.equals(other.name);  // not invoke other.getName()
      }
 }
…
if( harry.equals(boss)) …
…
```