AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

3-Classes & Objects & Methods

- Object-Oriented Programming(**OOP)**
    - classes & objects
    - Encapsulation
    - creating objects & constructor
- Methods
    - void method vs return method
    - calling method & parameters
    - method signature & overloaded method
    - method control flow
- References
    - Aliases
    - passing primitive data VS reference as parameter
    - null reference

Object-Oriented Programming (OOP)

- Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.
- An object is a collection of member variables (such as in a data structure) and functions that can operate on those member variables.
- From a terminology standpoint, a class is an object's type, and an object is a specific instance of a particular class. A class can be used to create multiple objects
- Classes and objects help us write complex software
- The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

Object-Oriented Programming（OOP)

- The following concepts are important to object-oriented programming:
    - Object :
        - fundamental element in oop
        - Each object has a **state**, defined by its **attributes**, and a set of **behaviors**, defined by its **methods**.
    - Attribute (instance variable):
        - An object’ s attributes are the values it stores internally, which may be represented as primitive data or as other objects.
    - Method:
        - a method is a group of programming statements that is given a name. When a method is invoked, its statements are executed.
    - Class:
        - An object is defined by a class . A class is the **model** or **blueprint** from which an object is created.
    - **Encapsulation**
    - **Inheritance**
    - **Polymorphism**

I’m getting a bit hungry…what could I have as a snack?

These are all instances of a snack!

Snack Graphics © Czajka, 2020

Computer Science

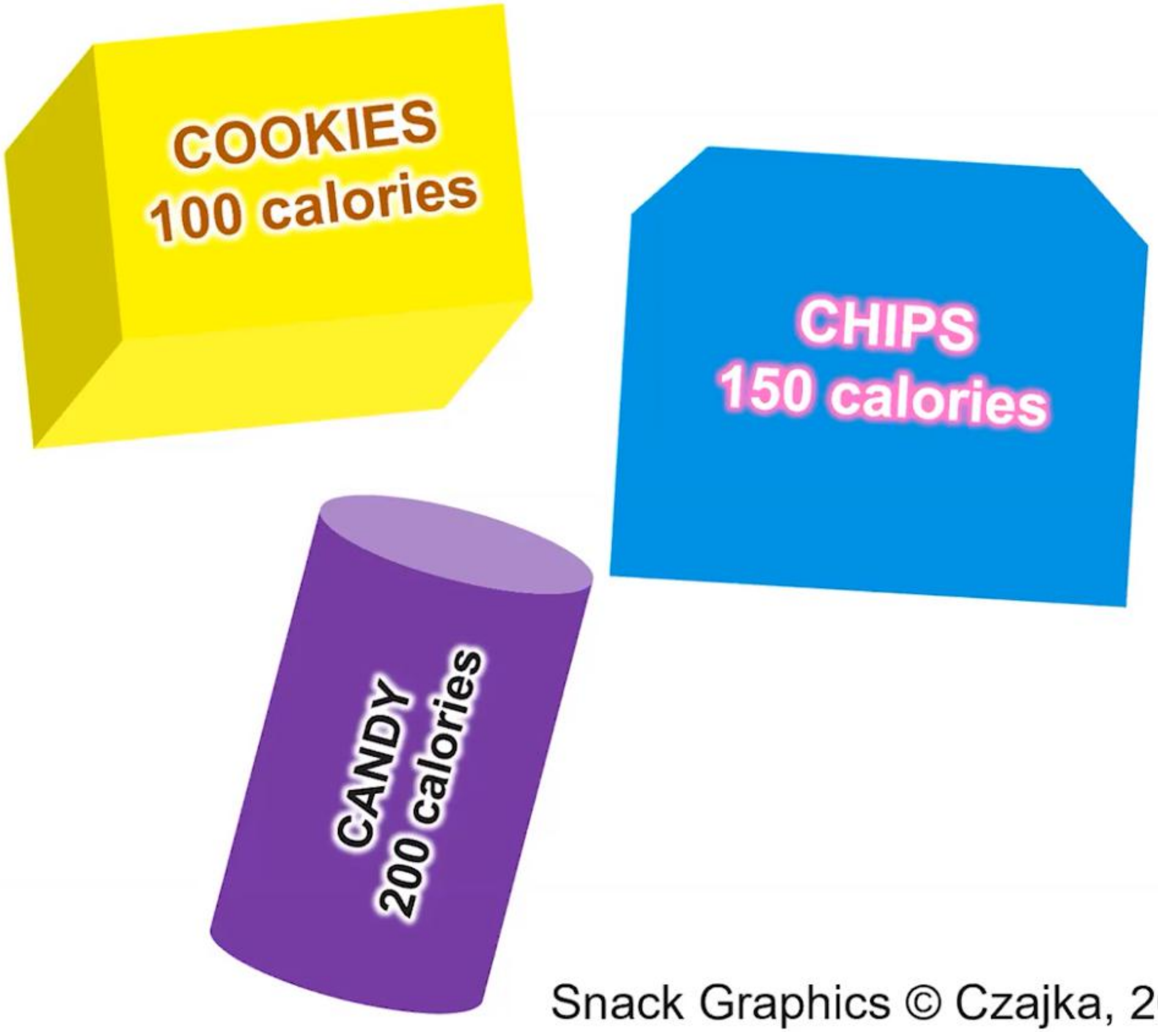
- Create models of things that exist in the real world
  - Blueprints → Class
  - Use the class → Instances of the class—Objects
  - Attributes of the objects → Instance variables
  - Behaviors of the objects → Methods

Class——Snack

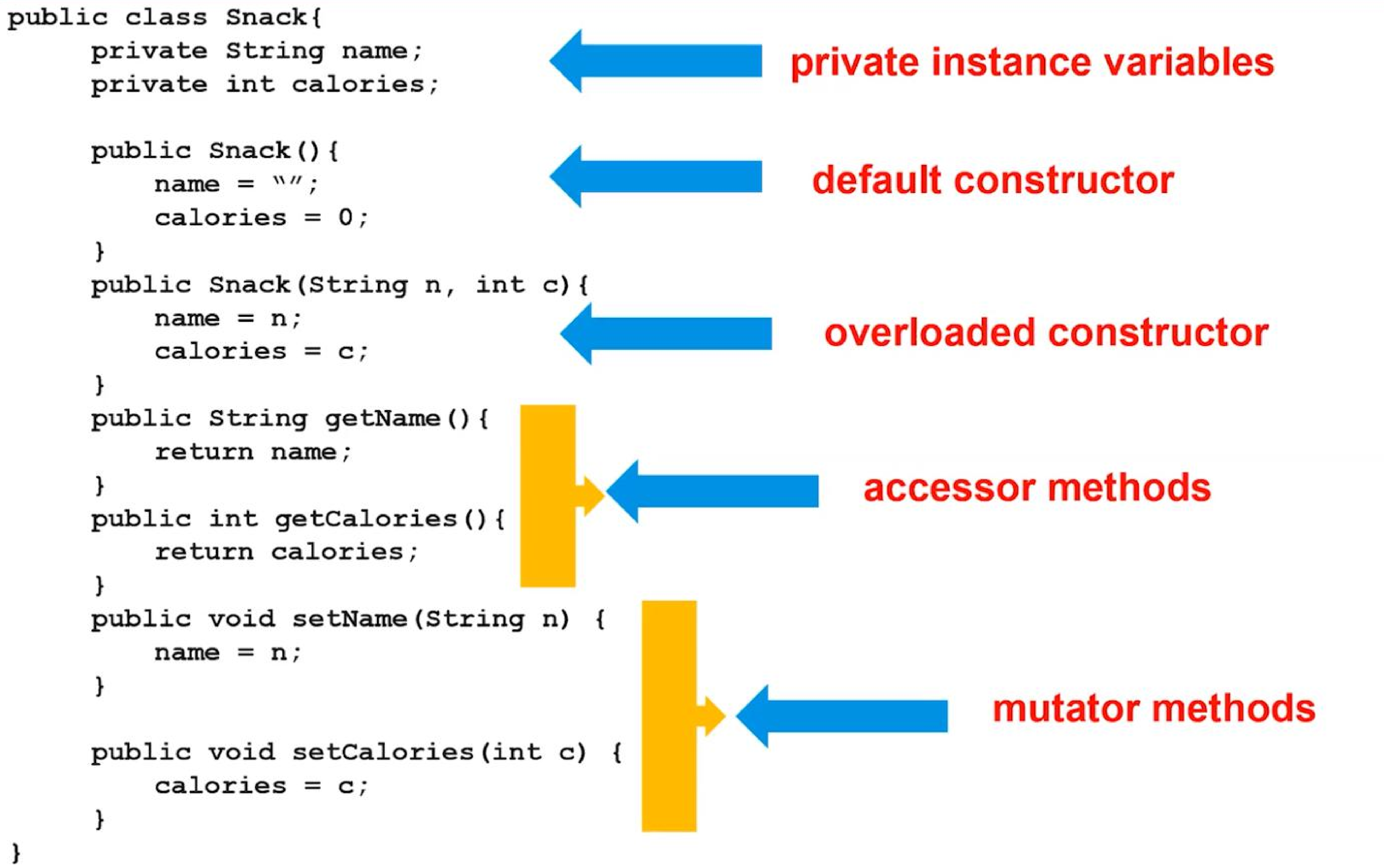
Class—Snack

- Attributes
  - name
  - calories
- Behaviors
  - get name/calories
  - set name/calories

Let’s take a look at what this class might look like…



Snack Graphics © Czajka, 2020



```
public class Snack{
     private String name;
     private int calories;

     public Snack(){
         name = “”;
         calories = 0;
     }
     public Snack(String n, int c){
         name = n;
         calories = c;
     }
     public String getName(){
         return name;
     }
     public int getCalories(){
         return calories;
     }
     public void setName(String n) {
         name = n;
     }

     public void setCalories(int c) {
         calories = c;
     }
}
```

private instance variables

default constructor

overloaded constructor

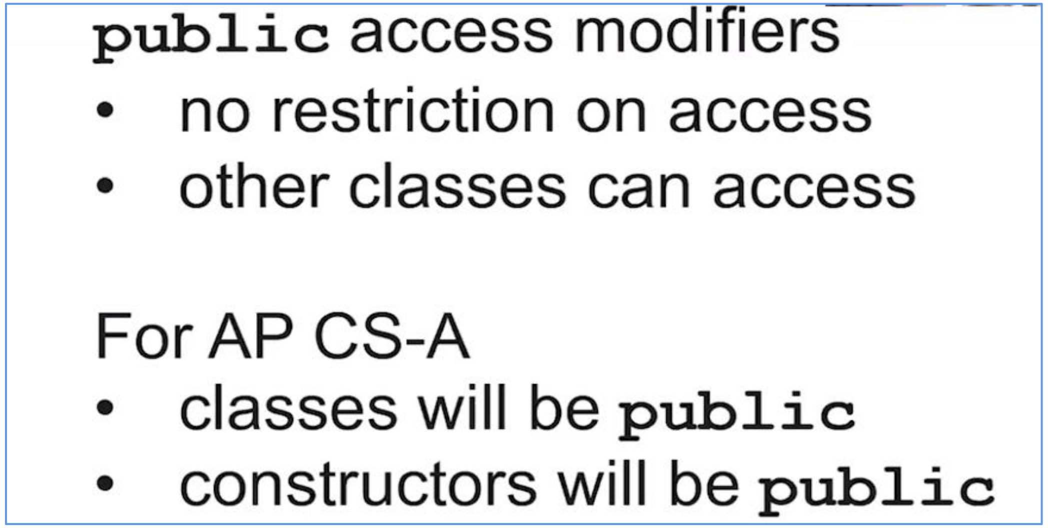
accessor methods

mutator methods

Encapsulation

- A fundamental concept of object-oriented programming
- Wrap the data (variables) and code that acts on the data (methods) in one unit (class)
- In AP CS-A, we will do this by:
  - Writing a class
  - Declaring the instance variables as **`private`**
  - Providing accessor (get) methods and modifier (set) methods to view and modify variables outside of the class

Visibility Modifiers

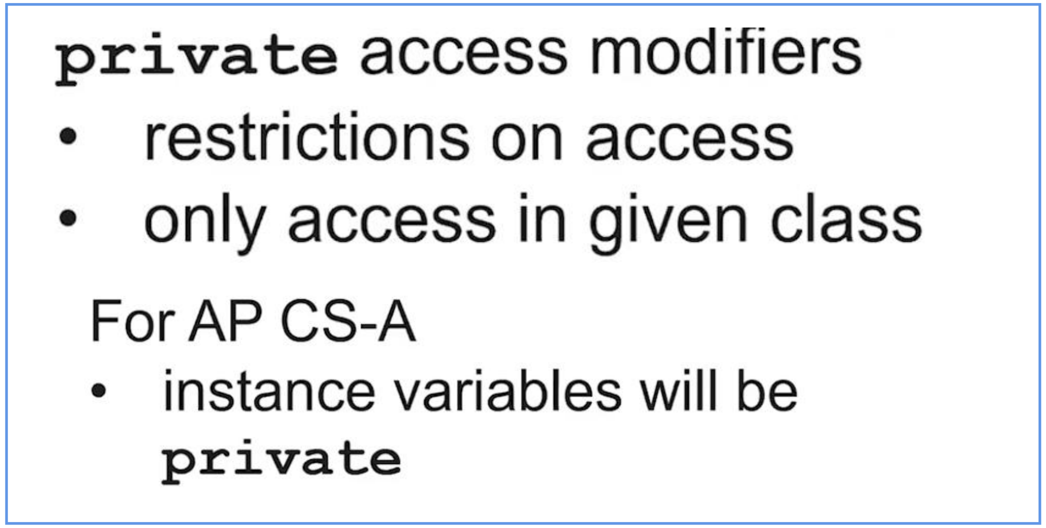


`public` access modifiers

- no restriction on access
- other classes can access

For AP CS-A

- classes will be `public`
- constructors will be `public`

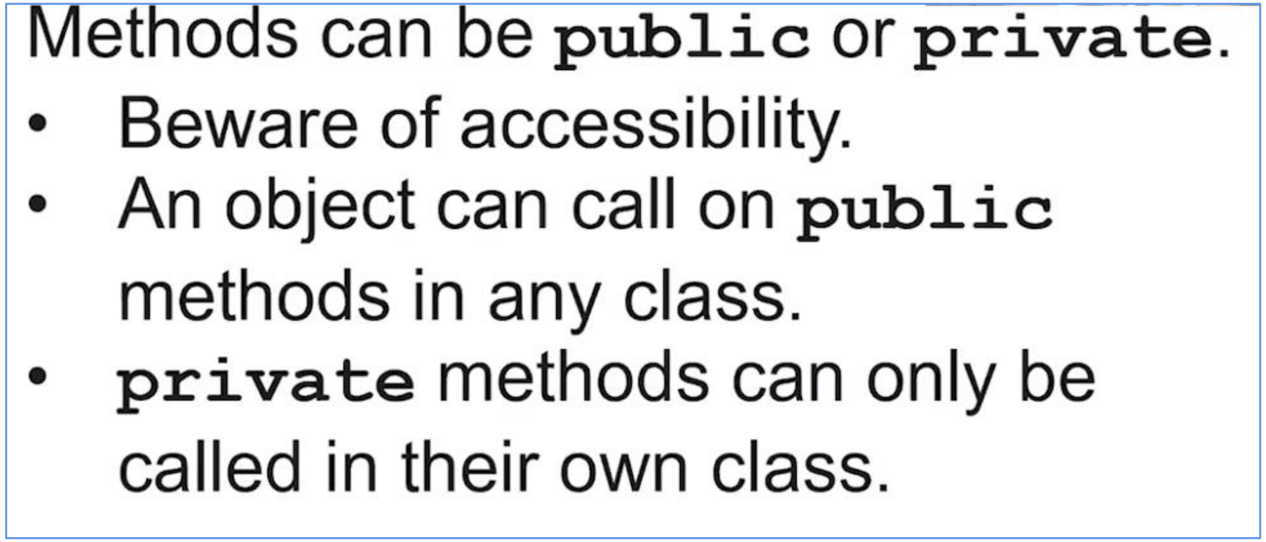


`private` access modifiers

- restrictions on access
- only access in given class

For AP CS-A

- instance variables will be `private`



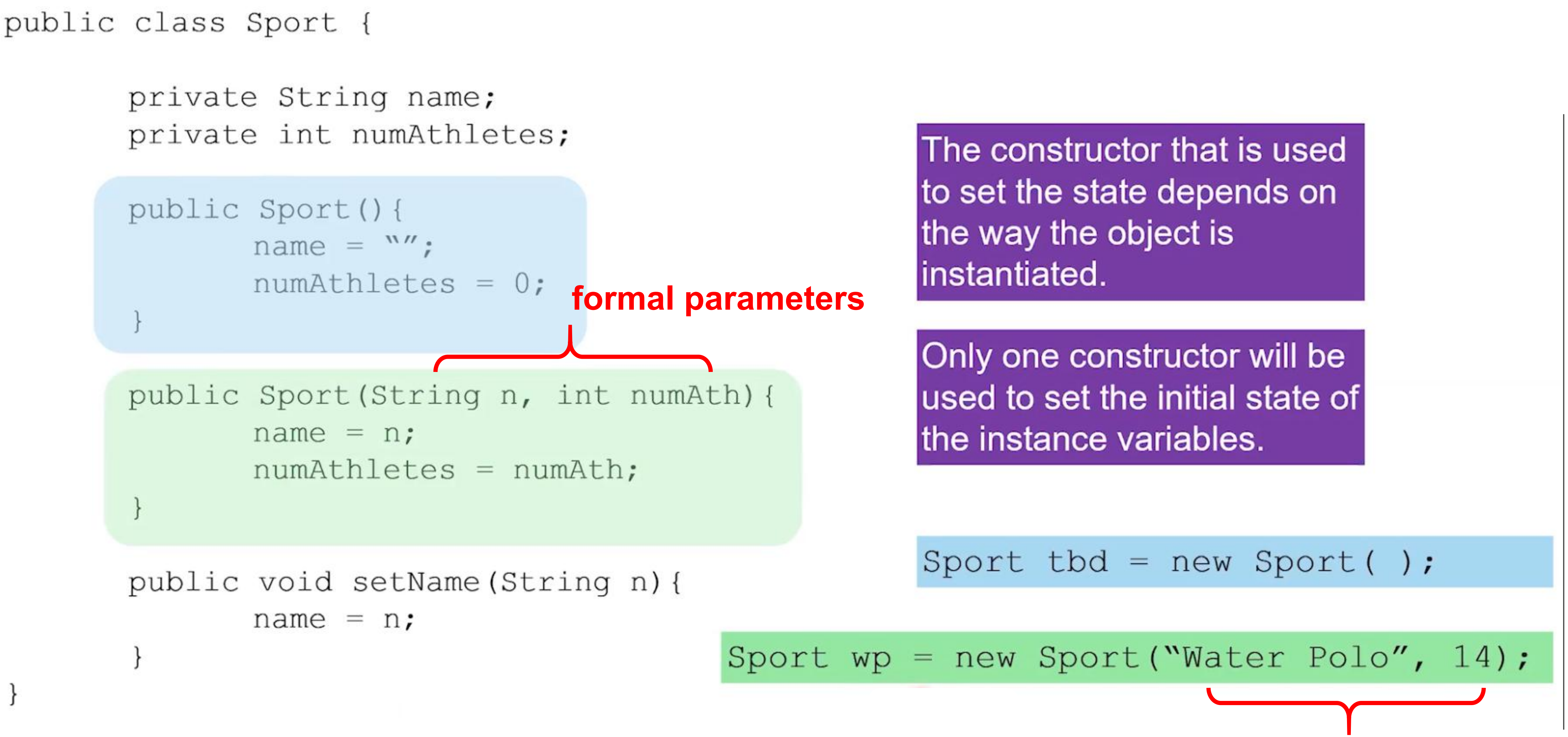
Methods can be `public` or `private`.

- Beware of accessibility.
- An object can call on `public` methods in any class.
- `private` methods can only be called in their own class.

Why make instance variables private?

- Restrict access (read-only)
- Option to provide validation checks
- For now, you need to make your instance variables private for AP CS-A

Constructor



```
public class Sport {

        private String name;
        private int numAthletes;

        public Sport(){
                name = “”;
                numAthletes = 0;
        }

        public Sport(String n, int numAth){
                name = n;
                numAthletes = numAth;
        }

        public void setName(String n){
                name = n;
        }
}
```

formal parameters

The constructor that is used to set the state depends on the way the object is instantiated.

Only one constructor will be used to set the initial state of the instance variables.

```
Sport tbd = new Sport( );
```

```
Sport wp = new Sport(“Water Polo”, 14);
```

actual parameters

Constructor

```
public class Sport {

        private String name;
        private int numAthletes;

        public String getName(){
                return name;
        }

        public int getNumAthletes(){
                return numAthletes;
        }
}
```

If no constructor is provided, Java provides a default constructor.

All instance variables are set to default values:
int – 0
double – 0.0
Strings and other objects – null

The keyword null is a special value used to indicate that a reference is not associated with any object.

Be careful! This may cause a null pointer exception when other methods are called.

Methods

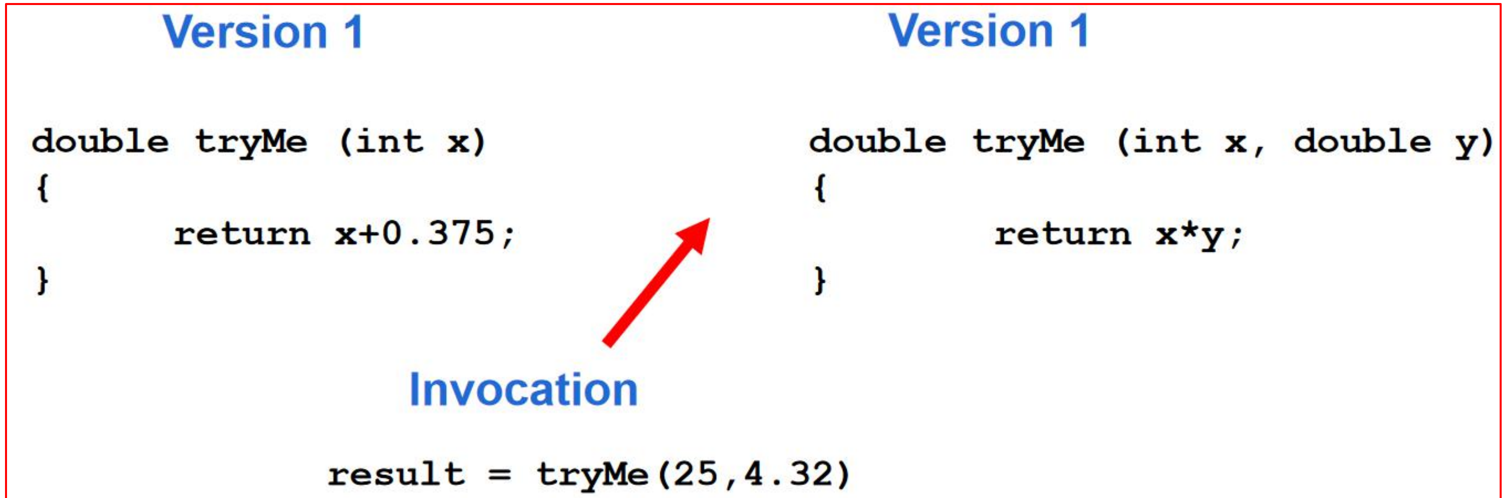
- Methods define the behaviors for all objects of a class and consist of a set of instructions for executing the behavior
- **Procedural abstraction** allows a programmer to use a method by knowing what the method does even if they do not know how the method was written. (eg,`System.out.println()`)
- The **dot operator** is used along with the object name to call non-static methods.
- **Types of methods**
  - void methods :
    - do not have return values and are therefore not called as part of an expression.
    - like a command or instruction, we don’t expect for response back
  - return methods:
    - Non-void methods return a value that is the same type as the return type in the signature. To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression.
    - like I am asking a question, and I’m waiting for something to come back

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

```
public void setName(String a){
        name=a;
}
```

Overloading Methods

- *Method overloading is the process of using the same method name for multiple methods*
- *The signature of each overloaded method must be unique*
- The signature includes the number, type, and order of the parameters
- The **compiler** determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature



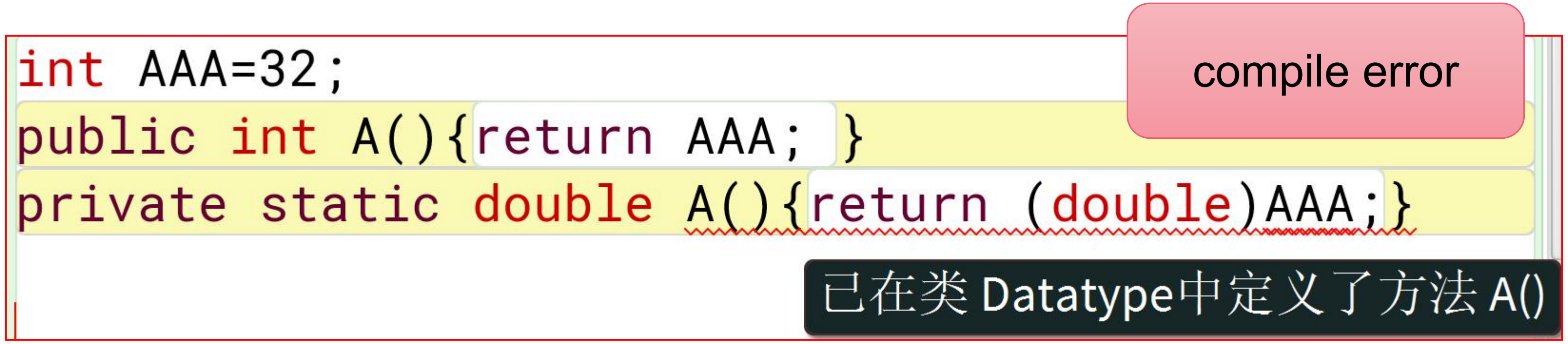
```
double tryMe (int x)
{
        return x+0.375;
}
```

```
double tryMe (int x, double y)
{
            return x*y;
}
```

```
result = tryMe(25,4.32)
```

Method signature

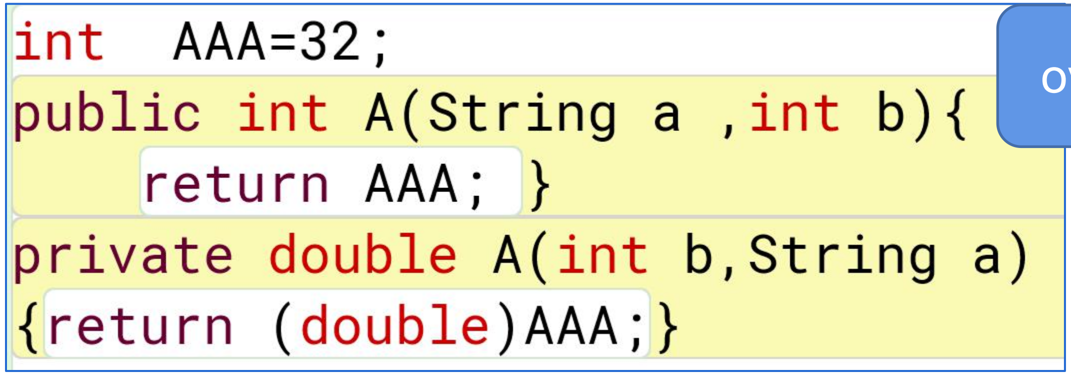
- The method signature consists of the **method name** and the **parameter list**.
- Method signature **does not include** the **return type** of the method.
- Method signature **does not include** the **Visibility Modifiers**.
- eg:



```
int AAA=32;
public int A(){return AAA; }
private static double A(){return (double)AAA;}
```

已在类 Datatype中定义了方法 A()

compile error

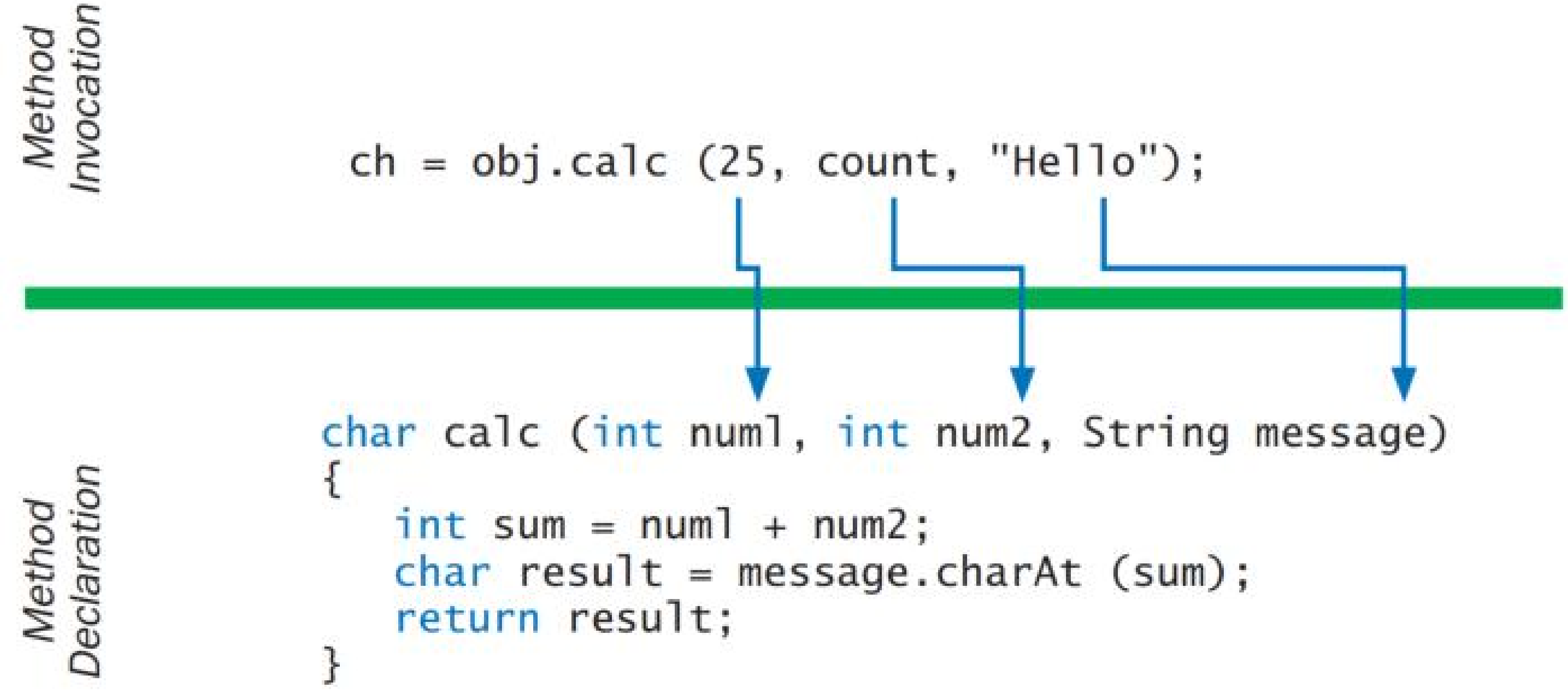


```
int  AAA=32;
public int A(String a ,int b){
     return AAA; }
private double A(int b,String a)
{return (double)AAA;}
```

overloaded method

Parameters

- Each time a method is called, the ***actual parameters*** in the invocation are copied into the ***formal parameters***



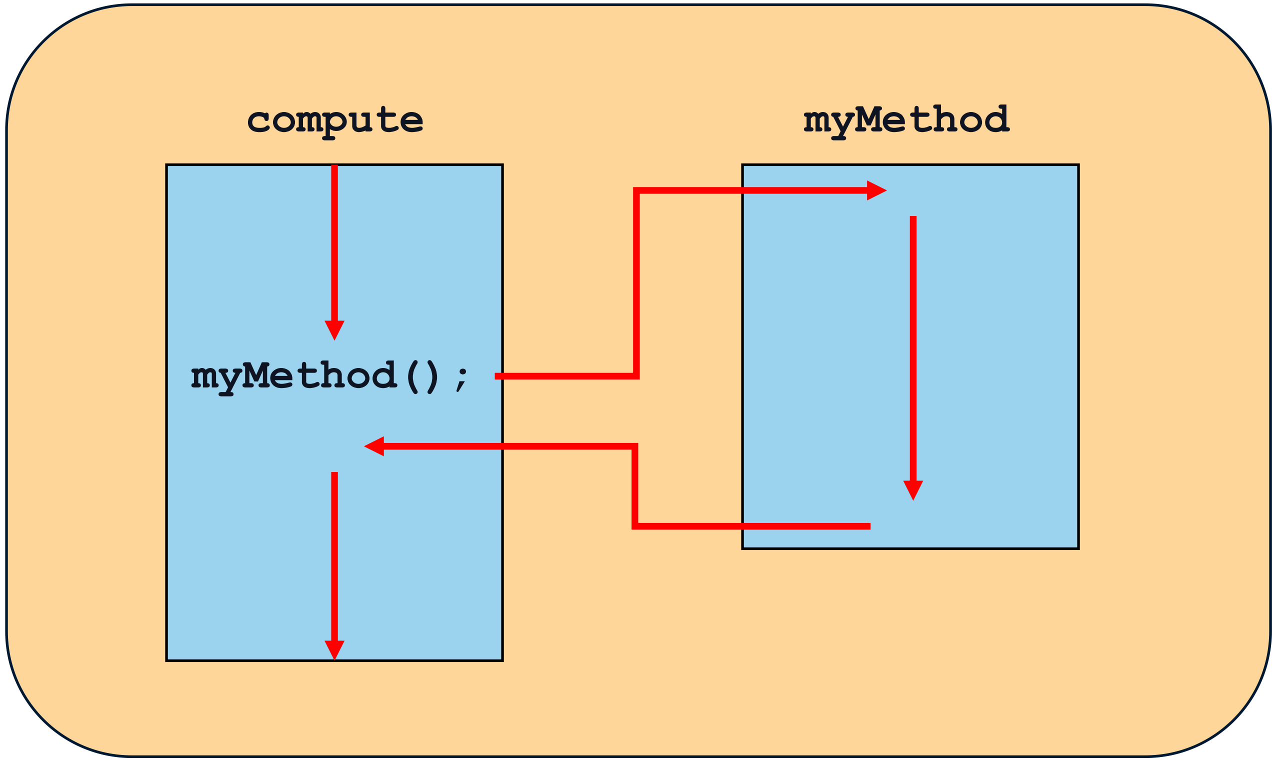
```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);
   return result;
}
```

passing parameters from the method invocation to the declaration

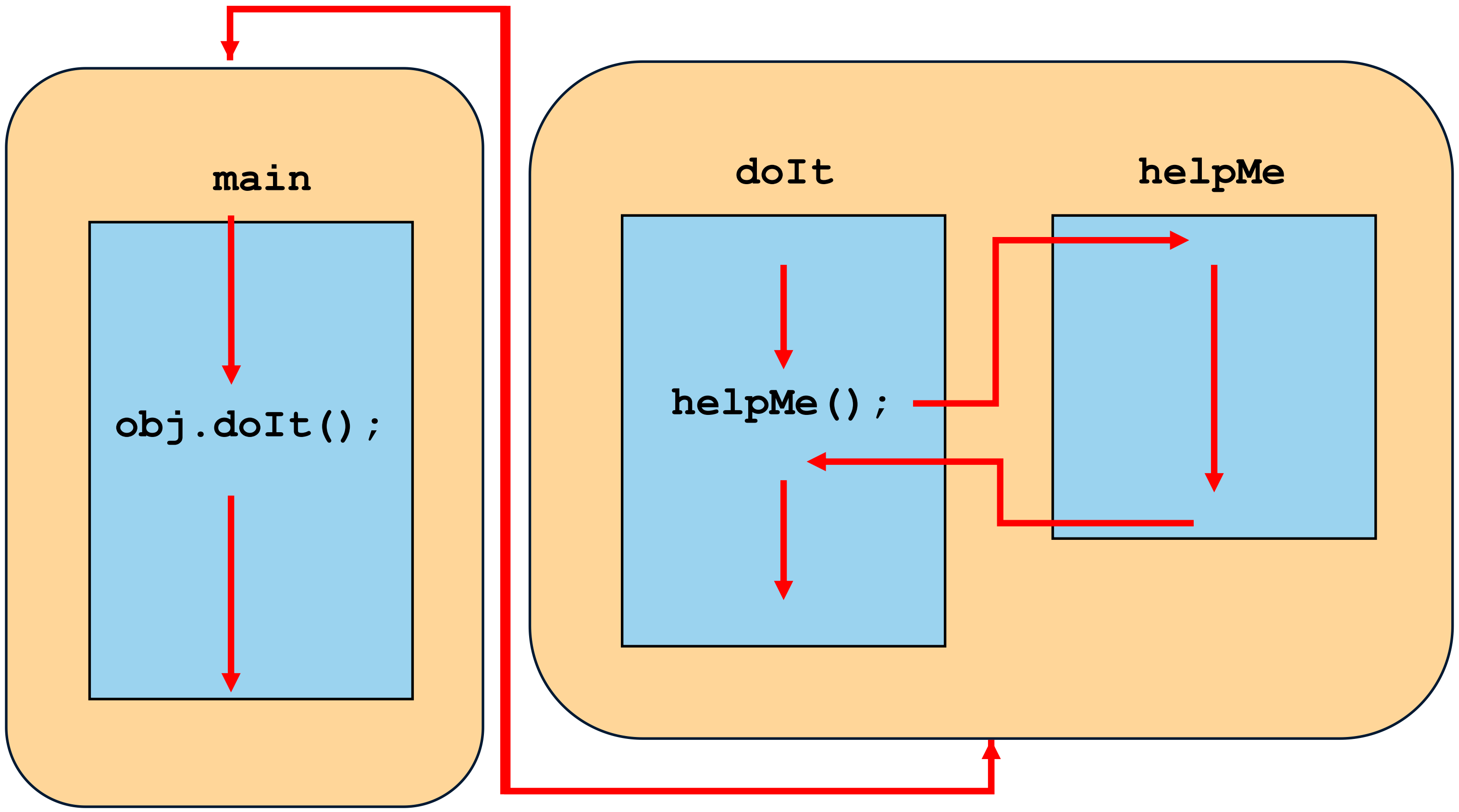
Method Control Flow

- The called method can be within the same class, in which case only the method name is needed



Method Control Flow

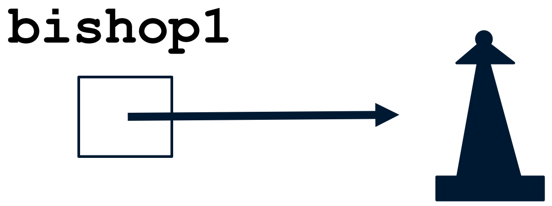
- The called method can be part of another class or object



References

- An object reference variable holds the memory address of an object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

```
ChessPiece bishop1 = new ChessPiece();
```



References

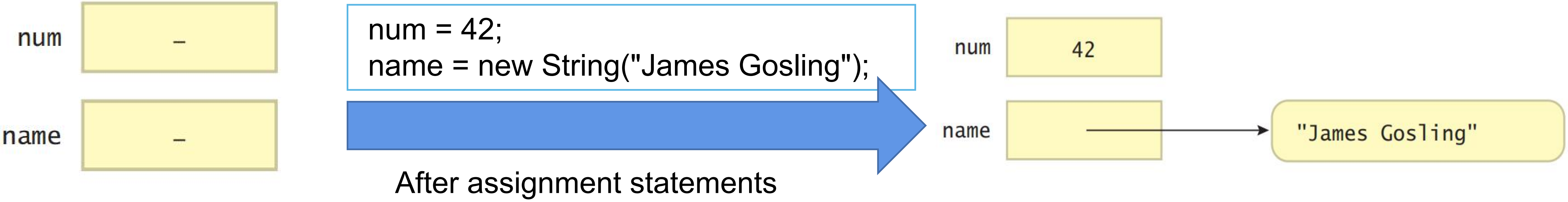
- Consider the following two declarations:

```
int num;

String name;
```

creates a variable that holds an integer value

creates a String variable that holds a reference to a String object

- An object variable doesn't hold an object itself, it holds the address of an object.
- Initially, the two variables declared above don't contain any data. We say they are uninitialized, which can be depicted as follows:



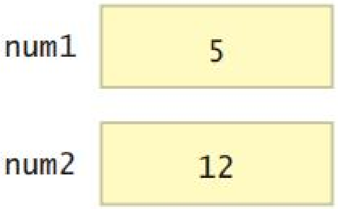
```
num = 42;
name = new String("James Gosling");
```

After assignment statements

Aliases

assignment on primitive values

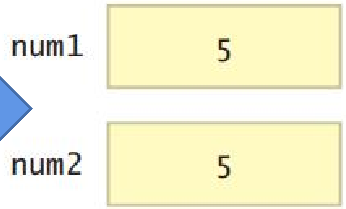
initialize
int num1=5;
int num2 =12;



num2 =num1;

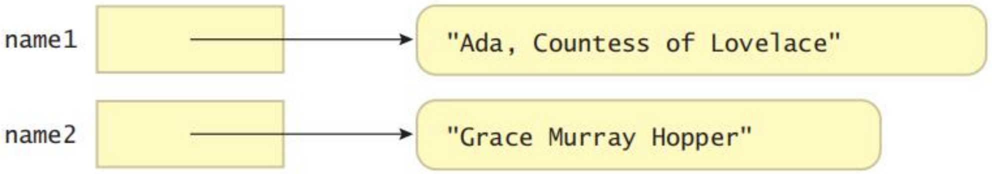
After assignment

num1 & num2 refer to different locations in memory ,both of those locations now contain the value 5



assignment on object reference

initialize
String name1 = "Ada, Countess of Lovelace";
String name2 = "Grace Murray Hopper";

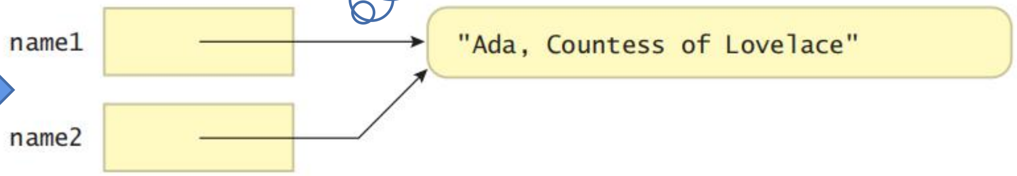


after initialized,name1 and name2 refer to two different String objects

name2 =name1;

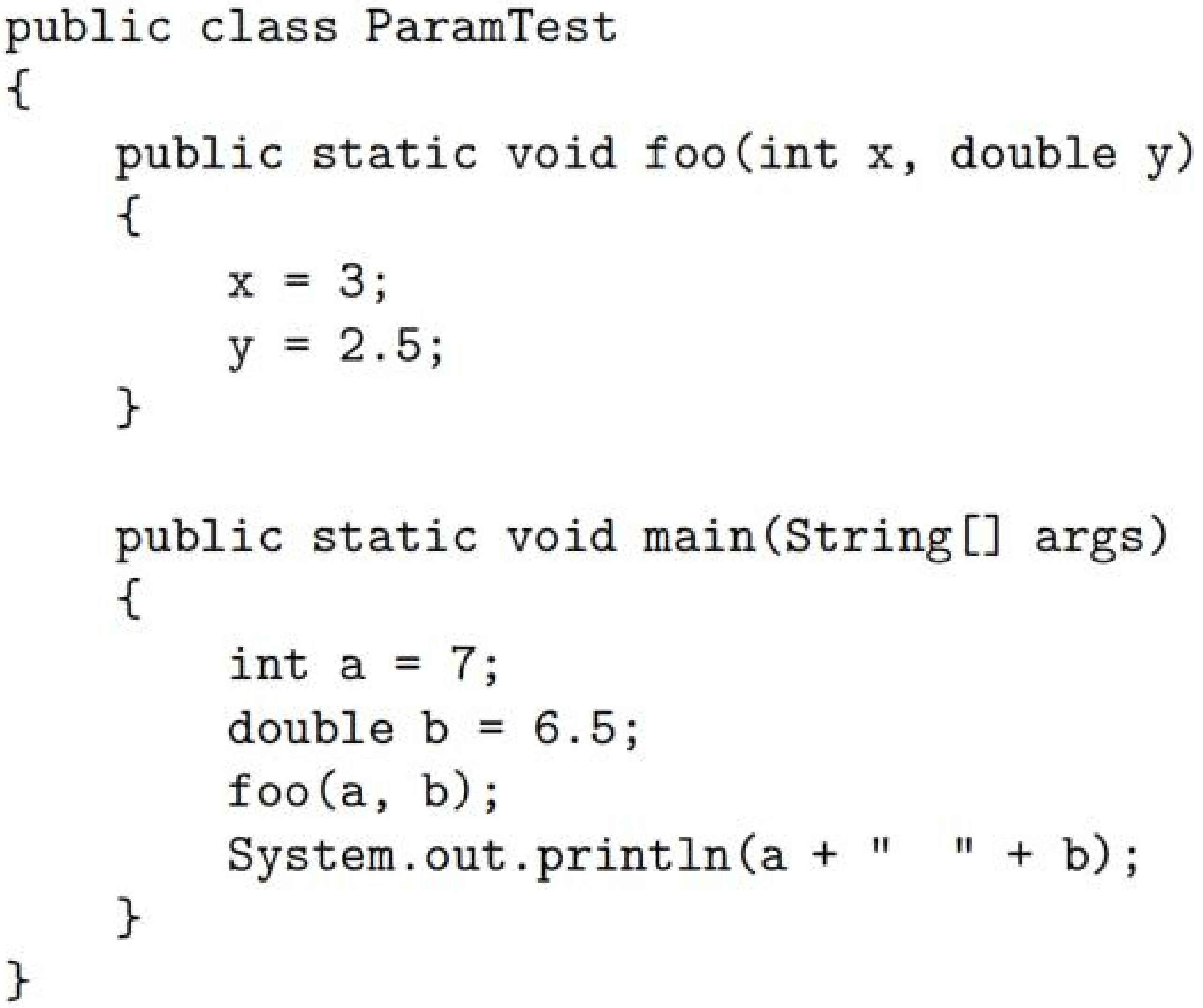
After assignment

both name1 and name2 contain the same address and therefore refer to the same object



the name1 and name2 reference variables are now aliases of each other

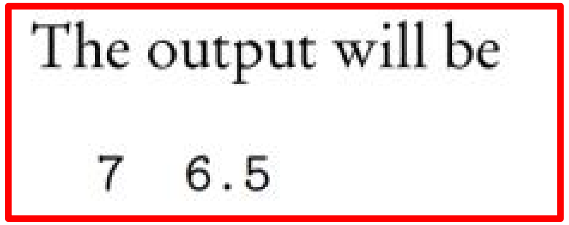
passing primitive data as parameter



```
public class ParamTest
{
    public static void foo(int x, double y)
    {
        x = 3;
        y = 2.5;
    }

    public static void main(String[] args)
    {
        int a = 7;
        double b = 6.5;
        foo(a, b);
        System.out.println(a + "   " + b);
    }
}
```

when primitive data （Boolean， byte， char， String， int， Long， float， double） as parameter， a copy of value transmit to the method， so no matter what you do with this copy in the method， the actual parameter won’t change.

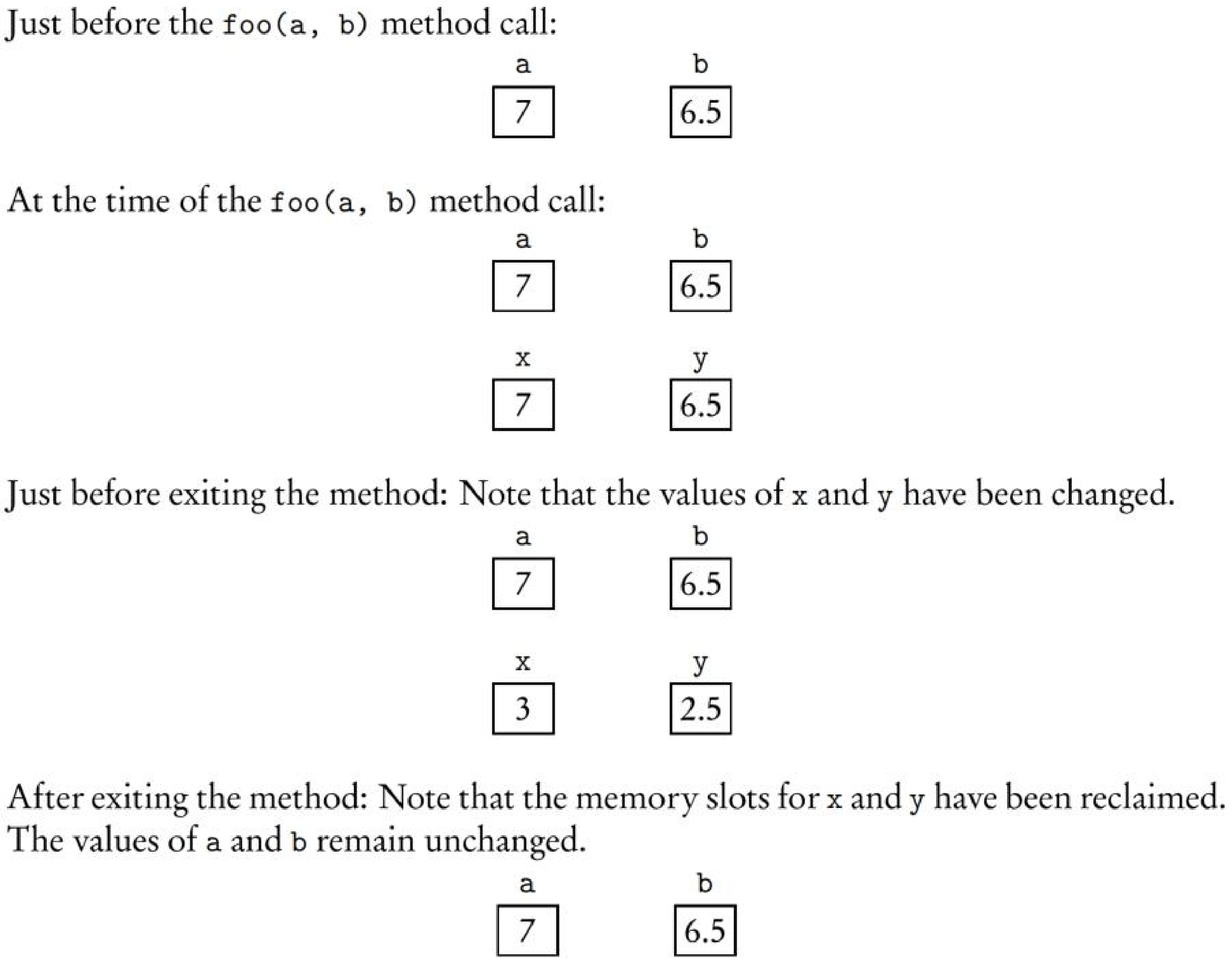
so the result is:



The output will be

```
  7   6.5
```

The arguments a and b remain unchanged, despite the method call!
This can be understood by picturing the state of the memory slots during execution of the program.



Just before the `foo(a, b)` method call:

a	b
7	6.5

At the time of the `foo(a, b)` method call:

a	b
7	6.5

x	y
7	6.5

Just before exiting the method: Note that the values of `x` and `y` have been changed.

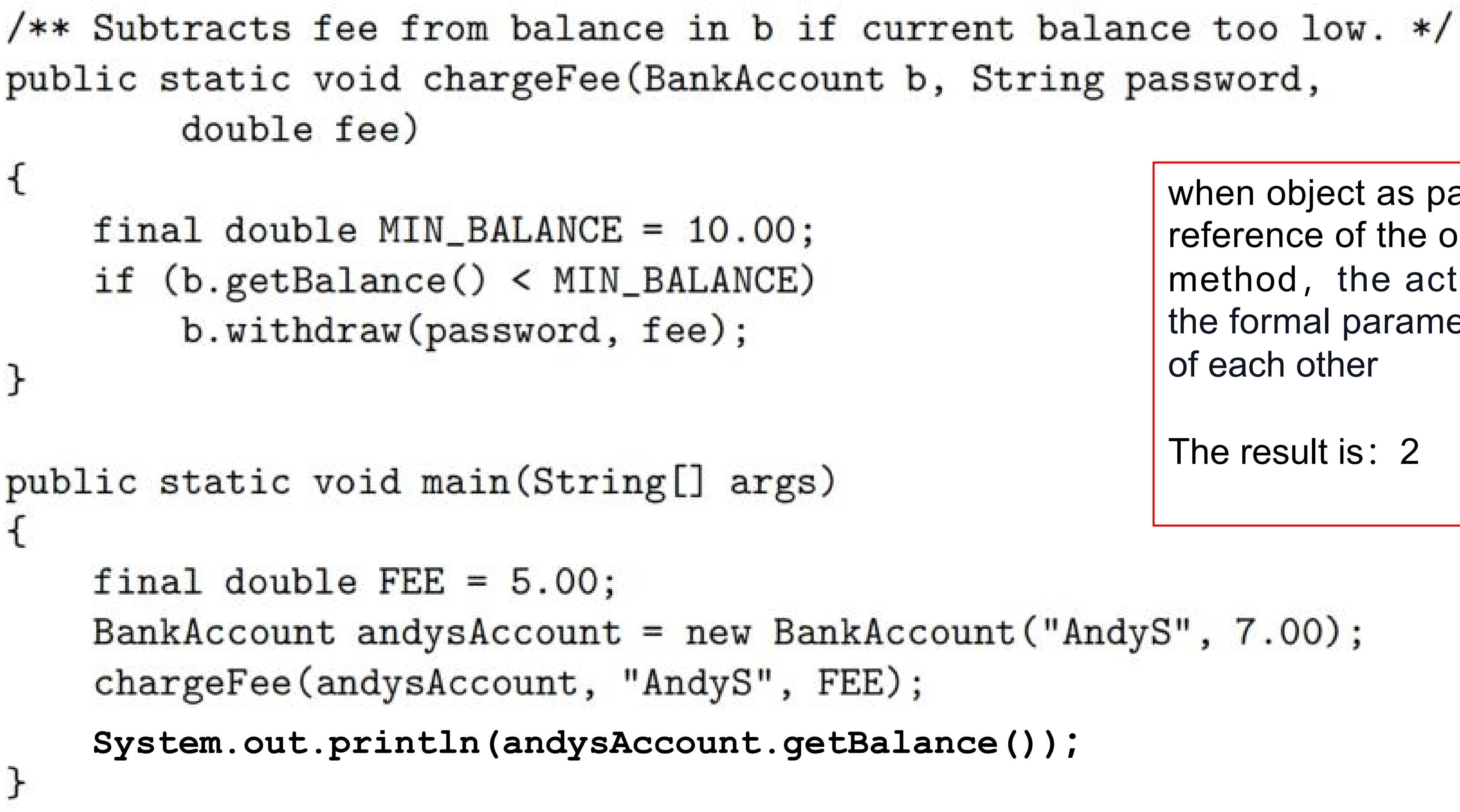
a	b
7	6.5

x	y
3	2.5

After exiting the method: Note that the memory slots for `x` and `y` have been reclaimed.
The values of `a` and `b` remain unchanged.

a	b
7	6.5

Passing objects as parameter

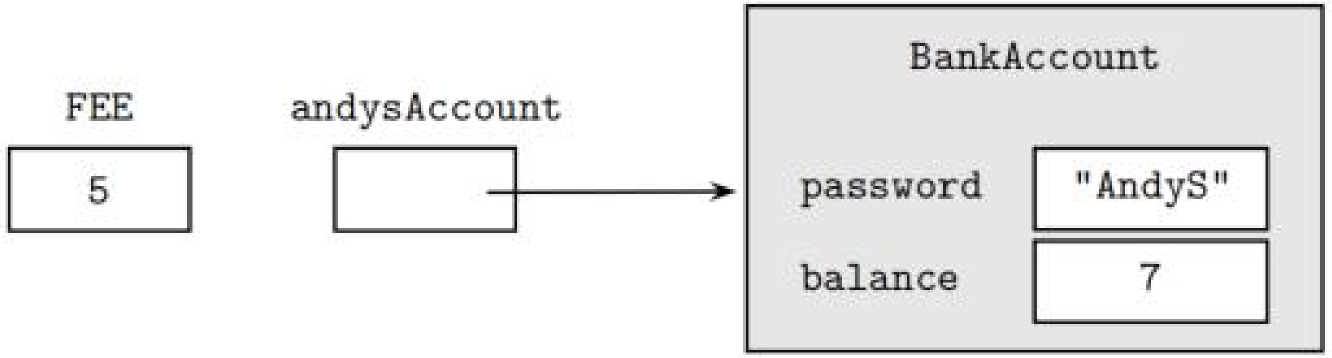


```
/** Subtracts fee from balance in b if current balance too low. */
public static void chargeFee(BankAccount b, String password,
        double fee)
{
    final double MIN_BALANCE = 10.00;
    if (b.getBalance() < MIN_BALANCE)
        b.withdraw(password, fee);
}

public static void main(String[] args)
{
    final double FEE = 5.00;
    BankAccount andysAccount = new BankAccount("AndyS", 7.00);
    chargeFee(andysAccount, "AndyS", FEE);
    System.out.println(andysAccount.getBalance());
}
```

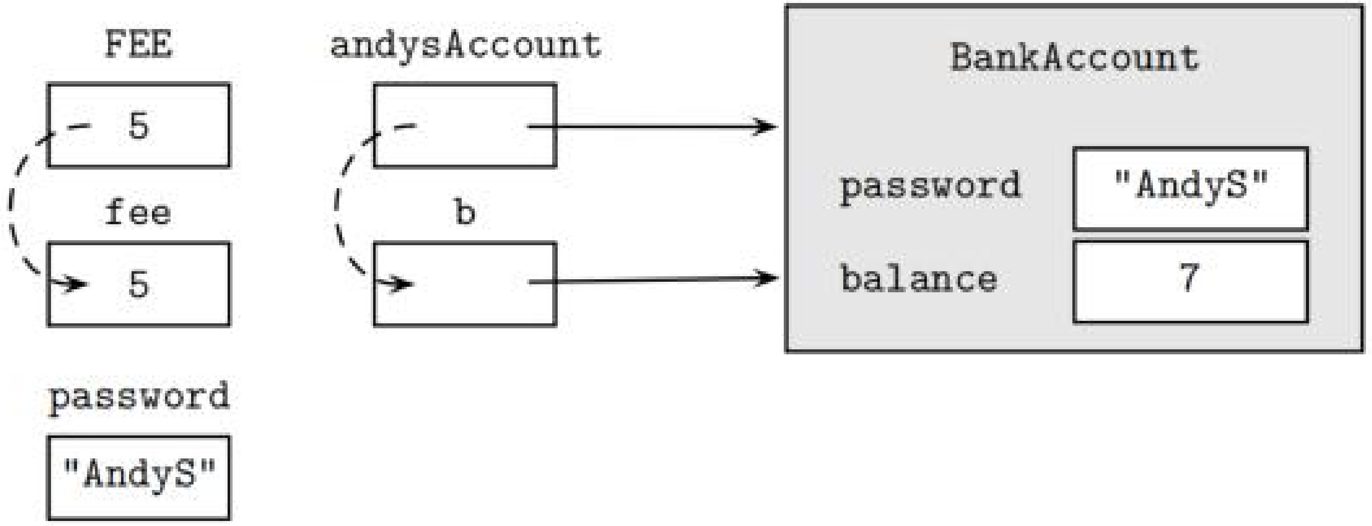
when object as parameter， a copy of reference of the object transmit to the method， the actual parameter and the formal parameter become **aliases** of each other

The result is：2

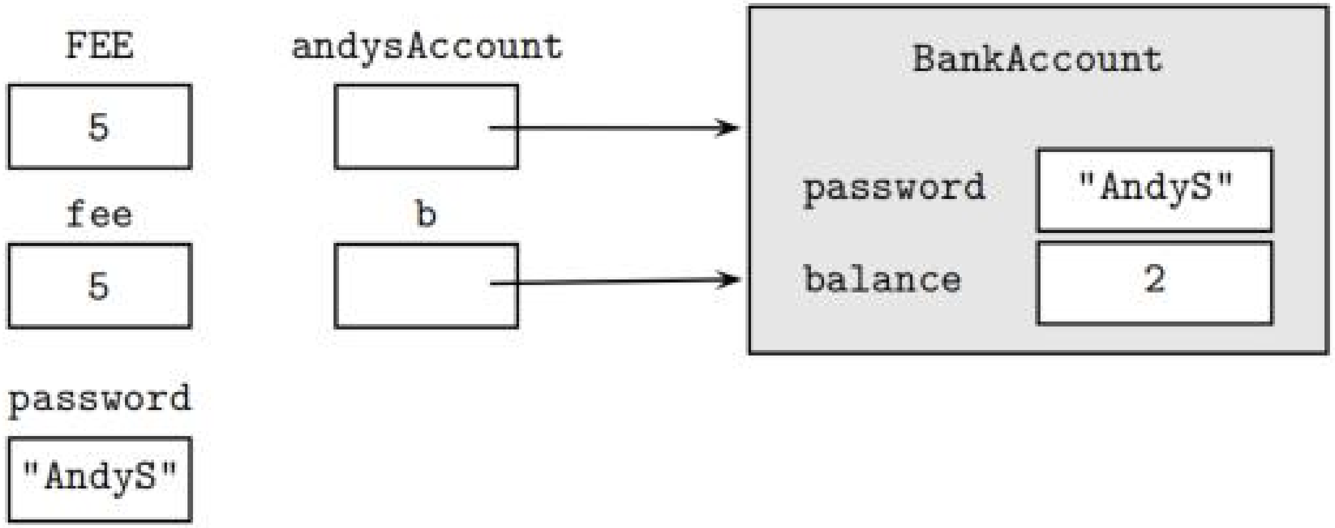
Here are the memory slots before the `chargeFee` method call:



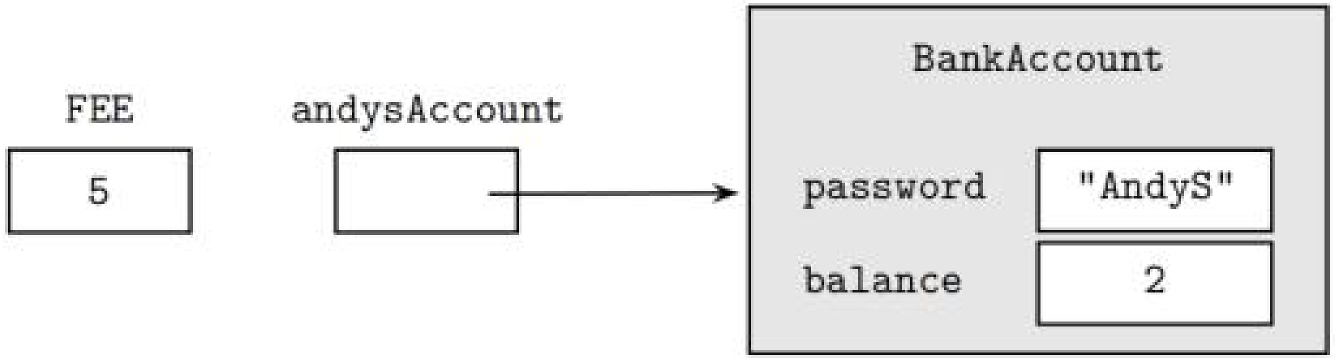
At the time of the `chargeFee` method call, copies of the matching parameters are made:



Just before exiting the method: The `balance` field of the `BankAccount` object has been changed.



After exiting the method: All parameter memory slots have been erased, but the object remains altered.



The null Reference

- An object reference variable that does not currently point to an object is called a *null reference*

- The reserved word `null` can be used to explicitly set a null reference:

```
      name = null;
```

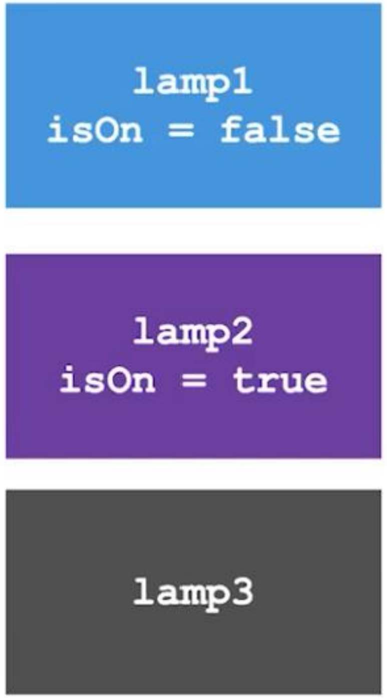
  or to check to see if a reference is currently null:

```
      if (name == null)
         System.out.println ("Invalid");
```

•Attempting to follow a null reference causes a **`NullPointerException`** to be thrown

The null Reference

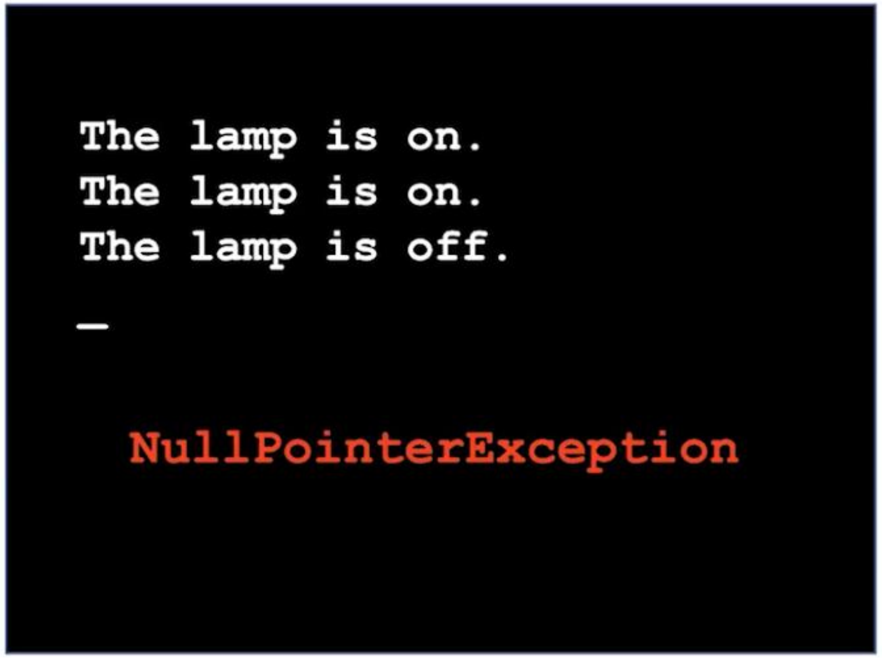
```java
public class Lamp {

   private boolean isOn;

   public void turnOn() {
      isOn = true;
      System.out.println("The lamp is on.");
   }

   public void turnOff() {
      isOn = false;
      System.out.println("The lamp is off.");
   }

   public static void main(String[] args) {
      Lamp lamp1 = new Lamp();
      lamp1.turnOn();
      Lamp lamp2 = new Lamp();
      lamp2.turnOn();
      lamp1.turnOff();
      Lamp lamp3;
      lamp3.turnOn();
   }
}
```



lamp1
isOn = false

lamp2
isOn = true

lamp3



```
The lamp is on.
The lamp is on.
The lamp is off.
_

  NullPointerException
```