

@NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

OBJECT-ORIENTED LANGUAGE AND THEORY

4. SOME TECHNIQUES IN CLASS BUILDING

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



1

2


Goals

- Understand notions, roles and techniques for overloading methods and overloading constructors
- Object member, class member
- How to pass arguments of functions

2

3

Outline

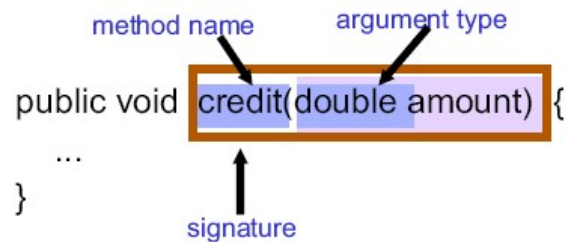
- 
1. Method overloading
 2. Classifier and constant members
 3. Passing arguments to methods

3

4

Method recalls

- Each method has it own signature
- A method signature is composed of:
 - Method's name
 - Number of arguments and their types



```
public void credit(double amount) {  
    ...  
}
```

The diagram illustrates the components of a method signature. In the code snippet, the word `credit` is highlighted in blue, and the text `(double amount)` is also highlighted in blue. A brown rectangular box encloses both `credit` and `(double amount)`. Three arrows point to these elements: one from the label `method name` to `credit`, one from the label `argument type` to `(double amount)`, and one from the label `signature` to the entire brown box.

4

5

1.1. Method overloading

- **Method Overloading:** Methods **in a class** might have the **same name** but **different signatures**:
 - **Numbers of arguments** are different
 - If the numbers of arguments are the same, **types of arguments** must be **different**
- **Advantages:**
 - The same name describes the **same task**
 - Is easier for developers because they don't have to remember **too many method names**. They remember only one with the appropriate arguments.

5

6

Method overloading – Example 1

- Method `println()` in `System.out.println()` has 10 declarations with different arguments: **boolean**, **char[]**, **char**, **double**, **float**, **int**, **long**, **Object**, **String**, and one without argument.
- Do not need to use different names (for example **"printString"** or **"printDouble"**) for each data type to be displayed.

6

7

Method overloading – Example 2

```
class MyDate {
    int year, month, day;
    public boolean setMonth(int m) { ...}
    public boolean setMonth(String s) { ...}
}

public class Test{
    public static void main(String args[]){
        MyDate d = new MyDate();
        d.setMonth(9);
        d.setMonth("September");
    }
}
```

7

8

Method overloading – More info.

- Methods are considered as **overloading** only if they belong to the **same class**
- Only apply this technique on methods describing the **same kind of task**; do not abuse
- When compiling, compilers rely on number or types of arguments to decide which **appropriate method** to call.
→ If there is no method or more than one method to call, an error will be reported.

8

9

Discussion

- Given a following method:
 - 0. `public double test(String a, int b)`
- Let select overloading methods of the given method 0 from the list below:
 1. `void test(String b, int a)`
 2. `public double test(String a)`
 3. `private int test(int b, String a)`
 4. `private int test(String a, int b)`
 5. `double test(double a, int b)`
 6. `double test(int b)`
 7. `public double test(String a, long b)`

9

10

Discussion

```
void prt(String s) { System.out.println(s); }
void f1(char x) { prt("f1(char)"); }
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }
```

- What will happens if we do as follows:
 - `f1(5);`
 - `char x='a'; f1(x);`
 - `byte y=0; f1(y);`
 - `float z = 0; f1(z);...`

10

11

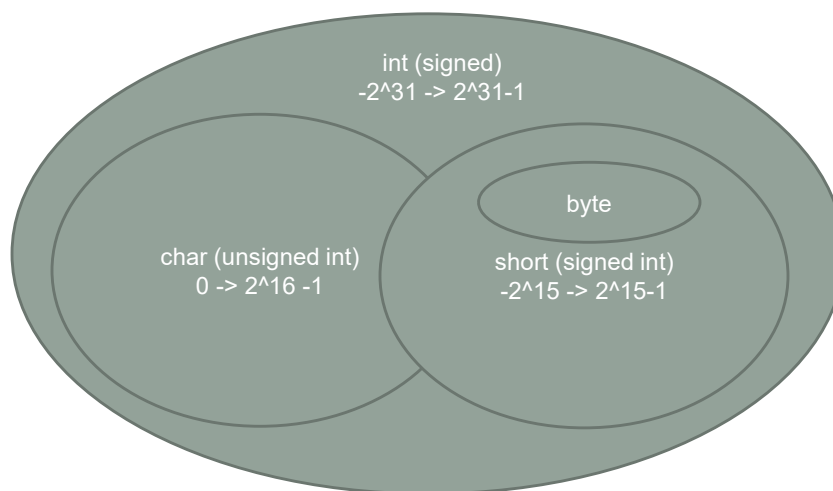
Discussion

```
void prt(String s) { System.out.println(s); }
void f2(short x) { prt("f3(short)"); } => 2 b
void f2(int x) { prt("f3(int)"); } => 4 b
void f2(long x) { prt("f5(long)"); } => 8 b
void f2(float x) { prt("f5(float)"); }
```

- What will happen if we do as follows:
 - `f2(5);`
 - `char x='a'; f2(x);` => 2 b
 - `byte y=0; f2(y);`
 - `float z = 0; f2(z);`
- What will happen if we call `f2(5.5)`?

11

12



12

13

1.2. Constructor overloading

- In different contexts => create objects in different ways
- Any number of constructors with different parameters (following constructor overloading principles)
- Constructors are commonly overloaded to allow for different ways of initializing instances

```
BankAccount new_account =
    new BankAccount();

BankAccount known_account =
    new BankAccount(account_number);

BankAccount named_account =
    new BankAccount("My Checking Account");
```

13

14

Example

```
public class BankAccount{
    private String owner;
    private double balance;
    public BankAccount(){owner = "noname";}
    public BankAccount(String o, double b){
        owner = o; balance = b;
    }
}

public class Test{
    public static void main(String args[]){
        BankAccount acc1 = new BankAccount();
        BankAccount acc2 =
            new BankAccount("Thuy", 100);
    }
}
```

14

15

this keyword

- “this” refers to the **current object**, it is used **inside the class** of the object that it refers to.
- It uses attributes or methods of object through “.” operator, for example:

```
public class BankAccount{
    private String owner;
    public void setOwner(String owner){
        this.owner = owner;
    }
    public BankAccount() { this.setOwner("noname"); }
    ...
}
```

- Call another constructor of the class:
 - `this(parameters);` //first statement in another constructor

15

16

this keyword

In a constructor, the keyword `this` is used to refer to other constructors in the same class

```
...
public BankAccount(String name) {
    super();
    owner = name;
}

public BankAccount() {
    this("TestName");
}

public BankAccount(String name, double initialBalance) {
    this(name);
    setBalance(initialBalance);
}
...
```

16

17

• Example

```

public class Ship {
    private double x=0.0, y=0.0
    private double speed=1.0, direction=0.0;
    public String name;

    public Ship(String name) {
        this.name = name;
    }
    public Ship(String name, double x, double y) {
        this(name); this.x = x; this.y = y;
    }
    public Ship(String name, double x, double y,
        double speed, double direction) {
        this(name, x, y);
        this.speed = speed;
        this.direction = direction;
    }
}
//to be continued...

```

17

18

```

//(cont.)
private double degreeToRadian(double degrees) {
    return(degrees * Math.PI / 180.0);
}
public void move() {
    move(1);
}
public void move(int steps) {
    double angle = degreesToRadians(direction);
    x = x + (double)steps*speed*Math.cos(angle);
    y = y + (double)steps*speed*Math.sin(angle);
}
public void printLocation() {
    System.out.println(name + " is at ("
        + x + "," + y + ").");
}
} //end of Ship class

```

18

19

Outline

1. Method overloading
- ➔ 2. Classifier and constant members
3. Passing arguments to methods

19

20

2.1. Constant members

- An attribute/method that can not change its values/content during the usage.
- Declaration syntax:

```
access_modifier final data_type  
    CONSTANT_VARIABLE = value;
```

- For example:

```
final double PI = 3.141592653589793;  
public final int VAL_THREE = 39;  
private final int[] A = { 1, 2, 3, 4, 5, 6 };
```

20

21

2.1. Constant members (2)

- Typically, constants associated with a class are declared as **static final** fields for easy access
 - A common convention is to use only uppercase letters in their names

```
public class MyDate {
    public static final long SECONDS_PER_YEAR =
        31536000;
    ...
}
...
long years = MyDate.getMillisSinceEpoch() /
    (1000*MyDate.SECONDS_PER_YEAR);
```

javafx.swing

Class JOptionPane

ERROR_MESSAGE

public static final int ERROR_MESSAGE

21

22

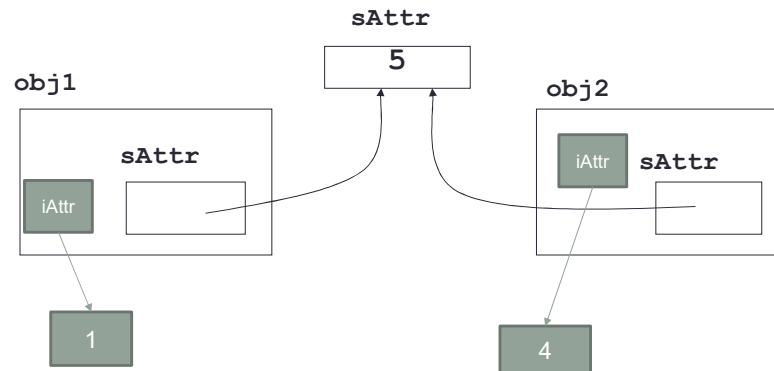
2.2. Classifier members

- Members may belong to either of the following:
 - The whole class (class variables and methods, indicated by the keyword **static** in Java)
 - Individual objects (instance variables and methods)
- Static attributes and methods belong to the class
 - Changing a value in one object of that class changes the value for all of the objects
- Static methods and fields can be accessed without instantiating the class
 - Static methods and fields are declared using the static keyword

22

Static parts: are shared between all objects

- sAttr: static (class/classifier scope)
- iAttr: instance (object/instance scope)



23

24

Instance member

vs. Classifier member

- | | |
|---|---|
| <ul style="list-style-type: none"> • Attributes/methods can only be accessed via objects • Each object has its own copy of an object's attribute • Values of an attribute of different objects are different. | <ul style="list-style-type: none"> ▫ Attributes/methods can be accessed through class ▫ All objects have the same copy of class attributes ▫ Values of a class attribute of different objects are the same. |
|---|---|

24

25

Static members in Java

- Regular members are members of objects
- Class members are declared as **static**
- Syntax for declaring static member:
access_modifier static data_type varName;
- Example:

```
public class MyDate {
    public static long getMillisSinceEpoch() {
        ...
    }
    public String getMonth(){
        long ms = getMillisSinceEpoch();
        ...
    }
    long millis = MyDate.getMillisSinceEpoch();
}
```

```
MyDate date1 = new MyDate();
date1.getMonth(); date1.getMillisSinceEpoch();
```

25

26

Example: Class JOptionPane in javax.swing

- Attributes

Field Summary	
static int CANCEL_OPTION	Return value from class method if CANCEL is chosen
static int CLOSED_OPTION	Return value from class method if user closes window CANCEL_OPTION or NO_OPTION.
static int DEFAULT_OPTION	Type used for showConfirmDialog.
static int ERROR_MESSAGE	Used for error messages.
static int WARNING_MESSAGE	Used for warning messages.
static int YES_NO_CANCEL_OPTION	Type used for showConfirmDialog.
static int YES_NO_OPTION	Type used for showConfirmDialog.
static int YES_OPTION	Return value from class method if YES is chosen.

- Methods:

static void	showMessageDialog (Component parentComponent, Object message)	Brings up an information-message dialog titled "Message".
static void	showMessageDialog (Component parentComponent, Object message, String title, int messageType)	Brings up a dialog that displays a message using a default icon determined by the messageType parameter.
static void	showMessageDialog (Component parentComponent, Object message, String title, int messageType,	Brings up a dialog displaying a message specifying all parameters

26

27

Example – using static attributes and methods in class JOptionPane

```
JOptionPane.showMessageDialog(null, "Ban da thao tac loi", "Thong bao loi", JOptionPane.ERROR_MESSAGE);
```



```
JOptionPane.showConfirmDialog(null, "Ban co chac chan muon thoat?", "Hay lua chon", JOptionPane.YES_NO_OPTION);
```



27

28

Example – using static attributes and methods in class JOptionPane (2)

```
Object[] options = { "OK", "CANCEL" };
JOptionPane.showOptionDialog(null, "Nhan OK de tiep tuc", "Canh bao", JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[0]);
```



28

29

Static member (2)

- Modifying value of a **static** member in an object will modify the value of this member in all other objects of the class.
- **Static** methods can access only **static** attributes and can call **static** methods in the same class.

29

30

Example 1

```
class TestStatic{
    public static int iStatic;
    public int iNonStatic;
}

public class TestS {
    public static void main(String[] args) {
        TestStatic obj1 = new TestStatic();
        obj1.iStatic = 10; obj1.iNonStatic = 11;
        System.out.println(obj1.iStatic+", "+obj1.iNonStatic);
        TestStatic obj2 = new TestStatic();
        System.out.println(obj2.iStatic+", "+obj2.iNonStatic);
        obj2.iStatic = 12;
        System.out.println(obj1.iStatic+", "+obj1.iNonStatic);
    }
}
```



```
10,11
10,0
12,11
```

30

31

Example 2

```
public class Demo {
    int i = 0;
    void increase(){ i++; }
    public static void main(String[] args) {
        increase() ;
        System.out.println("Gia tri cua i la" + i);
    }
}
```

non-static method increase() cannot be referenced from a static context
non-static variable i cannot be referenced from a static context

31

Java static methods – Example

```
class MyUtils {
    . . .
    //===== mean
    public static double mean(int[] p) {
        int sum = 0;
        for (int i=0; i<p.length; i++) {
            sum += p[i];
        }
        return ((double)sum) / p.length;
    }
    . . .
}
...

// Calling a static method from outside of a class
double avgAtt = MyUtils.mean(attendance);
```

32

When static?

33

34

Outline

1. Method overloading
2. Classifier and constant members
- 3. Passing arguments to methods

34

3. Arguments passing to methods

- We can use any data types for arguments for methods or constructors
 - Primitive data types
 - References: array and object
- Example:

```
public Polygon polygonFrom(Point[] corners){
    // method body goes here
}
```

35

36

3.1. Variable arguments

- An arbitrary number of arguments, called *varargs*
- Syntax in Java:
 - `methodName(data_type... parameterName)`

- Example

- Declaration:

```
public PrintStream printf(String format,
                          Object... args)
```

- Usage:

```
System.out.printf ("%s: %d, %s\n",
                  name, idnum, address);
System.out.printf ("%s: %d, %s, %s, %s\n",
                  name, idnum, address, phone, email);
```

36

- Example

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
        * (corners[1].x - corners[0].x)
        + (corners[1].y - corners[0].y)
        * (corners[1].y - corners[0].y) ;
    lengthOfSide1 = Math.sqrt(squareOfSide1);
    //create & return a polygon connecting the Points
}
```

- **corners** is considered as an array
- You can pass an array or a sequence of arguments

37

3.2. Passing by values

- C++
 - Passing values, pointers
- Java
 - Passing values

38

39

Java: Pass-by-value for all types of data

- Java passes all arguments to a method in form of pass-by-value: Passing value/copy of the real argument
 - For arguments of value-based data types (primitive data types): passing value/copy of primitive data type argument
 - For argument of reference-based data types (array and object): passing value/copy of original reference.
- Modifying formal arguments does not effect the real arguments

39

Discussion:

- What will happen if:
 - We modify the internal state of object parameters inside a method?
 - We modify the reference to an object?

40

41

a. With value-based data type

- Primitive values can not be changed when being passed as a parameter

```
public void method1(){
    int a = 0;
    System.out.println(a); // outputs 0
    method2(a);
    System.out.println(a); // outputs 0
}
```

```
void method2(int a){
    a = a + 1;
}
```

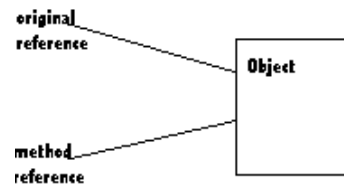
- Is this swap method correct?
- ```
public void swap(int var1, int var2) {
 int temp = var1;
 var1 = var2;
 var2 = temp;
}
```

41

42

## b. With reference-based data type

- Pass the references by value, not the original reference or the object



- After being passed to a method, a object has at least two references

42

## Passing parameters

```
public class ParameterModifier
{
 public void changeValues (int f1, Num f2, Num f3)
 {
 System.out.println ("Before changing the values:");
 System.out.println ("f1\tf2\tf3");
 System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

 f1 = 999;
 f2.setValue(888);
 f3 = new Num (777);

 System.out.println ("After changing the values:");
 System.out.println ("f1\tf2\tf3");
 System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
 }
}
```

43

## Passing parameters

```
public class ParameterTester
{
 public static void main (String[] args)
 {
 ParameterModifier modifier = new ParameterModifier();

 int a1 = 111;
 Num a2 = new Num (222);
 Num a3 = new Num (333);

 System.out.println ("Before calling changeValues:");
 System.out.println ("a1\ta2\ta3");
 System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

 modifier.changeValues (a1, a2, a3);

 System.out.println ("After calling changeValues:");
 System.out.println ("a1\ta2\ta3");
 System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
 }
}
```

Before calling changeValues:

a1 a2 a3  
111 222 333

Before changing the values:

f1 f2 f3  
111 222 333

After changing the values:

f1 f2 f3  
999 888 777

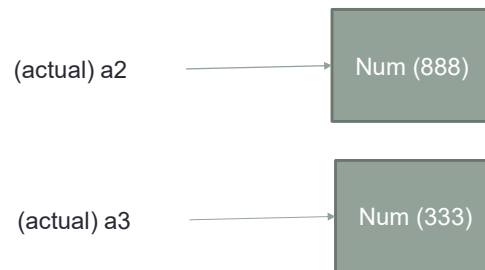
After calling changeValues:

a1 a2 a3  
111 888 333

44

45

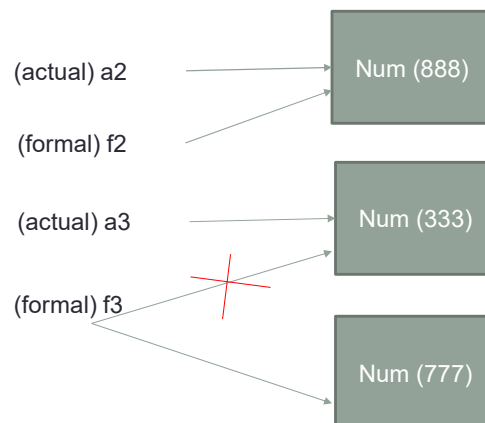
## Inside the method changeValues()



45

46

## Inside the method changeValues()



46

47

## For example

```
public class Point {
 private double x;
 private double y;
 public Point() { }
 public Point(double x, double y) {
 this.x = x; this.y = y;
 }
 public void setX(double x) { this.x = x; }
 public void setY(double y) { this.y = y; }
 public void printPoint() {
 System.out.println("X: " + x + " Y: " + y);
 }
}
```

47

48

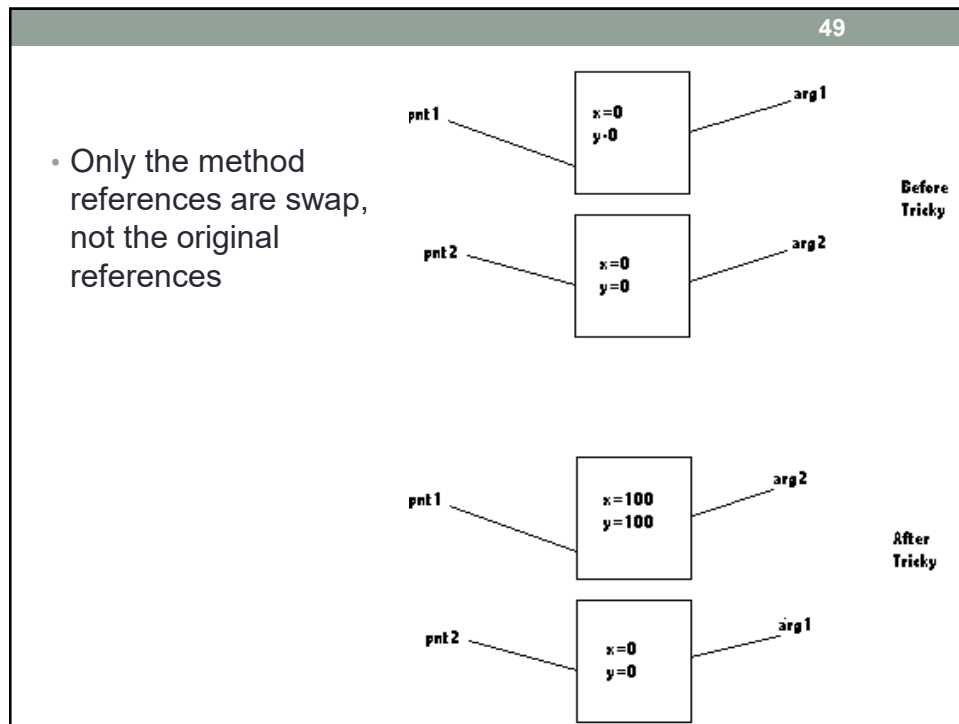
```
public class Test {
 public static void tricky(Point arg1, Point arg2) {
 arg1.setX(100); arg1.setY(100);
 Point temp = arg1;
 arg1 = arg2; arg2 = temp;
 }
 public static void main(String [] args) {
 Point pnt1 = new Point(0,0);
 Point pnt2 = new Point(0,0);
 pnt1.printPoint(); pnt2.printPoint();
 System.out.println(); tricky(pnt1, pnt2);
 pnt1.printPoint(); pnt2.printPoint();
 }
}
```



```
X: 0.0 Y: 0.0
X: 0.0 Y: 0.0
X: 100.0 Y: 100.0
X: 0.0 Y: 0.0
Press any key to continue . . . _
```

48





49