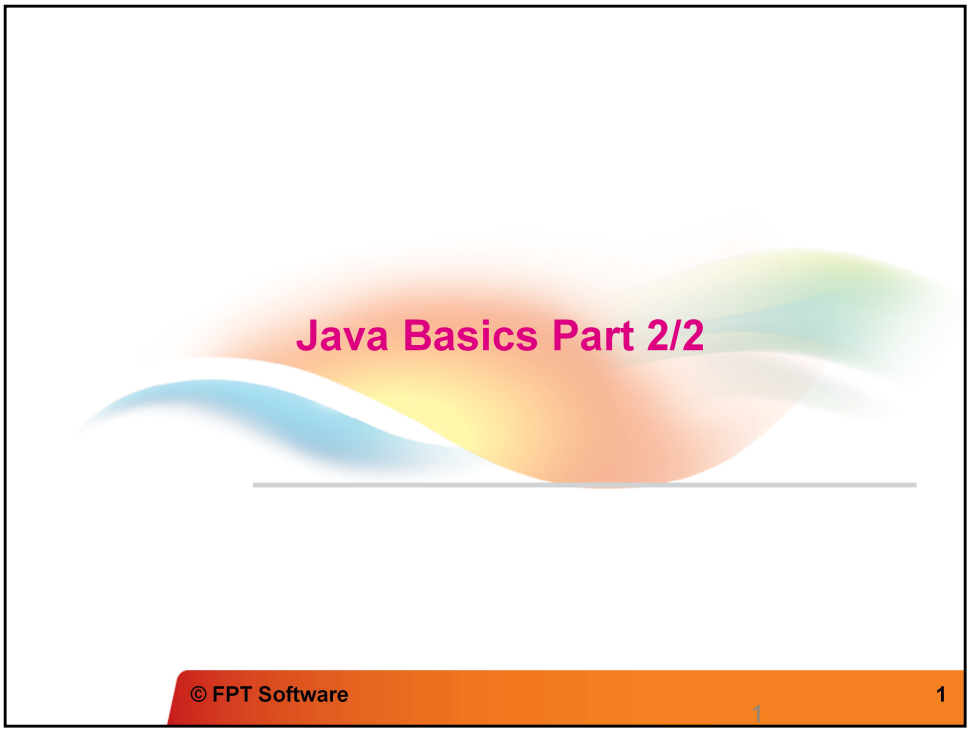


<*Lưu ý: không xóa nội dung cũ mà viết thêm lên trên như nhật ký*>

Tài liệu đào tạo: <*môn học, buổi học*>

Phiên bản tài liệu:

Người cập nhật:

Ngày cập nhật:

Tóm tắt nội dung cập nhật chính:

Ngày ban hành sử dụng:

Agenda

- More about class
- Inheritance
- Polymorphism
- Abstract class and Interface
- Value type/Referenced type
- String class
- Date/Time operators
- MVC pattern

© FPT Software

MORE ABOUT CLASS

Content

- Property
- Overload
- Constant and read-only members
- Static class and static class features
- Inner/Nested/Anonymous class

© FPT Software

Getter and Setter

```
//package ?
public class Car{
   private int numberWheels;

   public int getNumberWheels{ // get + member name
      return numberWheels;
   }

   public void setNumberWheels(int value){
                                // set + member name
      if ((value % 2 == 0) && (value >= 4) &&
          (value <= 10)){
         numberWheels = value;
      }
   }
   …
}
//Example
Car aCar = new Car();
aCar.setNumberWheels(6);            // method call
```

Overload

```
class Car{
   …
   public void speedUp(){
      speedUp(1.5);
   }
    // overload: same return type and name,
    //           different parameter set
   public void speedUp(float step){…}
   …
}
```

© FPT Software

Same return type

Constant with final keyword

```
class A{
    final int a = 1;            // a constant
    void increase(){ a++;}      // error
}
```

© FPT Software

Same return type

Static keyword

```
class A{
   private static int counter = 0;
   public static int increase(){
      return ++counter;
   }
}
int i = A.increase(); // i = 1
A a = new A();
int j = a.increase(); // error: no static method
                      // call from object
int k = A.increase(); // k = 2
…
static class B{} // Error: no static class in Java
```

Inner nested class

```
class OuterClass{
   private int i;
   class InnerClass{          // an inner member class
      void methodA(){
         i = 5;               // OK, event i is private
      }
      void methodB(){
         int i = 3;           // hide/shadowing the outer i
                              // the outer i member is unchanged
      }
   }
   void methodA(){InnerClass oIC = new InnerClass();}
}
…
OuterClass oOC = new OuterClass();
// Inner class is public by default
OuterClass.InnerClass oIC = oOC.new InnerClass();
```

© FPT Software

A member class can take any access modifier as another member.

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

It is a way of logically grouping classes that are only used in one place.

It increases encapsulation.

Nested classes can lead to more readable and maintainable code.

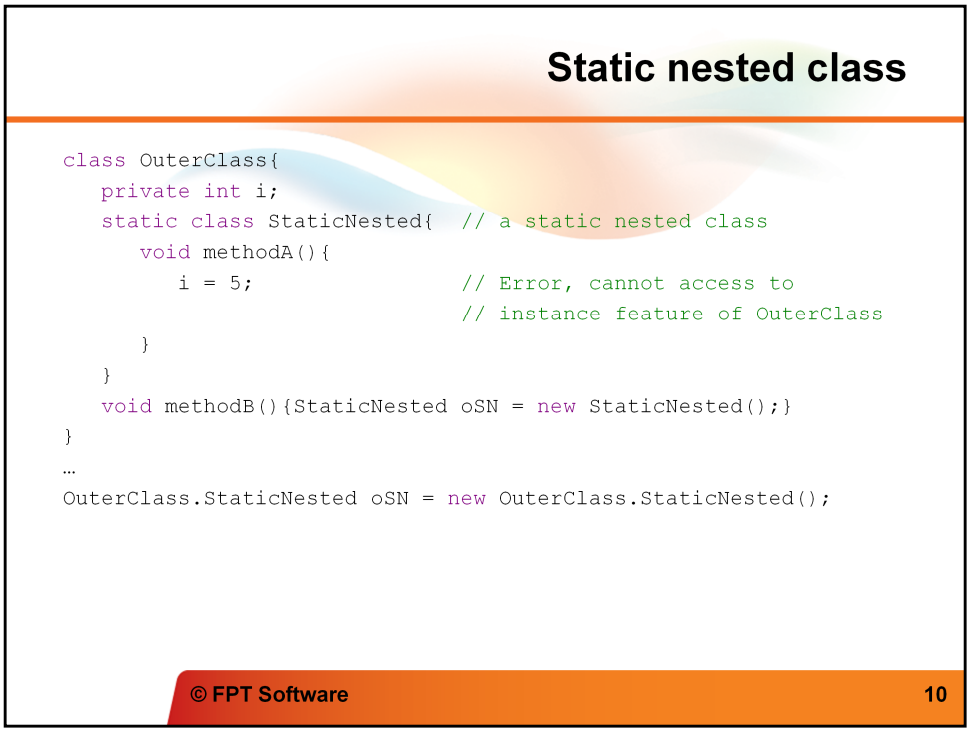
Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.

Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.



A member class can take any access modifier as another member.

A nested class is a member of its enclosing class. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

It is a way of logically grouping classes that are only used in one place.

It increases encapsulation.

Nested classes can lead to more readable and maintainable code.

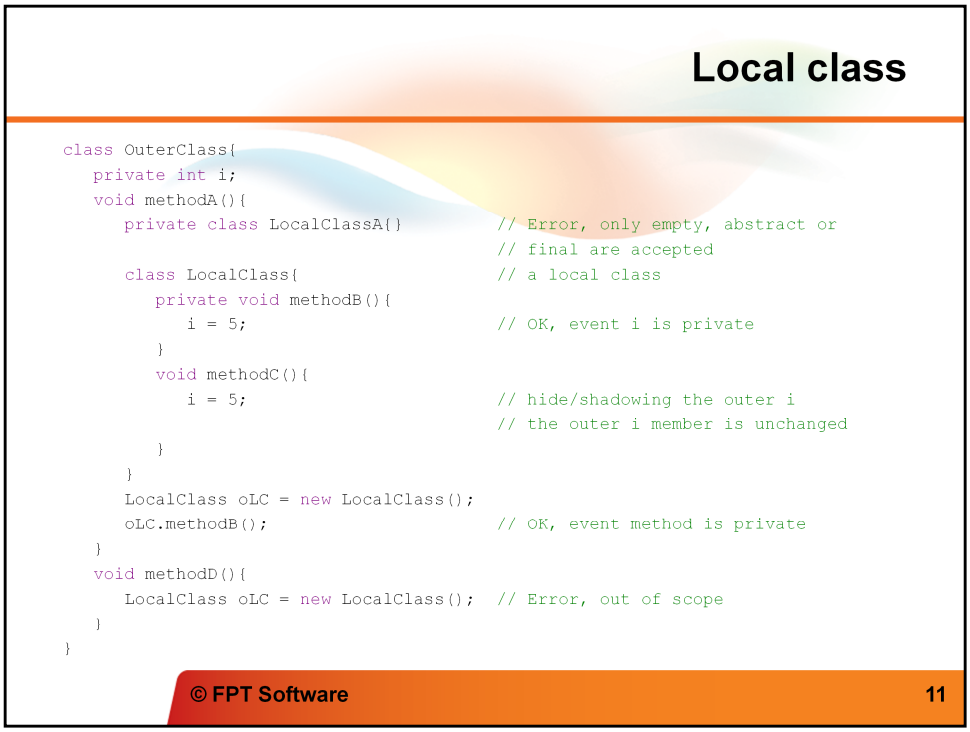
Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.

Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.



A local class is declared inside a method of another class

Only an empty, abstract or final access modifier is accepted for local class

Local class: Use it if you need to create more than one instance of it, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

It is a way of logically grouping classes that are only used in one place.

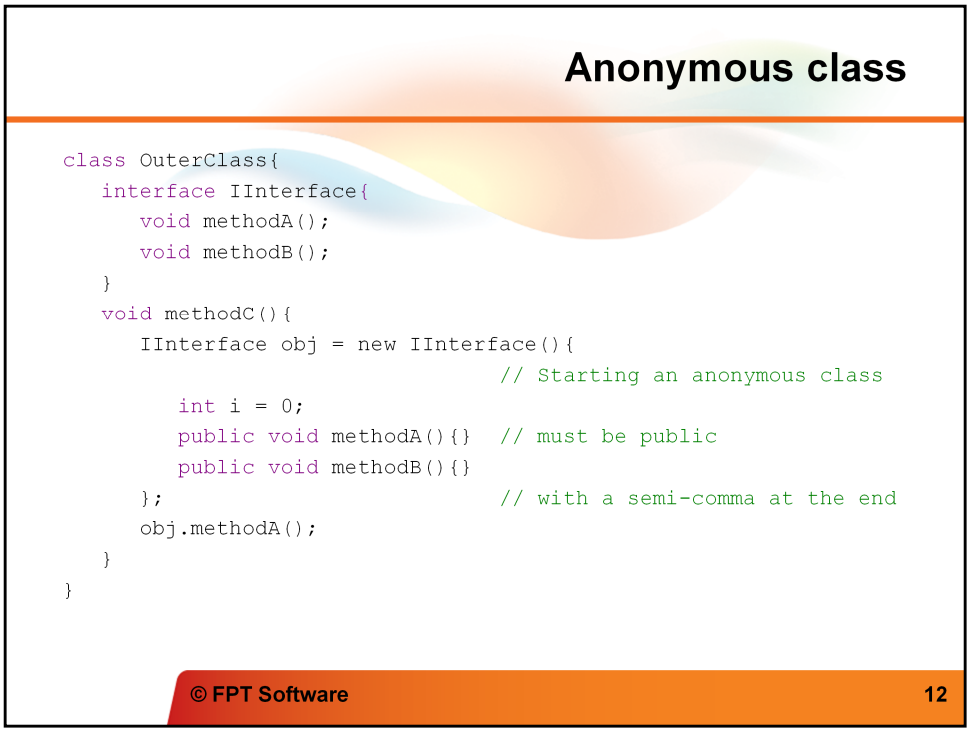
It increases encapsulation.

Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.



A local class is declared inside a method of another class

Only an empty, abstract or final access modifier is accepted for local class

Local class: Use it if you need to create more than one instance of it, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

It is a way of logically grouping classes that are only used in one place.

It increases encapsulation.

Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

Summary

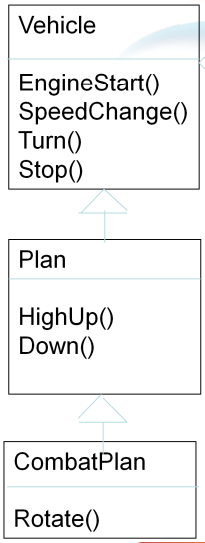
- Property: method with getter and setter
- Method overload: methods have the same return type and name, different parameter list
- Constant member: with final keyword
- Static features: Class feature, not object's one
- Inner class: class inside another class
- Anonymous class: class used once

INHERITANCE

Content

- Hierarchy
- Base/super class
- Derived/sub class
- Protected access modifier
- Final declaration

Inheritance – Extension



Car

```
class Car extends Vehicle {…}
// Car is a kind of Vehicle
// Car: derived/sub class
// Vehicle: base/super class
Vehicle v = new Car();        // OK
Vehicle v = new Plan();       // OK too
Vehicle v = new CombatPlan(); // OK
Plan p = new CombatPlan();    // OK
Car c = new Vehicle();        // error
Car c = new Plan();           // error
```

© FPT Software

Talk about Object class: Object class is the base class of all class

Accessibility modifier: protected

```
class Car{
   protected int NumberWheels;
   protected String MainColor;
   protected int NumberRearPorts;
   protected bool isWithUpperWindow;
   protected int NumberSeats;
   protected float CylinderVolume;
   …
}
```

this and super

```
class A{
   protected int i;
   protected int aMethod(int i){
      this.i = i;
      return i;
   }
}
class B extends A{
   public int aMethod(int i){
      return super.aMethod(i);
   }
}
```

© FPT Software

"this" => current class object

"super" => parent class object

Final: redefinition forbidden

```
final class A{}
class B extends A              // error

class D{
   final int a(){return 1;}
}
class E extends D{
   final int a(){return 2;} // error
}
```

Summary

- Hierarchie: Inheritance structure
- Base/super class: ascendant in inheritance
- Derived/sub class: descendant in inheritance
- Protected access modifier: for descendent access
- Final declaration: no redefinition allowed

POLYMORPHISM

Content

- Feature hiding
- Feature overriding
- Virtual method

© FPT Software

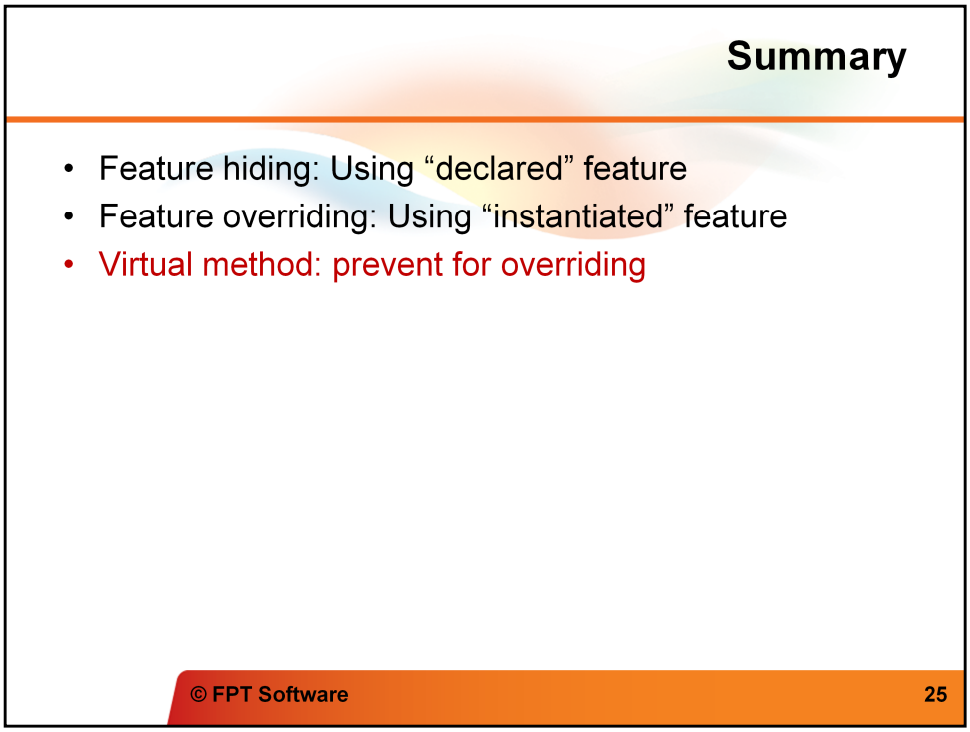
Feature hiding

```
class A{
   public static int defaultTempeture(){
      return 25;
   }
}
class B extends A{
    @Override //Annotation
   public static int defaultTempeture(){
      return 28;
   }
}
A a = new B();
int x = a.defaultTempeture(); // x = 25
```


Present also “static” term

Feature override

```
class A{
    public int DefaultTempeture(){
        return 25;
    }
}
class B extends A{
    // override: used for "deferred loading"
    @Override
    public int DefaultTempeture(){
        return 28;
    }
}
A a = new B();
int x = a.DefaultTempeture(); // x = 28
```



Virtual method: prevent for overriding

→ final method

ABSTRACT CLASS AND INTERFACE

Content

- Abstract method / Abstract class
- Interface
- Different between these two items

© FPT Software

Abstract

```
abstract class A{        // having at least one abstract method
   // abstract method: no implementation
   public abstract int DefaultTempeture();

   // ordinal method: with implementation
   public int TempIncStep(){
      return 1;
   }
}
class B extends A{}
class C extends B{}
   @override
   public int DefaultTempeture(){
      return 25;
   }
}
A a = new A(); // error: no infor about DefaultTempeture behavior
B a = new B(); // error: no infor about DefaultTempeture behavior
C a = new C(); // OK
A a = new C(); // OK
B a = new C(); // OK
```

© FPT Software

Abstract class has abstract method (and normal method too)

Abstract method has not implementation

Derived class has to implement ALL abstract inherited methods

IShape

AShape

|

Cirle Rectangle

Interface

```
interface IA{                    // Interface is a special "abstract" class
   int i;                        // Error : no member is allowed
   int DefaultTempeture();       // no abstract keyword, no access modifier,
                                 // public access modifier is fixed
   int TempIncStep(){            // Error: no ordinal method allowed
      return 1;
   }
}
abstract class A implements IA{} // abstract class can prevent the
                                    // implementation of an interface
class B implements IA{           // non-abstract class, when declared to use
                                 // an interface, must implement all methods
                                 // declared in the interface
   public int DefaultTempeture(){return 1;}
}
class C extends A{
   public int DefaultTempeture(){return 2;}
}
IA a = new B(); IA b = new C();  // Interface is a type
```

© FPT Software

No normal method in interface

Declare variable of interface type

All feature are public

"Multi inheritance"

```
class A1{void a1(){}}
class A2{void a2(){}}
class A:A1, A2{}
                // Error: multi inheritance forbidden

interface IA1{void ia();}
interface IA2{void ia();}
class A:IA1, IA2{
                // OK multi interface implementation
   void ia(){}
   // one implementation for all interface
   // no way to implement an interface specific
   // behavior
}
```

© FPT Software

No normal method in interface

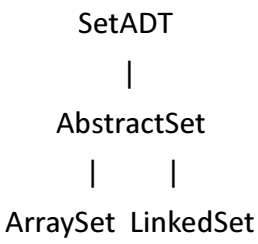
Abstract class vs Interface

- Abstract
  - With member
  - With non-abstract method
  - No multi-heritance
- Interface
  - Without member, only constant
  - Without non-abstract method
  - With multi-implementation (multi-heritance)

Summary

- Abstract method: No implementation
- Abstract class: At least one abstract method
- Interface: only abstract method
- Multi inheritance forbidden
- Multi interface implementation allowed
- Different between these two terminations

© FPT Software



VALUE TYPE AND REFERENCE TYPE

Value type vs reference type

```
int a, b;
a = 1; b = 1;
boolean x = a == b;     // true
b = a; a = 2;           // b = 1
class A{
  public int i = 1;
}
A a = new A(), b = new A();
boolean x = a == b;     // false
b = a; a.i = 2;         // b.i = 2
a = null;
a = new A();
a.i = 3;                // b.i = 2
```

Pass by value/by “reference”

```
class A{
   private int i;
   public int setI(int i){this.i = i;}
   public int getI(){return i;}
}
…
public void M1(A a){
   A b = new A(); b.setI(7); a = b;
}
public void M2(A a){a.setI(9);}
…
A a = new A(); a.setI(5); // a.i = 5
M1(a);                    // a.i = 5
M2(a);                    // a.i = 9
```

© FPT Software

There is no pass by reference in java: we cannot change the reference passed to a new value as in M1

Pass by value: copy the value of input to a new copy

For primitive type: copy primitive value

For object/reference type: copy the object reference (“pointer”) => cannot change the object but can do with object content

String literal/interning

```
String x1 = "ABC";          // Literal/Intern
String x2 = "ABC";          // Literal/Intern
String x3 = "A" + "BC";     // Literal/Intern
String x4 = new String("ABC"); // Object
boolean b = x1 == x2;       // true
b = x1 == x3;               // true
b = x2 == x4;               // false
b = x1.equals(x3);          // true
b = x2.equals(x4);          // true
b = x1 == "ABC";            // true
b = x4 == "ABC";            // false
b = x4.equals("ABC");       // true
```

© FPT Software

String intern/literal is immutable. Any reference to this intern/literal string uses an unique value.

Summary

- Value type: work on value
- Referenced type: work on reference/pointer

© FPT Software

STRING CLASS

String class (1/3)

```
String s1 = "fsoft", s2 = "  fpt.vn ", s3, s4;
int i = s1.length();              // i = 5
boolean b = s1.isEmpty();         // false
char c = s1.charAt(i - 1);        // c = 't'
i = s1.compareTo(s2);             // i > 0, sort order
i = s1.compareToIgnoreCase(s2); // case sensitive
s3 = s1.concat(s2);               // "fsoft  fpt.vn "
s4 = s1 + s2;                     // "fsoft  fpt.vn "
b = s3 == s4;                     // false - String is an object
b = s3.equalsIgnoreCase(s4);      // true
s3 = s4.substring(3);             // "ft  fpt.vn "
s3 = s4.substring(3, 5);          // "ft"
s3 = s1 + s2;                     // "fsoft  fpt.vn "
b = s3.contains(s1);              // true
b = s3.endsWith(s2);              // true
b = s3.startsWith(s1);            // true
b = s3.startsWith("soft", 1);   // true
```

String class (2/3)

```
i = s3.indexOf('f');              // 0
i = s3.indexOf(s2);               // 5
i = s3.indexOf("f", 2);           // 3
i = s3.lastIndexOf('f');          // 7
i = s3.lastIndexOf("f", 6);       // 3

s4 = s3.replace("t ", "t.");      // "fsoft. fpt.vn "
s4 = s3.trim();                   // "fsoft  fpt.vn"
String[] s5 = s3.split("[ f]");// regular expression
                        // {"", "so", "t", "", "", "pt.vn"}
s5 = s3.split("[f ]", 3);         // {"", "so", "t  fpt.vn"}
char[] s7 = s3.toCharArray();  // {'f', 's', 'o', 'f',
           // 't', ' ', ' ', 'f', 'p', 't', '.', 'v', 'n'}
s4 = s3.toUpperCase();            // "FSOFT  FPT.VN"
s2 = s4.toLowerCase();            // "fsoft  fpt.vn"
```

String class (3/3)

```
// Removes whitespace between a word character and . or ,
String pattern = "(\\w)(\\s+)([\\.,])";
String s1 = "abc . def ,";
s1 = s1.replaceAll(pattern, "$1$3"); // "abc. def,"
```

DATE TIME OPERATOR

Date operators (1/2)

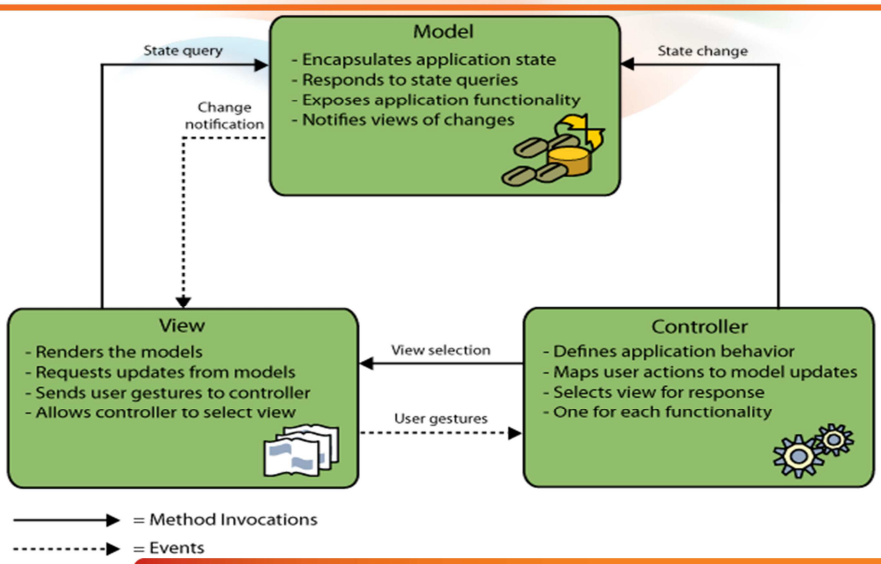
```
import java.util.*;
import java.text.SimpleDateFormat;
SimpleDateFormat df = new SimpleDateFormat(
                                        "yyyy-MM-dd hh:mm:ss.SSS");
GregorianCalendar cld1 = new GregorianCalendar();
                                                  // current date time
try {
   Date d = df.parse("2014-13-36 36:65:82.976");
   String s = df.format(d); // "2015-02-06 13:06:22.976"
   cld1.setTime(d);
} catch (ParseException e) {}
int year = cld1.get(Calendar.YEAR);             // 2015
int month = cld1.get(Calendar.MONTH);           // 02
boolean b = month == Calendar.JANUARY;          // false
int day = cld1.get(Calendar.DAY_OF_MONTH);      // 02
int dayw = cld1.get(Calendar.DAY_OF_WEEK);      // 06
b = dayw == Calendar.FRIDAY;                    // true
```

Date operators (2/2)

```
int hour = cld1.get(Calendar.HOUR);            // 04
int minute = cld1.get(Calendar.MINUTE);        // 06
int second = cld1.get(Calendar.SECOND);        // 22
int milisec = cld1.get(Calendar.MILLISECOND); // 976

GregorianCalendar cld2 = (GregorianCalendar)cld1.clone();
cld2.add(Calendar.YEAR, -1);
                            // same operator for other fields too
year = cld2.get(Calendar.YEAR);                // 2014
b = cld1.after(cld2);                          // true
b = cld1.before(cld2);                         // false
```

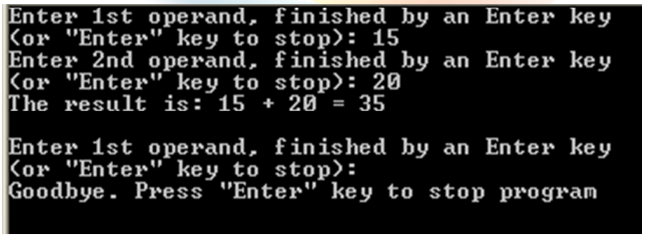
MVC PATTERN

MVC Pattern

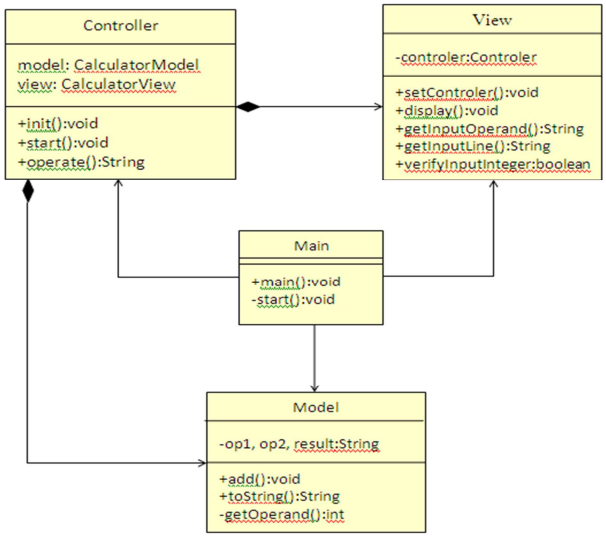


© FPT Software

MVC Pattern Example



MVC Pattern Example class diagram



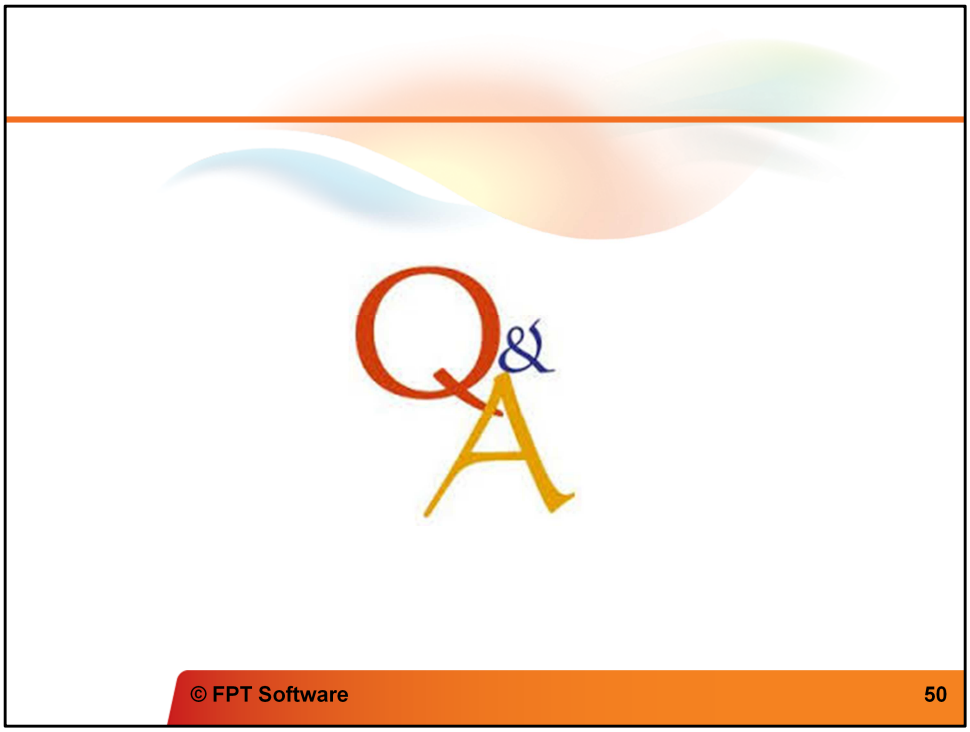

```
package com.fpt.hwa.example;

public class Main {

	/**
	 * @param args
	 */
	public static void main(String[] args) {

	}
	private void start(){


	}
}
**********
package com.fpt.hwa.example;

public class Model {

	private int op1, op2, result;
	public void add(String op1, String op2){


	}
	public String toString(){

	}
	private int getOperand(String inputLine){

		int operand =0;
		try{

		}catch (NumberFormatException e){}
		return operand;

	}
}
**********
package com.fpt.hwa.example;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class View {

	private Controller controller;
	public void setControler(Controller controler){

	}
	public void display(){


		BufferedReader reader = new BufferedReader(

		boolean stop = false;
		String op1 = "", op2 = "", result = "";
		while(!stop){


		}
		System.out.println("Goodbye. Press \"Enter\" key to stop program");
		try{

		}catch (IOException e) {}
	}
	private String getInput(String opPrompt, BufferedReader reader){

		String inputLine = "";
		boolean stop = false, inputOK = false;
		while(!inputOK && !stop){


		}
		if (stop) return "";
		return inputLine;
	}
	private String getInputLine(String opPrompt, BufferedReader reader){

		System.out.print("Enter " + opPrompt +

		String inputLine = "";
		try{

		}catch(IOException e){}
		return inputLine.trim();
	}
	private boolean verifyInputInteger(String inputLine){

		try{

		}catch (NumberFormatException e){

		}
		return true;

	}
}
**********
package com.fpt.hwa.example;

public class Controler {

	private Model model;
	private View view;
	public Controler(Model model, View view){


	}
	public void start(){

	}
	public String operate(String op1, String op2){


	}

}
**********
```

```
new Main().start();

Model model = new Model();
View view = new View();
Controller controler =new Controler(model, view);
controler.start();

this.op1 =getOperand(op1);
this.op2 =getOperand(op2);
result = this.op1 + this.op2;

return String.format("%d + %d =%d", op1, op2, result);

operand =Integer.parseInt(inputLine);

this.controler = controler;

new InputStreamReader(System.in));

op1 =getInputOperand("1st", reader);
stop =op1.isEmpty();
if (!stop){

}
if (!stop){

}

op2 =getInputOperand("2nd", reader);
stop =op2.isEmpty();

result = controler.operate(op1, op2);
System.out.println("The result is: " +result + "\n");

op1 =reader.readLine();

InputLine =getInput(opPrompt, reader);
stop =inputLine.isEmpty();
if (!stop){

}

inputOK =verifyInputInteger(inputLine);

" operand, finished by \"Enter\"  (only\n" +
"(or \"Enter\" key only to stop)  ");

inputLine =reader.readLine();

Integer.parseInt(inputLine);

System.out.println("Wrong integer number format");
return false;

this.model = model;
this.view = view;
view.setControler(this);

view.display();

model.add(op1, op2);
return model.toString();
```

Lesson summary

- More about class: Overload, constant, static…
- Inheritance: extension of base class
- Polymorphism: feature hiding/overriding
- Abstract class: at least one abstract method
- Interface: only abstract method
- Value type/Referenced type
- String class: Common operators
- Date/Time operators: using Calendar class
- MVC pattern: separation of concern

© FPT Software


Q&A
© FPT Software
50