AP CSA
writing class II

zhang si 张思

zhangsi@rdfz.cn

ICC 609

Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*

Local variables

- Local variables can be declared inside a method
- initialize before using local variables
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists
- Any method in the class can refer to instance variables

Find errors

- initialize before using local variables

```
public class test
{
    // 实例变量 - 用你自己的变量替换下面的例子

    public static void main(String[] args)
    {
        // 在这里加入你的代码
        //int i=0;
        int b=5;
        String a;
        String c="happy";
        for (int i=0;i<b;i++){
                a=c.substring(i,i+1);
        }
        System.out.println(a);
    }
}
```

可能尚未初始化变量a

local variable & instance variable

difference	instance variable	local variable
scope	class scope	method scope
extend	extends from the opening brace to the closing brace of the class definition	extends from the point where it is declared to the end of the block in which its declaration occurs
initialization	if you don‘t initialize an instance variable ， the compiler provides reasonable default values for primitive variables (0 for numbers, false for booleans） and provides null for reference	If you fail to initialize a local variable in a method before you use it, you will get a compile-time error.
access modifier	public:can be access by arbitrary instance of this class private: can only be access in this class file	can’t be declared as private or public
others		Local variables take precedence over instance variables with the same name. (You should avoid usingthe same name,or use this keyword)

Quiz

Consider the following class declaration.

```
public class Circle
{
   private double radius;
   public double computeArea()
   {
      private double pi = 3.14159;
      public double area = pi * radius * radius;
      return area;
   }
   // Constructor not shown.
}
```

Which of the following best explains why the `computeArea` method will cause a compilation error?

(A) Local variables declared inside a method cannot be declared as `public` or `private`.

(B) Local variables declared inside a method must all be `private`.

(C) Local variables declared inside a method must all be `public`.

(D) Local variables used inside a method must be declared at the end of the method.

(E) Local variables used inside a method must be declared before the method header.

Comments

- Ignored by compilers/interpreters
- Help make code more readable
- Prevent execution when testing alternative code
- Not required on AP Exam FRQs but are an important habit
- Helpful in reading FRQs
- Software development—team effort
  - communicate among programmers
  - maintain the code for years

Comments

- There are three types of comments

```
// Single-line comment

/* Multi-line
     comment */

  /** Documentation
    * comment
    * to create Javadoc
    */
```

Javadoc Comment

```
/**
 * This program is an abstract representation of a Snack
 * Created 10/1/2020
 * @author  Sandy Czajka
 * @version  1.0
 */
public class Snack {

        private String name;
        private int numCalories;

        // Constructors not shown

        /** Method to set value of calories
         * @param num – value to update numCalories
         */
        public void setNumCalories(int num) {
                numCalories = num;
        }
}
```

Here we see documentation comments.

Java has a tool called Javadoc that comes with the Java JDK and will pull out all of these comments to make documentation of the class in the form of a web page.

There are tags that can be used in Javadoc.

Search for “Javadoc” to get more information about these comments and tags.

Preconditions and Postconditions

- A *precondition* is a condition that should be true when a method is called
- A *postcondition* is a condition that should be true when a method finishes executing
- These conditions are expressed in comments above the method header
- Both preconditions and postconditions are a kind of *assertion*, a logical statement that can be true or false which represents a programmer´s assumptions about a program

Quiz

Consider the following method, which is intended to return the product of 3 and the nonnegative difference between its two int parameters.

```
public int threeTimesDiff (int num1, int num2)

{

     return 3 * (num1 - num2);

}
```

Which, if any, precondition is required so that the method works as intended for all values of the parameters that satisfy the precondition?

(A) num1 > 0 , num2 > 0
(B) num1 >= 0 , num2 >= 0
(C) num1 >= num2
(D) num2 >= num1
(E) No precondition is required.

ESSENTIAL KNOWLEDGE

- Comments are ignored by the compiler and are not executed when the program is run.
- Three types of comments in Java include /* */, which generates a block of comments, //, which generates a comment on one line, and /** */, which are Javadoc comments and are used to create API documentation.
- A precondition is a condition that must be true just prior to the execution of a section of program code in order for the method to behave as expected. There is no expectation that the method will check to ensure preconditions are satisfied.
- A postcondition is a condition that must always be true after the execution of a section of program code. Postconditions describe the outcome of the execution in terms of what is being returned or the state of an object.
- Programmers write method code to satisfy the postconditions when preconditions are met.

the toString method

- The **default** `toString` method is declared in the Object class, it will transform the parameters into `String` type.

- when you call the `System.out.print` or `System.out.println`, the `toString` method is called also.

- If System.out.print or System.out.println is passed an **`object`**, it will printed the reference,

- EG (student1 is an object of Student class, it has attribute: name and score1 and score2):

```
System.out.println("Student 1: " + student1);
```

results(return the reference):

```
Student 1: Student@58f2d2e9
```

the toString method

- **How to print the attribute of the object?**
- The toString method is an **overridden** method that is included in classes to provide a description of a specific object. It generally includes what values are stored in the instance data of the object.
- If System.out.print or System.out.println is passed an **`object`**, that object's `toString` method is called, and the returned string is printed.*(if you didn't override the toString method ,the default one will call ,and you'll get the reference of the object printed)*
- EG (in the Student class, we override the toString method):

```
public String toString(){
    return "Name: "+name+" Test1: "+score1+" Test2: "+score2;
}
```

- then when you print student1:

```
System.out.println("Student 1: " + student1);
```

- you'll get the result:

```
Student 1: Name: Mary Test1: 100.0 Test2: 90.0
```

the equals method

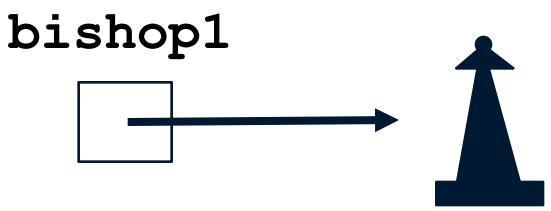
- The == operator compares object references for equality, returning `true` if the references are aliases of each other
    - `bishop1 == bishop2`
- A method called `equals` is defined for all objects, but unless we redefine it when we write a class, it has the same semantics as the == operator
    - `bishop1.equals(bishop2)`
- We can redefine the `equals` method to return `true` under whatever conditions we think are appropriate

References

- Recall from Chapter 2 that 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();
```



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");
```

The null Reference

- An object reference variable declared at the class level (an instance variable) is automatically initialized to null

- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used

- Attempting to follow a null reference causes a `NullPointerException` to be thrown

- Usually a compiler will check to see if a local variable is being used without being initialized

The this Reference

- Within a non-static method or a constructor, the keyword `this` is a reference to the current object—the object whose method or constructor is being called.
- The keyword `this` can be used to pass the current object as an actual parameter in a method call.

```
public class Person
{
   // instance variables
   private String name;
   private String email;
   private String phoneNumber;

   // constructor: construct a Person copying in the data into the instance variab
   public Person(String name, String email, String phone)
   {
      this.name = name;
      this.email = email;
      this.phoneNumber = phone;
   }
```

The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`
- If `tryMe` is invoked as follows, the `this` reference refers to `obj1`:
  - `obj1.tryMe();`
- But in this case, the `this` reference refers to `obj2`:
  - `obj2.tryMe();`

Quiz

```
public class MenuItem
{
   private double price;
   public MenuItem(double p)
   {
      price = p;
   }
   public double getPrice()
   {
      return price;
   }
   public void makeItAMeal()
   {
      Combo meal = new Combo(this);
      price = meal.getComboPrice();
   }
}
```

```
public class Combo
{
   private double comboPrice;
   public Combo(MenuItem item)
   {
      comboPrice = item.getPrice() + 1.5;
   }
   public double getComboPrice()
   {
      return comboPrice;
   }
}
```

The following code segment appears in a class other than `MenuItem` or `Combo`.

```
MenuItem one = new MenuItem(5.0);
one.makeItAMeal();
System.out.println(one.getPrice());
```

What, if anything, is printed as a result of executing the code segment?

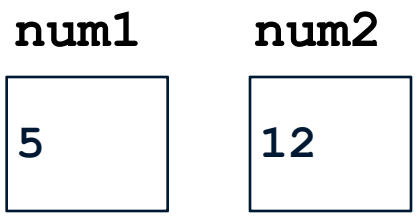
Assignment Revisited

- The act of assignment takes a copy of a value and stores it in a variable

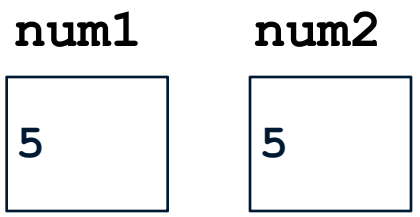
- For primitive types:

```
num2 = num1;
```

Before



After

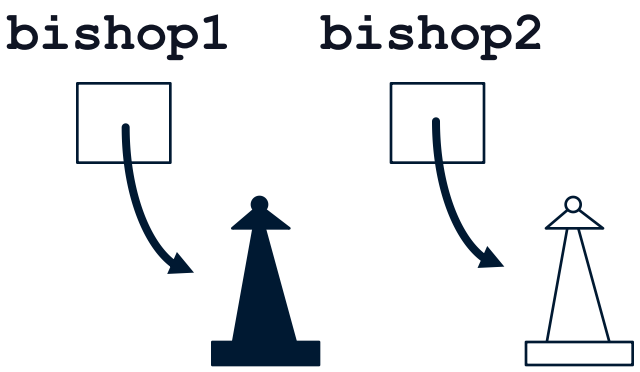


Reference Assignment

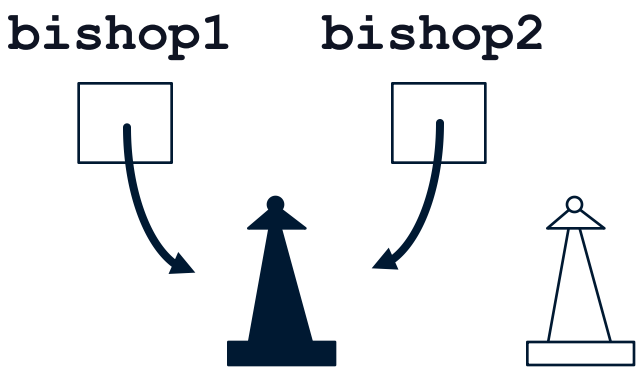
- For object references, assignment copies the memory location:

```
bishop2 = bishop1;
```

Before



After



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- One object (and its data) can be accessed using different reference variables
- Aliases can be useful, but should be managed carefully
- Changing the object's state (its variables) through one reference changes it for all of its aliases

Quiz

Consider the following class declaration

```
public class SomeClass{

	private int num;

	public SomeClass(int n){
		num = n;
	}

	public void increment(int more) {
		num = num + more;
	}
	public int getNum()  {
		return num;
	}

}
```

The following code segment appears in another class.

```
SomeClass one = new SomeClass(100);
SomeClass two = new SomeClass(100);
SomeClass three = one;
one.increment(200);

System.out.println(one.getNum() + "
" + two.getNum() + " " +
three.getNum());
```

What is printed as a result of executing the code segment?

Objects as Parameters

- Parameters in a Java method are *passed by value*
- This means that a copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- Passing parameters is therefore similar to an **assignment statement**
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

primitive data as parameter

```
public class test {
    public static void main(String[] args) {
        int i = 1;
        System.out.println("before change, i = "+i);
        change(i);
        System.out.println("after change, i = "+i);
    }
    public static void change(int i){
        i = 5;
    }
}
```

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:

```
before change, i = 1
after change, i = 1
```

Passing Objects to Methods

- What you do with a parameter inside a method may or may not have a permanent effect (outside the method)

- See ParameterTester.java
- See ParameterModifier.java
- See Num.java

- Note the difference between changing the reference and changing the object that the reference points to

```
public class test {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello
");
        System.out.println("before change, sb is
"+sb.toString());
        change(sb);
        System.out.println("after change, sb is
"+sb.toString());
    }
    public static void change(StringBuffer
stringBuffer){
        stringBuffer.append("world !");
    }
}
```

Passing Objects to Methods

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

so the result is:

```
before change, sb is Hello
after change, sb is Hello world !
```

```
public class test {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello
");
        System.out.println("before change, sb is
"+sb.toString());
        change(sb);
        System.out.println("after change, sb is
"+sb.toString());
    }
    public static void change(StringBuffer
stringBuffer){
        stringBuffer = new StringBuffer("Hi ");
        stringBuffer.append("world !");
    }
}
```

passing objects to methods

for more information :
https://www.cnblogs.com/huanghuanghui/p/9386047.html

we construct a new object stringBuffer, so it is not aliases with the former one. that's why afterchange , the state of sb is not change.

```
before change, sb is Hello
after change, sb is Hello
```

Quiz2

Consider the following method.

```
public int pick(boolean test, int x, int y)
{
      if (test)
      return x;
      else
      return y;
}
```

What value is returned by the following method call?

```
pick(false, pick(true, 0, 1), pick(true, 6, 7))
```

The static Modifier

- “static” is a reserved word in Java and indicates that a variable or method is attached to a class instead of an object.
- EG: a static variable may count how many objects of a specific class get created or may hold a value which is true for all objects.
- A static method cannot access non-static instance variables (because all of those instance variables are attached to objects).

The static Modifier

- In Chapter 2 we discussed static methods (also called class methods) that can be invoked through the class name rather than through a particular object
- For example, the methods of the `Math` class are static:
  - `Math.sqrt (25)`
- To write a static method, we apply the `static` modifier to the method definition
- The `static` modifier can be applied to variables as well
- It associates a variable or method with the class rather than with an object

The static Modifier

- Static variables are also called *class variables*
- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists
- ```
  private static float price;
  ```
- Memory space for a static variable is created when the class in which it is declared is loaded
- All objects created from the class share static variables
- The most common use of static variables is for constants
  ```

The static Modifier

```
class Helper
```

```
public static int triple (int num)
{
   int result;
   result = num * 3;
   return result;
}
```

Because it is static, the method can be invoked as:

```
value = Helper.triple (5);
```

The static Modifier

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static; it is invoked by the system without creating an object
- Static methods cannot reference instance variables, because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

The static Modifier

- Static methods and static variables often work together
- See CountInstances.java
- See Slogan.java

instance variable VS static variable

difference	instance variable	static variable
scope	class scope	class scope
attached to	object	class
values	for different object in this class, the instance variable may assign different values	all the object in this class share the same value

Quiz

Consider the following class definition.

```
public class Something
{
   private static int count = 0;
   public Something()
   {
      count += 5;
   }
   public static void increment()
   {
      count++;
   }
}
```

The following code segment appears in a method in a class other than `Something`.

```
Something s = new Something();
Something.increment();
```

(A) The code segment does not compile because the `increment` method should be called on an object of the class `Something`, not on the class itself.

(B) The code segment creates a `Something` object `s`. The class `Something`'s static variable `count` is initially `0`, then increased by `1`.

(C) The code segment creates a `Something` object `s`. The class `Something`'s static variable `count` is initially `0`, then increased by `5`, then increased by `1`.

(D) The code segment creates a `Something` object `s`. After executing the code segment, the object `s` has a `count` value of `1`.

(E) The code segment creates a `Something` object `s`. After executing the code segment, the object `s` has a `count` value of `5`.

Quiz

Consider the following class definition.

```
public class Element
{
   public static int max_value = 0;
   private int value;
   public Element (int v)
   {
      value = v;
      if (value > max_value)
      {
         max_value = value;
      }
   }
}
```

The following code segment appears in a class other than `Element`.

```
for (int i = 0;  i < 5;  i++)
{
   int k = (int) (Math.random() * 10 + 1);
   if (k >= Element.max_value)
   {
      Element e = new Element(k);
   }
}
```

Which of the following best describes the behavior of the code segment?

(A) Exactly 5 `Element` objects are created.

(B) Exactly 10 `Element` objects are created.

(C) Between 0 and 5 `Element` objects are created, and `Element.max_value` is increased only for the first object created.

(D) Between 1 and 5 `Element` objects are created, and `Element.max_value` is increased for every object created.

(E) Between 1 and 5 `Element` objects are created, and `Element.max_value` is increased for at least one object created.

ESSENTIAL KNOWLEDGE

- Static methods are associated with the class, not objects of the class.
- Static methods include the keyword staticin the header before the method name.
- Static methods cannot access or change the values of instance variables.
- Static methods can access or change the values of static variables.
- Static methods do not have a this reference and are unable to use the class’ s instance variables or call non-static methods.
- Static variables belong to the class, with all objects of a class sharing a single static variable.
- Static variables can be designated as either public or private and are designated with the static keyword before the variable type.
- Static variables are used with the class name and the dot operator, since they are associated with a class, not objects of a class.