Object-Oriented Programming: Java

Yi-ting Chiang

Department of Applied Mathematics
Chung Yuan Christian University

November 28, 2022

Class Declaration in Java

Each class you create in Java, C++, and Python becomes a new data type that can be used to declare variables and create objects. This example demonstrates how to do it in Java:

```
public class Account {
    private String name; // instance variable
    // method to set the name in the object
    public void setName(String name) {
        this.name = name; // store the name
    }
    // method to retrieve the name from the object
    public String getName() {
        return name; // return value of name to caller
    }
} // end class Account
```

Account.java[1]

Declare a new class in Java:

- Every class declaration contains keyword class followed immediately by the class's name
- Each class declaration that begins with the access modifier public must be stored in a file that has the same name as the class and ends with the .java filename extension

[1] Example from: Java How To Program, Late Objects, 10th Ed., 2015

Class Declaration in Java

```
public class Account {
    private String name; // instance variable
    // method to set the name in the object
    public void setName(String name) {
        this.name = name; // store the name
    }
    // method to retrieve the name from the object
    public String getName() {
        return name; // return value of name to caller
    }
} // end class Account
```

Listing:

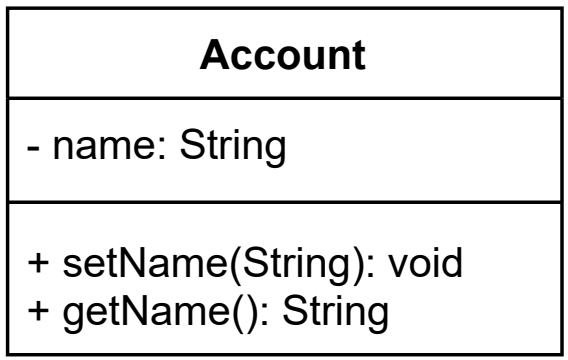

Figure: The class diagram of class Account

Class Declaration in Java

About instance variables in Java:

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime
- Instance variables on an object exist before methods are called, while the methods are executing, and after the methods complete execution
- Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations.
- Each object (instance) of the class has its own copy of each of the class's instance variables
- Most instance-variable declarations are preceded with the keyword private, which is an access modifier
  - Private variables and methods are accessible only to methods of the class in which they are declared
  - Using keyword private to encapsulate the instance variables, and allow the clients to access them by using only the public methods

Class Declaration in Java

About the instance method setName:

```
public class Account {
    private String name; // instance variable
    // method to set the name in the object
    public void setName(String name) {
        this.name = name; // store the name
    }
    // method to retrieve the name from the object
    public String getName() {
        return name; // return value of name to caller
    }
} // end class Account
```

- setName is an instance method in Account
- setName receives parameter name of type String, which represents the name that will be passed to the method as an argument
- The string variable "name" is a local variable fo setName
  - The instance variable "name" is shadowed in setName
  - To use this variable in setName, explicitly specify the keyword this to refer to the shadowed variable

Class Declaration in Java

About the instance method getName:

```
public class Account {
    private String name; // instance variable
    // method to set the name in the object
    public void setName(String name) {
        this.name = name; // store the name
    }
    // method to retrieve the name from the object
    public String getName() {
        return name; // return value of name to caller
    }
} // end class Account
```

- getName is also an instance method in Account
- getName returns the instance variable "name" to the caller
- getName does not require any parameter to do its job

Create and Use Instances

The following class uses Account to perform its task:

```
import java.util.Scanner;
public class AccountTest {
    public static void main(String[] args) {
        // create a Scanner object to obtain input from the command window
        Scanner input = new Scanner(System.in);
        // create an Account object and assign it to myAccount
        Account myAccount = new Account();
        System.out.printf("Initial name is: %s%n%n",
                                myAccount.getName());
        // prompt for and read name
        System.out.println("Please enter the name:");
        String theName = input.nextLine(); // read a line of text
        myAccount.setName(theName); // put theName in myAccount
        System.out.println(); // outputs a blank line
        // display the name stored in object myAccount
        System.out.printf("Name in object myAccount is:%n%s%n",
            myAccount.getName());
    }
} // end class AccountTest
```

AccountTest.java[2]

[2]Example from: Java How To Program, Late Objects, 10th Ed., 2015

Create and Use Instances

The result of running AccountTest is:

```
Initial name is: null
Please enter the name:
John Smith
Name in object myAccount is:
John Smith
```

From the example:

- In Java, use keyword new to create a new instance of a class
- Local variables are not automatically initialized
  - Accessing uninitialized local variable (including primitive data types) results in compilation error
- Instance variables are automatically initialized to a default initial value
- The default value depends on its data type. For reference-type variables such as String, it is "null"
  - The default value of primitive numeric data types (int, char, double, etc) is 0, and boolean is false

Create and Use Instances

About AccountTest:

```
import java.util.Scanner;
public class AccountTest {
    public static void main(String[] args) {
        // create a Scanner object to obtain input from the command window
        Scanner input = new Scanner(System.in);
        // create an Account object and assign it to myAccount
        Account myAccount = new Account();
        System.out.printf("Initial name is: %s%n%n", myAccount.getName());
        // prompt for and read name
        System.out.println("Please enter the name:");
        String theName = input.nextLine(); // read a line of text
        myAccount.setName(theName); // put theName in myAccount
        System.out.println(); // outputs a blank line
        // display the name stored in object myAccount
        System.out.printf("Name in object myAccount is:%n%s%n", myAccount.getName());
    }
} // end class AccountTest
```

- Use keyword new to create an instance of a class. Then you can get the reference of the object of this class
  - The constructor of the class will be called
  - The created object can be used as a variable
- To call the instance methods in an object, use the object name followed by a dot, and then specify the method's name
- main() is declared as static. We will explain it later

Create and Use Instances

We mentioned that the private members in a class are encapsulated. The clients can only use the interface (public members) to access these members. Directly accessing private members in Java causes compilation error:

```
public class AccountTestError {
    public static void main(String[] args) {
        Account myAccount = new Account();
        System.out.printf("Initial name is: %s%n%n",
                          myAccount.getName());
        // directly access the private member
        myAccount.name = "John Smith";
        System.out.printf("Name in object myAccount is:%n%s%n",
            myAccount.getName());
    }
}
```

Trying to compile the code above will get the following error message:

```
AccountTestError.java:7: error: name has private access in Account
      myAccount.name = "John Smith";
                ^
1 error
```

Class Constructor

A class constructor is used to initialize an object's instance variables when the object is created. See the following example

```
public class Account {
    private String name; // instance variable
    public Account(String name) { // the constructor
        this.name = name;
    }
    // method to set the name in the object
    public void setName(String name) {
        this.name = name; // store the name
    }
    // method to retrieve the name from the object
    public String getName() {
        return name; // return value of name to caller
    }
} // end class Account
```

Define the class with a constructor[3]

[3] Example from: Java How To Program, Late Objects, 10th Ed., 2015

Class Constructor

The following example creates two instances of the new Account class, using its constructor to initialize them.

```
public class AccountTest {
   public static void main(String[] args) {
      // create two Account objects
      Account account1 = new Account("John Smith");
      Account account2 = new Account("Bill Gates");
      System.out.printf("account1 has name: %s%n",
                        account1.getName());
      System.out.printf("account2 has name: %s%n",
                        account2.getName());
   }
} // end class AccountTest
```

4

[4] Example from: Java How To Program, Late Objects, 10th Ed., 2015

Class Constructor

The class constructor in Java

- The name of the class constructor has to be the same with the class's name
- Constructors can specify parameters but cannot return values
- Each Java class must have at least one class constructor
  - A class without its own constructor will be provided with a default constructor, which has no parameters, to initialize its instance variables
  - A class with any constructor will not have the default constructor
    - So a compilation error will occur if a class has a constructor with any parameter but create an instance without giving any parameter

Class Constructor

Class constructor can do more things than merely initialize instance variables. Here is an example[5]:

```
public class Account {
    private String name;
    private double balance; // record the balance
    public Account(String name, double balance) {
        this.name = name;
        if ( balance > 0 ) this.balance = balance;
    }
    public void setName(String name) {
        this.name = name; // store the name
    }
    public void deposit(double depositAmount) {
        if ( depositAmount > 0.0) { // check validity
            balance += depositAmount;
        }
    }
    public double getBalance() {
        return balance; //return the current balance
    }
    public String getName() {
        return name;
    }
}
```

Notice that besides to initialize the name and the balance, the constructor also checks if the initial balance value is valid (a positive value)

[5] Example from: Java How To Program, Late Objects, 10th Ed., 2015

Class Constructor

```
public class AccountTest {
   public static void main(String[] args) {
      Account account1 = new Account("John Smith", 3.0);
      Account account2 = new Account("Bill Gates", 1000.8);
      System.out.printf("%s has balance %.2f%n",
                        account1.getName(), account1.getBalance());
      System.out.printf("%s has balance %.2f%n",
                        account2.getName(), account2.getBalance());
      account1.deposit(-1.5); // invalid deposit
      System.out.printf("%s has balance %.2f after invalid deposit %n",
                        account1.getName(), account1.getBalance());
      System.out.printf("%s has balance %.2f%n",
                        account2.getName(), account2.getBalance());
      account2.deposit(562.7); // valid deposit
      System.out.printf("%s has balance %.2f%n",
                        account1.getName(), account1.getBalance());
      System.out.printf("%s has balance %.2f after valid deposit%n",
                        account2.getName(), account2.getBalance());
   }
}
```

The output of the program above is:

```
John Smith has balance 3.00
Bill Gates has balance 1000.80
John Smith has balance 3.00 after invalid deposit
Bill Gates has balance 1000.80
John Smith has balance 3.00
Bill Gates has balance 1563.50 after valid deposit
```

Throw Exceptions

The following example demonstrates how to throw an exception to indicate that a problem occurs[6]:

```
public class Time1 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59
   public void setTime(int hour, int minute, int second)
   {
      // validate hour, minute and second
      if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
         second < 0 || second >= 60) {
         // throw an exception for invalid arguments!
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
      }
      this.hour = hour; this.minute = minute; this.second = second;
   }
   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString() {
      return String.format("%02d:%02d:%02d", hour, minute, second);
   }
   // convert to String in standard-time format (H:MM:SS AM or PM)
   public String toString() {
      return String.format("%d:%02d:%02d %s",((hour == 0 || hour == 12) ? 12 : hour % 12),
         minute, second, (hour < 12 ? "AM" : "PM"));
   }
} // end class Time1
```

[6] Example from: Java How To Program, Late Objects, 10th Ed., 2015

Throw Exceptions

The setTime method checks if the time is valid:

```
public void setTime(int hour, int minute, int second)
{
    // validate hour, minute and second
    if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
        second < 0 || second >= 60) {
        // throw an exception for invalid arguments!
        throw new IllegalArgumentException(
            "hour, minute and/or second was out of range");
    }
    this.hour = hour;
    this.minute = minute;
    this.second = second;
}
```

- The if statement checks and throws an IllegalArgumentException if any of the arguments is illegal
  - The following codes will not be executed
- The string given to IllegalArgumentException is the custom error message set by the programmer

Throw Exceptions

The code below shows how to catch an exception:

```
public class Time1Test {
   public static void main(String[] args) {
      Time1 time = new Time1(); // invokes Time1 constructor
      displayTime("After time object is created", time);
      System.out.println();
      // change time and output updated time
      time.setTime(13, 27, 6);
      displayTime("After calling setTime", time);
      System.out.println();
      try {
         time.setTime(99, 99, 99); // values out of range
      }
      catch (IllegalArgumentException e) {
         System.out.printf("Exception: %s%n%n", e.getMessage());
      }
      // display time after attempt to set invalid values
      displayTime("After calling setTime with invalid values", time);
   }
   // displays a Time1 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time1 t) {
      System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
         header, t.toUniversalString(), t.toString());
   }
} // end class Time1Test
```

- Use try-catch statement to catch exceptions. The braces following try indicates the statements to check exception
- One try can have more than one catch to deal with different kinds of exceptions
- If an exception occurs, the program executes the codes in the corresponding catch's braces

Overloaded Constructors

A class in Java can have more than one constructors, which enable objects of a class to be initialized in different ways. This is called overloaded constructors[7]:

```
public class Time2 {
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   public Time2() {
      this(0, 0, 0); // invoke constructor with three arguments
   }
   public Time2(int hour) {
      this(hour, 0, 0); // invoke constructor with three arguments
   }
   public Time2(int hour, int minute) {
      this(hour, minute, 0); // invoke constructor with three arguments
   }
   public Time2(int hour, int minute, int second) {
      setTime(hour, minute, second);
   }
   public Time2(Time2 time) {
      this(time.hour, time.minute, time.second); // can access its own private members
   }
```

There are five constructors in class Time2. When a client creates a object of Time2, the constructor with the corresponding signature (numbers and the types of the parameters) will be used to initialize the object.

[7] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Overloaded Constructors

The set and get methods of class Time2 are given below[8]:

```
public void setTime(int hour, int minute, int second) {
   setHour(hour); setMinute(minute); setSecond(second);
}

public void setHour(int hour) {
   if (hour < 0 || hour >= 24) throw new IllegalArgumentException("hour must be 0-23");
   this.hour = hour;
}
public void setMinute(int minute) {
   if (minute < 0 || minute >= 60)
      throw new IllegalArgumentException("minute must be 0-59");
   this.minute = minute;
}
public void setSecond(int second) {
   if (second < 0 || second >= 60)
      throw new IllegalArgumentException("second must be 0-59");
    this.second = second;
}
public int getHour() { return hour; }
public int getMinute() { return minute; }
public int getSecond() { return second; }
```

[8]Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Overloaded Constructors

Finally, the public methods of Time2 which can be used to print the time are[9]:

```
    public String toUniversalString() {
        return String.format(
            "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
    }

    public String toString() {
        return String.format("%d:%02d:%02d %s",
            ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
            getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
    }
} // end class Time2
```

[9] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Overloaded Constructors

Notice that in this example, four of the five constructors calls another constructor to initialize the instance variables:

```
public class Time2 {
    private int hour; // 0 - 23
    private int minute; // 0 - 59
    private int second; // 0 - 59

    public Time2() {
        this(0, 0, 0); // invoke constructor with three arguments
    }
    public Time2(int hour) {
        this(hour, 0, 0); // invoke constructor with three arguments
    }
    public Time2(int hour, int minute) {
        this(hour, minute, 0); // invoke constructor with three arguments
    }
    public Time2(int hour, int minute, int second) {
        setTime(hour, minute, second);
    }
    public Time2(Time2 time) {
        this(time.hour, time.minute, time.second); // can access its own private members
    }
```

In Java, constructors can call other constructor by calling to this. Calling to this must be the first statement in the constructor. Otherwise, there will be compilation error. Here are the pros and cons:

- It is less efficient
- Less code to modify if the internal structure changes (for example, use one instead three integers to represent the time)

Overloaded Constructors

The following example shows how the objects of Time2 can be created by using different constructors[10]:

```
public class Time2Test {
   public static void main(String[] args) {
      Time2 t1 = new Time2(); // 00:00:00
      Time2 t2 = new Time2(2); // 02:00:00
      Time2 t3 = new Time2(21, 34); // 21:34:00
      Time2 t4 = new Time2(12, 25, 42); // 12:25:42
      Time2 t5 = new Time2(t4); // 12:25:42
      displayTime("t1: all default arguments", t1);
      displayTime("t2: hour specified; default minute and second", t2);
      displayTime("t3: hour and minute specified; default second", t3);
      displayTime("t4: hour, minute and second specified", t4);
      displayTime("t5: Time2 object t4 specified", t5);
      // attempt to initialize t6 with invalid values
      try {
         Time2 t6 = new Time2(27, 74, 99); // invalid values
      }
      catch (IllegalArgumentException e) {
         System.out.printf("%nException while initializing t6: %s%n", e.getMessage());
      }
   }
   // displays a Time2 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time2 t) {
      System.out.printf("%s%n   %s%n   %s%n", header, t.toUniversalString(), t.toString());
   }
} // end class Time2Test
```

[10] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Overloaded Constructors

```
public class Time2Test {
   public static void main(String[] args) {
      Time2 t1 = new Time2(); // 00:00:00
      Time2 t2 = new Time2(2); // 02:00:00
      Time2 t3 = new Time2(21, 34); // 21:34:00
      Time2 t4 = new Time2(12, 25, 42); // 12:25:42
      Time2 t5 = new Time2(t4); // 12:25:42
      displayTime("t1: all default arguments", t1);
      displayTime("t2: hour specified; default minute and second", t2);
      displayTime("t3: hour and minute specified; default second", t3);
      displayTime("t4: hour, minute and second specified", t4);
      displayTime("t5: Time2 object t4 specified", t5);
      // attempt to initialize t6 with invalid values
      try {
         Time2 t6 = new Time2(27, 74, 99); // invalid values
      }
      catch (IllegalArgumentException e) {
         System.out.printf("%nException while initializing t6: %s%n", e.getMessage());
      }
   }
   // displays a Time2 object in 24-hour and 12-hour formats
   private static void displayTime(String header, Time2 t) {
      System.out.printf("%s%n   %s%n   %s%n", header, t.toUniversalString(), t.toString());
   }
} // end class Time2Test
```

Notice that the method displayTime() in class Time2Test is defined as private. A private member function can only be called by another member functions in the same class.

Overloaded Constructors

The output of running the example is:

```
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM
Exception while initializing t6: hour must be 0-23
```

Aggregation

Recall that (shared) aggregation is a relationship between classes that one class is part of another, and this class can exist by itself (e.g. we can destroy a car but keep its wheels).

In Java, we can define a class that has references to objects of other classes as its members.

Aggregation

```
public class Date {
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year
   private static final int[] daysPerMonth =
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
   // constructor: confirm proper value for month and day given the year
   public Date(int month, int day, int year) {
      if (month<=0 || month > 12) throw new IllegalArgumentException(
         "month (" + month + ") must be 1-12");
      if (day<=0 || (day>daysPerMonth[month] && !(month==2 && day==29)))
         throw new IllegalArgumentException("day (" + day +
         ") out-of-range for the specified month and year");
      if (month==2 && day==29 && !(year%400==0 || (year%4==0 && year%100!= 0)))
         throw new IllegalArgumentException("day (" + day +
         ") out-of-range for the specified month and year");
      this.month = month; this.day = day; this.year = year;
      System.out.printf("Date object constructor for date %s%n", this);
   }
   // return a String of the form month/day/year
   public String toString() {
      return String.format("%d/%d/%d", month, day, year);
   }
} // end class Date
```

In this example, a class Date is defined[11]. Note that print this will automatically invoke the method toString() in Java.

[11] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Aggregation

```
public class Employee {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Date hireDate;
    // constructor to initialize name, birth date and hire date
    public Employee(String firstName, String lastName,
                    Date birthDate, Date hireDate) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthDate = birthDate;
        this.hireDate = hireDate;
    }
    // convert Employee to String format
    public String toString() {
        return String.format("%s, %s  Hired: %s  Birthday: %s",
            lastName, firstName, hireDate, birthDate);
    }
} // end class Employee
```

The class Employee has two Date and two String variables. Note that these variables exist before an Employee object is created. Class Employee just gets their reference.[12].

[12] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Aggregation

```
public class EmployeeTest {
    public static void main(String[] args) {
        Date birth = new Date(7, 24, 1949);
        Date hire = new Date(3, 12, 1988);
        Employee employee = new Employee("Bob", "Blue", birth, hire);

        System.out.println(employee);
    }
} // end class EmployeeTest
```

This Java code shows how to create an instance of class Employee. Two instances of Date is created and then used as the parameters to create the object employee.[13] The result of running this example is:

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Note that this example asks println() to print an object. In this case, method toSting() of this class will automatically invoked.

[13] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Java static Class Members

It sometimes may be necessary to record class-wide information in the application. For example, to remember how many instances a specific kind of class has been created. In Java, you can define a static member in a class. All instances of this class will have the same copy of the static member.
If the static member is a variable, the compile will assign it a default value if it is not initialized when it is declared.
In addition to access via objects of the class, static members can also be directly access from the class name. See the following example:

```
public class TestMath {
    public static void main(String args[]) {
        System.out.printf("%.8f%n",Math.PI);
        System.out.printf("%.8f%n",Math.E);
        System.out.printf("%.4f%n",Math.pow(2.0,0.5));
    }
}
```

Java API provides a Math class, which defines many static math functions and constants. These math functions can be directly called without creating any instances of class Math.

To design an operation whose output is independent of the state of the object, create a static member function.

Java static Class Members

In Java, it is possible to "import" static members defined in other classes then use these static members without mentioning the class. Here is an example:

```
import static java.lang.Math.*;
public class TestFinal {
    public static void main(String args[]) {
        System.out.printf("%.8f%n",PI);
        System.out.printf("%.8f%n",E);
        System.out.printf("%.4f%n",pow(2.0,0.5));
    }
}
```

The output is the same with the previous one.

Java static Class Members

```
public class Employee {
    private static int count = 0; // number of Employees created
    private String firstName;
    private String lastName;
    // initialize Employee, add 1 to static count and
    // output String indicating that constructor was called
    public Employee(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
        ++count;   // increment static count of employees
        System.out.printf("Employee constructor: %s %s; count = %d%n",
            firstName, lastName, count);
    }
    // get first name
    public String getFirstName() { return firstName; }
    // get last name
    public String getLastName() { return lastName; }
    // static method to get static count value
    public static int getCount() { return count; }
} // end class Employee
```

In the example above, a static member count is defined[14]. Note that in the constructor, the value of count is increased by one.

[14] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Java static Class Members

```
public class EmployeeTest {
   public static void main(String[] args) {
      // show that count is 0 before creating Employees
      System.out.printf("Employees before instantiation: %d%n",
         Employee.getCount());

      // create two Employees; count should be 2
      Employee e1 = new Employee("Susan", "Baker");
      Employee e2 = new Employee("Bob", "Blue");

      // show that count is 2 after creating two Employees
      System.out.printf("%nEmployees after instantiation:%n");
      System.out.printf("via e1.getCount(): %d%n", e1.getCount());
      System.out.printf("via e2.getCount(): %d%n", e2.getCount());
      System.out.printf("via Employee.getCount(): %d%n", Employee.getCount());

      // get names of Employees
      System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
         e1.getFirstName(), e1.getLastName(),
         e2.getFirstName(), e2.getLastName());
   }
} // end class EmployeeTest
```

This example demonstrates how the value of the static member "count" in class Employee changes[15].

[15] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Java static Class Members

The result of running the example code in the previous side is:

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2
Employee 1: Susan Baker
Employee 2: Bob Blue
```

The result shows

- Line 1: The value of count is 0
- Line 2: The instance Susan Baker is created and the constructor reports count=1
- Line 3: The instance Bob Blue is created and the constructor reports count=2
- Following lines: Both e1 and e2's static member count=2. The class also reports count=2
    - e1, e2, and the class itself share the same copy of this static member

Inheritance

In Java, a class can inherit from another class by using the keyword "extend". See the following example:

```
public class Circle {
    public static final double PI = 3.14159;
    protected double r;
    public Circle(double r) { this.r = r; }
    public double circumference() { return 2*PI*r; }
    public double area() { return PI*r*r; }
    public double radius() { return r; }
}
```

In this example, a class of circle is defined. This class has a constructor, three public methods, a public static variables, and a protected[16] variable[17]

[16] Will explain later

[17] Example from: Java In A Nutshell, 6th Ed., 2014.

Inheritance

The class Circle can represent a general circle. However, to define "a cycle in a plane", more properties are required.
A circle in a plane is a kind of circle, but a circle is not necessary in a plane. Therefore, we can define a new class to inherit from circle[18]:

```
public class PlaneCircle extends Circle {
    private final double cx, cy;
    public PlaneCircle(double r, double x, double y) {
        super(r);
        this.cx = x; this.cy = y;
    }
    public double getCenterX() { return cx; }
    public double getCenterY() { return cy; }
    public boolean isInside(double x, double y) {
        double dx = x-cx, dy = y-cy;
        return Math.sqrt(dx*dx+dy*dy) < r;
    }
}
```

[18] Example modified from: Java In A Nutshell, 6th Ed., 2014.

Inheritance

The following example defines an object of PlaneCircle and invokes their methods:

```
public class PlaneCircleTest {
   public static void main(String args[]) {
      PlaneCircle pc = new PlaneCircle(1.0,5.0,3.0);
      System.out.printf("circumference is %.2f%n", pc.circumference());
      System.out.printf("area is %.2f%n", pc.area());
      System.out.printf("radius is %.2f%n", pc.radius());
      System.out.printf("directly access radius: %.2f%n", pc.r);
      System.out.printf("Center: (%.2f,%.2f)%n", pc.getCenterX(), pc.getCenterY());
      System.out.printf("point (4,2.3) is in the circle? %b%n", pc.isInside(4.2,3));
      System.out.printf("point (6,0) is in the circle? %b%n", pc.isInside(6,0));
   }
}
```

The output is:

```
circumference is 6.28
area is 3.14
radius is 1.00
directly access radius: 1.00
Center: (5.00,3.00)
point (4,2.3) is in the circle? true
point (6,0) is in the circle? false
```

Inheritance

Something about inheritance in Java:

- A subclass inherits all the public and protected[19] members from its superclass.
  - A superclass's public/protected members are also public/protected in its subclass
  - A subclass can use "super()" (may with parameters) to access superclass's constructor
- A superclass's private is hidden from any classes

[19] Protected members can only be accessed by that class, by its subclasses, and by the classes in the same package

Inheritance

In summary, a class member in Java has one of the following visibility modifiers:

- public
- protect
- private
- default (given no modifier)

Here is a summary of the visibility of the class members between the superclass and subclass[20]:

Situation	public	protected	default	private
The class itself	yes	yes	yes	yes
From subclass	yes	yes	no[21]	no
From non-subclass	yes	no[22]	no	no

Declaring protected variables allows the subclasses to modify the superclass's instance variable, which can cause problems and increase the cost to maintain the program. Therefore, use private modifier to encapsulate instance variables as far as possible!

[20] Assume they are in different packages

[21] yes, if in the same package

[22] yes, if in the same package

Inheritance: More about Class Constructor

In Java, the constructor of each subclass actually calls superclass's constructor. See the following example:

```
class A {
    protected int a;
    public A() { // invoked implicitly by B() (case c2)
        System.out.printf("A() talks a=%d!%n", a);
    }
    public A(int i) {  // invoked explicitly by B() (case c1)
        System.out.printf("A(int) talks a=%d!%n", a);
    }
}
class B extends A {
    public B() {  // invoked implicitly by C(int i) (case c2)
        System.out.printf("B() talks a=%d!%n", a);
    }
    public B(int i) {  // invoked explicitly by C() (case c1)
        super(i);
        System.out.printf("B(int) talks a=%d!%n", a);
        a = i;
    }
}
public class C extends B  {
    public C() {  // invoked by c1
      super(25);
      System.out.printf("C() talks a=%d!%n", a);
    }
    public C(int i) { // invoked by c2
      a = i;
      System.out.printf("C(int) talks a=%d!%n", a);
    }
}
```

Note in this example:

- C is a subclass of B, which is a subclass of A
- C() calls B's constructor B(int)
- C(int) does not call any superclass's constructor
- B(int) calls A's constructor A(int)
- B() does not call any superclass's constructor
- Instance variable a is set in C(int)
- Each constructor print the current value of a

Inheritance: More about Class Constructor

The following example shows the sequence the constructors in a class hierarchy is called.

```
public class ConstructorCall {
    public static void main(String[] args) {
        C c1 = new C();
        System.out.println();
        C c2 = new C(50);
    }
}
```

The output is:

```
A(int) talks a=0!
B(int) talks a=0!
C() talks a=25!

A() talks a=0!
B() talks a=0!
C(int) talks a=50!
```

From the output, we can find

- Each subclass constructor calls its superclass's constructor
  - Explicitly use keyword super to call (with or without parameters)
  - Implicitly calls super() (without parameter)
- The superclass's constructor executes before the subclass's constructor

Inheritance: Override and Polymorphism

Not all methods in a superclass are suitable for its subclass. For example, a real number can be considered as a special kind of imaginary number whose imaginary part is 0. Define a class of imaginary number as follows:

```
public class ImaginaryNumber {
    private double r;
    private double i;
    public ImaginaryNumber(double a, double b) {
        this.r = a;
        this.i = b;
    }
    public ImaginaryNumber add(double a, double b) {
        return new ImaginaryNumber(r+a,i+b);
    }
    public ImaginaryNumber multiply(ImaginaryNumber v) {
        return new ImaginaryNumber(r*v.r-i*v.i,r*v.i+i*v.r);
    }
    public double getRealPart() { return r; }
    public double getImaginaryPart() { return i; }
    public String toString() {
        return new String(r+"+"+i+"i");
    }
}
```

Inheritance: Override and Polymorphism

Here is the class for real numbers. Notice the @Override annotation:

```
public class RealNumber extends ImaginaryNumber {
    public RealNumber(double v) {
        super(v,0); // call superclass's constructor
    }
    public double add(RealNumber v) {
        return super.add(v.getRealPart(), // call superclass's method
                v.getImaginaryPart()).getRealPart();
    }
    public double multiply(RealNumber v) {
        return super.multiply(v).getRealPart();
    }
    @Override
    public String toString() {
        return String.valueOf(super.getRealPart());
    }
}
```

@Override, which is optional, tells the compiler that this method overrides the method with the same signature (parameters) in the superclass. If there is no such method in the superclass, a compilation error occurs. Notice that add() and multiply() have different signatures with those in the superclass. Therefore, they do not override superclass's methods.

In addition, This example also shows how to call superclass's constructor and methods.

Inheritance: Override and Polymorphism

```
public class NumberTest {
   public static void main(String[] arg) {
      RealNumber rv = new RealNumber(21.5);
      RealNumber rv2 = new RealNumber(2.0);
      ImaginaryNumber iv = new ImaginaryNumber(4.3, 9.51);
      System.out.printf("Imaginary Number: %s%n", iv.toString());
      System.out.printf("Real number 1: %s%n", rv.toString());
      System.out.printf("Real number 2: %s%n", rv2.toString());
      double v = rv.add(rv2);
      System.out.printf("result of addition: %f%n", v);
      v = rv.multiply(rv2);
      System.out.printf("result of multiplication:  %f%n", v);
   }
}
```

This example shows how the methods in the two classes are invoked. The output is:

```
Imaginary Number: 4.3+9.51i
Real number 1: 21.5
Real number 2: 2.0
result of addition: 23.500000
result of multiplication:  43.000000
```

Note that the method with the same name "toString" applying to the subclass (RealNumber) and the superclass (ImaginaryNumber) performs differently. This is polymorphism.

Polymorphism allows the subclass to extend the superclass's behaviour.

Inheritance: Abstract Class and Methods

It is possible to define a class that can only be used for other classes to inherit from.

- To define the common parts of subclasses and leave the detailed implementation to the subclasses
- Can only be a superclass
- Cannot be used to instantiate objects

Inheritance: Abstract Class and Methods

The following example demonstrates how to define an abstract class in Java[23]:

```
public abstract class Employee {
  private final String name;
  private final String ID;
  public Employee(String name, String ID) {
    this.name=name;
    this.ID=ID;
  }
  public String getName() { return name; }
  public String getID() { return ID; }
  @Override
  public String toString() {
    return String.format("%s%nID: %s",getName(), getID());
  }
  public abstract double earning(); // an abstract method
}
```

In this example,

- Employee is an Java abstract class. It is defined with the abstract keyword
- The method toString() has @Override annotation
  - Each class in Java by default is a subclass of Object, the root of the class hierarchy in Java. Class Object provides several methods. toString() is one of them
- An abstract method earning() is declared without implementation
  - Each subclass of Employee has to implement this method
  - An class with abstract methods in it must be an abstract class

[23] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Inheritance: Abstract Class and Methods

Using the abstract class Employee, we can define a class for salaried employee[24]:

```
public class SalariedEmployee extends Employee {
  private double weeklySalary;
  public SalariedEmployee(String name, String ID, double salary) {
    super(name,ID);
    setSalary(salary);
  }
  public void setSalary(double salary) { this.weeklySalary=salary; }
  public double getSalary() { return weeklySalary; }
  @Override
  public double earning() { return getSalary(); }
  @Override
  public String toString() {
    return String.format("Salaried employee %s%n%s=%.4f",
    super.toString(), "weekly salary", getSalary());
  }
}
```

SalariedEmployee overrides two Employee's methods. As earning has being implemented, SalariedEmployee is not an abstract class.

[24] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Inheritance: Abstract Class and Methods

Here we defined a class for hourly employee[25]:

```
public class HourlyEmployee extends Employee {
  private double wage;
  private double hours;
  public HourlyEmployee(String name, String ID, double wage, double hour) {
    super(name, ID);
    setWage(wage); setHour(hour);
  }
  public void setWage(double wage) { this.wage=wage; }
  public void setHour(double hour) {this.hours=hour; }
  public double getWage() { return wage; }
  public double getHour() { return hours; }
  @Override
  public double earning() {
    if(getHour() <= 40) return getWage()*getHour();
    else return 40*getWage() + (getHour()-40)*getWage()*1.5;
  }
  @Override
  public String toString() {
    return String.format("Hourly employee %s%n%s: %.4f; %s: %.4f",
      super.toString(), "hourly wage", getWage(),
      "hours worked", getHour());
  }
}
```

An hourly employee is paid an hourly wage for each hour worked. If this employee works more than 40 hours, he/she will get overtime pay. Therefore, the implementation of earning() is different.

[25] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Inheritance: Abstract Class and Methods

The following example make use of these classes:

```
public class EmployeeTest {
  public static void main(String[] args) {
    SalariedEmployee se = new SalariedEmployee("John Smith", "12345", 777.0);
    HourlyEmployee heA = new HourlyEmployee("Alice Price", "22428", 20.5, 45);
    HourlyEmployee heB = new HourlyEmployee("Bob Lawrance", "28147", 16.75, 40);
    Employee[] e = new Employee[3];
    e[0] = se; e[1] = heA; e[2] = heB;
    for (int i = 0; i < 3; i++ ) {
      System.out.printf("%s%n",e[i].toString());
      if ( e[i] instanceof HourlyEmployee)
        System.out.printf("weekly earned: %.4f%n",e[i].earning());
    }
  }
}
```

The output is:

```
Salaried employee John Smith
ID: 12345
weekly salary=777.0000
Hourly employee Alice Price
ID: 22428
hourly wage: 20.5000; hours worked: 45.0000
weekly earned: 973.7500
Hourly employee Bob Lawrance
ID: 28147
hourly wage: 16.7500; hours worked: 40.0000
weekly earned: 670.0000
```

From this example, we can find that:

- Each object calls the overridden methods corresponding to its class
- In Java, the superclass can be used to refer to objects of its subclasses
- The "instanceof" operator in Java can be used to check the datatype (class) a variance belongs to

About the Modifier "final"

Java provides a mechanism to protect some constant from being modified by accident. By declaring an instance variable using keyword final, any attempt to modify it causes error. Here is an example:

```
class Triangle {
    public final int sum_internal_angle ;
    private double[] side;
    public Triangle(double a, double b, double c) {
        if ( a+b <= c || a+c <= b || b+c <= a)
           throw new IllegalArgumentException("Illegal side values!");
        side = new double[3];
        side[0] = a; side[1] = b; side[2] = c;
        sum_internal_angle = 180;
    }
    public double getArea() {
       double s = 0;
       for ( double v : side) s += v;
       s /= 2;
       return Math.sqrt(s*(s-side[0])*(s-side[1])*(s-side[2]));
    }
}
public class TestFinal {
    public static void main(String[] args) {
      Triangle t = new Triangle(3,4,5);
      System.out.printf("Area of the triangle is %.4f%n",t.getArea());
      t.sum_internal_angle = 360; // this line causes error!
      System.out.printf("Sum of internal angles is %d%n", t.sum_internal_angle);
    }
}
```

Compiling this example will get the following error message: "error: cannot assign a value to final variable sum_internal_angle"

About the Modifier "final"

The modifier "final" can define an instance variable with the following properties:

- Initialization: Have to be initialized when declaration or in the class's constructor
- Modify: Cannot be modified after being initialized

We can also define final classes or final methods. Final classes/methods have the following property:

- Final class: Cannot have any subclass
- Final method: Cannot be overridden

Interface

Java does not allow multiple inheritance. Any subclass can only have one superclass. For the case that multiple inheritance is necessary when developing systems, Java provides a mechanism called interface.
Similar to abstract classes, an interface defines abstract methods for other classes to implement. However, there are more restrictions and rules when using Java interface:

- An interface can only has public abstract methods
  - Therefore, you don't need to declare its methods as public abstract
  - Therefore, you cannot implement any method in it[26]
- An interface can only has public static final variables
  - Therefore, you don't need to declare its variables as public static final
- An interface cannot be instantiated. So there is no constructor
- An interface can use keyword "extends" to inherit from more than one interfaces
- A class can use keyword "implements" to declare that it will implement all the abstract methods in the interface
- A class can "implements" more than one interfaces
  - A class who "implements" an interface but does not actually implement all the abstract methods must be declared as abstract class

[26] in Java 8, you can implement methods which are declared as "default" in an interface

Interface

The following example demonstrates how to define and use Java interface[27]:

```
public abstract class Shape {
   public abstract double area();
   public abstract double circumference();
}
class Rectangle extends Shape {
   private double w, h;
   public Rectangle(double w, double h) { this.w=w; this.h=h;}
   public double getWidth() { return w; }
   public double getHeight() { return h; }
   public void setWidth(double w) { this.w=w; }
   public void setHeight(double h) { this.h=h; }
   public double area() { return w*h; }
   public double circumference() { return 2*(w+h); }
}
```

Shape is an abstract class. Rectangle defines a simple class for a rectangle.

[27] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Interface

```
public interface Centered {
  void setCenter(double x, double y);
  double getCenterX();
  double getCenterY();
}
interface Positionable extends Centered {
  void setUpperRightCorner(double x, double y);
  double getUpperRightX();
  double getUpperRightY();
}
interface Rotatable extends Positionable {
  void rotateClockwise(); // 90 degree clockwise rotation
  void rotateCounterclockwise(); //90 degree counterclockwise rotation
}
interface Movable {
  int m=0;
  void move(double x, double y);
}
```

Here are four interfaces[28]. The interface Centered provides methods for any class which wants to set its location of center. Positionable extends Centered and enables a object to set the position of its upper-right corner. Along with center, an object can now fixed its position. Rotatable and Movable make it possible for an object to respectively rotate and move.

[28] Example modified from: Java How To Program, Late Objects, 10th Ed., 2015

Interface

We define a class for the abstraction of a rectangle on a plane:

```
public class PlaneRectangle extends Rectangle implements Centered{
  private double cx, cy;
  public PlaneRectangle(double w, double h,
         double x, double y) {
    super(w,h);
    setCenter(x,y);
  }
  public void setCenter(double x, double y) {
    this.cx=x; this.cy=y;
  }
  public double getCenterX() { return cx;}
  public double getCenterY() { return cy;}
}
```

Note that this class "extends" Rectangle and "implements" Centered. Because class Centered assigned the location for a class, the rectangle can locate its position in a plane.

Interface

Finally, we define a plane rectangle which can change its position by implementing Movable and Rotatable:

```
public class DynamicPlaneRectangle extends PlaneRectangle implements Movable, Rotatable {
  public DynamicPlaneRectangle(double w, double h,
         double x, double y) {
     super(w,h,x,y);
  }
  //implement interface Positionable
  public void setUpperRightCorner(double x, double y) {
    super.setCenter(x-getWidth()/2.0,y-getHeight()/2.0);
  }
  public double getUpperRightX() { return getCenterX()+getWidth()/2.0; }
  public double getUpperRightY() { return getCenterY()+getHeight()/2.0; }
  //implement interface Rotatable
  public void rotateClockwise() {
    double tmp = getWidth();
    setWidth(getHeight());
    setHeight(tmp);
  }
  public void rotateCounterclockwise() {
    rotateClockwise();
  }
  // implement interface Movable
  public void move(double x, double y) {
    setCenter(getCenterX()+x,getCenterY()+y);
  }
}
```

Now we demonstrate how to use this class.

Interface

```
public class PlaneRectangleTest {
  public static void main(String[] args) {
    PlaneRectangle pr = new PlaneRectangle(5,7,1,2);
    DynamicPlaneRectangle dpr = new DynamicPlaneRectangle(
                                        8,6,3,4);
    System.out.println("PlaneRectangle:");
    System.out.printf("\tarea: %.4f%n\tcircumference: %.4f%n",
                      pr.area(), pr.circumference());
    System.out.printf("\tcenter: (%.2f,%.2f)%n",
                      pr.getCenterX(), pr.getCenterY());
    System.out.println("DynamicPlaneRectangle:");
    System.out.printf("\tarea: %.4f%n\tcircumference: %.4f%n",
                      dpr.area(), dpr.circumference());
    System.out.printf("\tcenter: (%.2f,%.2f)%n",
                      dpr.getCenterX(), dpr.getCenterY());
    System.out.printf("\tUpperRightCorner: (%.2f,%.2f)%n",
      dpr.getUpperRightX(), dpr.getUpperRightY());
    dpr.rotateClockwise();          // perform clockwise rotation
    System.out.println("  After clockwise rotation:");
    System.out.printf("\tUpperRightCorner: (%.2f,%.2f)%n",
                      dpr.getUpperRightX(), dpr.getUpperRightY());
    dpr.rotateCounterclockwise();// perform counterclockwise rotation
    System.out.println("  After Counterclockwise rotation:");
    System.out.printf("\tUpperRightCorner: (%.2f,%.2f)%n",
                      dpr.getUpperRightX(), dpr.getUpperRightY());
    dpr.move(-5,-2);
    System.out.println("  After moving (-5,-2):");
    System.out.printf("\tcenter: (%.2f,%.2f)%n",
                      dpr.getCenterX(), dpr.getCenterY());
    System.out.printf("\tUpperRightCorner: (%.2f,%.2f)%n",
      dpr.getUpperRightX(), dpr.getUpperRightY());
  }
}
```

In this example, two objects of rectangle on the plane are defined.

The variable pr represents a rectangle on the plane. We can ask this object to report its area and circumference.

The variable dpr is an instance of class DynamicPlaneRectangle. This class implements Movable and Rotatable, therefore, dpr can perform more tasks than pr.

The output of the example is:

```
PlaneRectangle:
        area: 35.0000
        circumference: 24.0000
        center: (1.00,2.00)
DynamicPlaneRectangle:
        area: 48.0000
        circumference: 28.0000
        center: (3.00,4.00)
        UpperRightCorner: (7.00,7.00)
  After clockwise rotation:
        UpperRightCorner: (6.00,8.00)
  After Counterclockwise rotation:
        UpperRightCorner: (7.00,7.00)
  After moving (-5,-2):
        center: (-2.00,2.00)
        UpperRightCorner: (2.00,5.00)
```

Recall that mixins is a class that provides only operations to augment the behaviour of other classes. Java interface is the mechanism for the developers to realize mixins.

Interface

What we do not coverred in Java class includes[?]

- Inner class: define classes inside a class
- Anonymous class: define a class (inside another class) without giving it a name
- Package: put related classes together