Java Fundamentals Part 4

关注微信公众号
享终身免费学习

东方瑞通® 终身学习 Founded in 1998

1. Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

Example

Method Overloading

东方瑞通® 终身学习 Founded in 1998

```
static int plusMethodInt(int x, int y) {
  return x + y;
}

static double plusMethodDouble(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethodInt(8, 5);
  double myNum2 = plusMethodDouble(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double`:

Example

Method Overloading

东方瑞通® 终身学习 Founded in 1998

```
static int plusMethod(int x, int y) {
  return x + y;
}

static double plusMethod(double x, double y) {
  return x + y;
}

public static void main(String[] args) {
  int myNum1 = plusMethod(8, 5);
  double myNum2 = plusMethod(4.3, 6.26);
  System.out.println("int: " + myNum1);
  System.out.println("double: " + myNum2);
}
```

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.

东方瑞通® 终身学习 Founded in 1998

2. Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

> The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Overriding

东方瑞通® 终身学习 Founded in 1998

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {

  public static void main(String args[]) {
    Animal a = new Animal();   // Animal reference and object
    Animal b = new Dog();   // Animal reference but Dog object

    a.move();   // runs the method in Animal class
    b.move();   // runs the method in Dog class
  }
}
```

This will produce the following result −

```
Animals can move
Dogs can walk and run
```

东方瑞通® 终身学习 Founded in 1998

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Consider the following example −

Overriding

东方瑞通® 终身学习 Founded in 1998

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
  public void bark() {
    System.out.println("Dogs can bark");
  }
}

public class TestDog {

  public static void main(String args[]) {
    Animal a = new Animal();   // Animal reference and object
    Animal b = new Dog();   // Animal reference but Dog object

    a.move();   // runs the method in Animal class
    b.move();   // runs the method in Dog class
    b.bark();
  }
}
```

东方瑞通® 终身学习 Founded in 1998

This will produce the following result −

```
TestDog.java:26: error: cannot find symbol
    b.bark();
     ^
  symbol:   method bark()
  location: variable b of type Animal
1 error
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

2.1. Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

2.2. Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    super.move();   // invokes the super class method
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {

  public static void main(String args[]) {
    Animal b = new Dog();   // Animal reference but Dog object
    b.move();   // runs the method in Dog class
  }
}
```

This will produce the following result −

Overriding

东方瑞通® 终身学习 Founded in 1998

Animals can move

Dogs can walk and run

3. Java Polymorphism

Polymorphism means “many forms”, and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Java Polymorphism

东方瑞通® 终身学习 Founded in 1998

Remember from the Inheritance chapter that we use the `extends` keyword to inherit from a class.

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
```

东方瑞通® 终身学习 Founded in 1998

```
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

东方瑞通® 终身学习 Founded in 1998

4. Java Abstraction

东方瑞通® 终身学习 Founded in 1998

4.1. Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let’s convert the Animal class we used in the Polymorphism chapter to an abstract class:

东方瑞通® 终身学习 Founded in 1998

Remember from the Inheritance chapter that we use the `extends` keyword to inherit from a class.

Example

Java Abstraction

东方瑞通® 终身学习 Founded in 1998

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

东方瑞通® 终身学习 Founded in 1998

5. Interfaces

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "**abstract class**" that is used to group related methods with empty bodies:

```
// interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void run(); // interface method (does not have a body)
}
```

东方瑞通® 终身学习 Founded in 1998

To access the interface methods, the interface must be “implemented” (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the “implement” class:

Interfaces

东方瑞通® 终身学习 Founded in 1998

```
// Interface
interface Animal {
  public void animalSound(); // interface method (does not have a body)
  public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

东方瑞通® 终身学习 Founded in 1998

5.1. Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

东方瑞通® 终身学习 Founded in 1998

5.2. Why And When To Use Interfaces?

- To achieve security - hide certain details and only show the important details of an object (interface).
- Java does not support “multiple inheritance” (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

东方瑞通® 终身学习 Founded in 1998

5.3. Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Interfaces

东方瑞通® 终身学习 Founded in 1998

```
interface FirstInterface {
  public void myMethod(); // interface method
}

interface SecondInterface {
  public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Thank you

全国统一咨询热线：400-690-6115

北京|上海|广州|深圳|天津|成都|重庆|武汉|济南|青岛|杭州|西安

easthome.com