多态通过分离“做什么”和“怎么做”，从另一角度将接口和实现分离开来。多态不但能够改善代码的组织结构和可读性，还能够创建“可扩展的”程序，即无论在项目最初创建时，还是在需要添加新功能时，都可以进行扩充。
更容易理解。而多态的作用则是消除类型之间的耦合关系多态方法调用允许一种类型表现出与其他相似类型之间的区别，只要它们都是从同一基类导出而来的。这种区别是根据方法行为的不同来而表示出来的，虽然这些方法都可以通过同一个基类来调用。
多态的意思是“多种形式”，多种形式的意思是可以出现不同的计算类型，并且在运行的时候动态的确定正确的计算。
多态是指多个方法使用同一个名字有多种解释，当使用这个名字去调用方法时，系统将选择重载自动的选择其中的一个方法。在多态中只关心一个对象做什么，而不关心如何去做。
“多态”意味着“不同的形式”。在面向对象的程序设计中，我们持有相同的外观（基类的通用接口）以及使用该外观的不同形式：不同版本的动态绑定方法。
如果不运用数据抽象和继承，就不可能去理解，进而也不可能创建一个多态例子。多态是一种不能单独来看待的特性（例如，像 switch 语句是可以的），相反它只能作为类关系“全景”中的一部分，与其它特性协同工作。
人们经常被Java语言中其他的非面向对象的特性所困扰，比如方法重载等，人们有时会被认为这些是面向对象的特性。但是不要被愚弄：如果不是后期绑定，就不是多态
(1)class Object is not abstract
(2)Some methods in class Object can be overrided but some cannot, because they are marked final.
(3)The Object class servers two main purpose: to act as a polymorphic type for methods that need to work on any class AnD provide real method code that every class needs at runtime
JAVA make sure you are calling a method on an object that is actually capable of reponsing
类型检查
必须按照方法形参的类型来检查消息的参数类型，必须按消息所期待的类型来检查方法的返回类型。
绑定
绑定在程序中的意思是把一个标志符与一个存储单元或者把一个标志符与一个变量，方法建立起联系。
动态绑定
在运行时，根据对象的类型进行绑定
• The compiler decides whether you can call a method based on the referencetype, not the actual object type.
• The compiler checks the class of the reference type– not the object type – to see if you can call a method using that reference.
• The JVM decides which actual method is called based on the actual object type.
An object contains everything it inherits from each of its superclasses.• That means every object – regardless of its actual class type – is also an instance of class Object.

① Arguments must be the same, and return types must be compatible.

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type, or a subclass type. Remember, a subclass object is guaranteed to be able to do anything its superclass declares, so it's safe to return a subclass where the superclass is expected.

② The method can't be less accessible.

That means the access level must be the same, or friendlier. That means you can't, for example, override a public method and make it private. What a shock that would be to the code invoking what it *thinks* (at compile time) is a public method, if suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!

The return types can be different.

You're free to change the return types in overloaded methods, as long as the argument lists are different.

You can't change ONLY the return type.

If only the return type is different, it's not a valid over*load*—the compiler will assume you're trying to over*ride* the method. And even *that* won't be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you *can* change the return type to anything.

You *can* vary the access levels in any direction.

You're free to overload a method with a method that's more restrictive. It doesn't matter, since the new method isn't obligated to fulfill the contract of the overloaded method.

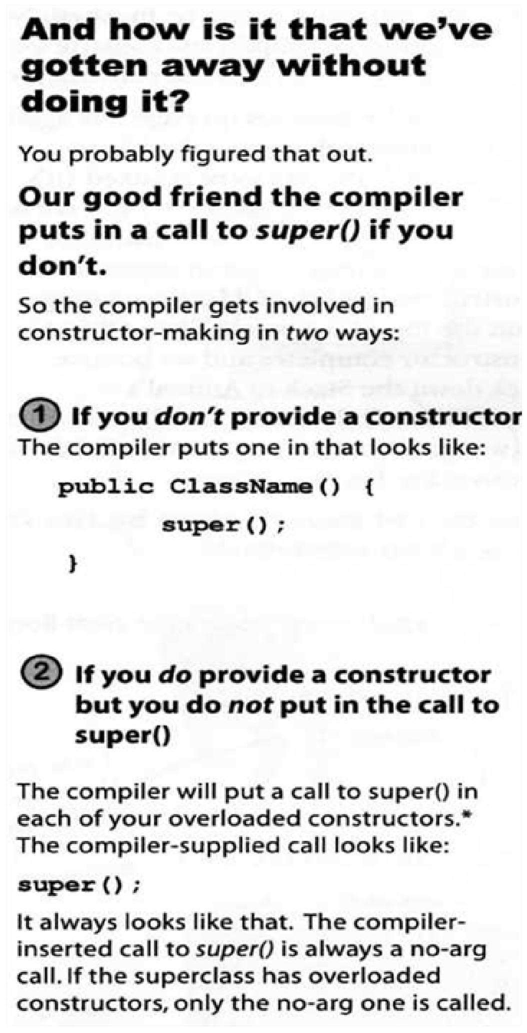
Legal examples of method overloading:

```
public class Overloads {

   String uniqueID;

   public int addNums(int a, int b) {
      return a + b;
   }

   public double addNums(double a, double b) {
       return a + b;
   }

   public void setUniqueID(String theID) {
      // lots of validation code, and then:
      uniqueID = theID;
   }

   public void setUniqueID(int ssNumber) {
      String numString = "" + ssNumber;
      setUniqueID(numString);
   }
}
```

```
public class JavaApplication3 {

    private void f() {
        System.out.println("private f()");
    }

    public static void main(String[] args) {
        JavaApplication3 po = new Derived();   .//Result-->private()
        po.f();                                              //private void f() is invisible in the subclass
                                   //the type decide what to do
    }
}

class Derived extends JavaApplication3 {

   public void f() {
        System.out.println("public f()");
    }
}
/*OUTPUT:
Class BaseL public callFoo
Class Base: private foo()
*/
public class Test1 {

    Child child;

    public static void main(String[] args) {
        Child child = new Child();
        child.callFoo();

    }
}

class Child extends Base {

    private void foo() {
        System.out.println("private foo()in Class Child");
    }
}

class Base {
  public void callFoo() {
    System.out.println("Class BaseL public callFoo");
    foo();
  }

  private void foo() {
    System.out.println("Class Base: private foo()");;
  }
}
```
组合技术通常用于你想要在新类中使用现有类的功能而非它的接口的情形。即，你在新类中嵌入某个对象，借其实现你所需要的功能，但新类的用户看到的只是你为新类所定义的接口，而非嵌入对象的接口。为取得此效果，你需要在新类中嵌入一个 private 的现有类的对象。
有时，允许类的用户直接访问新类中的组合成份是极具意义的；也就是说，将成员对象声明为public。如果成员对象自身都实现了具体实现的隐藏，那么这种做法就是安全的。当用户能够了解到你在组装一组部件时，会使得端口更加易于理解。
Car对象即为一个好例子：
```
//: c06:Car.java
// Composition with public objects.

class Engine {
 public void start() {}
 public void rev() {}
 public void stop() {}
}

class Wheel {
 public void inflate(int psi) {}
}

class Window {
 public void rollup() {}
 public void rolldown() {}
}

class Door {
 public Window window = new Window();
 public void open() {}
 public void close() {}
}
public class Car {
 public Engine engine = new Engine();
 public Wheel[] wheel = new Wheel[4];
 public Door
```

```
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

Constructor Chaining



And how is it that we've gotten away without doing it?

You probably figured that out.

Our good friend the compiler puts in a call to *super()* if you don't.

So the compiler gets involved in constructor-making in *two* ways:

① If you *don't* provide a constructor

The compiler puts one in that looks like:

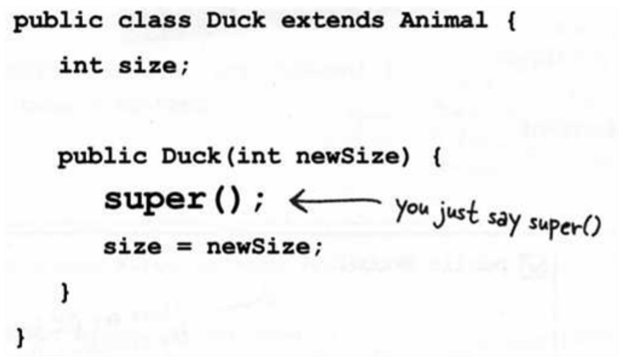
```
public ClassName() {
        super();
  }
```

② If you *do* provide a constructor but you do *not* put in the call to super()

The compiler will put a call to super() in each of your overloaded constructors.* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to *super()* is always a no-arg call. If the superclass has overloaded constructors, only the no-arg one is called.

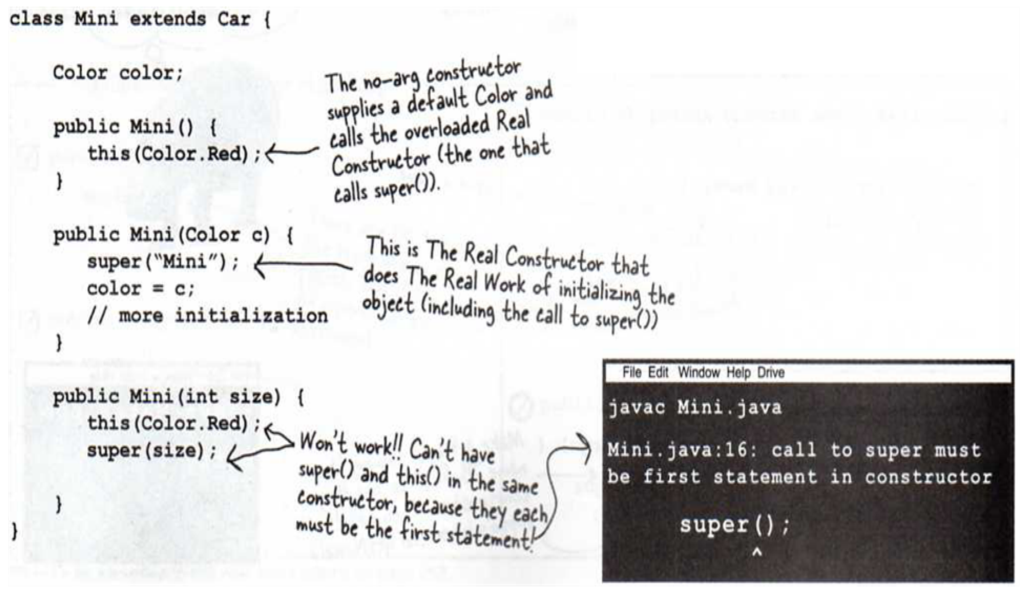


```
public class Duck extends Animal {
   int size;

   public Duck(int newSize) {
      super();
      size = newSize;
   }
}
```

you just say super()

The superclass parts of an objects have to be fully-formed (completely build) before the subclass parts can be constructed.

The call to super() must be the first statement in each constructor.



```
class Mini extends Car {

    Color color;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // more initialization
    }

    public Mini(int size) {
        this(Color.Red);
        super(size);

    }
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls super()).

This is The Real Constructor that does The Real Work of initializing the object (including the call to super())

Won't work!! Can't have super() and this() in the same constructor, because they each must be the first statement!

```
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor

     super();
       ^
```

Initialization with inheritance

Access Main(), load base class, Load until the root base class

Static Initialization in the root base class then the next derived class, and so on

All the primitives and the object reference are set to null

The base-class constructor will be called

The instance variables are initialized in textual order

The rest of the body of the constructor is executed

```
class Characteristic {
  private String s;
  Characteristic(String s) {
    this.s = s;
    System.out.println("Creating Characteristic " + s);
  }
  protected void dispose() {
    System.out.println("finalizing Characteristic " + s);
  }
}

class Description {
  private String s;
  Description(String s) {
    this.s = s;
    System.out.println("Creating Description " + s);
  }
  protected void dispose() {
    System.out.println("finalizing Description " + s);
  }
}
```

```
class LivingCreature {
  private Characteristic p = new Characteristic("is alive");
  private Description t =
    new Description("Basic Living Creature");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void dispose() {
    System.out.println("LivingCreature dispose");
    t.dispose();
    p.dispose();
  }
}
class Animal extends LivingCreature {
  private Characteristic p= new Characteristic("has heart");
  private Description t =
    new Description("Animal not Vegetable");
  Animal() {
    System.out.println("Animal()");
  }
  protected void dispose() {
    System.out.println("Animal dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}
class Amphibian extends Animal {
  private Characteristic p =
    new Characteristic("can live in water");
  private Description t =
    new Description("Both water and land");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void dispose() {
    System.out.println("Amphibian dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}
public class Frog extends Amphibian {
  private static Test monitor = new Test();
  private Characteristic p = new Characteristic("Croaks");
  private Description t = new Description("Eats Bugs");
  public Frog() {
System.out.println("Frog()");
  }
  protected void dispose() {
    System.out.println("Frog dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    Frog frog = new Frog();
    System.out.println("Bye!");
    frog.dispose();
}
monitor.expect(new String[] {
      "Creating Characteristic is alive",
      "Creating Description Basic Living Creature",
      "LivingCreature()",
      "Creating Characteristic has heart",
      "Creating Description Animal not Vegetable",
      "Animal()",
      "Creating Characteristic can live in water",
      "Creating Description Both water and land",
      "Amphibian()",
      "Creating Characteristic Croaks",
      "Creating Description Eats Bugs",
      "Frog()",
      "Bye!",
      "Frog dispose",
      "finalizing Description Eats Bugs",
      "finalizing Characteristic Croaks",
      "Amphibian dispose",
      "finalizing Description Both water and land",
      "finalizing Characteristic can live in water",
      "Animal dispose",
      "finalizing Description Animal not Vegetable",
      "finalizing Characteristic has heart",
      "LivingCreature dispose",
      "finalizing Description Basic Living Creature",
      "finalizing Characteristic is alive"
    });
```

```
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
finalizing Description Eats Bugs
finalizing Characteristic Croaks
Amphibian dispose
finalizing Description Both water and land
finalizing Characteristic can live in water
Animal dispose
finalizing Description Animal not Vegetable
finalizing Characteristic has heart
LivingCreature dispose
finalizing Description Basic Living Creature
finalizing Characteristic is alive
```

What happens if you're inside a constructor and you call a dynamically-bound method of the object being constructed?

The overridden method will be called before the object is fully constructed.

What happens if you're inside a constructor and you call a dynamically-bound method of the object being constructed?

The overridden method will be called before the object is fully constructed.