```
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();
```

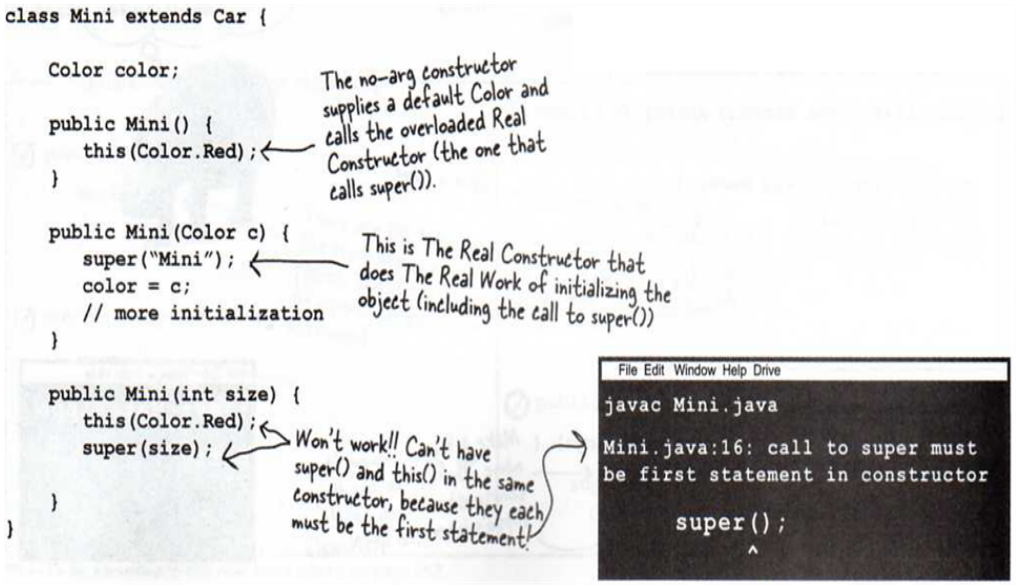
It always looks like that. The compiler-inserted call to *super()* is always a no-arg call. If the superclass has overloaded constructors, only the no-arg one is called.

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

    public Duck(int newSize) {
        super(); <-- you just say super()
        size = newSize;
    }
}
```

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.



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.