**Controlling Access to Members of a Class**

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

* At the top level—public, or *package-private* (no explicit modifier).
* At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

|  |
| --- |
| **Access Levels** |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| private | Y | N | N | N |

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

public abstract class HttpServlet {

..

protected abstract void doGet(HttpServletRequest req, HttpServletResponse
 resp);

}

public class HttpServer {

…

HttpServlet s = HttpServlet.getInstance(servlet);

if (s != null) {

ServletInvocation invocation = new ServletInvocation(parameters,out);

s.doGet(invocation,invocation);

}

|  |  |
| --- | --- |
| ] | Can an abstract class have a constructor?If so, how it can be used and for what purposes? |
|  |  |

## 5 Answers

|  |  |
| --- | --- |
|  | Consider this:abstract class Product {     int multiplyBy;    public Product( int multiplyBy ) {        this.multiplyBy = multiplyBy;    }    public int mutiply(int val) {       return muliplyBy \* val;    }}class TimesTwo extends Product {    public TimesTwo() {        super(2);    }}class TimesWhat extends Product {    public TimesWhat(int what) {        super(what);    }}The superclass Product is abstract and has a constructor. The concrete class TimesTwo has a default constructor that just hardcodes the value 2. The concrete class TimesWhat has a constructor that allows the caller to specify the value.NOTE: As there is no default (or no-arg) constructor in the parent abstract class the constructor used in subclasses must be specified.Abstract constructors will frequently be used to enforce class constraints or invariants such as the minimum fields required to setup the class. |

# [Is Constructor Overriding Possible?](http://stackoverflow.com/questions/5099924/is-constructor-overriding-possible)

|  |  |
| --- | --- |
|  | What i know is, the compiler writes a default no argument constructor in the byte code. But if we write it ourselves, that constructor is called automatically. Is this phenomena a constructor overriding? |
|  |  |

## 8 Answers

|  |  |
| --- | --- |
|  | What you describe isn't overriding. If you don't specify a default constructor, the compiler will create a default constructor. If it's a subclass, it will call the default parent constructor([super()](http://www.leepoint.net/notes-java/oop/constructors/constructor-super.html)), it will also initialize all instance variables to a default value determined by the type's default value(0 for numeric types, false for booleans, or null for objects).Overriding happens when a subclass has the same name, number/type of parameters, and the same return type as an instance method of the superclass. In this case, the subclass will override the superclass's method.   |
|  | Constructors are not normal methods and they cannot be "overridden". Saying that a constructor can be overridden would imply that a superclass constructor would be visible and could be called to create an instance of a subclass. This isn't true... a subclass doesn't have any constructors by default (except a no-arg constructor if the class it extends has one). It has to explicitly declare any other constructors, and those constructors belong to it and not to its superclass, even if they take the same parameters that the superclass constructors take.The stuff you mention about default no arg constructors is just an aspect of how constructors work and has nothing to do with overriding.Private constructors: |

Private constructors:

|  |  |
| --- | --- |
| 12345678910111213141516171819202122232425 | public class ClassA{   private int val;       public ClassA(int val, boolean dummy) {     this(val);   }       private ClassA(int val)   {      this.val = val;   }   public int getVal()   {      return val;   }} class ClassB extends ClassA{   public ClassB(int val)   {      super(val, true);   }} |

**NESTED CLASSES**

|  |  |
| --- | --- |
|  | Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called static nested classes. Non-static nested classes are called inner classes.Static nested classes are accessed using the enclosing class name:OuterClass.StaticNestedClassFor example, to create an object for the static nested class, use this syntax:OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();Objects that are instances of an inner class exist within an instance of the outer class. Consider the following classes:class OuterClass {    ...    class InnerClass {        ...    }}An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:OuterClass.InnerClass innerObject = outerObject.new InnerClass();see: [Java Tutorial - Nested Classes](http://download.oracle.com/javase/tutorial/java/javaOO/nested.html) |

|  |  |  |
| --- | --- | --- |
|  | The [Java tutorial says](http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html):Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called static nested classes. Non-static nested classes are called inner classes.In common parlance, the terms "nested" and "inner" are used interchangeably by most programmers, so I'll just use the term "inner class".Inner classes can be nested ad infinitum, e.g. class A can contain class B which contains class C which contains class D, etc. However, more than one level of class nesting is rare, as it is generally bad design.There are three reasons you might create an inner class:* organization: sometimes it seems most sensible to sort a class into the namespace of another class, especially when it won't be used in any other context
* access: inner classes have special access to the variables/fields of their containing classes (precisely which variables/fields depends on the kind of inner class).
* convenience: having to create a new file for every new type is bothersome, again, especially when the type will only be used in one context

There are **four kinds of inner class in Java**. In brief, they are:* **static inner class**: declared as a static member of another class
* **instance inner class**: declared as an instance member of another class
* **local inner class**: declared inside an instance method of another class
* **anonymous inner class**: like a local inner class, but written as an expression which returns a one-off object

In more detail:**static inner classes**Static inner classes are the easiest kind to understand because they have nothing to do with instances of the containing class.A static inner class is a class declared as a static member of another class. Just like other static members, such a class is really just a hanger on that uses the containing class as its namespace, e.g.the class Goat declared as a static member of class Rhino in the package pizza is known by the namepizza.Rhino.Goat.package pizza;public class Rhino {    ...    public static class Goat {        ...    }}Frankly, static inner classes are a pretty worthless feature because classes are already divided into namespaces by packages. The only real conceivable reason to create a static inner class is that such a class has access to its containing class's private static members, but I find this to be a pretty lame justification for the static inner class feature to exist.**instance inner classes**An instance inner class is a class declared as an instance member of another class:package pizza;public class Rhino {    public class Goat {        ...    }    private void jerry() {        Goat g = new Goat();    }}Like with a static inner class, the instance inner class is known as qualified by its containing class name,pizza.Rhino.Goat, but inside the containing class, it can be known by its simple name. However, every instance of an instance inner class is tied to a particular instance of its containing class: above, the Goatcreated in jerry, is implicitly tied to the Rhino instance this in jerry. Otherwise, we make the associatedRhino instance explicit when we instantiate Goat:Rhino rhino = new Rhino();Rhino.Goat goat = rhino.new Goat();(Notice you refer to the inner type as just Goat in the weird new syntax: Java infers the containing type from the rhino part. And, yes new rhino.Goat() would have made more sense to me too.)So what does this gain us? Well, the inner class instance has access to the instance members of the containing class instance. These enclosing instance members are referred to inside the inner class viajust their simple names, not via \*this\* (this in the inner class refers to the inner class instance, not the associated containing class instance):public class Rhino {    private String barry;    public class Goat {        public void colin() {            System.out.println(barry);        }    }}In the inner class, you can refer to this of the containing class as Rhino.this, and you can use this to refer to its members, e.g. Rhino.this.barry.**local inner classes**A local inner class is a class declared in the body of a method. Such a class is only known within its containing method, so it can only be instantiated and have its members accessed within its containing method. The gain is that a local inner class instance is tied to and can access the final local variables of its containing method. When the instance uses a final local of its containing method, the variable retains the value it held at the time of the instance's creation, even if the variable has gone out of scope (this is effectively Java's crude, limited version of closures).Because a local inner class is neither the member of a class or package, it is not declared with an access level. (Be clear, however, that its own members have access levels like in a normal class.)If a local inner class is declared in an instance method, an instantiation of the inner class is tied to the instance held by the containing method's this at the time of the instance's creation, and so the containing class's instance members are accessible like in an instance inner class. A local inner class is instantiated simply via its name, e.g. local inner class Cat is instantiated as new Cat(), not new this.Cat() as you might expect.**anonymous inner classes**An anonymous inner class is a syntactically convenient way of writing a local inner class. Most commonly, a local inner class is instantiated at most just once each time its containing method is run. It would be nice, then, if we could combine the local inner class definition and its single instantiation into one convenient syntax form, and it would also be nice if we didn't have to think up a name for the class (the fewer unhelpful names your code contains, the better). An anonymous inner class allows both these things:new \*ParentClassName\*(\*constructorArgs\*) {\*members\*}This is an expression returning a new instance of an unnamed class which extends ParentClassName. You cannot supply your own constructor; rather, one is implicitly supplied which simply calls the super constructor, so the arguments supplied must fit the super constructor. (If the parent contains multiple constructors, the “simplest” one is called, “simplest” as determined by a rather complex set of rules not worth bothering to learn in detail--just pay attention to what NetBeans or Eclipse tell you.)Alternatively, you can specify an interface to implement:new \*InterfaceName\*() {\*members\*}Such a declaration creates a new instance of an unnamed class which extends Object and implementsInterfaceName. Again, you cannot supply your own constructor; in this case, Java implicitly supplies a no-arg, do-nothing constructor (so there will never be constructor arguments in this case).Even though you can't give an anonymous inner class a constructor, you can still do any setup you want using an initializer block (a {} block placed outside any method).Be clear that an anonymous inner class is simply a less flexible way of creating a local inner class with one instance. If you want a local inner class which implements multiple interfaces or which implements interfaces while extending some class other than Object or which specifies its own constructor, you're stuck creating a regular named local inner class.

|  |
| --- |
|  |

 |
|  | I don't think the real difference became clear in the above answers.First to get the terms right:* A nested class is a class which is contained in another class at the source code level.
* It is static if you declare it with the **static** modifier.
* A non-static nested class is called inner class. (I stay with non-static nested class.)

Martin's answer is right so far. However, the actual question is: What is the purpose of declaring a nested class static or not?You use **static nested classes** if you just want to keep your classes together if they belong topically together or if the nested class is exclusively used in the enclosing class. There is no semantic difference between a static nested class and every other class.**Non-static nested classes** are a different beast. Similar to anonymous inner classes, such nested classes are actually closures. That means they capture their surrounding scope and their enclosing instance and make that accessible. Perhaps an example will clarify that. See this stub of a Container:public class Container {    public class Item{        Object data;        public Container getContainer(){                return Container.this;        }        public Item(Object data) {                super();                this.data = data;        }    }    public static Item create(Object data){        // does not compile since no instance of Container is available        return new Item(data);    }    public Item createSubItem(Object data){        // compiles, since 'this' Container is available        return new Item(data);    }}In this case you want to have a reference from a child item to the parent container. Using a non-static nested class, this works without some work. You can access the enclosing instance of Container with the syntax Container.this. |

# Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

**void** printCollection(Collection c) {

 Iterator i = c.iterator();

 **for** (k = 0; k < c.size(); k++) {

 System.out.println(i.next());

 }

}

And here is a naive attempt at writing it using generics (and the new for loop syntax):

**void** printCollection(Collection<Object> c) {

 **for** (Object e : c) {

 System.out.println(e);

 }

}

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes Collection<Object>, which, as we've just demonstrated, is **not** a supertype of all kinds of collections!

So what **is** the supertype of all kinds of collections? It's written Collection<?> (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a **wildcard type** for obvious reasons. We can write:

**void** printCollection(Collection<?> c) {

 **for** (Object e : c) {

 System.out.println(e);

 }

}

and now, we can call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them typeObject. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

Collection<?> c = new ArrayList<String>();

c.add(new Object()); // Compile time error

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

On the other hand, given a List<?>, we **can** call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

## Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

**public abstract class** Shape {

 **public abstract void** draw(Canvas c);

}

**public class** Circle **extends** Shape {

 **private int** x, y, radius;

 **public void** draw(Canvas c) {

 ...

 }

}

**public class** Rectangle **extends** Shape {

 **private int** x, y, width, height;

 **public void** draw(Canvas c) {

 ...

 }

}

These classes can be drawn on a canvas:

**public class** Canvas {

 **public void** draw(Shape s) {

 s.draw(**this**);

 }

}

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvasthat draws them all:

**public void** drawAll(List<Shape> shapes) {

 **for** (Shape s: shapes) {

 s.draw(**this**);

 }

}

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of **any** kind of shape:

**public void** drawAll(List<? **extends** Shape> shapes) {

 ...

}

There is a small but very important difference here: we have replaced the type List<Shape> with List<? **extends** Shape>. Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want.

List<? **extends** Shape> is an example of a *bounded wildcard*. The ? stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of Shape. (Note: It could be Shape itself, or some subclass; it need not literally extendShape.) We say that Shape is the *upper bound* of the wildcard.

There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into shapes in the body of the method. For instance, this is not allowed:

**public void** addRectangle(List<? **extends** Shape> shapes) {

 // *Compile-time error!*

 shapes.add(0, **new** Rectangle());

}

You should be able to figure out why the code above is disallowed. The type of the second parameter to shapes.add() is ? **extends** Shape-- an unknown subtype of Shape. Since we don't know what type it is, we don't know if it is a supertype of Rectangle; it might or might not be such a supertype, so it isn't safe to pass a Rectangle there.

|  |  |
| --- | --- |
|  | What is a raw type?The Java Language Specification defines a raw type as follows:[JLS 4.8 Raw Types](http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.8)A raw type is define to be either:* The name of a generic type declaration used without any accompanying actual type parameters.
* Any non-static type member of a raw type R that is not inherited from a superclass or superinterface of R.

Here's an example to illustrate:public class MyType<E> {    class Inner { }    static class Nested { }    public static void main(String[] args) {        MyType mt;          // warning: MyType is a raw type        MyType.Inner inn;   // warning: MyType.Inner is a raw type        MyType.Nested nest; // no warning: not parameterized type        MyType<Object> mt1; // no warning: type parameter given        MyType<?> mt2;      // no warning: type parameter given (wildcard OK!)    }}Here, MyType<E> is a parameterized type ([JLS 4.5](http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.5)). It is common to colloquially refer to this type as simply MyType for short, but technically the name is MyType<E>.mt has a raw type (and generates a compilation warning) by the first bullet point in the above definition;inn also has a raw type by the second bullet point.MyType.Nested is not a parameterized type, even though it's a member type of a parameterized typeMyType<E>, because it's static.mt1, and mt2 are both declared with actual type parameters, so they're not raw types.What's so special about raw types?Essentially, raw types behaves just like they were before generics were introduced. That is, the following is entirely legal at compile-time.List names = new ArrayList(); // warning: raw type!names.add("John");names.add("Mary");names.add(Boolean.FALSE); // not a compilation error!The above code runs just fine, but suppose you also have the following:for (Object o : names) {    String name = (String) o;    System.out.println(name);} // throws ClassCastException!  //    java.lang.Boolean cannot be cast to java.lang.StringNow we run into trouble at run-time, because names contains something that isn't an instanceof String.Presumably, if you want names to contain only String, you could perhaps still use a raw type andmanually check every add yourself, and then manually cast to String every item from names. **Even better**, though is NOT to use a raw type and let the compiler does all the work for you, harnessing the power of Java generics.List<String> names = new ArrayList<String>();names.add("John");names.add("Mary");names.add(Boolean.FALSE); // compilation error!Of course, if you DO want names to allow a Boolean, then you can declare it as List<Object> names, and the above code would compile.See also* [Java Tutorials/Generics](http://java.sun.com/docs/books/tutorial/extra/generics/index.html)

How's a raw type different from using <Object> as type parametersThe following is a quote from Effective Java 2nd Edition, Item 23: Don't use raw types in new code:Just what is the difference between the raw type List and the parameterized typeList<Object>? Loosely speaking, the former has opted out generic type checking, while the latter explicitly told the compiler that it is capable of holding objects of any type. While you can pass aList<String> to a parameter of type List, you can't pass it to a parameter of typeList<Object>. There are subtyping rules for generics, and List<String> is a subtype of the raw type List, but not of the parameterized type List<Object>. As a consequence, **you lose type safety if you use raw type like List, but not if you use a parameterized type likeList<Object>**.To illustrate the point, consider the following method which takes a List<Object> and appends a new Object().void appendNewObject(List<Object> list) {   list.add(new Object());}Generics in Java are invariant. A List<String> is not a List<Object>, so the following would generate a compiler warning:List<String> names = new ArrayList<String>();appendNewObject(names); // compilation error!If you had declared appendNewObject to take a raw type List as parameter, then this would compile, and you'd therefore lose the type safety that you get from generics.See also* [What is the difference between <E extends Number> and <Number>?](http://stackoverflow.com/questions/2770264/what-is-the-difference-between-e-extends-number-and-number/)
* [java generics (not) covariance](http://stackoverflow.com/questions/2660827/java-generics-covariance)

How's a raw type different from using <?> as a type parameter?List<Object>, List<String>, etc are all List<?>, so it may be tempting to just say that they're just List instead. However, there is a major difference: since a List<E> defines only add(E), you can't add just any arbitrary object to a List<?>. On the other hand, since the raw type List does not have type safety, you can add just about anything to a List.Consider the following variation of the previous snippet:static void appendNewObject(List<?> list) {    list.add(new Object()); // compilation error!}//...List<String> names = new ArrayList<String>();appendNewObject(names); // this part is fine!The compiler did a wonderful job of protecting you from potentially violating the type invariance of theList<?>! If you had declared the parameter as the raw type List list, then the code would compile, and you'd violate the type invariant of List<String> names.If it's unsafe, why is it allowed to use a raw type?Here's another quote from JLS 4.8:The use of raw types is allowed only as a concession to compatibility of legacy code. The use of raw types in code written after the introduction of genericity into the Java programming language is strongly discouraged. It is possible that future versions of the Java programming language will disallow the use of raw types.Effective Java 2nd Edition also has this to add:Given that you shouldn't use raw types, why did the language designers allow them? To provide compatibility.The Java platform was about to enter its second decade when generics were introduced, and there was an enormous amount of Java code in existence that did not use generics. It was deemed critical that all this code remains legal and interoperable with new code that does use generics. It had to be legal to pass instances of parameterized types to methods that were designed for use with ordinary types, and vice versa. This requirement, known as migration compatibility, drove the decision to support raw types.In summary, raw types should NEVER be used in new code. **You should always use parameterized types**.Are there no exceptions?Unfortunately, because Java generics are non-reified, there are two exceptions where raw types must be used in new code:* Class literals, e.g. List.class, not List<String>.class
* instanceof operand, e.g. o instanceof Set, not o instanceof Set<String>
 |

//baseline sitautation

/\*\* @abst (i1, i2, . . . , in) or () for the empty stack \*/

public class StackOfInts {

/\*\* @abst AF(this) == () \*/

public StackOfInts(){

/\*\* @abst $ret == i1 \*/

public int top(){

//what does the query return?

/\*\* @abst $ret == (AF(this) == ()) \*/

public boolean isEmpty()

//state after the command is execucted

/\*\* @abst AF(this) == (i2, i3, . . . , in) \*/

public void pop()

/\*\* @abst AF(this) == (x, i1, . . . , in) \*/

public void push(int x)

/\*\* @abst $ret == n \*/

public int count()

}

 in)