Provide an example and explanation of anonymous classes in Java

|  |
| --- |
|  |

Here we present an example and a short tutorial on anonymous classes in Java. Anonymous classes in Java are more accurately known as anonymous ***inner*** classes – there’s no such thing as anonymous classes without the “inner”. That distinction is important, because the fact that they are anonymous **inner** classes means that they are defined inside another class. If you’ve read our article on [inner versus nested classes](http://www.programmerinterview.com/index.php/java-questions/inner-vs-nested-classes/), then you should be familiar with how inner classes work by now.

An anonymous inner class is an inner class that is declared without using a class name at all – and that of course is why it’s called an *anonymous* class. An anonymous inner class also has some pretty unusual syntax.

Let’s go through an actual example with some code of an anonymous inner class to help you understand what it is exactly:

Anonymous inner class example:

class ProgrammerInterview {

public void read() {

System.out.println("Programmer Interview!");

}

}

class Website {

/\* This creates an anonymous inner class: \*/

ProgrammerInterview pInstance = new ProgrammerInterview() {

public void read() {

System.out.println("anonymous ProgrammerInterview");

}

};

}

Understanding our anonymous inner class example

In the code above, you can see that we have two classes – one called Website and another called ProgrammerInterview. The ProgrammerInterview class is pretty straightforward – there’s just a simple method called “read()” that prints the text “Programmer Interview!” when called.

The code that you need to really look closely at is inside the Website class, and is highlighted in the color red. It might look like we are creating an instance of the ProgrammerInterview class called pInstance in that code, but ***what’s actually happening in that code is that an instance of an anonymous class is being created.***

An anonymous inner class is a subclass

|  |
| --- |
|  |

Pay special attention to the fact that inside the curly braces – after the “new ProgrammerInterview()” code – there is actually a method definition for a method named “read()”. This certainly does not look like we are creating a normal instance of a class – because you don’t normally see methods being defined at the same time that an instance of a class is created.

What’s actually happening in the code above is that we are creating an instance of a***subclass (also known as a child class)*** of the ProgrammerInterview class. **And, the most important thing to understand here is that this instance (*pInstance*) is actually an instance of an anonymous subclass of the ProgrammerInterview class**.

Why is it called an anonymous inner class?

The reason it’s called an ***anonymous*** inner class is because the class that we have created clearly has no name! We jump straight to creating an instance of the class, but we do not even give the class a name – all we have is a reference variable (pInstance, in our example above) for the anonymous inner class.

Just to emphasize the syntax differences between creating an anonymous inner class instance and a normal class instance, here is the code for creating a ***normal*** class instance – assuming we want to create an instance of the ProgrammerInterview class :

/\*Pay attention to the semicolon at the end,

and the use of parentheses instead of braces:

\*/

ProgrammerInterview p = new ProgrammerInterview();

The syntax above to create an instance of the ProgrammerInterview class is nothing out of the ordinary, and something you’re probably already familiar with.

Now, let’s look at the code we have for an anonymous inner class:

Anonymous inner class syntax in Java

/\*Pay attention to the opening curly braces

and the fact that there's a semicolon

at the very end, once the anonymous class is created:

\*/

ProgrammerInterview pInstance = new ProgrammerInterview() {

//code here...

};

Anonymous inner classes and polymorphism

|  |
| --- |
|  |

When using anonymous inner classes, polymorphism is actually at work as well. Taking another look at our example above, note that pInstance is actually a superclass reference type that refers to a subclass object. In plain English, that means pInstance is of type ProgrammerInterview (which is the superclass), but pInstance refers to a subclass (or child class) of the ProgrammerInterview class – and this is polymorphism at work. That subclass is the anonymous inner class with no name that is created inside the Website class.

So, what exactly are the implications of an anonymous inner class using polymorphism? Well, it means that using the anonymous inner class reference variable type (pInstance in our example) you can only call methods that are defined inside the type (the class) of the reference variable. Using our example, this means that with pInstance we can only call methods that are defined inside the ProgrammerInterview class.

You might be confused, so let’s take a look at another example to understand exactly what we mean. Suppose we have the following simple classes:

class Animal{

void run() {

}

}

class Dog extends Animal {

void bark() {

}

}

Now, let’s create an instance of the Animal class, but make it so that it points to the class that derives from it, Dog:

class Testing{

public static void main(String[] args) {

Animal d = new Dog();

/\*This is totally legal, calling

the method run is no problem

because it is defined inside

the Animal class: \*/

d.run();

/\*Compliler Error! Calling

the method bark results

in an error because it is

not defined in the Animal

class: \*/

d.bark();

}

}

In the code above, the call to “d.run()” is perfectly legal, but the call to “d.bark()” results in a compiler error because the “bark()” method is not defined inside the Animal class, and our object “d” is of type Animal. To re-emphasize this point, this makes sense because our reference variable type is of type Animal, and even though it refers to a subclass object (from class Dog in this case), it still doesn’t know anything about methods defined in the Dog class.

Now, what does all this have to do with anonymous inner classes? Well, if we try to invoke a method that is defined inside our anonymous class which is not overridden from the superclass, using our anonymous inner class reference, then we will get an error. That sentence must be really confusing, right? But, if you understood the fairly simple example we gave above with the Animal and Dog class, then you shouldn’t have a problem understanding this concept.

And, as always we will give you an example of this scenario. So, take a look:

Example of anonymous class reference accessing non-overridden method

class ProgrammerInterview {

public void read() {

System.out.println("Programmer Interview!");

}

}

class Website {

ProgrammerInterview pInstance = new ProgrammerInterview() {

public void read() {

System.out.println("anonymous ProgrammerInterview");

}

public void learn() {

System.out.println("anonymous, learn ProgrammerInterview");

}

};

public void readIt() {

/\*

This is legal:

\*/

pInstance.read();

/\*

Compiler error, the learn method is

not also defined inside the ProgrammerInterview

class:

\*/

pInstance.learn();

}

}

In the code above, we have defined a readIt method that is a part of the Website class. Inside the readIt method, we use the pInstance object that is an instance of the anonymous class that we created earlier. And, we use that pInstance object to call the “learn()” method, as you can see which we highlighted in red.

But, because the learn() method was defined inside the anonymous inner class and **not in the ProgrammerInterview class**, the pInstance object (which is our anonymous inner class object and is of type ProgrammerInterview) has no idea what the learn() method is. The line “pInstance.learn()” will result in a compiler error – something like “cannot resolve symbol”. Hopefully that all made sense to you – if not, just read it again slowly!

What is the purpose of an anonymous inner class?

You have seen now that by creating an anonymous inner class, we can override one or more methods of a superclass. In our example above, the superclass is the ProgrammerInterview class, and the method being overridden is the read() method.

But, we could have easily done the same thing by just creating a separate class, having it extend the ProgrammerInterview class, and then just override the read() method. So, what is the need to create an anonymous inner class when we could have done the same thing using a normal, separate class?

Well, the main thing is that it is quicker to just create an anonymous inner class rather than create a new separate class. Anonymous inner classes are especially useful when you only need to override a small amount of functionality (like just one method) in a superclass, and don’t want to deal with the overhead of creating an entire class for something so simple.

The anonymous inner class and interfaces

There is actually another way – a second way – to create an anonymous inner class that you should be aware of. It’s basically an anonymous inner class that implements an interface. Read here to find out more about the second way: [Anonymous inner class interface](http://www.programmerinterview.com/index.php/java-questions/anonymous-class-interface/).

What is the difference between an inner and nested class in Java? What about the difference between an inner class and a static inner class?

This tutorial is a bit complex, but we’ll try to keep things as simple as possible. Withthat said, let’s start off with definition and explanation of nested classes.

Nested classes can be either static or non-static

Nested classes can be further classified into two different types of classes: non-static nested classes and static nested classes. Non-static nested classes are more formally known as inner classes. So, think of nested classes as a big container with 2 smaller boxes inside – 1 box is for static nested classes, and another box is for inner classes (also known as non-static nested classes).

Example of an inner class (aka non static nested class)

Here’s a simple example of an inner class – where InnerClass is the inner class:

class OuterClass {

/\* some code here...\*/

class InnerClass { }

/\* some code here...\*/

}

Note in the code above that InnerClass is literally declared inside the OuterClass class.

Inner classes are subsets of nested classes

|  |
| --- |
|  |

Remember that an ***inner class is a specific type of nested class*** thatoccurs when a nested class is non-static. And that is the “difference” between an inner class and a nested class – in other words, inner classes are subsets of nested classes. So, be careful, because the terms “inner class” and “nested class” are NOT interchangeable. You can say that an inner class is also a nested class, but you can NOT say that a nested class is also an inner class. This is because nested classes are part of the larger set that includes both inner classes and static nested classes.

What’s so special about inner classes?

So, what exactly is special about inner classes? Well, the main thing that you must remember about inner classes is that an instance of an inner class has access to all of the members of the outer class, even those that are marked “private”. So, when an instance of the inner class is created, there are no issues with having the inner class access the instance variables of the outer class.

Inner class versus static inner classes

Before we dive into the differences between inner classes and static inner classes, the most important thing you should know is that *static inner classes* is the wrongterminology – they should be called *static nested classes* instead.

Why you must use static “nested” classes instead of static “inner” classes

There is no such thing as a static ***inner*** class, because the term “***inner*** class” means that the inner class has access to the instance variables of the outer class.

An inner class is part of the “inner circle” of the outer class

Think of an inner class as being part of the “inner circle” of the outer class – an inner class instance can access all the members of the outer class, even the ones declared private. But, if an inner class were also declared to be static it would be impossible to have access to all of the members of the outer class – think about why on your own before you read our answer. You should be able to come up with an answer on your own as long as you know what static means.

Inner classes have access even to non-static members of outer class

The reason an inner class can’t be static is because of the fact that it’s impossible to access non-static variables and methods from within a static context. So, a static inner class would only have access to the static members of the outer class. And, by definition, an inner class should have access even to the non-static members of the outer class. So, instead of calling them static inner classes, we call them static nested classes, and remember that the reason is that ‘inner’ classes have a special relationship with the outer class. And to call it a static inner class would be a misuse of the terminology – so they are called static nested classes instead.

Example of a static nested class

Here’s a simple example of a static nested class:

class Outer {

static class NestedStatic { }

}

The correct way of thinking about the NestedStatic class in our example above is that it is a static member of the outer class. And because it is a static member, it means that it can be accessed ***without*** an instance of the Outer class.

How to instantiate a static nested class

The syntax used to instantiate a static nested class is different depending on whether the nested static class is a member of the current class or if the static nested class is nested in some other class. Confused? Some examples will help clarify what we mean:

Instantiating a static nested class from a non-enclosing class

Suppose we want to create an instance of a static nested class from another class. Here is some code where we do that – note that in the NonEnclosingClass class we instantiate the static class called Nested that is a member of the EnclosingClass class.

class EnclosingClass {

static class Nested {

void someMethod() { System.out.println("hello"); }

}

}

class NonEnclosingClass {

public static void main(String[] args) {

/\*instantiate the Nested class that is a static

member of the EnclosingClass class:

\*/

EnclosingClass.Nested n = new EnclosingClass.Nested();

n.someMethod(); //prints out "hello"

}

}

Instantiating a static nested class from an enclosing class

Here you can see an example of how to instantiate a static inner class that is already a part of the current class – note the more ‘normal’ syntax that you’re probably used to seeing of “Nested n = new Nested();” versus “EnclosingClass.Nested n = new EnclosingClass.Nested();” which is what was used to instantiate a static class that’s a member of a different class, as we showed in the example above.

class EnclosingClass {

static class Nested {

void anotherMethod() { System.out.println("hi again"); }

}

public static void main(String[] args) {

//access enclosed class:

Nested n = new Nested();

n.anotherMethod(); //prints out "hi again"

}

}

Accessing non-static instance variables from a static nested class

Now, let’s see what happens if we try to access some non-static instance variables that belong to the Outer class from inside of the NestedStatic class:

class Outer {

int instanceVar = 5;

static class NestedStatic {

public static void main(String[] args){

/\*instanceVar is a non-static variable

belonging to the Outer class: \*/

instanceVar = 10;

}

}

}

The code above throws a compiler error saying “non-static variable instanceVar cannot be referenced from a static context”. That error makes perfect sense because we are trying to access instanceVar, which is non-static, from the NestedStatic class, which is static. And, because of the fact that we can access the static NestedStatic class without an object of the Outer class, it makes sense that we should not be able to access an instance variable like instanceVar (which needs an object of the Outer class in order to be accessed since it’s not static).

Conclusion

If you are looking for another interesting article on inner classes, then check out: [When to use inner classes in Java](http://www.programmerinterview.com/index.php/java-questions/when-to-use-inner-classes-in-java/)