2885 users solved this topic. Latest completion was about 1 hour ago.

Theory: Anonymous classes

31 minutes 0 / 5 problems solved

Skip this topic Start practicing

§1. What is an anonymous class?

Sometimes developers need to use a small class which overrides some methods of another class or interface only once. In this case, declaring a new class may be superfluous. Fortunately, Java provides a mechanism for declaring and instantiating a class in a single statement without having to declare a new named class. Such classes are called **anonymous** because they don't have name identifiers like `String` or `MyClass` (but they do have an internal name).

Current topic:

Anonymous classes

Topic depends on:

✕ Abstract class

✕ Interface

Topic is required for:

Functional interfaces

Multithreading in Swing

§2. Writing anonymous classes

An anonymous class always implements an interface or extends another class (concrete or abstract). Here is the common syntax of creating an anonymous class:

```
new SuperClassOrInterfaceName() {

    // fields

    // overridden methods
};
```

The syntax of an anonymous class is similar to a constructor call except that there is a class definition contained in a block of code.

> An anonymous class must override all abstract methods of the superclass. That is, all interface methods must be overridden except default methods. If an anonymous class extends a class that has no abstract methods, it doesn't have to override anything.

Example. Let's assume we have the following interface with two methods:

```
interface SpeakingEntity {

    void sayHello();

    void sayBye();
}
```

Here is an anonymous class that represents an English-speaking person:

```
SpeakingEntity englishSpeakingPerson = new SpeakingEntity() {

    @Override
    public void sayHello() {
        System.out.println("Hello!");
    }

    @Override
    public void sayBye() {

        System.out.println("Bye!");

    }

};
```

The anonymous class is declared and instantiated inside a method. It overrides both methods of the interface.

We assign an instance of the **anonymous class** to the variable of the interface type. Now, we can invoke overridden methods:

```
englishSpeakingPerson.sayHello();
englishSpeakingPerson.sayBye();
```

Of course, the result is

```
Hello!
Bye!
```

Let's declare and instantiate another anonymous class:

```
SpeakingEntity cat = new SpeakingEntity() {

    @Override
    void sayHello() {
        System.out.println("Meow!");
    }

    @Override
    public void sayBye() {

        System.out.println("Meow!");

    }

};
```

When we invoke the same methods, we obtain the following result:

```
Meow!
Meow!
```

So, `englishSpeakingPerson` and `cat` are instances of different anonymous classes that implement the same interface.

§3. Accessing context variables

In the body of an anonymous class, it is possible to capture variables from a context where it is defined:

- an anonymous class can capture members of its enclosing class (the outer class);
- an anonymous class can capture local variables that are declared as `final` or are **effectively final** (i.e. the variable is not changed but it doesn't have the `final` keyword).

Here is another anonymous class that implements the `SpeakingEntity` interface:

```
public class AnonymousClassExample {


private static String BYE_STRING = "Auf Wiedersehen!"; // static constant

    public static void main(String[] args) {

        final String hello = "Guten Tag!"; // final local variable

        SpeakingEntity germanSpeakingPerson = new SpeakingEntity() {



            @Override

            public void sayHello() {


        System.out.println(hello); // it captures the local variable

            }



            @Override

            public void sayBye() {


        System.out.println(BYE_STRING); // it captures the constant field

            }

        };



        germanSpeakingPerson.sayHello();



        germanSpeakingPerson.sayBye();

    }

}
```

The anonymous class captures the constant field `BYE_STRING` and the local final variable `hello`. This code is successfully compiled and prints what we expect:

```
Guten Tag!
Auf Wiedersehen!
```

A declaration of a variable or a method in an anonymous class shadows any other declaration in the enclosing scope that has the same name. You cannot access any shadowed declarations by their names.

§4. Restrictions on anonymous classes

Anonymous classes have some restrictions:

- they cannot have static initializers or interface declarations;
- they cannot have static members, except the constant variables (final static fields);
- they cannot have constructors.

For example, let's see the following anonymous class that has a final static field and an instance initializer to substitute a constructor:

```
final String robotName = "Bug";
final int robotAssemblyYear = 2112;

SpeakingEntity robot = new SpeakingEntity() {

    static final int MAGIC_CONSTANT = 10;

    private String name;
    private int assemblyYear;



    { /* instance initialization block for setting fields */

        name = robotName;

        assemblyYear = robotAssemblyYear;

    }



    @Override

    public void sayHello() {

        System.out.println("1010001" + MAGIC_CONSTANT);

    }



    @Override

    public void sayBye() {

        System.out.println("0101110" + MAGIC_CONSTANT);

    }

};
```

§5. When to use anonymous classes

Generally, you should consider using an anonymous class when:

- only one instance of the class is needed;
- the class has a very short body;
- the class is used right after it's defined.

In this topic, we've considered rather simple anonymous classes to understand the basic syntax, but in real life applications, they provide a powerful mechanism for creating classes that encapsulate behaviors and pass them to suitable methods. This is a convenient way to interact with parts of our application or with some third-party libraries.

For instance, anonymous classes are actively used when writing user interfaces with the standard Java library called **Swing**. The same with developing a web user interface using **Google Web Toolkit (GWT)**. It is very common to have a lot of listeners that are used just once for one button, so using anonymous classes allows us to avoid writing a lot of classes and having useless files in the development of the code.

Some widespread libraries for working through the **HTTP** protocol also use anonymous classes. For example, this HttpAsyncClient. Right now you may not understand how to use it but you can see how many anonymous classes there are.

§6. Learn callbacks by example

Often, after creating an instance of an anonymous class we pass it to some method as an argument. In this case, the anonymous class is called a **callback**. A callback is a piece of executable code that is passed to another code that executes it (performs a call back) at a convenient time.

Let's consider an example. There is a special kind of calculator that can only divide numbers. The calculator takes a callback as its argument and executes the callback passing the result of the calculation or an error message.

The `Callback` interface has two abstract methods:

```
interface Callback {

    /**
     * Takes a result and processes it
     */
    void calculated(int result);

    /**
     * Takes an error message

     */

    void failed(String errorMsg);

}
```

The class `Divider` has only one static method (just an example, the demonstrated technique works with any methods):

```
class Divider {

    /**
* Divide a by b. It executes the specified callback to process results
     */
    public static void divide(int a, int b, Callback callback) {

        if (b == 0) {
            callback.failed("Division by zero!");

            return;

        }


        callback.calculated(a / b);

    }

}
```

Of course, in this case, you can perform the division and return the result without any callbacks. In general, though, callbacks can help you in large applications with multiple parts and layers (especially in multithreaded programs).

Calling a method with a callback:

Table of contents:

```
public class CallbacksExample {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int a = scanner.nextInt();
        int b = scanner.nextInt();


Divider.divide(a, b, new Callback() { // passing callback as an argument



            @Override

            public void calculated(int result) {


        String textToPrint = String.format("%d / %d is %d", a, b, result);

                print(textToPrint);

            }



            @Override

            public void failed(String errorMsg) {

                print(errorMsg);

            }

        });

    }


    public static void print(String text) {

        System.out.println(text);

    }

}
```

As you can see, we instantiate and pass the callback without any additional variables of the `Callback` type. It's a very common practice for working with callbacks, especially if they are small.

The callback captures the static method `print` and the local variables `a` and `b` from its context. The variables `a` and `b` are **effectively final** here.

Let's run the program.

Input 1:

```
8 2
```

Output 1:

```
8 / 2 is 4
```

Input 2:

```
10 0
```

Output 2:

```
Division by zero!
```

So, anonymous classes along with the context capture mechanism allow you to transfer logic between parts of your program. They are used as callbacks in large applications and when working with external libraries.

Report a typo

254 users liked this theory. **15** didn't like it. **What about you?**

Start practicing

Comments (21) Hints (0) Useful links (2) Show discussion