Theory: Abstract class

39 minutes

Verify to skip | Start practicing

Sometimes you have a set of fields and methods that you need to reuse in all classes within a hierarchy. It is possible to put all the common members to a special base class and then declare subclasses which can access these members. At the same time, you do not need to create objects of the base class. To achieve it, you can use an **abstract class** as the base class in the hierarchy.

§1. What is an abstract class?

An **abstract class** is a class declared with the keyword `abstract`. It represents an abstract concept that is used as a base class for subclasses.

Abstract classes have some special features:

- it's impossible to create an instance of an abstract class;
- an abstract class can contain abstract methods that must be implemented in non-abstract subclasses;
- it can contain fields and non-abstract methods (including static);
- an abstract class can extend another class, including abstract;
- it can contain a constructor.

As you can see, an abstract class has two main differences from regular (concrete) classes: **no instances** and **abstract methods**.

Abstract methods are declared by adding the keyword `abstract`. They have a declaration (modifiers, a return type, and a signature) but don't have an implementation. Each concrete (non-abstract) subclass must implement these methods.

Note, static methods can't be abstract!

§2. Example

Here is an abstract class `Pet`:

```
1  public abstract class Pet {
2
3      protected String name;
4      protected int age;
5
6      protected Pet(String name, int age) {
7          this.name = name;
8          this.age = age;
9      }
10
11     public abstract void say(); // an abstract method
12 }
```

The class has two fields, a constructor, and an abstract method.

Since `Pet` is an abstract class we cannot create instances of this class:

```
1
Pet pet = new Pet("Unnamed", 5); // this throws a compile time error
```

The method `say()` is declared abstract because, at this level of abstraction, its implementation is unknown. Concrete subclasses of the class `Pet` should have an implementation of this method.

1 required topic

✕ Polymorphism

4 dependent topics

Template method

Anonymous classes properties

Abstract class vs interface

Authentication

Below are two concrete subclasses of `Pet`. You can see that they override the abstract method:

```
class Cat extends Pet {

    // It can have additional fields as well

    public Cat(String name, int age) {
        super(name, age);
    }

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

class Dog extends Pet {

    // It can have additional fields as well

    public Dog(String name, int age) {
        super(name, age);
    }

    @Override
    public void say() {
        System.out.println("Woof!");
    }
}
```

We can create instances of these classes and call the `say()` method.

```
Dog dog = new Dog("Boss", 5);
Cat cat = new Cat("Tiger", 2);

dog.say(); // it prints "Woof!"
cat.say(); // it prints "Meow!"
```

Do not forget that Java doesn't support multiple inheritance for classes. Therefore, a class can extend only one abstract class.