Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

- ○ What is Java?
- ○ Installation
- ○ Hello World Example
- ○ Data Types
- ○ Packages
- ○ Naming Conventions Cheat Sheet
- ○ Flow of Control
- ○ Class Members
- ○ Operators
- ○ Conditionals
- ○ Iteration
- ○ Arrays
- ○ ArrayList
- ○ Enhanced For Loops
- ○ String Manipulation
- ○ Class Constructors
- ○ Access Modifiers
- ○ Installing Java & Maven To PATH
- ○ Object-Oriented Programming Principles
- ○ Encapsulation
- ○ Inheritance
- ○ Polymorphism
- ○ Abstraction
- ○ Interfaces
- ○ Type Casting
- ○ Static
- ○ Final
- ○ Garbage Collection
- ○ Input With Scanner
- ○ Pass by Value/Reference
- ○ JUnit

Inheritance

Contents

Overview

Of the four *object-oriented programming principles*, **inheritance** is the concept of one object acquiring all the *non-`private`* properties of another object. (If the two objects are in separate packages, the `default` properties would also not be inherited).

It's possible to inherit both *attributes* and *methods*.

This is useful for when an object is *based on* another, but needs to have *more specific* properties applied to it.

Generally, this takes the form of a *parent* and *child* relationship, where the *child object* inherits all behaviours from the *parent object* - just like in real life.

In Java, these are referred to as the `superclass` (parent) and the `subclass` (child).

Inheritance in action

Let's say that our program requires an `Owl` class and a `Chicken` class, as well as a `Runner` to contain our `main()` method.

Both `Owl` and `Chicken` need to represent two sets of *properties* (attributes) and *behaviours* (methods):

- their own, which make them unique and different from each other
- those which they have in common, since they're both birds

We can organise these two classes in a hierarchy, where they both inherit the attributes and methods of a common `superclass`.

We'll keep all our classes `public` and in the same package, so that our `Runner` can access our other classes easily.

Let's call this class `Bird`:

▶ Bird

Through inheritance, we can now give these generic attributes and methods to our more specific `subclass`es, *and* add the extra functionality they need to each, ~~killing two birds with one stone~~!

Note the `extends` keyword that we use to do this.

▶ Owl

▶ Runner

The object `owlfriend` we made can access both the `fly` boolean and the `noise()` method from the `Bird` class, but still has its own functionality.

Let's do the same for our `Chicken` (we'll update our `Runner` too):

▶ Chicken
▶ Runner

Our `Chicken` can contain completely different functionality to `Owl`, but can still access the `fly` boolean from the `Bird` `superclass`.

A note on the `Object` `superclass`

Every object you create, or use, automatically inherits from the Java `superclass` called `Object`.

Every object you make inherits a set of methods already defined in `Object`, all of which can be overridden if necessary:

- `clone()`
- `equals()`
- `finalize()`
- `getClass()`
- `hashCode()`
- `notify()`
- `notifyAll()`
- `toString()`
- `wait()`

A note on the `final` keyword

`final` does not play well with inheritance, since it is meant to apply an unchanging singular value to something.

When applied to a class, it *cannot be used as a superclass* (it cannot be *extended*) - so use it wisely.

Overriding methods

An instance method in a `subclass` which has the same *signature* (name, number of parameters, type of parameters) or *return type* (`void`, `String`, etc.) as an instance method in the `superclass` *overrides* the method in the `superclass`.

This allows a `subclass` to inherit a behaviour from a `superclass` and then *modify its behaviour* for its own needs.

In previous versions of Java, the `@Override` annotation was commonly used in the `subclass` to instruct the compiler to override the corresponding method in the `superclass`.

Nowadays, the `@Override` annotation is mostly used as a flag for developers: it's essentially there to say that one method *should* override another. Nevertheless, it's still common (and best!) practice to use it.

Let's see how this looks in practice with a `superclass` `Animal` and `subclass` `Cat`.

We'll call to them with a `Runner` class, which is situated in the same package.

(We'll look at what that `abstract` keyword is doing in the Abstraction module.)

▶ Overriding

Here, we access the `eat()` method through an instance `someAnimal` of the `Cat` class.

We can use `Animal` as the blueprint for our `Cat` object because it is the `superclass`.

Hiding methods

We can also *hide methods* using inheritance.

This can only be done with `static` methods.

If a `subclass` defines a `static` method with the same signature as a `static` method in the `superclass`, then the method in the `subclass` *hides* the ones in the `superclass`.

This can be handy for situations where you would need to invoke a method within the `superclass`, but also want to use methods in the `subclass` as well.

Let's rewrite our `Animal`, `Cat` and `Runner` classes:

▶ Hiding methods

Here, we've put a `static` method called `makeNoise()` to both the `Animal` and `Cat` classes.

When we run this program, the method that gets invoked is the one in `Animal`, not `Cat`:

```
growllll
```

This is because the `Cat` class has *hidden* the `static` method `makeNoise()` in `Animal`.

As a result, the version of the hidden `static` method that gets invoked is the one in the `Animal` `superclass`. We *expect* to see `hissss`, but instead, we see `growllll`.

You may find that Eclipse gives a warning when writing this code:

```
The static method makeNoise() from the type Animal should be accessed in a
static way
```

Eclipse reckons that if we're trying to access the method in `Animal`, we should simply reference it directly - which we can do, since it's `static` - so let's edit our `Runner` to do this:

▶ Runner

By referencing our `Animal` class directly, we're able to use the `makeNoise()` method in `Animal` *without making an instance of `Animal` first*, which can be useful for situations where creating an entire object is not necessary.

However, this can lead to the over-use of the `static` keyword - or *static-poisoning*, which is bad practice and goes against the OOP Principles - so use this knowledge wisely!

Tutorial

There is no tutorial for this module.

Exercises

Proof-of-concept

▶ Consider the following two classes:

Which method overrides a method in the `superclass`?

▶ Show answer

Which method hides a method in the `superclass`?

▶ Show answer

What do the other two methods produce?

▶ Show answer