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

Abstraction

Contents

Overview

Of the four *object-oriented programming principles*, **abstraction** states that you should *hide* the implementation *details* between modules, and only share essential *functionality*.

In other words, we don't need to show the implementation details to every class or method.

Abstraction in one form: `abstract`

One way this is done in Java is through the use of the `abstract` keyword, which can be applied to both *classes* and *methods*:

- `abstract` *classes* cannot be instantiated, so we cannot make objects from them - but they can be *inherited* (extended) from
- `abstract` *methods* can only exist in `abstract` classes, and contain no body - these methods need to be implemented in `subclasses`

Let's take a look at one in action:

▶ Abstraction with the abstract keyword
By making the `Bird` class `abstract`, we can't instantiate an object from it directly.

However, we can now use the `abstract` method `noise()`, whose functionality is used in the `subclass Magpie`.

We're still able to implement regular methods like `sleep()`, however.

This means that in the `subclass Magpie`, we're now able to use the `abstract` method `noise()` with our own implementation.

This allows us to hide the *details* of `Bird` from every class *except* those which need to implement its methods.

Abstraction in another form: `Interface`s and implementations

What if you had a class that needed to extend functionality from more than one place?

Java doesn't allow for the use of multiple `superclass`es - only one is allowed.

However, we can still accomplish abstraction with `interface`s:

- they are completely abstract by design (no methods in an interface have bodies)
- you can `implement` multiple `interface`s

○ Test Driven Development
○ UML Basics
○ JavaDoc
○ Peer Programming
○ Code Reviews
Maven
Testing (Foundation)
Java Intermediate
HTML
CSS
Javascript
Spring Boot
Selenium
Sonarqube
Advanced Testing (Theory)
Cucumber
MongoDB
Express
NodeJS
React
Express-Testing
Networking
Security
Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

Let's see how this works in action - here we've got two `interface`s:

```
public interface Flyable {
    public void spreadWings();
    public void takeOff();
}
```

```
public interface Hatchable {
    public void emergeFromEgg();
    public void cheep();
}
```

We have a `Chicken` class which we want to use to implement all the functionality from both `Flyable` and `Hatchable`, which we can easily do with the `implements` keyword:

```
public class Chicken implements Flyable, Hatchable {

    @Override
    public void spreadWings(){
        System.out.println("spreading wings...");
    }

    @Override
    public void takeOff(){
        System.out.println("taking off... WHOOSH");
    }

    @Override
    public void emergeFromEgg(){
        System.out.println("cracking egg...");
    }

    @Override
    public void cheep(){
        while (true) {
            System.out.println("cheep");
        }
    }
}
```

`Chicken` can implement an unlimited number of `interface`s without issue.

We can write our own implementation of the methods within each `interface`, too.

What if we don't want to implement every method from an `interface`?

Normally, when implementing an `interface`, *all* methods within it have to be implemented.

However, you can get around this problem in two ways:

- just make the `interface`s smaller - the most optimal `interface`s have one method
- implement `interface`s that shouldn't be split into `abstract` classes

Let's look at this second point a bit closer.

Let's say we've got an `interface` called `Y` with only a few methods we want to implement in class `X`:

```
public interface Y {
    public String methodA();
    public int methodB();
}
```

```
public abstract class X implements Y {

    @Override
    public String methodA(){
        return "A";
    }
    //public int methodB() is unimplemented because we don't want it here
}
```

Making `X` an `abstract class` stops us from needed to implement `methodB()`.

We can still extend `X` when we need to use all of `Y`'s methods:

```
public class XX extends X {

    @Override
    public String methodA(){
        return "A";
    }

    @Override
    public int methodB(){
        return 1355417;
    }
}
```

By extending `X` into a *non-abstract class*, we can:

- implement all the methods that `X` implements from `Y`
- implement `methodB()` from `Y`

Tutorial

There is no tutorial for this module.

Exercises

Shape

Consider the following program:

```
public abstract class Shape {
    protected abstract double getArea(double length);
    protected abstract double getPerimeter(double length);
}
```

```
public class Square extends Shape {

    @Override
    public double getArea(double side){
        return side * side;
    }

    @Override
    public double getPerimeter(double size){
        return side * 4;
    }

    public String getColour(){
        return "blue";
    }
}
```

```
public class Runner {

    public static void main(String[] args){
        Square s = new Square();
        System.out.println(s.getArea(3) + ", " + s.getPerimeter(3) + ", " +
s.getColour())
    }
}
```

Where is `Square` getting its methods from?

▶ Show solution