ABSTRACTION IN JAVA

THE ULTIMATE GUIDE

HUGH HAMILL

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Abstraction in Java

Contents

Preface

In software engineering and programming language theory, the abstraction principle (or the principle of abstraction) is a basic dictum that aims to reduce duplication of information in a program (usually with emphasis on code duplication) whenever practical by making use of abstractions provided by the programming language or software libraries. The principle is sometimes stated as a recommendation to the programmer, but sometimes stated as requirement of the programming language, assuming it is self-understood why abstractions are desirable to use. The origins of the principle are uncertain; it has been reinvented a number of times, sometimes under a different name, with slight variations.

When read as recommendation to the programmer, the abstraction principle can be generalized as the "don't repeat yourself" principle, which recommends avoiding the duplication of information in general, and also avoiding the duplication of human effort involved in the software development process.

- Wikipedia

http://en.wikipedia.org/wiki/Abstraction_principle_%28computer_programming%29

About the Author

Hugh Hamill is a Senior Software Engineer and Certified Scrum Master based in Galway, Ireland. He achieved his B.Sc. in Applied Computing from Waterford Institute of Technology in 2002 and has been working in industry since then. He has worked for a several large blue chip software companies listed on both the NASDAQ and NYSE.

Hugh has previously worked with both C++ and VB but currently concentrates on the Java stack. He has experience of building desktop applications using swing and enterprise back ends using both JEE and Spring. He has been tech lead and architect on several projects including thick client configuration tools, an event replicating DB access layer, a performance test infrastructure, a portlet framework, a rules based decision engine for DRM and is currently working on a next gen technology for processing large amounts of customer data.

His main areas of interest include Core Java, JEE, Spring, Design Patterns, TDD, Maven and Agile Software Development.

Chapter 1

Introduction

In this tutorial we will give an introduction to Abstraction in Java and define a simple Payroll System using Interfaces, Abstract Classes and Concrete Classes.

There are two levels of abstraction in Java - **Interfaces**, used to define expected behaviour and **Abstract Classes**, used to define incomplete functionality.

We will now look at these two different types of abstraction in detail.

Chapter 2

Interfaces

An interface is like a contract. It is a promise to provide certain behaviours and all classes which implement the interface guarantee to also implement those behaviours. To define the expected behaviours the interface lists a number of method signatures. Any class which uses the interface can rely on those methods being implemented in the runtime class which implements the interface. This allows anyone using the interface to know what functionality will be provided without having to worry about how that functionality will actually be achieved. The implementation details are hidden from the client, this is a crucial benefit of abstraction.

2.1 Defining Interfaces

You can use the keyword `interface` to define an interface:

```
public interface MyInterface {

void methodA();

int methodB();

String methodC(double x, double y);

}
```

Here we see an interface called `MyInterface` defined, note that you should use the same case conventions for Interfaces that you do for Classes. `MyInterface` defines 3 methods, each with different return types and parameters. You can see that none of these methods have a body; when working with interfaces we are only interested in defining the expected behaviour, not with it's implementation.

Note: Java 8 introduced the ability to create a default implementation for interface methods, however we will not cover that functionality in this tutorial.

Interfaces can also contain state data by the use of member variables:

```
public interface MyInterfaceWithState {

        int someNumber;

        void methodA();

}
```

All the methods in an interface are public by default and in fact you can't create a method in an interface with an access level other than public.

2.2 Implementing Interfaces

Now we have defined an interface we want to create a class which will provide the implementation details of the behaviour we have defined. We do this by writing a new class and using the `implements` keyword to tell the compiler what interface this class should implement.

```
public class MyClass implements MyInterface {

        public void methodA() {
                System.out.println("Method A called!");
        }

        public int methodB() {
                return 42;
        }

        public String methodC(double x, double y) {
                return "x = " + x + ", y = " y;
        }

}
```

We took the method signatures which we defined in `MyInterface` and gave them bodies to implement them. We just did some arbitrary silliness in the implementations but it's important to note that we could have done anything in those bodies, as long as they satisfied the method signatures. We could also create as many implementing classes as we want, each with different implementation bodies of the methods from `MyInterface`.

We implemented all the methods from `MyInterface` in `MyClass` and if we failed to implement any of them the compiler would have given an error. This is because the fact that `MyClass` implements `MyInterface` means that `MyClass` is guaranteeing to provide an implementation for each of the methods from `MyInterface`. This lets any clients using the interface rely on the fact that at runtime there will be an implementation in place for the method it wants to call, guaranteed.

2.3 Using Interfaces

To call the methods of the interface from a client we just need to use the dot (.) operator, just like with the methods of classes:

```
MyInterface object1 = new MyClass();
object1.methodA(); // Guaranteed to work
```

We see something unusual above, instead of something like `MyClass object1 =new MyClass();` (which is perfectly acceptable) we declare object1 to be of type `MyInterface`. This works because `MyClass` is an implementation of `MyInterface`, wherever we want to call a method defined in `MyInterface` we know that `MyClass` will provide the implementation. `object1` is a **reference** to any runtime object which implements `MyInterface`, in this case it's an instance of `MyClass`. If we tried to do `MyInterface object1 =new MyInterface()` we'd get a compiler error because you can't instantiate an interface, which makes sense because there's no implementation details in the interface - no code to execute.

When we make the call to `object1.methodA()` we are executing the method body defined in `MyClass` because the **runtime type** of `object1` is `MyClass`, even though the reference is of type `MyInterface`. We can only call methods on `object1` that are defined in `MyInterface`, for all intents and purposes we can refer to `object1` as being of type `MyInterface` even though the runtime type is `MyClass`. In fact if `MyClass` defined another method called `methodD()` we couldn't call it on `object1`, because the compiler only knows that `object1` is a reference to a `MyInterface`, not that it is specifically a `MyClass`.

This important distinction is what lets us create different implementation classes for our interfaces without worrying which specific one is being called at runtime.

Take the following interface:

```
public interface OneMethodInterface {

        void oneMethod();

}
```

It defines one void method which takes no parameters.

Let's implement it:

```
public class ClassA implements OneMethodInterface {

        public void oneMethod() {
                System.out.println("Runtime type is ClassA.");
        }
}
```

We can use this in a client just like before:

```
OneMethodInterface myObject = new ClassA();
myObject.oneMethod();
```

Output:

```
Runtime type is ClassA.
```

Now let's make a different implementation for `OneMethodInterface`:

```
public class ClassB implements OneMethodInterface {

        public void oneMethod() {
                System.out.println("The runtime type of this class is ClassB.");
}
}
```

And modify our code above:

```
OneMethodInterface myObject = new ClassA();
myObject.oneMethod();
myObject = new ClassB();
myObject.oneMethod();
```

Output:

```
Runtime type is ClassA.
The runtime type of this class is ClassB.
```

We have successfully used the same Reference (`myObject`) to refer to instances of two different runtime types. The actual implementation is completely unimportant to the compiler, it just cares that `OneMethodInterface` is implemented, by anything, and in any way. As far as the compiler is concerned `myObject` is a `OneMethodInterface`, and `oneMethod()` is available, even if it's reassigned to a different instance object of a different class type. This ability to provide more than one runtime type and have it resolved at run time, rather than compile time, is called **polymorphism**.

Interfaces define behaviour without any implementation details (ignoring Java 8) and implementing classes define all the implementation details for the classes they define, but what happens if we want a mix of the two concepts? If we want to mix some behaviour definition and some implementation together in the same place we can use an abstract class.

Chapter 3

Abstract Classes

An abstract class is like an incomplete blueprint, it defines some of the implementation details of the class while leaving others as simple behaviour definitions to be implemented later.

Imagine a blueprint of a house where the house is drawn in completely but there is a big empty square where the garage will go. We know there will be a garage but we don't know what it will look like. Somebody else will have to take a copy of our blueprint and draw in the garage. In fact several different people may take copies of our blueprint and draw in different types of garage. Houses built using these blueprints will all be recognizable variants of our house; the front door, the room layouts and the windows will all be identical, however the garages will all be different.

Much like the blueprints above our abstract classes will define some methods completely, and these method implementations will be the same in all implementations of the abstract class. The abstract class will define only the signature for other methods, in much the same way that interfaces do. The method implementations for these methods will vary in the implementing classes. An implementing class of an abstract class is commonly referred to as a **concrete** class. Due to the inheritance relationship between the concrete class and the abstract class we generally say that a concrete class **extends** an abstract class, rather than implements it as we say with interfaces.

Just like with interfaces any client code knows that if a concrete class is extending an abstract class the concrete class guarantees to provide method bodies for the abstract methods of the abstract class (the abstract class provides it's own method bodies for non-abstract methods, of course).

Again, just like interfaces, there can be several different concrete classes of a given abstract class, each may define very different behaviours for the abstract methods of the abstract class while satisfying the contract of the abstract class. The implementation details are hidden from the client.

3.1 Defining Abstract Classes

The keyword `abstract` is used to define both a class and its methods as abstract.

```
public abstract class MyAbstractClass {

        protected int someNumber = 0;

        public void increment() {
                someNumber++;
        }

        public abstract void doSomethingWithNumber();

}
```

Here we have defined an abstract class called `MyAbstractClass` which contains an integer and provides a method for incrementing that integer. We also define an abstract method called `doSomethingWithNumber()`. We don't yet know what this

method will do, it will be defined in any concrete classes which extend `MyAbstractClass`. `doSomethingWithNumber()` doesn't have a method body because it is abstract.

In interfaces all the methods are public by default, however the scope of an abstract method may be public, package or protected.

You can see that some behavioural implementation in increment() is mixed with some behavioural definition in `doSomethingWithNumber()` in this abstract class. Abstract classes mix some implementation with some definition. Concrete classes which extend Abstract class will reuse the implementation of `increment()` while guaranteeing to provide their own implementations of `doSomethingWithNumber()`.

3.2 Extending Abstract Classes

Now that we have created an abstract class let's make a concrete implementation for it. We make concrete implementations from abstract classes by using the `extends` keyword in a class which itself is not abstract.

```
public class MyConcreteClass extends MyAbstractClass {

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

        public void doSomethingWithNumber() {
                System.out.println("The number is " + someNumber);
}

}
```

We have created a concrete class called `MyConcreteClass` and extended `MyAbstractClass`. We only needed to provide an implementation for the abstract method `doSomethingWithNumber()` because we **inherit** the non private member variables and methods from `MyAbstractClass`. If any client calls `increment()` on `MyConcreteClass` the implementation defined in `MyAbstractClass` will be executed. We also created a new method called `sayHello()` which is unique to `MyConcreteClass` and wouldn't be available from any other concrete class which implements `MyAbstractClass`.

We can also extend `MyAbstractClass` with another abstract class where we don't implement `doSomethingWithNumber` - this means that another concrete class will have to be defined, which extends this new class in order to implement `doSomethingWithNumber()`.

```
public abstract class MyOtherAbstractClass extends MyAbstractClass {

        public void sayHello() {
                System.out.println("Hello there!");
}
}
```

We didn't have to make any reference to `doSomethingWithNumber()` here, whenever we create a concrete class for `MyOtherAbstractClass` we will provide the implementation for `doSomethingWithNumber()`.

Lastly, abstract classes can themselves implement interfaces. Because the abstract class can't be instantiated itself it does not have to provide an implementation for all (or any) of the interface methods. If the abstract class does not provide an implementation for an interface method, the concrete class which extends the abstract class will have to provide the implementation.

```
public abstract MyImplementingAbstractClass implements MyInterface {

        public void methodA() {
                System.out.println("Method A has been implemented in this abstract class");
        }
}
```

Here we see that `MyImplementingAbstractClass` implements `MyInterface` but only provides an implementation for `methodA()`. If any concrete class extends `MyImplementingAbstractClass` it will have to provide an implementation for `methodB()` and `methodC()` as defined in `MyInterface`.

3.3 Using Abstract Classes

Just like with Interfaces and regular classes, to call the methods of an abstract class you use the dot (.) operator.

```
MyAbstractClass object1 = new MyConcreteClass();
object1.increment();
object1.doSomethingWithNumber();
```

Again we see that `object1` is a reference to an instance that provides a concrete implementation for `MyAbstractClass` and the runtime type of that instance is `MyConcreteClass`. For all intents and purposes object1 is treated by the compiler as if it is a `MyAbstractClass` instance. If you were to try to call the `sayHello()` method defined in `MyConcreteClass` you would get a compiler error. This method is not **visible** to the compiler through object1 because `object1` is a `MyAbstractClass` reference. The only guarantee `object1` provides is that it will have implementations for the methods defined in `MyAbstractClass`, any other methods provided by the runtime type are not visible.

```
object1.sayHello(); // compiler error
```

As with interfaces we can provide different runtime types and use them through the same reference.

Lets define a new abstract class

```
public abstract class TwoMethodAbstractClass {

        public void oneMethod() {
                System.out.prinln("oneMethod is implemented in TwoMethodAbstractClass.");
        }

        public abstract void twoMethod();
}
```

It defines one implemented method an another abstract method.

Let's extend it with a concrete class

```
public class ConcreteClassA extends TwoMethodAbstractClass {

        public void twoMethod() {
                System.out.println("twoMethod is implemented in ConcreteClassA.");
        }
}
```

We can use it in a client just like before:

```
TwoMethodAbstractClass myObject = new ConcreteClassA();
myObject.oneMethod();
myObject.twoMethod();
```

Output:

```
oneMethod is implemented in TwoMethodAbstractClass.
twoMethod is implemented in ConcreteClassA.
```

Now let's make a different concrete class which extends `TwoMethodClass`:

```
public class ConcreteClassB extends TwoMethodAbstractClass {

        public void twoMethod() {
                System.out.println("ConcreteClassB implements its own twoMethod.");
        }
}
```

And modify our code above:

```
TwoMethodAbstractClass myObject = new ConcreteClassA();
myObject.oneMethod();
myObject.twoMethod();
myObject = new ConcreteClassB();
myObject.oneMethod();
myObject.twoMethod();
```

Output:

```
oneMethod is implemented in TwoMethodAbstractClass.
twoMethod is implemented in ConcreteClassA.
oneMethod is implemented in TwoMethodAbstractClass.
ConcreteClassB implements its own twoMethod.
```

We have used the same reference (`myObject`) to refer to instances of two different runtime types. When the runtime type of `myObject` is `ConcreteClassA` it uses the implementation of twoMethod from `ConcreteClassA`. When the runtime type of `myObject` is `ConcreteClassB` it uses the implementation of twoMethod from `ConcreteClassB`. In both cases the common implementation of `oneMethod` from `TwoMethodAbstractClass` is used.

Abstract classes are used to define common behaviours while providing contracts or promises that other behaviours will be available from a concrete class later. This allows us to build object models where we can reuse common functionality and capture differing requirements in different concrete classes.

Chapter 4

A Worked Example - Payments System

We have been asked to build a payments system for a company. We know that the company has different types of employees that can be paid in different ways; salaried and with commission. The boss of the company understands that his needs will change and the system may be changed later to accommodate different types of entities which will receive payments.

4.1 The Payee Interface

Let's start by considering the employees. They must receive payments, but we also know that later on we may need to have different entities also receive payments. Let's create an interface, `Payee`, which will define the sort of behaviour we'd expect from entities which can receive payments.

```
public interface Payee {

        String name();

        Double grossPayment();

        Integer bankAccount();
}
```

Here we have a `Payee` interface which guarantees three behaviours; the ability to provide a name for the `Payee`, the ability to provide a gross payment amount which should be paid and the ability to provide a bank account number where the funds should be deposited.

4.2 The Payment System

Now that we have a `Payee` defined let's write some code that uses it. We'll create a `PaymentSystem` class which maintains a list of Payees and on demand will cycle through them and transfer the requested amount into the appropriate bank account.

```
public class PaymentSystem {

        private List<Payee> payees;

        public PaymentSystem() {
                payees = new ArrayList<>();
        }

        public void addPayee(Payee payee) {
                if (!payees.contains(payee)) {
                        payees.add(payee);
```

```
                }
        }

        public void processPayments() {
                for (Payee payee : payees) {
                                Double grossPayment = payee.grossPayment();

                                System.out.println("Paying to " + payee.name());
                                System.out.println("tGrosst" + grossPayment);
                                System.out.println("tTransferred to Account: " + payee. ↩
                                    bankAccount());
                }
        }
}
```

You can see that the `PaymentSystem` is totally `agnostic` as to the runtime types of the Payees, it doesn't care and doesn't have to care. It knows that no matter what the runtime type the `Payee` being process is guaranteed to implement `name()`, `grossPayment()` and `bankAccount()`. Given that knowledge it's simply a matter of executing a for loop across all the Payees and processing their payments using these methods.

4.3 The Employee Classes

We have been told by the boss that he has two different types of `Employee` - Salaried Employees and Commissioned Employees. Salaried Employees have a base salary which doesn't change while Commissioned Employees have a base salary and also can be given sales commissions for successful sales.

4.3.1 The SalaryEmployee Class

Let's start with a class for Salaried Employees. It should implement the Payee interface so it can be used by the Payment System.

Let's represent our changes using class diagrams in addition to code. In class diagrams a dotted arrow means an `implements` relationship and a solid arrow means an `extends` relationship.

4.3.1.1 Class Diagram

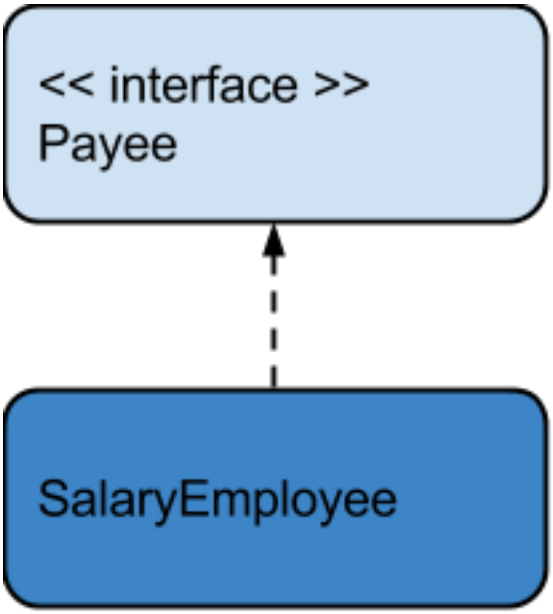

Figure 4.1: screenshot

4.3.1.2 Code

```
public class SalaryEmployee implements Payee {

        private String name;
        private Integer bankAccount;
        protected Double grossWage;

        public SalaryEmployee(String name, Integer bankAccount, Double grossWage) {
                this.name = name;
                this.bankAccount = bankAccount;
                this.grossWage = grossWage;
        }

        public Integer bankAccount() {
                return bankAccount;
        }

        public String name() {
                return name;
        }

        public Double grossPayment() {
                return grossWage;
        }
}
```

4.3.2 The CommissionEmployee Class

Now let's make a `CommissionEmployee` class. This class will be quite similar to the `SalaryEmployee` with the ability to pay commissions.

4.3.2.1 Class Diagram

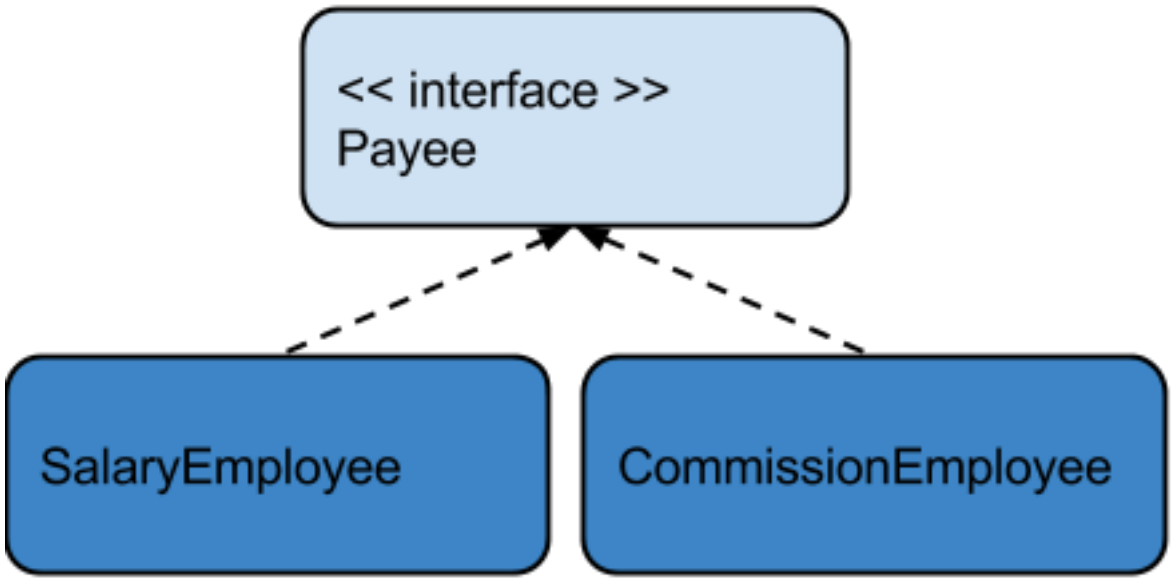

Figure 4.2: screenshot

4.3.2.2 Code

```
public class CommissionEmployee implements Payee {

        private String name;
        private Integer bankAccount;
        protected Double grossWage;
        private Double grossCommission = 0.0;

        public CommissionEmployee(String name, Integer bankAccount, Double grossWage) {
                this.name = name;
                this.bankAccount = bankAccount;
                this.grossWage = grossWage;
        }

        public Integer bankAccount() {
                return bankAccount;
        }

        public String name() {
                return name;
        }

        public Double grossPayment() {
                return grossWage + doCurrentCommission();
        }

        private Double doCurrentCommission() {
                Double commission = grossCommission;
                grossCommission = 0.0;
                return commission;
        }
```

```
        public void giveCommission(Double amount) {
                grossCommission += amount;
        }
}
```

As you can see a lot of the code is duplicated between `SalaryEmployee` and `CommissionEmployee`. In fact the only thing that's different is the calculation for `grossPayment`, which uses a commission value in `CommissionEmployee`. Some of the functionality is the same, and some is different. This looks a like a good candidate for an Abstract Class.

4.3.3 The Employee Abstract Class

Let's abstract the name and bank account functionality into an `Employee` abstract class. `SalaryEmployee` and `CommissionEmployee` can then extend this abstract class and provide different implementations for `grossPayment()`.

4.3.3.1 Class Diagram

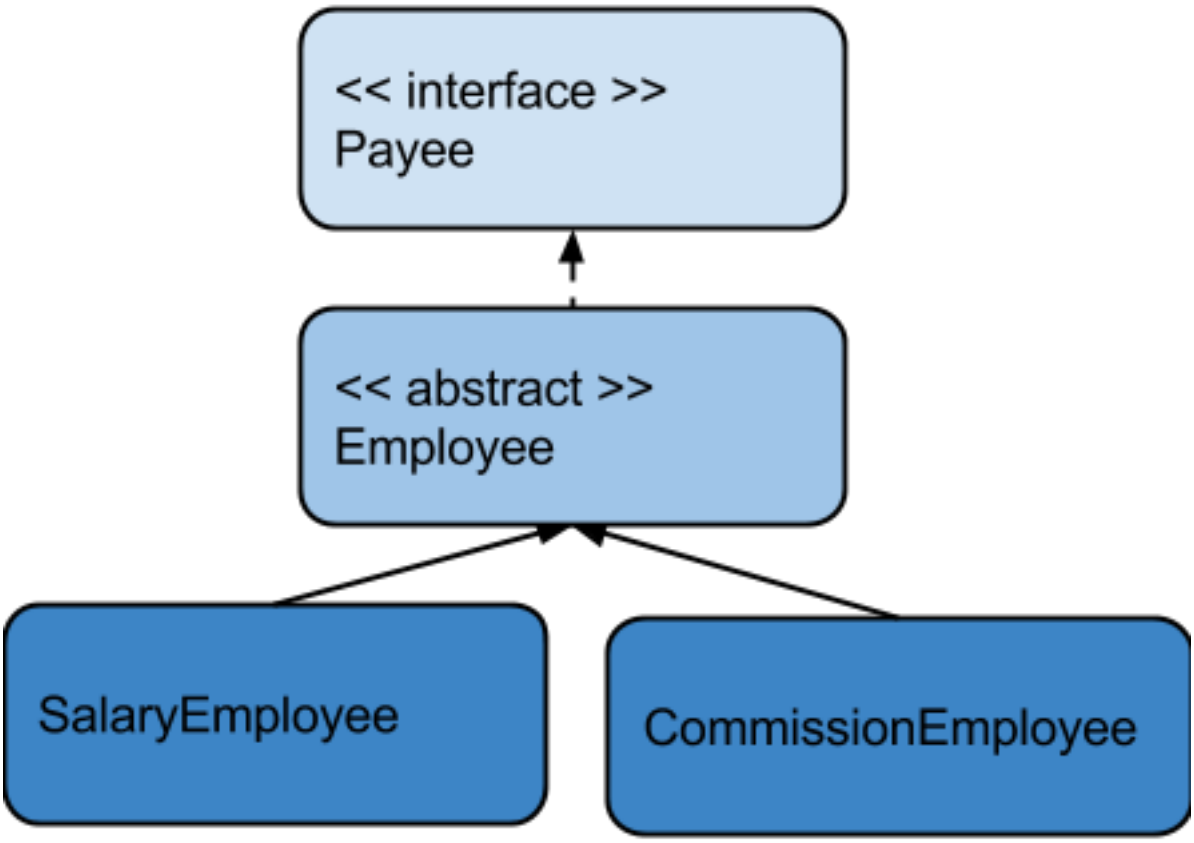

Figure 4.3: screenshot

4.3.3.2 Code

```
public abstract class Employee implements Payee {

        private String name;
        private Integer bankAccount;
        protected Double grossWage;
```

```
        public Employee(String name, Integer bankAccount, Double grossWage) {
                this.name = name;
                this.bankAccount = bankAccount;
                this.grossWage = grossWage;
        }

        public String name() {
                return name;
        }

        public Integer bankAccount() {
                return bankAccount;
        }
}
```

Note that Employee doesn't have to implement the `grossPayment()` method defined in `Payee` because `Employee` is abstract.

Now let's rewrite the two Employee classes:

```
public class SalaryEmployee extends Employee {

        public SalaryEmployee(String name, Integer bankAccount, Double grossWage) {
                super(name, bankAccount, grossWage);
        }

        public Double grossPayment() {
                return grossWage;
        }
}

public class CommissionEmployee extends Employee {

        private Double grossCommission = 0.0;

        public CommissionEmployee(String name, Integer bankAccount, Double grossWage) {
                super(name, bankAccount, grossWage);
        }

        public Double grossPayment() {
                return grossWage + doCurrentCommission();
        }

        private Double doCurrentCommission() {
                Double commission = grossCommission;
                grossCommission = 0.0;
                return commission;
        }

        public void giveCommission(Double amount) {
                grossCommission += amount;
        }
}
```

Much neater!

4.4 The Application

Let's try using our new `Employee` types in an application. We'll create an application class that initializes the system by creating a Payment System, some Employees and simulating a week of work.

4.4.1 The PaymentApplication Class

The application class looks like this:

```
public class PaymentApplication {

        public static void main(final String... args) {
                // Initialization
                PaymentSystem paymentSystem = new PaymentSystem();

                CommissionEmployee johnSmith = new CommissionEmployee("John Smith", 1111,  ↩
                    300.0);
                paymentSystem.addPayee(johnSmith);

                CommissionEmployee paulJones = new CommissionEmployee("Paul Jones", 2222,  ↩
                    350.0);
                paymentSystem.addPayee(paulJones);

                SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 3333, 500.0);
                paymentSystem.addPayee(maryBrown);

                SalaryEmployee susanWhite = new SalaryEmployee("Susan White", 4444, 470.0);
                paymentSystem.addPayee(susanWhite);

                // Simulate Week
                johnSmith.giveCommission(40.0);
                johnSmith.giveCommission(35.0);
                johnSmith.giveCommission(45.0);

                paulJones.giveCommission(45.0);
                paulJones.giveCommission(51.0);
                paulJones.giveCommission(23.0);
                paulJones.giveCommission(14.5);
                paulJones.giveCommission(57.3);

                // Process Weekly Payment
                paymentSystem.processPayments();
        }
```

We create two commissioned employees and two salaried employees, each with their own names, base wages and bank account numbers. We load each of the employees into the Payment System we created. We then simulate a week by giving commissions to the two commissioned employees and then ask the Payment System to process all the payments for the week.

Output:

```
Paying to John Smith
    Gross    420.0
    Transferred to Account: 1111
Paying to Paul Jones
    Gross    540.8
    Transferred to Account: 2222
Paying to Mary Brown
    Gross    500.0
    Transferred to Account: 3333
Paying to Susan White
    Gross    470.0
    Transferred to Account: 4444
```

4.5 Handling Bonuses

The boss is very happy with the system so far, but he's told us that in order to motivate his employees he wants the ability to give them weekly percentage bonuses. He tells us that because Commissioned Employees are typically on a lower salary we should give them a bonus multiplier of 1.5x in order that their percentage bonus is bumped up. The bonus should be reflected in the gross payment of each employee.

4.5.1 The Employee Class for Bonuses

Let's add a field to employee to capture any bonuses that are given, a protected method to return and reset the bonus and an abstract method to give the bonus. The `doBonus()` method is protected so that it can be accessed by the concrete Employee classes. The `giveBonus` method is abstract because it will be implemented differently for Salaried and Commissioned Employees.

```
public abstract class Employee implements Payee {

        private String name;
        private Integer bankAccount;

        protected Double currentBonus;
        protected Double grossWage;

        public Employee(String name, Integer bankAccount, Double grossWage) {
                this.name = name;
                this.bankAccount = bankAccount;
                this.grossWage = grossWage;
                currentBonus = 0.0;
        }

        public String name() {
                return name;
        }

        public Integer bankAccount() {
                return bankAccount;
        }

        public abstract void giveBonus(Double percentage);

        protected Double doCurrentBonus() {
                Double bonus = currentBonus;
                currentBonus = 0.0;
                return bonus;
        }
}
```

Updates to SalaryEmployee:

```
public class SalaryEmployee extends Employee {

        public SalaryEmployee(String name, Integer bankAccount, Double grossWage) {
                super(name, bankAccount, grossWage);
        }

        public void giveBonus(Double percentage) {
                currentBonus += grossWage * (percentage/100.0);
        }

        public Double grossPayment() {
                return grossWage + doCurrentBonus();
        }
```

```
}
```

Updates to CommissionEmployee:

```
public class CommissionEmployee extends Employee {

        private static final Double bonusMultiplier = 1.5;

        private Double grossCommission = 0.0;

        public CommissionEmployee(String name, Integer bankAccount, Double grossWage) {
                super(name, bankAccount, grossWage);
        }

        public void giveBonus(Double percentage) {
                currentBonus += grossWage * (percentage/100.0) * bonusMultiplier;
        }

        public Double grossPayment() {
                return grossWage + doCurrentBonus() + doCurrentCommission();
        }

        private Double doCurrentCommission() {
                Double commission = grossCommission;
                grossCommission = 0.0;
                return commission;
        }

        public void giveCommission(Double amount) {
                grossCommission += amount;
        }
}
```

We can see that both classes implement `giveBonus()` differently - `CommissionEmployee` uses the bonus multiplier. We can also see that both classes use the protected `doCurrentBonus()` method defined in `Employee` when working out the gross payment.

Let's update our application to simulate paying some weekly bonuses to our Employees:

```
public class PaymentApplication {

        public static void main(final String... args) {
                // Initialization
                PaymentSystem paymentSystem = new PaymentSystemV1();

                CommissionEmployee johnSmith = new CommissionEmployee("John Smith", 1111, ↩
                    300.0);
                paymentSystem.addPayee(johnSmith);

                CommissionEmployee paulJones = new CommissionEmployee("Paul Jones", 2222, ↩
                    350.0);
                paymentSystem.addPayee(paulJones);

                SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 3333, 500.0);
                paymentSystem.addPayee(maryBrown);

                SalaryEmployee susanWhite = new SalaryEmployee("Susan White", 4444, 470.0);
                paymentSystem.addPayee(susanWhite);

                // Simulate Week
                johnSmith.giveCommission(40.0);
                        johnSmith.giveCommission(35.0);
```

```
                        johnSmith.giveCommission(45.0);
                        johnSmith.giveBonus(5.0);

                        paulJones.giveCommission(45.0);
                        paulJones.giveCommission(51.0);
                        paulJones.giveCommission(23.0);
                        paulJones.giveCommission(14.5);
                        paulJones.giveCommission(57.3);
                        paulJones.giveBonus(6.5);

                        maryBrown.giveBonus(3.0);

                        susanWhite.giveBonus(7.5);

                // Process Weekly Payment
                paymentSystem.processPayments();
        }
}
```

Output:

```
Paying to John Smith
    Gross    442.5
    Transferred to Account: 1111
Paying to Paul Jones
    Gross    574.925
    Transferred to Account: 2222
Paying to Mary Brown
    Gross    515.0
    Transferred to Account: 3333
Paying to Susan White
    Gross    505.25
    Transferred to Account: 4444
```

The Gross amounts now reflect the bonuses paid to the employees.

4.6 Contracting Companies

The boss is delighted with the Payment System, however he's thought of someone else he needs to pay. From time to time he will engage the services of a contracting company. Obviously these companies don't have wages and aren't paid bonuses. They can get several one off payments and when processed by the payment system should be paid all cumulative payments and have their balance cleared.

4.6.1 The ContractingCompany Class

The `ContractingCompany` class should implement `Payee` so it can be used by the Payments System. It should have a method for paying for services which will be tracked by a cumulative total and used for payments.

4.6.1.1 Class Diagram

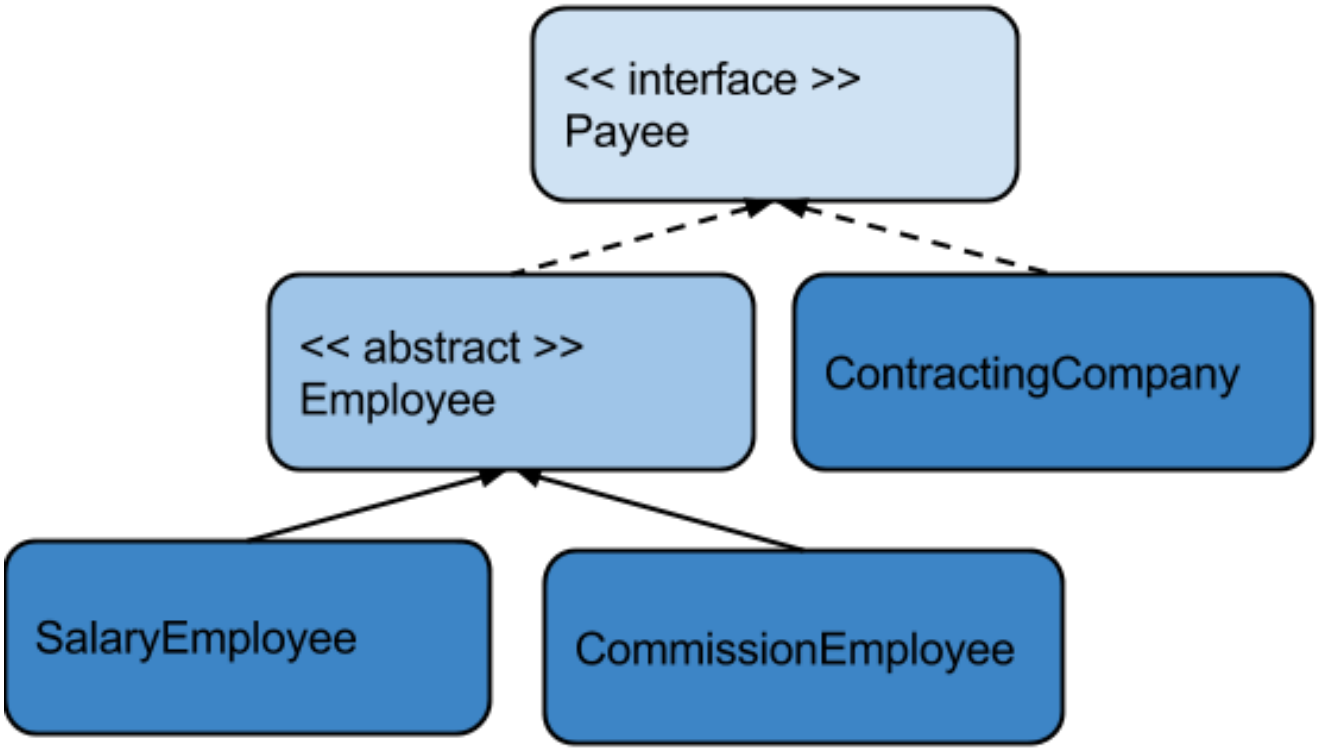

Figure 4.4: screenshot

4.6.1.2 Code

```
public class ContractingCompany implements Payee {

        private String name;
        private Integer bankAccount;
        private Double currentBalance;

        public ContractingCompany(String name, Integer bankAccount) {
                this.name = name;
                this.bankAccount = bankAccount;
                currentBalance = 0.0;
        }

        public String name() {
                return name;
        }

        public Double grossPayment() {
                return doPayment();
        }

        private Double doPayment() {
                Double balance = currentBalance;
                currentBalance = 0.0;
                return balance;
        }
```

```
        public Integer bankAccount() {
                return bankAccount;
        }

        public void payForServices(Double amount) {
                currentBalance += amount;
        }
}
```

Lets now add a couple of contracting companies to our Payment Application and simulate some payments to them:

```
public class PaymentApplication {

        public static void main(final String... args) {
        // Initialization
        PaymentSystem paymentSystem = new PaymentSystemV1();

        CommissionEmployee johnSmith = new CommissionEmployee("John Smith", 1111, 300.0, ↩
            100.0);
        paymentSystem.addPayee(johnSmith);

        CommissionEmployee paulJones = new CommissionEmployee("Paul Jones", 2222, 350.0, ↩
            125.0);
        paymentSystem.addPayee(paulJones);

        SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 3333, 500.0, 110.0);
        paymentSystem.addPayee(maryBrown);

        SalaryEmployee susanWhite = new SalaryEmployee("Susan White", 4444, 470.0, 130.0);
        paymentSystem.addPayee(susanWhite);

        ContractingCompany acmeInc = new ContractingCompany("Acme Inc", 5555);
        paymentSystem.addPayee(acmeInc);

        ContractingCompany javaCodeGeeks = new ContractingCompany("javacodegeeks.com", ↩
            6666);
        paymentSystem.addPayee(javaCodeGeeks);

        // Simulate Week
        johnSmith.giveCommission(40.0);
        johnSmith.giveCommission(35.0);
        johnSmith.giveCommission(45.0);
        johnSmith.giveBonus(5.0);

        paulJones.giveCommission(45.0);
        paulJones.giveCommission(51.0);
        paulJones.giveCommission(23.0);
        paulJones.giveCommission(14.5);
        paulJones.giveCommission(57.3);
        paulJones.giveBonus(6.5);

        maryBrown.giveBonus(3.0);

        susanWhite.giveBonus(7.5);

        acmeInc.payForServices(100.0);
        acmeInc.payForServices(250.0);
        acmeInc.payForServices(300.0);

        javaCodeGeeks.payForServices(400.0);
        javaCodeGeeks.payForServices(250.0);
```

```
        // Process Weekly Payment
        paymentSystem.processPayments();
        }
```

Output:

```
Paying to John Smith
    Gross    442.5
    Transferred to Account: 1111
Paying to Paul Jones
    Gross    574.925
    Transferred to Account: 2222
Paying to Mary Brown
    Gross    515.0
    Transferred to Account: 3333
Paying to Susan White
    Gross    505.25
    Transferred to Account: 4444
Paying to Acme Inc
    Gross    650.0
    Transferred to Account: 5555
Paying to javacodegeeks.com
    Gross    650.0
    Transferred to Account: 6666
```

We have now successfully added a brand new type of `Payee` to the system without having to change one single line of code in the `PaymentSystem` class which handles Payees. This is the power of abstraction.

4.7 Advanced Functionality: Taxation

The boss is over the moon with the Payment System, however the taxman has sent him a letter telling him that he has to incorporate some sort of tax withholding into the system or he'll be in big trouble. There should be a global tax rate for the system and each employee should have a personal tax free allowance. The tax should only be collected on any payments made to an employee over and above the tax free allowance. The employee should then only be paid the net amount of payment due to them. Contracting Companies are responsible for paying their own tax so the system should not withhold tax from them.

In order to handle taxation we'll need to create a new, special, type of Payee which is taxable and can provide a tax free allowance figure.

In Java we can extend interfaces. This let's us add behaviour definitions without modifying our original interface. All of the behaviour defined in the original interface will be automatically carried into the new interface.

4.7.1 The TaxablePayee Interface

We'll extend `Payee` to create a new `TaxablePayee` interface, we'll then have `Employee` implement that interface, while letting `ContractingCompany` remain as a regular, untaxed, `Payee`.

4.7.1.1 Class Diagram

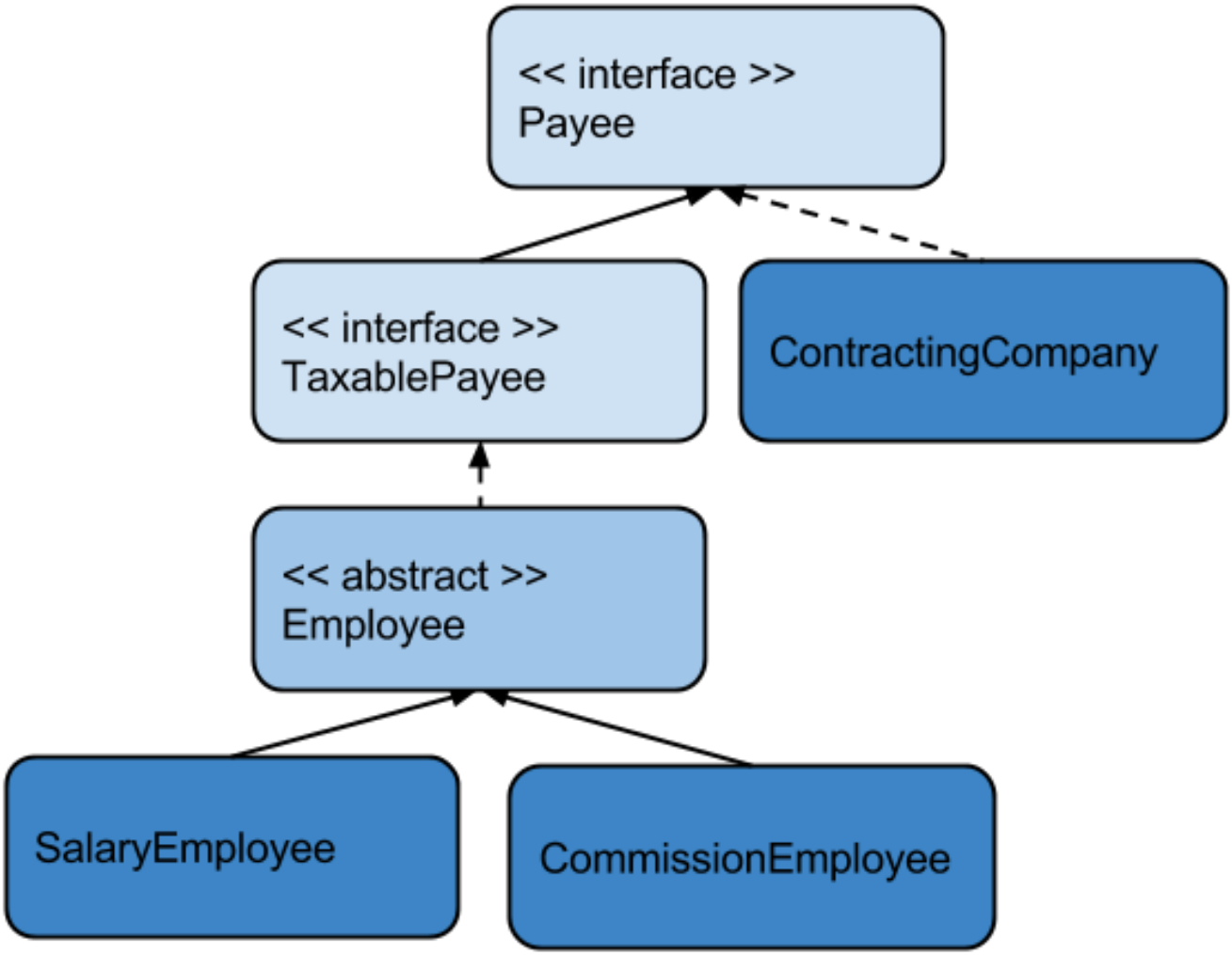

Figure 4.5: screenshot

4.7.1.2 Code

```
public interface TaxablePayee extends Payee {

        public Double allowance();

}
```

Here we see that `Payee` is extended and an additional method is defined - `allowance()` which returns the tax free allowance of the `TaxablePayee`.

We'll now need to update `Employee` to implement `TaxablePayee` and we will see that this will have a knock on effect on both concrete `Employee` classes.

```
public abstract class Employee implements TaxablePayee {

        private String name;
        private Integer bankAccount;
        private Double allowance;
```

```
        protected Double currentBonus;
        protected Double grossWage;

        public Employee(String name, Integer bankAccount, Double grossWage, Double ↩
            allowance) {
        this.name = name;
        this.bankAccount = bankAccount;
        this.grossWage = grossWage;
        this.allowance = allowance;
        currentBonus = 0.0;
        }

        public String name() {
        return name;
        }

        public Integer bankAccount() {
                return bankAccount;
        }

        public Double allowance() {
                return allowance;
        }

        public abstract void giveBonus(Double percentage);

        protected Double doCurrentBonus() {
                Double bonus = currentBonus;
                currentBonus = 0.0;
                return bonus;
        }

}
```

We have had to change the constructor of Employee which implies that the constructors of our two abstract classes will also need to change.

```
public class SalaryEmployee extends Employee {

        public SalaryEmployee(String name, Integer bankAccount, Double grossWage, Double ↩
            allowance) {
                super(name, bankAccount, grossWage, allowance);
        }

        @Override
        public void giveBonus(Double percentage) {
                currentBonus += grossWage * (percentage/100.0);
        }

        @Override
        public Double grossPayment() {
                return grossWage + doCurrentBonus();
        }
}
```

CommissionEmployee:

```
public class CommissionEmployee extends Employee {

        private static final Double bonusMultiplier = 1.5;
```

```
        private Double grossCommission = 0.0;

        public CommissionEmployee(String name, Integer bankAccount, Double grossWage,  ↩
            Double allowance) {
                super(name, bankAccount, grossWage, allowance);
        }

        @Override
        public void giveBonus(Double percentage) {
                currentBonus += grossWage * (percentage/100.0) * bonusMultiplier;
        }

        @Override
        public Double grossPayment() {
                return grossWage + doCurrentBonus() + doCurrentCommission();
        }

        private Double doCurrentCommission() {
                Double commission = grossCommission;
                grossCommission = 0.0;
                return commission;
        }

        public void giveCommission(Double amount) {
                grossCommission += amount;
        }
}
```

4.7.2 Taxation changes in PaymentSystem

We now need to update `PaymentSystem` to withhold tax and to do this we'll need some way of determining if a given `Payee` is a `TaxablePayee` or a regular `Payee`. We can take advantage of a Java keyword called `instanceof` to do this.

`Instanceof` lets us check to see if a given reference variable's runtime type matches a test type. It returns a `boolean` and can be called like so: if (`object1 instanceof MyClass`). This will return true if object1's runtime type is `MyClass` or a subclass or implementing class (if `MyClass` is an interface) of `MyClass`. This means we can test anywhere along the inheritance tree making it a very powerful tool. We can use this tool to determine if a given `Payee` is an instance of a `TaxablePayee` and take appropriate measures based on that knowledge.

We now update the `PaymentSystem` as follows:

```
public class PaymentSystem {

        private List<Payee> payees;
        private Double taxRate = 0.2;

        public PaymentSystem() {
                payees = new ArrayList<>();
        }

        public void addPayee(Payee payee) {
                if (!payees.contains(payee)) {
                        payees.add(payee);
                }
        }

        public void processPayments() {
                for (Payee payee : payees) {
                                Double grossPayment = payee.grossPayment();
                                Double tax = 0.0;
                                if (payee instanceof TaxablePayee) {
```

```
                                Double taxableIncome = grossPayment - ((TaxablePayee)payee) ↩
                                    .allowance();
                                tax = taxableIncome * taxRate;
                                }
                                Double netPayment = grossPayment - tax;

                            System.out.println("Paying to " + payee.name());
                            System.out.println("tGrosst" + grossPayment);
                            System.out.println("tTaxtt-" + tax);
                            System.out.println("tNettt" + netPayment);
                            System.out.println("tTransferred to Account: " + payee.bankAccount ↩
                                ());
                    }
            }
}
```

The new code first checks if the `Payee` being processed is an instance of `TaxablePayee`, if it is it then does a **cast** on the `Payee` to treat it as a reference to a `TaxablePayee` for the purposes of accessing the `allowance()` method defined in `TaxablePayee`. Remember; if the reference stayed as a `Payee` the `allowance()` method would not be visible to the `PaymentSystem` because it is defined in `TaxablePayee`, not `Payee`. The cast is safe here because we have already confirmed that the `Payee` is an instance of `TaxablePayee`. Now that `PaymentSystem` has the allowance it can work out the taxable amount and the tax to be withheld based on the global tax rate of 20%.

If the `Payee` is not a `TaxablePayee` the system continues to process them as normal, not doing anything related to tax processing.

Let's update our application class to give our employees different tax free allowances and execute the application again:

```
public class PaymentApplication {

        public static void main(final String... args) {
        // Initialization
        PaymentSystem paymentSystem = new PaymentSystemV1();

        CommissionEmployee johnSmith = new CommissionEmployee("John Smith", 1111, 300.0,  ↩
            100.0);
        paymentSystem.addPayee(johnSmith);

        CommissionEmployee paulJones = new CommissionEmployee("Paul Jones", 2222, 350.0,  ↩
            125.0);
        paymentSystem.addPayee(paulJones);

        SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 3333, 500.0, 110.0);
        paymentSystem.addPayee(maryBrown);

        SalaryEmployee susanWhite = new SalaryEmployee("Susan White", 4444, 470.0, 130.0);
        paymentSystem.addPayee(susanWhite);

        ContractingCompany acmeInc = new ContractingCompany("Acme Inc", 5555);
        paymentSystem.addPayee(acmeInc);

        ContractingCompany javaCodeGeeks = new ContractingCompany("javacodegeeks.com",  ↩
            6666);
        paymentSystem.addPayee(javaCodeGeeks);

        // Simulate Week
        johnSmith.giveCommission(40.0);
        johnSmith.giveCommission(35.0);
        johnSmith.giveCommission(45.0);
        johnSmith.giveBonus(5.0);

        paulJones.giveCommission(45.0);
```

```
        paulJones.giveCommission(51.0);
        paulJones.giveCommission(23.0);
        paulJones.giveCommission(14.5);
        paulJones.giveCommission(57.3);
        paulJones.giveBonus(6.5);

        maryBrown.giveBonus(3.0);

        susanWhite.giveBonus(7.5);

        acmeInc.payForServices(100.0);
        acmeInc.payForServices(250.0);
        acmeInc.payForServices(300.0);

        javaCodeGeeks.payForServices(400.0);
        javaCodeGeeks.payForServices(250.0);

        // Process Weekly Payment
        paymentSystem.processPayments();
        }
}
```

Output:

```
Paying to John Smith
    Gross       442.5
    Tax         -68.5
    Net          374.0
    Transferred to Account: 1111
Paying to Paul Jones
    Gross       574.925
    Tax          -89.985
    Net          484.93999999999994
    Transferred to Account: 2222
Paying to Mary Brown
    Gross       515.0
    Tax          -81.0
    Net          434.0
    Transferred to Account: 3333
Paying to Susan White
    Gross       505.25
    Tax          -75.05
    Net          430.2
    Transferred to Account: 4444
Paying to Acme Inc
    Gross       650.0
    Tax          -0.0
    Net          650.0
    Transferred to Account: 5555
Paying to javacodegeeks.com
    Gross       650.0
    Tax          -0.0
    Net          650.0
    Transferred to Account: 6666
```

As you can see tax is calculated and withheld for Employees, while Contracting Companies continue to pay no tax. The boss couldn't be happier with the system and is going to gift us an all expenses paid vacation for keeping him out of jail!

Chapter 5

Conclusion

Abstraction is a very powerful tool in Java when used in the correct way. It opens up many possibilities for advanced java use and for building complex, extensible and maintainable software. We have just scratched the surface of what abstraction can do for us and hopefully built a foundation for learning about the different ways abstraction can be used in more detail.