

Topic: Abstract Classes.

OOP concepts involved: Inheritance, Abstract Classes, Class Instantiation and Declaration, Class Members, Polymorphism, Constructor Methods, Class Modifiers, Getters and Setters.

Programming generic concepts involved: Access Modifiers, Functions, Primitive Variables, Variable Declaration, Function Parameters.

➤ Theoric introduction

WHAT IS ABSTRACTION?

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces. Abstraction is one of the four major concepts behind object-oriented programming (OOP).

ABSTRACT CLASSES

An abstract class is a class that is declared *abstract* (it may or may not include abstract methods). Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY); //Abstract method which  
receives 2 parameters of type double and returns void
```

If a class includes abstract methods, then the class itself must be declared *abstract*, as in:

```
public abstract class GraphicObject {  
    // Members Variables (Attributes/Fields) declaration  
    // Non-Abstract methods declaration
```

```
    abstract void draw();  
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared *abstract*.

It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

WHY WOULD I NEED ABSTRACT CLASSES?

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method, because everyone of the subclasses need to have the same behavior but with a different implementation.

ABSTRACT CLASSES COMPARED TO INTERFACES

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

❑ Consider using abstract classes if any of these statements apply to your situation:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

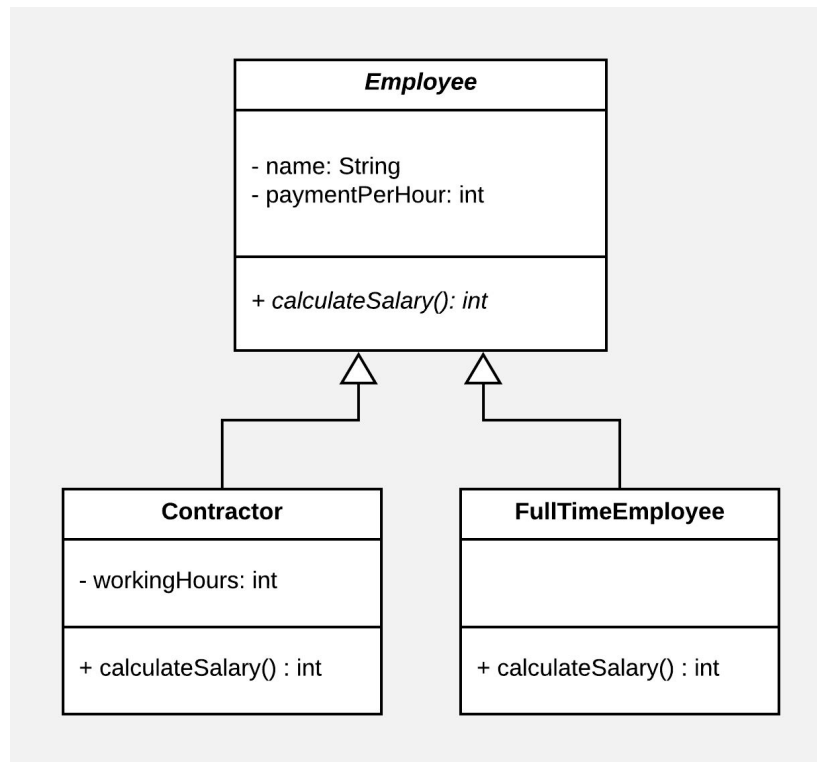
❑ Consider using interfaces if any of these statements apply to your situation:

- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

➤ **Statement**

Make use of the principle of abstraction and create a simple class relationship between a parent Employee Class, a child Contractor Class and a child FullTimeEmployee Class. Make sure of using Abstract Classes.

➤ Class design (UML)



➤ Program Code

Employee.java

```
public abstract class Employee {
    // Declaring private attributes
    private String name;
    private int paymentPerHour;

    // Constructor Methods
    public Employee(String name, int paymentPerHour) {
        this.name = name;
        this.paymentPerHour = paymentPerHour;
    }

    // abstract method (without implementation)
    public abstract int calculateSalary();

    // Setters and getters
    public String getName() {
```

```

        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPaymentPerHour() {
        return paymentPerHour;
    }

    public void setPaymentPerHour(int paymentPerHour) {
        this.paymentPerHour = paymentPerHour;
    }
}

```

Contractor.java

```

public class Contractor extends Employee{
    // Declaring attributes
    private int workingHours;

    // Constructor Methods
    public Contractor(String name, int paymentPerHour, int workingHours) {
        super(name, paymentPerHour);
        this.workingHours = workingHours;
    }

    // Overriding parent abstract method
    @Override
    public int calculateSalary() {
        return getPaymentPerHour() * workingHours;
    }

    // Getters and setters of the attribute belong only to the Contractor Class
    public int getWorkingHours() {
        return workingHours;
    }

    public void setWorkingHours(int workingHours) {
        this.workingHours = workingHours;
    }
}

```

FullTimeEmployee.java

```
public class FullTimeEmployee extends Employee {
    // Constructor method
    public FullTimeEmployee(String name, int paymentPerHour) {
        super(name, paymentPerHour);
    }

    // Overriding parent abstract method, writing our own implementation
    @Override
    public int calculateSalary() {
        return getPaymentPerHour() * 8;
    }
}
```

Main.java

```
public class Main {

    public static void main(String[] args) {
        // Instantiating Contractor and FullTimeEmployee Object, passing
        // values to the constructor
        Contractor contractor = new Contractor("Joe Strace", 50, 6);
        FullTimeEmployee fullEmployee = new FullTimeEmployee("Riley Mascot",
13);

        System.out.println(contractor.getName() + " is a recognized Contractor
that has a daily salary of " + contractor.calculateSalary() +
" by working " + contractor.getWorkingHours() + " hours.");

        System.out.println(fullEmployee.getName() + " is a full time employee
that has a daily salary of " + fullEmployee.calculateSalary() +
" by working a complete 8 hours shift.");

    }

}
```

➤ Program execution

```
Joe Strace is a recognized Contractor that has a daily salary of 300 by working 6 hours.  
Riley Mascot is a full time employee that has a daily salary of 104 by working a complete 8 hour shift.
```

➤ Conclusions

It is convenient to use Abstract Classes when you want to have the same behavior on different subclasses using a different implementation within each one of them. It is a good practice due to the code recycling part, when you need to use the same members as the parent class inside your different subclasses.

Abstract methods only have the definition of the behavior of Class, in the other hand, subclasses provide the specific behavior implementation.