Abstract Classes and Interfaces: Practical Examples

Andrew Lalis — *a.lalis@student.rug.nl*
George Argyrousis — *g.argyrousis@rug.nl*

Last modified: May 2, 2018

1 Abstract Classes

1.1 Definition

Oracle Abstract Classes and Methods Tutorial

An abstract class, in the context of Java, is a superclass that cannot be instantiated and is used to state or define general characteristics. An object cannot be formed from a Java abstract class; trying to instantiate an abstract class only produces a compiler error. The abstract class is declared using the keyword abstract.

Subclasses extended from an abstract class have all the abstract class's attributes, in addition to attributes specific to each subclass. The abstract class states the class characteristics and methods for implementation, thus defining a whole interface.

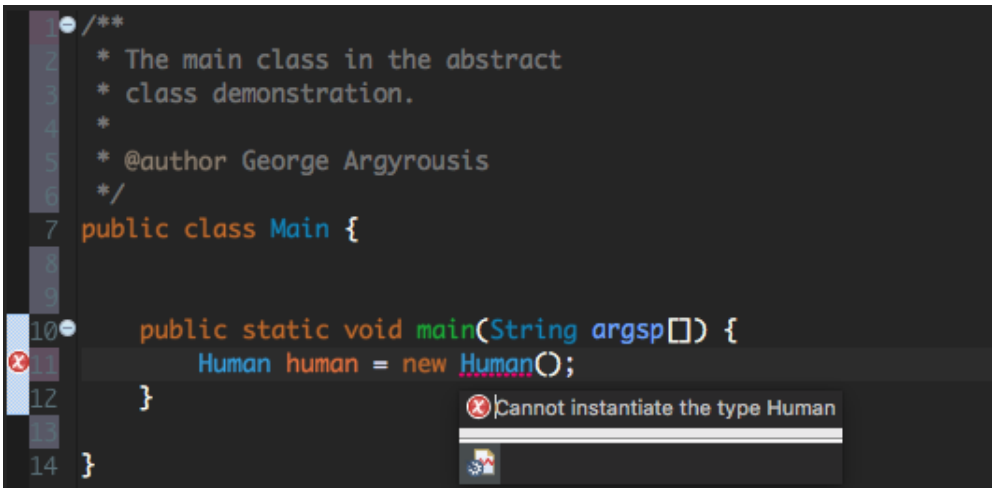
1.2 Example

We will illustrate the abstract class functionality by creating a small application that should act as a model, creating a Student database for a University. We will begin by defining an **abstract** Human class that reflects the idea of the Human characteristics that would be universal in the database.

```java

/**
 * Abstract class to illustrate
 * why we would extantiate an
 * abstract class over a regural
 * class.
 *
 * @author George Argyrousis
 */
public abstract class Human {

    /* The first Name of a person */
    private String firstName;

    /* The last Name of a person */
    private String lastName;

    /* The person's gender */
    private Gender gender;

    /* The date that the peson was born */
    private Date birthdate;

    /* Initialising all relevant components */
    public Human(String firstName, String lastName, Gender gender, Date birthdate) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = gender;
        this.birthdate = birthdate;
    }

    /* Method that needs to be implemented in the immedeate subclass */
    public abstract boolean isWorking();

    /* Simple getters. */

    public String getFirstName() { return this.firstName; }

    public String getLastName() { return this.lastName; }

    public String getGender() { return this.gender.getGender(); }

    public String getBirthDate() { return this.birthdate.toString(); }
}
```

We have defined attributes such as *firstName*, *lastName*, *gender* and *birthdate*. Representing common attributes among all humans. We have also defined one abstract method that will be implemented when the abstract class is extended by another regular class.

Of course we would not be able to create a **Human** object.



Human is an abstract class that has defined general characteristics of the objects we would want in our database. We could possibly have multiple classes such as, **Professor**, **Student**, **Employee** that extend **Human**.

Thus inheriting all information provided by the abstract class itself. But we wouldn't want to have Human objects in the database as it is only reflecting the core idea.

We will proceed my making the **Student** object which extends Human. A Student might have multiple other attributes but for this demonstration we will just add two ArrayLists and a StudentID.

```java
import java.util.ArrayList;
import java.util.Date;
/**
 * Student class extending
 * the abstract superclass Human.
 *
 * @author George Argyrousis
 */
public class Student extends Human{

    /* The student ID */
    private String studentID;

    /* A list of grades */
    private ArrayList<Double> grades;

    /* A list of attending courses */
    private ArrayList<String> attendingCourses;

    /* Initialise all relevant components */
    public Student(String firstName, String lastName, Gender gender, Date birthdate, String studentID) {
        super(firstName, lastName, gender, birthdate);
        this.studentID = studentID;
        grades = new ArrayList<Double>();
        attendingCourses = new ArrayList<String>();
    }

    @Override
    public boolean isWorking() {
        /* add extra functionality as you see fit */
        return false;
    }

    /* Simple getters */

    public String getStudentID() { return this.studentID; }

    public ArrayList<Double> getGrades(){ return this.grades; }

    public ArrayList<String> getAttendingCourses(){ return this.attendingCourses; }

}
```

Last but not least, we initialize the object and print it's attributes in the command line.

```java
import java.util.Date;

/**
 * The main class in the abstract
 * class demonstration.
 *
 * @author George Argyrousis
 */
public class Main {

    /* Student for this demonstration */
    private static Student george;

    /* Main class, instantiating the student */
    public static void main(String argsp[]) {
        george = new Student("George", "Argyrousis", Gender.MALE, new Date(), "S<number>");
        printStudentAttributes(george);
    }

    /* printing a student's attributes */
    private static void printStudentAttributes(Student student) {
        System.out.println("Student number : " + student.getStudentID());
        System.out.println("Student name : " + student.getFirstName() + " " + student.getLastName());
        System.out.println("Student was born : " + student.getBirthDate() + " and his gender is : " + student.getGender());
    }
}
```

```
Student number : S<number>
Student name : George Argyrousis
Student was born : Tue May 01 21:43:23 CEST 2018 and his gender is : Male
```

2 Interfaces

2.1 Definition

Oracle Java Interface Tutorial

In the most literal sense, an interface is a list of unimplemented methods that should all be related to each other.

When an object *implements* an interface, it promises that it will provide behavior for the methods that the interface defines.

This allows for very useful behavior: Two potentially unrelated classes may implement a common interface, allowing them to be interpreted simply as two arbitrary objects that both have some common methods, thanks to the interface.

For example, suppose you are making a video game and need your player to interact with things that have items, like a chest, another player, an NPC, or perhaps some stones or trees.

Without interfaces, we would have have each of these objects inherit from some superclass which defines how to take and give items to these objects. However, conceptually a player is not very related to a tree.

To avoid some unintuitive inheritance structure, we can instead introduce an interface which defines methods such as `takeItem(Item item)` and `giveItem(Item item)`.

With our new `Tradeable` interface, *or whatever you want to name it, just something descriptive*, each of these objects can now promise that it will implement these methods so that they can be called on a player, a chest, an NPC, or a stone.