2024-11-19

# Abstract Classes

## Motivation

Consider the following situation:

* We want to implement a class for students, and one for employees.
* We realize that those class overlap heavily: they both need properties for an id, a name, an emergency phone number, an address, etc., identical methods to e.g., implement an automated alert system, etc.
* However, they do not overlap perfectly: for example, students will have a major but employees won’t, and employee will have an hourly wage but students won’t. Also, some checks will be different: while both students and employees will have an id, the former will always start with the letter ‘S’, and the latter with the letter ‘E’.
* So we really do need two different classes, but would like for them both to inherit a “Person” class that implements all the overlapping properties, attributes and methods.
* **But** we **do not** want persons “objects” to be created: a “person” in isolation does not make sense in our model, we only want to implement students or employees, not “persons”.

The mechanism used to obtain this behavior (being able to inherit from a class while disallowing instantiating it) is achieved using the abstract keyword.

## Example

Consider a (shortened) version of the example above. We start by implementing an *abstract* Person class:

abstract class Person  
{  
 public string Name { get; set; }  
 public abstract string Id { set; }  
}

Note that the Id property is *also* marked as abstract: this means that the derived class will have to re-implement this property’s setter. Then, we can implement the Student and Employee classes by inheriting from the Person class:

using System;  
  
class Student : Person  
{  
 private string major;  
 public override string Id  
 {  
 set  
 {  
 if (value[0] != 'S')  
 throw new ArgumentException(  
 "A student ID must start with an 'S'."  
 );  
 }  
 }  
}

using System;  
  
class Employee : Person  
{  
 private decimal hourlyPay;  
 public override string Id  
 {  
 set  
 {  
 if (value[0] != 'E')  
 throw new ArgumentException(  
 "An employee ID must start with an 'E'."  
 );  
 }  
 }  
}

Using this code, the statement

Person test = new Person();

would return the error message “Cannot create an instance of the abstract type or interface ‘Person’”.

Furthermore, the following exemplifies the expected behavior:

﻿using System;  
  
class Program  
{  
 static void Main()  
 {  
 // Person test = new Person(); // Cannot create an instance of the abstract type or interface 'Person'  
 Employee Harley = new Employee();  
 Harley.Id = "E8190";  
  
 Student Morgan = new Student();  
 try  
 {  
 Morgan.Id = "E8194";  
 }  
 catch  
 {  
 Console.WriteLine(  
 "We cannot set the Id of a student to a string not starting with 'S'!"  
 );  
 }  
 Morgan.Id = "S8194";  
 }  
}

The statement Morgan.Id = "E8194"; will raise exception, but Morgan.Id = "S8194"; will execute without throwing an error.

## Additional Details: Abstract Properties and Methods

* As we’ve seen above with the Id property, not only classes can be marked as abstract.
* For abstract properties, using {get; set;}, only {get;} or only {set;} indicates if the derived class needs to implement both a setter and a getter, or only one of them.
* In addition to properties, *methods* can also be marked as abstract: in that case, their body need to be absent (not simply empty: missing).
  + For example, the Person class could also contain
  + public abstract string GenerateLogin();
  + to “force” any derived class to implement a GenerateLogin method that does not take any parameter and returns a string. The derived classes would need to implement a method that overrides the Person’s GenerateLogin method:
  + public override string GenerateLogin(){  
     // Insert method body.  
    }
* However, abstract attributes are not allowed.

## UML Class Diagram Representation

* An abstract class is represented by as a class with its name prefixed by <<Abstract>>, «Abstract», or with its name displayed in *italics*,
* An abstract method or property is represented as a usual, except that it is displayed in *italics*.
* Since, for example, Person’s GenerateLogin() method is to be overridden (it *has* to be, actually, since it is abstract), it is indicated again in the Student and Employee classes: this indicates that those method override the one they have inherited from the Person class.

A UML diagram for the Person ⇽ Student class ([text version](https:/princomp.github.io/uml/cla/Person.txt))