Abstract Classes and Methods

In this lesson, you'll get to know about abstract classes and methods.

WE'LL COVER THE FOLLOWING

Abstract Methods

> An **abstract method** is declared using the `abstract` keyword and does not have a body/ implementation.

There are certain rules we should follow when working with abstract methods. Let's have a look at these rules.

Rules to be Followed

- In contrast to a non-abstract/normal C# method, an *abstract method* does not have a body/definition, i.e., it only has a declaration or method signature.
- An *abstract method* can be declared inside an *abstract class* or an *interface* only, more on these later.
- To contain any *abstract method* in its implementation, a class has to be declared as an *abstract class*. *Non-abstract classes* **cannot** have abstract

methods.

- An *abstract method* **cannot** be declared with the `private` or `sealed` modifier as it has to be implemented in some other class.
- *Abstract methods* are implicitly `virtual` so we cannot use the keyword `virtual` in their declaration.

> Just like abstract methods, an **abstract property** declaration does not provide an implementation of the property accessors, i.e., `get` and `set` blocks. It declares that the class supports properties but leaves the accessor implementation to derived classes.

Declaration

Let's move on to the *syntax* part. Syntactically, the generalized declaration of an abstract method is as follows:

```
<accessModifier> abstract <retunType> MethodName();
//Example:
public abstract double GetPrice();
```

An abstract method's declaration has:

1. An *access modifier*
2. The keyword `abstract`
3. A `return` type
4. A name of the method
5. The parameter(s) to be passed
6. A semicolon, `;`, to end the declaration

At this point, one may raise a question about the definition or the body of an abstract method: *"Where do we implement the body of an abstract method?"* or *"If abstract methods have no implementation, why are they even declared?*

The upcoming topics will address the above question.

Abstract Class

> An **abstract class** is a class which is declared using the `abstract` keyword and cannot be instantiated, i.e., one cannot create its objects straightaway.

Rules to be Followed

- An *abstract class* ***cannot*** be instantiated i.e. one cannot create an object of an *abstract class*.
- An *abstract class* can have the declaration of *abstract method(s)* (as an abstract method's body cannot be implemented in an abstract class) and *abstract properties* (more on these in the next lesson) but it is not compulsory to have any.
- Non-abstract/normal methods can also be implemented in an *abstract class*.
- To use the members of an *abstract class*, it needs to be **inherited**.
- The class which *inherits* from the *abstract class* **must** implement all the abstract members declared in the *parent abstract class*.
- An *abstract class* can have everything else , i.e. constructors, static variables and methods, the same way as a normal C# class has them.
- An *abstract class* cannot be declared `sealed` or `static` as it becomes useless until some class inherits from it and implements its abstract methods.
- An *abstract class* can inherit from both abstract and non-abstract classes.
- A non-abstract class can inherit from only **one** abstract class.

Declaration

Talking about the syntax, the *declaration* of an `abstract` class in C# is as follows:

```
abstract class ClassName {
  // abstract member(s) here
}
```

Implementation

Abstraction has already been discussed in the previous lesson. Abstract classes are used to achieve abstraction in C#.

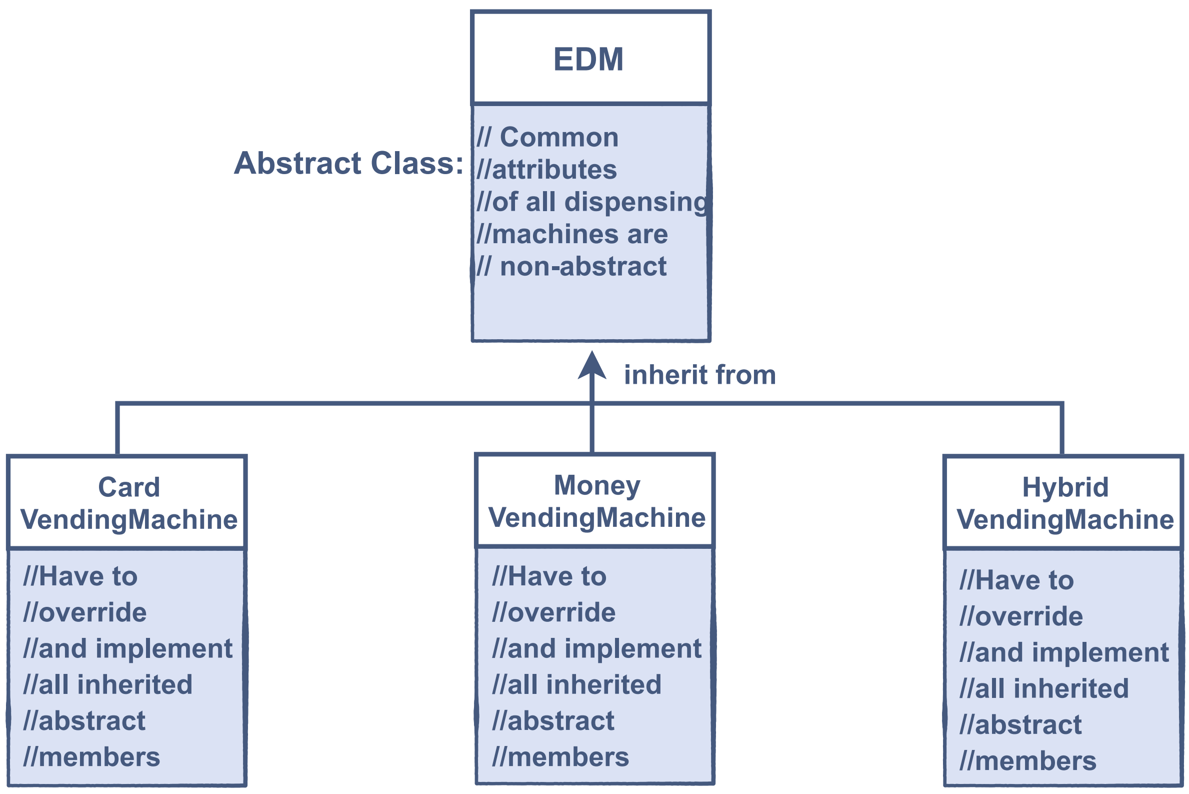
Consider modeling a Vending Machine using C# having:

- A base `abstract` class named `EDM` (Electronic dispensing machine).
- A child class named `CardVendingMachine`
- A child class named `MoneyVendingMachine`
- A child class named `HybridVendingMachine`

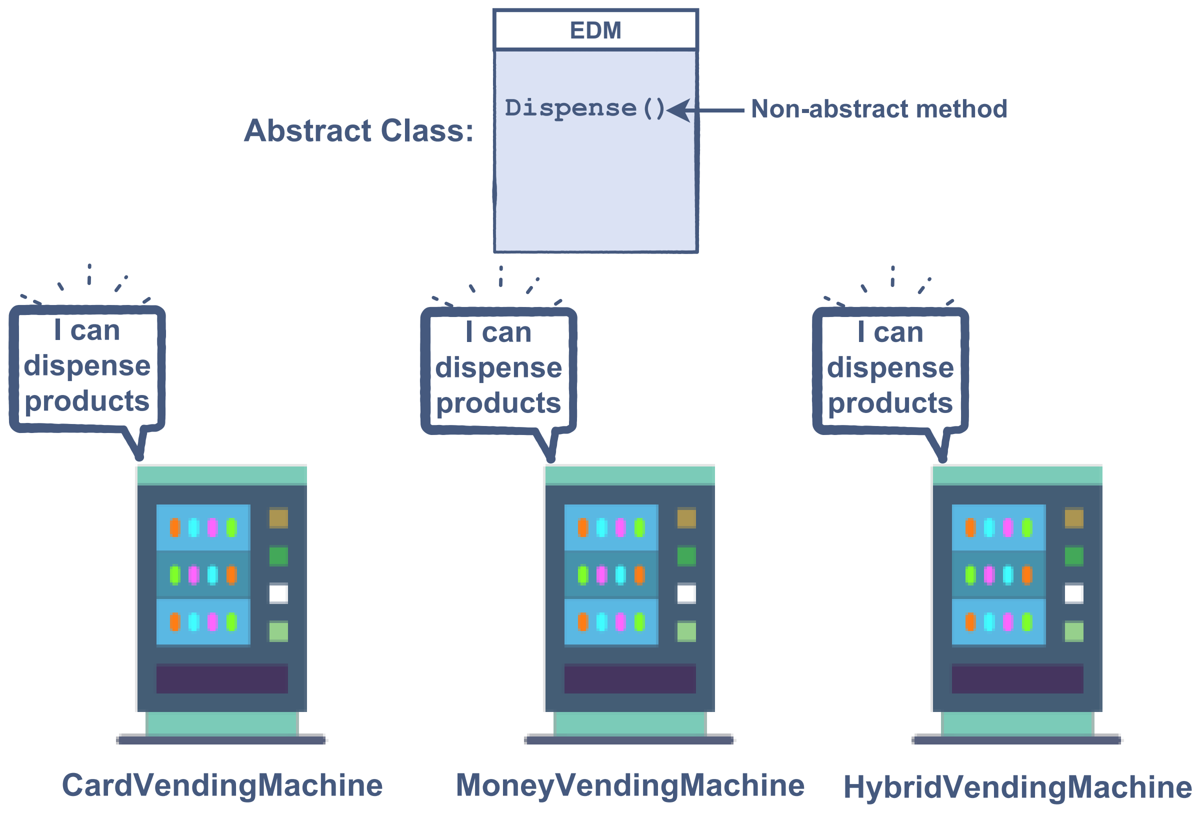
We have already discussed that abstract classes cannot be instantiated, so even if we want to initiate our base class named `EDM`, it's of no use because the electronic dispensing machine is a generalized category. But what this base abstract class can do is add all the common attributes of this category of machines in its implementation by declaring these attributes non-abstract. This way, the derived classes don't have to override these in their implementation. After all, what's the point in repeating the code for a method that is being performed in the same fashion in every type of dispensing machine? The `dispense` method is an example of such a method; every dispensing machine will have a dispenser.

Now let's talk about the part where implementation varies in every sub-category of the machines. We can think of it as the transaction process of these machines. Some machines accept cards while others don't, and some have the acceptance for both cards and cash. So, the `transact()` method **must** be there in any kind of dispensing machine but its implementation should be overridden according to each specific category. That's the reason we should add it as an `abstract` method in the `EDM` class.

Let's have a look at the following illustration to get a clearer understanding:

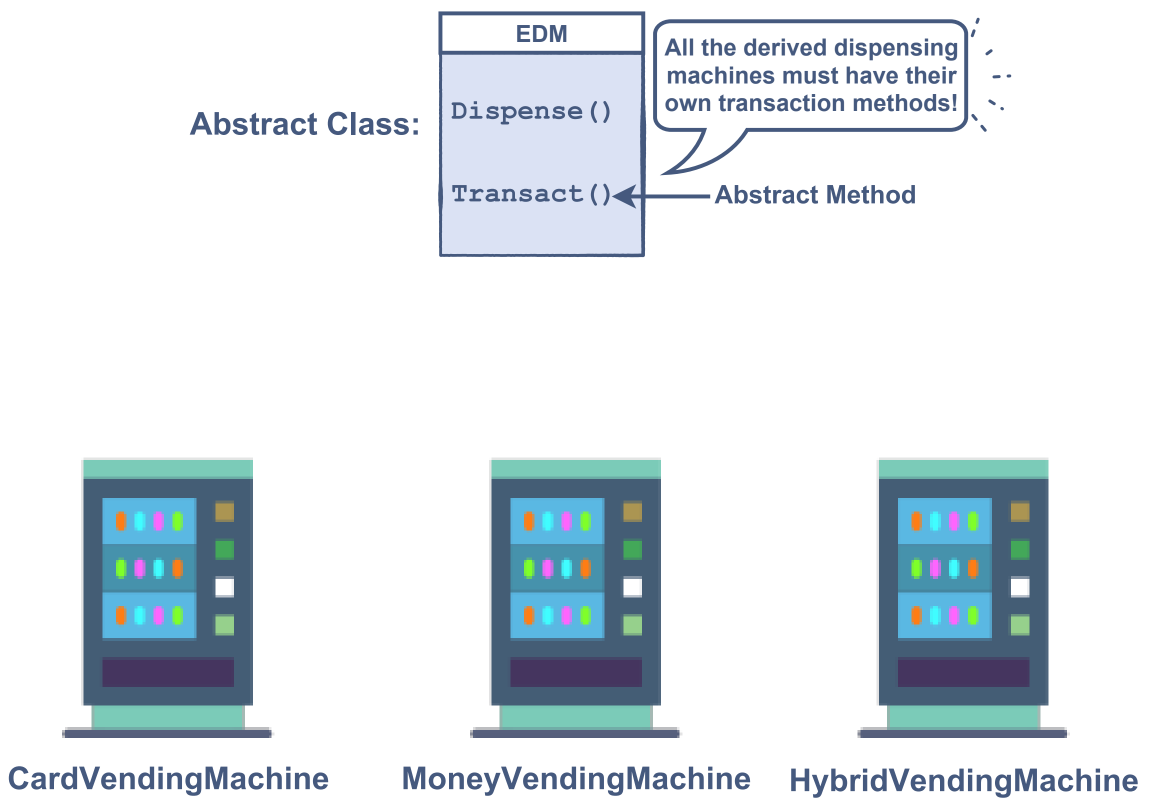


1 of 4

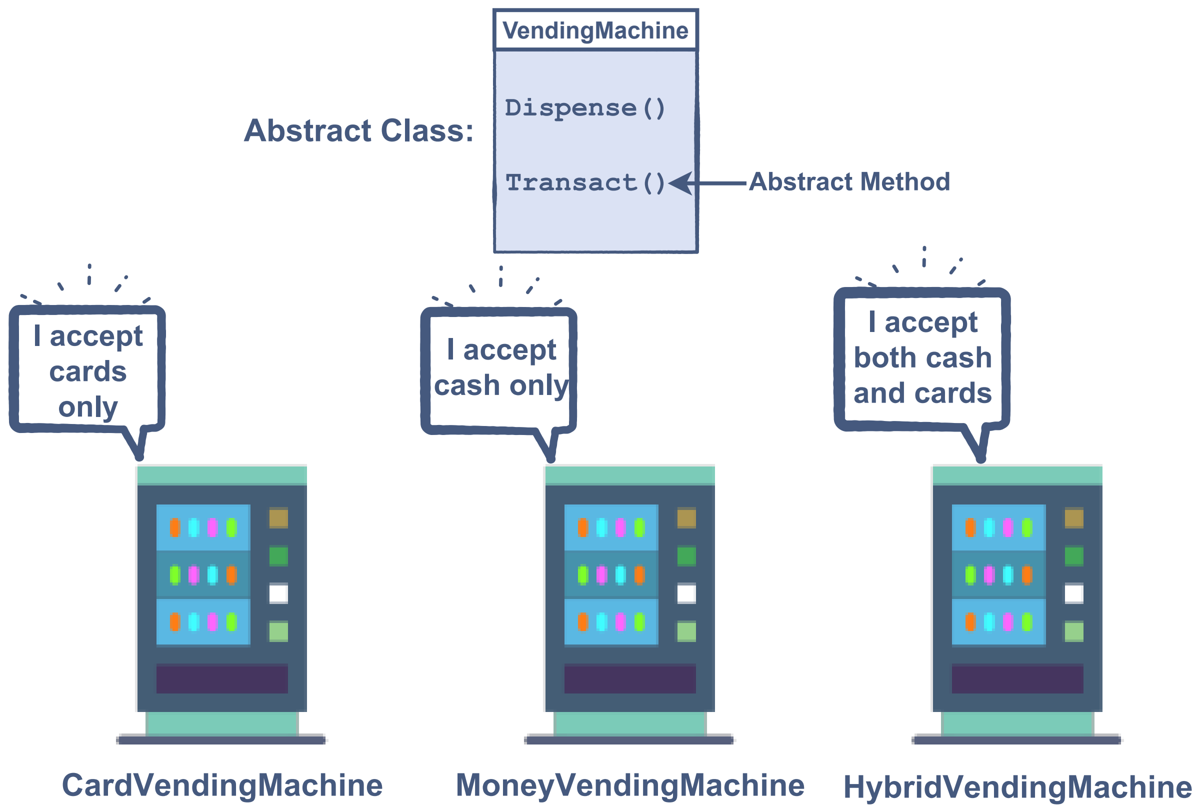


Inherited the implementation from the base abstract class

2 of 4



3 of 4



Overridden the implementation in the derived classes

4 of 4

Let’s look at the implementation of this example below:

```
abstract class EDM {

  public EDM() {
    //Parameter-less constructor
  }
  public abstract void Transact();

  public void Dispense() {
    Console.WriteLine("{0} is dispensing the product!",this.GetType().Name);
    //this.GetType().Name is an inbuilt functionality of C#
    //to get the class name from which the method is being called
  }

}

class CardVendingMachine : EDM {

  public override void Transact() {
    Console.WriteLine("I accept cards only!");
  }

}

class CashVendingMachine : EDM {

  public override void Transact() {
    Console.WriteLine("I accept cash only!");
  }

}

class HybridVendingMachine : EDM {

  public override void Transact() {
    Console.WriteLine("I accept both cards and cash!");
  }

}

class Demo {

  public static void Main(string[] args) {
    // Creating the objects
    EDM cardVendy = new CardVendingMachine();
    EDM cashVendy = new CashVendingMachine();
    EDM hybridVendy = new HybridVendingMachine();

    cardVendy.Dispense();    // Calling methods from CardVendingMachine
    cardVendy.Transact();
    Console.WriteLine();
    cashVendy.Dispense();   // Calling methods from CashVendingMachine
    cashVendy.Transact();
    Console.WriteLine();
    hybridVendy.Dispense(); // Calling methods from HybridVendingMachine
    hybridVendy.Transact();
  }


}
```

From the example above, we can observe just how beneficial an abstract class can be. An abstract class can be implemented fully *(all members are non-abstract)*, partially *(a mixture of both abstract and non-abstract members)*, not implemented at all *(purely abstract if all the members are abstract)* according to the requirements of the subclasses.

This covers the abstract classes and abstract methods. In the next lesson, you'll get to know about the interfaces.