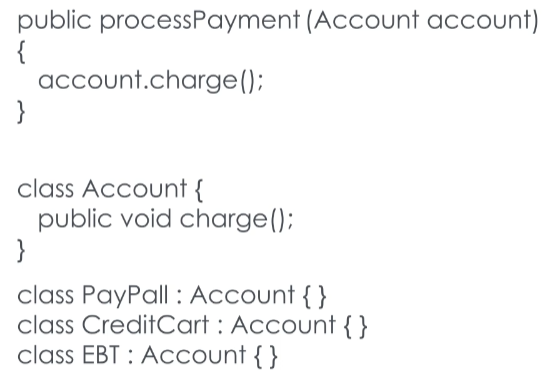
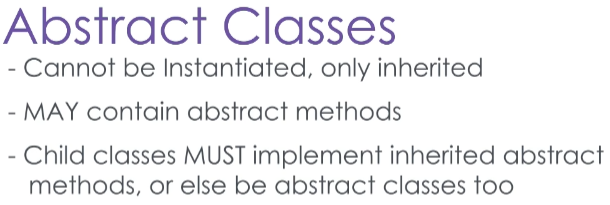
The benefits of inheritance are plain to see – you can write generic and adaptable code, and you can write less of it. But sometimes, inheritance can be tricky.

Suppose you’re writing an application that tracks animals at a zoo. You have bears and orangutans and fruit bats, and they all inherit from a base class called Animal. Obviously you’re going to be creating instances of bears and fruit bats, but it doesn’t make sense to actually instantiate a generic Animal, does it? How do you protect your code from unruly interns who might make that mistake?

Consider another scenario – you’re writing a method for a payment processing system. The method accepts a payment account as a parameter, and charges it. You would like this method to be able to accept a PayPal account, a credit card, a bank transfer…

wouldn’t it be nice if you could guarantee that each of those payment methods had a charge() method of their own?

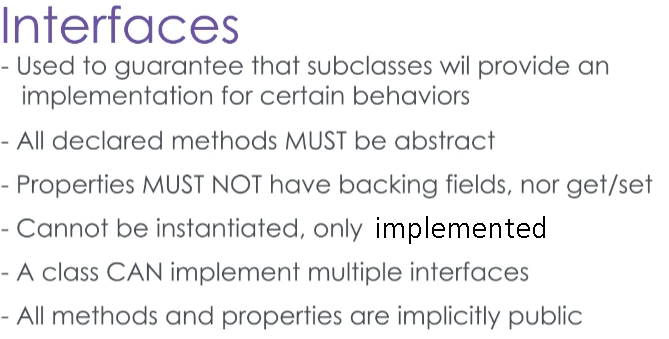
These are the problems that interfaces and abstract classes were made to solve.

Let’s start with Abstract classes;

these are classes that cannot be instantiated, only inherited. They may contain methods that are abstract, which means they have no implementation.

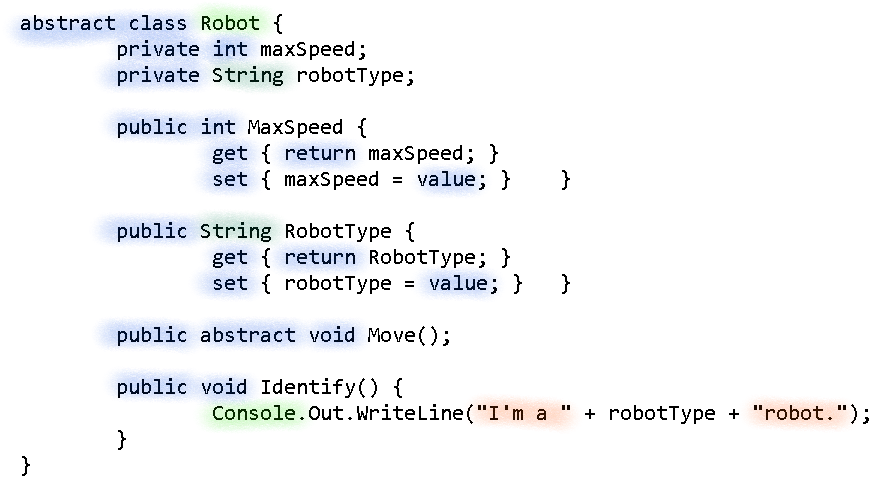
A child class that inherits an abstract class must implement any abstract methods by overriding them.

Remember though an abstract class doesn’t HAVE to contain an abstract method but it may. You could declare a class with only concrete methods as abstract, to prevent it from being instantiated.

Interfaces on the other hand…

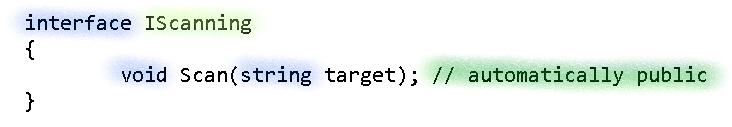
serve as a contract that sort of guarantees implementation of certain behaviors. This is because an interface may contain only abstract methods, and properties with no backing field and no implementation of get or set.

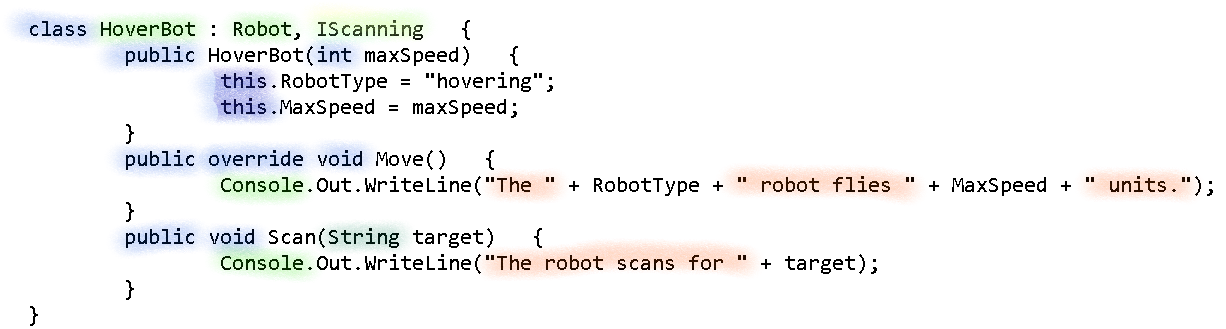
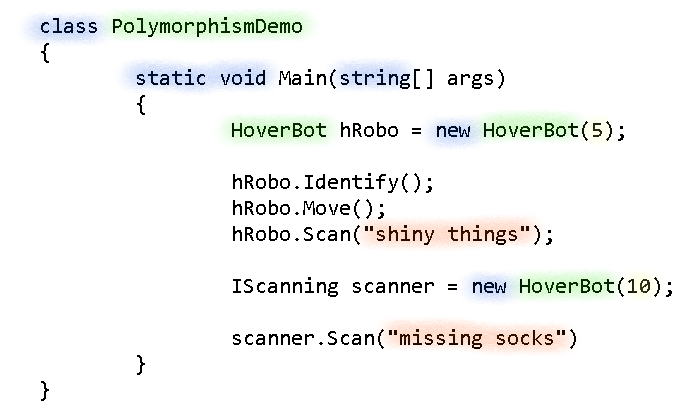
Like abstract classes, interfaces cannot be instantiated, only implemented. But unlike abstract classes, a concrete class can implement several interfaces at once. This is because they provide no implementation of their methods, and so this is not considered a form of inheritance. Also, every method or property declared in an interface is automatically public.

Here I have the usual Robot class, but you’ll note that it’s now an abstract class. It has the usual maxSpeed and type properties, but it now has an abstract move() method and a default implementation for an “Identify” method.

Extending the Robot class is the HoverBot class, which is empty save for an overridden move() method.

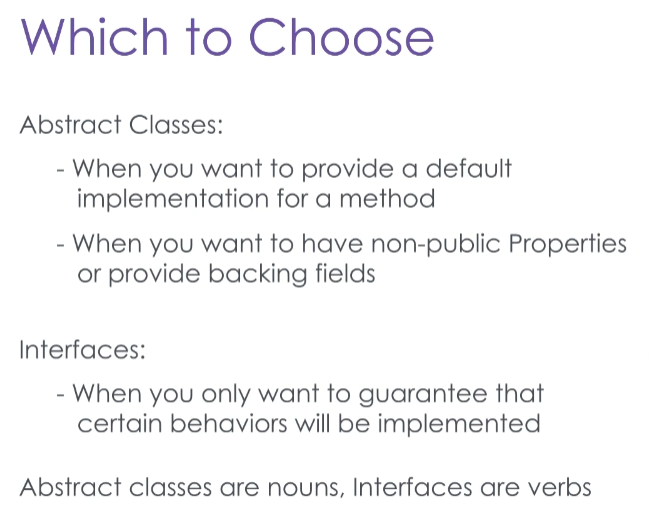
This hoverbot is also going to implement the IScanning interface, which declares a method scan(). Because of this, the HoverBot must implement the scan() method as well or be declared an abstract class.



Recall though, that through inheritance every child class shares an IS-A relationship with a parent class. A HoverBot IS-A Robot, and IS-A Scanning thing. We also know that because an Interface guarantees that an implementing class will have an implementation of all of its methods, we can use a reference variable of the Interface’s type to invoke those methods.

Here in my PolymorphismDemo class, I have created an instance of a HoverBot class, but I have stored it in an IScanning type reference.

While I cannot invoke any Robot or HoverBot methods, nor access their variables, I can invoke the methods declared in the IScanning interface.

When do you use an abstract class, and when do you use an interface?

**Use an abstract class when** you want to represent a concept with some default behaviors, or some preset properties.

For example, it’s perfectly acceptable to have a Mammal abstract class that has a variable called “hair thickness” because that is a property of all mammals. You can also have a default method called “giveBirth,” though obviously you’ll want to override that one for Platypi, but in most cases the default implementation will do.

**Use an interface only when** you want to guarantee that an object will be able to do certain things.

As a general rule with lots of exceptions, you can **think of an abstract class as a noun, and an interface as a verb**.

You might have an interface called “Movable” which declares a method “move.” This could be applied to the abstract class “cart,” which is inherited by both the “wagon” and “carriage” classes.

Knowing how to use abstract classes and interfaces is key to writing maintainable, versatile code.