

Interface vs. Abstract Class

We saw that an abstract class requires its subclasses, to implement its abstract methods.

An **interface** is similar to an abstract class, although it **isn't a class** at all.

It's a **special** type, that's more like a **contract** between the class and client code, that the compiler enforces.

By declaring it's using an interface, your class must implement all the abstract methods, on the interface.

A class agrees to this, because it wants to be **known by that type**, by the outside world, or the client code.

An **interface** lets **classes** that might have little else in common, be recognized as a special reference type.

Declaring an interface

Declaring an interface is similar to declaring a class, using the keyword `interface`, where you would use `class`.

On this slide, I'm declaring a public interface named `FlightEnabled`.

```
public interface FlightEnabled {}
```

An interface is usually named, according to the set of behaviors it describes.

Many interfaces will end in 'able', like `Comparable`, and `Iterable`, again meaning something is capable, or can do, a given set of behaviors.

Using an interface

A class is associated to an interface, by using the **implements** clause in the class declaration.

In this example, the class Bird implements the FlightEnabled interface.

```
public class Bird implements FlightEnabled {  
  
}
```

Because of this declaration, we can use FlightEnabled as the reference type, and assign it an instance of bird.

In this code sample, we create a new Bird object, but we assign it to the FlightEnabled variable named flier.

```
FlightEnabled flier = new Bird();
```

A class can use extends and implements in same declaration

A class can only **extend** a **single class**, which is why Java is called single inheritance.

But a class can **implement many interfaces**. This gives us **plug and play functionality**, which is what makes them so powerful.

A class can **both extend** another class, and **implement** one or more interfaces.

```
package dev.lpa;
```

```
public class Bird extends Animal implements FlightEnabled, Trackable {  
}
```

In this example, the Bird class extends, or inherits from Animal, but it's implementing both a FlightEnabled, and Trackable interface.

We can describe Bird by what it is, and what it does.

The abstract modifier is implied on an interface

We don't have to declare the interface type abstract, because this modifier is implicitly declared, for all interfaces.

```
abstract interface FlightEnabled {    // abstract modifier here is unnecessary  
                                     // and redundant
```

Likewise, we don't have to declare any method abstract.

In fact, any method declared without a body, is really **implicitly declared both public and abstract**.

The three declarations shown on this slide, result in the same thing, under the covers:

```
public abstract void fly();    // public and abstract modifiers are redundant,  
                               // meaning unnecessary to declare  
abstract void fly();          // abstract modifier is redundant, meaning  
                               // unnecessary to declare  
void fly();                   // This is PREFERRED declaration, public and  
                               // abstract are implied.
```

All members on an interface are implicitly public

If we omit an access modifier on a **class member**, it's implicitly package private.

If we omit an access modifier on an **interface member**, it's implicitly public.

This is an important difference, and one you need to remember.

Changing the access modifier of a method to **protected**, on an interface, is a **compiler error**, whether the method is concrete or abstract.

Only a concrete method can have private access.