Interface

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface.

Objectives

Upon completion of this topic, we will be able to:

- Define and give the uses of interfaces

Interface

One class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so that the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non-extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to use more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritances in a language such as C++.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {

return-type method-name1 (parameter-list);

return-type method-name2(parameter-list);
```

```
type final-varname1 = value;

type final-varname2 = value;

// ...

return-type method-nameN (parameter-list);

type final-varnameN = value;

}
```

Here, access is either public or not used. When no access specifier is included, it results in default access, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. 'name' is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

Here is an example of an interface definition. It declares a simple interface which contains a method called callback ( ) that takes a single integer parameter.

```
interface Callback {

void callback (int param);

}
```

Some Uses of Interfaces

- Interfaces are a way of saying, "You need to plug some code in here for this thing to fully work ". The interfaces specify the exact signatures of the methods that must be provided.
- Use the interface type as a parameter for a method. Inside the method you can invoke any of the methods promised by the interface parameter. When you actually call the method, you will have to provide an object that has all the methods promised by the interface. It might help to think of this as a bit like passing in a subclass as an argument and having the correct overriding methods called.
- Interfaces can be used to mix in generally useful constants. So, for example, if an interface defines a set of constants, and then multiple classes use those constants, the values of those constants could be globally changed without having to modify multiple classes. This basically means that interfaces separate design from implementation.

Interfaces versus Abstract Classes

While an interface is used to specify the form that something must have, it does not actually provide the implementation for it. In this sense, an interface is a little like an abstract class that must be extended in exactly the manner that its abstract methods specify.

- An abstract class is an incomplete class that requires further specialization. An interface is just a specification or a prescription for behaviour.
- An interface does not have any overtones of specialization that are present with inheritance.
- A class can implement several interfaces at once, whereas a class can extend only one parent class.
- Interfaces can be used to support callbacks (inheritance does not help with this). This is a significant coding idiom. It essentially provides a pointer to a function, but in a type-safe way.

Summary

Here are the key takeaways:

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritances in a language such as C++.
- Variables can be declared inside interface declarations.