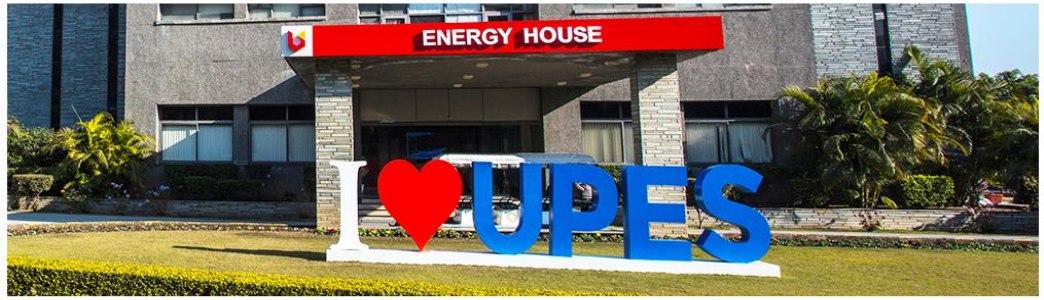

UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

UPES
UNIVERSITY WITH A PURPOSE

Default Interface Methods

- For all previous versions of Java, the methods specified by an interface were **abstract**, containing no body. This is the traditional form of an interface and is the type of interface.
- The release of JDK 8 changed this by adding a new capability to **interface** called the ***default method***.
- A default method lets us define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract.
- During its development, the default method was also referred to as an *extension method.*
- In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method.
- The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided.

Default Interface Methods

- Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.
- For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called remove( ), and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then remove() is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of remove( ), even though it was not needed.
- Addition of default methods does not change a key aspect of interface: its inability to maintain state information. An interface still cannot have instance variables, for example.
- It is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

Default Method Fundamentals

- An interface default method is defined similar to the way a method is defined by a class. The primary difference is that the declaration is preceded by the keyword **default**.

```
public interface MyIF {
// This is a "normal" interface method declaration.
int getNumber();
// This is a default method. Notice that it provides a default implementation.
default String getString() {
return "Default String"; }  }
```

- Because **getString( )** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
// Only getNumber() defined by MyIF needs to be implemented.
// getString() can be allowed to default.
public int getNumber() {
return 100; }  }
```

UPES
UNIVERSITY WITH A PURPOSE

Default Method Fundamentals (cont..)

- The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {
public static void main(String args[]) {
MyIFImp obj = new MyIFImp();
// Can call getNumber(), because it is explicitly implemented by MyIFImp:
System.out.println(obj.getNumber());
// Can also call getString(), because of default implementation:
System.out.println(obj.getString());
}
}
```

The output is shown here:

```
100
Default String
```

Default Method Fundamentals (cont..)

- As you can see, the default implementation of **getString( )** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString( )**, implementation by a class is optional.
- It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**:

```
class MyIFImp2 implements MyIF {
// Here, implementations for both getNumber( ) and getString( ) are provided.
public int getNumber() {
return 100;
}
public String getString() {
return "This is a different string.";
}
}
```

Now, when **getString( )** is called, a different string is returned.

UPES
UNIVERSITY WITH A PURPOSE

static Methods in an Interface

- JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods.
- Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object.
- Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

```
InterfaceName.staticMethodName
```

- Notice that this is similar to the way that a **static** method in a class is called.

static Methods in an Interface (cont..)

- The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {
// This is a "normal" interface method declaration.
// It does NOT define a default implementation.
int getNumber();
// This is a default method. Notice that it provides a default implementation.
default String getString() {
return "Default String"; }
// This is a static interface method.
static int getDefaultNumber() {
return 0; }  }
```

- The **getDefaultNumber( )** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber()** because it is **static**.

UPES
UNIVERSITY WITH A PURPOSE

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.