Rex Jones II

(Transcript) Interface How To Implement

How To Implement An Interface

Let's look at 2 email service providers: Outlook and GMail. With Outlook, we see a lot of functions such as File, Home, Send/Receive, Folder, View, and Help. If we wanted to create a new email, we can click New Email. If we wanted to close, we can click X in the top right corner or go to File then click Exit.

Those same functions are available with GMail and other email service providers. In GMail, to create a new email, we select Compose. It's the same function in Outlook but just a different name. The Spam folder is used to store email that we don't want in our inbox. Outlook has this same function but it's called Junk. Although both providers have functions that operate the same, the name is different and implementation details are different.

For consistency, we can create an interface so Outlook and GMail have the same name with their own unique implementation. Outlook has 3 methods: newEmailMessage, openJunkFolder, and closeOutlook.

```
public class Outlook {

   public void newEmailMessage () {
      System.out.println("Outlook - New Email");
   }

   public void openJunkFolder () {
      System.out.println("Outlook - Open Junk Folder");
   }

   public void closeOutlook () {
      System.out.println("Outlook - Close Email Provider");
   }
```

GMail also has 3 methods but different names: composeEmailMessage, openSpamFolder, and closeGmail.

```
public class GMail {

    public void composeEmailMessage () {
        System.out.println("GMail - Compose Email");
    }

    public void openSpamFolder () {
        System.out.println("GMail - Open Spam Folder");
    }

    public void closeGmail () {
        System.out.println("GMail - Close Email Provider");
    }
```

Recall from the introduction, an interface is a contract with a class. It's an agreement that determines what a class must do but not how the class must do it. Therefore, both classes will have the same method name but different details.

To create an interface, we go to New > Interface. The name will be EmailServiceProvider then click Finish. It looks like a class but we see public interface and not public class. We create a method by writing public void then the method name createEmailMessage () add a ; semicolon.

Writing keyword abstract after public is optional. As a convention, we do not write abstract. It's an assumption, that all methods in an interface are abstract although we can write default methods and static methods. For example, if I remove the semicolon then add a body which start and stop with the curly braces. An error states "Abstract methods do not specify a body" but show 3 quick fixes which include changing the method to default or static. Remove the braces then add back the semicolon. Abstract methods define the signature of a method but no implementation for the method. Some more methods that we can add to this interface are public void openJunkSpamFolder (); and public void closeEmailProvider ();.

```
public interface EmailServiceProvider {

    public void createEmailMessage ();
    public void openJunkSpamFolder ();
    public void closeEmailProvider ();
```

So, this interface says; all implementation classes should be able to create an email message, open a junk spam folder, and close the email provider.

Now, let's implement the interface. Go to Outlook and write implements EmailServiceProvider. Instantly, we see an error "The type Outlook must implement the inherited abstract method EmailServiceProvider.openJunkSpamFolder()". All 3 methods are recognized as an abstract method. The 2 quick fixes are Add unimplemented methods and Make type 'Outlook' abstract. Let's add the unimplemented methods. We see each method from the interface EmailServiceProvider with an Override annotation.

This is where the Outlook class can implement its unique details for each method. I'm going to copy and paste the print statements. Outlook – New Email, Outlook – Open Junk Folder, and Outlook – Close Email.

```
@Override
public void createEmailMessage() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - New Email");
}

@Override
public void openJunkSpamFolder() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - Open Junk Folder");
}

@Override
public void closeEmailProvider() {
    // TODO Auto-generated method stub
    System.out.println("Outlook - Close Email Provider");
}
```

The same for Gmail, it implements EmailServiceProvider and let's Make the type abstract. The keyword is added before class. We should not make the class abstract because it can lead to an error when creating an object. Objects cannot instantiate an abstract type. Remove abstract, save, then select Add unimplemented methods. Also copy and paste the print statements. GMail – Compose Email, GMail – Open Spam Folder, and GMail – Close Email Provider.

```
@Override
public void createEmailMessage() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Compose Email");
}

@Override
public void openJunkSpamFolder() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Open Spam Folder");
}

@Override
public void closeEmailProvider() {
    // TODO Auto-generated method stub
    System.out.println("GMail - Close Email Provider");
}
```

The contract agreement between the interface and each class has been fulfilled. Look what happens when the interface breaks the contract agreement by adding another abstract method.

public void saveDraftFolder(); then click Save. We see an error in the Outlook class and GMail class. The error exists because both classes was not part of the original agreement with saveDraftFolder.

To solve this error, we can do 1 of 2 things. First, we can go to each implementation class and implement the abstract method but what if our program had more than 2 classes? What if I wanted to add another method later on? I prefer option 2 and extend the interface. Java allows us to extend the interface so we don't have the fear of breaking an implementation class by creating a default method. The key is to add default and include a body. There will be an error if we only add default. The error states "This method requires a body instead of a semicolon". Add a print statement for the body by writing sysout("Save Draft \n") then Save. Both errors are gone but if we wanted to, we could override the default method.

However, when it comes to a static method, it provides a protection for an interface. It provides protection by not allowing an implementation class to override the static method. You can watch the Java Bindings video where I speak about how Java restricts static methods from being overridden. We

can write public static void sentEmailFolder () {
sysout(“Sent Email”) }.

```
public interface EmailServiceProvider {

   public void createEmailMessage ();
   public void openJunkSpamFolder ();
   public void closeEmailProvider ();

   public default void saveDraftFolder () {
      System.out.println("Save Draft \n");
   }

   public static void sentEmailFolder () {
      System.out.println("Sent Email");
   }
}
```

Let’s test the interface and both implementation classes. Go to New > Class > Main > TestEmailServiceProvider then Finish. EmailServiceProvider object outlook = new Outlook (); outlook. (dot) and we see 4 methods. All 3 abstract methods and the default method (closeEmailProvider, createEmailMessage, openJunkSpamFolder and saveDraftFolder). Do you see how the static method is not here?. Select openJunkSpamFolder / outlook.createEmailMessage / outlook.closeEmailProvider / outlook.saveDraftFolder.

We can write the same for GMail. EmailServiceProvider gmail = new GMail (); / gmail. openJunkSpamFolder / gmail.createEmailMessage / gmail.closeEmailProvider / gmail.saveDraftFolder.

```
public class TestEmailServiceProvider {

   public static void main(String[] args) {
      EmailServiceProvider outlook = new Outlook ();
      outlook.openJunkSpamFolder();
      outlook.createEmailMessage();
      outlook.closeEmailProvider();
      outlook.saveDraftFolder();

      EmailServiceProvider gmail = new GMail ();
      gmail.openJunkSpamFolder();
      gmail.createEmailMessage();
      gmail.closeEmailProvider();
      gmail.saveDraftFolder();
   }
}
```

Let's run. We see 8 print statements. 4 for Outlook and 4 for GMail.

```
Outlook - Open Junk Folder
Outlook - New Email
Outlook - Close Email Provider
Save Draft

GMail - Open Spam Folder
GMail - Compose Email
GMail - Close Email Provider
Save Draft
```

Next, we are going to look at how Interfaces support multiple implementation.