WA3524 Lab Guide

Java Refresher for ADP

ascendient learning

Part of **Accenture**

Revision 2.0.1 published on 2025-06-18.

Ascendient, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.ascendientlearning.com/

USA: 1-877-517-6540, email: getinfousa@ascendientlearning.com
Canada: 1-877-812-8887 toll free, email: getinfo@ascendientlearning.com

Table of Contents

Creating A Simple Object

MODULE 1

Time for Lab: 45 minutes

In this lab exercise, you will create a Java object.

Thus far, we have created several Java classes - but we have been using them in a strictly procedural manner. The classes we have created have only served to run via a single main method; in this context we have not really built a real object per se.

Recall that an object is meant to model a real life entity, such as a Person or a BankAccount. Objects have a state (which is maintained by *fields*) and operations (which are maintained by *methods*).

For example, let us say we are modeling a Person object. A Person object may have a field called **name** (of type String) and a field called **age** (of type int). Additionally, this Person object may have methods like **purchaseItem()** and **getHairCut()**.

Obviously, the fields and methods for this model would be pertinent to the overall object model we are building. If our model revolves around a Person operating a car, the **Person** would probably have fields like **driversLicenseNumber** and methods like **driveCar()**.

We will build a simple BankAccount class in this lab exercise.

1.1. Create The SimpleObject Project

To start this lab, you will be creating a new project to contain your code.

1. Open `your selected IDE`
2. Follow instructions from `your IDE intro lab` to open the IDE and create a new project.
3. Start by setting up the `New Project` dialog as described in `your IDE intro lab`.

4. Next you should override:
    - The project name to be: "simple-object"
5. Review your `New Project` dialog settings, and when you are ready, click the `Create` button.
6. Wait for project creation to complete.

    The message "JDK not defined" may appear. It should go away on its own once the project is fully created.

    When the status bar is free of messages you can continue.

1.2. Create the BankAccount Class

Now that we have a project, we can start creating our Java class. We will be creating a class called **BankAccount**. This class will model a simple bank account. It will have fields to represent the state of the bank account and methods to represent operations on the bank account.

You will add the following items manually.

- A package called **com.simple.account**
- A class called **BankAccount**

1. Follow the instructions found in `your IDE intro lab` to **create a Java Package**. Enter the package name shown above as the name of the package.
2. Follow the instructions found in `your IDE intro lab` to **create a Java Class**. Create the class in the package you just created. Enter the class name shown above as the name of the class.
3. Save the class file

    You should see the following code in the editor:

```
package com.simple.account;

public class BankAccount {
}
```

The first thing we should do is add some fields to the class. This can be done by adding field declarations to the class. Remember that an object should model some real world entity, and fields are used to model an entity's state.

Fields declarations are at the *class scope*, which means they are within the curly braces that form a class, but not within any method.

4. Add the following field declarations, after the opening '**{**', following the class declaration.

```
public class BankAccount {
    public int accountID;
    public String ownerName;
    public float balance;
}
```

What has happened here? We have specified that this class, when instantiated, will have three fields; one called ***accountID*** (which is an *int*), another called ***ownerName*** (which is a *String*) and a *float* called ***balance***.

accountID will represent the bank account's ID number.

ownerName will represent the name of the owner of the bank account.

balance will represent how much money the bank account has left in it.

This is a fairly standard representation of a real world bank account. Note that the types (e.g. int, String, float) are sensibly chosen based on the type of data they are representing.

Note that field names should always be in lower case. (e.g. **accountID** not **AccountID**). Also notice that we have made all these fields public by using the keyword **public.** This is also known as the *access modifier.* We will discuss access modifiers in more detail later.

Note: Strictly speaking, float is not the correct type to represent a currency value, because its internal representation is binary, which leads to round-off errors when representing decimal numbers. The 'java.lang.BigDecimal' class would be correct, but its usage is more complicated than the primitive 'float'. We are choosing to keep the code simple at this stage.

5. Save your code. There should be no errors. It should look like this:

```
package com.simple.account;

public class BankAccount {
    public int accountID;
    public String ownerName;
    public float balance;
}
```

Our next step is to add an operation to the class. We will do this by adding a method. The method we will create will be a simple one; we will write a **deposit** method. This method takes a float as an argument; it should add the float to the existing balance.

6. Enter the following code after the field declarations, but before the closing curly bracket for the class.

```
public void deposit(float amount) {
    balance = balance + amount;
}
```

The method is called **deposit**. It takes the **amount** passed in as a parameter, and then the body of the method adds it to the **balance** field. Note that we refer to the **balance** field as if it were a local variable.

7. Save the code. There should be no errors. Your code should look like the following:

```
package com.simple.account;

public class BankAccount {
    public int accountID;
    public String ownerName;
    public float balance;

    public void deposit(float amount) {
```

```
        balance = balance + amount;
    }
}
```

1.3. Create the Tester Class

Since our **BankAccount** class is complete, we now want to test it. Specifically, we want to write some Java code to create an instance of the BankAccount and test its fields and method.

But where do we do this? Recall that all Java execution starts from a **main** method somewhere. Our **BankAccount** class, however, does not have a **main** method. We could add one - but doing so would sort of break the idea of a **BankAccount** modeling a real life entity. After all, when was the last time you had a bank account with a **main** operation? A part of object oriented programming is devising good models.

A better approach would be to build a separate class that is designed specifically to test our **BankAccount** object. We could put a **main** method in this tester class and in that main method we could write code to test the **BankAccount** object. This is exactly the approach we will take.

1. When creating the **BankAccountTester** class, use the following settings:
    - Package name: **com.simple.test**
    - Class name: **BankAccountTester**
    - Main method: `public static void main(String [] args){}`
2. Follow the instructions found in `your IDE intro lab` to **create a Java Package**. Enter the package name shown above as the name of the package.
3. Follow the instructions found in `your IDE intro lab` to **create a Java Class**. Create the class in the package you just created. Enter the class name shown above as the name of the class.
4. Add the `Main method` listed above to the class.

5. Save the class file

   When you are done you should see the following in the editor window:

```
package com.simple.test;

public class BankAccountTester {
    public static void main(String[] args) {

    }
}
```

6. Now, let us begin coding. In the **main** method, add the following code *inside* the **main** method, but before the closing curly brace.:

```
BankAccount account = new BankAccount();
```

   We declare a new local variable called **account** and call the constructor to initialize it, much as we have done so before.

   You have probably noticed that Eclipse is saying there is an error on the line of code you just entered. Assuming you have not made any typos, this is because of the packaging issue. The BankAccount class is in the **com.simple.account** package, so it must either be referenced by fully qualified name, or imported. We'll take the easy route and import the package.

7. An easy way to do this is to click anywhere on the code editor and type

8. If needed add an import for BankAccount. This can be done manually or by executing your IDE's organize imports command.

```
import com.simple.account.BankAccount;
```

9. Save the file. There should be no errors.

   We have now created code to create an instance of the BankAccount class, giving us a BankAccount instance. Let us now perform some interesting code on it.

10. Add the following statement in the **main** method immediately after the line that creates the account instance.:

```
account.accountID = 1;
```

This line takes that instance of the BankAccount and sets its field accountID to the value 1. This means we are explicitly making the value of this data field to be 1. This instance will now remember that state.

Note that we use the dot (.) operator to access a field on the class. We use that access and the = sign to set the value of the field. This is known as *direct method access*.

11. Set the state of the other two fields by adding the following two lines.

```
account.ownerName = "Jeff Lebowski";
account.balance = 100f;
```

This is fairly straightforward, except for the 'f' at the end of the 100. This is done because the **balance** field is a **float** and must be assigned to a **float**. Adding the 'f' at the end of 100 tells Java to use the **float** equivalent of 100.

We have now created an instance of a BankAccount and set its fields to some sample data. Let us now try an operation on it. We should now test to see if it worked. This can be done by using the standard println methods. Specifically, we will **println** out the values of the fields to see they were set properly.

12. Add the following code immediately after the setting of the balance.

```
System.out.println("A Bank Account");
System.out.println("ID: " + account.accountID);
System.out.println("Balance: " + account.balance);
System.out.println("Owner: " + account.ownerName);
```

This code is quite simple. It simply uses the dot (.) operator to access the fields of the **account** instance so they can be sent to **println**.

13. Save the code.

14. Run the `BankAccountTester` class. You should see the following output:

What do you see?

```
A Bank Account
ID: 1
```

```
Balance: 100.0
Owner: Jeff Lebowski
```

As expected, the BankAccount's fields contain the correct data and were printed out successfully. It looks like our object works!

15. Now, let us try invoking an operation on the class. Recall that we have written one method: the **deposit** method. We will use it now. Immediately before the first println statements, insert the following:

```
account.balance = 100f;
account.deposit(50f);
```

16. Save the file.

    We invoke the method on our account by using the dot (.) operator, much like we accessed the fields. However, when invoking a method, the brackets (and) are present and they often contain parameters. In this case, we invoked the method with the parameter 50f. *By invoking the method, we tell Java to go to the class that we invoking the method on, and execute the method with the same name*. In our case, Java looks up the BankAccount class, locates the deposit method and executes the code there.

    What should this code do? Well, if you look back at the source code for the BankAccount class, you will see that the code takes the passed in parameter and adds it to the current balance.

    So, following this logic, if we run the code again, we should now see an updated balance of 150.0.

17. Test this by running the class.

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
```

    As expected, the balance has been correctly updated.

    Congratulations! You have successfully written and tested your first Java object!

1.4. Review

In this exercise, you create a Java object and gave it a state (via fields) and operations (via methods). You saw how to define a field and a method. You then wrote a tester class with a main method that created an instance of and then operated on the Java object.

Although this all seems quite straightforward, we actually made quite a few _bad _ coding decisions. We will address this in the next exercise.

Getters and Setters

MODULE 2

Time for Lab: 30 minutes

In the `simple-object` exercise, you created the `BankAccount` class which consisted of 3 fields, and a method. You then tested the class by creating an instance of it and settings its three fields to some values by *direct method access*.

In this exercise, we will remove direct method access, and instead use proper *getter and setter* methods.

2.1. Problems of Direct Method Access

Direct method access is the use of an object instance's fields by using the dot (.) operator. While there is technically nothing wrong with this (Java fully allows this), it is not good objected oriented design.

Remember that an object is supposed to hide its implementation details from any other class that uses it. A public face (or *application programming interface - API*) to the object should be made available for other classes to use, and it is that public API that should operate on the implementation. By allowing another piece of code to access the fields directly, we are allowing access to the internal representation of the class. This is not good encapsulation.

Additionally, think of this potential error situation. Currently, what mechanism is in place to prevent the balance from being set to a negative number? By using direct access, the code:

```
account.balance = -140f;
```

Could be executed. This is a violation of business rules. Ideally, when attempting to set this value, the BankAccount class should raise some sort of error and prevent the actual value from changing. This sort of error handling cannot be done with direct field access.

So, the idea here is that we want to prevent the direct access of fields. Instead, we would like to have some sort of way of accessing the fields that could be made public, and those public ways could have error checking code inside them.

This can be done with the use of *accessors*, also known as *get/set* methods, or simply *getters and setters*.

A *get* method is a public method that retrieves a field value and a *set* method is a public method that sets it. These methods are typically public, while the fields they touch are made private.

In order to change the value of a field on a class, the public *accessors* should be used. The fields themselves would be made **private**, thus preventing direct method access.

Let us make these changes now.

1. Open `your selected IDE`
2. Load the `simple-object` project.

   > ⓘ **Note**
   >
   > If you have not done the `simple-object` lab, you will need to do so before proceeding, or you can choose to run the lab solution for the `simple-object` lab and work from there. Information on running lab solutions can be found in the `your IDE intro lab`.

3. Open the **BankAccount.java** class.

   The first thing we need to do is disable direct method access for the fields. We can do this by changing the *access modifier* for the fields. The access modifier specifies what other classes can touch these fields. Right now, the modifier for the fields is **public** which means any other class can – including our tester class. We should change this.

4. Change the modifiers to **private** as shown in bold below.

```
private int accountID;
private String ownerName;
private float balance;
```

5. Save the file. There should be no errors in the BankAccount class itself but there will be some in `BankAccountTester`

The BankAccountTester class is still trying to use direct method access - but is not able to do so, since we made the fields **private**. So, the errors are expected - and even wanted. Direct method access has now been disallowed!

6. Our next step is to begin coding the *accessor* methods. Add the following method, after the closing curly brace } of the **deposit** method but before the last closing bracket for the class itself.

```
public void setAccountID(int newID) {
    this.accountID = newID;
}
```

7. Save the file.

   Examine this method. It is called **setAccountID** and it takes a parameter of type **int** called **newID**. What does it do? Simple. It assigns the field called **accountID** (of **this** object instance) to the passed in parameter. It has a **void** return type – because it does not return anything.

   This more or less has the same effect of the direct method access that we were using before, except now it is being done from within the class, and not an external class. It is now an implementation detail.

   Since the method is **public**, however, other classes can now use this method. Let's try this.

8. Switch back to **BankAccountTester.java.** There should be several errors on the class. Locate the line:

```
account.accountID = 1;
```

9. Then change it to:

```
account.setAccountID(1);
```

   Instead of using direct method access, we use the newly created **set** method. The error should go away in this line!

10. Now, let us write the corresponding *get* method. Switch back to **BankAccount.java** and add the following method, after the previous **set** method:

```
public int getAccountID() {
    return this.accountID;
}
```

This method is quite simple. It is public, but it now has a *return type*. This means the method will return an **int** when it is executed. What is it that gets returned? Simple. The accountID field. This is done in the method body. (Notice, again, the use of the **this** keyword).

11. Save the file.

12. We have now written a *get* method. We should change our BankAccountTester class to use it. Switch back to the BankAccountTester class and locate the line:

```
System.out.println("ID: " + account.accountID);
```

13. Change it to the following:

```
System.out.println("ID: " + account.getAccountID());
```

14. Save the file.

    One error should go away. Previously, the line was using direct access method to *get* the **accountID**. Now, it is using the public **getAccountID()** method.

    We have now written the get and set methods for the **accountID** field. The next step is to write get and set methods for the other fields. We will do the **ownerName** next.

15. Switch back to BankAccount.java and add the following two methods to the class:

```
public void setOwnerName(String newName) {
    this.ownerName = newName;
}
public String getOwnerName() {
    return this.ownerName;
}
```

    These work almost identically to the get and set methods for the accountID, except they operate on the **String** field **ownerName**.

16. Save the file.

There should be no errors in this class.

17. Switch back to BankAccountTester. We will change it to use the new get and set methods.

18. Locate the line:

```
account.ownerName = "Jeff Lebowski";
```

19. Change it to:

```
account.setOwnerName("Jeff Lebowski");
```

20. Now, locate the line:

```
System.out.println("Owner: " + account.ownerName);
```

21. Change it to:

```
System.out.println("Owner: " + account.getOwnerName());
```

22. Save the file. Two more errors should go away.

    You have now written and used two get/set methods!

2.2. Generating Accessors

We have now seen that we create get/set methods for the fields on our class. In general, it is a good idea to create these accessors for every field that needs to be publicly accessed. However, creating them can be monotonous; we've created two, and have to create one more. Imagine a complex class with 10 fields, all of which need accessors!

Fortunately, since the code behind an accessor is quite straightforward, we can actually have the IDE generate the code for us, saving us the trouble of writing them. We will do this now.

1. Switch back to BankAccount.java

2. Follow instructions in `your IDE intro lab` to `Generate Getters and Setters` for the **balance** variable.

   When you are done the following code should have been added to the `BankAccount` class:

```
public void setBalance(float newBalance) {
    this.balance = newBalance;
}
public float getBalance() {
    return this.balance;
}
```

   Notice that it named the methods much in the way we named the other ones: a **get** or a **set** with followed by the field name, with the lettering in "camel casing" style (e.g. **getOwnerName()**, **setBalance()**, etc) This is a general naming convention that all Java programmers should follow.

3. Save the `BankAccount` file. There should be no errors in this class.

4. Switch back to the **BankAccountTester.java** and change the code to use the new getter and setter methods for the balance.

5. When you are finished, the code should look like this:

```
package com.simple.test;
import com.simple.account.BankAccount;
public class BankAccountTester {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.setAccountID(1);
        account.setOwnerName("Jeff Lebowski");
        account.setBalance(100f);
        account.deposit(50f);
        System.out.println("A Bank Account");
        System.out.println("ID: " + account.getAccountID());
        System.out.println("Balance: " + account.getBalance());
        System.out.println("Owner: " + account.getOwnerName());
    }
}
```

6. Save the file. All the errors should be gone.

7. Run BankAccountTester. It should look the same as before.

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
```

While we have not actually changed any of the logic of the code, we have refactored it to use better object oriented programming techniques.

2.3. Accessor Error Handling

Earlier, we discussed how a potential use for accessors was that of error handling. Specifically, we stated that by using direct method access, there was no way of ensuring that a negative value would not be passed into the balance field. Now that we are using accessors, however, such a mechanism is available.

1. Open **BankAccount.java**.
2. Change the code for the **setBalance(float balance)** method to the following:

```
public void setBalance(float balance) {
    if(balance > 0f) {
        this.balance = balance;
    } else {
        System.out.println("Error. Balance cannot be negative.");
    }
}
```

What is happening here? Simple; the method first checks to see if the balance is larger than 0; if it is, then it is valid and the field is set. Otherwise, an error message is printed out. (Note that using System.out.println is a bad way of error handling; a better way would be to use the **exception** framework, which we will discuss later in the course)

3. Save the file. There should be no error.
4. Switch back to the BankAccountTester.java.
5. Let's test this. Change the **setBalance()** amount to be **-100f** and save.

6. Now run the code. What do you see?

```
Error. Balance cannot be negative.
A Bank Account
ID: 1
Balance: 50.0
Owner: Jeff Lebowski
```

   We see an appropriate error message displayed. Our error handling code has worked!

   However, below in the console, we see the balance is still 50.0. Can you explain why that is? (Hint: floats default to a value of 0)

7. Change the value of the **setBalance** back to the original amount (100f) and save the file.

```
account.setBalance(100f);
```

8. Run the code again. It should show again 150 as Balance.

2.4. Review

In this lab exercise, you modified your BankAccount class to use better OO techniques. Specifically, you made the fields private, and then wrote public accessors for them. You then updated the BankAccountTester to use the new accessors - again, better OO programming style.

In writing the accessors, you saw they provide a public interface to what should otherwise be a private implementation, and you saw that one advantage over direct field access is the ability to put in error handling code on set time.

Finally, you saw that writing accessors can be very tedious; fortunately, Eclipse can generate them for you.

Using Constructors

MODULE 3

Time for Lab: 30 minutes

In the `getters-and-setters` lab, you took the `BankAccount` class and refined it to be more in line with good Object Oriented programming conventions by making the fields private and adding accessors.

In this exercise, we will continue with our `BankAccount` class, and examine other Java features such as constructors and class members.

3.1. Creating Another Instance Of The BankAccount

1. Open `your selected IDE`
2. Load the `simple-object` project.

> ⓘ Note
>
> If you have not done the `getters-and-setters` lab, you will need to do so before proceeding, or you can choose to run the lab solution for the `getters-and-setters` lab and work from there. Information on running lab solutions can be found in the `your IDE intro lab`.

3. Open the **BankAccountTester.java**.

   Recall that in this code, we are only using one BankAccount instance. We know this because we only declare a single instance (called **account**) and *initialize* it once.

   Let us add some code to create another instance of a BankAccount, this one called **account2**.

4. Add the following code inside the **main** method, after the **println** statements.

```
BankAccount account2 = new BankAccount();
```

5. Now, set some data on that instance.

```
account2.setAccountID(2);
account2.setOwnerName("Bunny Lebowski");
account2.setBalance(5000f);
```

6. Immediately following this, create yet another instance of the account, and set its data as follows:

```
BankAccount account3 = new BankAccount();
account3.setAccountID(3);
account3.setOwnerName("Walter Sobcheck");
account3.setBalance(1000000f);
```

7. Save the code. There should be no errors.

   The pattern here is all the same. We declare and initialize an new instance, and then set its fields using the set methods.

   The initialization step (via the **new**) keyword is implicitly calling a constructor on the class. The constructor is essentially a special method that is invoked when the **new** keyword is used.

   Without knowing it, we have been using the constructor on our BankAccount class. This may seem strange, because we never created one. This is because, for every Java class we create, Java itself *implicitly* gives us a constructor 'for free'. This 'free' constructor is the one that has been used whenever we were initializing our class.

   If it is not adequate for us, we can always create our own constructor or constructors. Why would we do this? Look at the previous code we just entered.

   Every time we create a new instance of a BankAccount, we have to explicitly set the **accountID**, the **ownerName** and the **balance**, using the set methods. Four lines of code; one for the initialization, and 3 for the *setters*. The *setters* are necessary because we do not want a BankAccount instance to exist without having the data in place; after all, how much sense would it make for an account to not have an **accountID**, or a **balance** or an **ownerName**?

   Keeping this in mind, we can create a new constructor. We will make this new constructor require that the **accountID**, the **balance** and the **ownerName** be

passed in as parameters. This means that a BankAccount will always have some appropriate data set on it after construction.

Let us do this now.

3.2. Create The New Constructor

1. Open the **BankAccount.java** class.
2. Add the following constructor code right after the field declarations.

```
public BankAccount(int accountID, String ownerName, float balance) {
    super();
    this.accountID = accountID;
    this.ownerName = ownerName;
    setBalance(balance);
}
```

Even though this looks like a regular Java method, this is actually a constructor. How do we know this? First of all, there is no return type for the method. Secondly, the name of the method is simply the name of the class.

How does this method work? Firstly, note that it takes 3 arguments; an int, a String and a float. Coincidentally, these are the same types as the three fields on the class.

Secondly, the code makes a call to the **superclass's** constructor. Superclasses will be discussed in a later chapter, so you can ignore this for now.

Finally, the method merely sets the three fields on the class to the three arguments passed in. The balance is set using the 'setBalance' method since there is validation logic in this method.

Note the use of the **this** keyword here. In this context, **this** is used to separate the field (**this.accountID**) from the passed-in parameter (**accountID**). Imagine how the code would look if **this** was not used.

3. Save the code. There should be no errors on the **BankAccount** class. But there will be problems in the BakAccountTester.java file. You may see one or more of the following errors with each of the lines that create a new instance of the BankAccount class:

```
'Expected Three arguments, found 0'
'The constructor BankAccount() is undefined'
```

`BankAccount()` with no parameters, was the "free" constructor that Java was supplying for us, and which we were using.

However, since we have written our own 3 argument constructor, *Java has removed the "free" constructor* so it can no longer be used. **If you create your own constructor for a class, the "free" constructor is no longer available**.

What now? Simple. We change our code to use our new constructor or create a new **no argument** constructor.

Lets try using the new constructor.

4. Replace the following code:

```
BankAccount account = new BankAccount();
account.setAccountID(1);
account.setOwnerName("Jeff Lebowski");
account.setBalance(100f);
```

5. With this, much simpler code:

```
BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
```

The new code uses our new 3 argument constructor. We pass in the 3 parameters that we would have otherwise set using the setter methods, and we know what the constructor code will do with them. Four lines of code have now become one line.

Let's Use this constructor elsewhere in the code as well.

6. Replace this code:

```
BankAccount account2 = new BankAccount();
account2.setAccountID(2);
```

```
account2.setOwnerName("Bunny Lebowski");
account2.setBalance(5000f);
```

7. With this, simpler code:

```
BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
```

8. Finally, replace this code:

```
BankAccount account3 = new BankAccount();
account3.setAccountID(3);
account3.setOwnerName("Walter Sobcheck");
account3.setBalance(1000000f);
```

9. With this, simpler code:

```
BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
```

10. Save the file. Your code has now been greatly streamlined. It should look like the following:

```
public class BankAccountTester {
        public static void main(String[] args) {
                BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
                account.deposit(50f);
                System.out.println("A Bank Account");
                System.out.println("ID: " + account.getAccountID());
                System.out.println("Balance: " + account.getBalance());
                System.out.println("Owner: " + account.getOwnerName());

                BankAccount account2 = new BankAccount(2, "Bunny Lebowski",
5000f);
                BankAccount account3 = new BankAccount(3, "Walter Sobcheck",
1000000f);
        }
}
```

11. Run the code. It shouldn't look any different, but it is definitely cleaner in terms of style.

12. Although the code runs correctly you may see some warnings like this.

```
The value of the local variable account2 is not used
```

These are normal. The IDE is giving us a hint that we've declared some variables but are not using them for anything, yet. This is a possible waste of memory. We will use them later and so can ignore the warnings for now.

3.3. Object Instances

Let us now play around a bit with the various instances.

1. Go back to the **BankAccountTester.java** class and add some println statements following the construction of **account2** and **account3**, like the following:

```
System.out.println("account2 is owned by " + account2.getOwnerName());
System.out.println("account3 is owned by " + account3.getOwnerName());
account2.deposit(100f);
account3.deposit(900f);
System.out.println("account2's balance is " + account2.getBalance());
System.out.println("account3's balance is " + account3.getBalance());
```

2. Save the file and run the code.

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
```

You should be seeing here that each instance is considered unique. Changing the fields (e.g. depositing money) into **account2** does not affect the fields of **account3**.

3.4. Static Fields

So far, we have seen fields on a class; the BankAccount class had fields **ownerName**, **balance** and **accountID**. Those were *member fields* (also known as

instance variables), meaning each instance can have a unique value. As you have seen in your BankAccountTester class, we created three instances of a BankAccount, each one having its own **accountID**, **ownerName** and **balance**.

What if, however, we wanted a variable that was the same for **all** instances of an object? For example, all BankAccounts may have different balances, ownerNames and accountIDs, but they all might have the same *interest rate*.

We could create a field called **interestRate** on the class as we have done so before, and treat it as a normal field. Let us do this now.

1. Open the **BankAccount.java** class.
2. Add a new field after the other field declarations as follows:

```
private float interestRate;
```

3. Follow instructions in `your IDE intro lab` to `Generate Getters and Setters` for the **interestRate** variable.

   When you are done the following code should have been added to the `BankAccount` class:

```
public float getInterestRate() {
        return interestRate;
}

public void setInterestRate(float interestRate) {
        this.interestRate = interestRate;
}
```

4. Save the file. There should be no errors.
5. Go back to the **BankAccountTester.java**.
6. Let's try out these fields. We will set the field on account2. Locate the following line:

```
BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
```

7. Immediately after, insert the following line:

```
account2.setInterestRate(5.5f);
```


8. We are simply setting the field value on the instance **account2**.

9. Now, at the end of the method, after the println of account3's balance, add the following 2 lines of code.

```
System.out.println("account2's interestRate is " + account2.getInterestRate());
System.out.println("account3's interestRate is " + account3.getInterestRate());
```

10. Save the file and run the class.

    What should happen here? The printlns should show the interest rate for both instances. What do you expect to see?

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
account2's interestRate is 5.5
account3's interestRate is 0.0
```

    We see two interest rates ... which is what we expected to see. The interest rate for **account2** is 5.5 (as we stipulated in our code), but the interest rate for **account3** is 0.0. The 0.0 is because that is the *default* value for a float (i.e. the value that is assigned if no other value is assigned.

    This is nothing new. We have already seen that two different instances will have two different values. If, however, we wanted the interest rate for **account2** and **account3** (and even **account1**) to be the same (as it should be), we would have to call the **setInterestRate()** method on all three instances. Having to do this would be highly impractical.

    So a better approach would be to use a *static field*. A static field (also known as a *class field*) is a field that is the same for *all* instances of the class. Think of it as a field on the actual *class* object as opposed to the *instance* of the class. A field can be made static by using the **static** keyword. We will do this now.

11. In **BankAccount.java**, locate the field definition for **interestRate** and change it to the following:

```
private static float interestRate;
```

We add the keyword **static** here.

12. Also change the **setInterestRate** method and change it as follows:

```
public static void setInterestRate(float interestRate) {
    BankAccount.interestRate = interestRate;
}
```

> ⓘ Note
>
> This sets the static field and also modifies the method so it is declared as static.

13. Modify the '**getInterestRate**' method so it is declared static also.

```
public static float getInterestRate() {
    return interestRate;
}
```

14. Save the class.

15. Go back to the **BankAccountTester.java** class and modify all of the places where the interest rate is used to use the '**BankAccount**' class instead of one of the instance objects. You should have code like:

```
BankAccount.setInterestRate(5.5f);
...
BankAccount.getInterestRate()
BankAccount.getInterestRate()
```

16. Now, let us test this. Save and Run the code again. What do you see?

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
account2's interestRate is 5.5
account3's interestRate is 5.5
```

Notice that the interestRates are now the same! By making the field static, we have made it the same for all instances of the class. You could now go and change the value (by calling **setInterestRate()**) on any of the instances (account1, account2 or account3) and all 3 instances would share the same value.

Congratulations! You have created your first static field.

3.5. Review

In this lab, you saw how Java provides a default zero-argument constructor for every class. You then created your own constructor which took parameters and used that in place of the default constructor. This new constructor made it easier to create and initialize new BankAccount objects.

Additionally, you saw how a **static** field could be used to maintain states across all instances of a class.

Looping

MODULE 4

Time for Lab: 30 minutes

In this lab, you will experiment with looping constructs in Java. Using a simple **for** loop, you will see how to repeat logic in a structured and controlled manner.

4.1. Create the Project

In this part you will created a Java project in `your selected IDE`.

1. Open `your selected IDE`
2. Follow instructions in `your IDE intro lab` to create a new Java project:
    - Set the "Project Name" to: **looping**
3. Wait for the project creation to complete.
4. Follow instructions in `your IDE intro lab` to create a new Java package:
    - Set the "Package Name" to: **com.simple**

4.2. Create the Class

We will create an entirely new Java class just to handle this logic. It will be similar to the **SimpleArithmetic** and **HelloWorld** classes we created earlier in that the code will reside in a single **main** method.

The class will use a **for** loop to repeatedly ask the user to enter a number (10 times). Each number entered will be added to a running sum; at the end of the loop, the sum total will be displayed.

1. Create a new Class with the following properties:
    - Create the Class in the **com.simple** package.
    - Name the Class: **SimpleAdder**.

2. Add the following code to the class:

```
public static void main(String args[]) {}
```

3. Enter the following code in the **main** method:

```
int sum = 0;
Scanner scan = new Scanner(System.in);
for(int x = 0; x < 10; x++) {
    //
}
System.out.println("The sum is " + sum);
scan.close();
```

4. Add the following import statement to the top of the class:

```
import java.util.Scanner;
```

5. Save the class.

   We first declare an **int** variable (called **sum**) that will represent our running total.

   We then declare an instance of a **Scanner** that we will use to obtain input from the user.

   We then begin a **for** loop and set up (but do not code) the loop body. We will add the body in a moment. First, notice the **for** loop syntax. The first part of the loop statement (*int x = 0*) is the *initial condition*. In it, we declare a loop variable which in our case is a simple **int** called **x**.

   The next part of the statement (*x < 10*) is the *continuation condition*; the loop will continue until this condition is reached. In this case, the code is saying that while the value of **x** is less than 10, the loop should continue.

   Finally, the last part of the statement (*x*++) is the *increment clause*. This reflects what should be done on every iteration through the loop. In this case, we merely increment the value of **x** by one.

   From this, you should see that our loop will iterate 10 times; once for **x** being 0, another for **x** being 1, another for **x** being 2, and so on.

(We place an open and close curly brace pair after the **for** statement. This is the loop body, and we will place the code we want repeated within these curly braces. We will do that later, however.)

Finally, we print out a simple statement showing the sum and close the input scanner.

6. Save the `SimpleAdder.java` file. There should be no errors. Your code should now look like this:

```
package com.simple;

import java.util.Scanner;

public class SimpleAdder
{
    public static void main(String[] args) {
        int sum = 0;
        Scanner scan = new Scanner(System.in);
        for(int x = 0; x < 10; x++) {
            //
        }
        System.out.println("The sum is " + sum);
        scan.close();
    }
}
```

We can now go ahead and add the loop body code. The loop body will be quite simple.

As stated earlier, we will just prompt the user for a number (using a **Scanner** for **System.in** as we did in a previous exercise) and then add that number to the sum.

7. Add the following code into the body of the **for** loop.

```
System.out.println("Please enter integer #" + (x+1));
int input = scan.nextInt();
sum += input;
```

Three lines of code. The first line merely displays a meaningful prompt for the user. It tells the user which "number" to enter (e.g. the first number, second number, third number all the way to the 10^{th} number).

The second line performs the actual input of data from the user, and the third line adds it to the current sum.

8. Save the code. There should be no errors regarding to the SimpleAdder class.

   The `for` statements should look like this:

```
for(int x = 0; x < 10; x++) {
    System.out.println("Please enter integer #" + (x+1));
    int input = scan.nextInt();
    sum += input;
}
```

9. Run the `SimpleAdder` class.

10. Enter numbers into the console as prompted.

11. Keep entering numbers until the program completes. Make sure to enter valid numbers.

```
...
Please enter integer #8
8
Please enter integer #9
9
Please enter integer #10
10
The sum is 55
```

    Success! The loop is working.

    Note that at the moment, there is no concept of error handling.

12. Try running the program again, but entering some non-numeric (e.g. some character data) into the program.

```
Please enter integer #1
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
        at com.simple.SimpleAdder.main(SimpleAdder.java:12)
```

Java throws a nasty message, raising an **exception**. The program then terminates ungracefully. Naturally, this is not a good user interface; instead of behaving this way, the program should handle itself more gracefully, and that can be done with the use of *exception handling*. We will learn about proper exception handling later on. For now, we will proceed.

4.3. Break and Continue

So far, we have seen a simple **for** loop which shows how we can iterate through code. A loop also has two other constructs that can be helpful: **break** and **continue**.

Within a loop, issuing a **continue** statement will tell the current iteration to cease executing, and go to the next iteration.

Issuing a **break** statement will cause the loop to terminate completely. We will use these in our code now.

Let us change the logic of our code a little. Right now, we are simply adding integers, with no regard for error handling. What happens right now if the user enters a *negative* number? The program will still execute, adding the negative number to our sum, thereby *decreasing* it. We will now change our code to disallow this.

Specifically, if a negative number is entered, we will want our program to merely skip over the addition of the entered number to the sum, and continue on with the next iteration. This can be achieved with a **continue** statement.

1. Right before the line:

   ```
   sum += input;
   ```

2. Insert the following code:

   ```
   if (input < 0) {
       continue;
   }
   ```

This simply checks to see if the number is less than zero. If it is, skip the remainder of the code in this iteration and go to the next one.

3. Save the code. There should be no errors.
4. Run the `SimpleAdder` Class.
5. Enter 10 for the first number, and -1 for all the remaining others.

```
...
Please enter integer #9
-1
Please enter integer #10
-1
The sum is 10
```

You should see that the negative numbers have been ignored, as expected.

Note: You could also have achieved this behavior just by using an **if** statement and proper use of true/false clauses; the **continue** was just one way of achieving this.

Now, let us consider another modification to the program. Right now, we are forced to enter 10 numbers; the loop will continue until all 10 have been entered. Let us change the design allowing the user to complete the program 'early'. Specifically, we will have the program quit the loop immediately if the number 0 is added. Let us add code to this now.

6. Insert the following code right after the previous **if** statement:

```
if(input == 0) {
    break;
}
```

This code is similar to what we entered before. We check to see if the input was some value (0 in this case). If it is, we issue a **break** statement. Recall that a **break** signals that Java should quit the loop completely.

Your code should look like this:

```
public static void main(String[] args) {
    int sum = 0;
    Scanner scan = new Scanner(System.in);
```

```
    for (int x = 0; x < 10; x++) {
        System.out.println("Please enter integer #" + (x + 1));
        int input = scan.nextInt();
        if (input < 0) {
            continue;
        }
        if(input == 0) {
            break;
        }
        sum += input;
    }
    System.out.println("The sum is " + sum);
    scan.close();
}
```

7. Save the code.
8. Run the `SimpleAdder` Class.
9. Enter two integers, a few negative numbers and then 0.

```
Please enter integer #1
15
Please enter integer #2
20
Please enter integer #3
-4
Please enter integer #4
0
The sum is 35
```

   As you can see, the negative numbers were ignored (thanks to the **continue** statement) and the 0 caused the loop to complete immediately (due to the **break** statement).

10. Close all open files.

4.4. Review

In this lab, you examined a looping construct; specifically, you saw how to use a **for** loop to repeatedly iterate through code. In addition to that, you saw how you can

use the **break** and **continue** statements to exhibit a finer sense of control over loop execution.

Subclasses

MODULE 5

Time for Lab: 45 minutes

In this lab exercise, you will examine the concept of subclasses and inheritance.

This lab will builds on the the code in the `constructors` lab. You will need to have completed that lab before starting this one or as an alternative you can open the solution code for the `constructors` lab and start from there.

The previous solution included very simple BankAccount class with 3 fields (**ownerName**, **accountID** and **balance**), one static field (**interestRate**) and one method (**deposit()**).

Now, let us extend the model a little. We wish to represent a **Savings** account. A savings account is more or less the same as a BankAccount (same fields, and method), but with a few extra bells and whistles.

Firstly, a savings account has an operation to calculate interest and add it to the balance. Secondly, every savings account has a minimum balance that must be maintained if interest is to be calculated.

How could we model this? The easy thing to do would be to create a new **SavingsAccount** class, give it the same fields/operations as the **BankAccount** class, and then add the savings account specific facets. We could save some code by copying and pasting a lot of the code from the BankAccount into the new SavingsAccount class.

A better approach is to use *inheritance*. In object oriented programming, a class can inherit from another class. An inheriting class (also known as the *subclass*) automatically shares all the code from its parent (also known as the superclass), and so has all its fields and methods.

So, if we created **SavingsAccount** as a subclass of **BankAccount, SavingsAccount** would automatically get the fields and method “for free”. All we would have to do is write the SavingsAccount specific code. We will do this now.

5.1. Create the SavingsAccount Class

We will now create the SavingsAccount class that subclasses the existing BankAccount class.

1. Create a new Class with the following properties:
   - Create the class in the: `com.simple.account` package
   - Name the Class: `SavingsAccount`
2. Edit the new class so that it subclasses `BankAccount` as shown below:

```
package com.simple.account;

public class SavingsAccount extends BankAccount{
}
```

   There will be some errors. This is because the class is not yet complete. We will fix this in a moment.

   Notice that the class declaration now has a new keyword: **extends**. This keyword is used to denote that the class being defined is a subclass of the class specified. In our case, the SavingsAccount **extends** BankAccount.

3. Take a look at the error message for the class.

   Java is complaining that there is no default constructor here. Why is this happening?

   Remember that a class gets the default constructor “for free” **if** it has no other constructor defined for it. The problem is because the BankAccount class **does not** have a “default” constructor since we defined a constructor in the BankAccount class. When Java tries to define a “default” constructor for the SavingsAccount class with an implicit call to the “default” constructor in the superclass this creates problems.

   So, in effect, we need to create a constructor for this new class. We will do this now.

4. Add the following constructor to the class. This constructor will take the same arguments as the BankAccount constructor:

```
public SavingsAccount(int accountID, String ownerName, float balance) {
    super(accountID, ownerName, balance);
}
```

   Make sure to add the constructor inside the curly brackets that define the class!

   When you are done the Class should look like this:

```
package com.simple.account;

public class SavingsAccount extends BankAccount {
    public SavingsAccount(int accountID, String ownerName, float balance) {
        super(accountID, ownerName, balance);
    }
}
```

5. Save the file. There should be no errors.

5.2. Test the New Subclass

Let us now see if the new SavingsAccount class works. We will create an instance in the `BankAccountTester` Class.

1. Open **BankAccountTester.java**.
2. Add the following code at the end of the **main** method.

```
SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 6000f);
```

3. Make sure to add an import statement for the new `SavingsAccount` class. The import statement should be at the top of the file, just below the package statement. It should look like this:

```
import com.simple.account.SavingsAccount;
```

4. Save the `BankAccountTester.java` file, then continue coding.

5. Add the following code to the `BankAccountTester` `main` method after the instantiation of the **SavingsAccount**.

```
System.out.println("The SavingAccount's balance is " + sAccount.getBalance());
```

6. Save the file.

7. Run the **BankAccountTester** class.

   You should see the following output:

```
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
account2's interestRate is 5.5
account3's interestRate is 5.5
The SavingAccount's balance is 6000.0
```

   Note that we use the **getBalance()** method on sAccount. Even though we never wrote a **getBalance()** method for **SavingsAccount**, the invocation still works.

   We see that the SavingsAccount is a new class - but it has the same behavior and understands the same methods as the BankAccount class. This is due to the inheritance structure.

5.3. Adding A Method

Now, let's add some business logic to the SavingsAccount class. Specifically, we will create a method called payInterest() that will update the balance based on the interest rate.

1. Open **SavingsAccount.java**

2. Before the closing curly brace } for the class, add the following method:

```
public void payInterest() {
    float newBalance = this.getBalance() * (1
        + (this.getInterestRate() / 100));
    this.setBalance(newBalance);
}
```

This method calculates the interest based on the existing balance plus the **interestRate**.

3. Save the class. There should be no errors.

   We have now created a method on SavingsAccount class!

5.4. Adding A Field

We can now add the field representing the minimum balance to the class.

1. Open **SavingsAccount.java**.
2. Add the following field declaration, right after the class declaration, before the constructor:

   ```
   private float minimumBalance = 1000f;
   ```

   Ordinarily, we would generate accessors for this field, but in the context of our code - we will never allow any other class to get or set this **minimumBalance**. We will leave it **private** and hence inaccessible to other classes. The only other code that will use this minimum balance will be our **payInterest**() method.

   What is new here is the setting of the default value for the field. The “= 1000f” chunk of code states that when the class is initialized, it should automatically set the minimumBalance to be 1000. This is handy since we are not allowing public accessors, so nobody else will be able to set it to a different value.

3. Change the payInterest() method to check the minimum balance before updating the balance, as follows:

   ```
   public void payInterest() {
       float newBalance = this.getBalance() * (1 + (this.getInterestRate() / 100));
   ```



```
    if (this.getBalance() >= this.minimumBalance) {
        this.setBalance(newBalance);
    }
}
```

All we have done is added an **if** clause right before the call to **setBalance()**

4. Save the file. There should no errors.

5.5. Test the Code

We can now test the new features of our `SavingsAccount` class.

1. Go back to the **BankAccountTester.java** class and add the following code to the **main** method, immediately after the **println** showing the **sAccount**'s balance.

```
sAccount.deposit(500f);
sAccount.payInterest();
System.out.println("The SavingAccount's new balance is " + sAccount.getBalance());
```

Note that we are invoking the **deposit()** method, which is from the superclass (**BankAccount**). We then invoke the **payInterest()** method and print out the new interest.

2. Save the `BankAccountTester.java` file

3. Run the `BankAccountTester` Class.

The output you see should include:

```
...
account3's interestRate is 5.5
The SavingAccount's balance is 6000.0
The SavingAccount's new balance is 6857.4995
```

As expected, the balance is updated. The call to **deposit()** worked, and so did the **payInterest()** call.

Now, let us try setting the balance below the minimum.

4. Locate the line in the `main` method where the constructor is called for the **SavingsAccount** and change it to:

```
SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);
```

   We set the balance to $1, which is below the minimumBalance as defined on the class. The deposit that takes place after is only for $500, which is still below the minimum balance.

5. Save and run the code.

   The output you see should include:

```
...
account3's interestRate is 5.5
The SavingAccount's balance is 1.0
The SavingAccount's new balance is 501.0
```

   We see that the balance was not updated from paying interest, even though we called **payInterest()**. It looks like the **payInterest()** logic is working.

   Now, let us do an experiment. We know that the **SavingsAccount** inherited all the **BankAccount's** methods. We've seen that we could call **deposit()** and the **get/set** methods on **SavingsAccount**. But does it work the other way? Can a BankAccount use the **payInterest()** method? Let us see.

6. Add the following code after all the other lines in the **main** method:

```
account3.payInterest();
```

7. Save the file.

   You should immediately see a problem.

   The error message that shows up is:

```
The method payInterest() is undefined for the type BankAccount
```

   This is because, a superclass cannot use methods from its subclass. This is one of the rules of inheritance; it only works "down", not up.

8. Delete the line you just entered.

9. Save the file. There should no errors.
10. Close all open files.

5.6. Review

In this exercise, you created a *subclass* of **BankAccount** called **SavingsAccount**. You saw that **SavingsAccount** *inherited* all the fields and methods of the superclass. You also added a method and field to the subclass, while keeping all the behavior of the superclass.

From this, you should see that inheritance is an excellent mechanism for sharing code between common classes.

Arrays

MODULE 6

Time for Lab: 35 minutes

In this lab exercise, you will examine the use of arrays in Java.

An array is a consecutive collection of 0 or more elements. So far, we have only been dealing with single **int**, **String** and even **BankAccount** instances. Often, however, it is desirable to deal with many at once. For example our **BankAccount** may have multiple owners, meaning that it should have multiple **ownerNames** instead of just the one that it has right now. This can be achieved through the use of arrays. We could modify our BankAccount class to use an array of String objects for the **ownerName** field instead of just the String. Let us start with something simpler, however

Note: This lab builds on the the code in the `subclass` lab. You will need to have completed that lab before starting this one or as an alternative you can open the solution code for the `subclass` lab and start from there. .

6.1. Object Arrays

We will now experiment with some simple array operations by creating an array and inserting some BankAccount objects.

Think of an array as a box of consecutive 'things' (e.g. **int**s). An array has a fixed size (e.g. 10) and each element can be accessed via an index which represents its position in the array. Elements are inserted into an array by means of an assignment operator (=), and can be retrieved via square brackets and the index of the element being retrieved.

Arrays must first be declared. Once they are declared, objects can be placed inside them.

We will create a new class for our experiments; this class will create an array of BankAccounts. and will then insert some BankAccount instances into this array. We will then see how to use indexes to access them.

1. Create a new class called **ArrayTester** in the package **com.simple.test**.

2. Add the following main method to the new Class:

   ```
   public static void main(String[] args) {}
   ```

   When you are done you should see this in the editor:

   ```
   package com.simple.test;

   public class ArrayTester {
       public static void main(String[] args) {

       }
   }
   ```

3. Add the following code to the **main** method.

   ```
   BankAccount[] accounts = new BankAccount[4];
   ```

4. Add the following import statement to the top of the file after the `package` statement:

   ```
   import com.simple.account.BankAccount;
   ```

   The code line declares the array. Note that we use the square brackets [] to indicate this is an array. Also note that we we initialize the array by giving it a *size*. In this case, the size of the array is 4. So, at this moment, the variable **accounts** points to an array of 4 **BankAccount**s. It is ready for use!

   Note that an array is *typed,* meaning that only one type of object can be placed in an array; that type of object is specified at declaration time. So, this declaration states that our array can only hold instances of BankAccounts. Four of them, to be precise.

5. Add the following code after the array declaration:

   ```
   accounts[0] = new BankAccount(1, "Jeff Lebowski", 100f);
   ```

   This line tells Java to create a new instance of a BankAccount (with id 1) and places that at element 0 in the array. Here, we see that we use the square brackets and a number [0] to indicate index position.

It is important to realize that arrays use a zero-based index. This means that the first element in the array is considered to be at index (i.e. location) 0. The second element is considered to be at index 1, and so on.

6. Add the following two lines of code:

```
accounts[1] = new BankAccount(2, "Maude Lebowski", 5000f);
accounts[2] = new BankAccount(3, "Bunny Lebowski", 1f);
```

Nothing special here; we create 2 more instances and then add them to the array.

7. Save and run the code. There should be no errors – but no output either. That is fine for now.

Realize now that we have an array of size 4, but we have only inserted 3 elements.

8. Now, let us try something else. Add the following line of code:

```
accounts[4] = new BankAccount(5, "Ulli Wenk", 14354f);
```

This does not look like anything special, but notice the index; we are trying to access *accounts[4] _ which would be the 5th position in the array; this is _out of bounds* for the array since the array is only 4 elements long.

9. Save and run the code.

An **Array IndexOutOfBoundsException** should be raised.

A nasty error has popped up, telling us that we tried to access an array outside its bounds. In the case of our code, it makes sense; we tried to index a position that was not within the array. Such a mistake (“out of bounds exception”) is a common and care must be taken to avoid this whenever possible.

10. Delete the offending line of code.

We’ve seen how to insert elements into an array (by using the assignment = operator). Let us now see how to get elements out of an array.

11. Add the following code:

```
BankAccount first = accounts[0];
System.out.println("First account has id:" + first.getAccountID());
```

The first line declares a new variable called **first** (of type BankAccount) and sets it to the first element in the array – which should be our first BankAccount. This, in effect, "pulls out" the first element of the array and sticks it inside that **first** variable. We then print out **first**'s id – which should be **1**. (We could also print out **first**'s ownerName and balance, but we will assume that if the ID is correct, the rest will be as well)

12. Save and run the code. As expected:

```
First account has id:1
```

13. Now add the following code:

```
BankAccount second = accounts[1];
System.out.println("Second account is owned by " + second.getOwnerName());
```

Similar to the previous code, we 'pull out' the account at position 1 (the second element) and then print out its **ownerName**.

14. Save and run the code.

```
First account has id:1
Second account is owned by Maude Lebowski
```

As expected, the correct account was located.

15. Finally, let us try something different. Add the following code:

```
BankAccount fourth = accounts[3];
System.out.println("Fourth account has balance " + fourth.getBalance());
```

Notice that we are attempting to access index 3, which is the *fourth* position in the array. However, we *never inserted* anything into this position! So what should happen when we execute this code? Let us see.

16. Save and run the code. What happens?

+

```
First account has id:1
Second account is owned by Maude Lebowski
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "com.simple.account.BankAcc
        at com.simple.test.ArrayTester.main(ArrayTester.java:17)
```

+ We see Java has raised a **NullPointerException.** What happened?

+ Since we never assigned **accounts[3]** to be anything, it currently has a value of **null** which is a special Java state meaning “nothing”. So, the variable **fourth** is pointing to null. Then, when we try to print out **fourth.getBalance()**, we are essentially calling **null.getBalance()** which should return an error – you cannot obtain a balance from **null**!

+ From this, we see that when declaring an array of objects, it initially sets its elements to null. (Note that if the array is composed of primitives, the *default value* of the primitive will be used instead).

1. Remove the two lines of code added above ('fourth' variable) that introduced the errors.

6.2. Use Subclass in Array

An array can store any object as an element that can be treated like the type of object declared for the type of the array. An instance of a subclass can also be treated like the superclass type. Storing an instance of subclasses in an array of the superclass type can be a powerful way to write better code. We can write code to deal with all of the objects in the group, for example all of the accounts, even if some of the objects are a more specific type of object. We will see this in action.

1. Modify the code of the first object to be created and stored in the array to be a **SavingsAccount** object instead of a BankAccount object as shown below. You will have errors until the next steps.

```
accounts[0] = new SavingsAccount(1, "Jeff Lebowski", 100f);
```

2. Add an import statement for the SavingsAccount class at the top of the file:

```
import com.simple.account.SavingsAccount;
```

3. Run the ArrayTester class and notice the output is the same as before.

```
First account has id:1
Second account is owned by Maude Lebowski
```

4. Add the `first.payInterest();` line as shown below:

```
BankAccount first = accounts[0];
System.out.println("First account has id:" + first.getAccountID());
first.payInterest();
```

5. Save the code and notice it doesn't compile. This is because the 'payInterest' method is only defined in the SavingsAccount class. Even though the object is actually a SavingsAccount object, the 'first' variable and the element of the array that refer to the object only can be sure that the object is a BankAccount.

6. Delete the line with the 'payInterest' method so the code again compiles and runs.

7. Save the file.

6.3. Add An Array To The SimpleAdder

Let us now use our knowledge of arrays to tackle something a little more complex.

1. Open the **SimpleAdder.java** code that you wrote in a previous lab under the package `com.simple`.

2. If you need a copy of this code, it is listed below:

```
package com.simple;
import java.util.Scanner;

public class SimpleAdder {
        public static void main(String[] args) {
                int sum = 0;
                Scanner scan = new Scanner(System.in);
                for(int x = 0; x < 10; x++) {
```

```
                        System.out.println("Please enter integer #" + (x+1));
                        int input = scan.nextInt();
                        if (input < 0) {
                                continue;
                        }
                        if(input == 0) {
                                break;
                        }
                        sum += input;
                }
                System.out.println("The sum is " + sum);
                scan.close();
        }
}
```

Currently, it just loops and prompts the user to enter a number 10 times before printing out the sum. Each number that is entered by the user is added to the sum and then "thrown away". What if we wished to change the design of the program such that it remembered all of the numbers that are entered? That way, at the end of the program, the code could display all of the numbers that were entered in addition to the sum.

We will make this change and use an array to help us with this.

As we iterate through the loop, we will want to "save" each number that is entered. We can do this using an array.

At the start of our code, we will declare a new array (of size 10). As we iterate through the loop, we will take every number that the user enters and insert it into the array.

Then, at the end of the code (after the prompting loop), we will iterate through the array and print out its contents. Let us do this now.

3. In the **SimpleAdder.java** main method, locate the following line:

```
Scanner scan = new Scanner(System.in);
```

4. Immediately after this line, add the following code:

```
int numbers[] = new int[10];
```

5. Inside the loop body, locate the line:

```
sum += input;
```

6. Immediately after this line, add the following code:

```
numbers[x] = input;
```

This line is saying “Put the number that was just entered into the **x**th position in the array”.

This works nicely for us because our loop starts with **x** being zero. Recall that **x** is our loop variable; we are using it to keep track of how many times we have been through the loop. So, on the first iteration, the value of **x** is zero; accordingly, the first number that the user enters will be placed in **numbers[0]** which is the *first* element in the array. The second iteration will place the number in **numbers[1]** and so on. It is important that although the array has a size of 10, it is indexed from 0 to 9!

Finally, after the loop has completed, we should iterate through the array and print out all the numbers that were entered.

7. Locate the line:

```
System.out.println("The sum is " + sum);
```

8. And enter the following code immediately before it:

```
System.out.println("The numbers you entered were: ");
for(int eachNumber : numbers) {
    System.out.println(eachNumber);
}
```

This code sets up another loop, which uses the "enhanced for loop" or "for-each loop" syntax to iterate over the 'numbers' array and set a variable 'eachNumber' to the current element of the array to display.

9. Save the code. There should be no errors.

10. Run the code.

```
9
Please enter integer #10
10
The numbers you entered were:
```

```
1
2
3
4
5
6
7
8
9
10
The sum is 55
```

11. Close all open files.

    Notice that the numbers you entered were printed out. We have used the array as a storage mechanism and iterated through it using loops.

6.4. Review

In this lab, you examined arrays. You saw how an array can be used to store a series of objects, and how they can be accessed using indexes. You also saw how loops can be used to iterate through arrays.

Method Overriding

MODULE 7

Time for Lab: 45 minutes

A common practice in object oriented programming - especially in inheritance cases - is to *override* methods. Simply put, to override a method means to change the logic of an inherited method with the same name. We will examine this here.

This lab will builds on the the code in the `arrays` lab. You will need to have completed that lab before starting this one or as an alternative you can open the solution code for the `arrays` lab and start from there.

7.1. Adding a Print Method

1. Open the **BankAccountTester.java** class.

   Right now, every time we wish to inspect a BankAccount instance, we are using a sequence of **println** statements. This is tedious and unnecessarily complex.

   A better approach would be to write a *method* to do this for us. Ideally, we could write a method on the **BankAccount** class, called **print()** which would print the data of the current BankAccount to **System.out**. Let us do this now.

2. Open **BankAccount.java**.

3. Add the following method to the class.

```
public void print() {
    System.out.println("\nAn Account");
    System.out.println("Account ID:" + this.getAccountID());
    System.out.println(" Owner:" + this.getOwnerName());
    System.out.println(" Balance:" + this.getBalance());
}
```

   This method prints out the field information to the console. (The '\n' character is to force the print to go to a new line. This makes the output easier to read.)

4. Save the class.

   We can now simplify the BankAccountTester code quite a bit.

5. Open **BankAccountTester.java**.

6. Right now, it includes a lot of **println** statements. Let us start clean. Delete *all* the code inside the **main** method *except* for all the constructor calls. When you are done, it should look like this:

```
public static void main(String[] args) {
        BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
        BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
        BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
        SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);
}
```

7. Let's test our new **print** method. Add the following code to the **main** method after the constructor calls:

```
account.print();
account2.print();
account3.print();
```

8. Save and run the code.

```
An Account
Account ID:1
 Owner:Jeff Lebowski
 Balance:100.0

An Account
Account ID:2
 Owner:Bunny Lebowski
 Balance:5000.0

An Account
Account ID:3
 Owner:Walter Sobcheck
 Balance:1000000.0
```

   Examine the output. You may have to re-size the console view to see the entire output. It looks like our **print** method is working.

9. Let us see if the **print** statement works on our **SavingsAccount**. Add the following code at the end of the `main` method:

```
sAccount.print();
```

10. Save and run the code.

    You should see the following in the output:

```
...
An Account
Account ID:3
 Owner:Donny K
 Balance:1.0
```

    We see that our **SavingsAccount** is indeed printed out. This is thanks to inheritance.

7.2. Printing the SavingsAccount

What if, however, we wanted the **SavingsAccount** to print out the interestRate as well as the minimumBalance required?

We could go back to the BankAccount class and edit the **print** method to include those fields, but that may not be a good solution for two reasons: firstly, we are not interested in seeing the interestRate for a regular BankAccount and secondly, the BankAccount itself does not have a minimumBalance field – and hence would not be able to print it out.

It is clear that we will need to add a method to the **SavingsAccount** class. So, we could add a method called savingsAccountPrint() that would print all the fields. That, however, is not convenient because any class using the SavingsAccount would have to be aware of both the **print()** and the **savingsAccountPrint()** methods. A better approach, however, would be *override* the **print** method.

Overriding the method means to implement an inherited method with the same name. This means we will create a method called **print** (with the same method signature) as the superclass's **print** method. Now, when code calls .print() on a SavingsAccount instance, this code will be executed instead. We will override this method now.

We will keep things simple for now. Our code will just print the **interestRate** and **minimumBalance**.

1. Open **SavingsAccount.java**.
2. Add the following method to the class:

```
public void print() {
    System.out.println("\nSavings Summary:");
    System.out.println(" Interest rate:" + this.getInterestRate());
    System.out.println("Minimum Balance:" + minimumBalance);
}
```

   Note that we use the *get* method to get the interest rate, but do not use a get method for the minimum balance. Can you explain why?

3. Save the code. There should be no errors.
4. Switch back to the **BankAccountTester**. Without changing any code, run the class.

```
...
Savings Summary:
 Interest rate:0.0
Minimum Balance:1000.0
```

   It looks like the **print** method is working well! However, as we stipulated, it is only printing the interest rate and minimumBalance. What if we want the account id, owner name and balance to be printed as well? This is easy to fix.

5. Go back to **SavingsAccount.java**.
6. In the **print** method, add the `super.print()` method as shown below:

```
System.out.println("\nSavings Summary:");
super.print();
System.out.println(" Interest rate:" + this.getInterestRate());
System.out.println("Minimum Balance:" + minimumBalance);
```

   Here, we invoke the **print** method from the superclass. Let us see the effect this has.

7. Save the class.

8. Switch back to and then run the **BankAccountTester**.

    +

```
...
Savings Summary:

An Account
Account ID:3
 Owner:Donny K
 Balance:1.0
 Interest rate:0.0
Minimum Balance:1000.0
```

+ Much better!

+ Now, when we call the **print** method on either a **BankAccount** or **SavingsAccount** instance, the appropriate text will be printed. The one **print** method will behave differently, based on whichever class is invoking it. Anyone using either of these classes will only have to know about the one **print** method. This makes for a much simpler programming model.

7.3. Using System.out.println

So, we have now written a **print** method for our classes. There is one tiny issue, however. When printing to the console it is not common to call a special method on a class instance; instead, it is more common to call **System.out.println()** on the object we wish to see on the console. We have not done that so far; why not? Why have we not just been able to call **System.out.println(account)**? After all, we could call this method on other **String**s and **int**s.

Let us see why.

1. Open **BankAccountTester.java**.

2. Add the following line of code at the end of the **main** method.

```
System.out.println(account);
```

3. Save and run the code. What do you see?

    +

```
...
com.simple.account.BankAccount@1c20c684
```

+ Not quite what we were expecting, is it?

+ When **System.out.println()** is invoked on an object, Java tries to convert that object into a String first, and it is that String that is sent to the console.

+ To convert an object to a String, **System.out.println()** will call a method named **toString()** on the class being printed. That method should return a String representation of the class being printed, and it is that String that is sent to the console.

+ But wait a moment: we have no such **toString()** method on our classes. Where is it coming from? Answer: **Object**

+ In Java, all classes ultimately inherit from a class called **Object**.

+ So, the text that is printed right now "com.simple.account.BankAccount@addbf1" is the result of the **toString()** call of **Object()**.

+ If we do not like this, we can simply *override* the **toString** method in our classes! We will do this now.

1. Open **BankAccount.java**.

    We will override the method here.

2. Add the following method to the end of the class (after all the other methods):

```java
@Override
public String toString() {
    return "An account with id " + this.getAccountID() +
        " with balance " + this.getBalance() +
        " owned by " + this.getOwnerName();
}
```

Note the use of the + signs to concatenate a longer String. It is this String that is returned.

3. Save the code. There should be no errors.

4. Switch back to **BankAccountTester** and run it.

5. Replace all the calls to **.print()** call shown here:.

```
account.print();
account2.print();
account3.print();
sAccount.print();
```

With these lines:

```
System.out.println(account);
System.out.println(account2);
System.out.println(account3);
```

Your code should now look like this:

```
public static void main(String[] args) {
        BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
        BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
        BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
        SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);

        System.out.println(account);
        System.out.println(account2);
        System.out.println(account3);
}
```

6. Save and run the code. What do you see?

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
```

From a functional perspective, it is more or less the same as the old **print** methods that we were using. However, from a coding perspective, this was a better approach. **System.out.println()** is commonly used by developers everywhere, and adding the **toString()** method as we did guaranteed that calling **System.out.println()** on the BankAccount will behave in a proper way.

Let us see what happens when we try this on the **SavingsAccount** class.

7. Add the following line to the **main** method of the BankAccountTester class.

```
System.out.println(sAccount);
```

8. Save and run the class.

   You should see the following:

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K
```

   It worked in exactly the same way as calling **println** on the **BankAccount** did. This is because the **SavingsAccount** *inherited* the **toString()** method from **BankAccount**.

7.4. Override toString In The SavingsAccount

As before, we should see the interestRate and minimumBalance for the SavingsAccount. How can we make this happen? Simple; as we did with the **print** method, we can override the **toString** method in **SavingsAccount**.

1. Open **SavingsAccount.java**.

2. Add the following method after all the other methods:

```
@Override
public String toString() {
    return super.toString() +
        " with interest rate " + this.getInterestRate() +
        " and minimum balance " + this.minimumBalance;
}
```

   Note the call to the **super.toString()** method.

3. Save the code. There should be no problems.

4. Go back to the **BankAccountTester** and run the code. You should see the following:

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and
minimum balance 1000.0
```

   As expected, the **SavingsAccount** output has changed.

   Congratulations! You have successfully overridden a method from **Object** and made using **System.out.println()** much simpler!

5. Close all open files.

7.5. Review

In this exercise, you examined method overriding and saw how it can be used to increase functionality of classes in an inheritance structure.

You also saw one of the core methods on the **Object** class (**toString()**) and used method overriding on that method to make **System.out.println()** usable with our classes.

Exception Handling

MODULE 8

Exceptions are a mechanism that Java uses as a notification that something has gone wrong. Erroneous code can *throw* an exception; any code that calls that erroneous code can *catch* that exception. Catching an exception typically entails handling it; handling an exception might include things like re-attempting the code, or even ignoring the exception completely.

An Exception in Java is a regular Object, like String or BankAccount. They can be initialized and created using a constructor, and have their own hierarchy. It is also possible to write your own Exception class.

Note: This lab builds on the the code in the `method-override` lab. You will need to have completed that lab before starting this one or as an alternative you can open the solution code for the `method-override` lab and start from there.

This lab exercise will examine exception handling in Java.

8.1. The Try-Catch Block

Any code that can potentially throw an exception (i.e. any code that could cause a failure) should be surrounded with a **try** block. Immediately following the **try** block should be a **catch** block.

Any code throwing an error within the **try** block will cause the code in the **try** block to cease executing, and control will be turned over to code in the **catch** block.

Think back to the **SimpleAdder.java** that we wrote in an earlier lab. If you recall, this program would iterate through a loop and ask the user to enter a number.

1. Open **SimpleAdder.java**.
2. Run it and enter a String instead of a number. What is returned?

```
Please enter integer #1
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
```



```
        at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
        at com.simple.SimpleAdder.main(SimpleAdder.java:14)
```

We see here that Java has thrown a **java.util.InputMismatchException** and exited the program ungracefully.

Immediately following the exception is a *stack trace*. This shows the sequence of method calls that led up to the exception. If you look at the bottom line of the stack trace, we can see that the exception was initially caused by code on line 13 of our SimpleAdder.java class, in the **main** method.

The error is actually occurring because of the call to **scan.nextInt()**. This method is expecting an **int** to be entered; when a String was entered instead, Java threw an exception; our code did not handle the exception, so Java handled it in its own way (i.e. by exiting ungracefully).

What we need to do is modify the code and **try** the call to **scanInt()** and follow that with a **catch** of our particular exception type (**java.util.InputMismatchException**)

3. In **SimpleAdder.java** locate the line

```
int input = scan.nextInt();
```

4. Delete it and replace it with the following code:

```
int input = 0;
try {
    input = scan.nextInt();
} catch (java.util.InputMismatchException exception) {
    System.out.println("Please enter only numeric values.");
    break;
}
```

This is a **try/catch** block. The **try** block (denoted by the curly braces) is the code that Java will 'try' to execute. The **catch** block specifies what sort of error it is looking for; in this case, our code is on the lookout for an **InputMismatchException**.

Note that the syntax of the catch clause; it specifies the *type* (**InputMismatchException**) of the exception it is expecting and assigns it a

variable name – in this case, the variable is simply named **exception**. This variable exists because the exception object itself may need to be referred to from within the body of the catch message. We will see an example of this later.

The InputMismatchException is an exception that is a part of the Java language. Java itself contains an entire hierarchy of exceptions; InputMismatchException is one of them, and it just so happens that it is potentially thrown by **scan.nextInt()**. Other exceptions that we have seen so far: **NullPointerException** and **ArrayIndexOutOfBoundsException**

If the code within the **try** block encounters a String instead of the expected **int*_, _it will 'throw' an *InputMismatchException** which will then be 'caught' by the **catch** block. There, we put our error handling code.

In this case, our handling code in the **catch** block is simple; it merely notifies the user that the wrong input type was entered, and then exits the loop. It may seem quite simple, but note that it is much nicer than seeing a stack trace and then exiting ungracefully.

Let us now test our code and see what happens.

5. Save the `SimpleAdder` class and run it.
6. Again, enter a String instead of a number. What do you see?

```
Please enter integer #1
One
Please enter only numeric values.
The numbers you entered were:
0
0
0
0
0
0
0
0
0
0
The sum is 0
```

This is much better. Instead of seeing the stack trace and ungraceful exit, our code continues and prints out the data that it had received. (A nasty side effect is all the extra 0's that are printed out because they were in the array; this is easily fixed, however, and we will not worry about this for now)

This shows that by adding a simple **try/catch** block, we have started using good (but not great) error handling code.

8.2. Creating Your Own Exception

In addition to using the exceptions provided by Java itself, it is possible to create your own exception class to handle situations specific to your program. We will do this now.

Firstly, we will add a new method on our BankAccount called **withdraw**, which will take one parameter: the *amount* to be withdrawn. This method will have some very simple error checking code; if the amount to be withdrawn is larger than the current balance, an **InsufficientFundsException** will be thrown. We will create the InsufficientFundsException class ourselves.

We will start things off by creating our exception class. An exception class can, at its simplest, merely extend the **java.lang.Exception** class. Recall that extending another class implies inheriting all its functionality, including all its methods.

1. Create a new Class called **InsufficientFundsException** in the package **com.simple.account**.

2. After the Class is created, change its *Superclass* to be **java.lang.Exception**

3. Add the following constructor:

```
public InsufficientFundsException(String message) {
    super(message);
}
```

And our exception class is done! We do not actually have to add any code here; all the required functionality will be inherited from the superclass.

4. Save the new Class.

   Now, we can add the method to our BankAccount.

5. Open **BankAccount.java**.

6. Add the following method to the class:

```
public void withdraw(float amount) throws InsufficientFundsException {
    if (amount > this.getBalance() ) {
        throw new InsufficientFundsException("Amount " +
            amount + " exceeds balance " + this.getBalance());
    }
    this.setBalance(this.getBalance() - amount);
}
```

   A few things to note here. Firstly, note that the method declaration has a **throws** clause. This specifies that any code trying to use this method must be aware that an **InsufficientFundsException** might occur and that it **must** be handled. This is a good way of forcing any code that calls this to have proper error handling logic in place.

   Secondly, note the error checking logic; we check to see if the amount to be withdrawn is larger than the balance; if it is, a new instance of our **InsufficientFundsException** is created and **thrown**. Note that the constructor of the class takes a single string as a parameter, and the string is simply a message saying what the problem is.

   Finally, the code goes ahead and decrements the balance. Note that this code will *not* be executed if the exception is thrown.

7. Save the code. There should be no errors.

   Now, let us test this method.

8. Open **BankAccountTester.java**.

9. Add the following code at the end of the **main** method.

```
account.withdraw(20f);
```

10. Save the file.

11. What do you see?

    The following problem has come up: "Unhandled exception: com.simple.account.InsufficientFundsException". This is because we are trying to call the **withdraw** method, which has a **throws** clause in its declaration. This means that any code calling this method must handle the exception. In this case, we are not yet handling it.

    To fix this you will need to add a try/catch around the problematic line of code.

12. Replace the `account.withdraw(20f);` line with the following code:

```
try {
        account.withdraw(20f);
} catch (InsufficientFundsException e) {
        e.printStackTrace();
}
```

    Remember: the code in this **catch** clause will only be executed if the code within the try block throws the appropriate exception.

    We will change the code inside the catch block; instead of printing the stack trace, we want to see the *message* of the exception that was caught.

13. Locate the line:

```
e.printStackTrace();
```

14. Delete it. In its place, enter:

```
System.out.println("There was an error withdrawing funds");
System.out.println(e.getMessage());
```

    Note that we use the variable **e**. **e** is the exception object; it is declared in the **catch** clause. It refers to the actual exception object which has been thrown, and we can get information from it. Here we get the *message* from the exception. Where does this *message* come from? Simple: it was passed into the constructor of the exception instance, in our **withdraw** method.

15. Save the file.

8.3. Run the Code and Trigger the Exception

1. Run the **BankAccountTester** class.

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and
minimum balance 1000.0
```

   Nothing interesting; pretty much the same as before. The **withdraw** is working properly because we are trying to withdraw 20.0 from account 1 which had a balance of 100.0

   Let us trigger a defect by changing the amount to be withdrawn.

2. Locate the line:

```
account.withdraw(20f);
```

3. Change it to:

```
account.withdraw(200f);
```

4. This should trigger a defect, which our code should catch.

5. Save and run the code.

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and
minimum balance 1000.0
There was an error withdrawing funds
Amount 200.0 exceeds balance 100.0
```

   It looks like our error handling code was indeed executed. The **withdraw** method noticed an error condition and raised an appropriate exception; the BankAccountTester which was using the method caught that exception and then handled it gracefully if rather simplistically. (Better error handling code might ask the calling class to re-attempt the deposit with a smaller amount)

Congratulations! You have created your own exception class and seen how to throw/handle it.

6. Close all open files.

8.4. Review

In this lab, you examined exceptions. You saw how code can be surrounded with a **try** block and a particular exception can be caught with a **catch** block. You also saw how to create your own exception class; finally, you saw how declaring a **throws** clause in a method declaration can force any calling code to handle a potential exception scenario.

Interfaces

MODULE 9

Time for Lab: 30 minutes

In this exercise, you will examine the use of Java *interfaces*.

So far, when dealing with methods, we have seen that Java is a typed language. When you write a method that takes parameters, you specify the parameter types. Similarly, when a method returns an object, that object type is specified in the method signature.

Consider the following two methods, taken from our `BankAccount` class.

- `public String getOwnerName()` - the method signature here indicates that a **String** is returned
- `public void setOwnerName(String newName)` – here, we state the argument must be of type String

In either case, not using the correct type would cause a problem. Consider:

- `int result = account.getOwnerName();` would fail because we are attempting to assign an int to a String return.
- `account.setOwnerName(2333);` would fail because we are passing an int instead of a String.

Both of these statements would cause compile time errors, and the code would never run.

We can see that typing enforces a degree of correctness in code; after all, if we force the correct object type into a method signature, it is less likely that careless Java coding can occur.

However, there are some cases where we want some degree of flexibility; it might be desirable for a method to accept different types. We will see how this is possible using *interfaces*.

Let us consider a completely new class to add to our BankAccount model. Think of an **AccountManager** class. An **AccountManager** is essentially just a holder for an account. It has one field (called **account**) which represents the account it manages. It can perform operations on the account (like managing storage, batch updating,

etc) and even broker withdrawals. We can say that the **AccountManager** will act as a public interface to the account that it holds.

Right now, to perform deposits and withdrawals on an account, we are accessing the BankAccount directly; for encapsulation purposes, we will want our AccountManager to act as the front end for that. An end user could operate an ATM teller, and the ATM teller would operate on an instance of an AccountManager. The AccountManager would then pass deposit/withdrawal requests to the BankAccount instance that it manages.

We will use this class as a basis for our *interface* experiments. Let us create this class.

Note: This lab builds on the the code in the `exceptions` lab. You will need to have completed that lab before starting this one or as an alternative you can open the solution code for the `exceptions` lab and start from there.

9.1. Create the AccountManager Class

1. Create a new Class in the `com.simple.account` package called **AccountManager**.

   The created class should look like this.

   ```
   package com.simple.account;

   public class AccountManager {

   }
   ```

2. Add a field called **account** of type **BankAccount** as follows:

   ```
   private BankAccount account;
   ```

3. Add getter and setter methods for this field:

   ```
   public BankAccount getAccount() {
       return account;
   }
   ```

```
public void setAccount(BankAccount account) {
    this.account = account;
}
```

4. Now, add the deposit/withdraw methods.

```
public void deposit(float amount) {
    this.getAccount().deposit(amount);
}
public void withdraw (float amount) throws InsufficientFundsException {
    this.getAccount().withdraw(amount);
}
```

   Note that these methods are just *wrappers* for the deposit/withdraw methods on the account themselves. These methods do not actually have any withdraw/deposit business logic, but rather delegate it to the account itself.

   Remember, this **AccountManager** class is essentially a public wrapper class for our old BankAccount.

5. Save the code. There should be no errors.

9.2. Test the AccountManager

We will now create a tester class to make sure our new **AccountManager** works.

1. Create a Class in the **com.simple.test** package called **AccountManagerTester**.
2. Add a **main** method:

```
public static void main(String[] args) {}
```

3. Add the following code to the **main** method:

```
AccountManager manager = new AccountManager();
BankAccount account = new BankAccount(1, "Jeff Lebowski", 1000f);
```

   We create a new instance of a BankAccount as well as an AccountManager.

4. Add the following imports to the Class:

```
import com.simple.account.AccountManager;
import com.simple.account.BankAccount;
```

5. Continue adding the following code in the `main` method:

```
manager.setAccount(account);
```

   Remember that the manager will act as a broker for our account; accordingly, we set the account as an instance of the manager.

   Now, we can actually perform withdrawals and deposits via the manager.

6. Add the following code to test this.

```
manager.deposit(400f);
try {
    manager.withdraw(20f);
} catch (InsufficientFundsException e) {
    System.out.println(e.getMessage());
}
```

   We perform a simple deposit and then a withdrawal. Notice that the **withdrawal** invocation has to be wrapped with a **try/catch**. (Look at the method signature for **withdrawal** in the **AccountManager** class; it has the **throws** clause.)

7. Add an import for the `InsufficientFundsException`.

8. Finally, let's add some output.

```
System.out.println("Managing " + manager.getAccount());
```

9. Save and run the `AccountManagerTester`. The output should be expected:

   `Managing An account with id 1 with balance 1380.0 owned by Jeff Lebowski`

   Looks like it worked.

   We can now use the AccountManager to handle all operations on the BankAccount. This might seem strange; after all, why add this extra level of abstraction? What was wrong with accessing the BankAccount directly?

The reasons are twofold: firstly, this AccountManager could act as a **UI** (user interface) class which is capable of interacting with the end user (drawing menus, handling button presses, etc) while leaving the actual account logic on the BankAccount itself. While we will not actually be doing this, it is a common practice.

Secondly, this AccountManager will act as a perfect example of using *interfaces*, which is the whole reason we are performing this lab exercise.

9.3. Changing The AccountManager

We've seen that the **AccountManager** can work with instances of **BankAccounts**. But can it work with **SavingsAccount**s? Let's try this.

1. Open the **AccountManagerTester.java**.
2. Locate the line:

```
BankAccount account = new BankAccount(1, "Jeff Lebowski", 1000f);
```

3. Change it to:

```
SavingsAccount account = new SavingsAccount(1, "Jeff Lebowski", 1000f);
```

   What we are doing here is making the AccountManager handle an instance of a SavingsAccount instead of a regular BankAccount.

4. Add an import for `SavingsAccount`.
5. Save and run the `AccountManagerTester`.
6. The output returns:

   `Managing An account with id 1 with balance 1380.0 owned by Jeff Lebowski with interest rate 0.0 and minimum balance 1000.0 `

   Looks like it worked! So, the **AccountManager** can work with **BankAccounts** and **SavingsAccounts**.

This is possible because the BankAccount and SavingsAccount are related by inheritance; since a SavingsAccount is a subclass of BankAccount, any method that can work with a BankAccount can work with a SavingsAccount.

Let us consider, however, another type of account. Say we now wish to have a new account type called **BusinessAccount**. A **BusinessAccount** is an account that is used by corporate entities. It is similar to the regular BankAccount, in that it has a **balance**, and an **ID**; however it will not have an **ownername**; instead it will have **companyName** and **companyAddress** fields.

Could we model this class as subclass of BankAccount? Unfortunately, not; because our BusinessAccount does not have an **ownerName**, forcing it to inherit would not make for good object oriented behavior. So, it will not be able to belong to the same inheritance hierarchy. Let us go ahead and create this class.

9.4. Create the BusinessAccount Class

1. Create a new Class in the **com.simple.account** package called **BusinessAccount**.

   The Class should look like this:

   ```
   package com.simple.account;

   public class BusinessAccount {
   }
   ```

2. Add the following fields:

   ```
   private float balance;
   private int accountID;
   private String companyName;
   private String companyAddress;
   ```

3. Follow instructions in `your IDE intro lab` for generating getters and setters. Use the instructions to generate getter and setter methods for each of the four fields.

4. Add the following constructor to the Class:

```
public BusinessAccount(float balance, int accountID, String companyName, String
companyAddress) {
    this.balance = balance;
    this.accountID = accountID;
    this.companyName = companyName;
    this.companyAddress = companyAddress;
}
```

5. Add a **toString** method as follows:

```
public String toString() {
    return "A BusinessAccount belonging to " + this.getCompanyName()
        + " with balance " + this.getBalance();
}
```

6. Add the deposit method as follows:

```
public void deposit(float amount) {
    this.setBalance(this.getBalance() + amount);
}
```

7. Finally, add the withdraw methods as follows:

```
public void withdraw(float amount) throws InsufficientFundsException {
    if (amount > this.getBalance() ) {
        throw new InsufficientFundsException("Amount " + amount +
            " exceeds balance " + this.getBalance());
    }
    this.setBalance(this.getBalance() - amount);
}
```

8. Save the code. There should be no errors.

   We have now created a new class; even though it is semantically similar to the **BankAccount** and **SavingsAccount** classes, it was not similar enough in implementation to warrant being a subclass of either of those two classes.

9.5. Managing the BusinessAccount

We want the **AccountManager** to still be able to manage a **BusinessAccount**. Can this be done? Let us try this.

1. Open **AccountManagerTester.java**.
2. Locate the line:

   ```
   SavingsAccount account = new SavingsAccount(1, "Jeff Lebowski", 1000f);
   ```

3. Change it to the following:

   ```
   BusinessAccount account = new BusinessAccount(1000f, 1, "SimpleCorp", "123 Fake Street");
   ```

   Note here that we use the 4 argument constructor for the **BusinessAccount**.

4. Check your imports and add an import for `BusinessAccount` if needed.
5. An error occurs that indicates that the **setAccount** method cannot accept a **BusinessAccount** as a parameter.

   The problem is in the **setAccount** call. **setAccount** expects an instance of a **BankAccount** or one of its subclasses; we are passing in a **BusinessAccount** which is neither of those. Java's *type* behavior is now causing a problem.

   (Ignore this error for now; it will be fixed shortly.)

   So what can we do? We already said that we cannot make the **BusinessAccount** a subclass of the **BankAccount** hierarchy. We could try *method overloading;* this means writing duplicate methods for each type (e.g. **setAccount(BankAccount)** as well as **setAccount(BusinessAccount)** but that would lead to unnecessarily complicated code.

   A better idea is to use an **interface**.

9.6. Examining Interfaces

An interface is an abstract definition of a class. An interface has no code, but instead lists several methods and fields. A class can declare that it conforms to an interface by **implementing** it.

Think of an interface as a contract. You want to ensure that classes behave a certain way by having certain methods, but you do not want to use inheritance. You can write this contract to stipulate that any implementing class has code for those methods (but you do not actually specify the logic for the methods; just the names). So if a class implements an interface, it conforms to the contract.

This helps us because we can use an interface in a method signature.

Think about our account problems. The **AccountManager** needs some form of account. But really, according to the **AccountManager**, all an account really is is a class that has a **deposit** and a **withdrawal** method. Our **BankAccount**, **SavingsAccount** and **BusinessAccount** all conform to this.

What we can now do is write an interface to represent such an account. Then we can change our **AccountManager** to operate on the *interface* instead of the actual object type.

So, what we will do is this: we will firstly create an _interface _(called **Account**) that has two operations: **deposit** and **withdraw**. We will then change the **BankAccount** and **BusinessAccount** classes to say they **implement** the **Account** interface. Finally, in the **AccountManager** class, we will change the method signature of the **setAccount()** method to use our new more general interface **Account** instead the more specific BankAccount class. Our tester code should then work; the **AccountManager** will be able to use any of our **Account** types.

Let us start by creating the interface.

1. Using instructions from `your IDE intro lab` create a new `interface` in the **com.simple.account** package called **Account**.

   The `Interface` should look like this:

```
package com.simple.account;

public interface Account {

}
```

Remember, an interface is just a contract that outlines what methods an implementing class must provide code for. It does not provide code itself.

2. Add the following two method declarations to the interface.

```
public void withdraw(float amount) throws InsufficientFundsException;
public void deposit(float amount);
```

3. Save the code.

4. Our contract is complete.

   We can now state that our classes (**BankAccount** and **BusinessAccount**) conform to this contract.

5. Open **BankAccount.java** and change the class declaration to the following:

```
public class BankAccount implements Account{
    ...
}
```

   All we do here is add the **implements** keyword and specify what interface this class is implementing.

6. Save the code.

7. Now, open **BusinessAccount.java.** As before, change the class declaration to the following:

```
public class BusinessAccount *implements Account* {
    ...
}
```

   What exactly is the implication of doing this? The implication is that these **BusinessAccount** and **BankAccount** classes *must* have code for the **deposit()** and **withdraw** method. By doing so, they can say that they conform to the contract specified by the **Account** interface.

8. Test this. In the **BusinessAccount.java** file, locate the method **deposit()** and rename it to something else (like **deposit2()**).

```
public void deposit2(float amount) {
    //
}
```

9. Save the file.

   A problem appears with the Class. It is no longer fully implementing the interface. Unless the methods specified on the interface exist in the implementing class, Java will throw an error. This forces us to conform to the contract.

10. Undo the change (i.e. rename the method back to **deposit())** and save the file. This error should go away.

9.7. Update the AccountManager

We can now tell the **AccountManager** class to use the more generic **Account** interface in its method signatures.

1. Open **AccountManager.java**.

2. Locate the field declaration:

```
private BankAccount account;
```

3. Change it to use the interface as follows:

```
private Account account;
```

4. Now, locate the code for the getter method:

```
public BankAccount getAccount() {
    //
}
```

5. Change its return type as follows:

```
public Account getAccount() {
    //
}
```

6. Finally, locate the code for the setter method:

```
public void setAccount(BankAccount account) {
    //
}
```

7. Change the parameter declaration as follows:

```
public void setAccount(Account account) {
    //
}
```

8. Save the `AccountManager` file.

   There should be no errors. Additionally, the error that we had before (when we first switched the **AccountManagerTester** to use a **BusinessAccount**) should be gone.

   What have we done here? Simple. We have changed the **AccountManager** to no longer work on instances of BankAccount. Instead, we have coded it to use any class that implements the **Account** interface – which is precisely a category that our **BusinessAccount** and **BankAccount** both fit into.

   Note that in the **deposit** and **withdraw** methods, our **AccountManager** is still able to invoke the **deposit** and **withdraw** methods of the field; this is because the field **account** is an implementation of our **Account** interface – which we know for a fact will have **deposit** and **withdraw** methods on it.

9. Switch back to the **AccountManagerTester**. There should be no errors here.

10. Run the code.

```
Managing A BusinessAccount belonging to SimpleCorp with balance 1380.0
```

    Everything works. The **AccountManager** is working with the **BusinessAccount**, as it did with the **BankAccount** before it. Switching to an interface has given us flexibility.

11. Close all open files.

9.8. Review

In this exercise, we examined the concept of interfaces. An interface is a contract; a class that **implements** an interface is guaranteed to conform to the contract (i.e. has code for methods specified in the contract).

We saw how interfaces can help with class 'compatibility'. Even though classes may not be related by an inheritance relationship, they might both have a similar functional appearance, and that common appearance can be expressed using an interface.

Method signatures can then be changed to use the interface for return and parameter types, allowing for greater flexibility and cleaner code.

JUnit

MODULE 10

In this exercise you will use JUnit to develop a class by writing the tests first.

10.1. Create a Java project with JUnit Support

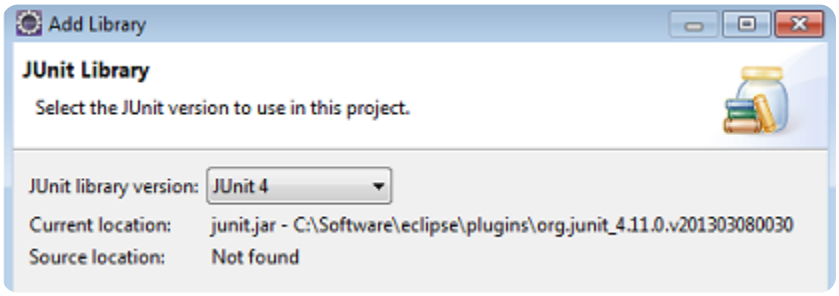
In this part you will create projects related to the testing. We will follow the best practice of creating separate projects for the code being tested and the tests themselves. This may not always be possible but it is a good idea when possible.

1. From the Eclipse menus, select **File** → **New** → **Java Project**.
2. Enter **JUnit** as **Project Name** and click **Next**.
3. On the *Java Settings* dialog, uncheck the Create module-info.java file checkbox.
4. Click **Finish**. Click yes if asked to switch perspective.
5. From the menu, select **File** → **New** → **Java Project**.
6. Enter **JUnitTest** as **Project Name** and click **Next.**
7. On the *Java Settings* dialog, uncheck the Create module-info.java file checkbox.

> ⓘ Note
>
> Although it is not required to create a project with 'Test' in the project name, it is common practice to name projects for test code to easily identify what they are intended to test.

8. Click **Finish**.
9. Right click the **JUnitTest** project and select **Build Path** → **Configure Build Path...**
10. Select the **Projects** tab in the dialog and choose **Classpath** then click the **Add** button.
11. Check the **JUnit** project and press the **OK** button.

12. Select the **Libraries** tab in the dialog and choose **Classpath**, then click the **Add Library...** button.
13. Select **JUnit** and click the **Next** button.
14. Change the drop-down to select **JUnit 4**



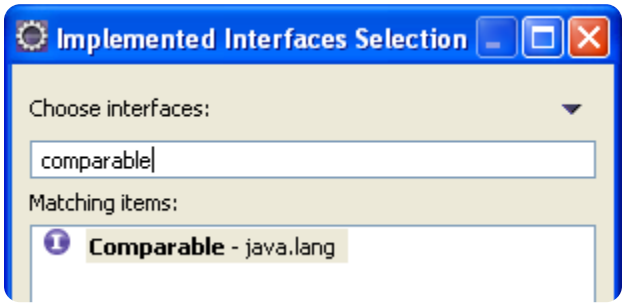
15. Click the **Finish** button.
16. Click the **Apply and Close** button to close the build path properties.

> ⓘ Note
>
> These settings will allow the test classes to compile with the classes being tested on the classpath. This is required to write the tests but this setup allows the classes “under test” to be unaware of the tests. Also notice that JUnit support is already included in the basic Eclipse plugins.

10.2. Create a Java Class and Test Class

In this part you will create a Java class and the class that will be used to test it.

1. Right click the **JUnit** project and select **New → Class**.
2. Fill in a package of **com.webage.domain**
3. Fill in a class name of **Money** but do not yet create the class.
4. Press the **Add** button next to the list of implemented interfaces.
5. Fill in '**Comparable**' to select the **java.lang.Comparable** interface and press the **OK** button.



6. Click the **Finish**.
7. Define the class Money as follows:

```
public class Money implements Comparable<Money> {
    int dollars;
    int cents;
    String currencySymbol;
    public Money(int dollars, int cents, String currencySymbol) {
        this.dollars = dollars;
        this.cents = cents;
        this.currencySymbol = currencySymbol;
    }

    public int compareTo(Money other) {
        return 0;
    }

    public boolean equals(Object other) {
        return false;
    }
```



```
    public String toString() {
        return currencySymbol + " " + dollars + "." + cents;
    }

}
```

8. Save the changes. There should not be any compile errors.

   Now we will write some tests for this code.

9. Right click on the **JUnitTest** project and select **New → JUnit Test Case**.

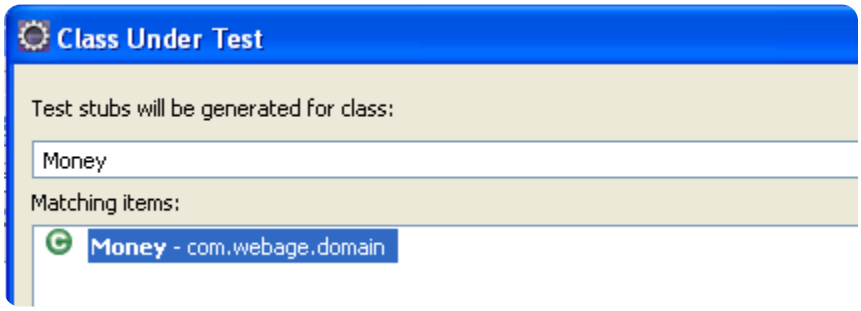
10. Fill in a class name of **MoneyTest** but do not yet create the class.

11. Fill in a package of **com.webage.domain.test**

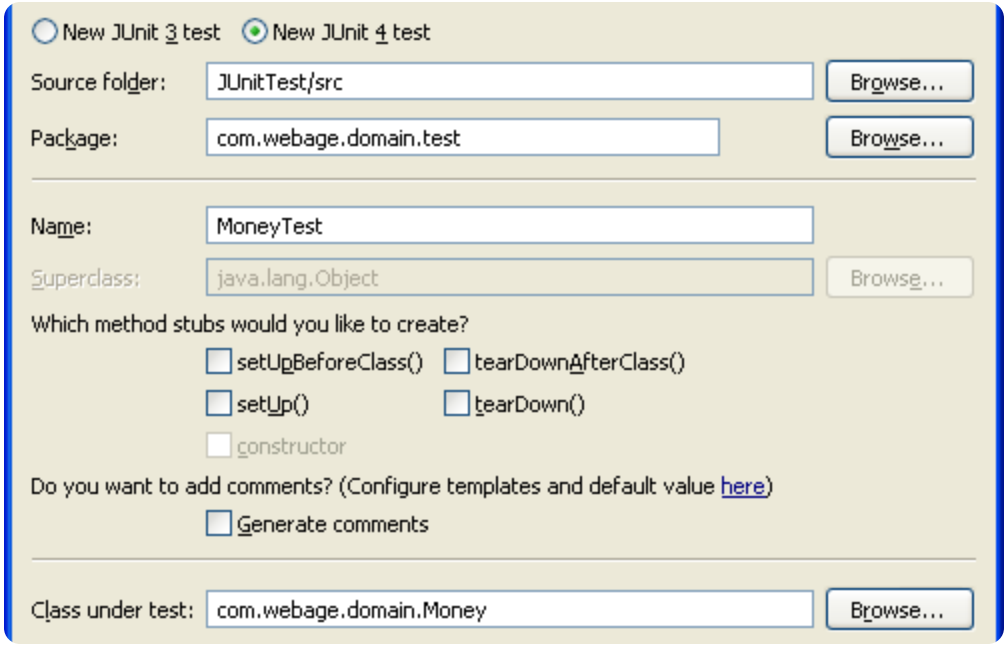
> ⓘ Note
>
> Again the 'test' on the package name and test class is not required but common.

12. Press the **Browse** button at the bottom of the dialog next to the “Class under test”.

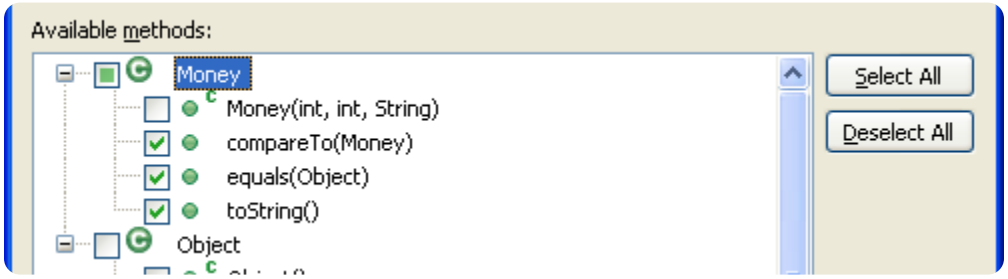
13. Type **Money** and then select the **Money** class that appears and press the **OK** button.



14. Press the **Next** button once the dialog appears as below.



15. Check the box next to the **compareTo, equals** and **toString** methods. This will create tests for these methods.



16. Press the **Finish** button to create the test class.

17. Notice that JUnit 4 test methods are marked with the @Test annotation. The wizard in Eclipse that created the test also put a statement that would guarantee the test to fail until we change it.

```
public class MoneyTest {

    @Test
    public void testCompareTo() {
        fail("Not yet implemented");
    }
```


10.3. Test and Implement toString

The purpose of having unit tests is to verify small pieces of functionality. By focusing on the toString method first we will write tests and modify the implementation if they do not pass.

1. Modify the **testToString** method as follows. You will have an error until the next step.

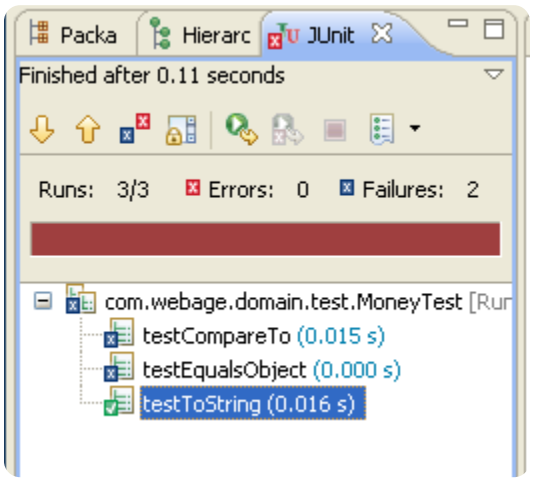
```
@Test
public void testToString() {
    assertEquals("CAD 7.32", new Money(7, 32, "CAD").toString());
}
```

2. From the Eclipse menus select **Source** → **Organize Imports**. This should add an import for the Money class.

3. Save the code and be sure there are no errors.

4. Right click on the **MoneyTest.java** file and select **Run as** → **JUnit Test**.

5. The JUnit view should eventually open on the left. There will be two failures but the testToString test should pass.



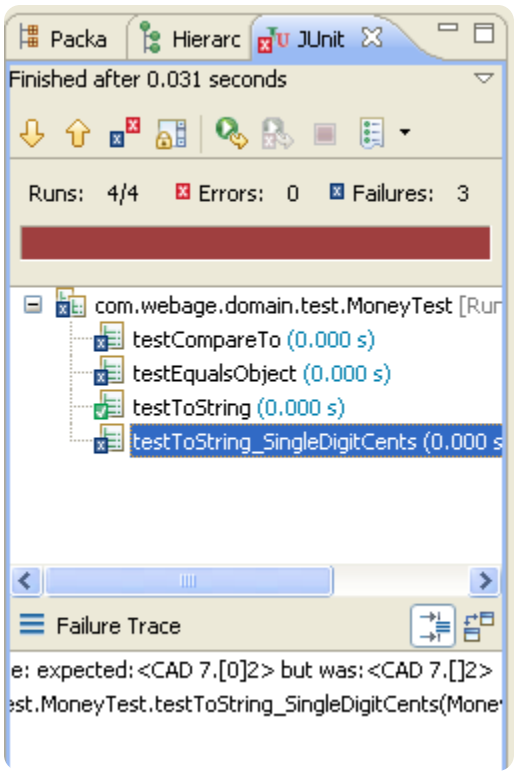
   Although this test passes it is a fairly simple test. We also want to test other conditions. For the Money class a good test will be the case of single digits cents so we will test that now.

6. Back in the **MoneyTest** class add the following new method.

```
@Test
public void testToString_SingleDigitCents() {
    assertEquals("CAD 7.02", new Money(7, 2, "CAD").toString());
}
```

7. Save the code and be sure there are no errors.
8. Right click on the **MoneyTest.java** editor and select **Run as → JUnit Test**.

   The new test (testToString_SingleDigitCents) will fail.
9. Select the testToString_SingleDigitCents method and scroll in the failure trace section. You will see that the zero was not present in one of the things being compared. Obviously this means we need to adjust the implementation of the toString method to handle this condition.



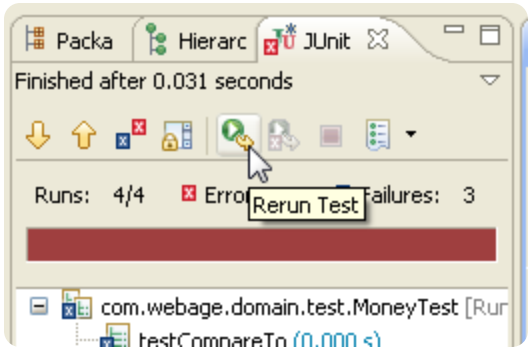
10. Switch to the **Money.java** editor or open the file if you closed it.
11. Modify the implementation of the **toString** method to match the following.

```
public String toString() {
    String centsAsString;
    if (cents < 10) {
```

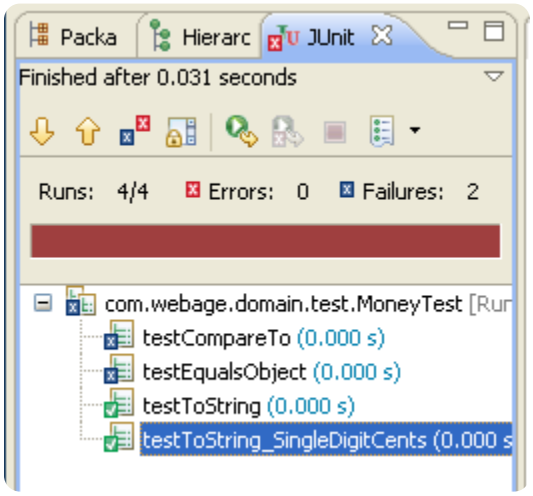
```
            centsAsString = "0" + cents;
        } else {
            centsAsString = String.valueOf(cents);
        }
        return currencySymbol + " " + dollars + "." + centsAsString;
    }
```

12. Save the code and be sure there are no errors.

13. In the JUnit view, click the **Rerun Test** button.



14. This time the test should pass in the **testToString** and **testToString_SingleDigitCents** tests. There are still other failures but the two tests we have implemented have passed. You could add other types of formatting function and define the tests but we will not do that now.



10.4. Implement compareTo and Tests

It will be good to implement compareTo first as we might be able to use this in the equals method.

1. Open **MoneyTest.java**
2. Modify the **testCompareTo** method as follows. You will have an error until the next step.

```
@Test
public void testCompareTo() {
    Money ten = new Money(10, 0, "USD");
    Money hundred = new Money(100, 0, "USD");
    assertTrue(ten.compareTo(hundred) < 0);
}
```

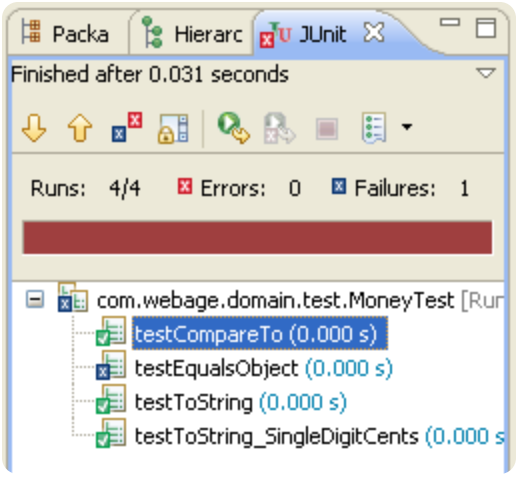
3. Add the following import before the class declaration.

```
import static org.junit.Assert.assertTrue;
```

4. Save the code and be sure there are no errors.
5. Rerun the JUnit tests. The test should fail in the testCompareTo method indicating that our implementation is not correct.
6. Switch to the **Money.java** file and change the implementation of the **compareTo** method.

```
public int compareTo(Money other) {
    int difference = (this.dollars * 100 + this.cents)
                - (other.dollars * 100 + other.cents);
    return difference;
}
```

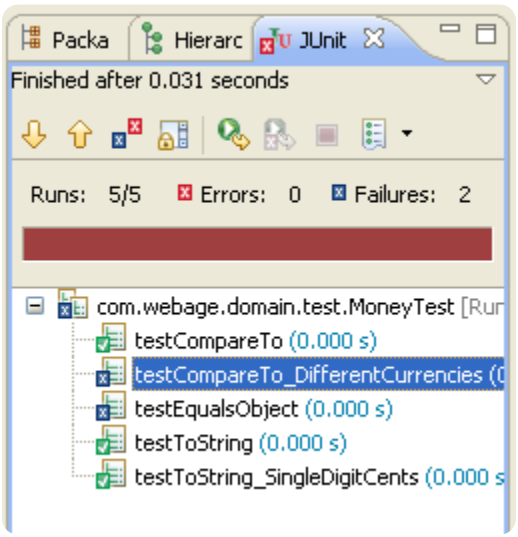
7. Save the code and be sure there are no errors.
8. Rerun the JUnit tests. This time the test should pass in the **testCompareTo** test.



9. Back in the **MoneyTest** class add the following new method. 100 US dollars is less than 100 Euros so this test makes sense for now.

```
@Test
public void testCompareTo_DifferentCurrencies() {
    Money american = new Money(100, 0, "USD");
    Money euros = new Money(100, 0, "EUR");
    assertTrue(american.compareTo(euros) < 0);
}
```

10. Save the changes and run the tests. The test fails in the new method.



At this point, we need to decide what behavior we expect. There are two reasonable possibilities. The first is that we disallow comparing money amounts from different currencies, throwing an exception if someone tries to

do it. The second is that we get into exchange rates and try to convert the money amounts into a common currency so that they can be compared. In a production setting, you would make the decision according to business need, but for this lab, we will simply throw an exception.

11. Change the implementation of the **testCompareTo_DifferentCurrencies** to the following. This uses the 'expected' property of the @Test annotation to indicate that this test expects an IllegalArgumentException to be thrown.

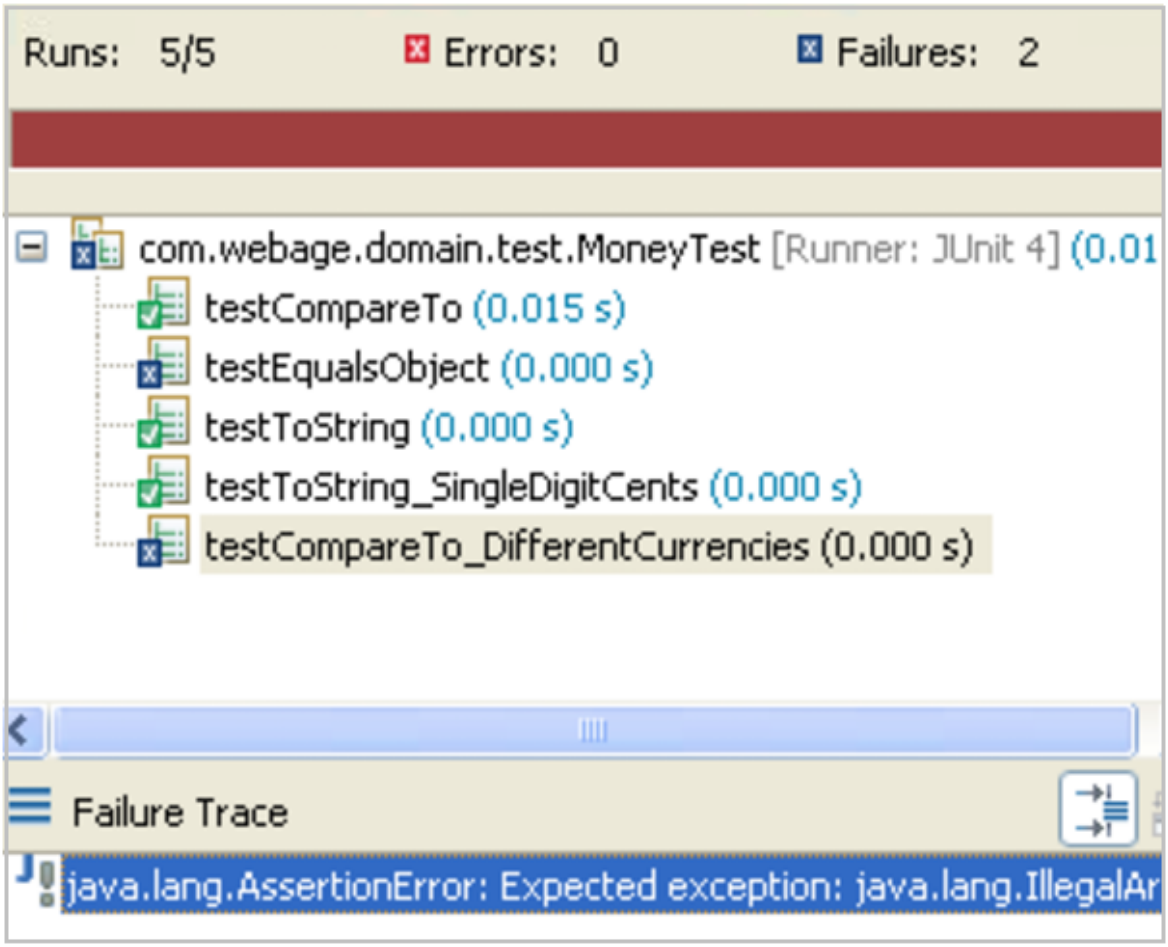
```
@Test(expected = IllegalArgumentException.class)

public void testCompareTo_DifferentCurrencies() {
    Money american = new Money(100, 0, "USD");
    Money euros = new Money(100, 0, "EUR");
    american.compareTo(euros);
}
```

12. Make sure you delete the *assertTrue(american.compareTo(euros) < 0);* line.

13. Save the code and be sure there are no errors.

14. Rerun the tests.

15. Select the **testCompareTo_DifferentCurrencies** method. The test will still fail but this time the failure message indicates the expected exception was not thrown.



16. Switch to the **Money.java** file and change the implementation of the **compareTo** method.

```
public int compareTo(Money other) {
    if (this.currencySymbol.equals(other.currencySymbol) == false) {
        throw new IllegalArgumentException("Cannot compare "
                    + this.currencySymbol + " with "
                    + other.currencySymbol + ".");
    }
    int difference = (this.dollars * 100 + this.cents)
                - (other.dollars * 100 + other.cents);
    return difference;
}
```

17. Save the code and be sure there are no errors.

18. Rerun the JUnit tests. This time the test should pass for the **testCompareTo_DifferentCurrencies** test.

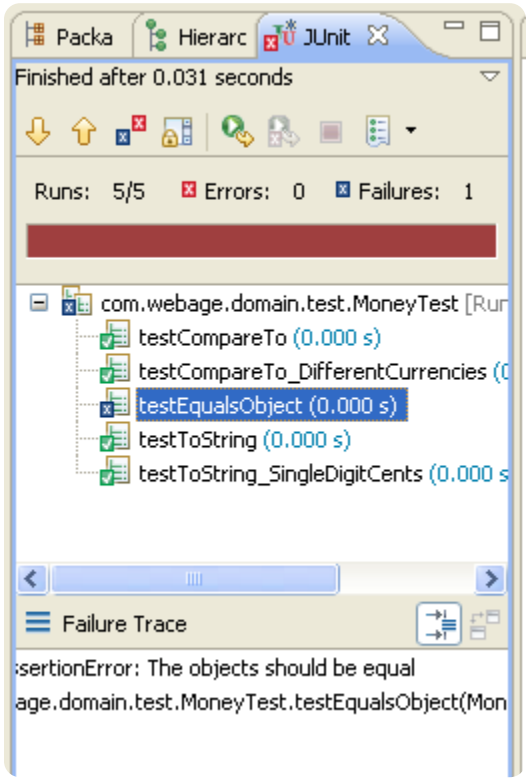
10.5. Implement equals and Tests

We will now move on to the testing and implementation of the equals method.

1. Select **MoneyTest.java**
2. Modify the **testEqualsObject** method as follows.

```
@Test
public void testEqualsObject() {
    Money money = new Money(100, 0, "CAD");
    Money other = new Money(100, 0, "CAD");
    assertTrue("The objects should be equal", money.equals(other));
}
```

3. Save the code and be sure there are no errors.
4. Rerun the JUnit tests. The test should fail and if you select it you can see the custom failure message.



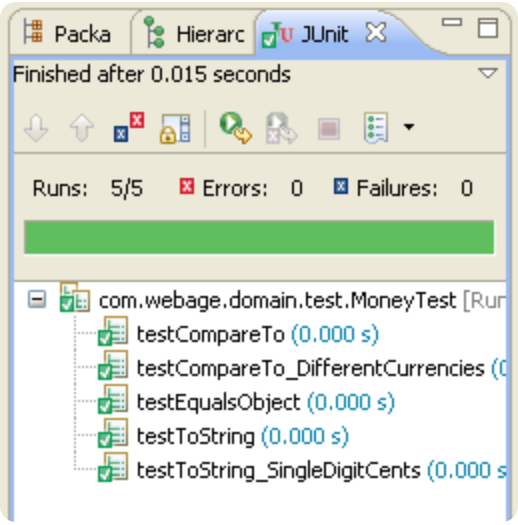
5. Switch to the **Money.java** file and change the implementation of the equals method.


```
public boolean equals(Object other) {
    if (other != null && other instanceof Money) {
        Money that = (Money) other;
        return (this.compareTo(that) == 0);
    } else {
        return false;
    }
}
```

6. Save the code and be sure there are no errors.

7. Rerun the JUnit tests.

8. This time the test should pass in all the methods. Notice when all the tests pass the bar turns green to indicate success.



9. Close all the files.

10.6. Create a Test Suite

Although writing several tests in one class is OK the goal of test classes is to test a single class in the codebase. If we are testing multiple classes it is best to have a "Test Suite" that will combine the tests of several classes into one executable unit.

1. In Eclipse, switch to the **Package Explorer** view on the left which has likely been hidden by the JUnit view.
2. Expand the **JUnitTest** → **src** folders until you see the **com.webage.domain.test** package.
3. Right click this package and select **New** → **Class**.

> ⓘ Note
>
> There is a "JUnit Test Suite" wizard but this is for JUnit 3.x and you will see we will not really need support from the wizard.

1. Fill in a class name of **AllTests** and click the **Finish** button.
2. Add the following code to the declaration of the class. You will have some errors until the next step.

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    MoneyTest.class
})
public class AllTests {

}
```

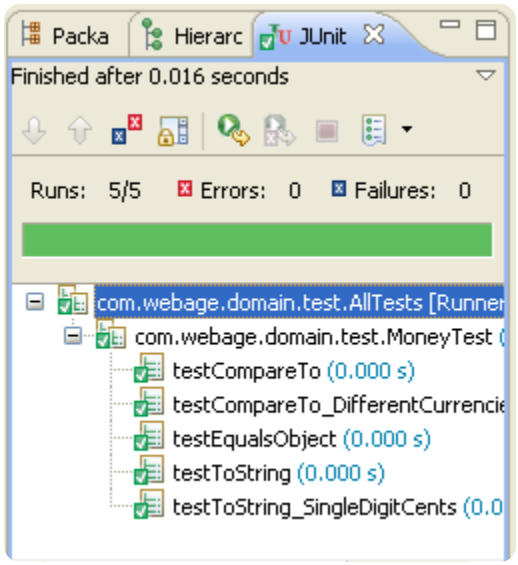
> ⓘ Note
>
> Notice how easy it is to declare a test suite with JUnit 4. If you had multiple classes you wanted to run as part of the suite you would add them as a comma separated list to the @Suite.SuiteClasses annotation.

3. From the Eclipse menus select **Source** → **Organize Imports**. This should add some JUnit imports.

4. Save the code and be sure there are no errors.

5. Right click on the **AllTests.java** file and select **Run as → JUnit Test**.

6. You should get the same test passing results. Expand all in the JUnit view.



   The difference now is that the results of the MoneyTest class are nested within the results of the AllTests test suite. If you had multiple test classes as part of the suite you would have several nested results.

7. Close all open files.

10.7. Review

You have seen how to implement “test driven development”. Before we spend too much time on the implementation of a class we write tests that indicate behavior we wish to see. Although we have not covered all possible conditions to test you have seen the general process.

If the tests covered all the behavior you wanted to test, then at this point you could release the class to the rest of your development team. The team could then look at your tests as examples of how your class behaves. The tests provide a supplement to other kinds of class documentation. At the same time, the tests verify that the

class behaves according to a certain specification, and if someone else changes the class and its behavior no longer passes the tests, you'll know right away by simply executing the tests against the new version of the class. Now there is no need to fear "what if I change this"? You'll know exactly how that change affects the system by seeing whether it causes any tests to fail.