Java Programming for Developers

Course Labs Contents

Introduction to your computer

Software

The following software should be installed on your Windows XP / Vista / 7 machine.

- Java Developers Kit 7 or above
- Text Pad 4.x
    - You may use this if you wish to develop your Java code. It will automatically pick up the compiler in the JDK, and there is an option to compile from the menu.
- Zip Utility
    - If you wish to take the solutions or your labs away with you, you can use this to compress your work.
- Eclipse

Lab Directories

For the exercises, everything you need unless otherwise directed can be found in c:\ConygreJava\labs. The solutions to the exercises can also be found in c:\ConygreJava\solutions. Various demonstrations can be found in c:\ConygreJava\demos.

Chapter 1: Introduction to Java

Aims

In this lab, your aim is to gain familiarity with using the Java Developers Kit (JDK) compiler and runtime environment. We will write a simple Java class that will output some text to the console.

Your First Java Program

1. Using a text editor like notepad, create a new file and save it as **MyFirstClass.java**.
2. At the top of our file we need to firstly declare our class with the line **public class MyFirstClass** *(remember Java is case sensitive so file names and your java code must be correct).*
3. We will then insert curly braces within which to put our class definition. You should therefore have something like this;

```
public class MyFirstClass
{


}
```

4. Within the curly braces we will put a main method. This, if you recall, is where your applications will start from when you invoke the java runtime environment. The signature for the main method looks like this;

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

}
```

   Don't worry if you do not understand why at this stage. You will later.

5. **Printing something out to the screen**
   Now we can inset some code to print something out to the console. Again, you may not understand why at this stage, but the code to do this is as follows, and it goes inside your main method's curly braces.

```
System.out.println("Hello from my first Java program!");
```

6. Your code should now look something like this;

```
public class MyFirstClass
{
    public static void main(String[] args)
    {
        System.out.println("Hello from my first java program!");
     }
```

```
}
```

7. Once you have finished you will need to compile your code, so from the command line whilst in your working directory, you will need to enter the line;

   javac MyFirstClass.java

8. This will invoke the compiler to generate a class file called MyFirstClass.class. If nothing appears to happen, that is good news! It has compiled fine. If you have any errors, check that your code is the same as that written above, and saved in a file called **MyFirstClass.java.**

9. Once it has been compiled, you can run your first java program. To do this, type the following at the command line;

   java MyFirstClass

10. It should print out you're the text that is in your System.out.println("xxx"); statement.

Chapter 2 Working with Eclipse

You will begin by creating a basic Java project using Eclipse. This will familiarize you with the Eclipse Java development environment.

1. From the desktop shortcut launch **Eclipse**.
2. At the **Workspace Launcher** dialog, make a mental note of your default workspace location, then select, "**Use this as the default and do not ask again**". Then click **OK**.

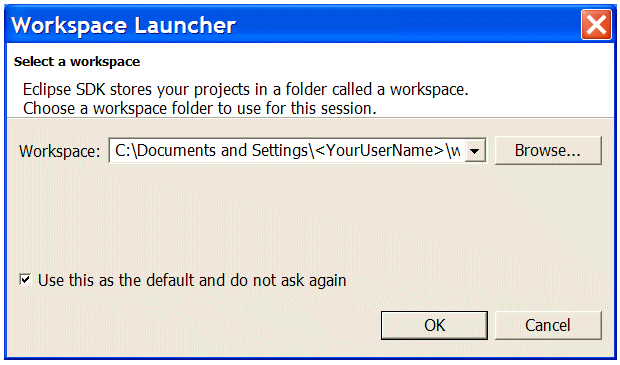


Figure 1 Eclipse Workspace Launcher

3. Close the **Eclipse** welcome screen that appears.
4. Click **File**, then click **New**, and then click **Project**.
5. At the **New** dialog, expand the **Java** folder, and then select **Java Project**.

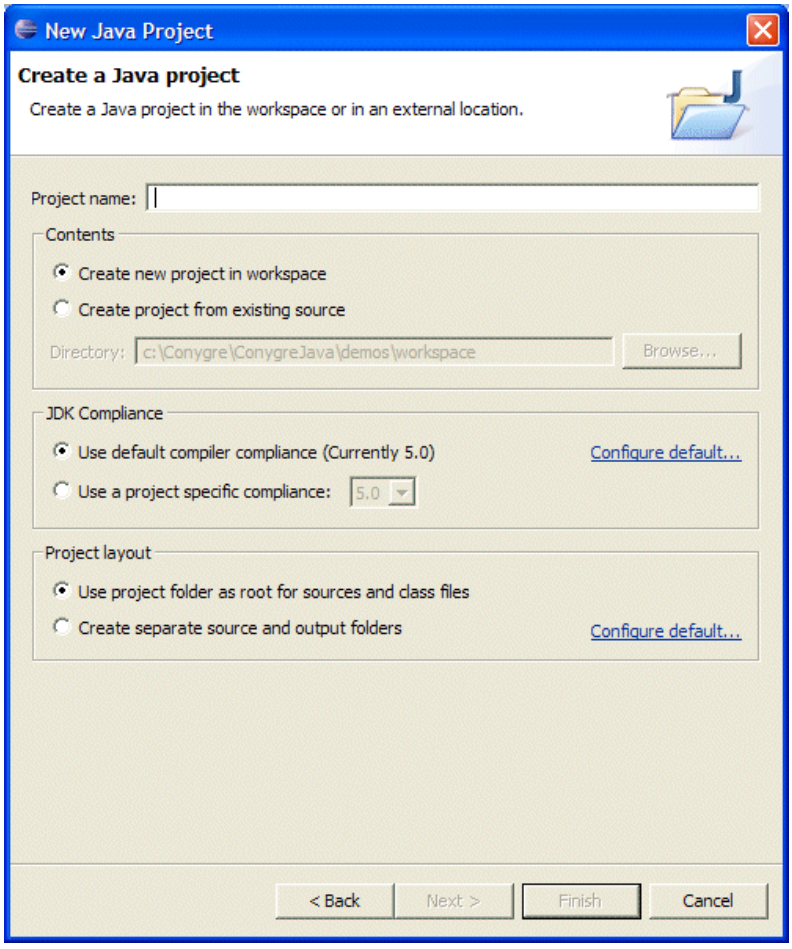


Figure 2 Creating a Java Project using Eclipse

6. At the **New Java Project** dialog, enter the name MyFirstProject, and then click **Finish**.
7. If prompted to switch to the Java perspective, select **Yes**.

You will now create a Java class within this project, and run it from Eclipse.

1. From the **File** menu, point to **New**, and then click **Class**.
2. In the **New Class** dialog, set the **Name** to be MyFirstClass, and in the **"Which methods would you like to create?"** section, select **public static void main(String[] args)**. Then click **Finish**.

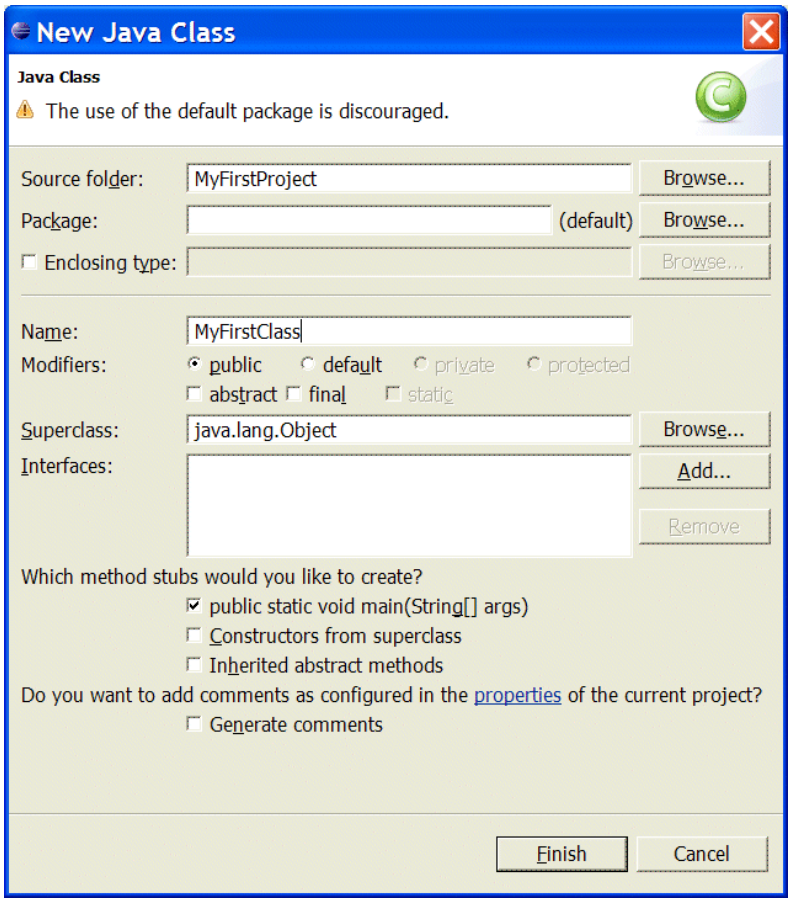


Figure 3 Creating a basic Java Class using Eclipse

3. Within the curly braces of the main method, type System, enter the full stop (period) character, and then watch what happens. A drop down list of all of the method options appears below. You will find this very helpful as we progress through the course.

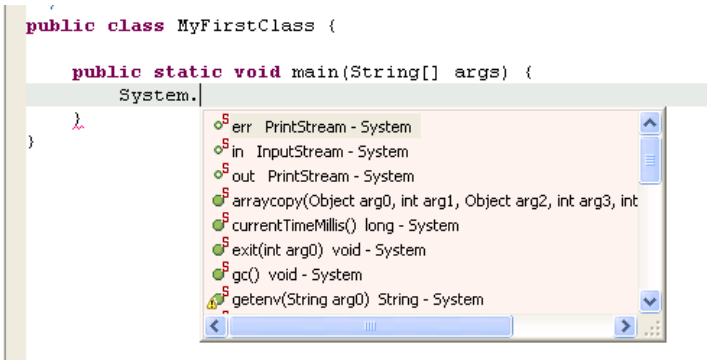


Figure 4 Using Eclipse Intellisense

4. From the intellisense drop down, select **out**, and then add a further full stop (period), and then you will get a further drop down for the methods and properties of out. Either select or type **println**, and then in the brackets add some text in double quotes. Don't forget the semicolon from the end of the line! Your final statement should read:

   System.out.println("Hello from my application");

5. Eclipse will compile your code automatically, so you can now run it. To run the application, from the **Run** menu, click **Run As**, and then click **Run as Application**. If you are prompted to save any resources, click **OK**.
6. The output from the application will appear in the console at the base of the screen.
7. Finally, deliberately introduce an error into the application code, and see what happens when you try to run it. Note that tasks appear in the task list. Try clicking on a task. Note that it takes you straight to the line with the error.

You can now if you wish use Eclipse for the remaining exercises. Each exercise can be done in your existing project unless the exercise recommends a new project.

Chapter 3: Java – The Basics

The Aims

This lab will introduce you to using the primitive variable types, the basic operators, and the concept of casting.

Remember – Java is case sensitive!

Declaring and initialising variables

1. Within the main method of **MyFirstClass** developed in the last lab exercise, you will declare some variables to represent the car you drive, or if you do not have a car – a car you would like to drive! Firstly declare two variables of type *String* called **make** and **model**, then declare a variable of type *double* to be the **engineSize**, and declare a variable of type *byte* to be the **gear** your car is in.

2. Now initialise those variables with appropriate values. Put in some code so that the program will print out the values of these variables, things like;

   ```
   The make is x
   The gear is y
   The engine size is z
   ```

   You will need to use the string concatenator (+).

3. Fix any errors and run the application.

   Your code should look something like this;

```
public class MyFirstClass
{
    public static void main(String[] args)
    {
        String make = "BMW";
        String model = "530D";
        double engineSize = 3.0;
        byte gear = 2;

        System.out.println("The make is " + make);
        System.out.println("The model is " + model);
        System.out.println("The engine size is " + engineSize);
    }
}
```

Casting

4. Now declare a variable of type ***short*** to be the speed. This variable will be the result of the *gear* multiplied by 20 (Not scientific but it will do for now!).

5. Put in a line of code to set the ***speed*** to equal the ***gear*** multiplied by 20. You will need to cast to get it to compile (why?).

6. Print out the speed to the console, and run your program to check that it works correctly.

Optional Extras

7. Modify your code so that you have a number of other variables of different types. Build in some simple arithmetic, printing out your results to the console. You could place in an ***int*** for example to be the revs, which could be based on the ***speed*** variable multiplied by the ***gear***!

Chapter 4: Looping and Branching

The Aims

This lab will introduce you to using the various flow control constructs of the Java language.

Part 1 Using if / else

The main method from your previous lab exercise should look something like this.

```
public static void main(String[] args)
{
    String make = "Renault";
    String model = "Laguna";
    double engineSize = 1.8;
    byte gear = 2;

    System.out.println("The make is " + make);
    System.out.println("The model is " + model);
    System.out.println("The engine size is " + engineSize);
}
```

1. Firstly, put in some logic to print out either that the car is a powerful car or a weak car based on the engine size, for example, if the size is less than or equal to 1.3.
2. Now, using an if / else if construct, display to the user a suitable speed range for each gear, so for example, if the gear is 5, then display the speed should be over 45mph or something. If the gear is 1, then the speed should be less than 10mph etc.

Part 2 Looping

3. We will now need to generate a loop which loops around all the years between 1900 and the year 2000 and print out all the leap years to the command console. You can use either a for or while loop to do this.
4. Once you have done this, set it so that after 5 leap years have been displayed, it breaks out of the loop and prints 'finished'.

Part 3 Using switch / case

5. Now re-write part 1 to use a switch / case construct. It should do exactly the same thing as the if / else if construct. Don't forget to use break!

Part 4 (Optional)

6. This will possibly require some research by you to find out how to do this. Create yourself an array capable of containing 10 ints.
7. Now modify your loop above, so that it no longer displays the first five years, but stores the first 10 in your array instead.
8. Now provide another loop to print out the values in the array. Use the length property of the array object to specify how many times to loop

*** Note that there is no lab for chapter 5 ***

Chapter 6: Introduction to Objects in Java

The Aims

This lab will introduce you to defining classes with instance methods and variables, and then instantiating and manipulating the resulting objects from a main method.

Defining a class

We are going to create a new class called Account, and save it in a file called **Account.java.**

1. In Eclipse, right click on your project src folder, and click **New** and the click **Class**. Set the class name to be **Account**. Do not change other options and click **Finish**.
2. Provide two properties called **balance** and **name**, setting them to be a *double* and a *String* respectively. These variables should be declared as private.
3. Now provide methods which will set and return the balance and the name. Ie. Provide, **get** and **set** methods for your two variables. You can use the Eclipse **Source / Generate Getters and Setters** if you wish.
4. Define a new method called **addInterest**, which does not take in any parameters or return any value, but increases the balance by 10%. We will be using this method later.
5. Check for any errors before proceeding.

Instantiating the class

6. Create another class called **TestAccount** and this time add a main method to it. You can do this at the **New Class** dialog or in the blank class, type the word *main*, and the press **Ctrl/Space**.
7. Within this main method, declare and instantiate a new Account object called **myAccount**, and then call the set methods to give it a name and a balance. Set name to be your name, and you can give yourself as much money as you like!
8. Now print out the name and balance to the screen using the get methods. Place appropriate text before the values using the string concatenator (+).
9. Fix any errors, and run your program. It should print out your name and balance variables.
10. Call the **addInterest** method and print out the balance again. It should be different when you run it.

Chapter 7: Working with Arrays

Aims

You will now take the previous exercise a little further and modify the code to work with an array of accounts rather than single account references.

Creating an array of Accounts

We will now create an array of Account objects called **arrayOfAccounts**. Remember, creating arrays of objects in Java involves three steps, declaring the array, initialising the array, and populating the array.

1. Within the same main method that you used in the previous exercise, declare an array of accounts called **arrayOfAccounts**, and initialise it to be 5 elements in size.
2. Create two other arrays, one of doubles and one of Strings, which contain values for the names and balances of your account objects. Do this using two array initialisers. Make up values for these two arrays. Something like:

```
double[] amounts = {23,5444,2,345,34};
String[] names = {"Picard", "Ryker", "Worf", "Troy", "Data"};
```

3. Using a for loop, populate the object referenced by arrayOfAccounts with account objects specifying a name and a balance using values from your predefined arrays.
4. Within the loop print out the name and balance from each account object.
5. Finally, within the loop, call the **addInterest** on each object in the array. Print out the modified balances.

Chapter 8: More on Java Classes

The Aims

We will now modify our classes to incorporate method overloading, constructors, and static variables and methods.

Adding Constructors

We will now add a constructor to our Account class so that we can create accounts and give them a value for the name and balance variable upon instantiation.

1. In your **Account** class, define a constructor that takes a *String* and a *double*. Assign those values to your instance variables **name** and **balance**.
2. In your **Account** class, define a no argument constructor that passes *your* first name and a balance of 50 to the constructor taking a String and a double. Remember, this is done using the **this** keyword. You will not need this constructor for rest of the exercise, but it is useful to see how it can be done.
3. Now create a new class called **TestAccount2** that has a main method.
4. In the main method, create an array of account objects as you did in the last lab. You need 5 Account objects as before. Set the name and balance when you create each account object. To do this, use the constructor you added that takes in a name and a balance.

Static variables and methods

So far, the interest rate for our account object is somewhat arbitrarily defined by our addInterest method. We will now create a variable, which will be the interest rate for our Account objects. This will be a static variable – why?

5. In the Account class, declare and initialise a static variable of type *double* to be the interest rate for the Account class. Call it **interestRate**. Also, provide a set and get method for the variable - remember that these will be static methods.
6. Modify the addInterest instance method to use this variable instead of our fixed value.
7. From your main method, change the interest rate using the Account class static method **Account.setInterestRate(someDouble)** that you defined earlier in step 5.
8. In the TestAccount2 main method, loop though your array. In the loop you should:
    a. Display the balance
    b. Call the addInterest method
    c. Display the new balance
9. Run your class. Is it using your new interest rate value?

Method Overloading

Finally, we will put in some method overloading. We will provide two methods with the same name that take in different arguments. Our methods will both be called

withdraw. One will take in a parameter called amount, and the other will not require a parameter.

10. In class **Account**, define a **withdraw** method that takes in a value and checks that there is enough money in the account and if there is, the balance is reduced by the amount passed in. The signature will be:

    ```
    public boolean withdraw(double amount)
    ```

    This method will return a boolean, true if there was enough money and false if there was not.

11. Now define a second **withdraw** method. This takes in no parameters, and deducts an arbitary value of 20 from the account. This also should return a boolean, true if there is enough money, and false if not, as before.
    HINT: Is there a way to write this method in a way that does not duplicate the code you created in the previous step.
12. Now in your test class, try calling withdraw on some of your account objects. Make calls to both methods, passing in values that are higher than the balance, and check that it doesn't let you draw the money out (ie. Returns false).
13. You could modify your withdraw methods to also display a message on the screen as well as return true or false, so your users know whether it has been a successful transaction or not.

Chapter 9: Organising Java Classes

The Aims

In this lab you will write a simple class in a package structure. You will then experiment with different compilation and runtime options.

Writing the Class

1. Write a basic class called HelloWorld.java with a main method that simply outputs the String "Hello World" to the command line via a call to System.out.println() when the program is run.
2. Add a package declaration putting this class in com.conygre.simple
3. Open a command prompt in the directory where the Java source file is located.
4. From the command prompt, enter the command javac –d . HelloWorld.java. Note the spaces either side of the dot.
5. From the command prompt, enter the command: dir. Notice that a new directory structure has been created so that your class file is in the proper structure according to the package declaration. The '.' in the javac command simply specifies that the classes should go in the current directory – in their package structure.
6. From the command prompt, change directory to the new folder containing your generated class file. This can be done with cd com\conygre\simple. Try running the class with java HelloWorld. What is the result? It should fail. Why?
7. Now try running it with java com.conygre.simple.HelloWorld Now what is the result? It should fail again. Why?
8. Now change directory to the root on your machine by typing cd \. Run the same command as in the previous step, but this time use the –classpath switch to specify the directory where your package is located. The full entry will be:

```
java –classpath <location-of-package> com.conygre.simple.HelloWorld
```

Where the <location-of-package> is the folder containing the folder structure.

Chapter 10: Working with Strings

The Aims

This lab will introduce you to using the Java string classes.

Part 1 Manipulating Text

1. Create a new Java class with a main method called TestStrings. Each of the following questions can be completed in the main method.
2. Using a String with the value example.doc, change the String to example.bak.
3. Input two Strings and test them see if they are equal. If they are not equal show which of them lexicographically further forward (in the dictionary!).
4. Find the number of times "ow" occurs in "the quick brown fox swallowed down the lazy chicken"
5. Check to see whether a given string is a palindrome (eg "Live not on evil")

Part 2 Formatting Text

1. Print today's date in various formats.

Part 3 Optional Splitting Text

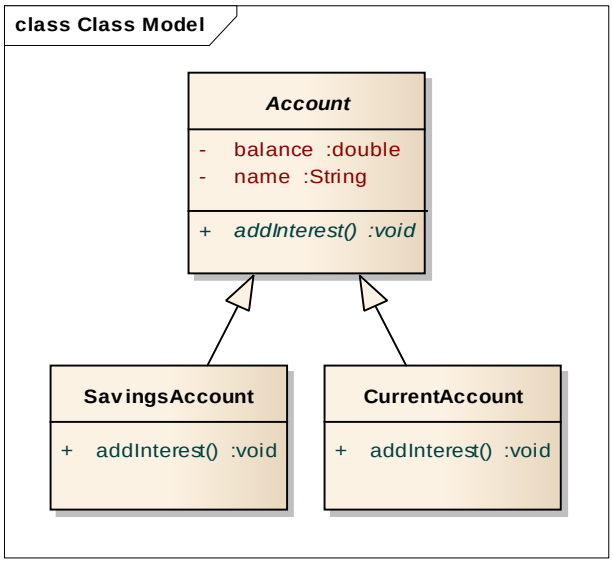
1. Copy a page of news or other text from your favourite website. Split the text into sentences, count and display them. Next split each sentence into words, count and display the words. Finally split the whole text into words and count them, checking that this is consistent with the results of the previous step.

Chapter 11: Inheritance and Abstraction

The Aims

We all know that you cannot simply walk into a bank and open an 'account'. It has to be a particular 'type' of account. We are going to modify our classes so that we have a number of subclasses of the Account class, and we will be instantiating them, and we will exploit polymorphism to manipulate our account objects. We ultimately want to change the addInterest method so that it works differently for each type of account object.

Finally, if you have time, you will make the Account class abstract, so it can no longer be instantiated.



Part 1: Create a new Eclipse Project

1. Close all open files in Eclipse using **File / Close All.**
2. Create a new Java Project called **Inheritance.**
3. Using the **Package Explorer**, expand your first project and select the **Account.java** class and while holding down the control key, drag it into the *src* folder of the new **Inheritance** project.
4. From the Inheritance project, open the Account class.

   Note that all work from now on in this exercise will be completed in this project.

Part 2: Defining the Subclasses

We will firstly define two subclasses of our class called **Account**. One will be **SavingsAccount**, and one will be **CurrentAccount**.

1. Define a new class called **SavingsAccount** that extends Account. You can either add the 'extends Account' text in the editor, or in the **New Class** dialog box.
2. Define a constructor that takes two arguments for name and balance. Why does this need to be here if there is one already defined in the superclass?

3. In the constructor you have defined, pass the two parameters to the superclass constructor using the *super* keyword.
4. Define a second class called **CurrentAccount**, and repeat steps 2 and 3 above.
5. Now empty the contents of the **addInterest** method in the Account class. If you have time you will make it abstract later on. It should now look like this;

```
public void addInterest() {
}
```

6. Now, in your two subclasses, override the addInterest method. Use the @Override annotation.
7. Multiply the balance by 1.1 in the current account and multiply the balance by 1.4 in the savings account. Remember, the balance property is private in the superclass, so can only be accessed directly by the superclass. This means that in your addInterest methods, you will have to use get and set methods to access the balance variable. See below;

```
// add interest method from the subclass SavingsAccount
@Override
public void addInterest() {
        setBalance(getBalance() *  1.1);
}
```

Part 3: Instantiating our classes

1. Define a new class with a main method called *TestInheritance*.
2. Within the main method declare an array called *accounts* of type *Account*, and initialise it with 3 elements. One should be an Account object, one should be a SavingsAccount object, and one should be a CurrentAccount object. You do not need a loop for this bit.

   These objects should have balances of 2, 4, and 6 respectively. The names can be whatever you like.

   How can we get away with placing one of each of these types in the same array? – Polymorphism!

3. Now loop through the array, and call addInterest on each of the elements. Which addInterest will be called? There are three in total. One in each of the subclasses, and one in Account.
4. To prove that polymorphism has worked, within the loop display the names and balances. They should have changed by the appropriate amount.

Part 4: Abstract classes

Currently it is possible to instantiate Account, even though it is an abstract concept like mammal is in the animal world. We will finally make it impossible to instantiate Account, but maintain our inheritance hierarchy. We will make the Account class abstract.

1. Modify the Account class declaration to include the abstract keyword.

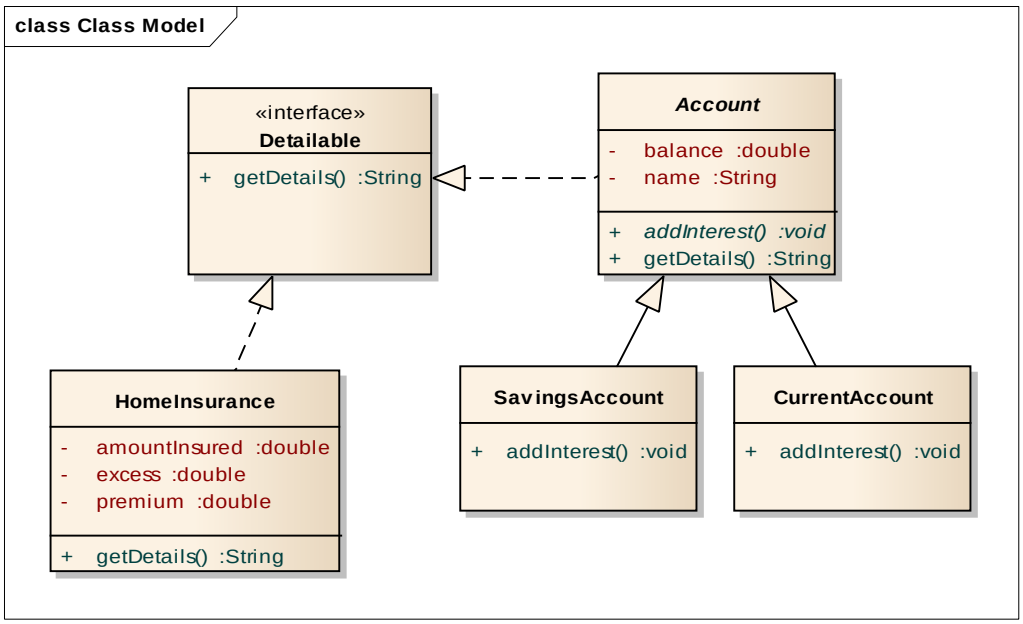
2. Modify the addInterest method and make it abstract. Don't forget the semi-colon.
14. You will now have an error in main because you are trying to instantiate an abstract class called Account.
15. Modify the code inside main so that it now creates two SavingsAccounts and one CurrentAccount.
16. Run your code. It should work as before. You have an array where the array type is abstract, and it contains objects of the various subclass types.

Chapter 12: Interfaces

The Aims

Banks have a variety of different products, not all of which are accounts. We want to provide an interface which forces classes which are products to have a method called getDetails(). This method will display the name and balance for account objects, but for insurance products, it will display price and amount insured information. We will need to define a further class called HomeInsurance, which has three instance variables, premium, excess, and amount insured.

We will then define an interface called Detailable, which contains the method, getDetails(). This will then be implemented in our Account classes and HomeInsurance class.



We will then be able to create an array of Detailable objects, and call getDetails() on each one.

Defining the HomeInsurance class

1. Define a class called **HomeInsurance** which has three properties of appropriate types, called premium, excess, and amount insured.
2. Provide an appropriate constructor for your new class.
3. Save your class.

Define your Interface

4. Right click on your **src** folder and click **New** and then click **Interface**.Specify the name of the interface to be **Detailable**. **Click Finish**.
5. In the interface, declare an abstract method with the signature **String getDetails()**. Remember this is an abstract method.

Implement the Interface

6. Within your class **HomeInsurance**, add the *implements Detailable* to the class declaration. This will induce an error as you have not provided the method implementation for getDetails.
7. Hover over the error and select the quick fix option **Add UnImplemented Methods.** This will then add the getDetails method for you.
8. Update the new method to return the information about the policy. To concatenate the numeric values together as a String, you can place an empty String at the start of your concatenation, something like:

   ```
   return  "" + premium + " " + excess;
   ```

   This initial empty String promotes all the values to a String.

9. Now implement the interface within class Account. Remember, Account is an abstract class, so therefore you now have a choice. You can either implement the interface method in Account, or in the individual subclasses. In our model, it would be best to put the method getDetails() inside Account, because the method content will be the same for all accounts.

Creating an array of Detailables

10. Create a new class with a main method called *TestInterfaces*.
11. In the main method, create an array of type **Detailable** with 3 elements.
12. Add into the array a CurrentAccount, SavingsAccount, and a HomeInsurance object.
13. Loop through the array and call the getDetails() method.
14. Test your code. It will use polymorphism to call the appropriate method.

Chapter 13: Lambda Expressions

Aims

This short exercise will be your first use of the Lambda expressions. You will then revisit using them throughout various other chapters in the course.

Creating a Lambda

1. Within the same project as your interfaces example from before, create a new class called TestLambdas and add into it a main method.
2. In the main method, declare a new Detailable reference and assign it to a lambda expression that returns "hello world".

```
Detailable det = () -> { return "hello world";};
```

3. Now invoke the getDetails method on the lambda and print the response.

```
System.out.println(det.getDetails());
```

Chapter 14: The Java Collections API

Aims

In this exercise, you will revisit the array exercise that you completed in an earlier chapter, and re-implement the array as a List. You will then sort the contents using a comparator object.

Part 1 Re-implement the array from the Inheritance chapter as a TreeSet.

1. In the **Inheritance** project, create a new class with a main method called **CollectionsTest**.
2. Within main, declare a variable that will reference a **HashSet** of Account references called **accounts**.

```
HashSet<Account> accounts;
```

3. Instantiate the **HashSet** using the default constructor.

```
accounts = new HashSet<Account>();
```

4. Create three account objects or subclass objects (if your account class is abstract), and add them to the **HashSet** using the **add** method.
5. Retrieve an **Iterator** for the **HashSet** and iterate over each account, and for each account:
    a. Display the output from the getName() and getBalance() methods
    b. Call addInterest().
    c. Display the output from the getBalance() method.
6. Repeat the previous step, but this time, refactor using the Java5 “for each” construct

```
for (Account a: accounts) {

}.
```

7. Refactor again, but this time use the Java 8 forEach method.

Part 2 Sort the Collection

1. Now define a new class called **AccountComparator**, that implements the **Comparator** interface.
2. Within the **AccountComparator** class, define a **compare** method that compares two account objects, and sorts them into balance order.
3. Re-implement your HashSet as a **TreeSet** passing an instance of your **AccountComparator** class to the **TreeSet** constructor.
4. Re-run the application, and you will see that the accounts are displayed in balance order regardless of the order that they were added in.
5. Now modify the code to use a Lambda based Comparator.

Chapter 15 Working with Enums

Aims

In this exercise you will create an Enum to represent the currency used by your Accounts.

Create a Basic Currency Enum

1. If you are using Eclipse, create a new Java project called **CurrencyEnum**.
2. Create a new Java file called Currency.java (if using Eclipse, use **New**, then click **Enum**.
3. Add three possible values for the Enum to be GBP, EUR, and USD.
4. Save the file and check there are no errors before proceeding.

Add a Currency property to the Account class

1. We have provided a simple Account class for you to use with this, so using **Windows Explorer**, drag the <LAB_HOME>\enums\Account.java file into your CurrencyEnum\src folder in Eclipse.
2. In Eclipse, open Account.java from within the project and add a new private property of type Currency called currency.
3. Add get and set methods to go with it.
4. Add a new constructor that takes in name, balance and now a currency.

Adding a Symbol to the Currency Enum

1. Open the Currency.java file again, and now add the following:
    - A private property called symbol of type char.
    - A private constructor that takes in a char and sets the symbol property to be the char passed in as a parameter to the constructor.
    - A standard getSymbol method that returns the symbol property.
2. Modify the three currencies USD, GBP and EUR to provide a value for the symbol. (to get the Euro symbol, on Windows, press Right-Alt and 4).

Your Currency Enum should now look something like this:

```
public enum Currency {

	USD('$'), GBP('£'), EUR('€');

	private char symbol;

	private Currency(char symbol) {
		this.symbol = symbol;
	}

	public char getSymbol() {
		return symbol;
	}
}
```

Testing the Enum

1. Create a class called **EnumTest** complete with a main method.
2. Within the main method, create an instance of the Account class using the constructor you added in the earlier part providing a name, balance and currency.
3. In a System.out.println() statement, print out the balance and currency of the account.

Chapter 16: Working with Dates and Times

Aims

In this short exercise you will use the Java 8 Date and Time API to solve the following problems:

Using the Date Time classes

Create a simple Java with a main method, and within the method add code that solves the problems below:

1. Write some code to calculate how many days until your birthday.
2. What time is it in NYC at the moment?
3. How many minutes until today's training is over?
4. What week day will your birthday fall on next year?
5. On what date will the next Friday 13th fall?

Chapter 17: Exception Handling

The Aims

In this exercise, you will create an exception type for when a blacklisted name is used to create an account, or have the name of an account set to the blacklisted name. If you want to keep solutions to previous exercises, you must create a new project for this exercise, and copy your **Account** class and its subclasses into a new Project.

Specifying your Account class uses Exceptions

1. Create a new class called **DodgyNameException** that extends **Exception**.
2. Add a default constructor to your exception class that calls the superclass constructor passing the String ***"Fingers is not allowed an account!"***.
3. Modify both the Account class constructors and the Account class **setName** method to show that they throw a **DodgyNameException**.
4. In the body of the constructors and the setName method, check that the name parameter is not "Fingers", and if it is, throw the exception.

Using the Modified Account Class

1. Create a **TestExceptions** class with a **main** method, and copy over one of your blocks of code that creates an array or collection of Accounts.
2. Add the appropriate try / catch blocks.
3. In the catch block, print the exception information using the **toString()** method.
4. Run the application and check that the exception does not get thrown. No output should appear.
5. Change the code so that one of the new accounts is in the name of "Fingers".
6. Run it again and see if the exception gets thrown this time. It should be.

Optional: Working with finally Blocks

1. Within the catch block, add a **return;** statement which will end the main method.
2. Now add a finally block just before the end of the main method.
3. Because of Death duties, to compensate for all the tax avoided by those OffshoreAccounts, collect 40% of the remaining balances of all of the Account objects in the collection or array. Within the finally block, display the amount of tax collected.
4. Run this code with a "Fingers" and without. Does the presence of "Fingers" make any difference as to whether this block runs?

Chapter 18: Introduction to JDBC

Aims

In this lab, you will create a connection to a database. The table in the database will contain information about people. You will retrieve this information and display it. The database used is an Access Database. The driver used will be the Type 1 JDBC-ODBC Bridge Driver.

Preparation: Setting up an ODBC Datasource

You will be using an ODBC data source called 'people' for this application. It will link to an Access database.

1. From Windows, click **Start**, point to **Settings**, and then click **Control Panel**.
2. Launch the **Data Sources (ODBC)** (on XP, this will be in the Administrative Tools section).
3. From the **ODBC Data Source Administrator** dialog**, s**elect the **System DSN** tab.
4. Click **Add**, and select the **Microsoft Access Driver (*.mdb)**.
5. From the **ODBC Microsoft Access Setup** dialog. Set the Data Source Name field to **people**.
6. From the **ODBC Microsoft Access Setup** dialog, click **Select**.
7. From the **Select Database** dialog, browse to **<LAB_HOME>\labs\jdbc\people.mdb**. This is the Access database.
8. Click **OK** to dismiss the Select Database dialog.
9. Click **OK** to dismiss the **ODBC Microsoft Access Setup** dialog.
10. Click **OK** to dismiss the **ODBC Data Source Administrator**.

You have now setup the database to be available with the name **people**, through ODBC.

Part 1 Create a connection

1. Open <LAB_HOME>\labs\jdbc\PersonConnector.java in your text editor or IDE.
2. Import the appropriate package.
3. In the main method and within the try block, load the driver using Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
4. Create a new connection using the driver. Your code will look something like this:

```
Connection conn = DriverManager.getConnection ("jdbc:odbc:people");
```

If there are any changes to this due to your classroom environment, your instructor will inform you.

Part 2 Retrieving the Data

1. Use the connection to create a new Statement
2. Create a String and give it the value "select * from people"
3. Use the executeQuery method on the Statement, passing in your String, to get a ResultSet

4. Now loop through the ResultSet and display the data using System.out.println() statements. The data in the table includes the following:

Column Name	Data Type
Name	String
Age	Int
Gender	String

5. At the end of each iteration, add an empty line.
6. Finally, you need to close your connection and statement objects. Do this from the finally block. Remember that the close method on each throw SQLExceptions, you will need to nest a try / catch block in your finally block.

Part 3 Adding Search Capability

In this section we are going to take an argument from the command line and use it to perform searches. A user will pass a name to search on, and the class will return the relevant details. To use the command line, use the **Run Dialog** described in the **File IO** exercise (see Figure 5).

1. Open <LABS_HOME>\labs\jdbc\DynamicPersonConnector.java.
2. (optional) Add some code to check that an argument has been passed into the main method
3. Declare a String called argument and assign it the value args[0]
4. Underneath the code which establishes a Connection, use the connection to prepare a Statement with the text: "select * from people where name = ?"
5. Use the PreparedStatement's setString() method to set the first parameter to argument
6. Obtain a ResultSet by calling the executeQuery() method on the PreparedStatement.
7. Add the clean up code as you did previously to close the connection and the statement.
8. Compile and run the class and pass in some test parameters. E.g.

```
java DynamicPersonConnector Paul
```

Chapter 19: Multithreading

The Aims

In this exercise you will create applications that work with multiple threads. You will then extend it to take advantage of synchronization.

Part 1 Creating and running a Basic Thread

1. Create a new class called MyRunnable that implements the Runnable interface.
2. Add a run() method, and in the MyRunnable run() method add a loop which should execute 3 times, each time printing a message of your choice.
3. Create another class that has a main method called ThreadTest.
   Within the main() method, create an instance of the MyRunnable class.

   ```
   Runnable run = new MyRunnable();
   ```

4. Within main() create and start 3 instances of the Thread class, passing in the instance of the MyRunnable object to the constructor.

   ```
   Thread thread1 = new Thread(run);
   thread1.start();
   ```

5. Run the program and check that the correct number of messages are printed, 3 from each thread instance.
6. Now modify the program to use a Lambda based runnable instead of the MyRunnable class.

Part 2 Working with Synchronization

1. In the MyRunnable class, add a private String variable containing your message.
2. In the MyRunnable class, add a slowMessage() method that displays the contents of the message one character at a time with a 10 ms pause between each character (use Thread.sleep(10)) for the pause.
3. In the MyRunnable class, from the run() method, call the slowMessage() method within the loop instead of simply printing the message directly.
4. Run the ThreadTest class now and see what happens.
   If the message was:
   "hello from the thread"

   You will get output something like this:

   ```
   hhheeellllolo of  rffrormoom mt  thtehh eet  httrhherraeedaa
   hdde
   h
   hleellloll oof  rfforrmoo mmt  httehh eet  ththhrrereeaadadd
   h
   he
   hleellllooo  frffrrooommm  t httehh ee t tthhrherreeaadadd
   ```

   Add the synchronized keyword to the slowMessage method and run ThreadTest again. What happens this time and why? If you have done this correctly you will see perfect output:

   ```
   hello from the thread
   ```

```
hello from the thread
hello from the thread
hello from the thread
hello from the thread
hello from the thread
hello from the thread
hello from the thread
hello from the thread
```

5. Refactor (change) the code so that are now using a synchronized(this) block within the method. Run it and check that it still behaves correctly.
6. Finally, change the synchronized block so that you are synchronizing on the String message variable. Run it again and check that it behaves correctly.

Appendix A: Java Networking with Sockets

Aims

In this exercise, you will create a server and client application that will exchange information across a network. If you wish, you may work in pairs for this exercise, with one of you creating the client application, and one of you creating the server application.

Part 1 Implementing the Server

To begin with, you will implement the server application. This application will read in lines of text from the client, and write it back to the client in upper case characters, so you a client can see that the server application has processed the data.

Set up the required properties

1. Create a new Java source file with a main method, and call the class MyServer and the file called MyServer.java.
2. Within the main method declare an int called serverPort, and assign it the value 8081.
3. Declare an int called clientPort that is not assigned.
4. Declare a ServerSocket called serverSocket and set it to null.
5. Declare a Socket called clientSocket and set it to null.
6. Declare an InputStreamReader (isr), a BufferedReader (br) , and a PrintWriter (out), and assign them all to null.

Create the ServerSocket

1. Now add a try block, and within the try block, complete the following steps.
2. Add a statement to print to the screen that you are waiting for a client application to connect.
3. Instantiate your serverSocket using the appropriate port number as defined earlier. The code will block at this point until a client connects.

Process a client request

1. Accept the connection from the client, and assign the return value to your Socket reference.
2. Obtain the client port number and display the value on the screen.
3. Use the InputStreamReader to decorate the input stream from the client.
4. Use the buffered stream reader to decorate the InputStreamReader.
5. Use the PrintWriter to decorate the client output stream.
6. Now read in the lines from the client and then right them on the screen, and then write an upper case version back to the client.
7. Catch and handle the IOException.
8. Add a finally block to close the streams, the server socket, and the socket.
9. Compile your code and fix any errors before proceeding.

Part 2 Implementing the Client

You will now implement the client application that will connect to the server application.

Define the required variables and import the packages

1. Define a new class called Client in a new source file called Client.java.
2. Add import statements for the java.net and the java.io packages.
3. Add a main method.
4. Define a String called host which is either set to localhost, or the IP address of the machine of the person you are working with.
5. Define an int called port, and set it to the port number 8081 used by the server application.
6. Declare a variable of type Socket called clientSocket. This will be your socket to the server.
7. Declare a buffered reader called keyboard and assign it to null.
8. Declare an InputStreamReader (isr), a BufferedReader (br) , and a PrintWriter (out), and assign them all to null. Ensure that the PrintWriter is set to autoflush using the boolean constructor parameter.

Create a connection to the server

1. Add a try block and instantiate your clientSocket object, passing in your server socket string and the port number to the constructor.
2. Get the OutputStream from the clientSocket, and decorate it with a new PrintWriter, assigning the return to the variable you declared earlier called out.
3. Decorate System.in using the InputStreamReader, assigning the return to your isr variable.
4. Decorate the InputStreamReader using the BufferedInputStream, assigning the return to your br variable.
5. Read information from the keyboard and send it to the server until the user types ‘Quit’. You can do this using the following line of code:

   ```
   while (!(line = keyboard.readLine()).equals("quit"))
   ```

6. Print out the response from the server each time you send a line to the server to the console using System.out. You can do this by writing out the content from the BufferedReader using br.readLine().
7. Catch the IOException, and add a finally block to close the socket and the streams.

Part 3 Optional Make your server class multithreaded

Currently, your server application is single threaded and can only process one client. Try extending your application so that it can handle multiple client requests. You will need a new Thread on the server for each client. The client application will not change.

Appendix B Inner Classes

The Aims

In the last exercise you created a TreeSet that worked in conjunction with a Comparator to order the items in the collection. You will modify this application to use a nested class, a local class, and an anonymous class.

Part 1 Creating a Nested Class

1. Modify your previous exercise so that the **AccountComparator** class is a nested class within the **CollectionsTest** class.
2. Ensure that the class definition is private and static.
3. Run your application again to ensure that it continues to work correctly.

Part 2 Creating a Local Class

4. Modify the application again so that the **AccountComparator** is now local to the main method.
5. Insert a new line at the beginning of main to declare a simple int with a value of 5.
6. Try accessing the variable from within your compare method in some way by perhaps trying to print it to the screen. What happens and why?
7. Change the variable declaration so that it can be accessed from the local class.
8. Run the application again to ensure that it continues to work correctly.

Part 3 Creating an Anonymous Class

9. Modify the application again, so that the **AccountComparator** is declared as an anonymous inner class.
10. Run the application again to ensure that it continues to run correctly.
11. Finally, using **Windows Explorer**, take a look in the Project folder for the project containing your exercise files. What do you notice about the created class files?

Appendix C: File IO

The Aims

In this chapter you will process a file path and check that it is a directory, and if it is, list the contents of a directory. You will then create a file copying program that will read a file and copy the contents in a different location.

Part 1 Using the java.io.File class

1. Create a new Java class with a main method called *DirList.java*.
2. Create a File object based upon a folder on your computer, such as c:\Program Files.
3. Using the File class methods, check that the directory exists, and print out a message to say if it does or not.
4. Now check that the path is a directory and not just a simple file. If it is not a directory, display a message and exit.
5. Finally, list the contents of the directory on the console.

Part 2 Using the Stream Classes

What we will do in this part of the practical is implement a file copying program, using readers and writers. We will assume a similar scenario to the previous part of the practical, except this time we will be expecting *two* parameters on the command line, and this time they will both be files. Instructions for how to do this using Eclipse are found at the end of the exercise. If you are using a different tool or are unsure how to do it from the command line, then ask your instructor for help.

1. Create a new Java class with a main method called *FileCopier.java*.
2. Create a **File** object to refer to the first file name passed in on the command line.

   File in = new File(args[0];

3. Create a second **File** object to reference the second command line argument which will be the file that you are going to write.
4. Check that the file you are going to read exists (use the *exists()* method). You cannot read a file that isn't there! If it does not exist, exit the application with a suitable message.
5. Check that the second argument file does not exist, and if it does, warn the user that they will have overwritten the file that was there.
6. To read in the file, use a **FileReader** decorated with a **BufferedReader** and read in the file line by line.
7. As you process each line, use a **FileWriter** decorated with a **BufferedWriter** to write the output to the target file.
8. You can use the *newline()* method of the BufferedWriter to put new lines between each write.
9. Flush and close the output streams.
10. Your code should look something like this:

```
File in = new File(args[0]);
File out = new File(args[1]);

        try {
                FileReader reader = new FileReader(in);
                BufferedReader bReader = new BufferedReader(reader);

                FileWriter writer  = new FileWriter(out);
                BufferedWriter bWriter = new BufferedWriter(writer);

                while (true)
                {
                        String line = bReader.readLine();
                        if (line == null) break;
                        bWriter.write(line);
                        bWriter.newLine();

                }
                bWriter.flush();
                bWriter.close();

        } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
```

Part 3 Running the Program

Eclipse users

To pass a parameter at the command line when the program runs.

1. Click **Run**, and then click **Run Configurations** (see Figure 5).
2. At the **Create, manage and run configurations** dialog, check the the project refers to your project and the **Main Class** refers to your **FileCopier** program.
3. Click the **Arguments** tab. In the **Program Arguments** text area, enter your file names as arguments separated by a space.

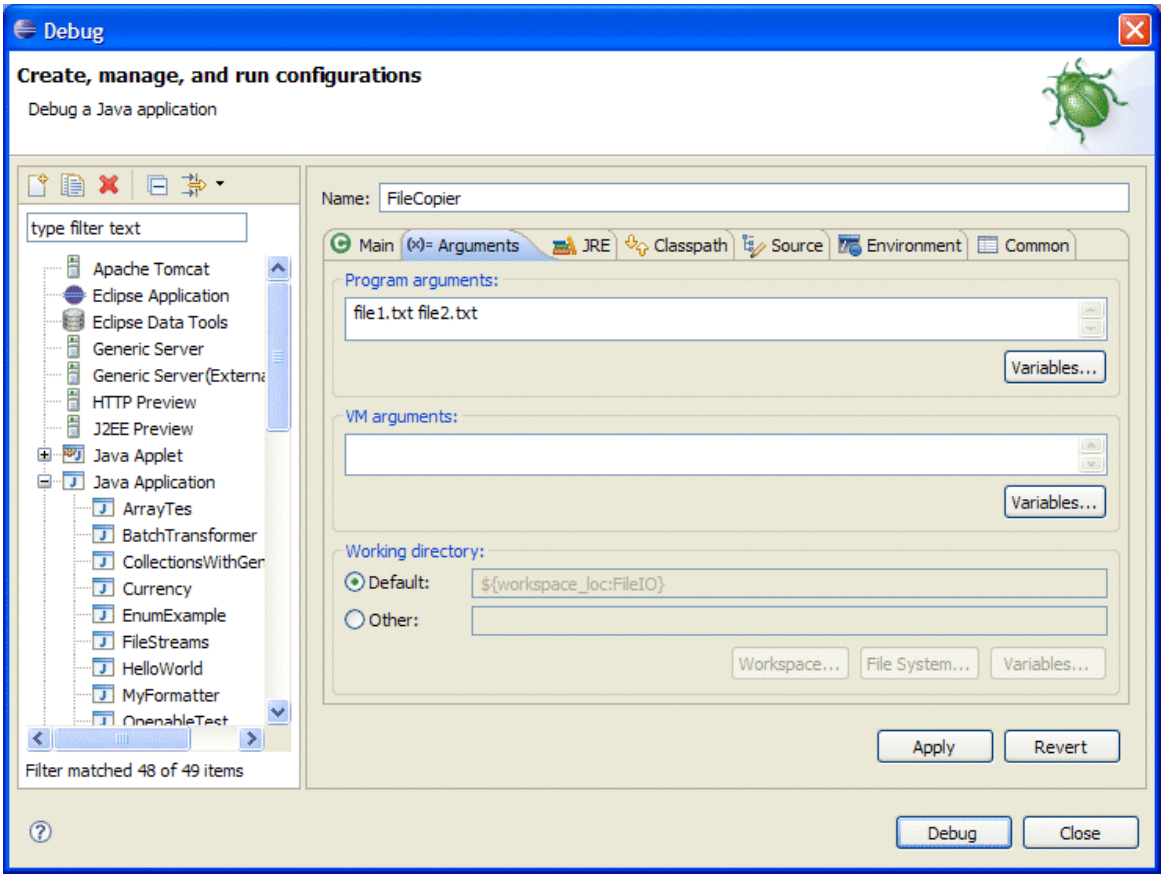


Figure 5 Passing Arguments to a Java Program from Eclipse

Optional Part

The practical you have just completed simply assumed that if the output file already exists, then it will write to a default location – even if that default location already exists!

Expand your previous work so that if the output file already exists, the program prompts the user to see if they wish to overwrite the existing file, or specify a different target instead. To do this, you will need to process System.in.