Java 11 Programming for Object Oriented (OO) Experienced Developers

TT2100H: Student Workbook

Lab Exercises

Exercises/Tutorials...4

Demonstration: Exploring MemoryViewer 5

Tutorial: Setup Projects in Eclipse 6

Exercise 1. Create a Simple Class...12

Exercise 2. Create a Class with Methods 16

Exercise 3. Looping 18

Exercise 4. Language Statements 19

Exercise 5. Fun With Strings 21

Exercise 6. Using StringBuffers and StringBuilders...23

Exercise 7. Creating Subclasses 25

Exercise 8. Field Test 27

Exercise 9. Creating an Array...29

Exercise 10. Using Local-Variable Type Inference...**30**

Exercise 11. Salaries - Polymorphism 33

Exercise 12. Mailable - Interfaces...35

Exercise 13. Exceptions 36

Exercise 14. Exceptional 38

Exercise 15. Using Primitive Wrappers 41

Exercise 16. Enumerations...42

Exercise 17. Agenda **43**

EXERCISES/TUTORIALS

TT2100H - Java 11 Programming for Object Oriented (OO) Experienced Developers

Version 20200403

DEMONSTRATION: EXPLORING MEMORYVIEWER

Overview	
In this exercise you will use a small Java application that monitors the amount of memory that its JVM is currently using. The application provides a user interface that allows you to request a garbage collection from the JVM.	
Objective	Understand how to run a Java application, what a basic Java GUI looks like, and experience the JVM's memory management.
Builds on Previous Labs	None
Time to Complete	15 minutes

Task List

Note: Some of the labs in this course are demonstrations that are already fully implemented. These labs are found as subfolders under **~/StudentWork/demos**

First, navigate to the **~/StudentWork/demos/MemoryViewer** folder and note that there are several files. Identify which is a source file, bytecode files, JAR file, and batch file.

1. When you double-click on the batch file a small window is created that shows memory usage and a garbage collection button. Watch the numbers for a minute. Several things are going on here.

NOTE: Windows does allow for jar files to be executed by doubling-clicking the jar file within the explorer. The batch-file is added to simplify execution of the jar file from the command prompt.

2. This Java application runs inside a JVM. That JVM controls access to memory and has a limited amount of memory initially available to it. The default amount is roughly 2MB (although this can be changed at startup of the JVM). Hence the upper limit on available memory. If an application uses or needs more memory, the JVM will increment the memory usage to accommodate the requirements. The upper limit for this memory usage can be established ahead of time or, if not explicitly defined, will be whatever the operating system will give the JVM. Again, this is done is increments.
3. In an object-oriented runtime environment, one of the issues you have to deal with is reclaiming memory associated with data structures that you no longer need. In Java, this process is called garbage collection. Periodically, a JVM will decide on its own to garbage collect. You can also invoke the garbage collection programmatically. If you watch the memory for a bit, you will see the JVM invoke garbage collection based on either time or on the available memory getting small. Try invoking it on your own with the garbage collection button.
4. Finally, a dynamic that you will notice is the amount of memory used will continue to rise and fall even when there is no apparent activity. This is because an update process is occurring every second that checks available memory and updates the display. Each of these updates generates some garbage that will eventually need to be collected.
5. While we have provided the source code for this small application, we do not expect you to understand it. The code is rather complex, using some of the more advanced aspects of Java to support communication with the JVM, the user interface, and handling user-generated events (pushing the buttons on the user interface).

TUTORIAL: SETUP PROJECTS IN ECLIPSE

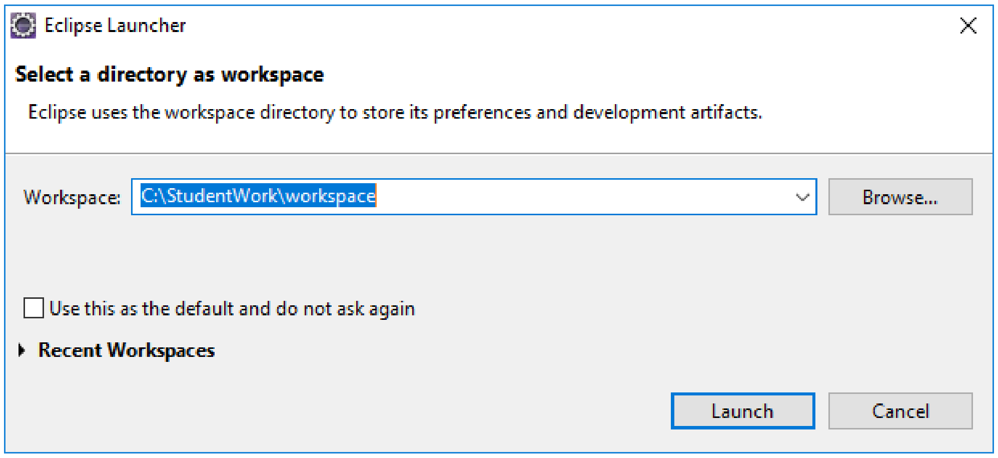
Overview	
This tutorial illustrates how to setup the Eclipse environment for use in implementing the exercises. During this training, you will be developing several miniature-projects using **Eclipse**. For each exercise, we have created skeleton code, which you will be editing. In some cases, we also provide additional files, which already have been implemented for you. (E.g. Helper classes and some Welcome pages for the web applications). Before each exercise, all of these files (skeleton code, helper classes, etc…) will have to be imported into Eclipse. This tutorial describes how to import these files into Eclipse.	
Objective	• Become familiar with the Eclipse Environment
Time to Complete	20 minutes

The Workshop directory

For the location of the Workshop directory, ask your trainer. During this tutorial, we will presume that the Workshop directory has been placed in **C:\StudentWork**

Starting Eclipse

1. From the Windows Desktop, double-click the Eclipse icon. If you do not have a shortcut icon on your desktop then add one that references the Eclipse executable.
   A popup will appear in which you can specify the location of the workspace.
2. Change the workspace location to: **~StudentWork\workspace**


Eclipse Launcher

Select a directory as workspace

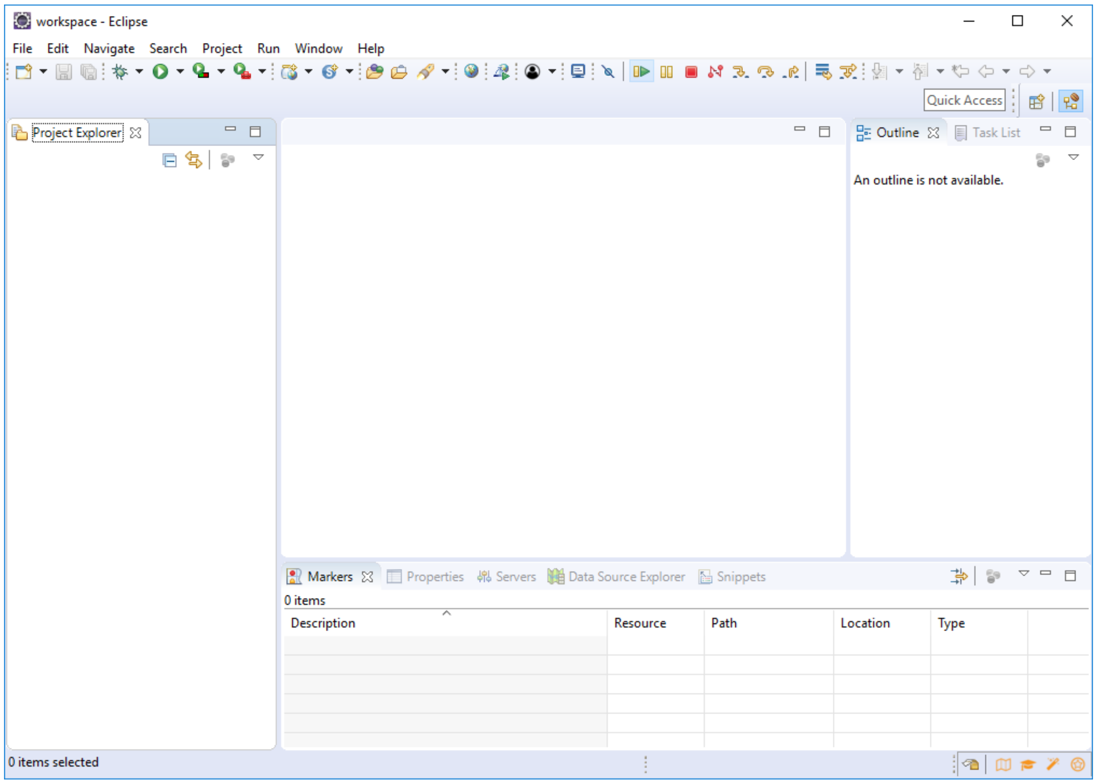
Eclipse uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\StudentWork\workspace

An empty workspace will now be created in the **~StudentWork\workspace** directory.

When the application is started, the Welcome page will be presented.

Select the **Workbench** icon Workbench on the right side of the Welcome window. This will take you to the workbench:

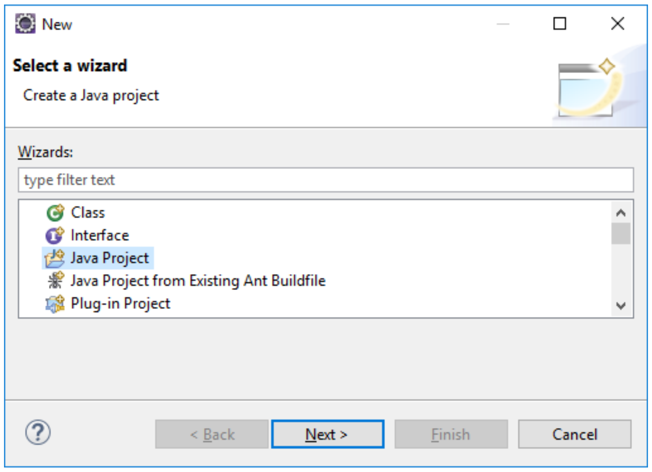


Creating new Java Projects and Classes

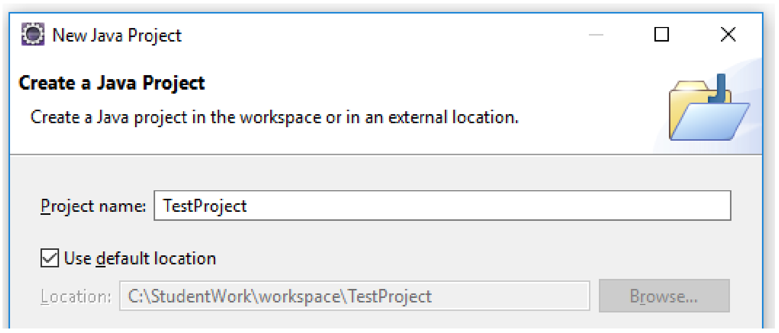
There are numerous ways to create new resources (projects, folders and files) in Eclipse. The three most common are:

- From the main menu: **File->New->Project**
- Right-click in the Navigator or Package Explorer view and select **New->Project...**
- Press **Ctrl+N**

3. In all three cases the **New** dialog will open giving you a selection of resources to choose from. Press **Ctrl+N** then select **Java Project**



4. Click **Next**. In the Project Name field of the Create a Java Project page enter the name **TestProject**.


New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

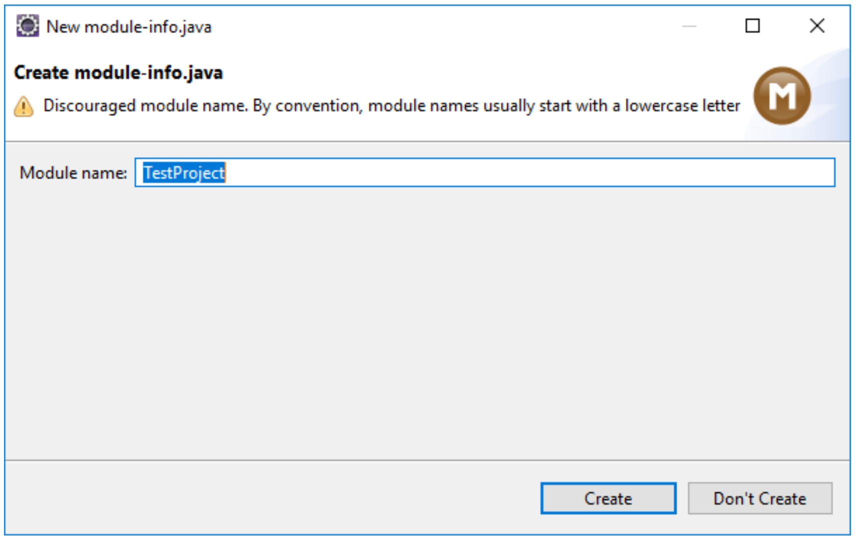
Project name: TestProject

☑ Use default location

Location: C:\StudentWork\workspace\TestProject

5. Insure that the **JRE** is pointing the correct version for use in this course
6. Press **Finish**.

Java 9 introduced the Java Modular system. When you are creating a new project using JDK version 9 or newer, Eclipse will ask if it should create a module-info.java module descriptor for the new project. Unless the exercise description explicitly tells you to have the IDE create a module descriptor you can click **Don't Create**. When a module descriptor is needed for the application you are about to create you will either create one during the exercise or it will be part of the code that is imported into the project at the start of the exercise.

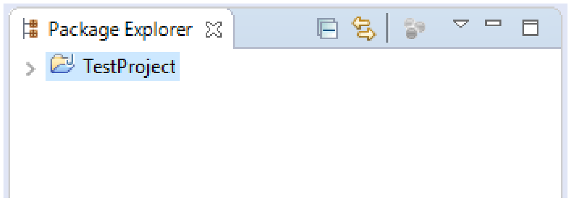

New module-info.java

Create module-info.java

Discouraged module name. By convention, module names usually start with a lowercase letter

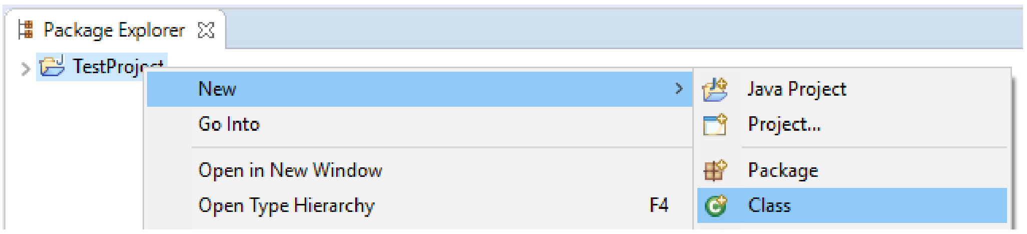
Module name: TestProject

7. Select **Don't Create** to skip the creation of the module descriptor
8. If the Open **Associated Perspective** dialog opens asking for permission to switch to the **Java perspective** click **Yes**. The new project will appear in the **Package Explorer** view.



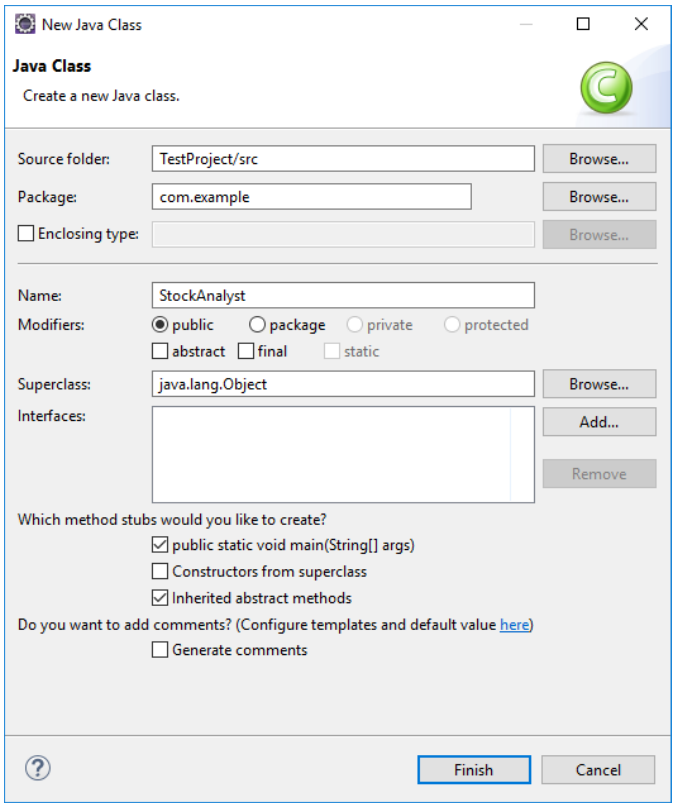
To create a class you can define the class with a non-existent package and Eclipse will create it when you click **Finish**.

9. **Right-Click** on the **TestProject** project and select **New -> Class** to begin the process of creating a new class.



The New Java Class dialog will open.

10. On the **Java Class** page set the **Package** to **com.example** and in the **Name** field enter **StockAnalyst**.
11. Select the option to create a **public static void main**.
12. Click **Finish**.


New Java Class

Java Class

Create a new Java class.

Source folder: TestProject/src

Package: com.example

☐ Enclosing type:

Name: StockAnalyst

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

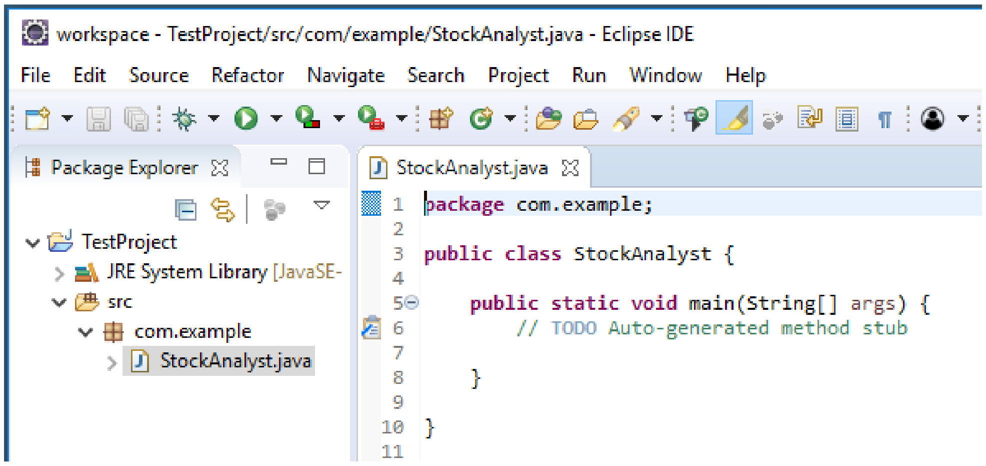
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

In addition to creating the package and the class Eclipse also opens the **Java editor** with the newly created class.


workspace - TestProject/src/com/example/StockAnalyst.java - Eclipse IDE

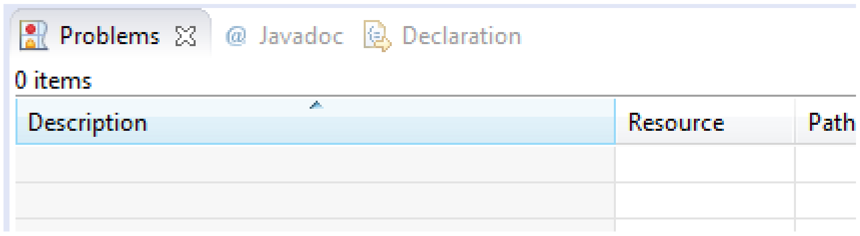
```
package com.example;

public class StockAnalyst {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
```

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
StockAnalyst.java
TestProject
JRE System Library [JavaSE-
src
com.example
StockAnalyst.java

13. Let's insert some functionality and run the new class. Replace the Java comment: **//TODO Auto-generated method stub with**:

```
System.out.println("Buy....buy!");
```

14. Save the class file; this will cause it to be compiled. There should not be any compilation errors listed in the **Problems** view.



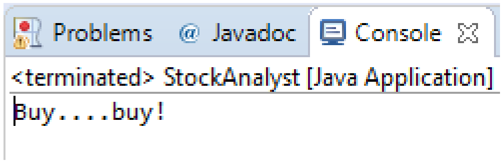
Problems | Javadoc | Declaration

0 items

Description	Resource	Path

15. Right-click on the StockAnalyst.java entry and select **Run As -> Java** Application

16. This will invoke a JVM, which will examine the class definition for an entry point (the main method) and then run that method, resulting in a message being sent to the Console:



Problems | Javadoc | Console

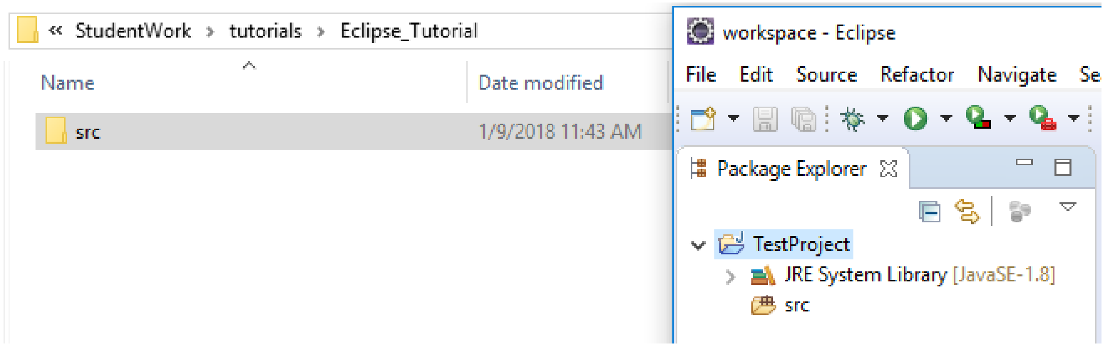
```
<terminated> StockAnalyst [Java Application]
Buy....buy!
```

Importing Lab content

The easiest way to import the exercise code into Eclipse is by using the Windows Explorer and drag and drop the content into an existing Eclipse project.

1. Using the Windows explorer, browse to the directory where the Java exercise resides (for example, **~StudentWork\tutorials\Eclipse_Tutorial**, though this path will vary based on your exercise)

2. Select all files and subfolders and drag the content to the Project in which you would like to import the code. (Make sure you drop the content at project level!)



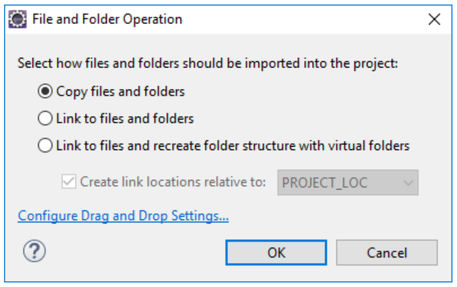
« StudentWork › tutorials › Eclipse_Tutorial

Name	Date modified
src	1/9/2018 11:43 AM

workspace - Eclipse

TestProject
JRE System Library [JavaSE-1.8]
src

3. A **File and Folder Operation** pop-up will be displayed. Make sure the option Copy files and folders is selected and click **OK**



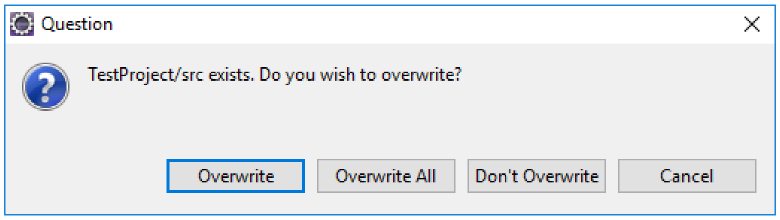
File and Folder Operation

Select how files and folders should be imported into the project:

[✓ Copy files and folders]
Link to files and folders
Link to files and recreate folder structure with virtual folders

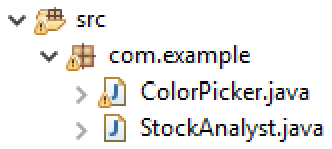
[✓ Create link locations relative to:] PROJECT_LOC

4. Eclipse will warn you to indicate that the project already has a src folder. Click **Overwrite**



TestProject/src exists. Do you wish to overwrite?

After import you should see that the ColorPicker class has been added to TestProject:



17. Run the ColorPicker class as a Java Application by **Right-Clicking** ColorPicker.java and selecting **Run as -> Java Application**.

Exercise 1. CREATE A SIMPLE CLASS

Overview	
In this exercise, you will create a very simple class with two attributes and no methods.	
Objective	Understand the basics of a class file
Exercise Folder	SimpleClass
Time to Complete	30 minutes

Task List

First: If you have not done so already, please complete the tutorials for your particular IDE.

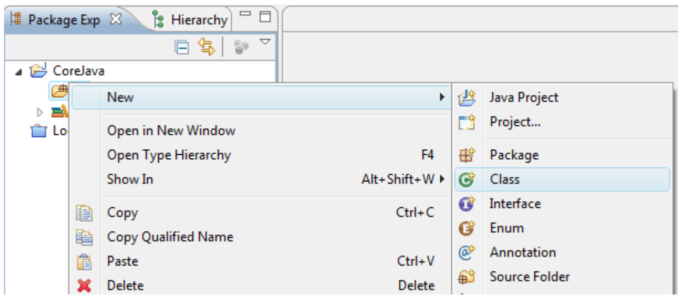
Note: Most of the labs in this course begin with class files that are skeleton code you can use as a starting point for a lab assignment. This skeleton code provides hints on what needs to be done and where it can be done. These starting points for the labs are all found as subfolders under **~/StudentWork/code/<exercise-folder>/lab-code**, organized by the lab they are associated with.

1. Create a new Java Project or module called **CoreJava**.

If you are not familiar with how to do so, please refer to the tutorial at the end of the lab manual for instructions on how to perform this and other operations with your IDE. You will be using this project for the next few exercises.

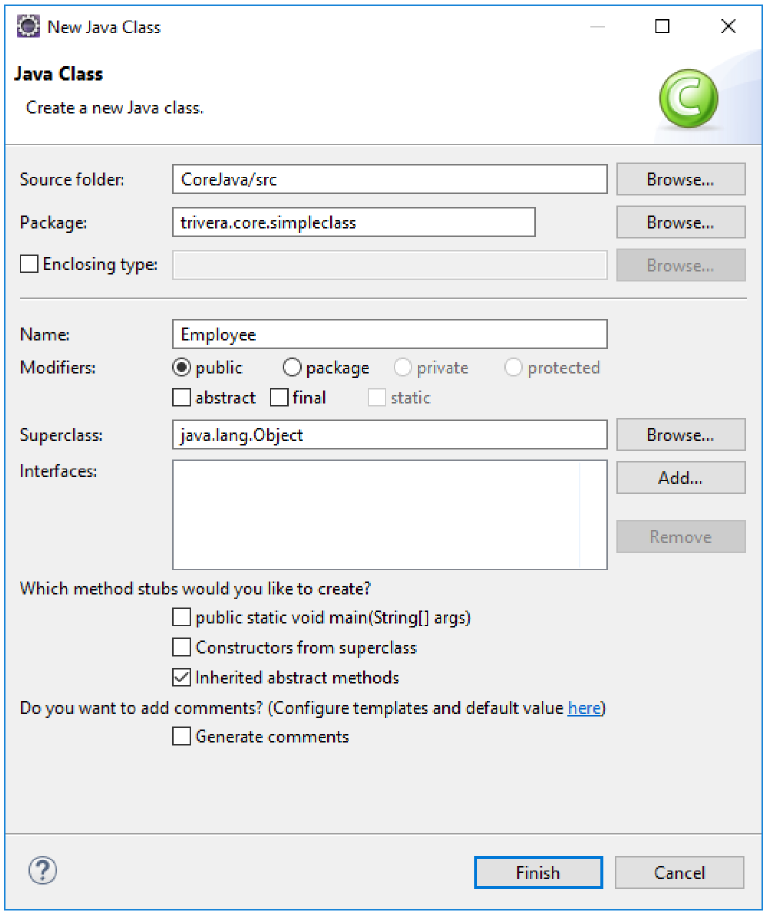
Create the Employee class

2. Next, we are going to use the Eclipse-based class creation wizard to create a new class called "**Employee**"
3. Right click on the project and select **New->Class**:



4. The class creation wizard will open. Use the package name **trivera.core.simpleclass**, a class name of **Employee**

See the picture below for the correct inputs in context.


New Java Class

Java Class

Create a new Java class.

Source folder: CoreJava/src

Package: trivera.core.simpleclass

☐ Enclosing type:

Name: Employee

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

5. Select **Finish** to return to the workspace and an editor will open with your class displayed.
6. Modify the class so that it contains these two attributes:

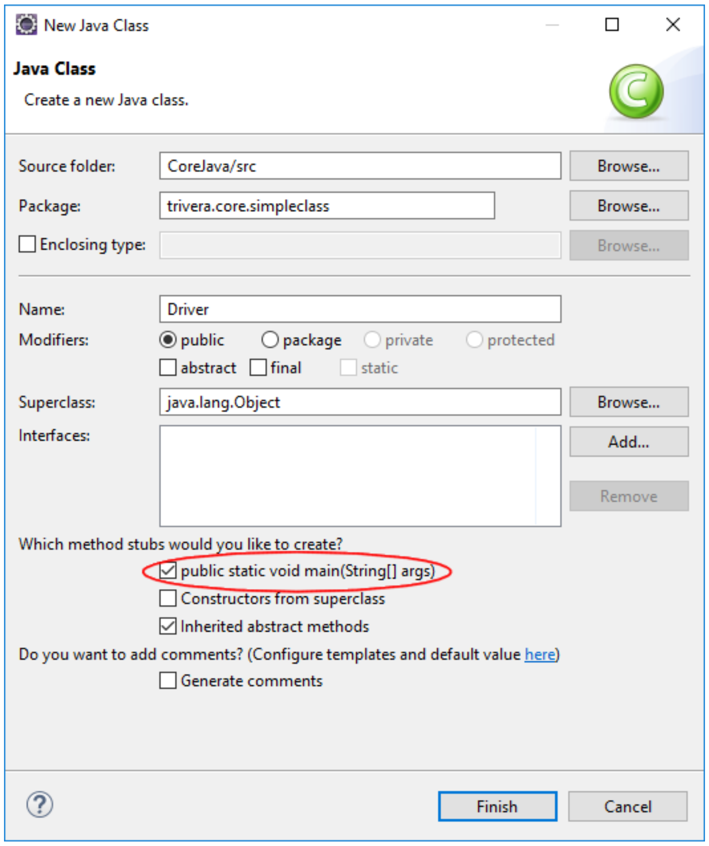
```
int         salary;
String      name;
```

These two fields represent the instance data, or state, of your class.

Create the 'driver' class

To run a Java application a least one class must contain a main method. You will be defining this main method in a second class called 'Driver'

7. Right-click on the project and select **New->Class**
8. Use the package name **trivera.core.simpleclass**, a class name of **Driver**, and **check the box to create the method** stub for **public static void main (String[] args)**. See the picture below for the correct inputs in context.


New Java Class

Java Class

Create a new Java class.

Source folder: CoreJava/src

Package: trivera.core.simpleclass

☐ Enclosing type:

Name: Driver

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

9. Modify the main method (public static void main (String args[])) adding the following lines of code:

```
Employee employee = new Employee();
employee.name = "Jim";
employee.salary = 30000;
```

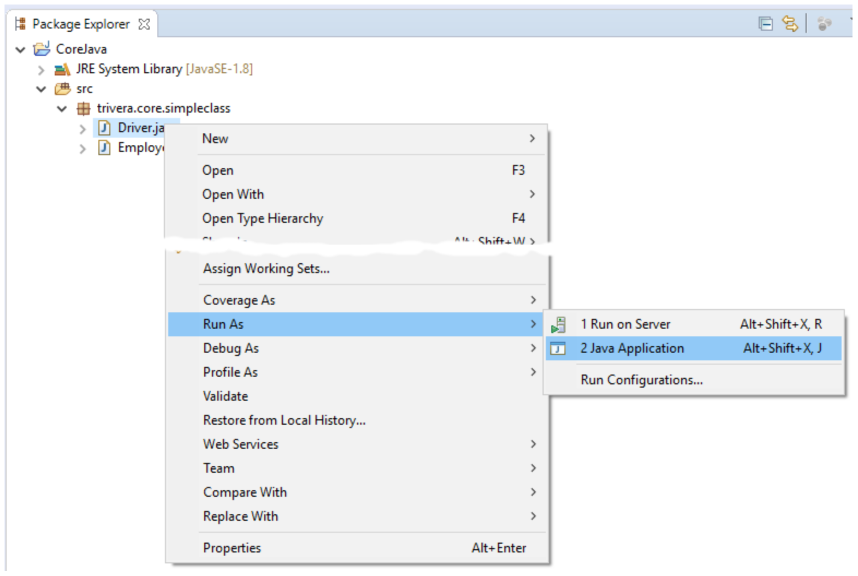
This will cause an instance of this class to be created, and the instance variables will be set.

10. Finally, add a print statement just to see what we entered:
11. `System.out.println(employee.name + " makes $" + employee.salary);`
12. Saving the file will cause the IDE to compile the recently edited code. Resolve any compilation errors prior to continuing.

Note: It is always important to keep in mind that compilation errors are generated by completely different software (compiler) than runtime errors (JVM). This is a first step in properly diagnosing a problem.

Testing Your Work

13. Run the **trivera.core.simpleclass.Driver** class as a Java application. (right-click on the Driver class and select **Run As -> Java Application**)



Expected Results

You should see the same data you entered as output:

```
Jim makes $30000
```

Notes

Note that when the main method first starts executing, there is no instance of this class. For now, each one of our main methods will contain the single line of code to construct an instance of its containing class, followed by any code needed to manipulate this new instance.

Saving Your Progress

Throughout the lab guide, we may use the same project over and over again. If you would like to save your progress along the way, one way to do so is to make a copy of the project in Eclipse. Do so by right-clicking on the project, and selecting **Copy**. Right click in the project explorer, select **Paste** and name the new project appropriately. At that point you can right click on the new copied project and select **Close Project** to minimize the number of resources Eclipse is using on your system.

Exercise 2. CREATE A CLASS WITH METHODS

Overview	
In this exercise you will use the code from the previous lab and add set and get methods to access the attributes.	
Objective	Understand the basics of different types of methods
Builds on Previous Labs	Exercise "Create a Simple Class"
Exercise Folder	Methods
Time to Complete	30 minutes

Task List

Note: Since this lab builds on the previous lab, you can continue without importing any new code. However, if you did not complete the previous lab and want to start on this lab, you will have to create a project named **Methods**, and you will find a starting point for this lab at **~/StudentWork/code/Methods/lab-code**

1. Edit the **Employee** class and add (and implement) the following methods to provide read and write access to your fields:

```
public void setSalary(int salary)
public int getSalary()
public void setName(String name)
public String getName()
```

Note that many editors have built in functionality for generating these methods. Try right-clicking in the editor window, and look for a menu option such as 'Generate setter and getter methods'

2. Create the following three constructors for your class:

```
public Employee()
public Employee (String name)
public Employee (String name, int salary)
```

- The first constructor should call the second constructor with a default name (for example, this ("John Doe");).
- The second constructor should call the third constructor with the name and a default salary (for example, this (name, 25000);).
- **Only the third** constructor should store data into the instance fields.

3. Edit the Driver class and modify the main method to create three instances of your class

An instance with the blank class should already exist – add two more Employee variables, empployee2 and employee3, one passing in just a name and the other passing in both a name and an initial salary).

4. Use the **set<XXX>** methods to change those values in your instance. Basically, give yourself a raise.
5. Change your output to print three lines, one for each Employee instance:

```
System.out.println(employee2.getName() + " makes $" +
employee2.getSalary());
```

6. Saving the file will cause the IDE to compile the recently edited code. Resolve any compilation errors prior to continuing.

Testing Your Work

7. Run the **trivera.core.simpleclass.Driver** class as a Java Application.

Expected Results

You should see the default values for two of your objects, and the final name and salary values you specified when you explicitly invoked the set<XXX> methods.

Exercise 3. LOOPING

Overview	
In this exercise you will write a simple for loop that increments a counter and prints on each iteration.	
Objective	Understand the basics of looping
Builds on Previous Labs	None
Exercise Folder	Looping
Time to Complete	15 minutes

Task List

1. Create a new Java Project called **Looping**.
2. Create a new class called **Looping**, with a package name of `trivera.core.lang`, and an auto-generated method stub for **public static void main(String[] args)**. (See the instructions in the **Create a Simple Class** lab for detailed instructions on how to do this)
3. This class has only one method, ‘main’.
4. In main, define a **for** loop.
5. In the loop declaration, declare and initialize an integer counter to 0.
6. Loop from 0 to less than 10.
7. Within the loop, print out the value of your counter.
   ```
   System.out.println ("Current iteration: " + i);
   ```
8. Don’t forget to close your loop, the main method and the class (three closing curly braces at the end)!
9. Resolve any compilation errors prior to continuing.

Testing Your Work

10. Run the `trivera.core.lang.Looping` class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

You should see ten lines of output, from 0 to 9, that show the value of the counter being incremented.

Notes

For this lab you do not need to construct an instance of your class from main. Simply put the loop directly in the main method.

Challenge:

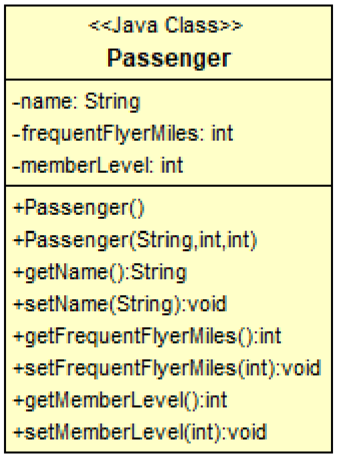
If you have time at the end of this lab, try breaking your code in a variety of ways (both compilation and runtime). Practice examining the error messages and how they are displayed in your IDE. Even though you often will know the error, try to diagnose it. This is also a great place to start trying out the debugger in your IDE.

Exercise 4. LANGUAGE STATEMENTS

Overview	
In this exercise, you will use the if and switch statement to conditionally print values	
Objective	Become familiar with the if and switch language statements
Exercise Folder	LanguageStatements
Time to Complete	30 minutes

About

During this exercise you will be implementing the checkIn method of the Airport class. This method accepts a reference to an object of type Passenger



A Passenger has a name, a certain number of frequent flyer miles and a member level. When the Passenger does not have any frequent flyer miles (zero miles), he/she will be a new passenger, otherwise he/she is considered a frequent flyer.

When the Passenger is at member level 1, the passenger will be a Bronze member. When the passengers collect more than 5000 miles, they will get to member level 2 (Silver member) and when they reach 10000 miles they will be given member level 3 (Gold member)

1. Create a new Java Project called **LanguageStatements**.
2. Import the source code at **~/StudentWork/code/LanguageStatements/lab-code**. Examine the newly imported files.

Besides the Passenger class, an **Airport** class has been imported. This is the only class you will need to edit during this exercise. The Driver class has already been implemented and contains the main method.

3. Open the **Passenger** class and inspect its implementation
4. Open the **Airport** class.
5. Implement the **checkIn** method

- Using the **if** statement, check the frequent flyer miles of the passenger. When the Frequent Flyer mileage of the passenger is zero, print **New Passenger,** else print **Frequent Flyer**
- Next, print the passenger **name** and **Frequent Flyer mileage**

- Using the **switch** statement print out the Member Level.The member level is defined as **int memberLevel** within the Passenger class
  1 = Bronze
  2= Silver
  3 = Gold

6. Open the **Driver** class (no editing required)

Four instances of the Passenger class have been created. On an instance of the Airport class the checkIn method will be invoked for each of the four passengers

7. Run the **Driver** class as a Java Application and make sure the output resembles the output shown below

```
Frequent Flyer
Passenger Fred has 16500 miles
Gold Member

New Passenger
Passenger Wilma has 0 miles
Bronze Member

Frequent Flyer
Passenger Barney has 8500 miles
Silver Member

Frequent Flyer
Passenger Betty has 4500 miles
Bronze Member
```

Exercise 5. FUN WITH STRINGS

Overview	
In this exercise you will create four strings, print out their attributes, and compare them to each other for equality.	
Objective	Understand the basics of String operations
Builds on Previous Labs	None
Exercise Folder	Strings
Time to Complete	30 minutes

Task List

1. Create a new Java Project called **Strings**.
2. Create a new class called **FunWithStrings**, with a package name of `trivera.core.strings`, and an auto-generated method stub for **public static void main(String[] args)**.
3. You will perform all of this work in your main method.   You do not need to construct an instance of this class.
4. You will start by creating four String objects. We recommend the following:
```
String     string1 =  "Hello";
String     string2 = new String("Hello");
String     string3 = "How are you?";
String     string4 = string3;
```
5. Print out each of the four String objects using the printing mechanism you previously used. In addition to printing the String, you should also print its length.
```
System.out.println (string1 + ":" + string1.length());
```
6. Use if-else statements to compare the equality of the first two strings. Use the equals method of String, and print out if they are the same or not
```
if (string1.equals(string2)) {
  System.out.println ("String 1 and 2 are equal");}
else {
  System.out.println ("String 1 and 2 are NOT equal");}
```
7. Create a second if-else statement but this time use the == to see if the two String references are equal. If they are, print out a message saying so. Otherwise, use the else block to print the message stating they are not.
```
if (string1==string2) {
  System.out.println ("String 1 and 2 are ==");}
else {
  System.out.println ("String 1 and 2 are NOT ==");}
```
8. Perform the same two tests, but this time with string3 and string4.
9. Convert string1 and string3 to upper case
```
string1 = string1.toUpperCase();
```
10. Again compare string1 to string2 and string3 to string4, but this time use the comparison method of String that is case-insensitive (equalsIgnoreCase).
11. Resolve any compilation errors prior to continuing.

Testing Your Work

12. Run the `trivera.core.strings.FunWithStrings` class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Take special note of the output to see if it is what you expected.

Expected Results

You should see that strings that have the same contents are equal when the equals method is used.

When a String reference is assigned to another String reference, both the contents and the reference values are equal.

The equalsIgnoreCase method allows the contents of two String objects to be compared and proven to be equal when the content characters are the same, but with differing case.

Notes

While doing this lab, feel free to experiment with other methods of String. You can explore the documentation for these through JavaDocs and other code assist mechanisms that are available to you (another excellent habit to get into when coding).

Exercise 6. USING STRINGBUFFERS AND STRINGBUILDERS

Overview	
In this exercise you will build up text in both StringBuffer and StringBuilder instances and print them out.	
Objective	Understand the how both StringBuffers and StringBuilders work
Builds on Previous Labs	None
Exercise Folder	StringBuilders
Time to Complete	30 minutes

Task List

1. Create a new class called **StringBuffers**, with a package name of `trivera.core.strings`, and an auto-generated stub for the **public static void main(String[] args)** method.
2. You will perform all of this work in your main method. You do not need to construct an instance of this class.
3. Instantiate a StringBuffer object, using the default constructor.
```
StringBuffer sb = new StringBuffer();
```
4. Add some text to the StringBuffer.
```
sb.append("Hello");
```
5. Add some more text to the StringBuffer.
```
sb.append(", I’m home!");
```
6. Convert the StringBuffer to a String.
```
String newString = sb.toString();
```
7. Print the contents of the String.
```
System.out.println(newString);
```
8. Construct a new StringBuffer instance using the String just obtained.
```
sb= new StringBuffer(newString);
```
9. Add yet some more text to the new StringBuffer.
```
sb.append(" What’s for dinner?");
```
10. Print the contents of the StringBuffer as a String (note that the println method automatically converts any object into a String):
```
System.out.println(sb);
```
11. Replicate the functionality and actions performed above using the StringBuilder class. You can copy and paste the code, change variable names and the type. All methods and results will be exactly the same
12. Resolve any compilation errors prior to continuing.

Testing Your Work

13. Run the target classes as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

You should see all of the text appear when you print the StringBuffer at the end of your method. This will be replicated in the StringBuilder implementation as well.

Exercise 7. CREATING SUBCLASSES

Overview	
In this exercise you will create a superclass and a subclass with overriding methods.	
Objective	Understand the basics of inheritance
Builds on Previous Labs	None
Exercise Folder	SubClasses
Time to Complete	15 minutes

Task List

1. Create a new Java Project called **SubClasses**.
2. Import the source code at **~StudentWork/code/SubClasses/lab-code**. Examine the newly imported files.
3. There are three classes to complete – the base class Person, the Derived class Employee, and an executable class Test, meant to do just that.
4. First, add a single String attribute `name` to the Person class. Make this attribute private and write (or generate) the set/get methods. Create a constructor for `Person`, which takes a String argument and assigns it to the instance variable `name`. Provide overriding implementations of the `equals` and `toString` methods:

```
public boolean equals (Object o) {
   if (o instanceof Person) {
      Person p = (Person) o; //cast the Object
      if (this.name.equals(p.name)) {
         return true;
      }
   }
   return false;
}
public String toString () {return name;}
```

5. Next, complete the `Employee` class. Make sure `Employee` is a subclass of `Person` (using the keyword `extends)`. Add two instance variables:

```
private     int empID;
private     Employee manager;
```

6. Create the set/get methods for your two new instance variables. Create a calcSalary method, which returns 0.0 (implementation is used later):

```
public double calcSalary() {return 0.0;}
```

7. Create a constructor for `Employee` that takes the name and employee ID (Note that we are not currently using the `manager` field). Have the first line of the Employee constructor call the superclass constructor, passing up the name (`super(name);`). Additionally, be sure to save the data for empID in the instance variable. (`this.empID = empID;`)
8. Override the equals and toString methods in Employee, using a call to the super method in each.
9. Complete the `Test` class by instantiating a `Person` (`pers`) and an `Employee` (`emp`) object. Use `System.out.println` to print each object (thus calling the objects' toString method).
10. Resolve any compilation errors prior to continuing.

Testing Your Work

11. Run the target classes as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

You should see a String representation of the person instance and one for the employee instance.

Notes

The equals and toString methods in Employee should call the corresponding methods in Person to get the correct behavior. If you are working with an IDE, use its debugger to examine what methods are being called where.

Exercise 8. FIELD TEST

Overview	
In this exercise you will print out the default values for different types of instance fields.	
Objective	Understand the nuances of instance fields
Builds on Previous Labs	None
Exercise Folder	FieldTest
Time to Complete	15 minutes

Task List

1. Create a new Java Project called **FieldTest**.
2. Import the source code at **~StudentWork/code/FieldTest/lab-code**. Examine the newly imported file: `trivera.core.fields.FieldTest`
3. Insert code for the following instance fields:
```
private String string;
private int x;
private double y;
private boolean yn;
```
4. For this lab, there is no need to create the set/get methods. In the main method, construct an instance of your class.
5. In your constructor define the following variable:
```
int x;
```
6. Also, print out the values of each of the four instance fields listed above using `System.out.println`.
7. Compile your class (usually this is done automatically when you save the file within an IDE).

At this point you should get an error regarding x.

8. Modify your code so that it is the attribute x and not the local variable x that is being printed.
9. Compile your class.
10. Modify your definition of the variable x so that it is initialized with some value (ex: `int x=5;`).
11. Add the code in the constructor to print out variable x.
12. Resolve any compilation errors prior to continuing.

Testing Your Work

13. Run the target class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**. Ignore the warning that local variable x is never read.

Expected Results

Your first compilation should fail. At that point you would have been attempting to reference the value of the local variable x, but it has not been initialized yet.

When you use the "`this`" keyword to indicate that you wish to reference the instance field x, the compilation succeeds. Although the instance field has not been explicitly initialized, all instance fields are implicitly initialized by the compiler, making your reference to its value valid.

Notes

A part of this exercise involves creating the distinction between variable x and field x.

You'll also discover that you cannot ask to read the value of variable x until it has been initialized with a value. The same is not true for the fields, since they will be initialized for you implicitly.

Exercise 9. CREATING AN ARRAY

Overview	
In this exercise you will create and populate an array.	
Objective	Understand the basics of array operations
Builds on Previous Labs	None
Exercise Folder	Array
Time to Complete	30 minutes

Task List

1. Create a new Java Project called **Arrays**.
2. Create a new class called **TestArray**, with a package name of `trivera.core.array`, and an auto-generated stub for the **public static void main(String[] args)** method.
3. In the main method, declare a String array:
   ```
   String myArray[];
   ```
4. Instantiate the array with five members:
   ```
   myArray = new String[5];
   ```
5. Assign a String value to each member of the array (optionally, do this in a for loop):
   ```
   myArray[0] = "First element"; //do this four more times
   ```
6. Use a "for loop" to print the elements of the array. Assume that you do not know the size of the array (use the `length` attribute of the array).
7. Resolve any compilation errors prior to continuing.

Testing Your Work

8. Run the target class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

The contents of the array should be displayed.

Challenge:

As with previous labs, try to break this lab. Use the debugger to step through the processing of the array.

Exercise 10. USING LOCAL-VARIABLE TYPE INFERENCE

Overview	
During this exercise, you will use the var keyword introduced in Java 10 to define type 'type' of each local variable while implementing the method.	
Objective	Using the var reserved word to define local variables
Exercise Folder	UsingVar
Approximate time	20 minutes

Setup for the exercise

1. Create a new Java Project (Eclipse) / Module (IntelliJ) called **UsingVar**.
2. Import the source code at **~StudentWork/code/UsingVar/lab-code**.

About the exercise

We provided a Mocked implementation of a Flight repository, which allows clients to provide a List of destination codes to retrieve a List of Flight instances going to these destinations.

```
List<Flight> getFlightsForDestination(
                      List<String> destinationCodes);
```

3. Inspect the **Flight** class

Each instance of this class represents a single departing flight.

4. Inspect the **getFlightDescription** method of the **Flight** class

This method returns a brief description of the Flight. You will be using this method when implementing the client class

5. Open **RepositoryClient**

You will be implementing the **listFlights** method. Within this method you will obtain a list of Flight objects from the repository. Once you have obtained the list, you need to create a List of flight descriptions (List of String objects), where each entry in the list does not only contain the description of each flight (obtained by invoking the **getFlightDescription** method on the flight class, but also the index (position) in the List.

Since the order of the flights being returned is significant to the application and we are certain that the number of flights that is being returned is limited, we will not be using the Java Stream API to implement the method, but will be using the good-old for loop to iterate over the elements.

Implement the listFlights method

You will only need to provide an implementation of this single method and will only be declaring local variables (within the method).

None of the variables defined within this method should have an explicit type! In other words, every time you need to define a variable, you should declare the variable using the var reserved word, instead of using an explicit type (like int or String)

6. Using the static Arrays.asList method, add the method parameters to a List and assign this list to a local variable.

```
var destinations = Arrays.asList(destinationCodes);
```

7. Obtain a reference to the **FlightRepository** by invoking the static **getDefaultInstance** method on the interface

```
var repository = FlightRepository.getDefaultInstance();
```

8. Invoke the getFlightsForDestination method on the repository, assign the result to another local variable
```
var flights = repository.getFlightsForDestination(destinations);
```
9. Create an instance of Arraylist and assign its reference to yet another local variable. You will be populating this results list with the description of each flight.
```
var results = new ArrayList<>();
```
10. Using a for loop, iterator over the elements in the flights list. (Make sure you are still not declaring explicit types!).
```
for (var i = 0; i < flights.size(); i++) {
```
11. Obtain a reference to each entry in the flights list
```
var flight = flights.get(i);
```
12. Obtain the description of the flight
```
var description = flight.getFlightDescription();
```
13. Create a String, using both the index in the list and the flight's description
```
var flightString = String.format("%d) %s", i, description);
```
14. Add the String to the List
```
results.add(flightString);
```

If you are using the IDE's code completion, notice the parameter type of the **add** method!

15. Return the results list…
```
return results;
```

You should not get a compilation error

```
Type mismatch: cannot convert from ArrayList<Object> to List<String>
```

The method declares to return a List<String>, but we are trying to return a List<Object>. When creating the results list, we used the diamond shape to declare the variable.

```
var results = new ArrayList<>();
```

Before the introduction of the var reserved word, you would declare the variable as

```
List<String> results = new ArrayList<>();
```

Allowing us to use the diamond shape when instantiating the object, because the generic type could be inferred from its context. Since is now no longer the case, we need to define the generic type of the right-hand of the assignment

16. Change the definition of the results List, adding the generic type
```
var results = new ArrayList<String>();
```

The compilation error(s) should now disappear.

17. Run the client, a total of 19 flights will be returned, sorted by departure time and starting with an index

```
0) 1077 - Las Vegas 7:00 AM
1) 351 - San Francisco 7:15 AM
2) 2531 - Los Angeles 7:19 AM
3) 809 - San Francisco 8:20 AM
4) 133 - San Francisco 8:25 AM
5) 1833 - San Francisco 10:35 AM
6) 477 - San Francisco 10:45 AM
7) 147 - Los Angeles 2:55 PM
8) 1572 - San Francisco 4:05 PM
9) 353 - San Francisco 4:25 PM
```

```
10) 487 - Los Angeles 4:55 PM
11) 641 - Las Vegas 5:17 PM
12) 367 - Los Angeles 5:45 PM
13) 305 - Los Angeles 5:59 PM
14) 833 - San Francisco 6:10 PM
15) 576 - San Francisco 6:15 PM
16) 357 - San Francisco 6:59 PM
17) 777 - Las Vegas 7:55 PM
18) 687 - Los Angeles 9:15 PM
```

Exercise 11. SALARIES - POLYMORPHISM

Overview	
In this exercise you will work with polymorphism in a set of inheritance-based classes.	
Objective	Understand the basics of polymorphism
Builds on Previous Labs	None
Exercise Folder	Polymorphism
Time to Complete	30 minutes

Task List

1. Create a new Java Project called **AdvJava**.
2. Import the source code at **~StudentWork/code/Polymorphism/lab-code**. Examine the newly imported files listed above.
3. Complete the `SalesEmployee` class by extending the `Employee` class.
4. Next define the get and set methods for the `salesAmount` and `commissionRate` attributes (these should already be declared in the skeleton code provided).
5. Define a constructor which accepts a name and an employee number and use `super` to call the `Employee` constructor, passing in the name and empID. Finally override the `calcSalary` method and calculate the salary based on sales:
```
public double calcSalary() {
  return salesAmount * commissionRate;
}
```
6. Complete the `HourlyEmployee` class by extending the `Employee` class. Add the get and set methods for `hoursWorked` and `hourRate`. Define a constructor which accepts a name and an employee number, and again use the constructor of Employee. Finally override the `calcSalary` method and calculate the salary based on hoursworked:
```
public double calcSalary() {return hoursWorked * hourRate;}
```
7. Complete the TestPayment class by instantiating the `jennifer` and `john` variables (already declared). Set values for salesAmount and commission for Jennifer:
```
jennifer.setSalesAmount(3000);
jennifer.setCommissionRate(.1);
```
8. Now set values for hourRate and hoursWorked for john:
```
john.setHourRate(15.5);
john.setHoursWorked(40);
```
9. Now set staff[0] to Jennifer and staff[1] to john (the staff array has already been declared and instantiated). Finally, loop through the list of Employees and print out their salaries.
10. Resolve any compilation errors prior to continuing.

Testing Your Work

11. Run the `trivera.core.employee.TestPayment` class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

On the output you should see a salary for Jennifer and John. Jennifer’s salary is based on sales and John’s salary is based on hours worked.

Exercise 12. MAILABLE - INTERFACES

Overview	
In this exercise you will work with an interface, creating it and then implementing it in several classes.	
Objective	Understand how to work with interfaces
Builds on Previous Labs	None
Exercise Folder	Interfaces
Time to Complete	20 minutes

Task List

1. Create a new Java Project called **Interfaces**.
2. Import the source code at **~StudentWork/code/Interfaces/lab-code**. Select **Yes to All** when prompted to overwrite existing resources. Examine the newly imported files listed above.
3. Complete the interface trivera.core.employee.Mailable by adding the following method:
```
        public void mailNote(String note);
```
4. Complete the `Company` class by implementing Mailable (and therefore overriding the mailNote method). Use a System.out.println as implementation for mailNote:
```
public void mailNote (String note){
  System.out.println ("Company note:" + note);
}
```
5. Complete the Employee class by implementing Mailable. Again use a System.out.println as implementation for mailNote (but use a different message so that you can distinguish later).
6. Complete the TestMailable class by constructing an array of type Mailable:
```
Mailable [] distribution = new Mailable[2];
```
7. Add a couple of elements of type Employee and Company:
```
distribution[0] = new Company();
distribution[1] = new Employee ("Jim", 111);
```
8. Loop through the mailable array, invoking the mailNote on each element.
```
for (int i=0;i<distribution.length;i++) {
       distribution[i].mailNote("Testing the mail System");
}
```
9. Resolve any compilation errors prior to continuing.

Testing Your Work

10. Run the `trivera.core.employee.TestMailable` class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

The output shows messages being sent by companies and employees.

Exercise 13. EXCEPTIONS

Overview	
In this exercise, you will work with exceptions.	
Objective	Understand how to use exceptions in an application
Exercise Folder	Exceptions
Time to Complete	25 minutes

About

During this exercise you will be implementing a small messaging application. In order to be able to send a message you will need to establish a connection with a messaging server, after which you can send a text message. Naturally any number of things can go wrong when trying to send a message to a remote server which would result in an exception being thrown by the Java implementation.

For this exercise, we will not really make use of a remote messaging service. Instead, we have created a mocked implementation that will only display the message onto the console.

Setup the project

1. Create a new Java Project called **Exceptions**.
2. Import the source code at **~StudentWork/code/Exceptions/lab-code.**
3. Inspect the trivera.exceptions.messaging.connection.ConnectionFactory interface.

This interface has a single **static** method, **getConnection**, which accepts the username and password to establish the connection. When, for any reason, the connection cannot be established, an **IOException** is thrown. When a connection can be made, an instance of **MessagingConnection** will be returned.

```
MessagingConnection connection =
      ConnectionFactory.getConnection("Jennifer", "123");
```

4. Inspect the **MessagingConnection** interface.

This interface defines the methods allowing us to send messages and close the connection when done. When using a connection to send messages, we need to make sure the connection is properly closed once we are done sending message to avoid any resource leaks.

5. Inspect **trivera.exceptions.MessagingClient**.

This is the test class that you will execute once you have completed the exercise. As you can see, it creates an instance of **MessagingService** and invokes the **sendMessages** method which provides username, password and the actual text message to be sent.

```
MessagingService service = new MessagingService();
service.sendMessage("Fred", "123", "Hello World");
```

6. Open **trivera.exceptions.messaging.MessagingService**.

It will be up to you to implement the **sendMessage** method.

7. Using the **ConnectionFactory**, obtain an instance of **MessagingConnection**.

Note: When invoking the **getConnection** method, keep in mind that an **IOException** might be thrown.

```
MessagingConnection connection =
  ConnectionFactory.getConnection("Fred", "123");
```

8. Using the **MessagingConnection** instance, send the message.

Note: Invoking **sendMessage** might also result in an **IOException**.

```
connection.sendMessage(message);
```

9. ‘Deal’ with any exceptions that might be thrown by printing their stacktrace onto the console.

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

Note: Naturally this is not a proper way to deal with an exception. Normally, you would probably log the error and notify the client application about the problem.

10. Make sure that the connection is properly closed.
11. Run the MessagingClient and inspect the output:

```
Fred (11/07/17 14:25) : Hello World
java.io.IOException: Could not send message (invalid message format)
…
```

An IOException ***should*** appear, since the second message that is sent by the client is invalid. 😉

Note: In case you did not properly close the connection, a message will appear on the console:

```
Possible resource leak, Connection still open!
```

If this is the case, go back to your code and make sure the connection is closed in a **finally** block!

```
MessagingConnection connection = null;
try {
   connection = ConnectionFactory.getConnection("Fred", "123");
   connection.sendMessage(message);
} catch (IOException e) {
   e.printStackTrace();
} finally {
   close(connection); //created helper method to close connection!
}
```

Exercise 14. EXCEPTIONAL

Overview	
In this exercise, you will work with AutoCloseable resources, creating Exception classes and throwing exceptions	
Objective	Understand how to define and then use an exception class Work with AutoCloseable resources
Builds on Previous Labs	Exceptions
Exercise Folder	Exceptional
Time to Complete	25 minutes

About

You will continue to work (and refactor) the MessagingService application you implemented in an earlier exercise. Beside improving the exception handling, you will also improve the resource handling (closing of the connection).

Making the Connection AutoCloseable

1. Open the **MessagingConnection** interface and have this interface extend the **AutoCloseable** interface.

```
public interface MessagingConnection extends AutoCloseable
```

The **AutoCloseable** interface defines the close method. **MessagingConnection** also defines the close method, but the close method of the **MessagingConnection** declares it throws **IOException** instead of **Exception**. Do ***not*** remove the close methods from the **MessageConnection** interface.

2. Open the **MessagingService** class.

Now that **MessagingConnection** extends **AutoCloseable**, an instance of this interface can be created within a try-with-resources block.

3. Modify the implementation of the **sendMessage** method and obtain the connection within a try-with-resources block.

```
public void sendMessage(String username, String password, String message)
{
    try(MessagingConnection connection =
ConnectionFactory.getConnection("Fred", "123");) {
            connection.sendMessage(message);
      }
   …
}
```

4. Remove the **finally** block. It is no longer needed since the resource is closed by the runtime.
5. Run the **MessagingClient** and make sure the resources are properly closed (and **IOException** will still be thrown for the second message that is sent by the Client).

Create a Custom Exception Class

When length of the username is less than 4 characters, an **InvalidUsernameException** should be thrown. Since this **Exception** subclass does not exist, you will have to create it yourself and add this logic to the **MessageService** class.

6. Create a new class called **trivera.exceptions.messaging.InvalidUsernameException**.
7. Make sure the **InvalidUsernameException** class extends **Exception**.

```
public class InvalidUsernameException extends Exception
```

8. Define an instance variable called username and create a getter method to get the value of this field.

```
private String username;

public String getUsername() {
      return username;
}
```

9. Create a constructor that accepts the message and the invalid username. Make sure the message is passed to the superclass and the username is stored in the instance field.

```
public InvalidUsernameException(String message, String username) {
   super(message);
   this.username = username;
}
```

10. Open the **MessagingService** class and edit the **sendMessage** method. Add a check on the username and make sure an **InvalidUsernameException** is thrown when the username contains less than 4 characters.

```
if(username == null || username.trim().length() < 4) {
  throw new InvalidUsernameException("Invalid username", username);
}
```

11. Update the method signature of the **sendMessage** to indicate that this method might throw an **InvalidUsernameException**.

```
public void sendMessage(String username,
    String password, String message) throws InvalidUsernameException
```

12. Update the **MessagingClient** and fix the compilation error by adding exception handling to the **sendMessages** method.

Rethrow Exceptions (Optional)

The **sendMessage** method of the **MessageService** does not yet notify the client when an exception occurs while sending the message, it only dumps the stacktrace onto the console.

13. Create a new class called **trivera.exceptions.messaging.MessagingException**, which extends **Exception**.

One of the constructors of the Exception class accepts both an error message and an instance of Throwable which represents the cause of the exception.

14. In the **MessagingException** class, create a constructor that accepts a String (error message) and a Throwable (cause). Make sure you send both parameter values to the constructor of the superclass.

```
public MessagingException(String message, Throwable cause) {
           super(message, cause);
}
```

15. Edit the **sendMessage** method of the **MessagingService** class. In the catch block, replace the ‘printStackTrace’ by creating and throwing a new instance of **MessagingException**, providing an error message and ‘wrapping’ the IOException.

```
} catch (IOException e) {
  throw new MessagingException("Error while sending message", e);
}
```

16. Change the method signature of the **sendMessage** method to indicate that this method now might throw a **MessagingException** in addition to the **InvalidUsernameException**.

```
public void sendMessage(String username, String password, String message)
throws InvalidUsernameException,MessagingException
```

17. Update the **MessagingClient** to fix any compilation errors that have occurred after making these changes to the service class.

Hint: You might want to use the ‘multi-catch’ construct in the client to deal with both the **InvalidUsernameException** and **MessagingException**.

Exercise 15. USING PRIMITIVE WRAPPERS

Overview	
In this exercise, you will construct primitive wrappers from strings and compare them against their primitive types.	
Objective	Understand the basics of what wrappers are and how to use them
Exercise Folder	PrimitiveWrappers
Time to Complete	20 minutes

Task List

1. Create a new Java Project called PrimitiveWrappers.
2. Import the source code at **~StudentWork/code/PrimitiveWrappers/lab-code**.
3. Examine the newly imported file: **trivera.core.lang.Primitives**
4. In the skeleton code, there are already four Strings declared and four primitive data types declared. You must declare the primitive wrapper class for each primitive, then use the static parseXXX, methods that lets you pass in a String to create the object (Use the pre-declared Strings):

```
Integer intObj = Integer.parseInt(intString);
```

5. Compare the primitives to the primitive wrappers to see if they are equal:

```
if (intVal == intObj.intValue())
     System.out.println("The ints match");
```

6. Repeat this for all four data types.
7. Resolve any compilation errors prior to continuing.

Testing Your Work

8. Run the target class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

You should see that all of the values are equal.

Exercise 16. ENUMERATIONS

Overview	
In this exercise, you will add state and behavior to an implementation of a USState enumeration.	
Objective	Understand how to create and work with your own enumerations
Exercise Folder	Enumerations
Time to Complete	30 minutes

Complete the `trivera.core.enumerations.USState` Class

1. Create a new Java Project called **Enumerations**.
2. Import the source code at **~StudentWork/code/Enumerations/lab-code**. Examine the newly imported files.
3. Within this exercise, you will be adding state and behavior to the new enumeration. In order to accomplish this, the enumeration has been placed into its own class, **trivera.core.enumerations.USState**. Working with this class:

- Inspect the definition of the enum and notice that each enum type, now has two additional arguments
- Define two private constants of type **String**, called **stateName** and **stateCapital**
- Create a constructor for this enum that takes the **stateName** and **stateCapital** as **String** objects. Set the value of the two constants to the values of these arguments
- Define two getter methods to return the values of **stateName** and **stateCapital**

Complete the `trivera.core.enumerations.TestClientAdv Class`

4. Working with the **trivera.core.enumerations.TestClientAdv** class:

- Use the new for-loop feature to iterate over all USState enumeration types
- Invoke **getStateName()** and **getStateCapital()** on each enum to display the name and the capital of each state in the U.S.

5. Resolve any compilation errors prior to continuing.
6. Run the **trivera.core.enumerations.TestClientAdv** class as a Java Application by right-clicking on the file entry and selecting **Run As -> Java Application**.

Expected Results

You should see the names and capitals of all 50 states.

Exercise 17. AGENDA

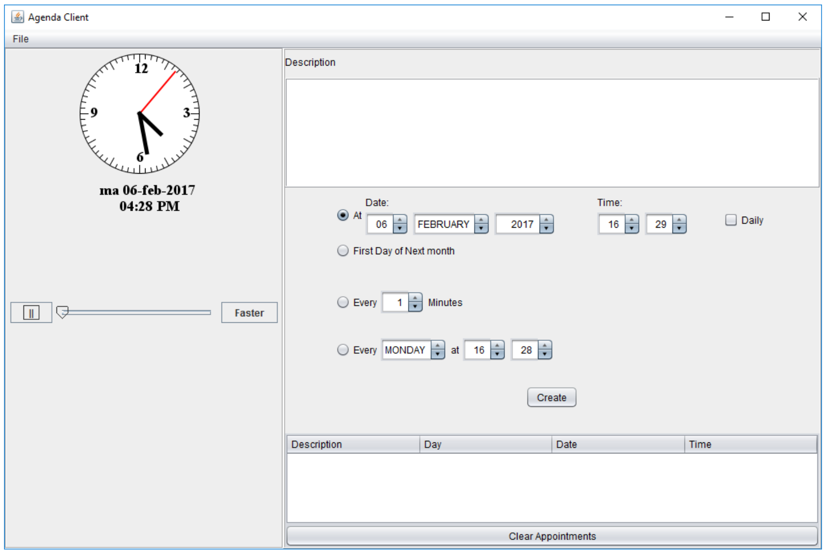
Overview	
You will be using the Date/Time API to perform several operations on date/time values. By using the API you will be creating Appointments within an agenda. Using the standard available methods on the classes you will be able to create repeatable appointments and appointments a certain moments in time.	
Objective	Work with the Date/Time API
Builds on Previous Labs	None. Creates project **Agenda**
Approximate time	45 minutes

Setup for the exercise

1. Create a new Java Project called **Agenda**.
2. Import the source code at **~StudentWork/code/Agenda/lab-code**. After importing the code, you should not have any compilation errors. Examine the newly imported files.

Run AgendaClient

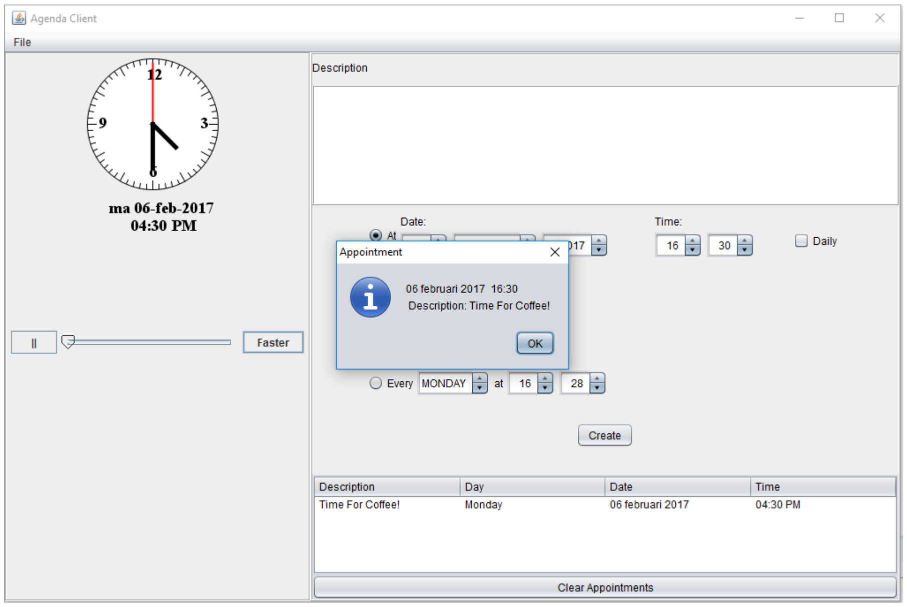
For this exercise a Swing application has been developed to simulate an agenda application.



On the right-hand side of the application, you can define you appointments at given moments in time. A List of scheduled appointments will be displayed on the bottom-right.

To make sure you don’t have to wait several hours for the appointment(s) to ‘fire’. You can use the slider (and button) underneath the clock to speed up time a bit.

At the moment an appointment is scheduled, a pop-up will be displayed.



ma 06-feb-2017
04:30 PM

06 februari 2017 16:30
Description: Time For Coffee!

Time: 16 30

Every MONDAY at 16 28

Description	Day	Date	Time
Time For Coffee!	Monday	06 februari 2017	04:30 PM

When you run the application, you will notice that no appointments can be scheduled yet (nothing happens when you click on create). That's because during this exercise you will be implementing the factory methods that create appointments, using the Date/Time API.

Inspect the `Appointment` class

3. Have a look at the `trivera.datetime.appointment.Appoinment` class. It only contains two properties (description and dateTime). During this exercise you will be responsible so creating instances of this class by implementing factory methods within the `AppointmentFactory` class.

Edit AppointmentFactory

4. The `AppointmentFactory` class is the only class you will be working on during this exercise. You will be implementing the different factory methods of the class that are used by the application client to create instances of `Appointment`

Within the methods you are about to implement you will be asked to create instances of the Appointment class for a given date and time. Once you have created an instance of this class, the appointment can be scheduled by invoking the `addAppointment` method:

```
String description = …
LocalDateTime dateTime = …
Appointment appointment = new Appointment(description, dateTime);
addAppointment(appointment);
```

Implement createAppointment

5. Create an instance of Appointment given the description and a moment in time. Make sure that the appointment is scheduled at exactly the given moment. So when the appointment is set of 11:45 AM, the appointment should be at exactly this time, not a random moment in between 11:45 and 11:46.

Implement createDailyAppointment

6. Create a number of appointments that are scheduled every day at the exactly the same time. Since this is not a real-life implementation but the purpose is to become familiar with the Date/Time API, you will be creating 7 instances of Appointment, one for each day of the week.

Implement createFirstDayOfNextMonthAppointment

7. Create an appointment that is scheduled for the first day of next month (from today) at noon.

Implement createRepeatingAppointment

8. Create 7 recurring appointments, where the appointments are scheduled every X minutes. (X is provided as method parameter)

Implement createDayAppointments

9. Create 7 recurring appointments where the appointment is scheduled every week (at the given day) at (exactly) the given time.

Run the AgendaClient

10. Run the client application and start scheduling appointments to test the different implementations of you class.
11. When appointments are created, make sure they are created at the expected moment(s) in time and are triggered at the exact time provided.

As a reminder… within this application, you can fast-forward time!