LABORATORY MANUAL

SC2002/CE2002 / CZ2002 : Object-Oriented Design & Programming

COMPUTER ENGINEERING COURSE
COMPUTER SCIENCE COURSE

CCDS
NANYANG TECHNOLOGICAL UNIVERSITY

OVERVIEW

1. LAB EXERCISES

The objective of laboratory experiments is to re-enforce your understanding on the course material and ***rely on your experiences and skills gained from prior level 1 courses***. As this is a level course, you will be expected to explore and discover means of completing the exercises – step-by-steps will be kept to the minimum. Some additional materials will be provided along for your references.
You may need to refer (cheatsheet/s) to other Java syntax like looping, branching, arrays etc to complete this exercise.

There are 5 lab sessions altogether and the lab sessions are conducted in labs of CCDS. The working programming environment runs on the Windows operating system.

Lab 1 – Windows Environment, My First Java Program, Fundamental Programming.
Lab 2 – Methods
Lab 3 – Classes & Objects, Array of Objects
Lab 4 – Inheritance and Polymorphism
Lab 5 – Modern Java Programming

The purpose of these exercises is for you to gain hands-on practical skill in developing Java program/application. Eventually, you will be required to complete an assignment using one of the OO languages.

2. EQUIPMENT

Hardware: Workstation running under the *Windows* environment.
Software: Eclipse IDE,
Java 2 Platform,
Microsoft Visual Studios 2010 (or above).

3. LAB ASSESSMENT *(IMPORTANT!)*

Upon completion of each lab session, you are to demonstrate the working of your work to the lab supervisor in charge. The lab supervisor may request that source codes are to be submitted. This will form part of the lab assessment. Any late submission/demonstration after the lab session will be penalized. Non-original work will also be penalized.

You are strongly encourage to check with your lab supervisor when in doubts and not wait till the last minute.

No make-up is allowed for students who have missed their stipulated lab classes without any acceptable excuse like having a valid leave of absence from the school or on medical leave. The students will be deemed to have failed the particular lab work. In case you have valid reasons to do makeup, you must inform the lecture coordinator in charge immediately. The makeup should be done within the same week, during the lab sessions attended by other groups.

4. ASSIGNMENT

You are also required to submit **an assignment** (to be annouced). You are required to work in groups (size to be determined later) and submit an report with your software deliverables. The assignment will also contribute to your final grade in this course.

The marking of your report is generally based on the following criteria:
1. Structure/organization/presentation of your Java codes;
2. Documentation of codes (whether the program is well commented to aid understanding);
3. Correctness of the program;
4. Thoroughness of the testing of the program;
5. Your understanding of the program;
6. User-friendliness of your program (how easy is it to use your program).

To achieve the above, you need to pay attention to:
1. Document your programs - This includes adding comments to your programs at appropriate locations, adding comments to state the purpose of the program, who the author is, and the time the program is written; in addition, you should also add comments on what is the purpose of each class and method.
2. Write reasonably indented codes and blank lines to enhance the readability of your programs.
3. Check all the test cases given to you to test the correctness of your programs.

Detail requirements for the assignment can be found on the website once the assignment is posted.

5. RULES OF CONDUCT

Please be reminded that **PLAGIARISM** (or copying part of/complete assignment) is considered as **CHEATING**, which is strictly prohibited. *Both the copier and code provider* may be **EXPELLED**.

Lab 1
WINDOWS ENVIRONMENT, MY FIRST JAVA PROGRAM & FUNDAMENTAL PROGRAMMING

1. OBJECTIVE

The objectives of Lab 1 are (1) to learn the basic IDE editor ; (2) to edit a program, (3) to compile and execute a program and (4) write working fundamental programs. This aims to help students to familiarize themselves with the working programming environment and fundamental Java syntax.

You can use either Eclipse IDE to develop your Java programs.
Appendix A & B give a guide on using the Eclipse IDE.

2. My First Java Program

In this exercise, one is going to write his/her first Java program by typing in a sample Java program *MyFirstProgram.java*. You may save your works in your preferred directory. Refer to Appendix B as a guide.

2.1 Create a Java Source File

Type in the following program *MyFirstProgram.java* in the editor window and then save the program in your home directory. Note that the program must be saved as *MyFirstProgram.java.*
[[In Java, the file name (at this instance, MyFirstProgram) must be the same as the class name]]

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

     System.out.println("Hello! This is my first program.");
     System.out.println("Bye Bye!")

  }
}
```

Figure 1

3. *Your Tasks for this LAB*

3.1 Write a program that reads a character from the user and then uses a *switch* statement to achieve what the following *if* statement does.

```
if  ((choice == 'A')  || (choice == 'a'))
       printf("Action movie fan\n");
else if  ((choice == 'C')  || (choice == 'c'))
       printf("Comedy movie fan\n");
else if  ((choice == 'D')  || (choice == 'd'))
       printf("Drama movie fan\n");
else
       printf("Invalid choice\n");
```

Important: Remember to name the source code of this program as **P1.java** and name the compiled class code as **P1.class** inside the sub-directory *lab1*.

Test cases: 'a', 'A', 'c', 'C', 'd', 'D', 'b', 'B'.

Expected outputs: 'a', 'A' – Action movie fan; 'c', 'C' – Comedy movie fan; 'd', 'D' – Drame movie fan; 'b', 'B' – Invalid choice.

3.2 The salary scheme for a company is given as follows:

Salary range for grade A: $700 - $899
Salary range for grade B: $600 - $799
Salary range for grade C: $500 - $649

A person whose salary is between $600 and $649 is in grade C if his merit points are below 10, otherwise he is in grade B. A person whose salary is between $700 and $799 is in grade B if his merit points are below 20, otherwise, he is in grade A. Write a program to read in a person's salary and his merit points, and displays his grade.

Important: Remember to name the source code of this program as **P2.java** and name the compiled class code as **P2.class** inside the sub-directory *lab1*.

Test cases: (1) salary : $500, merit : 10; (2) salaray : $610, merit : 5; (3) salary : $610, merit : 10; (4) salary : $710, merit : 15; (5) salary : $710, merit : 20; (6) salary : 800, merit : 30.

Expected outputs: (1) salary : $500, merit : 10 – Grade C; (2) salaray : $610, merit : 5 – Grade C; (3) salary : $610, merit : 10 – Grade B; (4) salary : $710, merit : 15 – Grade B; (5) salary : $710, merit : 20 – Grade A; (6) salary : 800, merit : 30 – Grade A.

3.3 Write a program to generate tables of currency conversions from Singapore dollars to US dollars. Use title and column headings. Assume the following conversion rate:

1 US dollar(US$) = 1.82 Singapore dollars (S$)

Allow the user to enter the starting value, ending value and the increment between lines in S$. The starting value, ending value and the increment are all integer values. Generate three output tables using the following loops with the same input data from the user:

1. Use a *for* loop to generate the first table;
2. Use a *while* loop to generate the second table; and
3. Use a *do/while* loop to generate the third table.

Place all the codes in the main() method.

Important: Remember to name the source code of this program as **P3.java** and name the compiled class code as **P3.class** inside the sub-directory *lab1*.

Test cases: (1) starting : 1, ending : 5, increment : 1; (2) starting : 0, ending : 40, increment: 5; (3) starting : 40, ending : 0, increment: 5 (treat this case as an error).

Expected outputs:
(1) starting : 1, ending : 5, increment : 1;

```
US$          S$
--------------
1            1.82
2            3.64
3            5.46
4            7.28
5            9.1
```

(2) starting : 0, ending : 40, increment: 5;

```
US$       S$
--------------
0         0.0
5         9.1
10        18.2
```

```
15        27.3
20        36.4
25        45.5
30        54.6
35        63.7
40        72.8
```

(3) starting : 40, ending : 0, increment: 5 (treat this case as an error) – Error input!!

4.4 Write a program that reads the height from a user and prints a pattern with the specified height. For example, when the user enters height = 3, the following pattern is printed:

AA
BBAA
AABBAA

If the height is 7, then the following pattern is printed:

AA
BBAA
AABBAA
BBAABBAA
AABBAABBAA
BBAABBAABBAA
AABBAABBAABBAA

Important: Remember to name the source code of this program as **P4.java** and name the compiled class code as **P4.class** inside the sub-directory *lab1*.

Test cases: 0, 3, 7

Expected outputs: (1) height = 0 – Error input!! (2) height = 3 & (3) height = 7 – same as the sample patterns.

LAB 2: METHODS

1. OBJECTIVE

The objectives of Lab2 are to practise on Java methods.

2. INTRODUCTION

A method is an independent collection of source code designed to perform a specific task. By dividing a problem into sub_problems and solve the sub_problems by methods, we obtain a program which is better structured, easier to test, debug and modify. We will practise on writing methods in Java in this lab.

3. *Your Tasks for this LAB*

3.1 In your preferred directory with sub-directory *lab2*, creatre and save the source code into the file Lab2p1.java, and generate the compiled class code as Lab2p1.class.

You may use the program template in Figure 1 to test your methods developed in this lab. The program contains a **main()** which includes a switch statement so that the following methods can be tested by the user. Write the code for each method and use the suggested test cases to test your code for correctness.

```
import java.util.Scanner;
public class Lab2p1 {
   public static void main(String[] args)
   {
      int choice;
      Scanner sc = new Scanner(System.in);
      do {
            System.out.println("Perform the following methods:");
            System.out.println("1:  miltiplication test");
            System.out.println("2:  quotient using division by subtraction");
            System.out.println("3:  remainder using division by subtraction");
            System.out.println("4:  count  the  number of digits");
            System.out.println("5:  position of a digit");
            System.out.println("6:  extract all odd digits");
            System.out.println("7:  quit");
            choice = sc.nextInt();

            switch (choice) {
               case 1: /* add mulTest() call */
                     break;
               case 2: /* add divide() call */
                     break;
               case 3: /* add modulus() call */
                     break;
               case 4: /* add countDigits() call */
                     break;
               case 5: /* add position() call */
                     break;
               case 6: /* add extractOddDigits() call */
                     break;
               case 7: System.out.println("Program terminating ….");
```

```
            }
        } while (choice < 7);
    }

    /* add method code here */

}
```

Figure 1: Program template for Lab 2.

3.2 Write a method that is to test students ability to do multiplication. The method will ask a student 5 multiplication questions one by one and checks the answers. The method prints out the number of correct answers given by the student. The method *random()* from the *Math* class of the Java library can be used to produce two positive one-digit integers (i.e. 1,2,3,4, …) in each question. A sample screen display when the method is called is given below:

How much is 6 times 7? *42*
How much is 2 times 9? *18*
How much is 9 times 4? *36*
……..
4 answers out of 5 are correct.

The input which is underlined is the student's answer to a question. The method header is:

```
public static void mulTest()
```

Test cases: (1) give 5 wrong answers; (2) give 1 correct answer; (3) give more than 1 correct answer.

Expected outputs: straightforward.

3.3 Write the method divide() which does division by subtraction and returns the quotient of dividing m by n. Both m and n are positive integers (i.e. 1,23,4,…). Division by subtraction means that the division operation is achieved using the subtraction method. For example, divide(12,4) will be performed as follows: 12-4=8, 8-4=4, and then 4-4=0, and it ends and returns the result of 3 as it performs three times in the subtraction operation. No error checking on the parameters is required in the method. The method header is given below:

```
public static int divide(int m, int n)
```

Test cases: (1) m = 4, n = 7; (2) m = 7, n = 7; (3) m = 25, n = 7.

Expected outputs: (1) 4/7 = 0; (2) 7/7 = 1; (3) 25/7 = 3.

3.4 Write the method modulus() which does division by subtraction and returns the remainder of dividing m by n. Both m and n are positive integers. No error checking on the parameters is required in the method. The method header is given below.

```
public static int modulus(int m, int n)
```

Test cases: (1) m = 4, n = 7 (2) m = 7, n = 7 (3) m = 25, n = 7.

Expected outputs: 4 % 7 = 4; (2) 7 % 7 = 0; (3) 25 % 7 = 4.

3.5 Write a method to count the number of digits for a positive integer (i.e. 1,2,3,4,…). For example, 1234 has 4 digits. The method countDigits() returns the result. The method header is given below:

```
public static int countDigits(int n)
```

Test cases: (1) n : -12 (give an error message); (2) n : 123; (3) n : 121456;

Expected outputs: (1) n : -12 - Error input!! (2) n : 123 - count = 3; (3) n : 121456 - count = 6.

3.6 Write the method position() which returns the position of the first appearance of a specified digit in a positive number n. The position of the digit is counted from the right and starts from 1. If the required digit is not in the number, the method should return -1. For example, position(12315, 1) returns 2 and position(12, 3) returns -1. No error checking on the parameters is required in the method. The method header is given below:

```
public static int position(int n, int digit)
```

Test cases: (1) n : 12345, digit : 3; (2) n : 123, digit : 4; (3) n : 12145, digit : 1;

Expected outputs: (1) position = 3; (2) position = -1; (3) position = 3.

3.7 Write a method extractOddDigits() which extracts the odd digits from a positive number n, and combines the odd digits sequentially into a new number. The new number is returned back to the calling method. If the input number n does not contain any odd digits, then returns -1. For examples, if n=1234567, then 1357 is returned; and if n=28, then –1 is returned. The method header is given below:

```
public static long extractOddDigits(long n)
```

Test cases: (1) n : 12345; (2) n : 54123; (3) n : 246; (4) n : -12 (give an error message)

Expected outputs: (1) oddDigits = 135; (2) oddDigits = 513; (3) oddDigits = -1; (4) oddDigits = Error input!!

Lab 3 : CLASSES & OBJECTS

1. OBJECTIVE

The objective of Lab3 is to practise on processing array of objects.

2. INTRODUCTION

Very often a program needs to process information in an array of objects in several ways. We will practise on writing array of objects in this lab.

Before you start your task, below is a brief introduction to Java syntax and constructs for arrays :

- **Arrays in Java (and C/C++) are indexed from *0* to SIZE-1.**

- **Creating** arrays with 12 elements of integer type that stores the number of days in each month (initialized to 0):

```
int [ ] days ;
days = new int[12] ;     // 2 lines code
```

OR

```
int [ ] days = new int[12];  // one line code
```

- **Assign a value to an array element :**

```
e.g.    days[0] = 31 ;
        days[1] = 28 ;
```

OR

```
int[ ] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
```

- **Accessing array elements:**

```
Ex 1.  sales[0] = 143.50 ;

Ex 2.  if ( sales[23] == 50.0 ) ...

Ex 3.  sales[8] = sales[5] – sales[2] ;

Ex 4.  while ( sales[364] != 0.0 ) {...}

Ex 5.  for ( int i=0 ; i < 365 ; i++ )
           sales[ i ] = 0 ;
```

- **Java stores the size of the array automatically in an instance variable named *length***

e.g.

```
public class PrintingDays
{
  public static void main( String[] args )
  {
    int i;
    int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
    // print the number of days in each month
    for ( i=0 ; i < days.length ; i++ ) // traverse array
      System.out.println( "Month " + (i+1) +
        " has " +  days[i] + " days." );
  }
}
```

```
Program Output
Month 1 has 31 days.
Month 2 has 28 days.
...
Month 12 has 31 days.
```

- **Creating an array of *objects*:**

  e.g. Rectangle[ ] rectArray = new Rectangle[5]; // allocate 5 spaces to hold the elements
  rectArray[0] = new Rectangle(15,10); // assign Rectangle object to the space
  rectArray[1] = new Rectangle(10,20); //
  ……

3. ***Your Tasks for this LAB***

The Colossus Airlines fleet consists of one plane with a seating capacity of 12. It makes one flight daily. In this experiment, you are required to write a seating reservation application program. The problem specification is given below:

A. Write a class *PlaneSeat* that has the following features. Each *PlaneSeat* object should hold a seat identification number (seatId), a marker (assigned) that indicates whether the seat is assigned and the customer number (customerId) of the seat holder. The class diagram is given below:

PlaneSeat
- seatId: int - assigned: boolean - customerId: int
+ PlaneSeat(seat_id: int) + getSeatID(): int + getCustomerID(): int + isOccupied(): boolean + assign(cust_id: int): void + unAssign(): void

where

PlaneSeat() - is the constructor for the class.
getSeatID() – a get method that returns the seat number.
getCustomerID() – a get method that returns the customer number.
isOccupied() – a method that returns a boolean on whether the seat is occupied.
assign() – a method that assigns a seat to a customer.
unAssign() – a method that unassigns a seat.

Implement the class PlaneSeat.

B. Write a class *Plane* that comprises 12 seats. The class should create an array of 12 objects from the class *PlaneSeat*.

The class diagram is given below:

Plane
- seat: PlaneSeat[ ]

- numEmptySeat: int
+ Plane() - sortSeats() : PlaneSeat[ ] + showNumEmptySeats(): void + showEmptySeats(): void + showAssignedSeats(bySeatId : boolean): void + assignSeat(seatId : int, cust_id : int): void + unAssignSeat(seatId : int): void

where

seat – instance variable containing information on the seats in the plane. It is declared as an array of 12 seat objects.
numEmptySeat – instance variable containing information on the number of empty seats.
Plane() – a constructor for the class Plane.
sortSeats() – a method to sort the seats according to ascending order of customerID. A copy of the original seat array is used for sorting instead of the original.
showNumEmptySeats() – a method to display the number of empty seats.
showEmptySeats() – a method to display the list of empty seats.
showAssignedSeat() – a method to display the assigned seats with seat ID and customer ID. If ***bySeatId*** is true, the order will be by seatID, else order is by customerID.
assignSeat() – a method that assigns a customer ID to an empty seat .
unAssignSeat() – a method that unassigns a seat.

Implement the class Plane.

C. Write an application class *PlaneApp* that implements the seating reservation program.

The class *PlaneApp* should be able to support the following:
(1) Show the number of empty seats
(2) Show the list of empty seats
(3) Show the list of customers together with their seat numbers in the order of the seat numbers
(4) Show the list of customers together with their seat numbers in the order of the customer ID
(5) Assign a customer to a seat
(6) Remove a seat assignment

The menu should also contain option (7) (i.e. quit) for terminating the program. After the user selects a particular option, the corrsponding operation will be executed. If the selected option is not (7), then the program shows the menu for user selection again. This application does not need to save data into a file between runs.

Important:
Remember to do all the programming inside the sub-directory *lab6* and name the source codes as **PlaneSeat.java**, **Plane.java** and **PlaneApp.java** and name the compiled codes as **PlaneSeat.class**, **PlaneSeat.class** and **PlaneApp.class**.

Test Data:
Test your application program with the following data:
1. Assign a customer to a seat with SeatID=10, CustomerID = 10001.
2. Assign a customer to a seat with SeatID=12, CustomerID = 10002.
3. Assign a customer to a seat with SeatID=8, CustomerID = 10003.
4. Show the the list of customers together with their seat numbers in the order of the seat numbers.
5. Show the number of empty seats.
6. Show the list of empty seats.
7. Assign (attempt) a customer to a seat with any existing CustomerID, and SeatID. (Should give a warning message!)
8. Remove the seat assignment with SeatID=10.
9. Assign (attempt) a customer to a seat with SeatID = 12.
10. Remove the seat assignment with SeatID=12.
11. Show the list of customers together with their seat numbers in the order of the seat numbers.
12. Show the number of empty seats.

13. Show the list of empty seats.
14. Quit

Expected outputs:

```
(1) Show number of empty seats
(2) Show the list of empty seats
(3) Show the list of seat assignments by seat ID
(4) Show the list of seat assignments by customer ID
(5) Assign a customer to a seat
(6) Remove a seat assignment
(7) Exit

  Enter the number of your choice: 5
Assigning Seat ..
  Please enter SeatID: 10
  Please enter Customer ID: 10001
Seat Assigned!

  Enter the number of your choice: 5
Assigning Seat ..
  Please enter SeatID: 12
  Please enter Customer ID: 10002
Seat Assigned!

  Enter the number of your choice: 5
Assigning Seat ..
  Please enter SeatID: 8
  Please enter Customer ID: 10003
Seat Assigned!

   Enter the number of your choice: 3
The seat assignments are as follow:
SeatID 8 assigned to CustomerID 10003.
SeatID 10 assigned to CustomerID 10001.
SeatID 12 assigned to CustomerID 10002.

   Enter the number of your choice: 4
The seat assignments are as follow:
SeatID 10 assigned to CustomerID 10001.
SeatID 12 assigned to CustomerID 10002.
SeatID 8 assigned to CustomerID 10003.
  Enter the number of your choice: 1
There are 9 empty seats

  Enter the number of your choice: 2
The following seats are empty:
SeatID 1
SeatID 2
SeatID 3
SeatID 4
SeatID 5
SeatID 6
SeatID 7
SeatID 9
SeatID 11

  Enter the number of your choice: 5
Assigning Seat ..
  Please enter SeatID: 8
  Please enter Customer ID: 10004
Seat already assigned to a customer.

  Enter the number of your choice: 6
  Enter SeatID to unassign customer from: 10
Seat Unassigned!

  Enter the number of your choice: 5
```

```
Assigning Seat ..
  Please enter SeatID: 12
  Please enter Customer ID: 10005
Seat already assigned to a customer.

  Enter the number of your choice: 6
  Enter SeatID to unassign customer from: 12
Seat Unassigned!

  Enter the number of your choice: 3
The seat assignments are as follow:
SeatID 8 assigned to CustomerID 10003.

   Enter the number of your choice: 1
There are 11 empty seats

  Enter the number of your choice: 2
The following seats are empty:
SeatID 1
SeatID 2
SeatID 3
SeatID 4
SeatID 5
SeatID 6
SeatID 7
SeatID 9
SeatID 10
SeatID 11
SeatID 12

  Enter the number of your choice: 7
```

Lab 4 : INHERITANCE & POLYMORPHISM

1. OBJECTIVE

The objective of Lab 4 is to practise on class inheritance and polymorphism.

Notes:
It is intended to be a short lab so that teams can get together to discuss/work on the assignment project which would have been published.

2. INTRODUCTION

Inheritance and Polymorphism are 2 important concepts in Object-Oriented Design and Programming. In this lab, you will get to learn more about the concepts in action. In one the tasks, you will need to decide whether ***concrete class, abstract class or interface*** is appropriate for the task required. In addition, you will also need to decide on the appropriate ***'is a'*** **or** ***'has a'*** relationship (inheritance/generalization VS delegation/object composition).

3. *Your Tasks for this LAB*

3.1 Polymorphic Sorting

Hints :
This is lab does not need a lot of coding, it is just a matter of where needs to be changed or added additional codes.

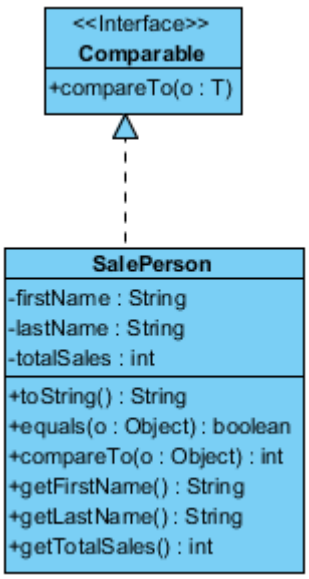
The file *Sorting.java* contains the Sorting class (in attachment). This class implements both the selection sort and the insertion sort algorithms for sorting any array of *Comparable* objects in ascending order. In this exercise, you will use the Sorting class to sort several different types of objects. (For details on the *Compareble* interface, refer to Java API doc http://docs.oracle.com/javase/7/docs/api/)

1. The file Numbers.java (in attachment) reads in an array of integers, invokes the selection sort algorithm to sort them, and then prints the sorted array. Save Sorting.java and Numbers.java to your directory. Numbers.java won't compile in its current form. Study it to see if you can figure out why.

2. Try to compile Numbers.java and see what the error message is. The problem involves the difference between primitive data and objects. Change the program so it will work correctly (note: you don't need to make many changes - the *autoboxing* feature of Java 1.5 (or higher) will take care of most conversions from int to Integer). You are to do research in the internet and understand better *autoboxing*.

3. Write a program Strings.java, similar to Numbers.java, that reads in an array of String objects and sorts them. You may just copy and edit Numbers.java.

4. Modify the insertionSort algorithm so that it sorts in descending order rather than ascending order. Change Numbers.java and Strings.java to call insertionSort rather than selectionSort. Run both to make sure the sorting is correct.



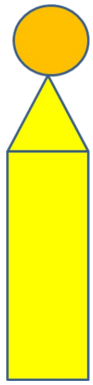
5. The class diagram on the right defines the SalePerson class that represents a sale person. The sale person has a first name, last name, and a total number of sales (an int).
   - The *toString* method will return the name of the sale person and total sales in the formal :
     <lastName> , <firstName> : <totalSales>
   - The *equals* method will check whether the first and last names of Object are the same as the current sale person.
   - The *compareTo* method make the comparison based on total sales; that is, return a negative number if the executing object has total sales less than the other object and return a positive number if the sales are greater. ***Use the name of the sales person's last name to break a tie (in ascending alphabetical order).***
   - ***Create and Write the SalePerson class***

6. The file WeeklySales.java (in attachment) contains a driver for testing the compareTo method and the sorting . Compile and run it. Make sure your compareTo method is correct. The sales staff should be listed in **the order of sales from most to least**. If the sale staffs have the same number of sales, they are listed in ascending alphabetical order of their last names.

3.2 Calculate Surface Area of a Figure.

By using the concepts of inheritance and **polymorphism**, you are required to design a program that calculate the total surface area of a figure. The following are the requirements an constraints :

- You should have a Class/Interface called *Shape and decide its appropriate attributes and behaviours*
- You should have basic shapes like *Square*, *Rectangle*, *Circle* and *Triangle*.
- The program will request the user to :
  - enter the total number of shapes
  - choose the shape and enter the required dimension/s for the selected shape
  - choose the type of calculation (for now, we will just calculate *Area,* with future plan to calculate *Volume* as well).
- The calculation/s should be done upon user's request and *NOT when dimensions are entered*.



1. For a start, use the 2-D figure on the right to verify your program. The figure consists of a Circle (radius=10), a Triangle (height=25, base =20) and a Rectangle (length=50, breadth = 20) . ***Calculate the total area of the 2- D figure***.
   (You will create an Application class **Shape2DApp.java** for this purpose)

2. We will now expand and extend your design to cater to 3-D figures. Imagine the figure on the right is turn into a 3-D figure – Circle becomes Sphere, Triangle becomes a square-based Pyramid and the Rectangle is a cubiod. ***Calculate the total surface area of the 3- D figure***. [Note : You need to think whether 'is a' or 'has a' relationship is more appropriate and relevant for between 2D and 3D shapes.
   (You will create an Application class **Shape3DApp.java** for this purpose)

3. We will include more Shapes. The square-based Pyramid will be replaced with a Cone and the Cubiod is replaced with a Cylinder. ***Calculate the total surface area of the new 3- D figure***.
   (You will **reuse** the Application class **Shape3DApp.java** with appropriate selection)

Continue on next page…..
IMPORTANT :

For this task, you are encouraged to work in pair. At the end of this task, you or the pair will demonstrate to your lab sup the working of your program.

Note :
The solutions for 3D Figures implementation can be many, simple or complex (like considering the overlapping regions). But the main objectives is to considered the idea and concept of polymorphism in the design and implementation. To build an application like AutoCAD, it can be just a discussion with your lab supervisor.

Lab 5 : Modern Java Programming- Library Management System

1. OBJECTIVE

By completing this lab exercise, students will:

1. **Apply Object-Oriented Design Principles:**
   - Design and implement a system using classes, interfaces, and relationships between objects.
2. **Understand and Use Generics in Java:**
   - Create a generic Library class to ensure type safety and reuse code across different types.
3. **Explore Java Collections Framework:**
   - Utilize core data structures like List, Set, and Map to manage library data effectively.
   - Work with immutable collections for predefined categories.
4. **Leverage the Streams API:**
   - Perform filtering, sorting, and other data processing tasks on collections using Streams.
5. **Write Functional and Concise Code with Lambda Expressions:**
   - Use lambdas for operations such as sorting, filtering, and iterating over collections.
6. **Master Iteration with Enhanced For Loops:**
   - Iterate over collections and arrays using the enhanced for loop for better readability and simplicity.
7. **Understand Default Methods in Interfaces:**
   - Implement default methods to provide shared functionality across multiple classes.
8. **Implement Enhanced Switch Statements:**
   - Write cleaner and more powerful conditional logic using modern switch expressions.
9. **Simulate Real-World Scenarios:**
   - Apply programming concepts to realistic use cases like managing books, and borrowers, , and recommendations in a library system.
10. **Develop Problem-Solving and Debugging Skills:**
    - Test various scenarios, validate outputs, and troubleshoot potential issues in the system.

2. INTRODUCTION

In this lab exercise, you will design and implement a Library Management System using modern Java programming concepts. This exercise serves as a practical application of Object-Oriented Design and Programming (OODP) principles, combined with advanced features.

The Library Management System will manage books, authors, and borrowers, simulating a real-world scenario. You will utilize essential Java features like Generics, Collections, Streams, and Lambda Expressions to create an efficient, maintainable, and extensible application. Additionally, modern constructs such as the enhanced for loop, default methods in interfaces, and the enhanced switch statement will be employed to simplify the design and improve readability.

By working through this exercise, you will not only reinforce your understanding of key programming constructs but also learn to apply them in solving real-world problems, preparing you for professional software development scenarios.

3. *Your Tasks for this LAB*

Library Setup:

- Create a Library class that uses generics.
- Add methods to add, remove, and retrieve items.

Collections Usage:

- Create a list of books, a set of genres, and a map of authors with their books.

Implement Filtering:

- Filter books by genre and author using Streams and Lambdas.

Display Data:

- Use an enhanced for loop to display all books.

Search Functionality:

- Use the Searchable interface to search for books containing a specific keyword in their title or description.

Switch-Based Recommendations:

- Implement the recommendBook method to provide recommendations based on categories.

Test Cases:

1. Adding Books to the Library

- **Input:**
  - Book: {"title": "1984", "author": "George Orwell", "genre": "Fiction", "publicationYear": 1949}
  - Book: {"title": "A Brief History of Time", "author": "Stephen Hawking", "genre": "Science", "publicationYear": 1988}
- **Expected Output:**
  - Library contains 2 books.

2. Filtering Books by Genre

- **Input:**
  - Genre: "Fiction"
- **Expected Output:**
  - List of books:
    - {"title": "1984", "author": "George Orwell", "genre": "Fiction", "publicationYear": 1949}

3. Filtering Books by Author

- **Input:**
  - Author: "Stephen Hawking"
- **Expected Output:**
  - List of books:
    - {"title": "A Brief History of Time", "author": "Stephen Hawking", "genre": "Science", "publicationYear": 1988}

4. Searching Books by Keyword

- **Input:**

- Keyword: "Time"

- **Expected Output:**
  - List of books:
    - {"title": "A Brief History of Time", "author": "Stephen Hawking", "genre": "Science", "publicationYear": 1988}

5. Add the following book to library:

"A Beautiful Mind" by Sylvia Nasar (Biography, 1998)

Sorting the Books by Title

- **Expected Output:**

  Sorted Order by Title:

  "1984"

  "A Beautiful Mind"

  "A Brief History of Time"

6. Providing Recommendations

- **Input:**
  - Category: "Science"
- **Expected Output:**
  - Recommendation: "Try 'A Brief History of Time' by Stephen Hawking."

7. Borrower Operations

- **Input:**
  - Borrower: {"name": "Alice", "borrowedBooks": []}
  - Borrow a book: "1984"
- **Expected Output:**
  - Borrower details:
    - {"name": "Alice", "borrowedBooks": ["1984"]}

8. Returning a Book

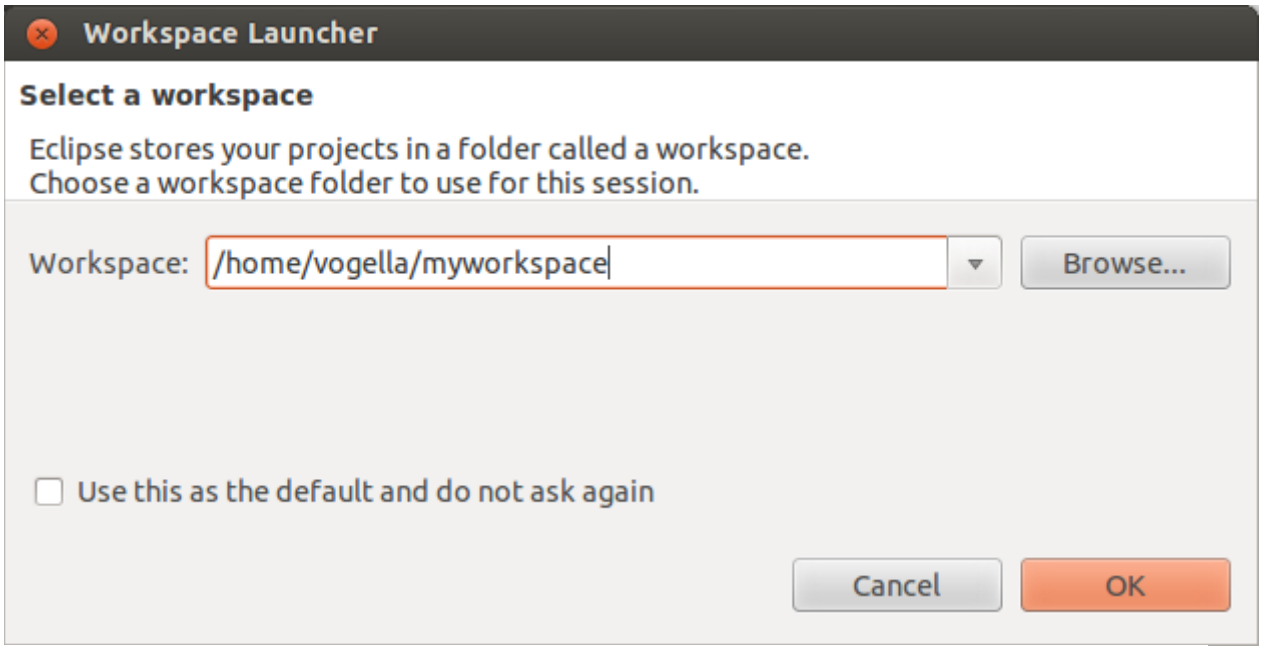
- **Input:**
  - Borrower: {"name": "Alice", "borrowedBooks": ["1984"]}
  - Return book: "1984"
- **Expected Output:**
  - Borrower details:
    - {"name": "Alice", "borrowedBooks": []}

APPENDIX A : Eclipse IDE

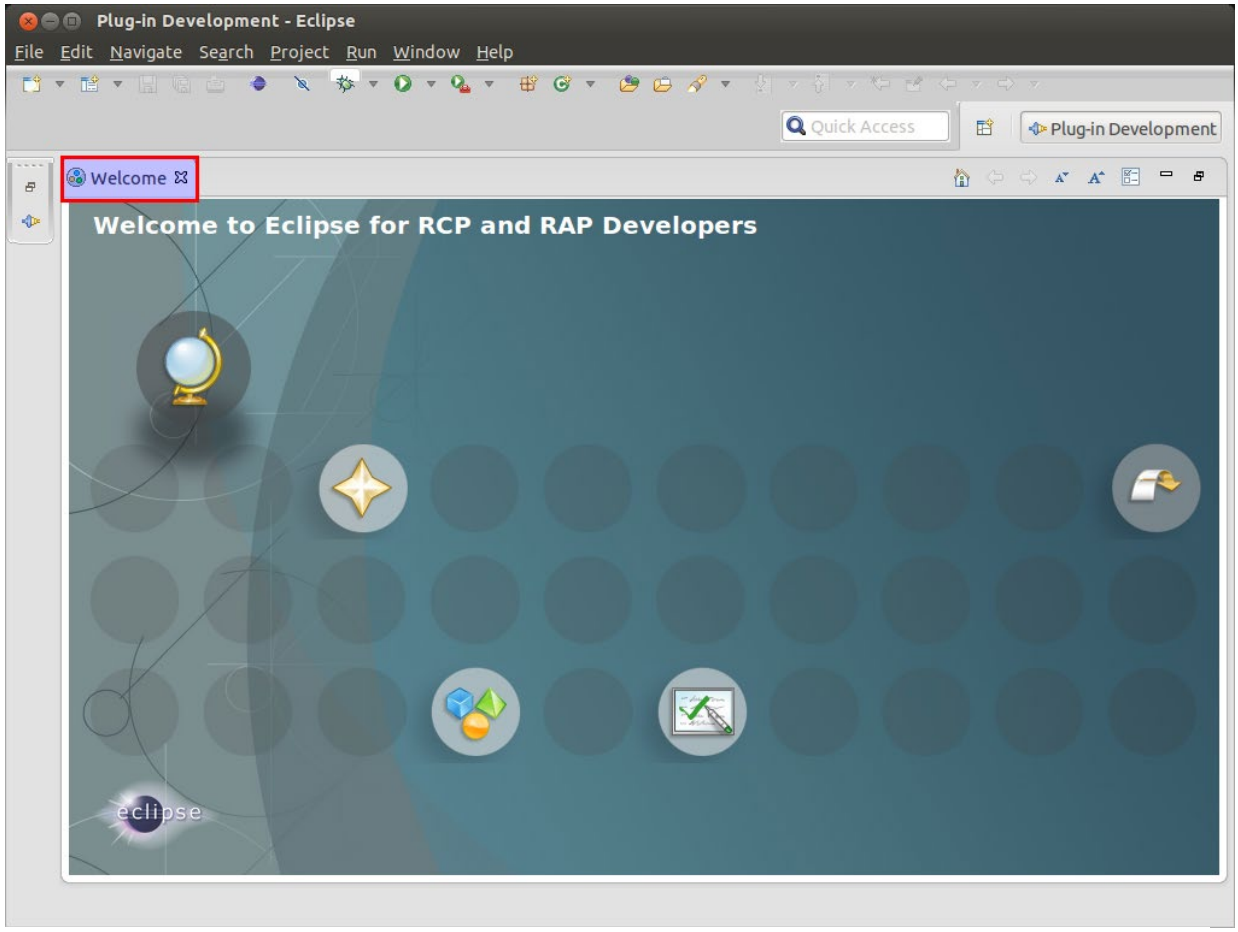
1. Starting Eclipse

To start Eclipse double-click on the file eclipse.exe (Microsoft Windows) on the Desktop. If you are unable to locate it, go to "Program" - you may want to create a shortcut for your convienience.

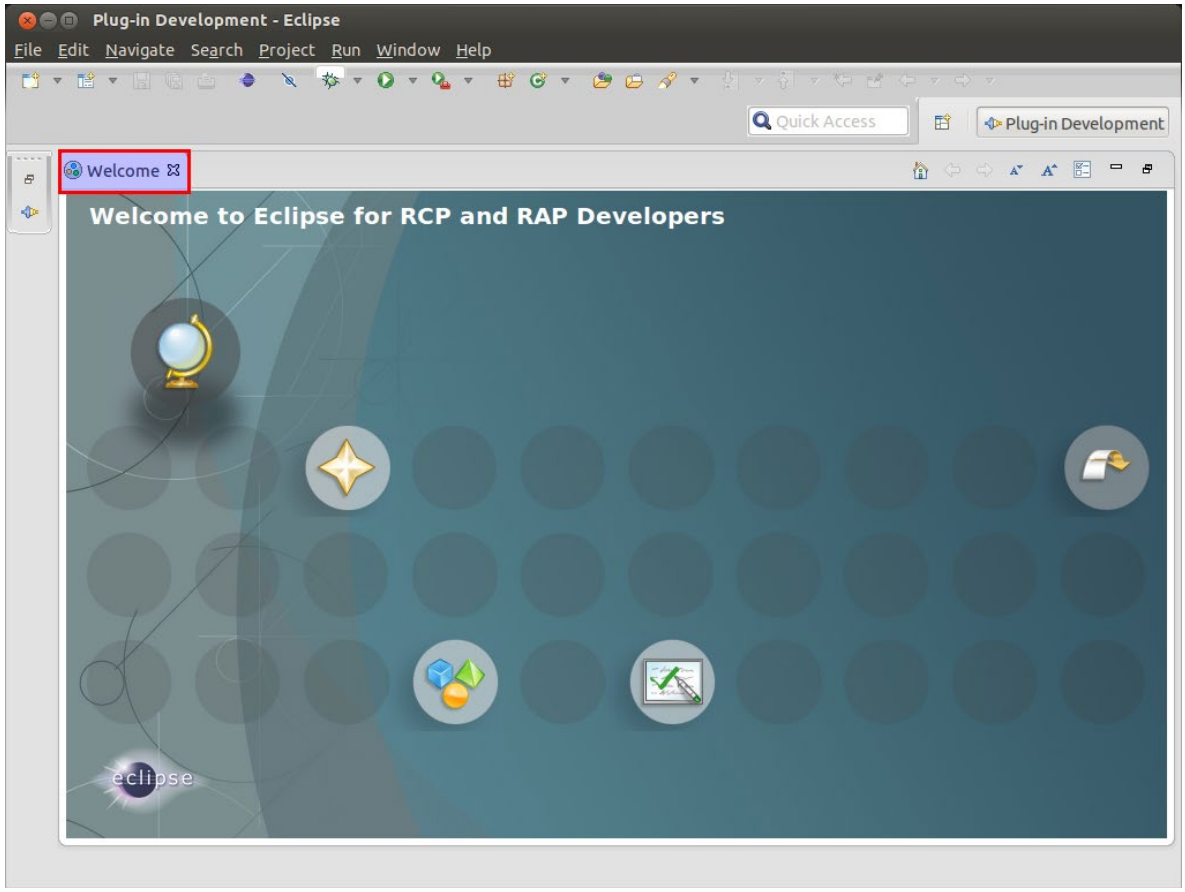
The system will prompt you for a *workspace*. The *workspace* is the place in which you work. Select an empty directory and press the *OK* button.



Eclipse will start and show the Welcome page. Close the welcome page by pressing the *X* beside *Welcome*.



After you closed the welcome screen you should see a screen similar to the following.



2. Appearance

The appearance of Eclipse can be changed. By default Eclipse ships with a few themes but you can also extend Eclipse with new themes.

To change the appearance, select from the menu Window → Preferences → General → Appearance

The <guilable>Theme</guilable> selection allows you to change the appearance of your Eclipse IDE. Please note that you need to restart Eclipse to apply a new styling correctly.

3. Eclipse user interface overview

Eclipse provides *Perspectives*, *Views* and *Editors*. *Views* and *Editors* are grouped into *Perspectives*.

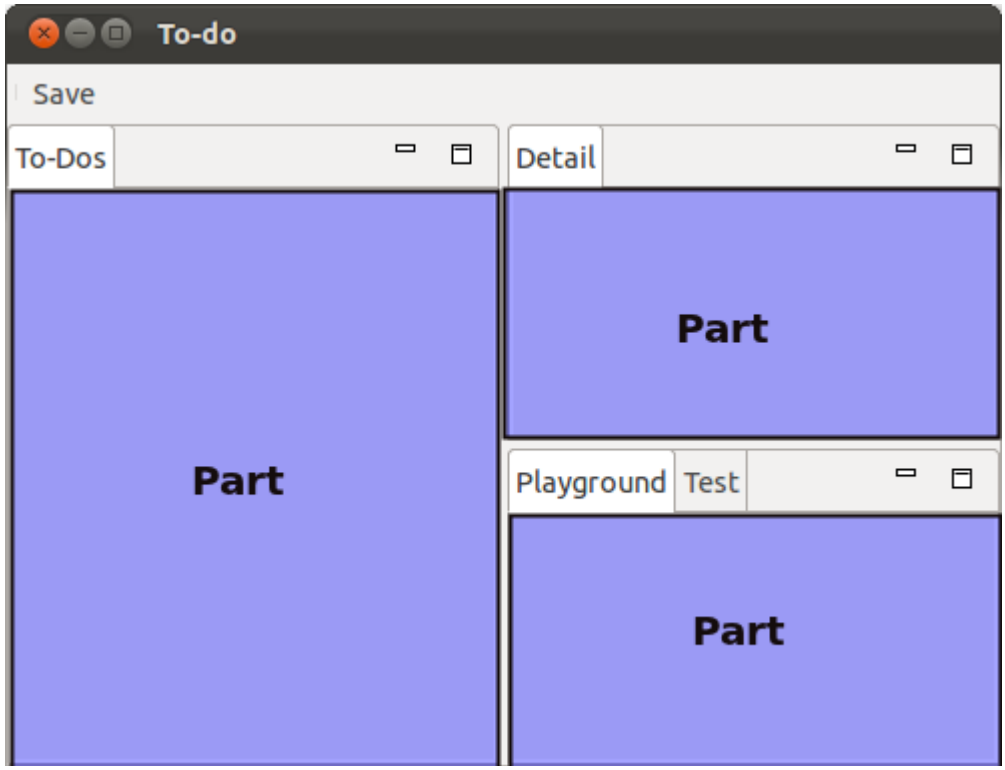
3.1. Workspace

The *workspace* is the physical location (file path) you are working in. Your projects, source files, images and other artifacts can be stored and saved in your workspace but you can also refer to external resources, e.g. projects, in your *workspace*.

You can choose the workspace during startup of Eclipse or via the menu (File → Switch Workspace → Others) .

3.2. Parts

Parts are user interface components which allow you to navigate and modify data. *Parts* are typically divided into *Views* and *Editors*.



The distinction into *Views* and *Editors* is primarily not based on technical differences, but on a different concept of using and arranging these *Parts*.

A *View* is typically used to work on a set of data, which might be a hierarchical structure. If data is changed via the *View*, this change is typically directly applied to the underlying data structure. A *View* sometimes allows us to open an *Editor* for a selected set of the data.

An example for a *View* is the *Java Package Explorer*, which allow you browse the files of Eclipse Projects. If you choose to change data in the Package Explorer, e.g. if you rename a file, the file name is directly changed on the file system.

Editors are typically used to modify a single data element, e.g. a file or a data object. To apply the changes made in an editor to the data structure, the user has to explicitly save the editor content.

Editors were traditionally placed in a certain area, called the *editor area*.

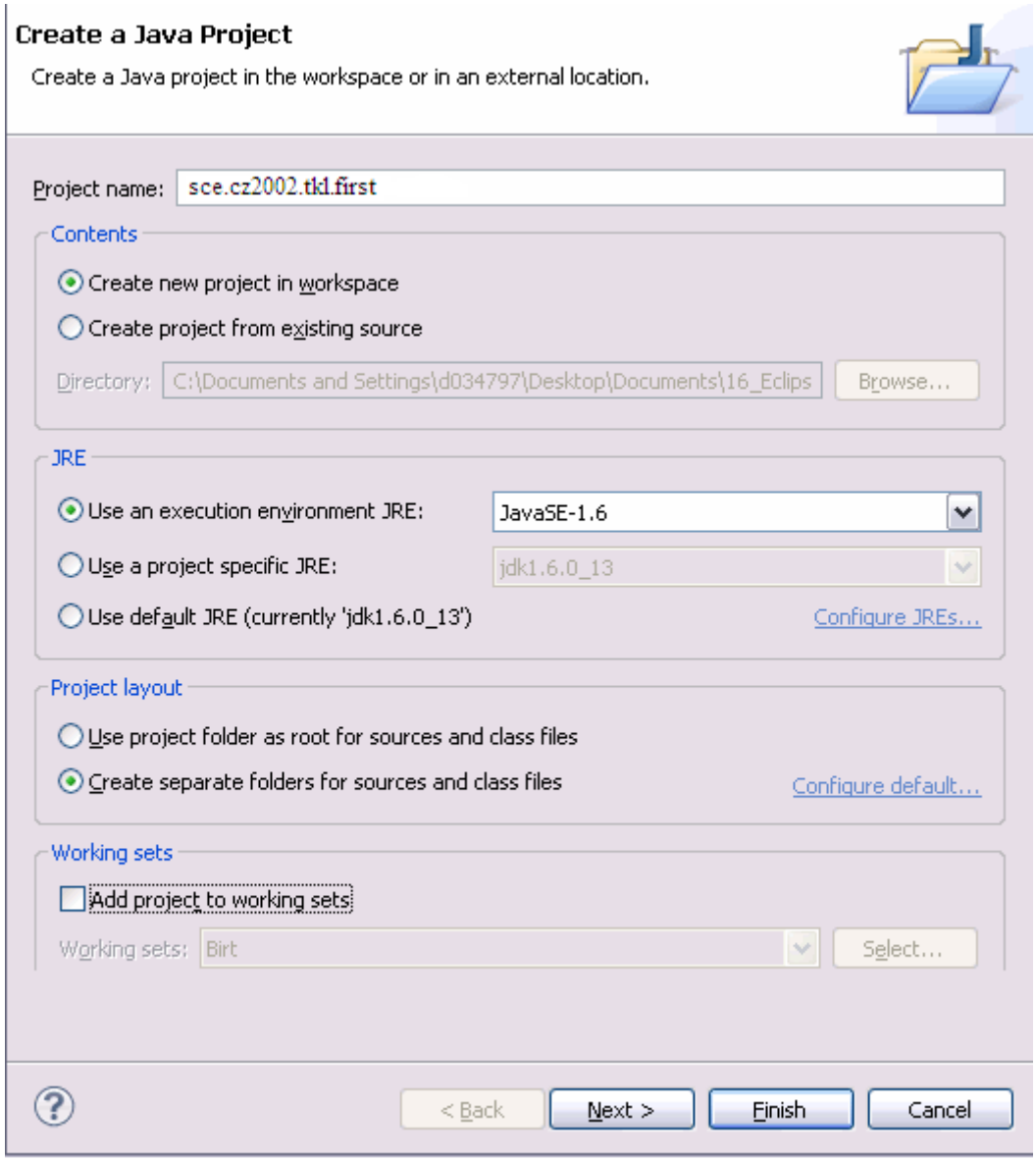
For example the Java Editor is used to modify Java source files. Changes to the source file are applied once the user selects the *Save* command.

APPENDIX B : Create your first Java program

The following describes how to create a minimal Java program using Eclipse. It is tradition in the programming world to create a small program which writes "Hello World" to the console. We will adapt this tradition and will write "Hello Eclipse!" to the console.

1. Create project

Select from the menu File → New → Java project. Enter ***sce.<coursecode>.<your initials>.first*** as the project name, replace <*coursecode*> with **ce2002** or **cz2002** and <*your initial*> as your name initial or others. Select the flag "Create separate folders for sources and class files".

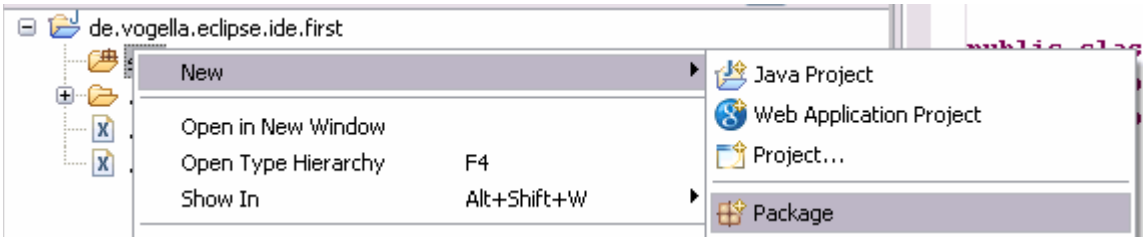


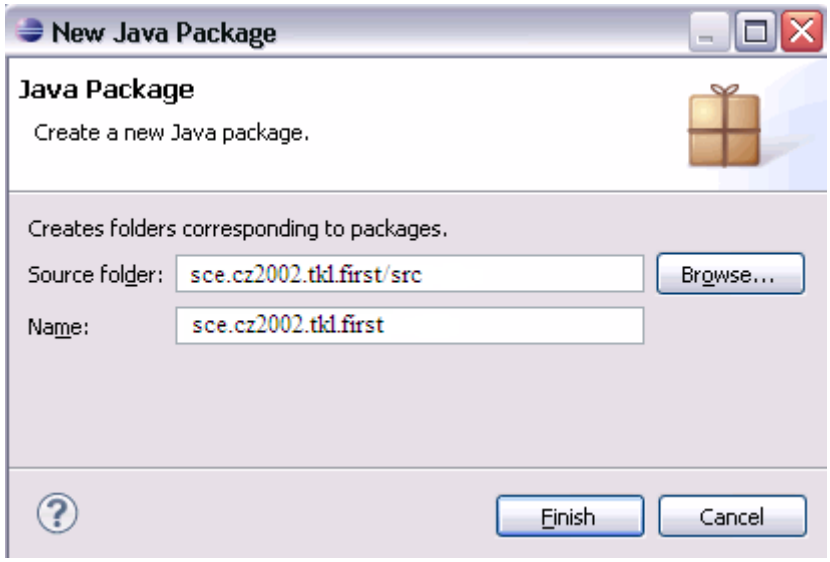
Press finish to create the project. A new project is created and displayed as a folder. Open the ***sce.<coursecode>.<your initials>.first*** folder and explore the content of this folder.

2. Create package

Create a new package. A good convention is to use the same name for the top package and the project. Create therefore the package ***sce.<coursecode>.<your initials>.first***.

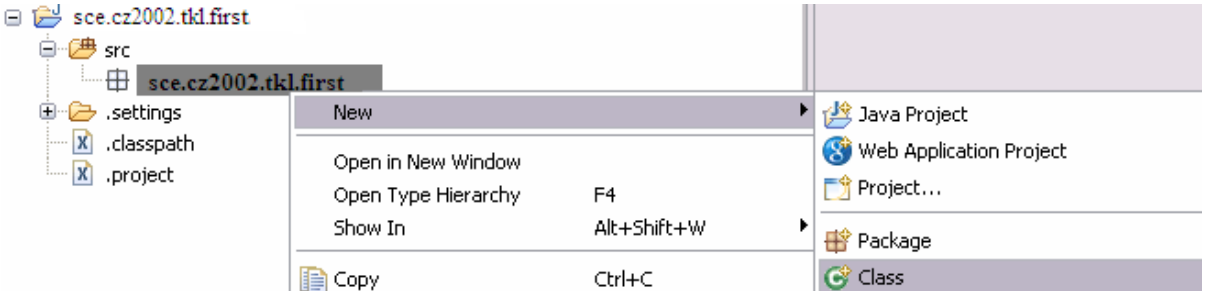
Select the folder src, right click on it and select New → Package.



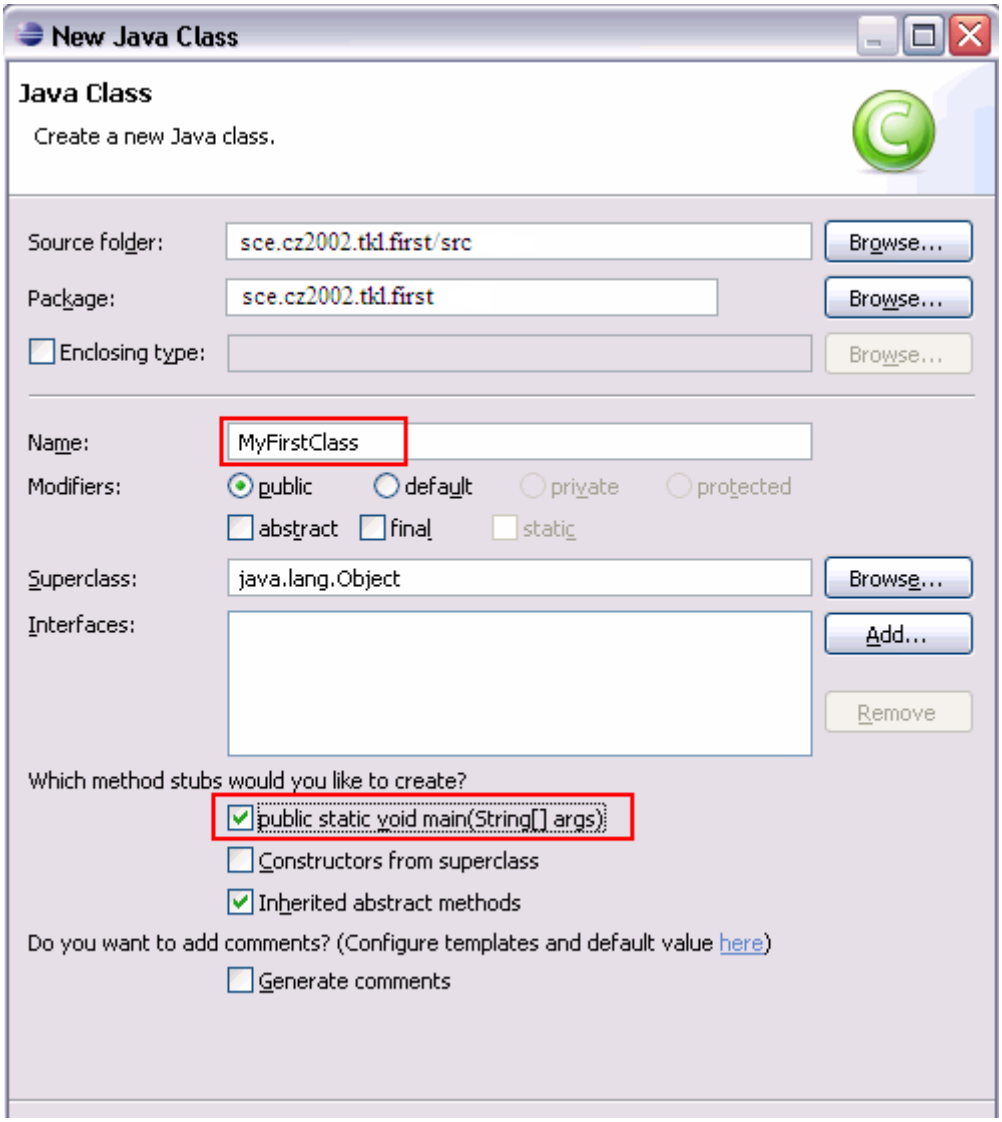


3. Create Java class

We will now create a Java class. Right click on your package and select New → Class.



Enter MyFirstClass as the class name and select the flag "public static void main (String[] args)".

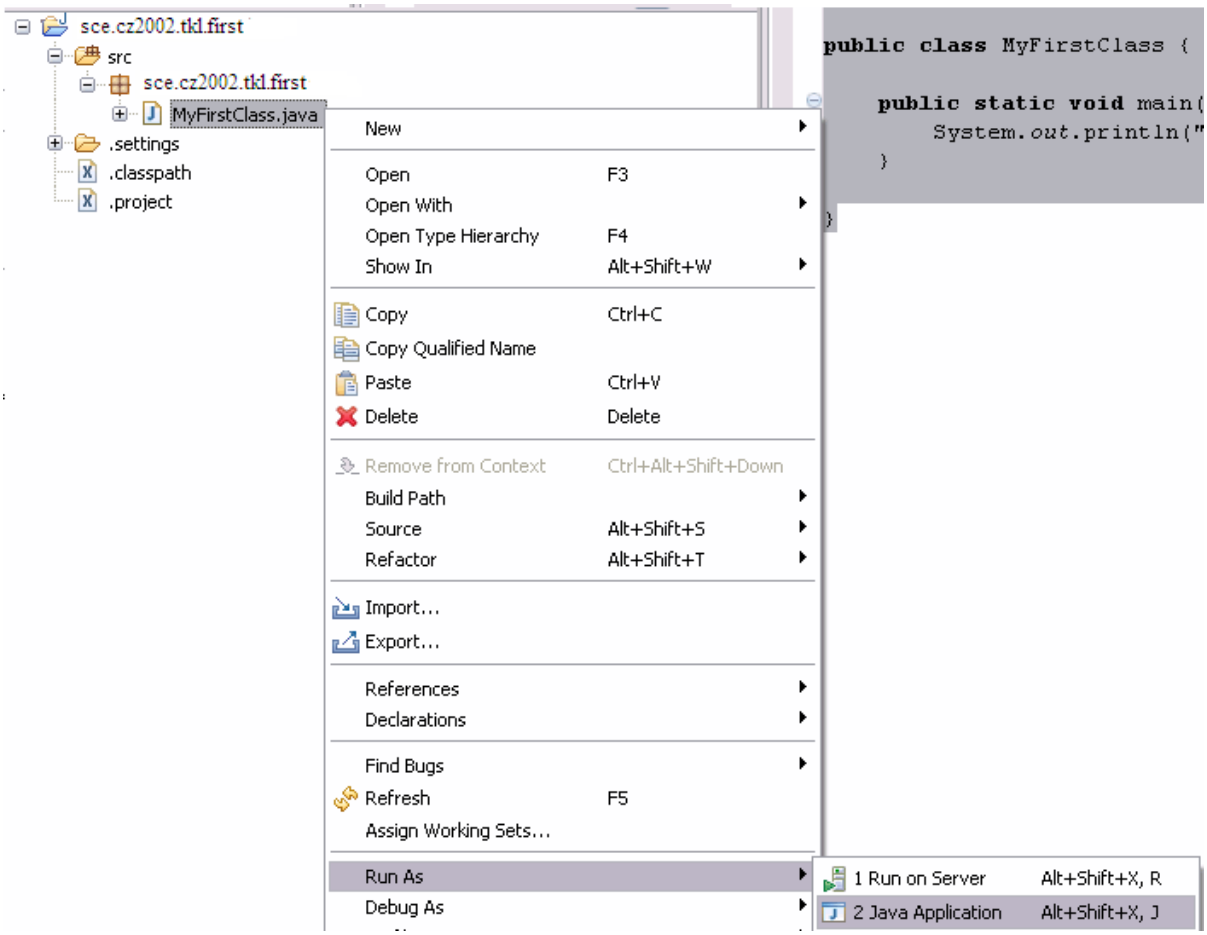


This creates a new file and opens an Editor to edit the source code of this file. Write the following code.

```
package sce.cz2002.tkl.first;

public class MyFirstClass {

        public static void main(String[] args) {

                System.out.println("Hello Eclipse!");

        }

}
```

4. Run your project in Eclipse

Now run your code. Right click on your Java class and select Run-as → Java application.



Finished! You should see the output in the console.