

## 48024 Object Oriented Design

### **Self-Study Module: Java and BlueJ refresher**

This module covers the basics of programming in Java using the BlueJ development environment. A series of exercises covers the following topics: object oriented programming in BlueJ; the concept of code reuse by composition; debugging in BlueJ; and Collection classes in Java.

You will need to download source code for each exercise from UTSONline Course documents/Self-Study Modules.

#### Contents:

Exercise 1 - BlueJ and Class definition .....	2
Exercise 2 - Debugging in BlueJ .....	5
Exercise 3 - Java Collections .....	8

**Exercise 1 - BlueJ and Class definition**

Objectives:

- To refresh object oriented programming in Java using BlueJ.
- To identify and modify the main elements of a class.
- To introduce the concept of *code reuse by composition*.

Resources:

**BlueJ Documentation:**

- BlueJ Tutorial (Downloadable from UTSONline OOD Course Documents).
- <http://www.bluej.org/doc/documentation.html>

**Recommended reading:**

- Wu, Thomas. An Introduction to OOP with Java. Chapter 4 - Defining Instantiable Classes.

**Java Code:**

- BlueJ Project: shapes

**Tasks**

The shapes project draws geometric shapes of different size, colour and initial location in a small window. Once you open the project you will find three Java classes each one corresponding to three different geometric shapes (Circle, Triangle and Square).

As a preliminary exercise go to the BlueJ Tutorial and review the structure of each one of the classes in the shapes project to learn how the original shapes are constructed. For example, edit the code of the class Square and identify the definition of its attributes (colour, size, position, etc.) and methods that allow changing the characteristics of a Square object. **Don't continue until you understand the basic structure and behaviour of these classes!**

**Task 1 – Drawing a house shape**

Using the classes included in the shapes project, you are required to create a new class called House which draws a shape similar to the following figure,

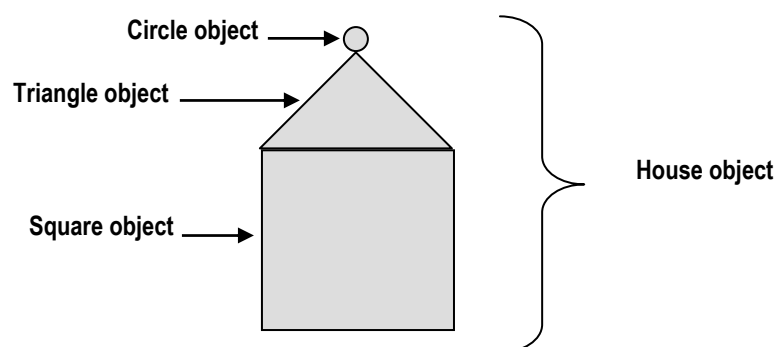


Fig. 1 - House Object

**Task 2 – Adding changeColour method**

**Self-Study Module: Java and BlueJ refresher**

Add to the House class you just created a public method `changeColour(String newColour)` that changes the colour of the entire house to the colour equal to the parameter `newColour` (e.g. "yellow", "blue", "red", etc.).

***Before starting, ask yourself...***

- do I clearly identify classes, objects, methods and attributes in Java?
- how do I create a new class in BlueJ?
- how can I create and use objects from class *A* in *B* using Java?
- once I create an object in a class, how can I change its characteristics?

**How do I do this? - First Approach**

One first approach to drawing the house is to create in the constructor of the class House (the new class) one object of the class Square, one of the class Triangle and one of the class Circle and then change their original positions in such a way that the triangle is placed above the square and the circle is placed on top of the triangle. Note that the user has no opportunity to choose the colour or the size of the house.

Following are some hints that can help you implement this approach:

1. Calculate the dimensions of the three shapes according to their role in the house.
2. Modify the constructors of the classes Circle, Square and Triangle to adjust the correct values of the attribute to the ones you calculated. For instance, if you want the house to be of size 50 (pixels) you may have to change the value of the attribute `size` in the constructor of the class Square by changing the line "`size = 30;`" to "`size = 50;`".
3. Define a new class House that creates three objects of the classes Circle, Triangle and Square respectively.
4. Use the public methods `changeSize`, `changeColor`, `moveHorizontal` and `moveVertical` of each shape object to fix it as part of the house.
5. Make visible each one of the objects using its `makeVisible` method.
6. To change the colour of the house call the `changeColor` method of each one of the shape objects that form the house.

**Second Approach (recommended)**

A more challenging and interesting implementation is that one where the user is asked to input the top-left corner co-ordinates ( $x, y$ ) of the rectangle of the house, the length of the base and the colour of the house that the class House then uses to create a house shape with all these characteristics at once. In this case, the size, colour and initial position of the house are not hard-coded but are defined by the user. This means you may need to modify the original classes of Shapes to be able to use constructors that accept input parameters.

Suggested tasks include:

1. Write a parameterised constructor for each of the three classes: Square, Triangle and Circle, which would permit the creation of shapes according to the characteristics chosen by the user (colour, size, etc.). These new constructors will have as input parameters the characteristics of the house mentioned before.
2. Create a constructor in the new class House that receives as input the colour, the position and size of the base of the house. Anytime you try to create a new House object, BlueJ will ask for the values of these input data.
3. Calculate the required ( $x, y$ ) coordinates of the position of the Square, the Triangle and the Circle that form the house according to the top-left corner co-ordinates of the house given as input. This is an important task that might make you think for a while since the reference point of each shape on the window varies from one shape to the other. Using the given size of the base of the house (the square size), you can obtain the triangle's base and height. Size the circle according to your own taste.
4. Create a Square, a Triangle and a Circle object inside the new House class passing all the calculated values as arguments to the corresponding constructor.
5. Follow the hints 5 and 6 described in the first approach.

**Final notes**

Initially, it is not necessary to get the input from a user interface since BlueJ allows you to feed the values into the parameters when the object is about to be created. However, as a further exercise, you can use the JDialog class of the Java javax.swing package for obtaining the values from the user. You may like to be kind to the user and use a JColorChooser object to select the colour as well.

Don't worry if you spend too much time working on this laboratory since these two first exercises will refresh your memory with what you learned in OOP – and even more!

**Self-Study Module: Java and BlueJ refresher**

**Exercise 2 - Debugging in BlueJ**

Objectives:

- To understand the process and advantages of debugging in the development of programming applications.
- To develop skill in the use of the BlueJ Debugger tool.

Resources:

***BlueJ Documentation:***

- BlueJ Debugger Tutorial
- <http://www.bluej.org/doc/documentation.html>

***Java Code:***

- ShapesWithBugs BlueJ Project
- JavaBook Java Library

**Tasks**

Debugging is the activity in which logical (not syntax) errors in a software unit (class, method) are found and corrected. These errors are termed *bugs*. A careful manual walkthrough of the code known as a code walkthrough or static code inspection is an important review activity and can pick up many errors. However other kinds of errors require dynamic code execution and that is when a debugging tool is very useful. Fortunately nowadays almost every programming environment offers a debugging tool (debugger) that facilitates this important task in the developer's journey. In this laboratory you are going to use the BlueJ debugger to find and correct the errors in the shape project.

**Task 1 – Learning how to use the BlueJ Debugger tool**

The first part of this laboratory consists of following the Debugging Tutorial to become familiar with what can be done using the BlueJ Debugger tool that will help you with your programming activity. The basic tasks of the debugger include:

- Setting breakpoints
- Stepping through the code
- Inspecting variables
- Halt and Terminate

Read the Debugging Tutorial and follow the steps described in the exercise; you will need what you learn to complete the second part of this laboratory exercise.

**Self-Study Module: Java and BlueJ refresher****Task 2 – Making the program work right!**

The ShapesWithBugs program is a modification of the original Shapes program (see Figure 1). The new class Pattern creates and draws a Square object and lets the user perform the following actions over the Square:

- `changeColour()`. Changes the colour of the square once the user selects a colour from a given list.
- `moveShape()`. The square is moved in such way that it describes a squared path, advancing with a clockwise direction and finally returning to its initial position. The square must keep the colour it has before it starts moving.

However, the original programmer of this new class did not understand the requirements very well and the results are not the expected ones. Although the code seems to be fine – there is no error after compiling – the application does not work properly. There must be some runtime errors that can be very difficult to discover and correct. Fortunately, these errors can be “attacked” by the effective use of a debugger tool.

Your task consists of debugging the application –using the BlueJ debugger tool and the skills you learned on the first part of the laboratory– so you can discover the programming bugs and then make the proper corrections to the code.

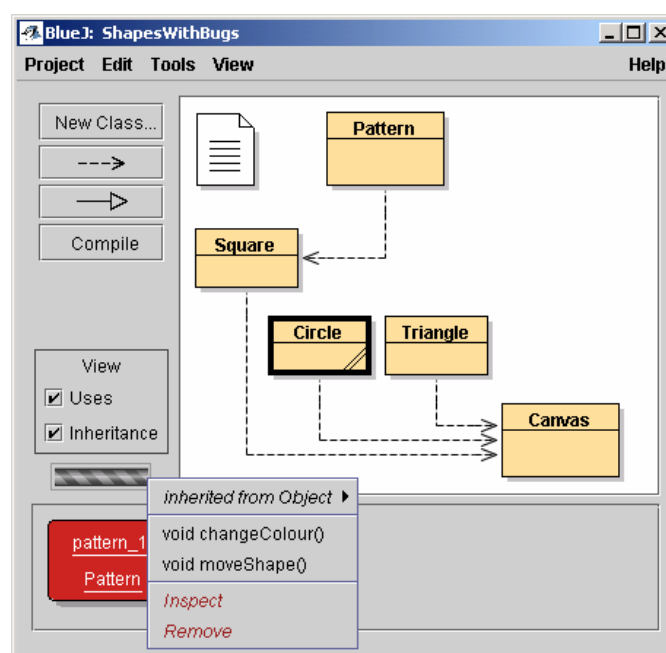


Fig. 1 – ShapesWithBugs Program

Some hints you can take into account to complete this laboratory are:

- In order to run the application successfully, you need to add the Javabook Java library to your BlueJ environment (follow Appendix C).

### Self-Study Module: Java and BlueJ refresher

- Create a new Pattern object and examine carefully how the program is behaving at the moment (call the methods `changeColour()` and `moveShape()`).
- Go through the code and add the breakpoints you consider necessary. Place them in the lines of code where you want to start debugging, i.e. where you need to check carefully the values of local and instance variables.
- In every piece of code you are analysing pay great attention to the value of local and instance variables the debugger window reports. Try to identify the role of each variable in your application; this will give you more control and understanding of the process that is taking place. Furthermore, once you learn the importance of identifying the variables (especially the reference to objects) you should be able to quickly detect and correct some common runtime errors such as the Java `NullPointerException`<sup>1</sup>.
- Be aware that the errors can be in any of the classes included in the project.
- Use the “Step”, “Step Into” and “Continue” buttons to navigate through the code.

### Review Questions

1. What do we use the “Step” button of the BlueJ Debugger tool for?
2. In which cases would you recommend the use of the “Step Into” button?
3. Are we able to see the value of all the variables (instance, local and class) involved in a process while debugging?

---

<sup>1</sup> For more information on Java Exceptions handling read Wu’s book section 11.4

**Exercise 3 - Java Collections**

Objectives:

- To gain skills in the creation and manipulation of a collection of objects using arrays and flexible-size collections in Java.
- To strengthen the use and understanding of basic concepts such as instance, reference, attribute and method of a class in a Java program.

Resources:

*Recommended reading:*

- Wu, Thomas. An Introduction to OOP with Java. Chapter 9: Arrays.
- Schildt Chapter 3 (Arrays section), and Chapter 15 (ArrayList section)

*Source Code:*

- AddressBookLab3 Project<sup>2</sup>

**Task 1 - Introducing Java arrays (fixed-size collection)**

A certain company has developed an address book system to keep information of all of its employees. The AddressBook class maintains a collection of Employee objects using an array of objects. The following are the declaration and creation of the attribute entry which is the array that keeps the collection of a maximum of 10 Employee objects.

Employee [] entry;	→	Declaration of an array of Employee objects
entry = new Employee [10];	→	Creation of the array which reserves space in memory for 10 Employee objects. When the array is created it doesn't contain any reference to Employee objects therefore the ten positions in the array are pointing to null.
Employee [] entry;	→	Declaration of an array of Employee objects
entry[3] = Employee1;	→	Adding employee1 to entry in the fourth position. Note that the index of the array is equal to 3 not 4, why?

Figure 1 shows graphically how the entry array would look like after executing the sequence of lines of code above.

---

<sup>2</sup> All the given classes are based on the Wu's AddressBook sample project.

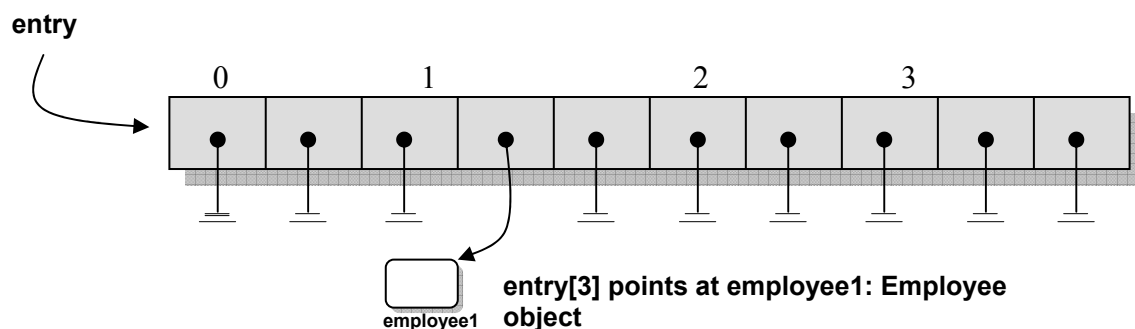


Fig. 1 – Collection of Employee objects

The AddressBook class contains three public methods to manage the information inside the array, namely:

- `add(Employee newEmployee)`: adds the input Employee object to the collection.
- `delete(String searchName)`: deletes the Employee object with the input name.
- `findIndex(String searchName)`: searches for the Employee object which name matches with the one passed as a parameter and returns the index of the array where the object is located. If no Employee is found then minus one (-1) is returned.

### Introducing Java ArrayList (flexible-size collection)

Besides the name, age and gender, an Employee object also has as part of its data members a collection of the names of the employees that are under his/her supervision. The attribute `incharge` is a Java **ArrayList** object that stores String objects corresponding to the names of employees that are already included in the address book and are under the supervision of the current Employee object (see Figure 2). Every employee in the company can be the boss of 0 or more employees, there's actually no restriction; that's why an ArrayList object is used. It is possible for an employee to have more than one boss. The name of an employee is unique; there are no employees with the same name.

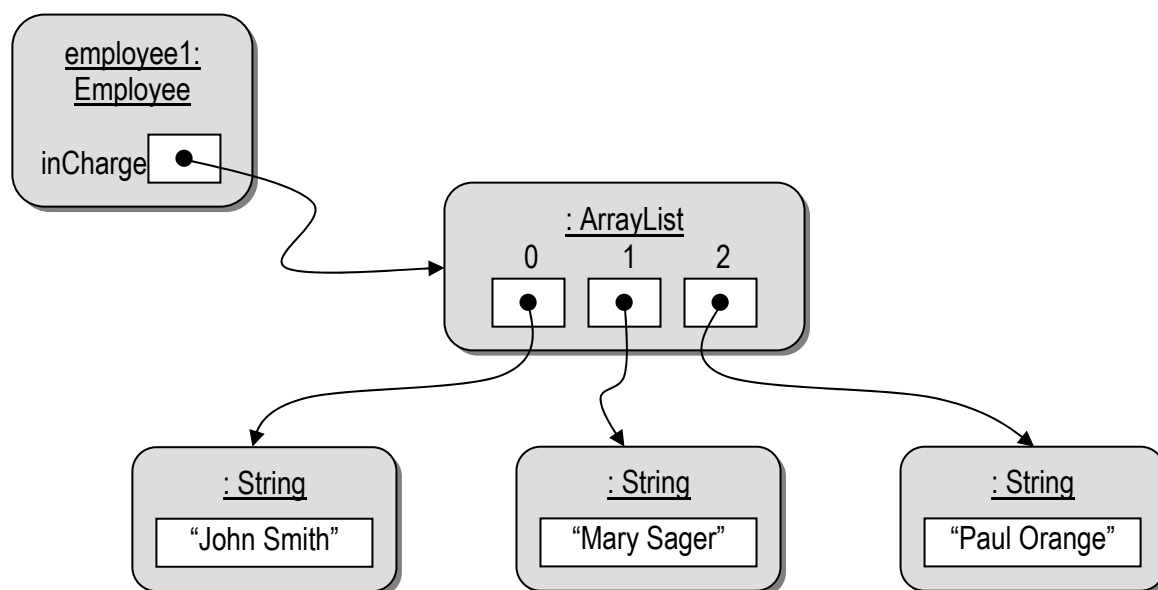
**Self-Study Module: Java and BlueJ refresher**

Fig. 2. Object Diagram of an Employee object

**Task 2 - Employment report**

The company needs to record the “boss-employee” relations among the personnel in such way that it is possible to determine for each employee who is his/her boss as well as the names of the employee(s) he/she supervises. To make this possible will be your responsibility.

To do this you will have to complete the following tasks:

1. Inside the class Employee create the public method `addInChargeName(String name)` which will be responsible for adding to `inCharge` the name of an employee that the current employee supervises. Remember that the names in `inCharge` must be unique.
2. Go to the class AddressBook and create a new public method called `displayStaffHierarchy(String nameEmployee)`. This method should print the name of the input employee, the name of his/her boss (if any) and the name of every employee under his/her command. The method displays for every employee in the address book a message similar to:

```

<<NAME CURRENT EMPLOYEE>>
  BOSS: <<Name of the bosses this employee has>> or “NONE”
  EMPLOYEES: <<Names of the employees the current employee supervises>> or
“NONE”
  
```

To test your modifications use the Lab3Tester class provided.

**Self-Study Module: Java and BlueJ refresher**

**Hints and development suggestions**

- To follow the OO encapsulation principle you are not allowed to access directly any of the attributes of the Employee objects in the entry array; these attributes are defined as private. Then, to retrieve any information of this objects you must use following methods of the class Employee:
  - getName(): Returns a string corresponding to the name of the employee. To get the name of the Employee object in the fifth position of the array entry you should write a line of code similar to: `String name = entry[4].getName();`
  - isInCharge(String name) : Returns true if the input name is in the ArrayList inCharge of the employee.
  - getInChargeListing(): Returns the content of the ArrayList inCharge of the employee as a single String with the names.
  
- The second task can be divided into two sub-tasks:
  - To find the name of the boss(es) of the input employee. If you study carefully the structure of the information in the system you will realise that the only way of knowing who is the boss of an employee is by checking if the name of the input employee is in the inCharge collection of any other employees in the address book. This means you have iterate over *every* employee's inCharge ArrayList object. *Tip: use the Java Iterator class to make the search easier.*
  - To find the name of the employee(s) under supervision of the current employee. As you might have already concluded, this information is recorded in inCharge.

**Review Questions**

1. What is the purpose of a collection class? What advantages does it offer? Explain in the context of the changes made to the Address Book system.
2. When should you use an array instead of an ArrayList object and vice versa?
3. What other Java collection classes are available?