COMP2311

Object Oriented Programming

Prepared by:
Dr. Mamoun Nawahdah

Approved by:
Computer Science Department
September 12, 2022

Contents

Introduction

The aim of this lab manual is to help COMP2311 students to understand and apply a variety of fundamentals of object oriented programming concepts. Every lab session is provided with lab objectives, a brief context about the experiment's topic(s) to strength the student understanding to the lab material; a Java language syntax for the commands or statements that will be used; A pre-lab that the students' should prepare beforehand; and a set of activities that allow the students to completely understand the topic. The activities in this manual are carefully prepared, studied and revised for students practice.

This lab manual continue the students' journey of learning OOP fundamentals. The students will begin with a revision for the basic concepts of the Object-Oriented paradigm (Lab 1), Inheritance and Polymorphism (Lab 2), Abstract Classes and Polymorphism (Lab 3), Interfaces and More Polymorphism (Lab 4), Exceptions and Error Handling, and Binary File Handling (Lab 5), Graphical User Interface - JavaFX concepts (Lab 6, Graphical User Interface - Layout Managers and Basic UI 7), Using Inner Classes and Lambda Expression (Lab 8), Graphical User Interface - Event Driven Programming (Lab 9), Graphical User Interface - UI controllers (Lab 10), Graphical User Interface - Advanced UI controllers and MVC (Lab 11), Multithreading and Parallel Programming (Lab 12), and Java Collections and Generic Types (Lab 13).

The material included in this manual has been adopted from the course's text-book: Y. Daniel Liang, *Introduction to Java programming and data structures*, Twelfth edition, Pearson, 2019. (ISBN-13: 978-0-13-651996-6)

1 Revision for the basic concepts of the Object-Oriented paradigm

1.1 Objectives

- To design programs using the object-oriented paradigm.
- To use the **String** class to process immutable strings.
- To use Ragged Arrays of objects to store data.

1.2 Pre-Lab

1. What is *association*? What is *aggregation*? What is *composition*?
2. How do you convert an integer into a string?
3. How do you convert a numeric string into an integer?
4. How do you convert a double number into a string?
5. How do you convert a numeric string into a double value?
6. What are *autoboxing* and *autounboxing*?
7. To create the string *Welcome to Java*, you may use a statement like this:

   ```
   String s = "Welcome to Java";
   ```

   or

   ```
   String s = new String("Welcome to Java");
   ```

   Which one is better? Why?
8. Does any method in the **String** class change the contents of the string? Why?
9. Can the rows in a two-dimensional array have different lengths?

1.3 Activities

Figure 1 shows a train seat numbering plan. seat numbering plan shows how the seats are laid out. In each row we have 4 seats. In total we have 23 rows.

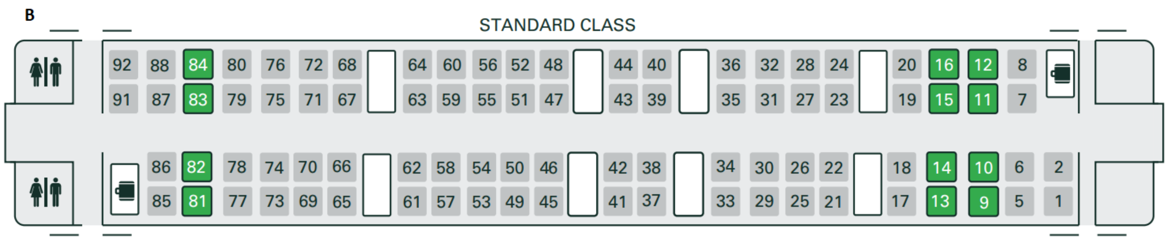


Figure 1: Standard Class Seat Numbering Plan

Activity 1:

Create a **Seat** class that has:

- Private int *seatNumber* with getter and setter methods.
- Private String for *passenger name* with getter and setter methods. Note: if name is *null* this means seat is empty.
- A constructor that takes seat number.
- A method *isEmpty* that returns if the seat is empty or not.
- A *toString* method that returns a string represents seat object's information.
- A static method *isValid* that takes a seat number and returns if the seat number is valid or not.
- A static method *getRow* that takes a seat number and returns the train row number if the seat number is valid.
- A static method *getColumn* that takes a seat number and returns the train column number if the seat number is valid.

Activity 2:

Draw the **UML** diagram for the **Seat** class in Activity 1.

Activity 3:

Create a **Train** class that has the following:

- A private static ragged array of *seats* reflecting the mentioned train seat plan.
- A method to reserve a seat. This method takes seat number and passenger's name. If the seat is empty, the seat will be reserved for the passenger and returns *true*. If not, it will returns *false*.
- A method to delete a reserved seat. If the seat is not empty, the seat will be deleted and returns *true*. Else, it will returns *false*.
- A method to delete all reserved seats.

Activity 4:

Write a driver program to create a *train* object and then keep displays a menu containing the following options:

1. Read passengers file.
2. Reserve a new empty seat.
3. Delete a reserved seat.
4. Delete all reserved seats.
5. Update passengers file.
6. Quit.

Regarding Option 1: consider the file *passengers.txt* file that contains reserved seats in the following format:

```
Seat Number:  Passenger name
```

Read the file line-by-line and update the train object accordingly. Make sure to handle any wrong seat numbers or duplicate reservations for the same seat.

Regarding option 5: you need to write back all the reserved seats information back to the passengers file.

2 Inheritance and Polymorphism

2.1 Objectives

- To define a subclass from a superclass through inheritance.
- To invoke the superclass's constructors and methods using the *super* keyword.
- To override instance methods in the subclass.
- To distinguish differences between overriding and overloading.
- To explore the *toString()* method in the **Object** class.
- To discover polymorphism and dynamic binding.
- To store, retrieve, and manipulate objects in an *ArrayList*.
- To enable data and methods in a superclass accessible from subclasses using the *protected* visibility modifier.
- To prevent class extending and method overriding using the final modifier.

2.2 Context

Superclasses and Subclasses

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

In Java terminology, a class **C1** extended from another class **C2** is called a *subclass*, and **C2** is called a *superclass*. A *superclass* is also referred to as a *parent* class or a *base* class, and a *subclass* as a *child* class, an *extended* class, or a *derived* class. A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.

```
public class C1 extends C2
```

The keyword *extends* tells the compiler that the C1 class extends the C2 class.

Note the following points regarding inheritance:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.
- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass.

- Inheritance is used to model the is-a relationship. A subclass and its superclass must have the is-a relationship.
- Java does not allow multiple inheritance.

Using the *super* Keyword

The keyword *super* refers to the superclass of the class in which super appears. It can be used in two ways:

1. To call a superclass constructor:
   The syntax to call a superclass's constructor is:

   ```
   super() or super(arguments);
   ```

   Note: the call must be the first statement in the constructor.

2. To call a superclass method. The syntax is:

   ```
   super.method(arguments);
   ```

Overriding Methods

To override a method, the method must be defined in the subclass using the same signature as in its superclass.

Overriding vs. Overloading

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.
Note the following:

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class, or in different classes related by inheritance.
- Overridden methods have the same signature; overloaded methods have the same name but different parameter lists.

The Object Class and Its *toString()* and *equals* Methods

Every class in Java is descended from the **java.lang.Object** class.
If no inheritance is specified when a class is defined, the superclass of the class is **Object** by default. The **Object** class has a method called *toString*. The signature of the *toString()* method is:

```
public String toString()
```

Invoking *toString()* on an object returns a string that describes the object. Usually you should override the *toString* method so that it returns a descriptive string representation of the object.

Another method defined in the **Object** class that is often used is the *equals* method. Its signature is:

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. You should override this method in your custom class to test whether two distinct objects have the same content.

Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

Dynamic Binding

A method can be implemented in several classes along the inheritance chain. The **JVM** decides which method is invoked at runtime.

The ArrayList Class

An **ArrayList** object can be used to store a list of objects. Java provides the **ArrayList** class, which can be used to store an unlimited number of objects. Table 1 shows some methods in ArrayList.

Table 1: An ArrayList stores an unlimited number of objects.

Method	Description
+ArrayList()	Creates an empty list.
+add(e: E): void	Appends a new element **e** at the end of this list.
+add(index: int, e: E): void	Adds a new element **e** at the specified index in this list.
+clear(): void	Removes all elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element **o**.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element CDT from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): E	Removes the element at the specified index. Returns the removed element.
+set(index: int, e: E): E	Sets the element at the specified index.

ArrayList is known as a generic class with a generic type **E**. You can specify a concrete type to replace **E** when creating an **ArrayList**.

The protected Data and Methods

A *protected* member of a class can be accessed from a subclass.

Preventing Extending and Overriding

Neither a *final* class nor a *final* method can be extended. A *final* data field is a constant.

2.3 Pre-Lab

1. What keyword do you use to define a subclass?
2. Does Java support multiple inheritance?
3. How do you invoke an overridden superclass method from a subclass?
4. If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?
5. Indicate true or false for the following statements:
   (a) When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.
   (b) You can override a private method defined in a superclass.
   (c) You can always successfully cast an instance of a subclass to a superclass.
   (d) You can always successfully cast an instance of a superclass to a subclass.
6. What modifier should you use on a class so a class in the same package can access it, but a class in a different package cannot access it?
7. What modifier should you use so a class in a different package cannot access the class, but its subclasses in any package can access it?
8. How do you prevent a class from being extended? How do you prevent a method from being overridden?

2.4 Activities

Activity 1:

Write the following method that returns the maximum value in an **ArrayList** of integers. The method returns *null* if the list is *null* or the list size is 0.

```
public static Integer max(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter a sequence of numbers ending with 0 and invokes this method to return the largest number in the input.

Activity 2:

Write a method that removes the duplicate elements from an array list of integers using the following header:

```
public static void removeDuplicate(ArrayList<Integer> list)
```

Write a test program that prompts the user to enter 10 integers to a list and displays the distinct integers in their input order and separated by exactly one space.

Activity 3:

Consider the UML in Figure 2. Implement the UML.
Note: in each class:

- Add all the setters and getters for the class attributes.
- override the *toString* method.

Note: keep the code as it will be used again in future labs.

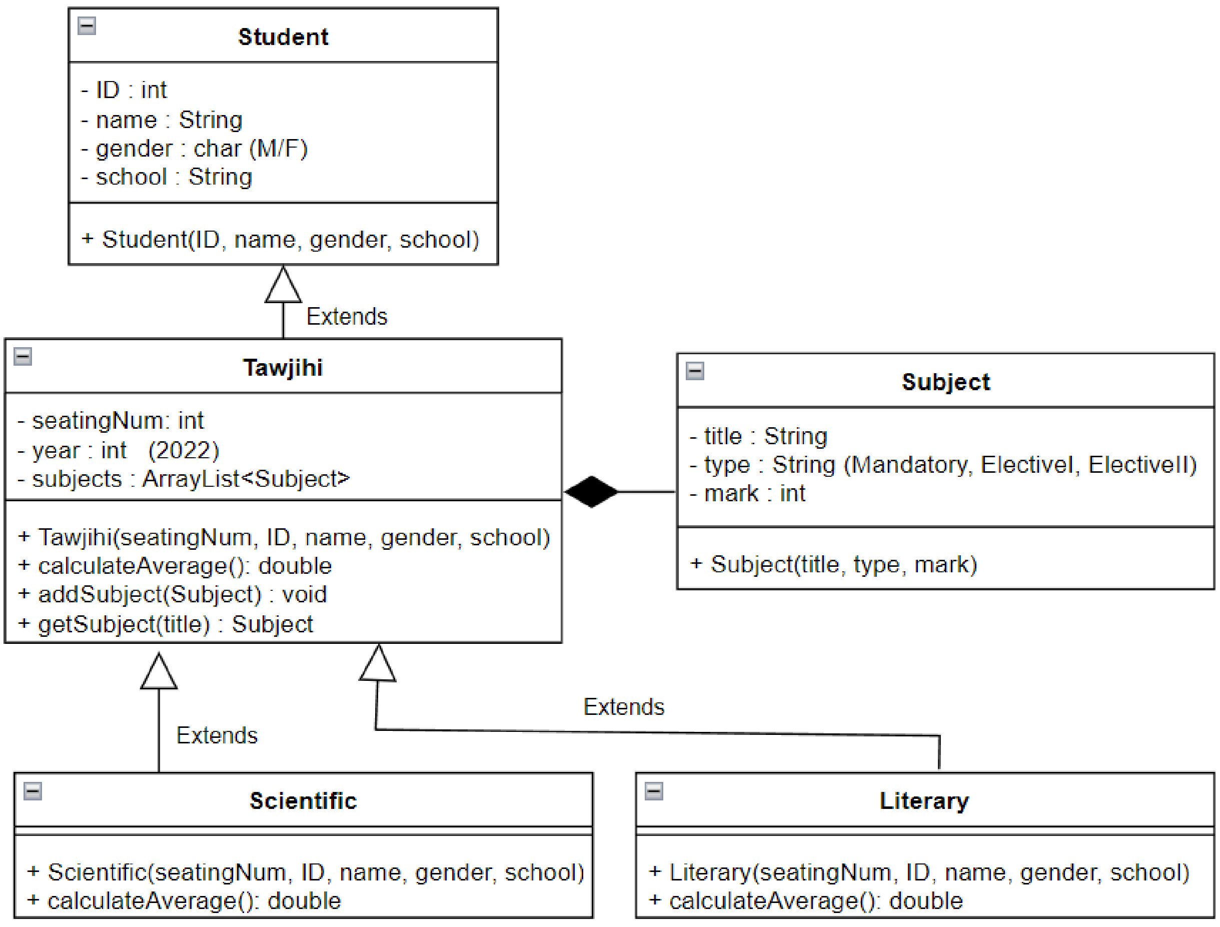

Student
- ID : int
- name : String
- gender : char (M/F)
- school : String
+ Student(ID, name, gender, school)
Extends
Tawjihi
- seatingNum: int
- year : int (2022)
- subjects : ArrayList<Subject>
+ Tawjihi(seatingNum, ID, name, gender, school)
+ calculateAverage(): double
+ addSubject(Subject) : void
+ getSubject(title) : Subject
Subject
- title : String
- type : String (Mandatory, ElectiveI, ElectiveII)
- mark : int
+ Subject(title, type, mark)
Extends
Extends
Scientific
+ Scientific(seatingNum, ID, name, gender, school)
+ calculateAverage(): double
Literary
+ Literary(seatingNum, ID, name, gender, school)
+ calculateAverage(): double

Figure 2: Tawjihi UML (version 1)

3 Abstract Classes and Polymorphism

3.1 Objectives

- To design and use abstract classes.
- To generalize numeric wrapper classes **BigInteger** and **BigDecimal** using the abstract **Number** class.
- To process a calendar using the **Calendar** and **GregorianCalendar** classes.

3.2 Context

Abstract Classes

In the inheritance hierarchy, classes become more specific and concrete with each new subclass. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure a superclass contains common features of its subclasses. Sometimes, a superclass is so abstract it cannot be used to create any specific instances. Such a class is referred to as an *abstract class.*

In some cases, some methods cannot be implemented in the superclass because their implementation depends on the specific type of subclass object. Such methods are referred to as *abstract methods* and are denoted using the ***abstract*** modifier in the method header. Abstract classes are denoted using the ***abstract*** modifier in the class header as well.

Abstract classes are like regular classes, but you cannot create instances of abstract classes using the new operator. An abstract method is defined without implementation. Its implementation is provided by the subclasses. A class that contains abstract methods must be defined as abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods. This abstract class is used as a base class for defining subclasses.

In UML graphic notation, the names of abstract classes and their abstract methods are ***italicized***.

The Abstract Number Class

Number is an abstract superclass for numeric wrapper classes **BigInteger** and **BigDecimal**. These classes have common methods *byteValue(), shortValue(), intValue(), longValue(), floatValue()*, and *doubleValue()* for returning a **byte, short, int, long, float,** and **double** value from an object of these classes. These common methods are actually defined in the **Number** class. With **Number** defined as the superclass for the numeric classes, we can define methods to perform common operations for numbers. For example, we can create an **ArrayList** of **Number** objects to adds an **Integer** object, a **Double** object, a **BigInteger** object, and a **BigDecimal** object to the same list.

Calendar and GregorianCalendar

java.util.Calendar is an abstract base class for extracting detailed calendar information, such as the year, month, date, hour, minute, and second. Subclasses of Calendar can implement specific calendar systems, such as the Gregorian calendar and the lunar calendar. Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in Java. The *add* method is abstract in the **Calendar** class because its implementation is dependent on a concrete calendar system. You can use *new GregorianCalendar()* to construct a default GregorianCalendar with the current time and *new GregorianCalendar(year, month, date)* to construct a GregorianCalendar with the specified year, month, and date. The month parameter is 0-based that is, 0 is for January.

The *get(int field)* method defined in the **Calendar** class is useful for extracting the date and time information from a **Calendar** object. The fields are defined as constants. The *set(int field, value)* method defined in the **Calendar** class can be used to set a field. For example, you can use *calendar.set(Calendar.DAY_OF_MONTH, 1)* to set the calendar to the first day of the month.

3.3 Pre-Lab

1. True or false?

   - An abstract class can be used just like a nonabstract class except that you cannot use the **new** operator to create an instance from the abstract class.
   - An abstract class can be extended.
   - A subclass of a nonabstract superclass cannot be abstract.
   - A subclass cannot override a concrete method in a superclass to define it as abstract.
   - An abstract method must be nonstatic.

2. Can you create a **Calendar** object using the **Calendar** class?

3. Which method in the **Calendar** class is abstract?

4. For a **Calendar** object c, how do you get its year, month, date, hour, minute, and second?

3.4 Activities

Activity 1:

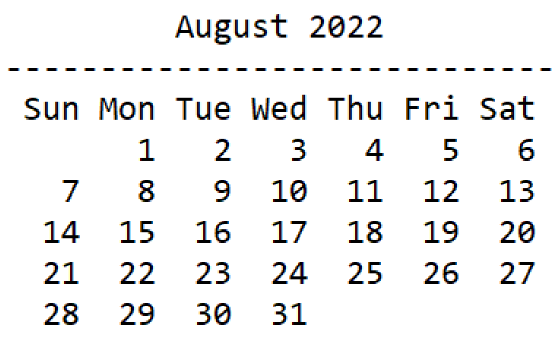
Write the following method that shuffles an **ArrayList** of numbers:

```
public static void shuffle(ArrayList<Number> list)
```

Hint: Array shuffling means to randomizes the order of the elements in the array.

Activity 2:

Write PrintCalendar class to display a calendar for the current month using the **Calendar** and **GregorianCalendar** classes. Figure 3 shows a sample execution for the month August, 2022:



```
          August 2022
-----------------------------
 Sun Mon Tue Wed Thu Fri Sat
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28  29  30  31
```

Figure 3: August 2022 Calendar

Activity 3:

Consider the updated Tawjihi UML in Figure 4.

- Make **Student** class abstract.
- Make **Tawjihi** class abstract and make *calculateAverage()* an abstract method.
- Override *toString()* method in **Scientific** class and **Literary** class.
- Override *calculateAverage()* method in **Scientific** class.
  Note: to calculate average in scientific branch:
  1. Calculate sum of mandatory subjects (Arabic, English, Physics, and Math).
  2. Add to the sum the largest between Elective I (Biology or Chemistry).
  3. Add to the sum the largest between Elective II (Religious Education or Technology).
  4. Divide sum over 7.

- Override *calculateAverage()* method in **Literary** class.
  Note: to calculate average in Literary branch:

  1. Calculate sum of mandatory subjects (Arabic, English, Physics, and History).
  2. Add to the sum the largest between Elective I (Geography or Religious Education).
  3. Add to the sum the largest between Elective II (Scientific Culture or Technology).
  4. Divide sum over 7.

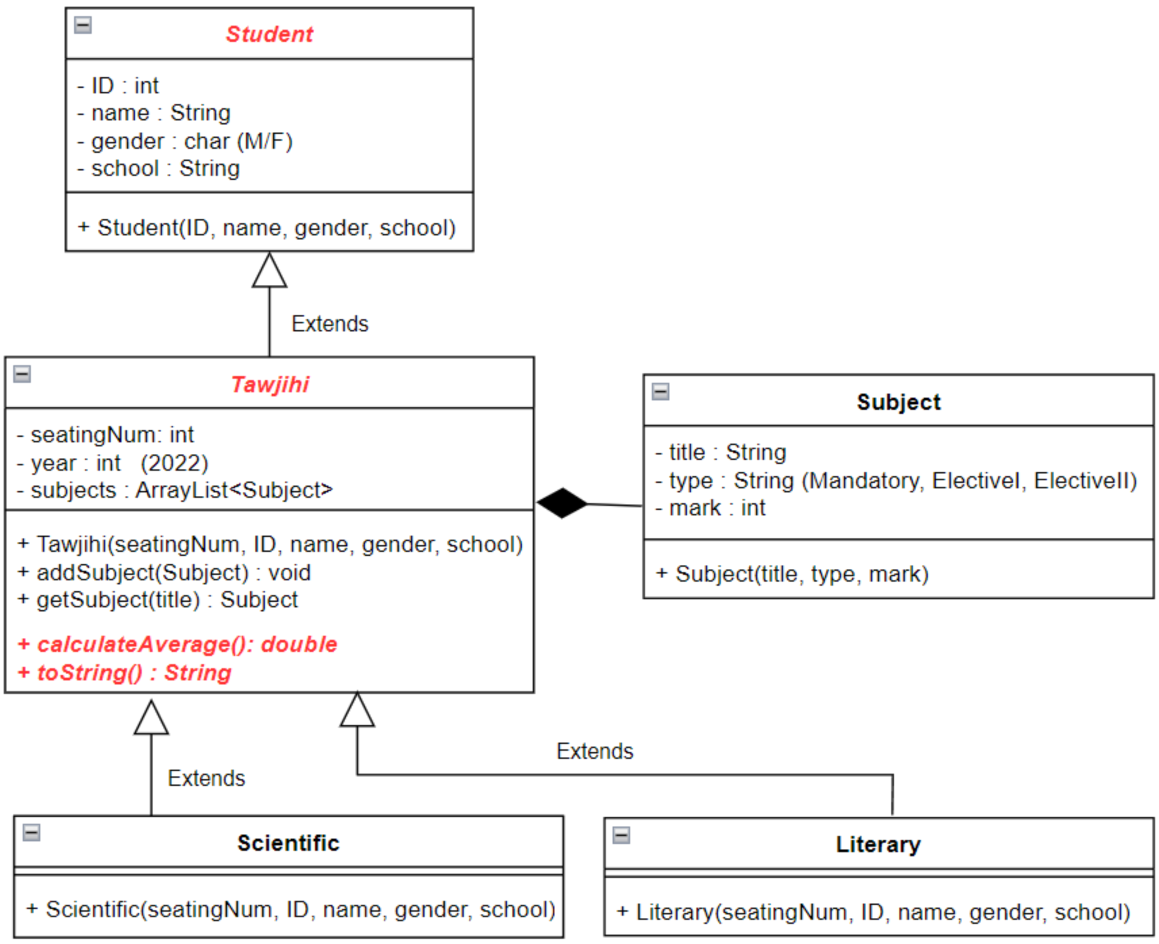

Student
- ID : int
- name : String
- gender : char (M/F)
- school : String
+ Student(ID, name, gender, school)
Extends
Tawjihi
- seatingNum: int
- year : int (2022)
- subjects : ArrayList<Subject>
+ Tawjihi(seatingNum, ID, name, gender, school)
+ addSubject(Subject) : void
+ getSubject(title) : Subject
+ calculateAverage(): double
+ toString() : String
Subject
- title : String
- type : String (Mandatory, ElectiveI, ElectiveII)
- mark : int
+ Subject(title, type, mark)
Extends
Extends
Scientific
+ Scientific(seatingNum, ID, name, gender, school)
Literary
+ Literary(seatingNum, ID, name, gender, school)

Figure 4: Tawjihi UML (version 2)

- Write a driver class, create an **ArrayList** of 6 Tawjihi students (3 Scientific and 3 Literary), then calculate the average of each student and report the results in a table, finally report the top students in each branch.

4 Interfaces and More Polymorphism

4.1 Objectives

- To specify common behavior for objects using interfaces.
- To define interfaces and define classes that implement interfaces.
- To define a natural order using the **Comparable** interface.
- To make objects cloneable using the **Cloneable** interface.

4.2 Context

Interfaces

In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects of related classes or unrelated classes. To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
  }
```

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. You can use an interface more or less the same way you use an abstract class. The relationship between the class and the interface is known as interface inheritance.

In UML graphic notation, the interface and its methods are italicized. The dashed line and hollow triangle are used to point to the interface.

- Note 1: The modifiers *public static final* on data fields and the modifiers *public abstract* on methods can be omitted in an interface.
- Note 2: Java 8 introduced default interface methods using the keyword *default*. A default method provides a default implementation for the method in the interface.
- Note 3: Java 8 also permits public static methods in an interface. A public static method in an interface can be used just like a public static method in a class.
- Note 4: In Java 9, you can also use private methods in an interface. These methods are used for implementing the default methods and public static methods.

The Comparable Interface

The **Comparable** interface defines the *compareTo* method for comparing objects.

The *compareTo* method determines the order of this object with the specified object **o** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than **o**. The **Comparable** interface is a generic interface. The generic type **E** is replaced by a concrete type when implementing this interface.

Since all **Comparable** objects have the *compareTo* method, the *java.util.Arrays.sort(Object[])* method in the Java API uses the *compareTo* method to compare and sorts the objects in an array, provided the objects are instances of the **Comparable** interface.

The Cloneable Interface

Often, it is desirable to create a copy of an object. To do this, you need to use the clone method and understand the Cloneable interface.

The **Cloneable** interface in the **java.lang** package is defined as follows:

```
package java.lang;
      public interface Cloneable {
  }
```

This interface is empty. An interface with an empty body is referred to as a **marker interface**. A marker interface is used to denote that a class possesses certain desirable properties. A class that implements the **Cloneable** interface is marked cloneable, and its objects can be cloned using the *clone()* method defined in the **Object** class.

To define a custom class that implements the **Cloneable** interface, the class must override the *clone()* method in the **Object** class.

4.3 Pre-Lab

1. Suppose **A** is an interface. Can you create an instance using *new A()*?
2. Suppose **A** is an interface. Can you declare a reference variable **x** with type **A**?
3. True or false? If a class implements **Comparable**, the object of the class can invoke the *compareTo* method.
4. Can a class invoke *super.clone()* when implementing the *clone()* method if the class does not implement **java.lang.Cloneable**?
5. Does the **Date** class implement **Cloneable**?
6. Give an example to show why interfaces are preferred over abstract classes.
7. What are the similarities and differences between abstract classes and interfaces?

4.4 Activities

Activity 1:

Define a class named **Time** for encapsulating a time. The class contains the following:

- A data field of the long *time* that stores the elapsed time since midnight, Jan 1, 1970.
- A no-arg constructor that constructs a **Time** object for the current time.
- A constructor with the specified *hour*, *minute*, and *second* to create a **Time**.
- A constructor with the specified elapsed time since midnight, Jan 1, 1970.
- The *getHour()* method that returns the current hour in the range 0-23.
- The *getMinute()* method that returns the current minute in the range 0-59.
- The *getSecond()* method that returns the current second in the range 0-59.
- The *getSeconds()* method that returns the elapsed total seconds.
- The *toString()* method that returns a string such as `"1 hour 2 minutes 1 second"` and `"14 hours 21 minutes 1 second"`.
- Implement the **Comparable<Time>** interface to compare this **Time** with another one based on their elapse seconds. The *compareTo* method returns the difference between this object's elapse seconds and the another's.
- Implement the **Cloneable** interface to clone a **Time** object.
- Write a test program to test **Time** class.

Activity 2:

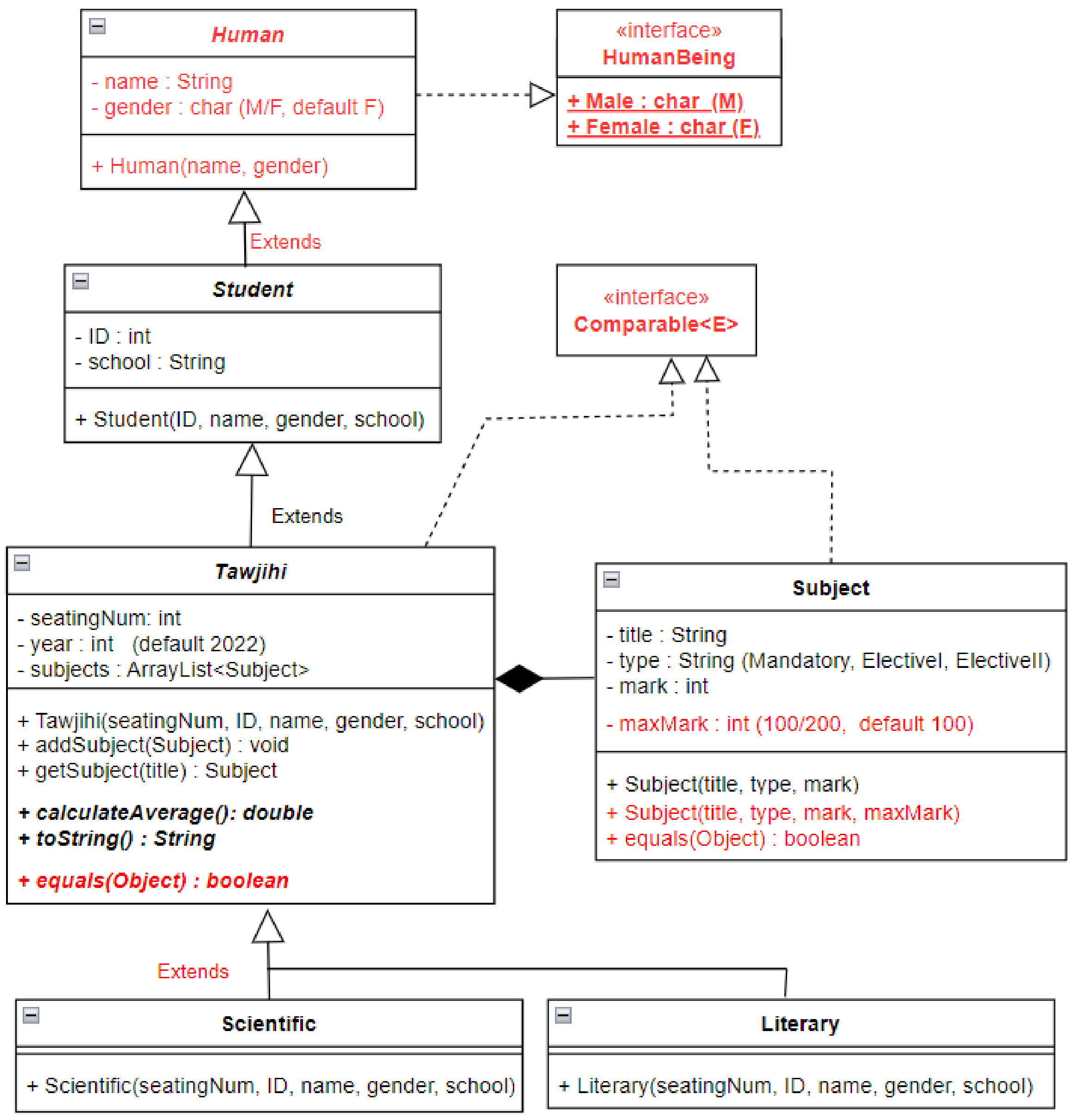
Design a class named **Point** that meets the following requirements:

- Two data fields *x* and *y* for representing a point with getter methods.
- A no-arg constructor that constructs a point for `(0, 0)`.
- A constructor that constructs a point with the specified `x` and `y` values.
- Override the *equals* method. Point *p1* is said to be greater than point *p2* if `p1.x == p2.x` and `p1.y == p2.y`.
- Implement the **Comparable<Point>** interface and the *compareTo* method. Point *p1* is said to be greater than point *p2* if `p1.x > p2.x` or if `p1.x == p2.x and p1.y > p2.y`.
- Override the *toString()* method to return a string as `[x value, y value]`.
- Implement the **Cloneable** interface and *clone* method.
- Write a test program to test **Point** class.

Activity 3:

Consider the updated Tawjihi UML in Figure 5.

- Create an interface **HumanBeing** that has to char constants (`Male/Female`).
- Create a new base class called **Human** that has two attributes *name* and *gender*. Make this class implements **HumanBeing** interface.
- Update **Student** class and make it extends **Human** base class.
- Update **Subject** class by adding a new attribute *maxMark*. Add a new constructor using the all the attributes. Make Subject class Comparable. Two subjects are compared if they both have the same title according to the mark. Override the *equals* method. Two subjects are equals if they both have the same title and same mark.
- Update **Tawjihi** class by make it Comparable. Two tawjihi students are compared according to their calculated average. Override *equals* method and make it `abstract`.
- Write a driver class, create an **ArrayList** of 6 Tawjihi students (3 Scientific and 3 Literary), then display the students in ascending order according to their calculated average.



Human
- name : String
- gender : char (M/F, default F)
+ Human(name, gender)
«interface»
HumanBeing
+ Male : char (M)
+ Female : char (F)
Extends
Student
- ID : int
- school : String
+ Student(ID, name, gender, school)
«interface»
Comparable<E>
Extends
Tawjihi
- seatingNum: int
- year : int (default 2022)
- subjects : ArrayList<Subject>
+ Tawjihi(seatingNum, ID, name, gender, school)
+ addSubject(Subject) : void
+ getSubject(title) : Subject
+ calculateAverage(): double
+ toString() : String
+ equals(Object) : boolean
Subject
- title : String
- type : String (Mandatory, ElectiveI, ElectiveII)
- mark : int
- maxMark : int (100/200, default 100)
+ Subject(title, type, mark)
+ Subject(title, type, mark, maxMark)
+ equals(Object) : boolean
Extends
Scientific
+ Scientific(seatingNum, ID, name, gender, school)
Literary
+ Literary(seatingNum, ID, name, gender, school)

Figure 5: Tawjihi UML (version 3)

5 Exceptions and Error Handling, and Binary File Handling

5.1 Objectives

- To explore the advantages of using exception handling.
- To distinguish exception types: **Error** (fatal) vs. **Exception** (nonfatal) and checked vs. unchecked.
- To write a *try-catch* block to handle exceptions.
- To use the *finally* clause in a *try-catch* block.
- To rethrow exceptions in a *catch* block.
- To define custom exception classes.
- To distinguish between text I/O and binary I/O.
- To read and write bytes using **FileInputStream** and **FileOutputStream**.
- To read and write files using the **RandomAccessFile** class.

5.2 Context

Exception-Handling Overview

Exceptions are runtime errors. Exception handling enables a program to deal with runtime errors and continue its normal execution. Java enables a method to throw an exception that can be caught and handled by the caller. When an exception is thrown, the normal execution flow is interrupted. The statement for invoking the method is contained in a *try* block. The *try* block contains the code that is executed in normal circumstances. The exception is caught by the *catch* block. The code in the *catch* block is executed to handle the exception. Afterward, the statement after the *catch* block is executed. In summary, a template for a try-throw-catch block may look as follows:

```
try {
      Code to run;
      A statement or a method that may throw an exception;
      More code to run;
 }
 catch (type ex) {
      Code to process the exception;
 }
```

An exception may be thrown directly by using a throw statement in a try block, or by invoking a method that may throw an exception.

Exception Types

The root class for exceptions is **java.lang.Throwable**. There are many predefined exception classes in the Java API. All Java exception classes inherit directly or indirectly from **Throwable**.

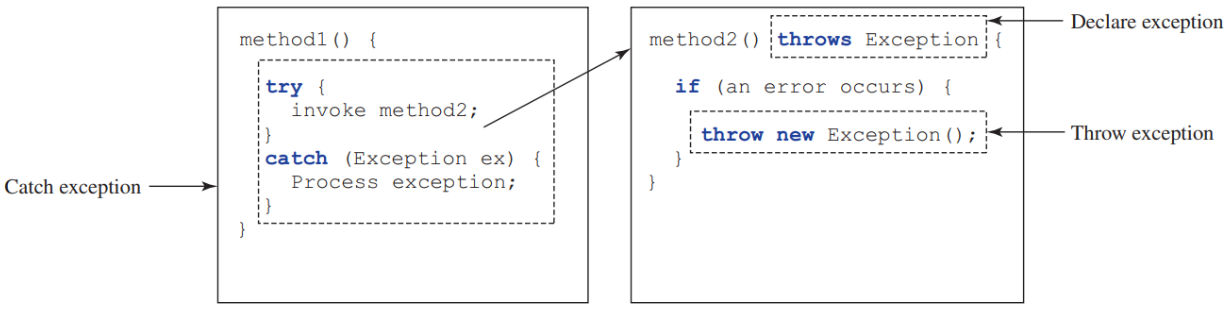
The exception classes can be classified into three major types:

- *System errors* are thrown by the JVM and are represented in the **Error** class. The **Error** class describes internal system errors, though such errors rarely occur.
- *Exceptions* are represented in the **Exception** class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program.
- *Runtime exceptions* are represented in the **RuntimeException** class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors.

RuntimeException, Error, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning the compiler forces the programmer to check and deal with them in a `try-catch` block or declare it in the method header.

Declaring, Throwing, and Catching Exceptions

Java's exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 6.



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

Figure 6: Exception handling

If no exceptions arise during the execution of the try block, the catch blocks are skipped. If one of the statements inside the try block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception.

The finally Clause

The finally clause is always executed regardless of whether an exception occurred or not. The syntax for the *finally* clause might look like this:

```
try {
      statements;
  }
  catch (TheException ex) {
      handling ex;
  }
  finally {
      finalStatements;
  }
```

Note: The *catch* block may be omitted when the *finally* clause is used.

Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception. The syntax for rethrowing an exception may look like this:

```
 try {
      statements;
  }
  catch (TheException ex) {
      perform operations before exits;
      throw ex;
  }
```

Defining Custom Exception Classes

You can create your own exception class, derived from **Exception** or from a subclass of **Exception**, such as **IOException**.

Text I/O vs. Binary I/O

All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding. Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. Binary I/O is more efficient than text I/O because binary I/O does not require encoding and decoding.

InputStream/OutputStream

The abstract **InputStream** is the root class for reading binary data, and the abstract **OutputStream** is the root class for writing binary data. Tables 2 and 3 list all the methods in the classes **InputStream** and **OutputStream**.

Table 2: The abstract **InputStream** class.

Method	Description
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0-255. If no byte is available because the end of the stream has been reached, the value -1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores them in b[off], b[off+1],..., b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+close(): void	Closes this input stream and releases any system resources occupied by it.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.

Table 3: The abstract **InputStream** class.

Method	Description
+write(int b): void	Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.
+write(b: byte[], off: int, len: int): void	Writes b[off], b[off+1],..., b[off+len-1] into the output stream.
+write(b: byte[]): void	Writes all the bytes in array b to the output stream.
+close(): void	Closes this output stream and releases any system resources occupied by it.
+flush(): void	Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream/FileOutputStream

FileInputStream/FileOutputStream are for reading/writing bytes from/to files. All the methods in these classes are inherited from **InputStream** and **OutputStream**. A **java.io.FileNotFoundException** will occur if you attempt to create a **FileInputStream** with a nonexistent file.

DataInputStream reads bytes from the stream and converts them into appropriate primitive-type values or strings. **DataOutputStream** converts primitive-type values or strings into bytes and outputs the bytes to the stream.

DataInputStream/DataOutputStream are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

Caution: You have to read data in the same order and format in which they are stored.

If you keep reading data at the end of an **InputStream**, an **EOFException** will occur. This exception can be used to detect the end of a file

5.3 Pre-Lab

1. What is a checked exception and what is an unchecked exception?
2. How do you declare an exception and where?
3. Can you declare multiple exceptions in a method header?
4. How do you throw an exception?
5. What is the keyword *throw* used for? What is the keyword *throws* used for?
6. How do you define a custom exception class?
7. What are the differences between text I/O and binary I/O?
8. Why should you always close streams? How do you close streams?
9. How do you check the end of a file in an input stream (**FileInputStream, DataInputStream**)?

5.4 Activities

Activity 1:

- Write the *hex2Dec(String hexString)* method, which converts a hex string into a decimal number. Implement the hex2Dec method to throw a **NumberFormatException** if the string is not a hex string. Write a test program that prompts the user to enter a hex number as a string and displays its decimal equivalent. If the method throws an exception, display (`Not a hex number`).

Activity 2:

Re-write the hex2Dec method in activity 1 to throw a **HexFormatException** if the string is not a hex string. Define a custom exception called **HexFormatException**.

Activity 3:

Write a program to create a file named `data.dat` if it does not exist. Append new data to it if it already exists. Write 100 integers created randomly into the file using binary I/O.

Activity 4:

Suppose a binary data file named `data.dat` has been created from Activity 3 and its data are created using *writeInt(int)* in **DataOutputStream**. The file contains an unspecified number of integers. Write a program to find the sum of the integers.

Activity 5:

Implement a class named **BitOutputStream**, as shown in Table 4 , for writing bits to an output stream. The *writeBit(char bit)* method stores the bit in a byte variable. When you create a **BitOutputStream**, the byte is empty. After invoking *writeBit('1')*, the byte becomes `00000001`. After invoking *writeBit("0101")*, the byte becomes `00010101`. The first three bits are not filled yet. When a byte is full, it is sent to the output stream. Now the byte is reset to empty. You must close the stream by invoking the *close()* method. If the byte is neither empty nor full, the *close()* method first fills the zeros to make a full 8 bits in the byte and then outputs the byte and closes the stream.

Table 4: BitOutputStream outputs a stream of bits to a file.

Method	Description
+BitOutputStream(file: File)	Creates a BitOutputStream to write bits to the file.
+writeBit(char bit): void	Writes a bit '0' or '1' to the output stream.
+writeBit(String bit): void	Writes a string of bits to the output stream.
+close(): void	This method must be invoked to close the stream.

Activity 6:

Create a file `Tawjihi.dat` that has tawjihi student records in the following format:

- ID:int
- name:String
- gender:char
- school:String
- Branch:String
- seatingNum:int
- year:int
- numOfSubjects:int
- subjectTitle1:String
- subjectType1:String
- subjectMark1:int
- subjectMaxMark1:int
- subjectTitle2:String
- subjectType2:String
- subjectMark2:int
- subjectMaxMark2:int
  :
- subjectTitleN:String
- subjectTypeN:String
- subjectMarkN:int
- subjectMaxMarkN:int

Using a **DataOutputStream** object, write 2 students records (one Scientific and one Literary) to the file. Then using a **DataInputStream** object, read the students records from the file and convert them to specific Tawjihi student object (Scientific or Literary) as in Lab 4 - Activity 3.

6 GUI - JavaFX concepts

6.1 Objectives

- To write a simple JavaFX program and understand the relationship among stages, scenes, and nodes.
- To create user interfaces using panes, groups, UI controls, and shapes.
- To create colors using the **Color** class.
- To create fonts using the **Font** class.
- To create images using the **Image** class, and to create image views using the **ImageView** class.
- To display text using the **Text** class, and create shapes using the **Line, Circle, Rectangle, Ellipse, Arc, Polygon**, and **Polyline** classes.

6.2 Context

The Basic Structure of a JavaFX Program

The **javafx.application.Application** class defines the essential framework for writing JavaFX programs. The main class overrides the *start* method defined in **Application** class as shown in Figure 7. The *start* method normally places UI controls in a scene and displays the scene in a stage. A **Scene** object can be created using the constructor *Scene(node, width, height)*. A **Stage** object is a window. A **Stage** object called primary stage is automatically created by the JVM when the application is launched. You can create additional stages if needed. Figure 8 illustrates the structure of every JavaFX application.

Panes, Groups, UI Controls, and Shapes

Panes, Groups, UI controls, and shapes are subtypes of Node. Panes are used for laying out the nodes in a desired location and size. A node is a visual component such as a shape, an image view, a UI control, a group, or a pane. A shape refers to a text, line, circle, ellipse, rectangle, arc, polygon, polyline, and so on. A UI control refers to a label, button, check box, radio button, text field, text area, and so on. A group is a container that groups a collection of nodes.

Note the coordinates of the upper-left corner of the pane is `(0, 0)` in the Java coordinate system

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class MyJavaFX extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place a button in the scene
    Button btOK = new Button("OK");
    Scene scene = new Scene(btOK, 200, 250);
    primaryStage.setTitle("MyJavaFX"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    Application.launch(args);
  }
}
```

Figure 7: MyJavaFX.java

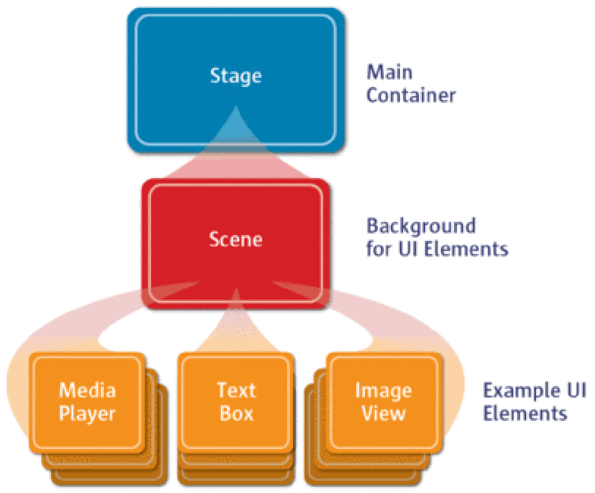


Figure 8: The structure of every JavaFX application

The Color Class

The **javafx.scene.paint.Color** class can be used to create colors. A color instance can be constructed using the following constructor:

```
public Color(double r, double g, double b, double opacity);
```

in which r, g, and b specify a color by its red, green, and blue components with values in the range from 0.0 (darkest shade) to 1.0 (lightest shade). The opacity value defines the transparency of a color within the range from 0.0 (completely transparent) to 1.0 (completely opaque).

The Font Class

A **javafx.scene.text.Font** describes font name, weight, and size.

The Image and ImageView Classes

The **javafx.scene.image.Image** class represents a graphical image and is used for loading an image from a specified filename or a URL. The **javafx.scene.image.ImageView** is a node for displaying an image. An **ImageView** can be created from an **Image** object. For example, the following code creates an ImageView from an image file:

```
Image image = new Image("image/mamoun.gif");
ImageView imageView = new ImageView(image);
```

Alternatively, you can create an ImageView directly from a file or a URL as follows:

```
ImageView imageView = new ImageView("image/mamoun.gif");
```

Shapes

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines. The **Shape** class is the abstract base class that defines the common properties for all shapes. Among them are the *fill*, *stroke*, and **strokeWidth** properties.

- **Text**: The **Text** class defines a node that displays a string at a starting point (x, y).
- **Line**: A line connects two points with four parameters *startX*, *startY*, *endX*, and *endY*.
- **Rectangle**: A rectangle is defined by the parameters *x, y, width, height, arcWidth*, and *arcHeight*.

- **Circle** and **Ellipse**: A circle is defined by its parameters *centerX, centerY*, and *radius*. An ellipse is defined by its parameters *centerX, centerY, radiusX*, and *radiusY*.
- **Arc**: An arc is conceived as part of an ellipse, defined by the parameters *centerX, centerY, radiusX, radiusY, startAngle, length*, and an arc type (*ArcType.OPEN, ArcType .CHORD,* or *ArcType.ROUND*).
- **Polygon** and **Polyline**: The **Polygon** class defines a polygon that connects a sequence of points. The **Polyline** class is similar to the **Polygon** class except that the **Polyline** class is not automatically closed.

6.3 Pre-Lab

1. How do you define a JavaFX main class?
2. What is the signature of the start method?
3. What is a stage? What is a primary stage?
4. How do you display a stage?
5. How do you create a **Scene** object?
6. How do you set a scene in a stage?
7. How do you place a circle into a scene?
8. What is a pane?
9. What is a node? How do you place a node in a pane?
10. Can you directly place a **Shape** or an **ImageView** into a **Scene**?
11. How do you create a color?
12. How do you create a **Font** object with font name Courier, size 20, and weight bold?
13. How do you create an **Image** from a URL or a filename?
14. How do you create an **ImageView** from an **Image** or directly from a file or a URL?

6.4 Activities

Activity 1:

Write a javaFX program that displays an image in a pane, as shown in Figure 9.

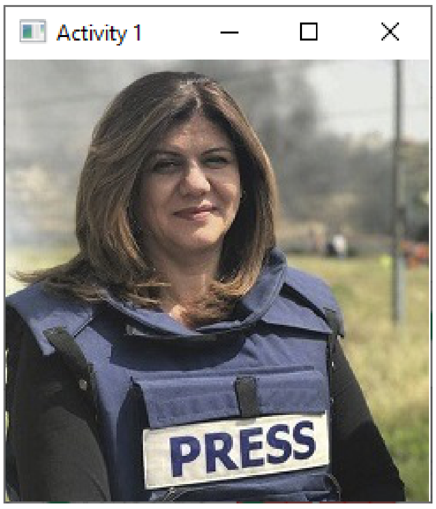


Figure 9: Activity 1

Activity 2:

Write a javaFX program that displays a face using shapes, as shown in Figure 10.

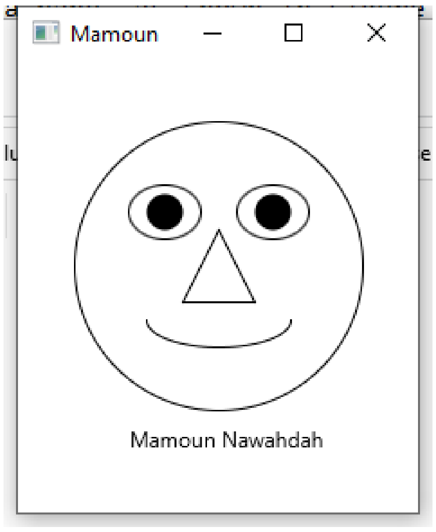


Figure 10: Activity 2

Activity 3:

Write a program that displays a checkerboard in which each white and black cell is a Rectangle with a fill color black or white, as shown in Figure 11.

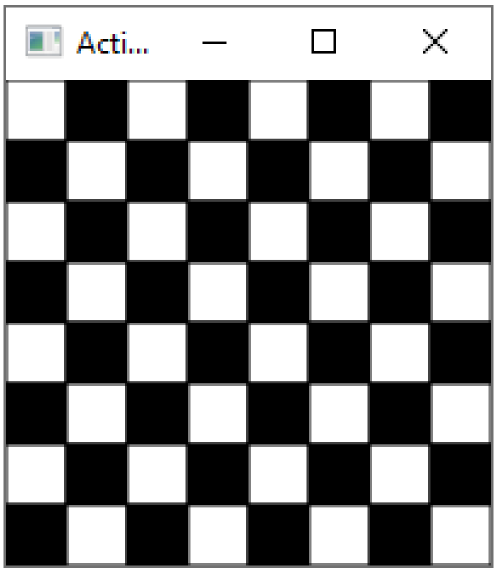


Figure 11: Activity 3

Activity 4:

Write a program that uses a bar chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 12. Suppose projects take 20% and are displayed in red, quizzes take 10% and are displayed in blue, midterm exams take 30% and are displayed in green, and the final exam takes 40% and is displayed in orange.

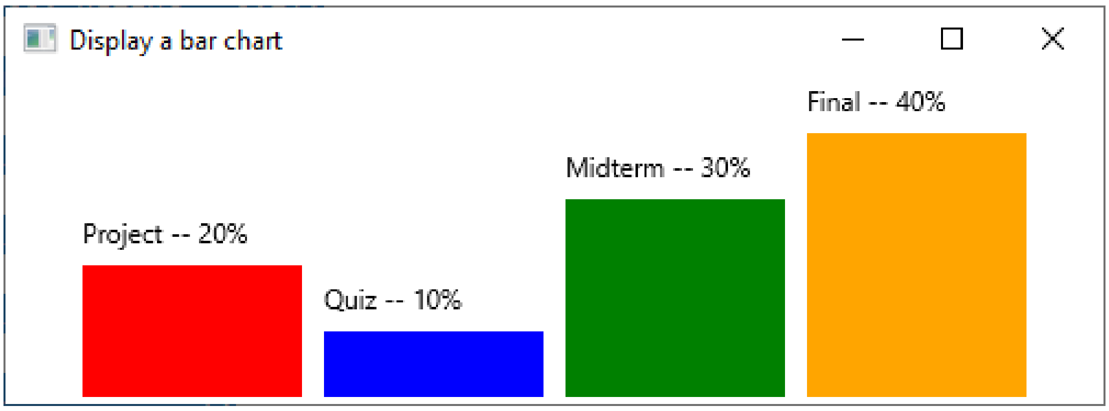


Figure 12: Activity 4

Activity 5:

Write a program that uses a pie chart to display the percentages of the overall Tawjihi statistics represented by Scientific, Literary, and others, as shown in Figure 13. Suppose Scientific take 35% and are displayed in red, Literary take 55% and are displayed in light blue, and others take 10% and are displayed in green.

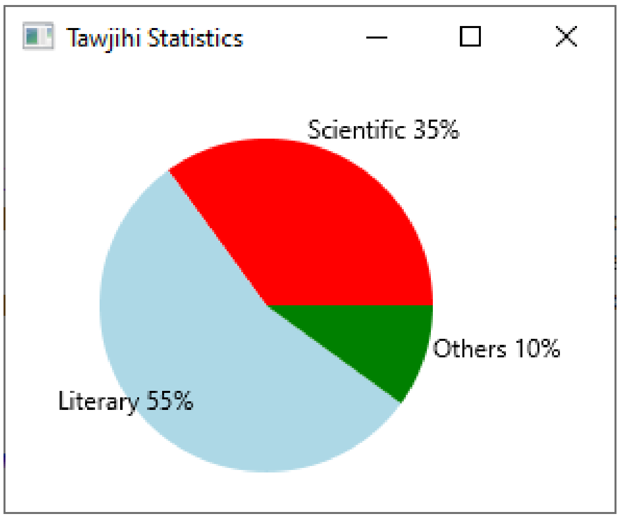


Figure 13: Activity 5

7 GUI - Layout Managers and Basic UI

7.1 Objectives

- To use the common property style for nodes.
- To update property values automatically through property binding.
- To layout nodes using **Pane, StackPane, FlowPane, GridPane, BorderPane, HBox**, and **VBox**.
- To create graphical user interfaces with various user-interface controls.
- To create a label with text and graphics using the **Label** class.
- To create a button with text and graphic using the **Button** class.
- To enter data using the **TextField** class and password using the **PasswordField** class.

7.2 Context

Style Property

JavaFX style property are similar to cascading style sheets (CSS) used to specify the styles for HTML elements in a Web page. Therefore, the style properties in JavaFX are called JavaFX CSS. In JavaFX, a style property is defined with a prefix **`-fx-`**. The syntax for setting a style is **`styleName:value`**. Multiple style properties for a node can be set together separated by semicolon (**;**). For example, the following statement:

```
circle.setStyle("-fx-stroke:  black; -fx-fill:  red;");
```

Property Binding

JavaFX introduces a new concept called *property binding* that enables a target object to be bound to a source object. If the value in the source object changes, the target object is also automatically changed.

A target binds with a source using the bind method as follows:

```
target.bind(source);
```

The bind method is defined in the **javafx.beans.property.Property** interface. A binding property is an instance of **javafx.beans.property.Property**. An observable source object is an instance of the **javafx.beans.value.ObservableValue** interface. An **ObservableValue** is an entity that wraps a value and allows to observe the value for changes.

Layout Panes and Groups

Panes and groups are the containers for holding nodes. The **Group** class is often used to group nodes and to perform transformation and scale as a group. JavaFX provides many types of panes for organizing nodes in a container:

- **Pane**: Base class for layout panes. It contains the *getChildren()* method for returning a list of nodes in the pane.
- **StackPane**: Places the nodes on top of each other in the center of the pane.
- **FlowPane**: Places the nodes row-by-row horizontally or column-by-column vertically.
- **GridPane**: Places the nodes in the cells in a two-dimensional grid.
- **BorderPane**: Places the nodes in the top, right, bottom, left, and center regions.
- **HBox**: Places the nodes in a single row.
- **VBox**: Places the nodes in a single column.

Each pane contains a list for holding nodes in the pane. This list is an instance of **ObservableList**, which can be obtained using pane's *getChildren()* method. You can use *add(node)* to add an element to the list and *addAll(node1, node2, ...)* to add a variable number of nodes.

JavaFX UI Controls

JavaFX provides many UI controls for developing a comprehensive user interface. The following are the frequently used UI controls in detail:

- **Labeled** and **Label**: A label is a display area for a short text, a node, or both. It is often used to label other controls (usually text fields).
- **Button**: A button is a control that triggers an action event when clicked.
- **TextField**: A text field can be used to enter or display a string.
  Note: If a text field is used for entering a password, use **PasswordField** to replace **TextField**.

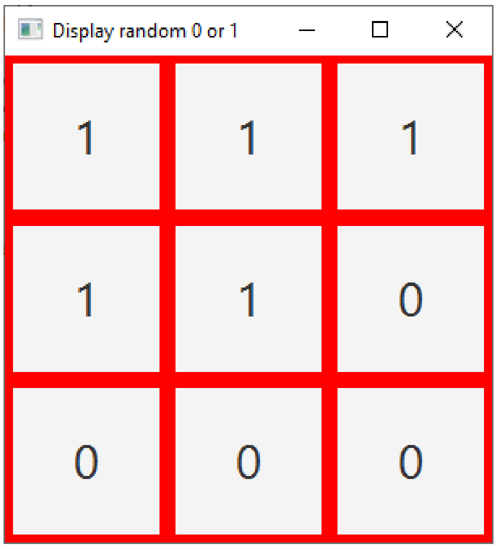
7.3 Pre-Lab

1. How do you set a style of a node with border color red? Modify the code to set the text color for the button to red.
2. What is a binding property? What interface defines a binding property? What interface defines a source object?
3. How do you create a label with a node without a text?
4. Can you display multiple lines of text in a label?
5. How do you create a button with a text and a node? Can you apply all the methods for **Labeled** to **Button**?
6. Can you disable editing of a text field?
7. How do you align the text in a text field to the right?

7.4 Activities

Activity 1:

Write a program that displays a 3-by-3 square matrix, as shown in Figure 14. Each element in the matrix is 0 or 1, randomly generated. Display each number centered in a label. Set the label's font size to 30 and set a red border to each label.

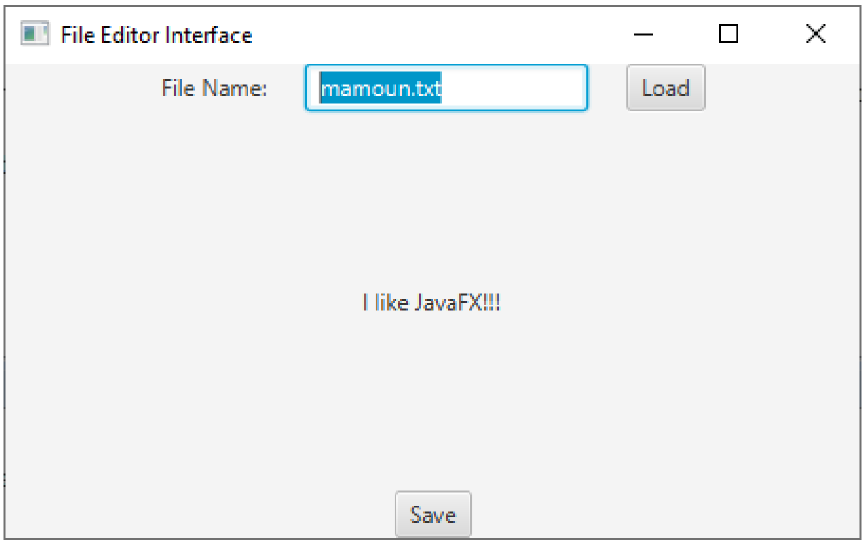


1	1	1
1	1	0
0	0	0

Figure 14: Activity 1

Activity 2:

Write a program that displays a text editor interface as shown in Figure 15.

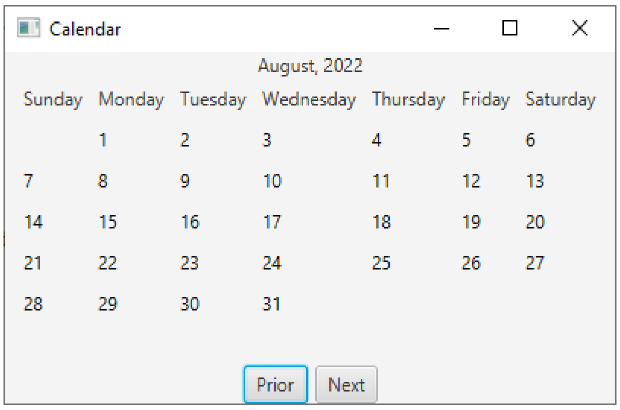


I like JavaFX!!!

Figure 15: Activity 2

Activity 3:

Write a program that displays the calendar for the current month , as shown in Figure 16. Note: refer back to Activity 2 in lab 3.



Calendar

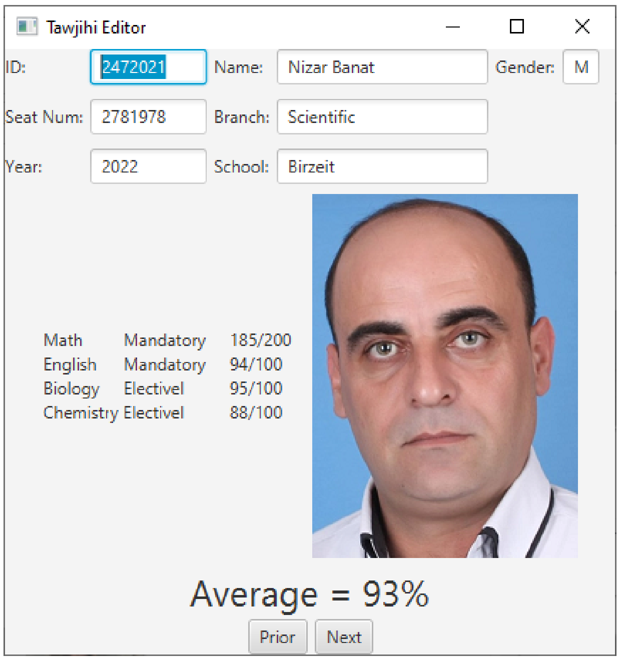
August, 2022

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Figure 16: Activity 3

Activity 4:

Write a program that displays the Tawjihi editor interface, as shown in Figure 17.



Tawjihi Editor

ID: 2472021 Name: Nizar Banat Gender: M

Seat Num: 2781978 Branch: Scientific

Year: 2022 School: Birzeit

```
Math      Mandatory  185/200
English   Mandatory  94/100
Biology   Electivel  95/100
Chemistry Electivel  88/100
```

Average = 93%

Figure 17: Activity 4

8 Using Inner Classes and Lambda Expression

8.1 Objectives

- To get a taste of event-driven programming.
- To describe events, event sources, and event classes.
- To define handler classes, register handler objects with the source object, and write the code to handle events.
- To define handler classes using inner classes.
- To define handler classes using anonymous inner classes.
- To simplify event handling using lambda expressions.

8.2 Context

Event-Driven Programming

To respond to a button click, you need to write the code to process the button-clicking action. The button is an *event source object* where the action originates. You need to create an object capable of handling the action event on a button. This object is called an *event handler*. To be a handler of an action event, two requirements must be met:

1. The object must be an instance of the **EventHandler<T extends Event>** interface. This interface defines the common behavior for all handlers. **<T extends Event>** denotes that **T** is a generic type that is a subtype of **Event**.

2. The **EventHandler** object handler must be registered with the event source object using the method *source.setOnAction(handler)*.

The **EventHandler<ActionEvent>** interface contains the *handle(ActionEvent)* method for processing the action event. Your handler class must override this method to respond to the event.

An event is an object created from an event source. Firing an event means to create an event and delegate the handler to handle the event. The component that creates an event and fires it is called the *event source object*, or simply *source object* or *source component*. The root class of the Java event classes is **java.util.EventObject**.

Registering Handlers and Handling Events

Java uses a delegation-based model for event handling: A source object fires an event, and an object interested in the event handles it. The latter object is called an *event handler* or an *event listener*. For an object to be a handler for an event on a source object, two things are needed:

1. The handler object must be an instance of the corresponding event handler interface to ensure the handler has the correct method for processing the event. For example, the handler interface for **ActionEvent** is **EventHandler<ActionEvent>**.

2. The handler object must be registered by the source object. For **ActionEvent**, the method is *setOnAction*.

Inner Classes

An inner class, or nested class, is a class defined within the scope of another class. Inner classes are useful for defining handler classes. A handler class is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is appropriate to be defined inside the main class as an inner class.

Anonymous Inner-Class Handlers

An anonymous inner class is an inner class without a name. It combines defining an inner class and creating an instance of the class into one step. The syntax for an anonymous inner class is shown below.

```
new SuperClassName/InterfaceName() {
     // Implement or override methods in superclass or interface
     // Other methods if necessary
}
```

Simplifying Event Handling Using Lambda Expressions

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an anonymous class with a concise syntax. The basic syntax for a lambda expression is either:

```
(type1 param1, type2 param2, .   .   .  )  ?> expression
```

or

```
(type1 param1, type2 param2, .   .   .  )  ?> { statements; }
```

for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *Single Abstract Method (SAM)* interface.

Using lambda expressions not only simplifies the syntax, but also simplifies the event-handling concept.

8.3 Pre-Lab

1. What is an event source object?
2. What is an event object?
3. Describe the relationship between an event source object and an event object.
4. Why must a handler be an instance of an appropriate handler interface?
5. Explain how to register a handler object and how to implement a handler interface.
6. What is the handler method for the **EventHandler<ActionEvent>** interface?
7. What is the registration method for a button to register an **ActionEvent** handler?
8. Can an inner class be used in a class other than the class in which it nests?
9. What is a lambda expression? What is the benefit of using lambda expressions for event handling?
10. What is the syntax of a lambda expression?
11. What is a functional interface?
12. Why is a functional interface required for a lambda expression?

8.4 Activities

Activity 1:

Write a program that moves the ball in a pane. You should define a pane class for displaying the ball and provide the methods for moving the ball left, right, up, and down, as shown in Figure 18. Use an outer-class handler to handle the moving events.

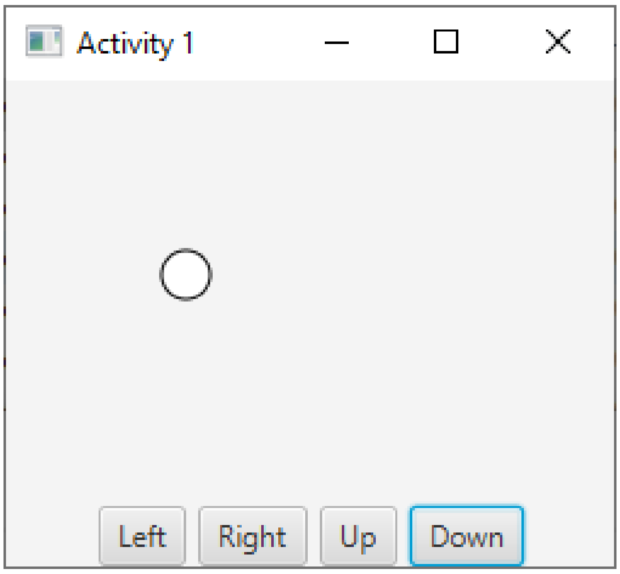


Figure 18: Activity 1

Activity 2:

Write a program to perform addition, subtraction, multiplication, and division, as shown in Figure 19. Use an inner-class handler to handle the calculation events.

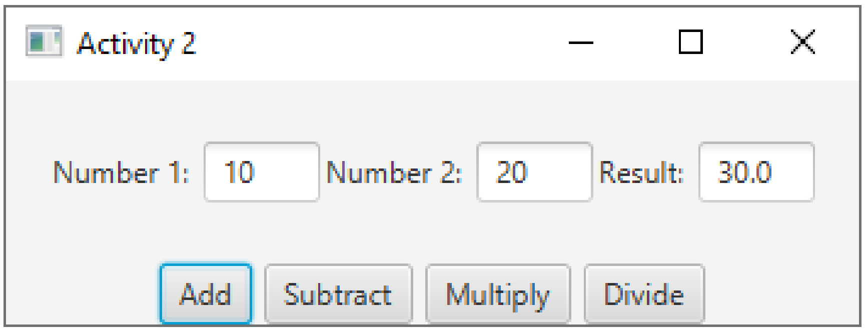


Figure 19: Activity 2

Activity 3:

Re-write Activity 1 in Lab 7 by displaying an empty 3-by-3 square matrix, as shown in Figure 20-a. When the user click on the label, a `1` or `0` will be displayed alternatively, as shown in Figure 20-b. Use anonymous inner-class handler to handle the click events.

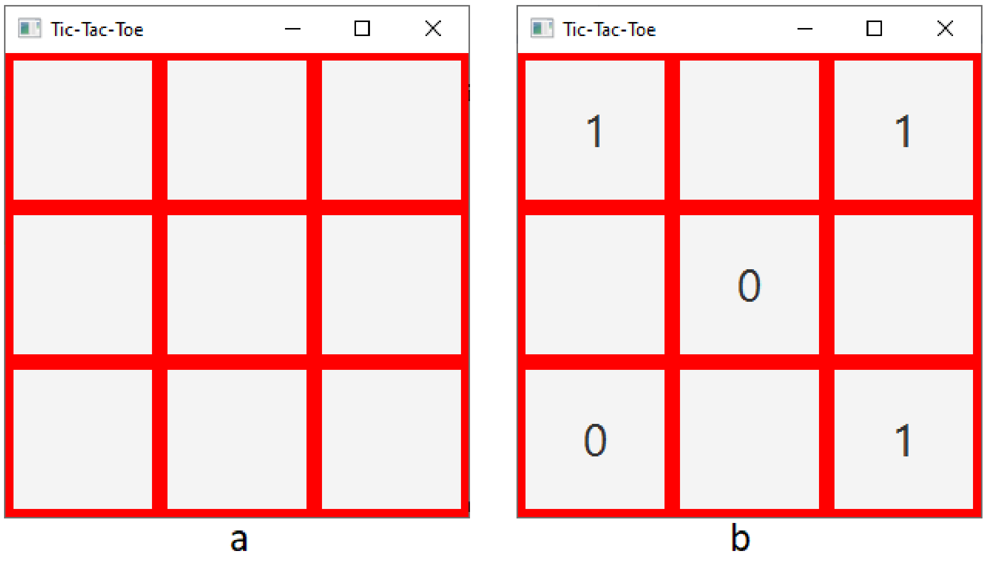


Figure 20: Activity 3

Activity 4:

Re-write Activity 2 in Lab 7 by activating the `Load` and `Save` buttons. Use lambda expressions to handle the click events.

Activity 5:

Re-write Activity 3 in Lab 7 by activating the `Prior` and `Next` buttons.

9 GUI - Event Driven Programming

9.1 Objectives

- To write programs to deal with **MouseEvent**s.
- To write programs to deal with **KeyEvent**s.
- To create listeners for processing a value change in an observable object.

9.2 Context

Mouse Events

The **MouseEvent** object captures the event, such as the number of clicks associated with it, the location (the x- and y-coordinates) of the mouse, or which mouse button was pressed. Four constants (**PRIMARY, SECONDARY, MIDDLE**, and **NONE**) are defined in **MouseButton** to indicate the left, right, middle, and none mouse buttons, respectively. You can use the *getButton()* method to detect which button is pressed. You can also use the *isPrimaryButtonDown()*, *isSecondaryButtonDown()*, and *isMiddleButtonDown()* to test if the primary button, second button, or middle button is pressed.

Key Events

Key events enable the use of the keys to control and perform actions, or get input from the keyboard. The **KeyEvent** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key. Every key event has an associated code that is returned by the *getCode()* method in **KeyEvent**. The key codes are constants defined in **KeyCode**. Only a focused node can receive **KeyEvent**. Invoking *requestFocus()* on text enables text to receive key input.

Listeners for Observable Objects

An instance of **Observable** is known as an *observable object*, which contains the *addListener(InvalidationListener listener)* method for adding a listener. The listener class must implement the functional interface **InvalidationListener** to override the *invalidated(Observable o)* method for handling the value change. Once the value is changed in the **Observable** object, the listener is notified by invoking its *invalidated(Observable o)* method. Every binding property is an instance of **Observable**.

9.3 Pre-Lab

1. Can a button fire a **MouseEvent**? Can a button fire a **KeyEvent**? Can a button fire an **ActionEvent**?
2. What method do you use to get the mouse-point position for a mouse event?
3. What methods do you use to register a handler for `mouse-pressed, -released, -clicked, -entered, -exited, -moved,` and `-dragged` events?
4. What methods do you use to register handlers for `key-pressed, key-released`, and `key-typed` events? In which classes are these methods defined?
5. What method do you use to get the key character for a key-typed event?
6. How do you set focus on a node so it can listen for key events?

9.4 Activities

Activity 1:

Write a program that moves the ball in a pane according to the mouse click location (the x- and y-coordinates), as shown in Figure 21. Use mouse click event to handle the moving events.

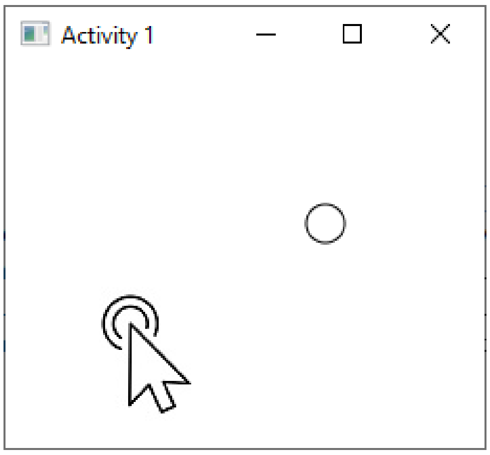


Figure 21: Activity 1

Activity 2:

Re-write Activity 1 to move the ball in a pane according to arrow-keys, as shown in Figure 22. Use key event to handle the moving events.

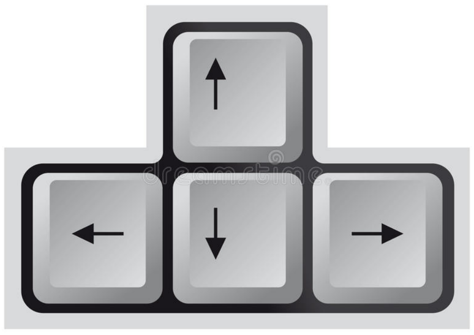


Figure 22: Activity 2

Activity 3:

Write a program that lets the user click on a pane to dynamically create and remove points (see Figure 23). When the user left-clicks the mouse (primary button), a point is created and displayed at the mouse point. The user can remove a point by pointing to it and right clicking the mouse (secondary button).

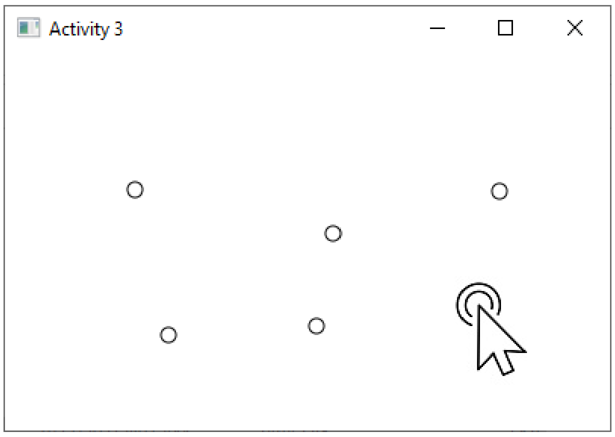


Figure 23: Activity 3

Activity 4:

Write a program that displays two circles with radius 10 at location (40, 40) and (120, 150) with a line connecting the two circles, as shown in Figure 24. The distance between the circles is displayed along the line. The user can drag a circle. When that happens, the circle and its line are moved, and the distance between the circles is updated.

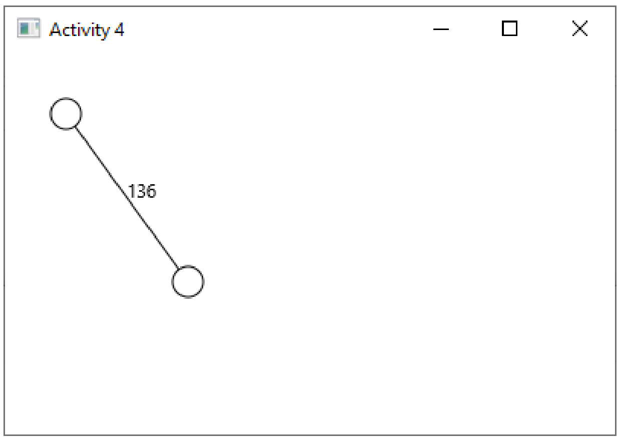


Figure 24: Activity 4

10 GUI - UI controllers

10.1 Objectives

- To create a check box using the **CheckBox** class.
- To create a radio button using the **RadioButton** class, and group radio buttons using a **ToggleGroup**.
- To enter data in multiple lines using the **TextArea** class.
- To select a single item using **ComboBox**.
- To select a single or multiple items using **ListView**.
- To select a range of values using **ScrollBar**.
- To select a range of values using **Slider**.

10.2 Context

CheckBox

A **CheckBox** is used for the user to make a selection. Like **Button**, **CheckBox** inherits all the properties such as *onAction, text, graphic, alignment, graphicTextGap, textFill*, and *contentDisplay* from **ButtonBase** and **Labeled**. In addition, it provides the *selected* property to indicate whether a check box is selected. When a check box is clicked (checked or unchecked), it fires an **ActionEvent**. To see if a check box is selected, use the *isSelected()* method.

RadioButton

Radio buttons, also known as option buttons, enable you to choose a single item from a group of choices. In appearance, radio buttons resemble check boxes, but check boxes display a square that is either checked or blank, whereas radio buttons display a circle that is either filled (if selected) or blank (if not selected). **RadioButton** is a subclass of **ToggleButton**. To group radio buttons, you need to create an instance of **ToggleGroup** and set a radio button's toggleGroup property to join the group. When a radio button is changed (selected or deselected), it fires an **ActionEvent**. To see if a radio button is selected, use the *isSelected()* method.

TextArea

A**TextArea** enables the user to enter multiple lines of text.
Tip: You can place any node in a **ScrollPane**. **ScrollPane** automatically provides vertical and horizontal scrolling if the node is too large to fit in the viewing area as follows:

```
ScrollPane scrollPane = new ScrollPane(textArea);
```

ComboBox

A combo box, also known as a choice list or drop-down list, contains a list of items from which the user can choose. A combo box is useful for limiting a user's range of choices and avoids the cumbersome validation of data input. **ComboBox** is defined as a generic class like the **ArrayList** class. The generic type **T** specifies the element type for the elements stored in a combo box. **ComboBox** can fire an **ActionEvent**. Whenever an item is selected, an **ActionEvent** is fired.

ListView

A list view is a control that basically performs the same function as a combo box, but it enables the user to choose a single value or multiple values. The *getSelectionModel()* method returns an instance of **SelectionModel**, which contains the methods for setting a selection mode and obtaining selected indices and items. The selection mode is defined in one of the two constants **SelectionMode.MULTIPLE** and **SelectionMode.SINGLE**, which indicates whether a single item or multiple items can be selected. The default value is **SelectionMode.SINGLE**.

ScrollBar

ScrollBar is a control that enables the user to select from a range of values. Normally, the user changes the value of a scroll bar by making a gesture with the mouse. For example, the user can drag the scroll bar's thumb, click on the scroll bar track, or the scroll bar's left or right buttons. When the user changes the value of the scroll bar, it notifies the listener of the change. You can register a listener on the scroll bar's *valueProperty* for responding to this change.

Slider

Slider is similar to **ScrollBar**, but **Slider** has more properties and can appear in many forms. **Slider** lets the user graphically select a value by sliding a knob within a bounded interval. The slider can show both major and minor tick marks between them. The number of pixels between the tick marks is specified by the *majorTickUnit* and *minorTickUnit* properties. Sliders can be displayed horizontally or vertically, with or without ticks, and with or without labels. You can add a listener to listen for the value property change in a slider in the same way as in a scroll bar.

10.3 Pre-Lab

1. How do you test if a check box is selected?
2. How do you test if a radio button is selected?
3. How do you group radio buttons?
4. How do you obtain the text from a text area?
5. How do you create a combo box and add three items to it?
6. How do you retrieve an item from a combo box? How do you retrieve a selected item from a combo box?
7. What events would a **ComboBox** fire upon selecting a new item?
8. What selection modes are available for a list view? What is the default selection mode? How do you set a selection mode?
9. How do you create a horizontal scroll bar? How do you create a vertical scroll bar?
10. How do you create a horizontal slider? How do you create a vertical slider?

10.4 Activities

Activity 1:

Write a program that simulates a traffic light. The program lets the user select one of three lights: red, yellow, or green. When a check box is selected, the light is turned on. (see Figure 25).

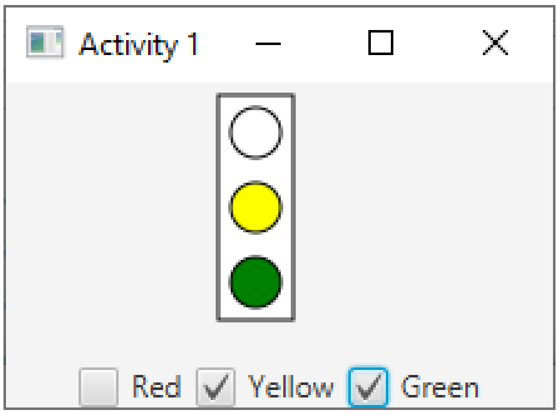


Figure 25: Activity 1

Activity 2:

Re-write Activity 1 so that the user select one of three lights: red, yellow, or green. When a radio button is selected, the light is turned on. Only one light can be on at a time (see Figure 26).

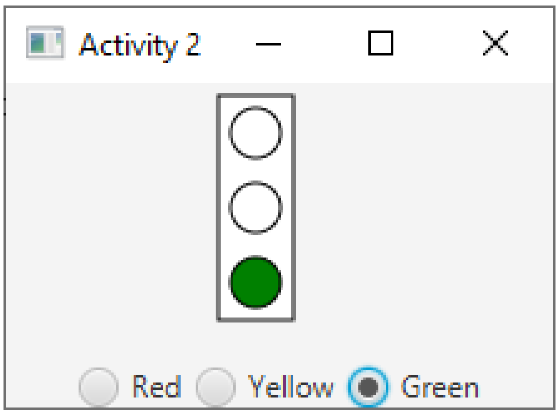
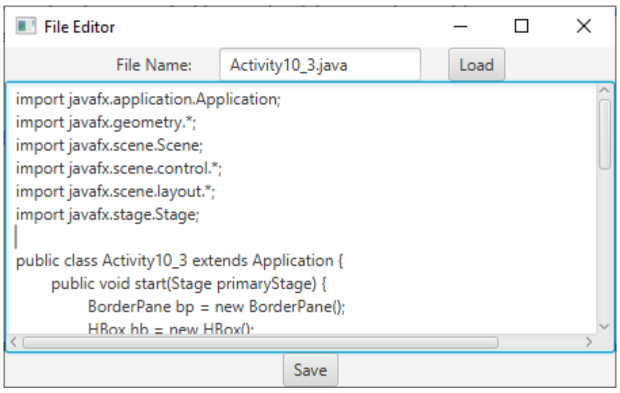


Figure 26: Activity 2

Activity 3:

Re-write Activity 2 from Lab 7 that displays a text editor interface as shown in Figure 27. Activate the `Load` and `Save` buttons.



File Name: Activity10_3.java

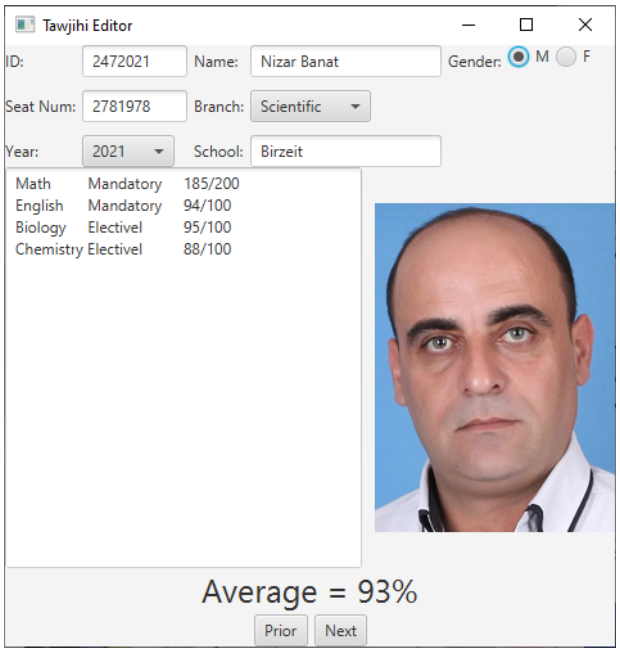
```
import javafx.application.Application;
import javafx.geometry.*;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

public class Activity10_3 extends Application {
      public void start(Stage primaryStage) {
            BorderPane bp = new BorderPane();
            HBox hb = new HBox();
```

Figure 27: Activity 3

Activity 4:

Re-write Activity 4 from Lab 7 that displays a tawjihi editor interface as shown in Figure 28. Activate the `Prior` and `Next` buttons. Consider reading the data from the binary file Tawjihi.dat that we used in Lab 5 - Activity 6.



ID: 2472021 Name: Nizar Banat Gender: [✓ M] F

Seat Num: 2781978 Branch: Scientific

Year: 2021 School: Birzeit

```
Math      Mandatory  185/200
English   Mandatory  94/100
Biology   Electivel  95/100
Chemistry Electivel  88/100
```

Average = 93%

Figure 28: Activity 4

11 GUI - Advanced UI controllers and MVC

11.1 Objectives

- To create tab panes using the **TabPane** control.
- To create and display tables using the **TableView** and **TableColumn** classes.

11.2 Context

TabPane

TabPane is a useful control that provides a set of mutually exclusive tabs, as shown in Figure 29. You can switch between a group of tabs. Only one tab is visible at a time. A Tab can be added to a **TabPane**. Tabs in a **TabPane** can be placed in the position top, left, bottom, or right. Each tab represents a single page. Tabs are defined in the **Tab** class. Tabs can contain any Node such as a pane, a shape, or a control.

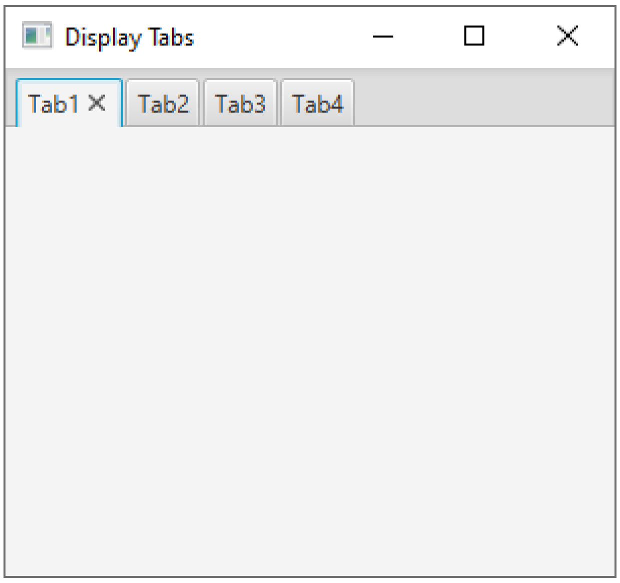
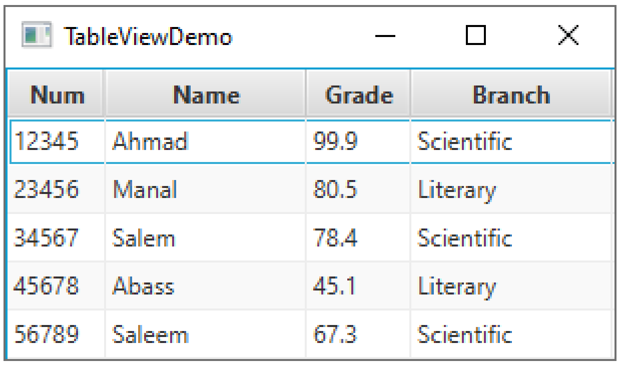


Figure 29: **TabPane** holds a group of 4 tabs

TableView

TableView is a control that displays data in rows and columns in a two-dimensional grid, as shown in Figure 30.



Num	Name	Grade	Branch
12345	Ahmad	99.9	Scientific
23456	Manal	80.5	Literary
34567	Salem	78.4	Scientific
45678	Abass	45.1	Literary
56789	Saleem	67.3	Scientific

Figure 30: **TableView** displays data in a table.

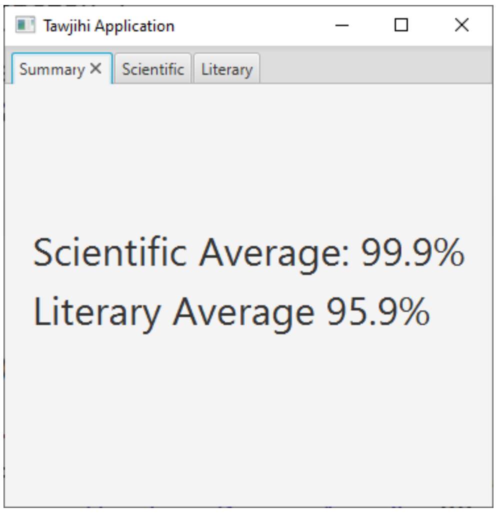
11.3 Pre-Lab

1. How do you create a tab pane? How do you create a tab? How do you add a tab to a tab pane?
2. How do you place the tabs on the left of the tab pane?
3. Can a tab have a text as well as an image?
4. How do you create a table view? How do you create a table column? How do you add a table column to a table view?
5. What is the data type for a **TableView**'s data model? How do you associate a data model with a **TableView**?
6. How do you set a cell value factory for a **TableColumn**?

11.4 Activities

Activity 1:

Write a program to load and display Tawjihi recorders in multi-tab application as shown in Figure 31.

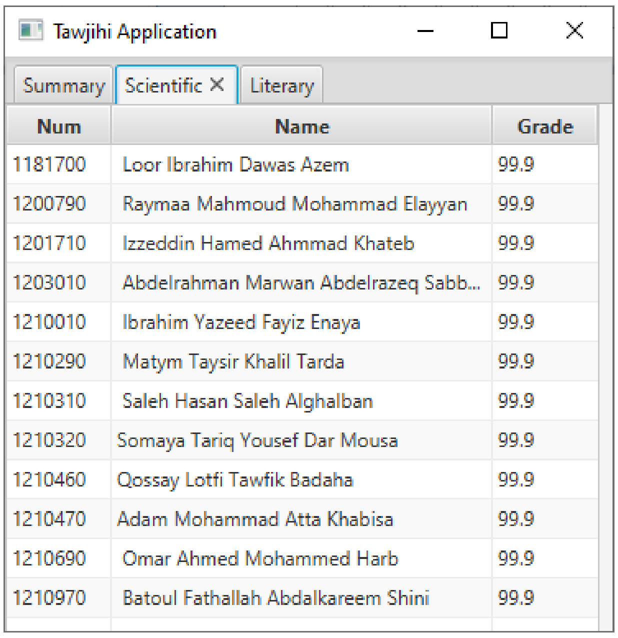


Scientific Average: 99.9%

Literary Average 95.9%

Figure 31: Activity 1 (a)

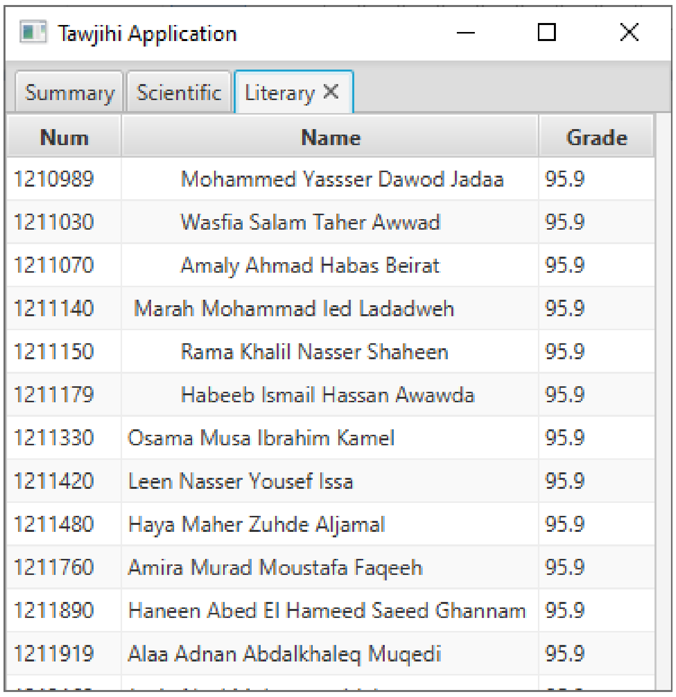
The first tab shows the summary of each branch as shown in Figure 31. The second tab shows a table containing all the scientific students as shown in Figure 32.



Num	Name	Grade
1181700	Loor Ibrahim Dawas Azem	99.9
1200790	Raymaa Mahmoud Mohammad Elayyan	99.9
1201710	Izzeddin Hamed Ahmmad Khateb	99.9
1203010	Abdelrahman Marwan Abdelrazeq Sabb...	99.9
1210010	Ibrahim Yazeed Fayiz Enaya	99.9
1210290	Matym Taysir Khalil Tarda	99.9
1210310	Saleh Hasan Saleh Alghalban	99.9
1210320	Somaya Tariq Yousef Dar Mousa	99.9
1210460	Qossay Lotfi Tawfik Badaha	99.9
1210470	Adam Mohammad Atta Khabisa	99.9
1210690	Omar Ahmed Mohammed Harb	99.9
1210970	Batoul Fathallah Abdalkareem Shini	99.9

Figure 32: Activity 1 (b)

The third tab shows a table containing all the Literary students as shown in Figure 33.



Num	Name	Grade
1210989	Mohammed Yassser Dawod Jadaa	95.9
1211030	Wasfia Salam Taher Awwad	95.9
1211070	Amaly Ahmad Habas Beirat	95.9
1211140	Marah Mohammad Ied Ladadweh	95.9
1211150	Rama Khalil Nasser Shaheen	95.9
1211179	Habeeb Ismail Hassan Awawda	95.9
1211330	Osama Musa Ibrahim Kamel	95.9
1211420	Leen Nasser Yousef Issa	95.9
1211480	Haya Maher Zuhde Aljamal	95.9
1211760	Amira Murad Moustafa Faqeeh	95.9
1211890	Haneen Abed El Hameed Saeed Ghannam	95.9
1211919	Alaa Adnan Abdalkhaleq Muqedi	95.9

Figure 33: Activity 1 (c)

12 Multithreading and Parallel Programming

12.1 Objectives

- To get an overview of multithreading.
- To develop task classes by implementing the **Runnable** interface.
- To create threads to run tasks using the **Thread** class.
- To control threads using the methods in the **Thread** class.
- To control animations using threads and use **Platform.runLater** to run the code in the application thread.

12.2 Context

Thread Concepts

A thread provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems. In single-processor systems the multiple threads share CPU time, known as `time sharing`, and the operating system is responsible for scheduling and allocating resources to them. Multithreading can make your program more responsive and interactive as well as enhance performance. You can create additional threads to run concurrent tasks in the program. In Java, each task is an instance of the **Runnable** interface, also called a `runnable object`. A thread is essentially an object that facilitates the execution of a task.

Creating Tasks and Threads

Tasks are objects. To create tasks, you have to first define a class for tasks, which implements the **Runnable** interface. The **Runnable** interface is rather simple. All it contains is the `run()` method. You need to implement this method to tell the system how your thread is going to run. A template for developing a task class is shown in Figure 34.

Once you have defined a **TaskClass**, you can create a task using its constructor. For example:

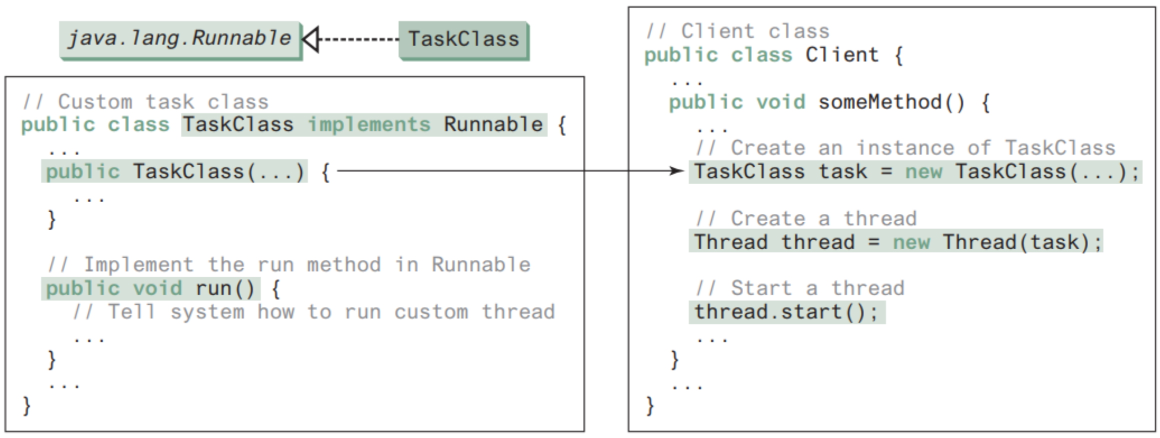
```
TaskClass task = new TaskClass(...);
```

A task must be executed in a thread. The **Thread** class contains the constructors for creating threads and many useful methods for controlling threads. To create a thread for a task, use:

```
Thread thread = new Thread(task);
```

You can then invoke the *start()* method to tell the JVM that the thread is ready to run, as follows:

```
thread.start();
```



```
// Custom task class
public class TaskClass implements Runnable {
  ...
  public TaskClass(...) {
    ...
  }

  // Implement the run method in Runnable
  public void run() {
    // Tell system how to run custom thread
    ...
  }
  ...
}
```

```
// Client class
public class Client {
  ...
  public void someMethod() {
    ...
    // Create an instance of TaskClass
    TaskClass task = new TaskClass(...);

    // Create a thread
    Thread thread = new Thread(task);

    // Start a thread
    thread.start();
    ...
  }
  ...
}
```

Figure 34: Define a task class by implementing the **Runnable** interface.

The Thread Class

The **Thread** class contains the constructors for creating threads for tasks and the methods for controlling threads. Table 5 shows the class diagram for the **Thread** class. The **Thread** class has the int constants **MIN_PRIORITY, NORM_PRIORITY**, and **MAX_PRIORITY**, representing 1, 5, and 10, respectively. The priority of the main thread is **Thread.NORM_PRIORITY**.

Table 5: The **Thread** class contains the methods for controlling threads.

Method	Description
+Thread()	Creates an empty Thread.
+Thread(task: Runnable)	Creates a Thread for a specified task.
+start(): void	Starts the thread that causes the `run()` method to be invoked.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts a thread to sleep for a specified time in milliseconds.
+yield(): void	Causes a thread to pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

Animation Using Threads and the Platform

You can use a thread to control an animation and run the code in JavaFX GUI thread using the *Platform.runLater* method.

12.3 Pre-Lab

1. Why is multithreading needed? How can multiple threads run simultaneously in a single-processor system?
2. What is a runnable object? What is a thread?
3. How do you define a task class? How do you create a thread for a task?
4. Which of the following methods are instance methods in java.lang.Thread?

   ```
   run, start, stop, suspend, resume, sleep, interrupt, yield, join
   ```

5. How do you set a priority for a thread? What is the default priority?
6. What is the purpose of using *Platform.runLater*?

12.4 Activities

Activity 1:

Write a program that simulates a bouncing ball as shown in Figure 35.

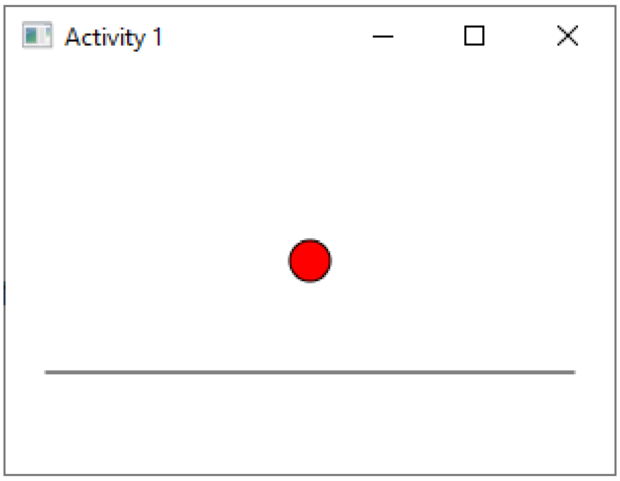


Figure 35: Activity 1

Activity 2:

Write a program that simulates 3 bouncing balls as shown in Figure 36. Run each ball in a separate thread.

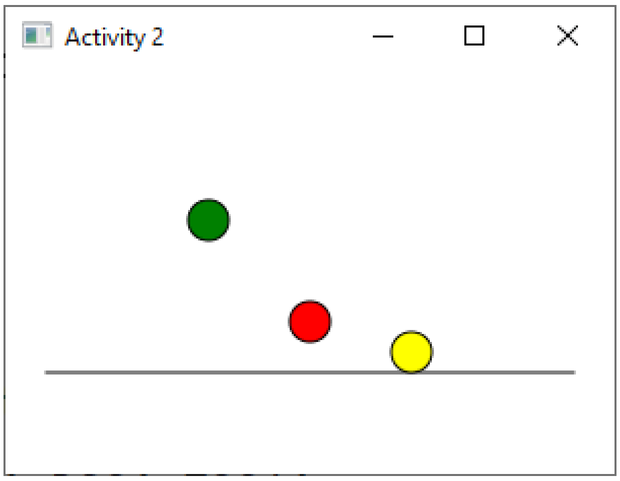


Figure 36: Activity 2

Activity 3:

Write a program that simulates a racing car as shown in Figure 37. The car starts moving forward when the `Start` button is clicked and stops when the Stop button is clicked.

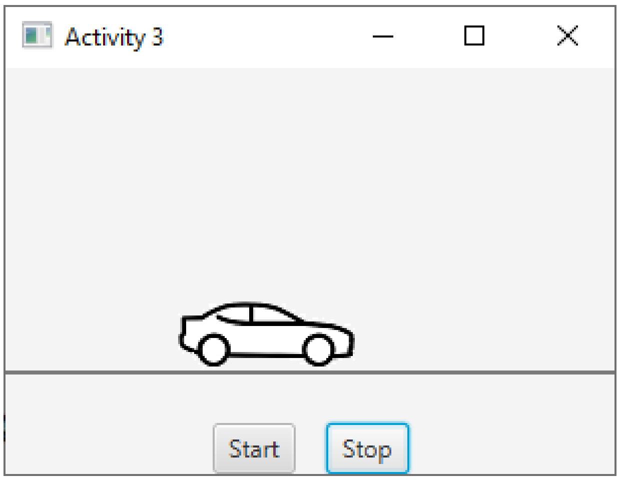


Figure 37: Activity 3

Activity 4:

Write a program to simulate the famous Flappy Bird game as shown in Figure 38. The pipes are moving backward and the bird is moving up and down using the up-key and down-key events.

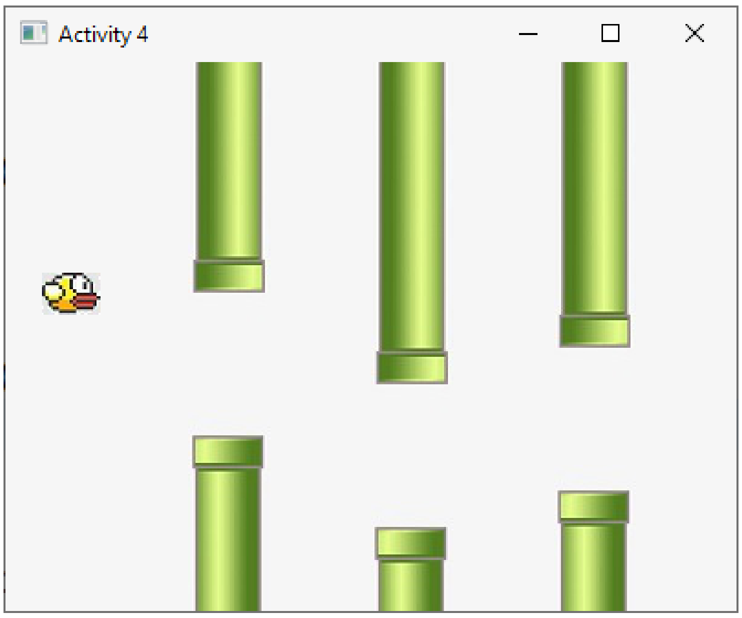


Figure 38: Activity 4

13 Java Collections and Generic Types

13.1 Objectives

- To describe the benefits of generics.
- To define and use generic classes and interfaces.
- To explain why generic types can improve reliability and readability.
- To define and use generic methods and bounded generic types.
- To use the common methods defined in the **Collection** interface for operating collections.
- To use the static utility methods in the **Collections** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections.

13.2 Context

Defining Generic Classes and Interfaces

Java has allowed you to define generic classes, interfaces, and methods since JDK 1.5. For example the **java.lang.Comparable<T>** interface. Here, `<T>` represents a formal generic type, which can be replaced later with an actual concrete type.

A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

Note: Occasionally, a generic class may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commasâ€”for example, `<E1, E2, E3>`.

Generic Methods

A generic type can be defined for a static method. To declare a generic method, you place the generic type `<E>` immediately after the keyword *static* in the method header. For example,

```
public static <E> void print(E[] list)
```

To invoke a generic method, prefix the method name with the actual type in angle brackets.

Collections

The **Collection** interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

- Sets store a group of non-duplicate elements.
- Lists store an ordered collection of elements.
- Stacks store objects that are processed in a last-in, first-out fashion.
- Queues store objects that are processed in a first-in, first-out fashion.
- PriorityQueues store objects that are processed in the order of their priorities.

The **Collection** interface provides the basic operations for adding and removing elements in a collection. The *add* method adds an element to the collection. The *addAll* method adds all the elements in the specified collection to this collection. The *remove* method removes an element from the collection. The *removeAll* method removes the elements from this collection that are present in the specified collection. The *clear()* method simply removes all the elements from the collection.

The *size* method returns the number of elements in the collection. The *contains* method checks whether the collection contains the specified element. The *containsAll* method checks whether the collection contains all the elements in the specified collection. The *isEmpty* method returns `true` if the collection is empty.

Lists

The **List** interface extends the **Collection** interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete classes: **ArrayList** or **LinkedList**. The *add(index, element)* method is used to insert an element at a specified index and the *addAll(index, collection)* method to insert a collection of elements at a specified index. The *remove(index)* method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the *set(index, element)* method. The *indexOf(element)* method is used to obtain the index of the specified element's first occurrence in the list and the *lastIndexOf(element)* method to obtain the index of its last occurrence. A sublist can be obtained by using the *subList(fromIndex, toIndex)* method.

ArrayList stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array. **LinkedList** stores elements in a linked list.

Static Methods for Lists and Collections

The **Collections** class contains the *sort, binarySearch, reverse, shuffle, copy*, and *fill* methods for lists and *max, min, disjoint,* and *frequency* methods for collections.

13.3 Pre-Lab

1. What are the benefits of using generic types?
2. How do you declare a generic type in a class?
3. What are the differences between **ArrayList** and **LinkedList**?
4. Which method can you use to sort the elements in an **ArrayList** or a **LinkedList**?

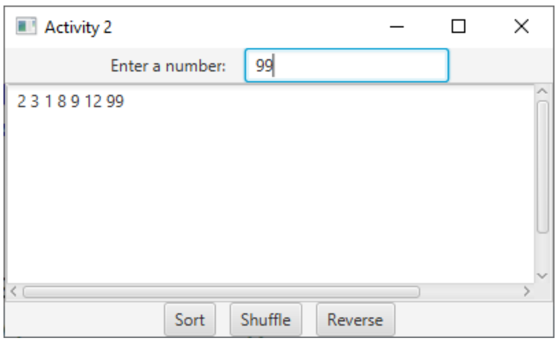
13.4 Activities

Activity 1:

Write a program that reads words from a text file and displays all the words (duplicates allowed) in ascending alphabetical order. The words must start with a letter.

Activity 2:

Write a program that lets the user enter numbers from a graphical user interface and displays them in a text area, as shown in Figure 39. Use a linked list to store the numbers. Do not store duplicate numbers. Add the buttons Sort, Shuffle, and Reverse to sort, shuffle, and reverse the list.



```
2 3 1 8 9 12 99
```

Figure 39: Activity 2

Activity 3:

A Java program contains various pairs of grouping symbols, such as:

- Parentheses: (and)
- Braces: { and }
- Brackets: [and]

Note the grouping symbols cannot overlap. For example, (a{b)} is illegal. Write a program to check whether a Java source-code file has correct pairs of grouping symbols.