

Ahmadu Bello University, Zaria

Department of Mathematics

Laboratory Manual for  
COSC 212: Object Oriented Programming II

Adapted from

ICS 201 Labs of

King Fahd University of Petroleum & Minerals

Dhahran, Saudi Arabia

May, 2011

## Lab 0: Revision

Implement a class Student. A student has an ID and GPA. Supply a constructor that has ID and GPA and another constructor that has ID only. Supply a method to return ID, a method to return GPA, a method to set GPA, and the toString method. Create an array of students using the above class. Read the information of the students from the file "student.txt" and store them in the array. The first line in the file is the number of students. Display the list of students followed by:

- Average GPA
- Percentage of students above average

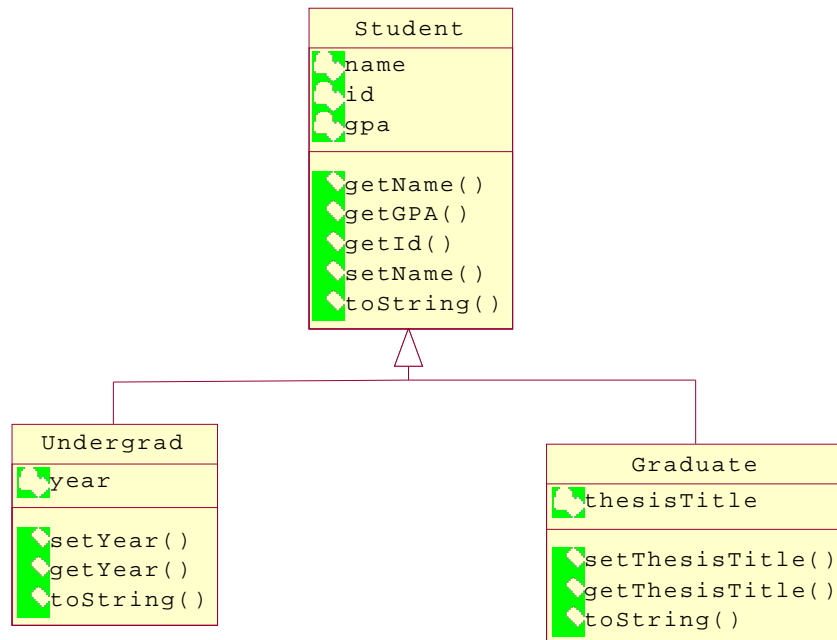
# Lab 01: Inheritance

**Objectives:** In this lab, the following topics will be covered

1. Inheritance (in Java),
2. Exercises for practice
3. Shadowing, Overriding, and Hiding

## 1. Inheritance (in Java):

Inheritance is an important object-oriented concept that allows classes to be reused in order to define similar, but distinct, classes. In this lab we walk through the development of a class hierarchy and a program that makes use of several classes in the hierarchy. We begin by looking at an example of inheritance hierarchy



The class Student is the parent class. Note that all the variables are private and hence the child classes can only use them through accessor and mutator methods. Also note the use of **overloaded** constructors.

```
public class Student{
    private String name;
    private int id;
    private double gpa;

    public Student(int id, String name, double gpa) {
        this.id = id;
        this.name = name;
        this.gpa = gpa;
    }

    public Student(int id, double gpa){
        this(id, "", gpa);
    }

    public String getName(){
        return name;
    }
    public int getId() {
        return id;
    }
    public double getGPA(){
        return gpa;
    }

    public void setName(String newName){
        this.name = newName;
    }
    public String toString(){
        return "Student: \nID: "+id+"\nName: "+name+"\nGPA: "+gpa;
    }
}
```

The class Undergrad **extends** the Student class. Note the **overridden** toString() method

```
public class Undergrad extends Student
{
    private String year;
    public Undergrad(int id, String name, double gpa, String year)
    {
        super(id, name, gpa);
        this.year = year;
    }

    public String getYear() {
        return year;
    }

    public void setYear(String newYear) {
        this.year = newYear;
    }

    public String toString() {
        return "Undergraduate "+super.toString()+"\nYear: "+year;
    }
}
```

```

public class Graduate extends Student
{
    private String thesisTitle;
    public Graduate(int id, String name, double gpa, String thesisTitle)
    {
        super(id, name, gpa);
        this.thesisTitle = thesisTitle;
    }

    public String getthesisTitle() {
        return thesisTitle;
    }

    public void setThesisTitle(String newthesisTitle) {
        this.thesisTitle = newthesisTitle;
    }

    public String toString() {
        return "Graduate " +super.toString()+"\nThesis: "+thesisTitle;
    }
}

```

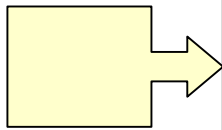
TestStudents is a driver class to test the above classes

```

public class TestStudents
{
    public static void main(String[] args)
    {
        Student s1 = new Student(97000, "Sameer", 3.51);
        Student s2 = new Student(98000, 3.22);
        Undergrad u1 = new Undergrad(99000, "Shahid", 2.91, "Junior");
        Graduate g1 = new Graduate(200000, "Mubin", 3.57,
                                "Algorithms and Complexity");

        System.out.println(s1);
        System.out.println(s2);
        System.out.println(u1);
        System.out.println(g1);
    }
}

```



```

Student:
ID: 97000
Name: Sameer
GPA: 3.51
Student:
ID: 98000
Name:
GPA: 3.22
Undergraduate Student:
ID: 99000
Name: Shahid
GPA: 2.91
Year: Junior
Graduate Student:
ID: 200000
Name: Mubin
GPA: 3.57
Thesis: Algorithms and Complexity
Press any key to continue...

```

### Exercise1:

Consider a superclass PurchaseItem which models customer's purchases. This class has:

- two private instance variables name (String) and unitPrice (double).
- One constructor to initialize the instance variables.
- A default constructor to initialize name to "no item", and unitPrice to 0. use this()
- A method getPrice that returns the unitPrice.
- Accessor and mutator methods.
- A toString method to return the name of the item followed by @ symbol, then the unitPrice.

Consider two subclasses WeighedItem and CountedItem. WeighedItem has an additional instance

variable weight (double) in Kg while CountedItem has an additional variable quantity (int) both private.

- Write an appropriate constructor for each of the classes making use of the constructor of the superclass in defining those of the subclasses.

- Override getPrice method that returns the price of the purchased item based on its unit price and weight (WeighedItem), or quantity (CountedItem). Make use of getPrice of the superclass

- Override also toString method for each class making use of the toString method of the superclass in defining those of the subclasses.

toString should return something that can be printed on the receipt.

For example

Banana @ 3.00 1.37Kg 4.11 SR (in case of WeighedItem class)

Pens @ 4.5 10 units 45 SR (in case of CountedItem class)

Write an application class where you construct objects from the two subclasses and print them on the screen.

## 2. Shadowing, Overriding, Hiding

Suppose you're reading some Java code, and you come across something like this:

```
class A {
    int A = 37;
    A() {
        int A = 47;
        A aref = new A() {
            int A = 57;
            void A() {}
        };
    }
}
```

This usage is legal, but not necessarily desirable. In fact, it raises an interesting question about how the Java programming language specification treats conflicting names. There are several terms used to describe various cases: [shadowing](#), [overriding](#), and [hiding](#)

First an important point needs to be made: just because the Java programming language allows you to do something, it doesn't always mean that it's a desirable thing to do. For example, it's legal to say:

```
class A {
    int A;
}
```

in a program, but you probably shouldn't because it's confusing. The best way to handle issues with conflicting names is to simply avoid them as far as possible. For example, you can avoid many problems if you follow a coding convention that specifies that the first letter of a type name (such as "class A") should be capitalized, while the first letter of a field name (such as "int A") should be lowercase.

Now let's look at an example of [shadowing](#):

```
public class Shadow {
    int a;
    int b;
    // parameters a/b shadow instance variables a/b
    public Shadow(int a, int b) {
        // set parameter equal to itself
        a = a;
        // set instance variable b equal to parameter b
        this.b = b;
    }
    public static void main(String args[]) {
        Shadow s = new Shadow(37, 47);
        System.out.println("a = " + s.a);
        System.out.println("b = " + s.b);
    }
}
```

When you run Shadow, you should see:

```
a = 0
b = 47
```

One place shadowing comes up is when you have field names and parameter names that are the same, and you want to use the parameters to set the fields:

```
int a;
public void f(int a) {
    a = a;
}
```

This doesn't work, because **the parameter "a" shadows the field "a"**, that is, the parameter name blocks access via a simple name to the field name. You can get around this problem by saying:

```
this.a = a;
```

which means "set field a to parameter a". Whether this style of usage is desirable or not depends on your particular biases; one point in its favor is that you don't have to invent parameter names like "a1" or "\_a".

The second example is one that illustrates **overriding**:

```
class B {
    void f() {
        System.out.println("B.f");
    }
}
public class Override extends B {
    // instance method f overrides instance method B.f
    void f() {
        System.out.println("Override.f");
    }
    void g() {
        // call Override.f
        f();
        // call B.f
        super.f();
    }
    public static void main(String args[]) {
        Override o = new Override();
        o.g();
    }
}
```

When you run Override, you should see:

```
Override.f
B.f
```

In this example, the method `Override.f` overrides the method `B.f`. If you have an object of type `Override`, and call `f`, `Override.f` is called. However if you have an object of type `B`, `B.f` is called. This approach is a standard part of object-oriented programming. For example, `java.lang.Object` declares a `hashCode` method, but subclasses, such as `String`, provide an overriding version of the method. The overriding version is tailored to the particular type of data represented by the class.

You can call the superclass method by using the notation:

```
super.f();
```

A third example is that of **hiding**

```

class A {
    static void f() { // static method
        System.out.println("A.f");
    }
    void g() { // instance method
        System.out.println("A.g");
    }
}

public class SubA extends A {
    static void f() { // static methods are not overridden
        System.out.println("Hide.f");
    }
    void g() {
        System.out.println("Hide.g");
    }
    public static void main(String args[]) {
        A aref = new SubA();
        // call A.f()
        aref.f();
        // call SubA.g()
        aref.g();
    }
}

```

When you run Hide, you should see:

```

A.f
Hide.g

```

In this example, Hide.f hides A.f, and Hide.g overrides A.g. One way of seeing the difference between hiding and overriding is to note that overriding applies to regular instance methods; the actual method that is called is determined at run time based on the type of the actual object. This sort of dynamic lookup does not happen for static methods or for fields. For example, in this code:

```

class C {
    int x = 37;
    void f() {
        System.out.println("C. f");
    }
}

public class Lookup extends C{
    int x = 47;
    void f() {
        System.out.println("Lookup. f");
    }
    public static void main(String args[]) {
        C cref = new Lookup();
        // call Lookup.f
        cref.f();
        // display C.x
        System.out.println(cref.x);
    }
}

```

the method reference through "cref" results in Lookup.f being called, but the field reference obtains C.x. Or to say it another way, the actual class of an object determines which instance method is called. But for fields and static methods, the type of the reference is used (here it's cref, of type C). When you run Lookup, you should see:

```

Lookup.f
37

```



## Exercise 2:

Use the sample Java programs provided above and modify them to answer the following questions.

- a- Can an instance method override a static method?
- b- Can a static method override (hide) an instance method?
- c- Can you override a final instance method?
- d- Can you override an instance method and make it final?
- e- Can you override an instance method and change its return type?
- f- Can you hide a final static method ?
- g- Can an instance field hide a static field?
- h- Can a static field hide an instance field?
- i- Can an instance method with public visibility override an instance method with default visibility?
- j- Can an instance method with default visibility override an instance method with public visibility?
- k- Can an instance method with protected visibility override an instance method with default visibility?
- l- Can an instance method with default visibility override an instance method with protected visibility?
- m- Based on the last four question, order the access visibility from the widest to the narrowest (weakest) and state the rule for overriding (instance methods) or hiding (static methods) ?

## Lab 02: Abstract classes and Interfaces

### Objectives:

To gain experience with:

- Abstract Classes and their purpose
- Interfaces and their purpose

### 1. Abstract Classes.

When an existing class is extended, the subclass programmer has the option of whether or not to override the methods of the super class. Sometimes we may wish to force the subclass to override a method. This is usually the case if there is no good default method in the super class or when the method implementation becomes meaningful only at the subclass level. To achieve this, we declare the methods we wish to force the subclass to override as **abstract**. A class that has one or more abstract methods is called an **abstract class** and must be declared as such.

#### Notice that:

- We cannot create an instance of an abstract class. However, we can declare a reference variable of abstract class.
- The principles of inheritance, up-casting and polymorphisms apply equally well to abstract classes.

Consider The classes covered in Lab 1, i.e. PurchaseItem, CountedItem, and WeighedItem. The method getPrice() which was supposed to return the total price was implemented in the superclass PurchaseItem to return the unit price. Now if the programmer of one of the subclasses forgets to override getPrice() method, it will return a wrong answer, i.e. a unit price instead of the total price. A better solution will be to declare the method getPrice() as abstract in the superclass PurchaseItem as shown below.

Now in order for the two subclasses CountedItem and WeighedItem to be concrete classes (i.e. they can be instantiated), they have to implement getPrice(). This way we are sure that the subclass will override getPrice() and provide a meaningful implementation for it.

```

public abstract class PurchaseItem {
    private String name;
    private double unitPrice;
    public PurchaseItem (String name, double unitPrice) {
        this.name=name;
        this.unitPrice=unitPrice;
    }
    public PurchaseItem () {
        this("no item",0);
    }
    public String getName() {
        return name;
    }
    public void setName (String name){
        this.name=name;
    }
    public double getUnitPrice() {
        return unitPrice;
    }
    public abstract double getPrice() ;
    public String toString() {

```

### Exercise 1:

Consider the following shapes; Ellipse, Circle, Triangle, EquilateralTriangle. Each shape should have a name, a method to compute its perimeter, and another method to compute its area. The name should be an instance variable of type String. Design your inheritance hierarchy with the common features in the superclass Shape. Notice that the area and perimeter are common to all Shapes, but we don't know how to compute the area or perimeter for a general shape.

The ellipse class has a major and minor axes a and b, respectively. The constructor should assign the largest value to a and smallest to b. The area and perimeters of an ellipse are:

$$\text{Perimeter} = P = \pi \sqrt{2(a^2 + b^2) - (a - b)^2} / 2 \quad [\text{Note that if } a = b = r, \text{ then } P = 2\pi r]$$

$$\text{Area} = A = \pi ab$$

The Triangle class has three instance variables side1, side2, and side3. The formula for the area and perimeter of a general Triangle with sides A, B, and C is given by.

$$\text{AREA} = \sqrt{S(S - A)(S - B)(S - C)}$$

$$\text{PERIMETER} = A + B + C$$

$$S = \frac{A + B + C}{2}$$

The condition for any three positive values to make sides of a Triangle is:

side1+side2>side3 and side2+side3>side1 and side3+side1>side2

You need to check this condition inside the constructor. If it is not satisfied, print an error message and terminate the program, otherwise make your Triangle object.

The three sides of the equilateral triangle are equal.

Make a Test class where you make objects from the different classes and store them in an array of type Shape. Then, make a loop and print the objects name, area, and perimeter through toString i.e. you need to override toString in the Shape class only.

## 2. Interfaces

Some OO languages such as C++ allow a sub-class to inherit from more than one super class (multiple inheritance). While this has some advantages, it makes such languages complex. To avoid such complexities, Java does not allow for multiple inheritance. However, a lot of the advantages of multiple inheritance can be achieved using **Interfaces**.

An interface is similar to a class but with the following restrictions:

- All methods are implicitly **abstract** and **public**
- An interface cannot have instance variables. However, an Interface may have constants (final variables) and these are implicitly public and static. Also they are inherited by any class that implements the interface.
- An Interface can extend another interface and it is implemented by a class using the **implements** keyword. In fact, a class may implement any number of interfaces.

### Exercise 2:

Consider an interface Scalable with a method scale of type void. It takes the scaling factor as a parameter. Make the shape class defined above implement the Scalable interface. Note that since Shape is abstract, it does not have to implement scale method.

Make the appropriate subclasses override scale method by multiplying their instance variables by the scale factor.

Modify the above Test class so that you add a static method that receives an array of Type Scalable, and a scale factor. This method should visit all the elements of the Scalable array and call the scale method with the scale factor passed to the static method. You should print your objects before and after scaling.

## Lab 03: Polymorphism

**Objectives:** In this lab, the following topics will be covered

4. Polymorphism
5. upcasting and downcasting
6. instanceof operator

### What is polymorphism?

The meaning of the word polymorphism is something like *one name, many forms*.

### How does Java implement polymorphism?

Polymorphism manifests itself in Java in the form of multiple methods having the same name (overridden methods). The same method call (invocation) will lead to different method definitions.

Polymorphism is implemented in java because of late binding. Deciding on which version of a method will be executed is based on the actual type of the object whose reference (address) is stored in the reference variable, and not on the type of the reference variable on which the method is invoked.

Late binding does not apply to final and private instance methods. It does not also apply to static methods. For these three cases the binding will be done during compilation time (early binding).

The binding decision cannot be made at compilation time because the compiler has no way of knowing (*when the program is compiled*) the actual type of the object whose reference is stored in the reference variable.

Polymorphism requires that the type of the reference variable (storing the object's address) be a superclass of the class from which the object is instantiated (created). The type of the reference variable must also be a class that either defines or inherits the method that will be invoked on the object.

Consider the following statements which are based on the assignment of Lab1:

```
PurchaseItem p= new WeighedItem("banana",3.5,3); // make a weighedItem object
```

The statement above is an example of upcasting. Although the actual object is of type WeighedItem, the methods that can be accessed through variable p are the methods of PurchaseItem class and those inherited by PurchaseItem. The following statement is illegal

```
p.getWeight(); // illegal
```

we can still access getWeight() by doing downcasting.

```
WeighedItem w =(WeighedItem) p;
```

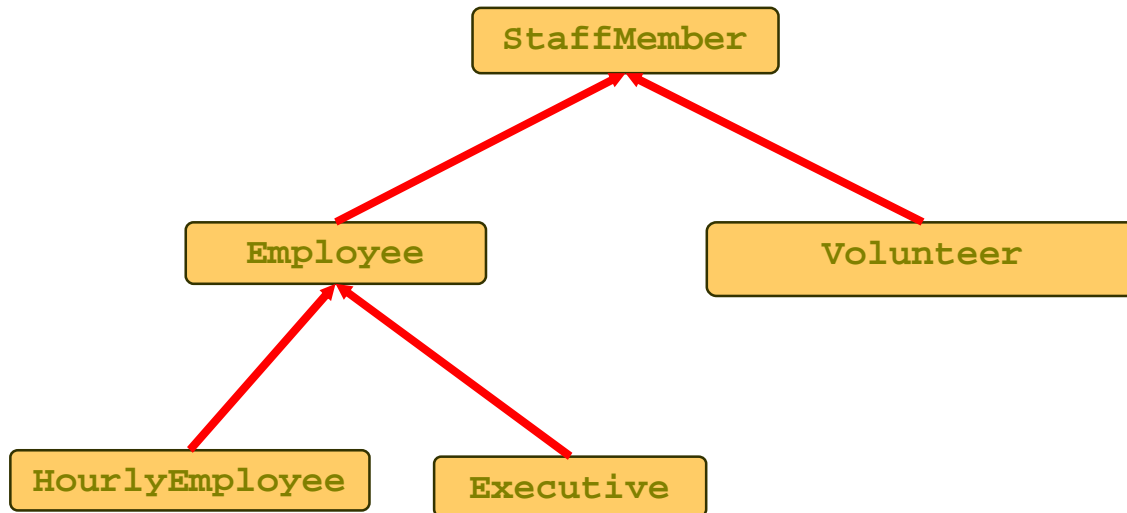
Now we can call getWeight() method using w reference variable i.e. w.getWeight().

Downcasting has to be explicit i.e. using the name of the class between brackets.

Downcasting has to be preceded by upcasting. Downcasting should not go lower than the class of the actual object, otherwise a run-time exception will be generated. To

make a safe downcast, the type of the object can be checked using `instanceof` operator.

Polymorphism is especially useful when we want to create a heterogeneous collection of objects i.e. making one array containing different objects. Such an example is represented below. Consider a class `StaffMember` that represents all workers in a certain organization, some of whom are employees and some are volunteers. Among the employees: some are regular employees paid monthly, some are temporary and are paid by the hour, while some are managers (executives) and get bonuses.



```
// StaffMember.java
class StaffMember {
    private String name;
    private String phone;
    public StaffMember (String name, String phone) {
        this.name = name;
        this.phone = phone;
    }
    public double pay(){
        return 0.0;
    }
}
```

```
// Volunteer.java
class Volunteer extends StaffMember {

    public Volunteer (String name, String phone) {
        super (name, phone);
    }

    // No need to override pay, used as inherited
}
```

```

// Employee.java
class Employee extends StaffMember
{
    private double payRate;

    public Employee (String name,String phone, double payRate) {
        super (name, phone);
        this.payRate = payRate;
    }
    public double getPayRate() {
        return payRate;
    }

    // override pay method
    public double pay () {
        return payRate;
    }
}

```

```

// HourlyEMployee.java
class HourlyEmployee extends Employee{
    private int hoursWorked;

    // constructor
    public HourlyEmployee (String name,String phone,double payRate) {
        super (name,phone, payRate);
        hoursWorked = 0;
    }
    // added method
    public void addHours (int moreHours) {
        hoursWorked += moreHours;
    }
    // override method pay of Employee: Compute and return the pay for this
    // HourlyEmployee
    public double pay () {
        double payment = getPayRate() * hoursWorked;
        hoursWorked = 0; // once pay computed set hoursWorked to 0
        return payment;
    }
}

```

```

// Executive.java
class Executive extends Employee
{
    private double bonus;

    // constructor
    public Executive (String name,String phone, double payRate)
    {
        super (name,phone, payRate);
        bonus = 0; // bonus has yet to be awarded
    }

    // unique method
    public void awardBonus (double execBonus)
    {
        bonus = execBonus;
    }

    // override method pay of Employee: Compute and
    // return the pay for an executive, which is the
    // regular employee payment plus a one-time bonus
    //-----
    public double pay ()
    {
        double payment = super.pay() + bonus;
        bonus = 0; // once bonus added reset it to 0
        return payment;
    }
}

```

Suppose we would like to store personnel of all kinds in a single array, so we can easily write code that takes care of all the workers.

For example, suppose we want to implement a method `getTotalCost()` in class `Staff` that will compute how much money is needed to pay all personnel at the end of the month



```

// Staff.java
class Staff {
    public static void main (String[] args) {
        StaffMember[] staffList;
        staffList = new StaffMember[4];
        staffList[0] = new Executive ("Ahmad", "860-1490", 1923.07);
        staffList[1] = new Employee ("Ali", "0555-0101", 846.15);
        staffList[2] = new HourlyEmployee ("Othman", "0555-0690", 8.55);
        staffList[3] = new Volunteer ("Bandar", "849-8374");
        for (int i=0;i< staffList.length;i++) {
            if (staffList[i] instanceof Executive) {
                Executive e=(Executive)staffList[i];
                e.awardBonus (5000.00);// downcasting to access awardBonus method
            }
            else if (staffList[i] instanceof HourlyEmployee) {
                HourlyEmployee h=(HourlyEmployee)staffList[i];
                h.addHours (40);// downcasting to access addHours method
            }
        }
        System.out.println("The total amount to pay is "+getTotalCost (staffList));
    }
// compute payday costs
    public static double getTotalCost (StaffMember[] stm)  {
        double amount = 0.0;

        for (int count=0; count < stm.length; count++)    {
            amount += stm[count].pay(); // polymorphism
        }
        return amount;
    }
}

```

### Exercise:

Consider the following inheritance hierarchy.

A class RentedVehicle that has:

- One private instance variable baseFee of type double
- One constructor to initialize the instance variable
- One instance method getCost () that returns the base fee
- One instance method travel of type void that takes one parameter of type int to indicate the distance traveled in KM. Here the method should print "I cannot define how to travel".

A subclass FuelVehicle that :

- has one additional private instance variable nbKms indicating the total number of kilometers traveled.
- one constructor to initialize the instance variables.
- one instance method getMileageFees to return the fees due to mileage based on the following:  
 If nbKms < 100 mileagefees=0.2\*nbkms  
 If 100<=nbKms<= 400 mileagefees=0.3\*nbkms  
 If nbKms>400 mileagefees=0.3 times 400 plus 0.5 times the extra kilometers above 400.

- overrides travel that updates nbKms by adding the parameter variable which represents the distance traveled

A Car class which is a subclass of FuelVehicle that :

- has one additional private instance variable nbSeats
- has one constructor to initialize the instance variables
- overrides getCost method by adding nbseats\*baseFee to mileageFees

A Truck class which is a subclass of FuelVehicle that:

- has one private instance variable capacity
- has one constructor to initialize the instance variables
- overrides getCost method by adding baseFee\*capacity to mileageFees

A Bicycle class that extends RentedVehicle that:

- has one additional private instance variable nbDays indicating the number of days it is rented.
- has one constructor to initialize the instance variables
- overrides travel method by printing "I don't consume fuel"
- overrides getCost method to return baseFee \* nbDays

Implement all five classes with their accessor and mutator methods.

Write an application class that generates 6 objects randomly from either a Car, or a Truck, or a Bicycle class. You need to generate an integer random number between 1 and 3 and based on its value you generate your object from one of the three classes. Your generated objects will be stored in a RentedVehicle array.

Write a loop of 20 iterations. During each iteration, it generates a random number from 0 to 5 (index of the element in the array). The selected element will call the travel method with some distance.

Now write another loop to find the total cost of the rented vehicles in the array. Meanwhile

- if the object is a car print its name and the number of seats.
- if it is a Truck, print its name and capacity
- if it is a Bicycle, print its name and how many days it is rented.

## Lab 04: Inner classes and Java Virtual Machine

**Objectives:** In this lab, the following topics will be covered

7. Inner classes
8. Java Virtual Machine
9. Examples & Exercises for practice

### 1- Inner classes:

Ordinary classes that we have covered so far are called top-level classes. The access modifiers allowed for them are public and default only. Nested classes are classes defined inside other classes. These nested classes can be static or non static.

Static nested classes are called static inner classes or top-level nested classes. A non static nested class is called an inner class. Static and non static inner classes can have all four access modifiers i.e. public, protected, default, and private.

Static inner classes are nested for organizational convenience. The additional thing added by nesting is the name and access level of the class. Static inner classes cannot access non static members of the enclosing class.

Non static inner classes cannot have static members. Each instance (object) of the inner class is linked to an object of the outer class. The non static inner class can access all members of the outer class (variables and methods) even if they are private. You can access the members of the inner classes from the enclosing class, but after an object from the corresponding inner class is made.

A local class is a class declared within a block of code. A local class is visible and usable within the block of code in which it is defined. A local class can access all members of the enclosing class even if they are private. In addition, it can access any final local variable or method parameters defined within its scope. Local classes cannot have access modifiers, and static members. If we are defining a local class and making only one object from it, then it can be converted into an anonymous class.

Thus, an anonymous class is a local class with no name. Since an anonymous class has no name, you can not declare a variable to be of that type. For this reason, anonymous classes must either be a subclass of another class or implement an interface. An anonymous class is always defined in a statement.

### Exercises

1- Consider the following example: go over the comments one by one and answer them.

```
class Outer {
    private String o1="instance variable from outer";
    private static String o2="static variable from outer";
    public void instMethOuter () {
        // print in1 of class InstInner
        // print in4 of class StatInner
    }
    public static void statMethOuter() {
        // call instMeth1() of class InstInner
        // call statMeth1 2()
    }
}
// non static inner class
private class InstInner { // First uncomment statements corresponding to in2 and
    // statMeth1 to check that non static inner classes cannot have static members
    private String in1="private instance variable from non static inner";
```

```

// private static String in2="private static variable from non static inner";
public void instMeth1() {
// print o1 and o2 of Outer
}
// public static void statMeth1() {
// }
} // end of non static inner class

// static inner class
private static class StatInner {
private String in3="private instance variable from static inner";
private static String in4="private static variable from static inner";
public void instMeth2() {
System.out.println("from instance method of static inner class");
// print o1 and o2 of Outer
}
public static void statMeth2() {
System.out.println("from static method of static inner class");

}
} // end of static inner class

// main method

}
class Test {
public static void main(String[] args) {
// make objects from both classes InstInner and StatInner. Change their access //modifiers
if needed
}
}

```

**2.** Which variables may be referenced correctly at line 8?

```

1. public class Outer{
2.     private int a = 0;
3.     public int b = 0;
4.     public void method(final int c) {
5.         int d = 1;
6.         class Inner{
7.             private void innerMethod( int e) {
8.                 //HERE
9.             }
10.        }
11.    }
12. }

```

**3.** Which variables may be referenced correctly at line 6?

```

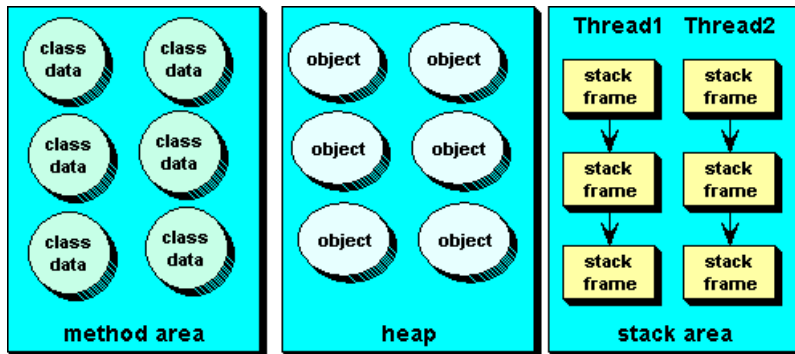
1. public class Outer{
2.     private int a = 0;
3.     static public int b = 0;
4.     static class Inner{
5.         private void innerMethod( int e) {
6.             //HERE

```

- 7.        }
- 8.        }
- 9.        }

**2. Java Virtual Machine:**

JVM is a component of the Java system that interprets and executes the instructions in our class files. Each instance of the JVM has one method area, one heap, and one or more stacks - one for each thread. When JVM loads a class file, it puts its information in the method area. As the program runs, all objects instantiated are stored in the heap. The stack area is used to store activation records as a program runs.



**Class Loading:**

Loading means reading the class file for a type, parsing it to get its information, and storing the information in the method area. For each type it loads, the JVM must store the following kinds of information in the method area:

- The fully qualified name of the type
- The fully qualified name of the type's direct superclass or if the type is an interface, a list of its direct super interfaces .
- Whether the type is a class or an interface
- The type's modifiers ( public, abstract, final, etc)
- Constant pool for the type: constants and symbolic references.
- Field info: name, type and modifiers of variables (not constants)
- Method info: name, return type, number & types of parameters, modifiers, bytecodes, size of stack frame and exception table.

The end of the loading process is the creation of an instance of java.lang.Class for the loaded type. The purpose is to give access to some of the information captured in the method area for the type, to the programmer.

### TestCircleClass.java:

The following program gives information regarding the class Circle [Refer to Lab 3].

```
import java.lang.reflect.Method;

public class TestCircleClass
{
    public static void main(String[] args)
    {
        Circle myCircle = new Circle(15.75); // according to your constructor

        Class circleClassInfo = myCircle.getClass();
        System.out.println("Class Name is: " + circleClassInfo.getName());
        System.out.println("Parent is: " + circleClassInfo.getSuperclass());
        System.out.println("Methods are: ");
        Method[] methods = circleClassInfo.getMethods();
        for(int i = 0; i < methods.length; i++)
            System.out.println(methods[i]);
    }
}
```

### Exercise :

4- Compile and execute the above program and observe its output. Now study the methods available in the class Class (from the java documentation). Some of the methods available are:

```
public String getName()
public Class getSuperClass()
public boolean isInterface()
public Class[] getInterfaces()
public Method[] getMethods()
public Field[] getFields()
public Constructor[] getConstructors()
```

Use these (and other) methods to get more information regarding the class Circle in your program TestCircleClass.java

### Linking: Verification, Preparation and Resolution:

The next process handled by the class loader is Linking. This involves three sub-processes: Verification, Preparation and Resolution. Verification is the process of ensuring that binary representation of a class is structurally correct. Example of some of the things that are checked at verification are:

- Every method is provided with a structurally correct signature.
- Every instruction obeys the type discipline of the Java language
- Every branch instruction branches to the start not middle of another instruction.

In the Preparation phase, the Java virtual machine allocates memory for the class (i.e static) variables and sets them to default initial values. Note that class variables are not initialized to their proper initial values until the initialization phase - no java code is executed until initialization. The default values for the various types are shown below:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

**Resolution** is the process of replacing symbolic names for types, fields and methods used by a loaded type with their actual references. Symbolic references are resolved into a direct references by searching through the method area to locate the referenced entity.

## Class Initialization and Instantiation

Initialization of a class consists of two steps:

- Initializing its direct superclass (if any and if not already initialized)
- Executing its own initialization statements

The above imply that, the first class that gets initialized is Object. Note that static final variables are not treated as class variables but as constants and are assigned their values at compilation.

After a class is loaded, linked, and initialized, it is ready for use. Its static fields and static methods can be used and it can be instantiated. When a new class instance is created, memory is allocated for all its instance variables in the heap. Memory is also allocated recursively for all the instance variables declared in its super class and all classes up in inheritance hierarchy. All instance variables in the new object and those of its superclasses are then initialized to their default values. The constructor invoked in the instantiation is then processed. Finally, the reference to the newly created object is returned as the result.

The following program demonstrates the order of class instantiation.

```

class Super {
    int x=10;
    Super() {
        this(9);
        printThree();
        x=20;
    }

    Super (int x) {
        printThree();
        this.x=x;
    }
    void printThree() {
        System.out.println("three");
    }
}
class Test extends Super {
    int three = (int)Math.PI;    // That is, 3
    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
    void printThree() {
        System.out.println("three = "+three+"  x= "+x);
    }
}

```

**Exercise:**

**5-** Try to give the output of the above example without compiling and executing it. Then, compile and execute the program to check your answer.

Comment the constructor `Super()`, now compile the program and explain the compilation error.

comment again the other the other constructor in `Super` and compile. Why is your program now compiling without errors; explain.



## Lab 05: Event Driven Programming

**Objectives:** In this lab, the following topics will be covered

10. Introduction to event-driven programming
11. Your first GUI program
12. Listeners
13. Layout Managers (a mild introduction)
14. JLabel component.
15. Colors
16. Exercises.

### Introduction to event-driven programming

Event-driven programming is a programming style that uses a signal-and-response approach to programming.

- **Example:**

- o *Signal:* Mouse scrolling the scrollbar of this window.
- o *Response:* the word document view moves up or down according to user's action.

An *event* is an object that acts as a signal to another object know as a *listener*.

*Clicking* of a *button* with a mouse is an example of an event.

#### Display 17.1 Event Firing and an Event Listener

---

*The component (for example, a button) fires an event.*



*This listener object invokes an event handler method with the event as an argument.*

In event driven programming, the program waits for the next event. The programmer might define some methods that might not be invoked unless an event on some object occurs.

## Your first GUI program

In order to create a GUI application that consists of a main window with objects inside in it (like buttons and text fields) to make user's interaction possible, we have to learn about java's way of implementing such application

- AWT (Abstract Windowing Toolkit) ***java.awt.\****

This is the original package of java's implementation of GUI application.

- Swing ***javax.swing.\****;

Swing is an improved version of AWT. It was introduced in JDK1.2. It does not completely replace AWT.



Swing window

AWT window

```
public class AWTSwing
{
    public static void main(String[] args)
    {
        java.awt.Frame awt = new java.awt.Frame("DeeDee");

        javax.swing.JFrame swg = new javax.swing.JFrame("JiiJii");

        java.awt.Button awtButton = new java.awt.Button("AWT Button");
        javax.swing.JButton swgButton = new javax.swing.JButton("SWI NG Button");

        awt.setSize(600,400);
        awt.setLayout(new java.awt.FlowLayout());
        awt.add(awtButton);
        awt.setVisible(true);

        swg.setSize(600,400);
        swg.setLayout(new java.awt.FlowLayout());
        swg.add(swgButton);
        swg.setVisible(true);
    }
}
```

## Listeners

In order to make these buttons capable of reacting to user's action, we would need register them with appropriate listeners. User action can be from the keyboard or they can be from the mouse. Let us consider the *ActionListener* interface. This method has one method that has the following signature

```
public void actionPerformed(ActionEvent ae);
```

Now, in order to add action listener to a component (say Button we will do the following):

```
buttonObject.addActionListener( myListener);
```

where myListener is an instance of a class that implements the interface ActionListener. Not that this implies that there are many ways of handling the event.

### Examples of Handling a simple button click events

```
import java.awt.event.*;
import javax.swing.*;

public class SimpleButtonAction
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Example");
        JButton butt = new JButton("Click Me");
        MyListenerClass myListenerObject = new MyListenerClass();
        butt.addActionListener(myListenerObject);
        frame.add(butt);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
} // end of SimpleButtonAction class

class MyListenerClass implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        JOptionPane.showMessageDialog(null,"I was clicked !");
    }
}
```

Note that myListenerObject is an instance of a class that implements the ActionListener interface. The class MyListenerClass can also be moved as an inner class inside the class SimpleButtonAction. Try copy-and-paste it. The advantage of moving the class inside is obvious: **The inner class can access instance variables of the main (outer) class easily.**

Now consider this example:

```
import java.awt.event.*;
import javax.swing.*;

public class SimpleButtonAction implements ActionListener
{
    public SimpleButtonAction()
    {
        JFrame frame = new JFrame("Example");
        JButton butt = new JButton("Click Me");
        butt.addActionListener(this);
        frame.add(butt);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SimpleButtonAction();
    }
    public void actionPerformed(ActionEvent ae)
    {
        JOptionPane.showMessageDialog(null,"I was clicked !");
    }
}
// end of SimpleButtonAction class
```

Note here that we have indicated with the keyword `this` that, the action is handled by the same class and that it should implement the interface `ActionListener`.

Finally look at this:

```
import java.awt.event.*;
import javax.swing.*;

public class SimpleButtonAction
{
    public SimpleButtonAction()
    {
        JFrame frame = new JFrame("Example");
        JButton butt = new JButton("Click Me");
        butt.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent ae)
            {
                JOptionPane.showMessageDialog(null,"I was clicked !");
            }
        });
        frame.add(butt);
        frame.setSize(200,200);
        frame.setVisible(true);
    }
    public static void main(String[] args)
    {
        new SimpleButtonAction();
    }
}
```

Note that the instance of the class implementing ActionListener is now an anonymous inner class.

### Example of more than one components using the same Listener

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

public class SimpleButtonAction
{
    int count1;
    int count2;

    JFrame frame = new JFrame("Example");
    JButton butt1 = new JButton("Click Me");
    JButton butt2 = new JButton("Click Me");

    public SimpleButtonAction()
    {
        count1 = 0;
        count2 = 0;

        InnerListener listener = new InnerListener();

        butt1.addActionListener(listener);
        butt2.addActionListener(listener);

        frame.setLayout(new FlowLayout());
        frame.add(butt1);
        frame.add(butt2);
        frame.setSize(200,200);
        frame.setVisible(true);
    }

    class InnerListener implements ActionListener
    {
        public void actionPerformed(ActionEvent ae)
        {
            if(ae.getSource() == butt1)
            {
                count1++;
                JOptionPane.showMessageDialog(frame,"1st Button "+count1 + " times");
            }
            else
            {
                count2++;
                JOptionPane.showMessageDialog(frame,"2nd Button "+count2 + " times");
            }
        }
    }
}
```

```

    public static void main(String[] args)
    {
        new SimpleButtonAction();
    }
} // end of SimpleButtonAction class

```

This example demonstrates how we can store the count of how Many times we press on a Button. We can create instance variables count1 and count2 and each one of these will be accessed by the listener class easily (unlike if the listener was outside the class).

We have also used the method getSource() to determine which button fired the event. Note that we could have defined a new class to handle actions of butt2 instead.

### Layout Managers (a mild introduction)

We have also used **setLayout(new FlowLayout())** to make the components of the JFrame laid automatically according to the size of the window. Java has different predefined ways to lay components. We will discuss layout managers in the next lab.

### JLabel Component

JLabel components represent a textual component that can be added to your Java Application. The following example demonstrate how it works

```

import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

public class SimpleButtonAction extends JFrame
{
    JButton butt1 = new JButton("Click me");
    JLabel label = new JLabel("");

    int count = 0;

    public SimpleButtonAction()
    {
        super("My Application");

        setSize(200,200);
        setLayout(new FlowLayout());
        add(butt1);
        add(label);
        setVisible(true);
        butt1.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                count++;
                label.setText("Button was clicked "+count+" Times");
            }
        });
    }
}

```

```

    public static void main(String[] args)
    {
        new SimpleButtonAction();
    }
} // end of SimpleButtonAction class

```

## Colors

Java.awt.Color class represent a Color. We can create objects of this class based on the RGB (Red Green Blue) properties. For Example

```
Color c = new Color(0,0,255);
```

Represent a blue color.

We can instead make use of the constant colors in the Color class.

### Display 17.5 The Color Constants

---

Color.BLACK	Color.MAGENTA
Color.BLUE	Color.ORANGE
Color.CYAN	Color.PINK
Color.DARK_GRAY	Color.RED
Color.GRAY	Color.WHITE
Color.GREEN	Color.YELLOW
Color.LIGHT_GRAY	

The class Color is in the java.awt package.

### Example of a Blue JFrame and Red JLabel

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ColoringFrame extends JFrame
{
    JLabel lbl = new JLabel("where do you want to go today ?");
    public ColoringFrame()
    {
        getContentPane().setBackground(new Color(0,0,255));
        setSize(500,100);
        add(lbl);
        lbl.setFont(new Font("Times New Roman", Font.BOLD, 34));
        lbl.setForeground(Color.red);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        new ColoringFrame();
    }
}

```



```
}
```

## Exercises

1. Create a JFrame application that has:

- One JButton labeled "AC Milan"
- Another JButton labeled "Real Madrid"
- A JLabel with the text "Result: 0 X 0"
- A JLabel with the text "Last Scorer: N/A "
- A Label with the text "Winner: DRAW";

Now whenever you click on the AC Milan button, the result will be incremented for AC Milan to become 1 X 0, then 2 X 0. The Last Scorer means the last team to have scored. In this case: AC Milan. And the winner is the team that has more button clicks than the other.

This is how your program should look like:



Scenario: Milan scores two goals, then Madrid scores two goals.

2. Implement a JFrame that has four Buttons that are labeled as follows:

- \* Red
- \* White.
- \* Blue.
- \* Green

Now when the button Red is pressed the main Frame should be colored Red. And so on for the other buttons.

## Lab 06: Event Driven Programming - II

**Objectives:** In this lab, the following topics will be covered

17. Text Fields and Text Areas.
18. Layout Managers (BorderLayout and GridLayout)
19. Applets
20. MouseListeners.
21. Menus.
22. Exercises.

Text Fields are text boxes that accommodates text. Example of text fields are the user name and user password fields used when u log in to your e-mail.

Java has a defined component class JTextField that can be created as follows

```
JTextField jta = new JTextField (10);
```

were 10 is the number of characters that can be visible in the text field. The text at the JTextField can be returned using the method `getText()`. It can also be set using the method `setText(String s)`.

like other components, we can change the color and the font of the text field.

*Example 1*

```
class LabExample extends JFrame
{
    JTextField jta = new JTextField(10);
    Font fnt = new Font("Times new roman",Font.BOLD,20);
    LabExample()
    {
        super("Example");
        setLayout(new FlowLayout());
        setSize(250,100);
        add(jta);
        jta.setForeground(Color.PINK);
        jta.setFont(fnt);
        setVisible(true);
    }

    public static void main(String[]args)
    {
        new LabExample();
    }
}
```



### Important Note

It is the responsibility of the programmer to check for errors when the input expected in the JTextField is a number. The compiler will not catch such error unless handled by the user.

Run the following example and observe the result when the input number is not in the correct format:

#### Example2

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class LabExample extends JFrame
{
    JTextField jta1 = new JTextField(10);
    JTextField jta2 = new JTextField(10);
    JButton button = new JButton(" Add them up");

    Font fnt = new Font("Times new roman",Font.BOLD,20);

    LabExample()
    {
        super("Example");
        setLayout(new FlowLayout());
        setSize(250,150);
        add(new JLabel("1st Number"));
        add(jta1);
        add(new JLabel("2nd Number"));
        add(jta2);
        add(button);

        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                try
                {
                    double x1 = Double.parseDouble(jta1.getText().trim());
                    double x2 = Double.parseDouble(jta2.getText().trim());

                    JOptionPane.showMessageDialog(null, "Result =
                    "+(x1+x2),"Alert",JOptionPane.INFORMATION_MESSAGE);

                }
                catch(Exception e)
                {

```

```

        JOptionPane.showMessageDialog(null, "Error in Numbers !","alert" ,
JOptionPane.ERROR_MESSAGE);
    }
}
});

setVisible(true);
}

public static void main(String[]args)
{
    new LabExample();
}
}

```

## JTextArea

TextAreas are similar to TextFields but they have more than one line. Examples of TextArea are those that we use in the body of e-mail messages

### Example3

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class TextAreaExample extends JFrame
{
    JTextArea jta1 = new JTextArea(10,25);
    JButton button = new JButton("Add some Text");
    public TextAreaExample()
    {
        super("Example");
        setSize(300,300);
        setLayout(new FlowLayout());
        add(jta1);
        add(button);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                String txt = JOptionPane.showInputDialog(null,"I nsert some text");
                jta1.append(txt);
            }
        });
    }
    public static void main(String[]args)
    {
        new TextAreaExample().setVisible(true);
    }
}

```

**Note**

We can easily add the possibility of scrolling a text area by adding it to a container named JScrollPane as follows:

```
JTextArea txtArea = new JTextArea(20,20)
JScrollPane jScroll = new JScrollPane(txtArea);
// ...
add(jScroll); // we add the scrollPane and not the text area.
```

Try it !



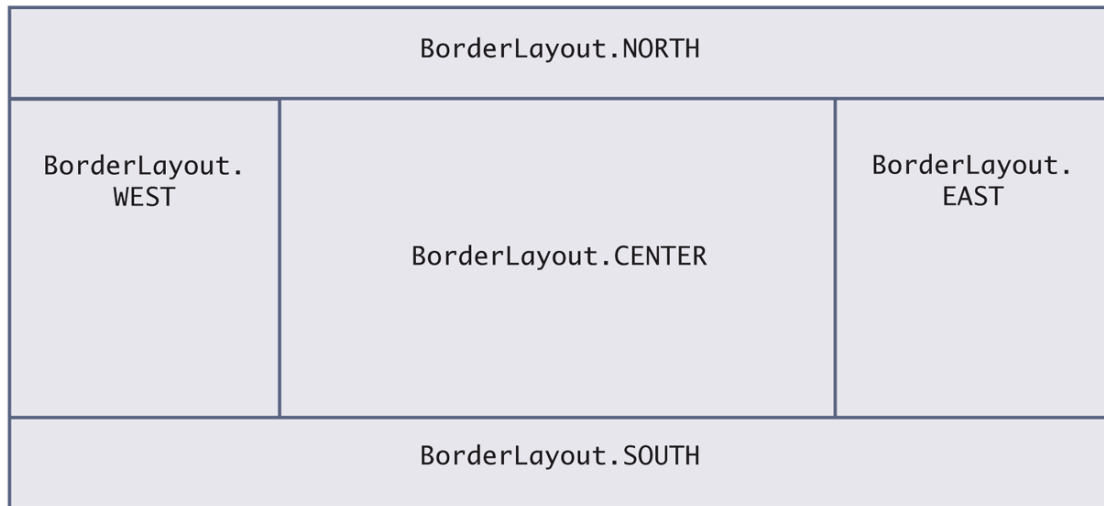
## Layout Managers:

### BorderLayout:

Divides the component into regions (WEST, EAST, NORTH, SOUTH and Center). Components can be added to any of those five components.

#### Display 17.8 BorderLayout Regions

---



The add method to a container that has a BorderLayout is different and looks like the following:

```
add( comp , BorderLayout.EAST);
```

Note that we can add JPanel to these regions and add components to those panels. We can set the layout of those JPanel to another layout.

### GridLayout

In GridLayout, the component take a tabular form, with a number of rows and number of columns.

1	2	3	4
5	6	7	8
9	10	11	12

If the component has a GridLayout of 3 rows and 4 columns then components will take the form of the table shown above and will be added in the order they are listed.

The following example illustrates a mixture of components layout

*Example4*

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class BorderExample extends JFrame
{
    JPanel[] pnl = new JPanel[12];

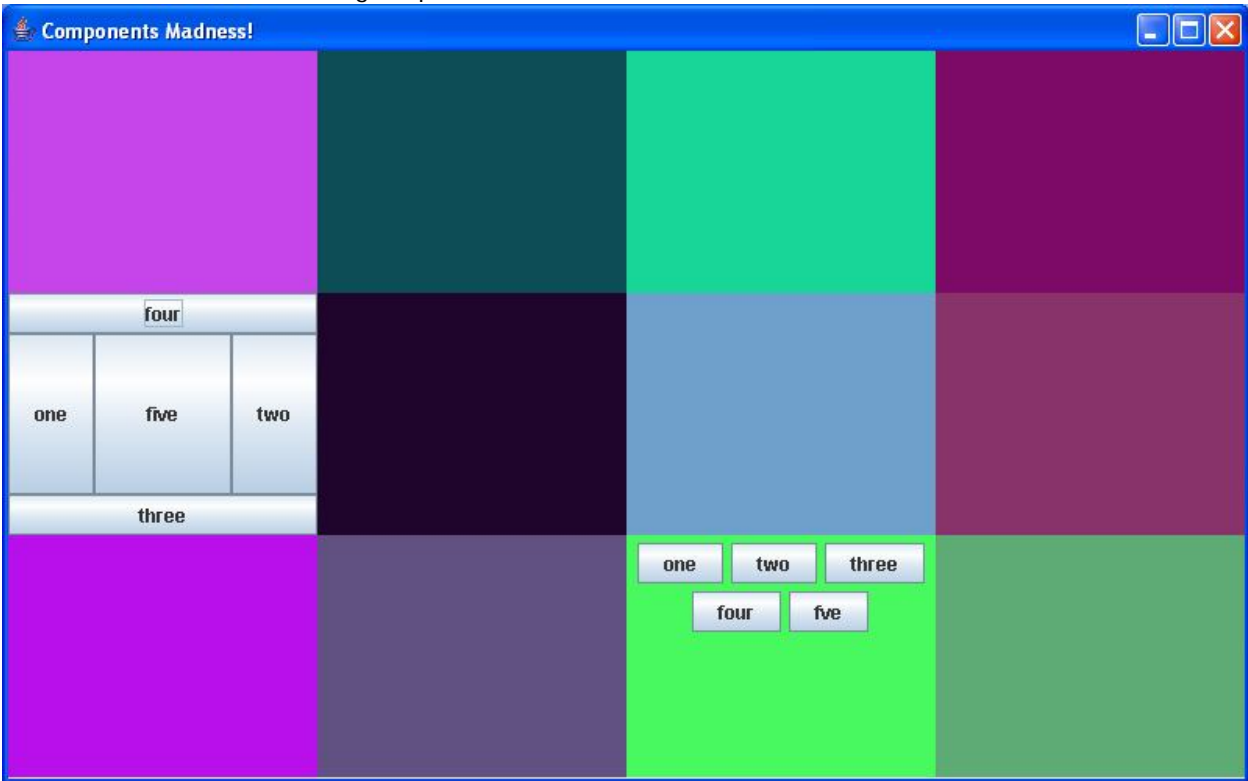
    public BorderExample()
    {
        setLayout(new GridLayout(3,4));
        for(int i = 0 ; i < pnl.length ; i++)
        {
            int r = (int) (Math.random() * 255);
            int b = (int) (Math.random() * 255);
            int g = (int) (Math.random() * 255);
            pnl[i] = new JPanel();
            pnl[i].setBackground(new Color(r,g,b));
            add(pnl[i]);
        }

        pnl[4].setLayout(new BorderLayout());
        pnl[4].add(new JButton("one"),BorderLayout.WEST);
        pnl[4].add(new JButton("two"),BorderLayout.EAST);
        pnl[4].add(new JButton("three"),BorderLayout.SOUTH);
        pnl[4].add(new JButton("four"),BorderLayout.NORTH);
        pnl[4].add(new JButton("five"),BorderLayout.CENTER);

        pnl[10].setLayout(new FlowLayout());
        pnl[10].add(new JButton("one"));
        pnl[10].add(new JButton("two"));
        pnl[10].add(new JButton("three"));
        pnl[10].add(new JButton("four"));
        pnl[10].add(new JButton("fve"));

        setSize(800,500);
    }
    public static void main(String[]args)
    {
        new BorderExample().setVisible(true);
    }
}
```

Which will have the following output

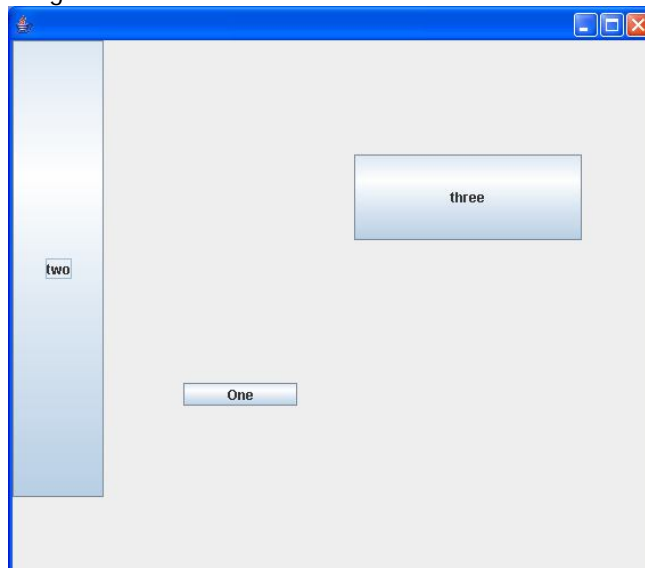


Note that the JFrame has a GridLayout of 3 by 4 while the JPanel at (2, 1) has a BorderLayout. The JPanel at (3, 3) has a FLOWLayout.

### Null Layout Manager

Sometimes you may want to fix the size and the location of the component within a container. Hence, you should tell the program not to use any Layout manager ( `setLayout(null)` )

So that we will get something like:





### Example5

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class NullLayout extends JFrame
{
    JButton but1 = new JButton("One");;
    JButton but2 = new JButton("two");;
    JButton but3 = new JButton("three");;

    public NullLayout()
    {
        setLayout(null);
        but1.setBounds(150,300,100,20); // added at 150,300 width = 100, height=20
        but2.setSize(80,400); // added at 0,0 width = 80, height=400
        but3.setLocation(300,100);
        but3.setSize(200,75);
        // those two steps can be combined in one setBounds method call
        add(but1);
        add(but2);
        add(but3);
        setSize(500,500);
    }
    public static void main(String[]args)
    {
        new NullLayout().setVisible(true);
    }
}
```

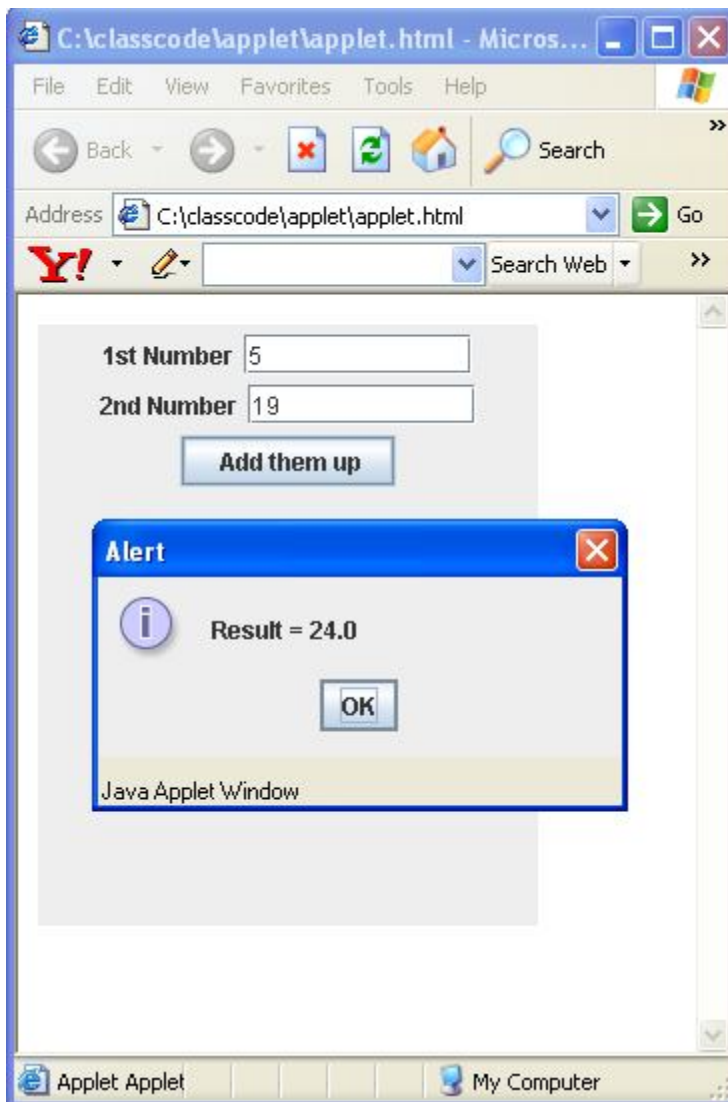
## Applets

Applet is a word used for small application. In Java we can create stand alone application and applets. Applets need to be run in an environment like a web browser.

After compiling the applet file, the resulting .class file can be embedded in an HTML page

```
<applet code="AppletExample.class" width=250 height=300></applet>
```

Then when opening the webpage with the browser (for example Internet Explorer) the applet will be run with the width =250 and height = 300



And the following is the modified code of example2 to change the application into an applet:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class AppletExample extends JApplet
{
    JTextField jta1 = new JTextField(10);
    JTextField jta2 = new JTextField(10);
    JButton button = new JButton(" Add them up");
    public void init()
    {
        setLayout(new FlowLayout());
        setSize(250,150);
        add(new JLabel("1st Number"));
        add(jta1);
        add(new JLabel("2nd Number"));
        add(jta2);
        add(button);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ae)
            {
                try
                {
                    double x1 = Double.parseDouble(jta1.getText().trim());
                    double x2 = Double.parseDouble(jta2.getText().trim());
                    JOptionPane.showMessageDialog(null, "Result =
"+(x1+x2),"Alert",JOptionPane.INFORMATION_MESSAGE);
                }
                catch(Exception e)
                {
                    JOptionPane.showMessageDialog(null, "Error in Numbers !","alert" ,
JOptionPane.ERROR_MESSAGE);
                }
            }
        });
        setVisible(true);
    }
}
```

## MouseListener

We can implement mouse listeners and key listeners on components. The interface `MouseListener` has the following methods:

Method Summary	
void	<a href="#"><code>mouseClicked(MouseEvent e)</code></a> Invoked when the mouse button has been clicked (pressed and released) on a component.
void	<a href="#"><code>mouseEntered(MouseEvent e)</code></a> Invoked when the mouse enters a component.
void	<a href="#"><code>mouseExited(MouseEvent e)</code></a> Invoked when the mouse exits a component.
void	<a href="#"><code>mousePressed(MouseEvent e)</code></a> Invoked when a mouse button has been pressed on a component.
void	<a href="#"><code>mouseReleased(MouseEvent e)</code></a> Invoked when a mouse button has been released on a component.

The `MouseListener` can be added to components as follows:

```
Component.addMouseListener(listener);
```

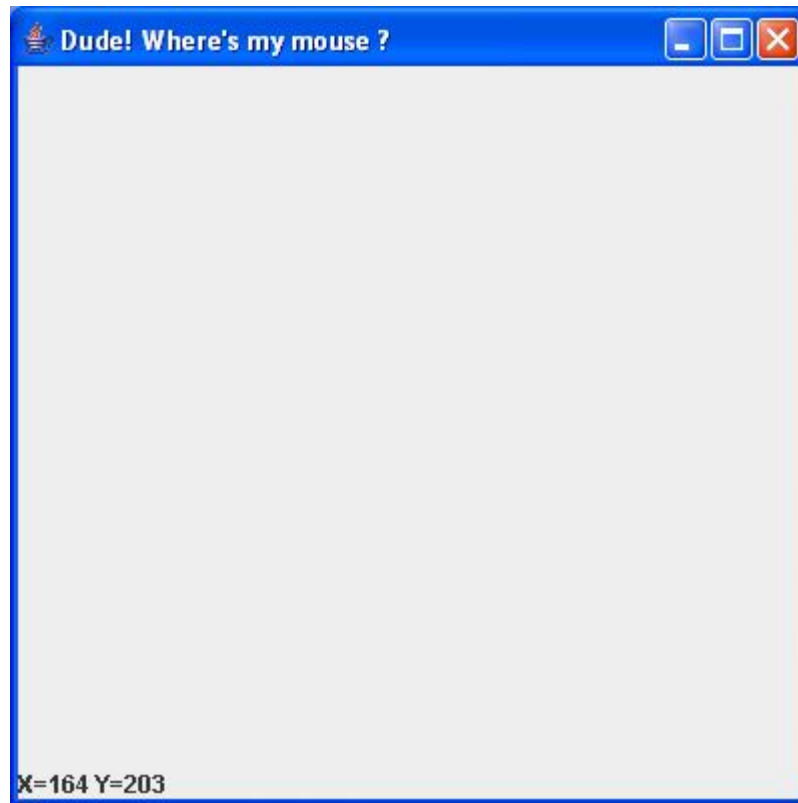
Where `listener` is an instance of a class that implements the interface `MouseListener`. Note that it should provide implementation of all the methods in the table.

### Example6

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MyMouse extends JFrame
{
    JLabel lbl = new JLabel("");
    public MyMouse()
    {
        super("Dude! Where's my mouse ?");
        setSize(400,400);
        setLayout(new BorderLayout());
        add(lbl,BorderLayout.SOUTH);
        addMouseListener(new MouseListener()
        {
            public void mouseExited(MouseEvent a){}
            public void mouseClicked(MouseEvent a) {lbl.setText("X="+a.getX()+" Y="+a.getY());}
            public void mouseEntered(MouseEvent a) {}
            public void mouseReleased(MouseEvent a) {}
            public void mousePressed(MouseEvent a) {}
        });
    }
}
```

```
}  
public static void main(String[]args)  
{  
    new MyMouse().setVisible(true);  
}  
}
```



## Menus

Adding menus to a java program is simple. Java defines three components to handle those

- JMenuBar: which represent the component that holds the menus.
- JMenu: which represent a menu of items to select from.
- JMenuItem: which represents an Item that can be clicked from a Menu.



Similar to Button (in fact MenuItem are subclasses of the AbstractButton class). We can add an ActionListener to them just like we did with buttons.

## Exercises

1. Implement a guessing game that has one text field and one button. It will ask the user to guess a number between 0-20 and gives him three trials. If he does not succeed it will tell the user made a mistake in the guess and that the number is smaller/larger. If the user consume three times of guessing the program should terminate with appropriate message. If the user got the right guess, then the program should terminate with an appropriate message.
2. Implement a Java application that has a border layout and labels for each region in the layout. Now defined mouse event to act as the following:
  - a. When mouse enters CENTER the program shows a dialog (Welcome to Riyadh)
  - b. When mouse enters WEST the program shows a dialog (Welcome to Jeddah)
  - c. When mouse enters SOUTH the program shows a dialog (Welcome to Abha)
  - d. When mouse enters NORTH the program shows a dialog (Welcome to Hail)
  - e. When mouse enters EAST the program shows a dialog (Welcome to Dhahran)
3. Implement a java program with a JTextArea and Two Menus:
  - a. Color: which has three Items: Blue, Red, and Black
  - b. Font: Which has three fonts: "Times New Roman", "MS Sans Serif", "Courier New" and "Bodoni MT Black"

You should be able, by using the menus to change the font and the color of the text written in the JTextArea.

## Lab 07: Graphics

**Objectives:** In this lab, the following topics will be covered

- 23. Drawing of objects: Lines, Rectangles and Ellipses.
- 24. User defined panels.
- 25. Drawing Images (user defined image component)
- 26. 2D Transformation
- 27. Exercises.

### **1. Drawing of objects: Lines, Rectangles and Ellipses.**

As we have studied in class. A JFrame application that extends the javax.swing.JFrame, it will inherit the paint method.

In that method, we can redefine the paint method and draw our own shapes. Note the methods, will draw on the component JFrame, so it might draw lines or objects OVER/UNDER buttons that are add to the JFrame, depending on which one comes first.

An explicit call to a method repaint() will invoke the paint method. However, whenever a window is resized, the repaint() is called.

Example:

```
import javax.swing.*;
import java.awt.*;

class AllShapes extends JFrame
{
    AllShapes()
    {
        super("All Shapes");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setVisible(true);
        setSize(328,583);
    }
    public static void main(String[] args)
    {
        new AllShapes();
    }
    public void paint(Graphics g)
    {
        super.paint(g);
        // LINE
        g.drawLine(100, 20, 200, 100);
        //RECTANGULAR
        g.drawRect(5, 150, 150, 100);
        // ROUND RECTANGULAR
        g.drawRoundRect(165, 150, 150, 100, 15, 15);
        // SQUARE
        g.drawRect(5, 295, 100, 100);
        // ROUND SQUARE
        g.drawRoundRect(165, 295, 100, 100, 15, 15);
        // OVAL
        g.drawOval(5, 440, 150, 100);
        // CIRCLE
        g.drawOval(165, 440, 100, 100);
        // Polygon
        //g.drawRoundRect(115, 440, 100, 100, 15, 15);
        g.drawString("Line", 120, 15); //LINE STRING
        g.drawString("Rectangular", 10, 140); //Rect
        g.drawString("Round Rectangular", 180, 140); //RoundRect
        g.drawString("Square", 10, 285); //Square
        g.drawString("Round Square", 180, 285); //RoundSquare
        g.drawString("Oval", 10, 430); //Oval
        g.drawString("Circle", 180, 430); //Circle
        setTitle(getWidth() + " " + getHeight());
    }
}
```



We can use a sort of global variables that can be modified by other methods in the class (i.e. listeners).

The following example shows a random generation of Shapes and Colors and drawing them. Each time a button is pressed, different shapes are drawn.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class RandomShapes extends JFrame
{
    JButton b= new JButton("Change");;
    Color c;

    RandomShapes()
    {
        super("Random Shapes");
        setLayout(new BorderLayout());
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        add(b, BorderLayout.SOUTH);

        setSize(600,400);

        b.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                repaint();
            }
        });
        setVisible(true);
    }
    public static void main(String[] args)
    {
        new RandomShapes();
    }
    public void paint(Graphics g)
    {
        int lineX1, lineX2, lineY1, lineY2;
        int ciX1, ciX2, ciY1, ciY2;
        super.paint(g);
        lineX1 = (int)(Math.random() * getWidth() / 4.0);
        lineY1 = (int)(Math.random() * getHeight()/ 4.0);
        lineX2 = (int)(Math.random() * getWidth());
        lineY2 = (int)(Math.random() * getHeight());

        ciX1 = (int)(Math.random() * getWidth() / 4.0);
```

```

        ciY1 = (int)(Math.random() * getHeight()/ 4.0);
        ciX2 = (int)(Math.random() * getWidth());
        ciY2 = (int)(Math.random() * getHeight());

        c = new Color( (int) (Math.random() * 255) , (int) (Math.random() * 255) , (int)
(Math.random() * 255));
        g.setColor(c);
        g.fillRect( lineX1 , lineY1 , lineX2 , lineY2);

        c = new Color( (int) (Math.random() * 255) , (int) (Math.random() * 255) , (int)
(Math.random() * 255));
        g.setColor(c);
        g.fillOval(ciX1, ciY1, ciX2, ciY2);

    }
}

```

## 2. User defined panels

We can create our own special panel, and give it special features, like default color and default layout manager.

The following example, will show an example of a user defined panel that draws a cup. This cup updates its size when the frame resizes.

This was possible by extending JPanel in our CupComponent class. The class will inherit methods like paintComponent(Graphics g) in which is similar to the one we have for the JFrame, but it is used to paint the component.

```

import javax.swing.*;
import java.awt.*;

public class CupComponent extends JPanel
{
    CupComponent()
    {
    }
    public static void main(String[] args)
    {
    }
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        g.drawOval(20, 20, getWidth() - 60, 50);
        //TOP SIDE OF THE CUP
        g.drawOval(40, getHeight() - 60, getWidth() - 120, 50);           //BOTTOM
SIDE OF THE CUP
        g.drawLine(20 , 20+ 25, 40, getHeight() - 35);                   //LEFT
LINE OF THE CUP
        g.drawLine(getWidth() - 60 + 20 , 20 + 25, 40 + getWidth() - 120, getHeight() - 35);
        //RIGHT LINE OF THE CUP
    }
}

```

```
}  
}
```

```
import java.awt.*;  
import javax.swing.*;  
import java.awt.event.*;  
  
class TestingCup extends JFrame  
{  
    CupComponent cc = new CupComponent();  
  
    public Example1()  
    {  
        super("resize me");  
        add(cc);  
        setSize(400,400);  
    }  
  
    public static void main(String[] args)  
    {  
        new TestingCup().setVisible(true);  
    }  
}
```

### 3. Drawing Images (user defined image component)

In java to draw image we can use the command "drawImage" to draw an Image. The following example shows how to create an image object. It does this as well as creating a user-defined JComponent that has an Image drawn in it. Note that the image will be stretched to the size of the component. This means that if the component is resized, the image will be redrawn differently (either growing or shrinking)

We can also choose different image by using the file menu and the JFileChooser dialog, which helps users to pick files for opening.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class ImageComponent extends JComponent
{
    Image img;

    ImageComponent(String imageUrl)
    {
        img = Toolkit.getDefaultToolkit().getImage(imageUrl);
        repaint();
    }

    public void setImage(String imageUrl)
    {
        img = Toolkit.getDefaultToolkit().getImage(imageUrl);
        repaint();
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        g.drawImage(img,0,0, this.getWidth(), this.getHeight(), Color.BLACK, this);
    }
}
```

And the test class in which we use our ImageComponent

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class TestingImageComponent extends JFrame
{
    ImageComponent ic;

    JMenuBar jBar = new JMenuBar();
    JMenu file = new JMenu("File");
    JMenuItem open = new JMenuItem("Open");

    TestingImageComponent()
    {
        ic = new ImageComponent("42.jpg");

        file.add(open);
        jBar.add(file);

        setJMenuBar(jBar);
        setSize(400,400);
        add(ic);

        open.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent ae)
            {
                JFileChooser fc = new JFileChooser();
                fc.showOpenDialog(null);

                ic.setImage(fc.getSelectedFile().getPath());
                setTitle(fc.getSelectedFile().getName());
                repaint();
            }
        });
    }

    public static void main(String[] args)
    {
        new TestingImageComponent().setVisible(true);
    }
}
```

## 4. 2D Transformation

### Scaling

Scaling a shape means either enlarging or shrinking the object.

For every point (x,y) we use

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax \\ by \end{bmatrix}$$

### Translation

Moving an object with a horizontal displacement dx and vertical displacement dy.

For every point (x,y) we use:

$$x' = x + tx$$

$$y' = y + ty$$

### Rotation

Rotating an object with an angle  $\theta$  for every point (x,y) we use:

$$x' = x \cos(\theta) - y \sin(\theta)$$

$$y' = x \sin(\theta) + y \cos(\theta)$$

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Transformation extends JFrame
{
    JButton up = new JButton("UP");
    JButton down = new JButton("DOWN");
    JButton left = new JButton("Left");
    JButton right = new JButton("Right");
    JPanel south = new JPanel();
    int xStart = 50;
    int yStart = 50;
    int width = 200;
    int height = 200;
    Transformation()
    {
        super("Transformation");
        setSize(400,400);

        setLayout(new BorderLayout());
        add(south, BorderLayout.SOUTH);
        south.setLayout(new FlowLayout());

        south.add(up);
        south.add(down);

        up.addActionListener(new ActionListener()
        {
```

```

        public void actionPerformed(ActionEvent ae)
        {
            yStart--;
            repaint();
        }
    });

    down.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            yStart++;
            repaint();
        }
    });

    setVisible(true);

}

public void paint(Graphics g)
{
    super.paint(g);
    g.drawRect(xStart,yStart,width,height);

}

public static void main(String[]args)
{
    new Transformation();
}
}

```

## 5. Exercises

1. Design a Program "Shape Factory" that has Buttons to produce:
  - a. Random Rectangles with random colors.
  - b. Random Ellipses with random colors.
  - c. Random Lines with random colors.

Whenever user clicks on a button, the corresponding shape will be *added* to the already drawn shapes in the JFrame.

2. Modify the translation example and make it ***translate*** the shape left and right.

## Lab 08: Searching Techniques

### Objectives:

- Comparable Interface
- Comparator Interface
- Searching

### Comparable Interface

java.lang package has Comparable interface

```
public interface Comparable{
    public abstract int compareTo(Object object);
}
```

This interface may be used to order objects of a class that have a natural ordering.

A user defined class that implements Comparable should implement the compareTo method such that

: object1.compareTo(object2) returns

- 0 if object1 "is equal to" object2
- > 0 if object1 "is greater than" object2
- < 0 if object1 "is less than" object2

### Example 1

```
class Student implements Comparable{
    private int iDNumber;
    private double gPA;

    public Student(int iDNumber, double gPA){
        this.iDNumber = iDNumber;
        this.gPA = gPA;
    }
    public Student(int iDNumber){
        this(iDNumber,0.0);
    }
    public String toString(){
        return iDNumber+"\t"+gPA;
    }
    public int compareTo(Object object){
        Student s = (Student) object;
        if (iDNumber < s.iDNumber)
            return -1;
        else if (iDNumber > s.iDNumber)
            return 1;
        else
            return 0;
    }
    public boolean equals(Object object){
        return compareTo(object) == 0;
    }
}
```



```
}
```

## Comparator Interface

`java.util` package has Comparator interface

```
public interface Comparator{
    public abstract int compare(Object object1, Object object2);
    public abstract boolean equals(Object object);
}
```

This interface allows imposing a *total ordering* on some collection of objects. Comparators can be passed to a sort method to allow precise control over the sort order.

A class that implements Comparator should implement the **compare** method such that **compare(object1, object2)** returns:

- 0 if object1 "is equal to" object2
- > 0 if object1 "is greater than" object2
- < 0 if object1 "is less than" object2

**Compare** method throws `ClassCastException` - if the arguments' types prevent them from being compared by this Comparator.

It is also preferable for the **compare** method to return 0 if and only if **object1.equals(object2)** is true.

### Example 2

See implementation of compare method. We compared the objects against natural ordering

```
import java.util.*;
class Student implements Comparator{
    private int iDNumber;
    private double gPA;

    public Student(int iDNumber, double gPA){
        this.iDNumber = iDNumber;
        this.gPA = gPA;
    }
    public Student(int iDNumber){
        this(iDNumber,0.0);
    }
    public String toString(){
        return iDNumber+"\t"+gPA;
    }
    public int compare(Object object1, Object object2){
        Student s1 = (Student) object1;
        Student s2 = (Student) object2;
        if (s2.iDNumber < s1.iDNumber)
```

```

        return -1;
    else if (s2.iDNumber > s1.iDNumber)
        return 1;
    else
        return 0;
    }
    public boolean equals(Object object){
        return compare(this, object) == 0;
    }
}

```

## Search

Searching is the process of determining whether or not a given value exists in the data.

### Linear Search

The linear (or sequential) search algorithm sequentially scans the array, comparing each array item with the searched value. If a match is found; return the index of the matched element; otherwise return -1.

### Performance

For small arrays, linear search is a good solution because it's so straightforward. In an array of a million elements linear search on average will take 500,000 comparisons to find the key. For a *much* faster search, take a look at Binary Search.

### Binary Search

In order to make binary search into a Java subroutine that searches an array A for an item N, we just have to keep track of the range of locations that could possibly contain N. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than N, then the second half of the range can be eliminated. If it is less than N, then the first half of the range can be eliminated. If the number in the middle just happens to be N exactly, then the search is finished. If the size of the range decreases to zero, then the number N does not occur in the array. Here is a subroutine that returns the location of N in a sorted array A. If N cannot be found in the array, then a value of -1 is returned instead:

#### Exercise 1

Modify the class **Student** to implement **Comparable interface** such that it compares students with their GPA in increasing order.

#### Exercise 2

Write a class **SortingStudentsByGPA** that implements **Comparator interface** such that it compares students with their GPA in decreasing order.

#### Exercise 3

Write a test class that create an array of class **Student** and searches the array for a student given his **iDNumber** to return his **GPA**.

## Lab 09: Sorting Techniques

### Objectives:

- Sorting
- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort
- Merge Sort

### Sorting

One of the most frequent tasks the computer must perform is sorting. In order to sort items efficiently they must be stored in an array. There are several ways in which the computer can sort a list numerically or alphabetically.

### Selection Sort

In a selection sort, the first element of the array is **selected** as an anchor, to which all other elements are compared. Once all other elements have been compared and exchanged as required, the second element is selected as the anchor and all subsequent elements are compared. This continues until all the elements have been selected as the anchor for comparison. This selection of an anchor element is where the name of the sort comes from.

### Insertion Sort

Take the first element and compare it with the second, if they are not in order fix them. Look at the third element and put it in the correct place with respect to the first two. Do the same to all other elements. Every time an element has to be placed before others in the sorted list, those others will have to be moved back one.

### Bubble Sort

In the bubble sort elements next to each other are compared and exchanged as needed. By repeating this process of comparing adjacent elements the highest value **bubbles** its way to the last position of the array, which gives rise to the name of this type of sorting.

### Quick Sort

- Divide
  - The array  $A[1..n]$  is partitioned into two non-empty sub-arrays with respect to a "*pivot element*"
- Conquer
  - Two sub-arrays are sorted recursively by Quick Sort.
- Combine
  - Nothing to do here

### Merge Sort

- Divide
  - Partition array  $A[1..n]$  into two equal parts

- Conquer
  - Merge sort the two sub-arrays recursively
- Combine
  - Merge (combine) the sorted two sub-arrays

### **Exercise 1**

Write a test class that creates an array of class Student and sorts the array by IDNumber.

### **Exercise 2**

Write a class SortingStudentsByGPA that implements Comparator interface such that it sorts students with their GPA in a decreasing order.

### **Exercise 3**

Write a program that merges two sorted data files into a single sorted data file. Assume that the data files have integer values. Do not store the files in an array or a list.

## LAB 10: Java Collections Framework

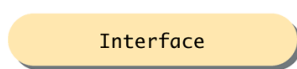
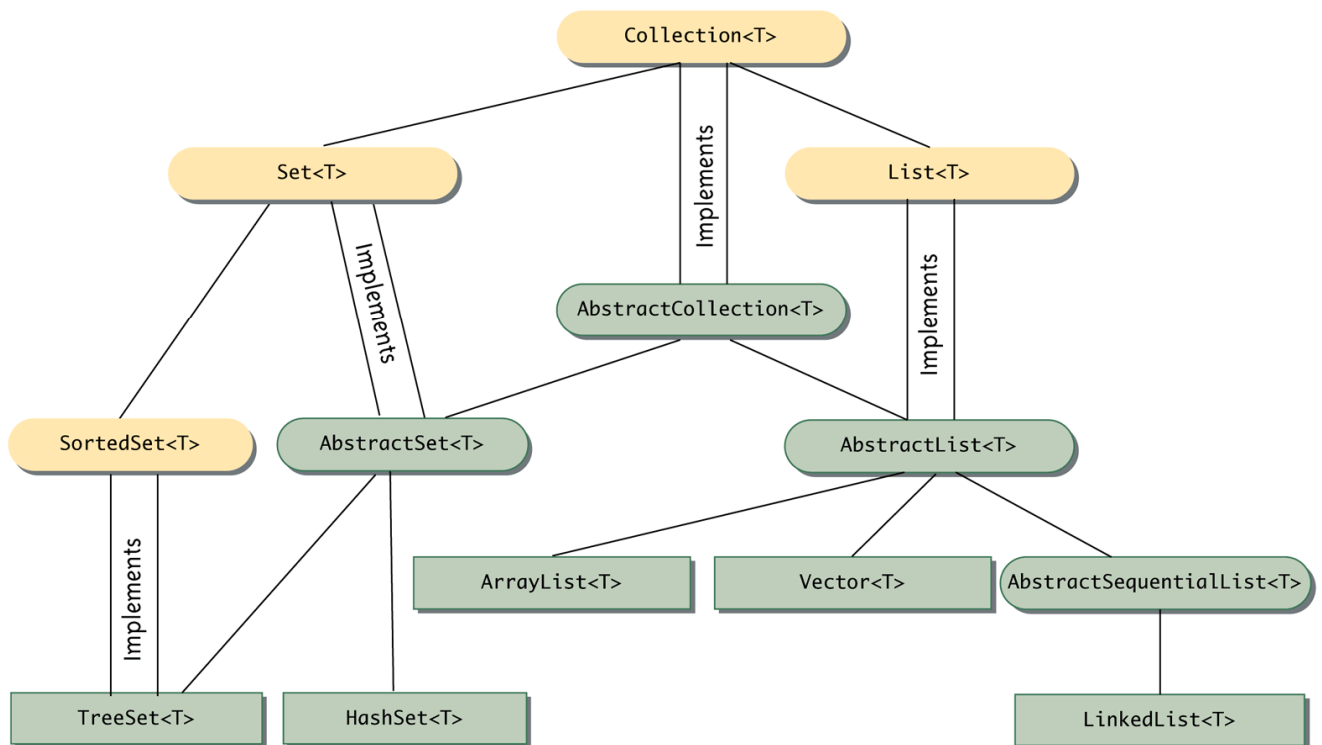
### Objectives:

In this lab we shall look at how the Java Collections framework handles collections, Specifically we concentrate on the `ArrayList` class. In general, an **ArrayList** serves the same purpose as an array, except that an **ArrayList** can change length while the program is running. An iterator is an object that is used with a collection to provide sequential access to the collection elements. This access allows examination and possible modification of the elements. An iterator imposes an ordering on the elements of a collection even if the collection itself does not impose any order on the elements it contains. If the collection does impose an ordering on its elements, then the iterator will use the same ordering

### Introduction to Java Collections framework

- A collection is a group of objects.
- The Java collections framework standardizes the way in which groups of objects are handled by a program.
- The collections framework defines several interfaces and classes in the **java.util** package. Some of these are:

## Display 16.1 The Collection Landscape



Interface



Abstract Class



Concrete Class

*A single line between two boxes means the lower class or interface is derived from (**extends**) the higher one.*

*T is a type parameter for the type of the elements stored in the collection.*

Interface	Description
Collection <T>	Declares the core methods that all collections will have.
List <T>	Extends Collection to handle sequences (lists of objects of type T). Classes that implement the <b>List&lt;T&gt;</b> interface have their elements ordered as on a list.
Set <T>	Classes that implement the <b>Set&lt;T&gt;</b> interface do not allow an element in the class to occur more than once
Iterator <T>	Defines the necessary methods to iterate through the elements of a collection.
ListIterator <T>	Extends the functionality of the Iterator interface to include two-way traversal of the List as well as the ability to modify elements.

- Methods of the **Iterator** interface

public boolean hasNext( )	Returns true if there is another element in the invoking collection object still to be visited by the iterator
public T next( )	Returns the next element in the invoking collection object. Throws NoSuchElementException if there is no next element.
public void remove( )	Removes from the collection the last element returned by the method next. It throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ). Each valid call to remove has to be preceded by a call to next.

To visit all the elements of a collection `c` containing `Integer` objects, code such as the following may be used:

<pre> Iterator&lt;Integer&gt; iter = c.iterator(); while(iter.hasNext()) {     Integer obj = iter.next();     process(obj); } </pre>	or	<pre> for(Iterator&lt;Integer&gt; i=c.iterator();     i.hasNext(); ) {     process(i.next()); } </pre>
--	----	--

**Note: When next( ) is invoked, the iterator jumps over the next element, and it returns a reference to the object that it just passed.**

The `remove( )` method of an iterator removes the element whose reference was returned by the last call to `next( )`. For example, the following code removes the first element in a collection `c` of `Strings`:

```

Iterator<String> it = c.iterator(); // sets the iterator to the beginning (first) of the
collection
it.next(); // jumps over the first element
it.remove(); // removes the first element because it is the one returned by the
previous next

```

It is illegal to call the `remove( )` method of an iterator if it was not preceded by a call to `next( )`. For example, the following is invalid:

```

Iterator<String> it = c.iterator();
it.remove();

```

Also in the following code, the second `remove( )` call is invalid. It will generate a run time exception `IllegalStateException`.

```

Iterator<String> it = c.iterator();
it.next();
it.remove(); // valid

```

```
it.remove(); // not valid
```

- **Methods of the ListIterator interface**

The list interface applies to classes that implement the List<T> interface. The listIterator object can move forward and backward.

public boolean hasNext()	Returns true if there is a next element in the invoking list object.
public boolean hasPrevious()	Returns true if there is previous element in the invoking list object.
public T next()	Returns the next element in the invoking collection object. Throws NoSuchElementException if there is no next element.
public T previous()	Returns the previous element in the invoking list object. Throws NoSuchElementException if there is no previous element.
public int nextIndex()	Returns the index of the next element. If there is no next element, returns the size of the list.
public int previousIndex()	Returns the index of the previous element. If there is no previous element, returns -1.
public void add(T newElem)	Inserts a new object into the list, immediately before the object whose reference would be returned by a subsequent call to next(). If the list is empty, the new object is added at the beginning of the list. The element added will returned by a call to the method previous.
public void remove()	Removes from the list the last element whose reference was returned by a call to next() or previous(). An IllegalStateException is thrown if remove() is called without being preceded by a call to next() or previous().
public void set(T newEl)	Replaces the last element returned by next or previous by the element newel. Cannot be used if there has been a call to add or remove after calling next or previous.

The add() method of a ListIterator adds a new element before the current listIterator position. Thus, calling add() does not affect the value returned by next(); however if you call previous() immediately after calling add(), the method returns a reference to the object that was just added.

Example1:

The program below outputs: [0, 25, 1, 2, 3, 4, 5]

```
import java.util.*;
class TestListIterator {

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<Integer> ();
        ListIterator<Integer> iter = list.listIterator();
        for(int i = 0; i < 6; i++)
            iter.add(new Integer(i));

        iter = list.listIterator();// reset to the beginning of the list
```



```

iter.next();          // jump over the first element
iter.add(new Integer(25)); //add immediately after the first
System.out.println(list); // using toString of the ArrayList
System.out.println("The elements using the new for loop");
for (Integer i :list)
    System.out.println(i);
}
}

```

The set() method of a ListIterator replaces the last element whose reference was returned by a call to next() or previous() with a new element. For example, the following code replaces the first element in a list with a new value:

```

// ...
ListIterator<T> iterator = list.listIterator();

T oldValue = iterator.next();
// ...
iterator.set(newValue);

```

## The ArrayList class

- The ArrayList class implements the List interface in terms of a **doubly linked list**. It allows list objects of any type, including null.
- Methods defined in the ArrayList class and those it inherits:

### 1. ArrayList constructors

public ArrayList ()	Creates an ArrayList<T> of default initial capacity of 10.
public ArrayList(Collection<? Extends T> c)	Creates an ArrayList<T> that contains all the elements of the collection c in the same order as they are in c.
public ArrayList (int initialCapacity)	Creates an ArrayList<T> with the specified capacity.

### 2. Methods to iterate an ArrayList

public Iterator<T> iterator()	This method is inherited from the AbstractSequentialList class. It returns an iterator to the start of the invoking ArrayList object.
public ListIterator<T> listIterator()	This method is inherited from the AbstractList class. It returns a list iterator to the start of the invoking ArrayList object.
public ListIterator<T> listIterator(int index)	Returns a list iterator to the invoking list that begins at the specified index. The first element to be returned by next is the element at index. 0<=index<=size()

The listIterator can move forward and backward for visiting the elements of the list. An example of a listIterator moving backward from the end of the list is shown below:

```
ListIterator<Integer> iter = c.listIterator(c.size());
while(iter.hasPrevious()) {
    Integer obj = iter.previous();
    process(obj);
}
```

### 3. Methods to add elements to an ArrayList<T> object

public boolean add(T newEI)	Inserts newEI to the end of the invoking ArrayList<T> object.
public void add(int index, T newEI)	Inserts newEI at position index in the invoking ArrayList object and increases the size of the calling object by 1. Any preexisting elements at and beyond the point of insertion are shifted by one location. Thus, their indices are increased by 1.
public boolean addAll(Collection<? Extends T> c)	Appends all the elements in c to the end of the elements in the calling ArrayList object.
public boolean addAll(int index, Collection<? Extends T> c)	Inserts all the elements in c in the calling object starting at position index

Note: Instead of using the add() method of an ArrayList object, one may use, as we saw before, the add() method of a ListIterator object to add a new object to an ArrayList object.

Example of using the add() method of a linked list object:

```
ArrayList<Integer> list = new ArrayList<Integer> ();
for(int i = 0; i < 6; i++)
    list.add(new Integer(i));
```

### 4. Methods to remove elements from an ArrayList

public void clear()	Removes all the elements from the invoking ArrayList object.
public boolean removeAll(Collection<?> c)	This method is inherited from the AbstractCollection class. It removes from the invoking list object all the elements contained in collection c. Returns true if the calling object is modified.
public boolean retainAll(Collection c)	This method is inherited from the AbstractCollection class. It retains only the elements in the calling object that are also contained in the collection c. It returns true if the calling object was changed
public boolean remove(Object obj)	Removes the first occurrence of obj from the invoking ArrayList object. Returns true if the element was successfully removed. The remaining elements beyond the point of removal are shifted to the left. If the element is not present, false is returned.
public T remove(int index)	Deletes the element at the specified index and returns

	the deleted element. The size of the calling object will be decreased by 1. Elements after the removed elements are shifted and their indices are reduced by 1. 0=<index< size()
public E set(int index, E el)	Replaces the element at the specified position (index) in this list with the specified element el. It returns the replaced element.
public void clear()	Removes all of the elements from this list. The list will be empty after this call returns.

Note: Instead of using the remove() method of an ArrayList object, one may use the remove() method of either an iterator object or a listIterator object to remove an element from a linked list:

```

Iterator<T> it = list.iterator();
it.next();    // skip over the first element
it.remove(); // remove the first element

```

#### 5. Methods to retrieve element from an ArrayList

public T get(int index)	Returns the object at position index in the invoking ArrayList object.
List<E> subList(int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive

#### 6. Methods to investigate the contents of an ArrayList.

public boolean contains(Object obj)	Returns true if the invoking ArrayList object contains the object obj; otherwise it returns false.
public boolean containsAll(Collection <?> c)	This method is inherited from the AbstractCollection class. It returns true if the invoking list object contains all of the elements of collection c.
public boolean equals(Object obj)	Returns true if and only if the specified object is also a list, both lists have the same size, and all corresponding pairs of elements in the two lists are <i>equal</i>
public boolean isEmpty()	This method is inherited from the AbstractCollection class. It returns true if the invoking list object is empty.
public int indexOf(Object obj)	Returns the index of the first occurrence of obj in the invoking List object, or -1 if the object does not contain obj.
public int lastIndexOf(Object obj)	Returns the index of the last occurrence of obj in the invoking List object, or -1 if the object does not contain obj.
public int size()	Returns the number of objects in the invoking

ArrayList object.
-------------------

7. Methods to return an array containing the elements of the ArrayList.

<code>public Object[ ] toArray()</code>	Returns an array containing all of the elements in this list in the correct order.
<code>Public &lt;T&gt; T[ ] toArray(T[ ] array)</code>	Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

8. Methods to indicate the size and capacity of the arraylist

<code>public int size()</code>	Returns the number of elements in this list.
<code>public void ensureCapacity (int minCapacity)</code>	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

### Example 2

```
import java.util.*;
class SplitList {
public static void main(String[] args) {
    ArrayList<Integer>a=new ArrayList<Integer>();
    ArrayList<Integer>b=new ArrayList<Integer>();
    ArrayList<Integer>c=new ArrayList<Integer>();
    Random rd=new Random();
    for (int i=0;i<20;i++)
        a.add(new Integer(rd.nextInt(101)-50)); // generates a random number between -50 and 50

// Go over the elements of a and split them. Put negative numbers in b and positive numbers in c.
// Once done, use the new for loop to print the elements of each arrayList.
    Iterator<Integer> ia = a.iterator(); // iterator for arrayList a
    while (ia.hasNext()) {
        int x= ia.next(); // automatic unboxing
        if(x<0)
            b.add(x); // automatic boxing
        else
            c.add(x); // automatic boxing
    }
    System.out.println("The original list");
    for (Integer i: a)
        System.out.println(i);
    System.out.println("The list of negative numbers");
    for (Integer i: b)
        System.out.println(i);
    System.out.println("The list of positive numbers");
    for (Integer i: c)
        System.out.println(i);
}
```

```
}}
```

### Exercise 1

The Sieve of Eratosthenes is a classic algorithm to find all primes numbers between 2 and  $n$ . The algorithm is as follows:

- 1-Begin with an ArrayList of integers from 2 to  $n$ .
- 2-The first integer, 2, is the first prime.
- 3- Go over the remaining elements and remove every multiple of this prime number.
- 4- take the next integer as the next prime number and repeat step 3.
- 5- stop the algorithm when the next prime number is greater than  $\sqrt{n}$

After all removals are done, the remaining numbers are prime numbers between 2 and  $n$ .

Implement the above algorithm using an arrayList with numbers from 2 to 100. After finding the prime numbers, print them.

### Exercise 2

Develop an application that maintains a linked list of books. The class Book has a book number, title, and author. Your application displays one book at a time. You should have options for displaying the next book, displaying the previous book, adding a new book, removing the displayed book, and updating the displayed book. Use an arrayList to store your book objects.

# Lab 11: Threads

**Objectives:** In this lab, the following topics will be covered

1. About Threads
2. Creating your first thread: Counter
3. Using threads:
4. Timer application.
5. Animation.
6. Exercises.

## 1. Threads

A thread can be thought as an execution path within a process. A process (i.e. a Java application) will at least have one running thread, the thread main. Up to this point, we have been creating single threaded java applications. However, almost all computer application are multi-threaded.

When creating more than one thread, these threads are run simultaneously, the CPU will switch from one thread to another very quickly that programmer would have the feeling that they run in parallel.

## 2. Creating your first thread: Counter

We will create a class Counter with one constructor that has the time to count down in seconds. To make the class a thread, it must extend the java.lang.Thread class.

Example:

```
public class Counter extends Thread
{
    int time;
    public Counter(int seconds)
    {
        time = seconds;
    }

    public void run()
    {
        while(time > 0)
        {
            System.out.println(Thread.currentThread().getName()+"time);
            try
            {
                Thread.currentThread().sleep(1000);
                time--;
            }
            catch(InterruptedException ex)
            {
                System.out.println("error :"+ex.getMessage());
            }
        }
    }
}
```

```
}
```

And to test this class with three different threads

```
public class MainThread
{
    public static void main(String[] args)
    {
        Counter c1 = new Counter(10);
        Counter c2 = new Counter(4);
        Counter c3 = new Counter(2);
        c1.start();
        c2.start();
        c3.start();
    }
}
```

### Note

The start() method for a thread class will invoke its run method. The "run" method for a thread class is the place from which thread execution starts.

Another way of creating a thread is by implementing the interface Runnable.

The counter example looks like

```
public class Counter implements Runnable
{
    ...
}
```

Accordingly the test class will be modified to the following:

```
Thread obj = new Thread (runnableObject);
```

We note that Counter class is a class that implements Runnable.

```
public class MainThread
{
    public static void main(String[] args)
    {
        Thread c1 = new Thread(new Counter(10));
        Thread c2 = new Thread(new Counter(4));
        Thread c3 = new Thread(new Counter(2));
        c1.start();
        c2.start();
        c3.start();
    }
}
```

### 3. Using Threads

#### Timer Application

Examples of threads are those applications that need a timer; think of an application of a window that will close in 5 seconds!

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class TimedBomb extends JFrame implements Runnable
{
    int time = 5;
    JLabel lbl = new JLabel("Will Self-destruct In :"+time);
    public TimedBomb()
    {
        super("Self Destruction");
        setSize(300,100);
        add(lbl);
        setVisible(true);
        Thread t = new Thread(this);
        //this refer to this class, as it implements runnable
        t.start();
    }
    public void run()
    {
        while(time > 0)
        {
            try
            {
                Thread.sleep(1000);
                time--;
                lbl.setText("Will Self-destruct In :"+time);
            }
            catch(InterruptedException e)
            {
            }
        }
        JOptionPane.showMessageDialog(this,"Window is closing.. good bye!");
        System.exit(0);
    }
    public static void main(String[]args)
    {
        new TimedBomb();
    }
}}
```



## Animation and Graphics

If we want an object to move on the screen, it will have to be redrawn continuously in a different location. If we are to put this in a traditional loop, it will be very fast that the user might not be able to observe the moving object. Hence Threads are used to slow down the execution of the loop.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class SalamShabab extends JFrame
{
    String salam = "Salam";
    String shabab= "Shabab";

    int y_salam = 50;
    int y_shabab = 75;
    int x_salam = 10;
    int x_shabab= 100;
    int salam_inc = 1;
    int shabab_inc=-1;

    public SalamShabab()
    {
        setSize(300,100);
        setVisible(true);
    }

    public void paint(Graphics g)
    {
        super.paint(g);
        g.setFont(new Font("Times New Roman",Font.BOLD,20));
        g.drawString(salam , x_salam , y_salam);
        g.drawString(shabab , x_shabab , y_shabab);

        x_salam = x_salam + salam_inc;
        x_shabab = x_shabab + shabab_inc;

        System.out.println(x_salam + " " + x_shabab);

        if(x_salam == getWidth()-60 || x_salam == 0)
        {
            salam_inc = salam_inc* (-1);
        }
        if(x_shabab == getWidth()-60 || x_shabab == 0)
        {
            shabab_inc = shabab_inc * (-1);
        }
    }
}
```

```

        try
        {
            Thread.sleep(10);
        }
        catch(InterruptedException r)
        {
            System.out.println(r.getMessage());
        }
        repaint();
    }

    public static void main(String[]args)
    {
        new SalamShabab();
    }
}

```

### Exercises

1. Create an array of 10 Counter threads, so that they will have a race. When a thread finishes it should print its name and its place in the race (finishing 1<sup>st</sup>, 2<sup>nd</sup> ... etc)



```

C:\Program Files\Xinox Software\JCreatorV3 LE\GE2001.exe
Thread-2 Place1
Thread-1 Place2
Thread-0 Place3
Press any key to continue...

```

2. Modify the Salam Shabab example, so that the strings will move from top to bottom.
3. Add a pause/resume button to the Salam Shabab that will stop/resume the movement of the text.