Al Imam Mohammad Ibn Saud Islamic University
College of Computer and Information Sciences
Computer Science Department

Tutorials

CS151 Object-Oriented Programming

(last updated: January 2019)

Table of Contents

Note to students

This tutorial exercises are prepared for tutorial sessions of the course CS141 – Computer Programming 2.

To take benefit from tutorial sessions, students must work continuously and must feel committed to the course during the whole semester. To do so, students have to:

- Before the tutorial: Review the course, read, understand, and, eventually, start answering the tutorial exercises.
- During the tutorial:
    - Pay attention to the instructor explanation and try learning how to deal with the problems and how to resolve them gradually
    - Take notes about the solution done during the tutorial session because sometimes the provided solution may be different from the shared one.
    - Participate to the tutorial session by asking pertinent questions and answering the instructor questions
    - Compare your eventual solution with the one provided by the instructor
- After the tutorial: try to resolve again the exercises of the tutorial especially those that you haven't correctly resolved, or you haven't resolved at all before the tutorial and compare again your solution to the one proposed by the instructor.

Each tutorial in this document is intended to be done in one tutorial session (2 hours) unless if it is indicated otherwise in the detailed course schedule. After the end of each tutorial session, the course instructor provides a soft copy of the detailed solution of the exercises to students.

We are happy to have you among our students and we wish you a successful course with an excellent level of learning and practicing.

Tutorial 1 – Classes & Objects

Topics:

- Classes v/s Objects
- Instance Variables
- set and get Methods
- private v/s public
- Default and Explicit Initialization for Instance Variables
- Constructor, Default and No-Argument Constructor

Exercise 1 (textbook 7.12)

(`Employee` Class) Create a class called `Employee` that includes three instance variables—a first name (type `String`), a last name (type `String`) and a monthly salary (`double`). Provide a constructor that initializes the three instance variables. Provide a *set* and a *get* method for each instance variable. If the monthly salary is not positive, do not set its value. Write a test app named `EmployeeTest` that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10% raise and display each `Employee`'s yearly salary again.

Exercise 2 (textbook 7.13)

(`Date` Class) Create a class called `Date` that includes three instance variables—a month (type `int`), a day (type `int`) and a year (type `int`). Provide a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a *set* and a *get* method for each instance variable. Provide a method `displayDate` that displays the month, day and year separated by forward slashes (/). Write a test app named `DateTest` that demonstrates class `Date's` capabilities.

Exercise 2 (textbook 7.22)

(Target-Heart-Rate Calculator) While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (www.americanheart.org/presenter.jhtml?identifier=4736), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. [Note: These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. **Always consult a physician or qualified health-care professional before beginning or modifying an exercise program.**] Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* methods. The class also should include a method that calculates and returns the person's age (in years), a method that calculates and returns the person's maximum heart rate and a method that calculates and returns the person's target heart rate. Write a Java app that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

Tutorial 2 – Classes, Objects & Other Concepts

Topics:

- Overloaded Constructors
- This reference
- static fields and methods
- static Import
- final instance variable
- Package Access

Remarks:

- Use this reference whenever you access to an instance variable or to call a constructor from another
- Use the static Import with the used System static members
- No need to have practice on the Package Access because we just have to remove an access modifier of any member, then it can be accessible by the test program which is in the same package.

Exercise 1 (textbook 8.4)

(*Rectangle Class*) Create a class `Rectangle` with attributes `length` and `width`, each of which defaults to 1. Provide a No-Argument constructor and another constructor that initializes the two instance variables.

Provide a set and a get method for each instance variable. Provide methods that calculate the rectangle's perimeter and area. It has *set* and *get* methods for both `length` and `width`. The *set* methods should verify that `length` and `width` are each floating-point number larger than 0.0 and less than 20.0 (these two values must be declared constants).

Write a program to test class `Rectangle`.

Exercise 2 (updated version of textbook 8.4; Exercise 01)

(*Rectangle2 Class*) Create a class `Rectangle2` which is a copy of Rectangle of the Exercise 01. In this version, each object has its own min and max constant values for `length` and `width`, that they default to 0.0 and 20.0.

Add 2 new constructors; the first takes as parameter another object of type Rectangle2, and the second takes 4 parameters which are the length and width of the rectangle, and the min and max values. Update the existing constructors to adhere to the new requirement.

Write a program to test the new capabilities added by class `Rectangle`.

Exercise 3 (extended version of textbook 7.11 that was given in Lab 01)

(*`ExtendedInvoice` Class*) This exercise extends the `Invoice` class described in Exercise 1 of the Lab 01. Recall that the `Invoice` class have 4 instance variables (`partNumber`, `partDescription`, `quantity`, `pricePerItem`), a no-argument constructor, another constructor that initializes the four instance variables, set and get method for each instance variable, and `getInvoiceAmount` method.

We need to use a `static` variable `totalInvoiceAmount` to store the all invoice amounts inside the class.

Provide the following `static` methods

- `resetTotalAmount` : resets the value of `totalInvoiceAmount` to 0.
- `updateTotalAmount`: `private` method that takes 2 parameters; subtracts the first parameter from the `totalInvoiceAmount` and adds the second to it.
- `getTotalAmount`: returns the value of `totalInvoiceAmount`.

Modify the existing constructor with arguments to call the static `updateTotalAmount`, where the first parameter is 0 and the second is the invoice amount (don't call `getInvoiceAmount`).

Modify the existing `setQuantity` to call the static `updateTotalAmount`, where the first parameter is the old value of the invoice amount and the second is the new invoice amount using the new value of the quantity. Similarly, modify `setPricePerItem` by considering the new value of `setPricePerItem`.

Based on `InvoiceTest` class, write a test app named `ExtendedInvoiceTest` that demonstrates class `ExtendedInvoice`'s new capabilities.

Tutorial 3 – Relationships among classes: Composition

Topics:

- Composition

Exercise 1

A Point class models a 2D point at (x,y), as shown in the code* on the next page.

In addition to the constructors, setters, and getters, the class `Point` define a special setter/getter method `setXY`/`getXY` that sets/gets the coordinates x and y at once. [Note that the getXY returns a 2-element array]. Also, the class `Point` propose 3 overloaded methods `distance` that calculates the distance from 2 points using the formula: $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$.

We say that "A line is composed of two points", create a new class called `Line`, by re-using the Point class. Provide constructors, getters and setters, toString (`"Line[begin=(x1,x2),end=(x2,y2)]"`) method, and a method that returns the length of the line (hint: use `distance` method from `Point` class).

Write a program to test class `Line`.

```
/*
 * The Point class models a 2D point at (x, y).
 */
public class Point {
   // The private instance variables
   private int x, y;

   // The constructors (overloaded)
   public Point() {  // The default constructor
      this.x = 0;
      this.y = 0;
   }
```

* https://www.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html#pointclass

```
  public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // The public getters and setters
   public int getX() {
      return this.x;
   }

   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return this.y;
   }
   public void setY(int y) {
      this.y = y;
   }

   // Return "(x,y)"
   public String toString() {
      return "(" + this.x + "," + this.y + ")";
   }

   // Return a 2-element int array containing x and y.
   public int[] getXY() {
      int[] results = new int[2];
      results[0] = this.x;
      results[1] = this.y;
      return results;
   }

   // Set both x and y.
   public void setXY(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Return the distance from this instance to the given point at (x,y).
   public double distance(int x, int y) {
      int xDiff = this.x - x;
      int yDiff = this.y - y;
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
   }
   // Return the distance from this instance to the given Point instance
(called another).
```

```
  public double distance(Point another) {
       int xDiff = this.x - another.x;
       int yDiff = this.y - another.y;
       return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
    }
    // Return the distance from this instance to (0,0).
    public double distance() {
       return Math.sqrt(this.x*this.x + this.y*this.y);
    }
}
```

Tutorial 4 – Relationships among classes: Inheritance

Topics:

- **Inheritance (extends, protected, super, override)**

Exercise 1

A class called Circle is designed as follow (the code on the next page*):

- Two private member variables: a radius (double) and a center (an instance of Point class, which ***we described in the previous Tutorial in Exercise 1***).
- The constructors, public getters, setters, and methods getCenterX, setCenterX, getCenterY, setCenterY, getCenterXY, setCenterXY, etc.
- A toString method that returns a string ("Circle[center=(x,y),radius=r]") describing this instance. We re-use the Point's toString() to print "(x,y)".
- Methods getArea and getCircumference that compute respectively the area ($\pi * radius^2$) and the circumference ($2 * \pi * radius$).
- A distance method that returns the distance from the center of this instance to the center of the given Circle instance (called another).

Create a subclass `Cylinder` derived from the superclass `Circle`. It has a height, its own constructors, toString method, and getArea method ($c * h + 2 * b$). In addition, a method that compute the volume ($b * h$).

Write a program to test class `Cylinder`, where:

- **c: base circumference (circumference of the base circle)**
- **h: height**
- **b: base area (area of the base circle)**

* https://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html#zz-2.1

```
/*
 * The Circle class composes a Point (as its center) and a radius.
 */
public class Circle {
    // The private member variables
   private Point center;  // Declare an instance of the Point class
   private double radius;

    // Constructors
   public Circle() {
      this(new Point(), 1.0); // Construct a Point at (0,0)
   }
   public Circle(double radius) {
      this(new Point(), radius); // Construct a Point at (0,0)
   }
   public Circle(int xCenter, int yCenter, double radius) {
      this (new Point(xCenter, yCenter), radius); // Construct a Point at
(xCenter,yCenter)
   }
   public Circle(Point center, double radius) {
      this.center = center;  // The caller constructed an Point instance
      this.radius = radius;
   }
   // Getters and Setters
   ...
   public Point getCenter() {
      return this.center;  // return a Point instance
   }
   public void setCenter(Point center) {
      this.center = center;
   }

   ...

   public String toString() {
      return "Circle[center=" + center + ",radius=" + radius + "]";  // invoke
center.toString()
   }

   public double getArea() {
      return Math.PI * radius * radius;
   }

   public double getCircumference() {
      return 2.0 * Math.PI * radius;
   }

   // Return the distance from the center of this instance to the center of
```

```
   // the given Circle instance called another.
   public double distance(Circle another) {
      return center.distance(another.center);   // Invoke distance() of the
Point class
   }
 }
```

Tutorial 5 – Polymorphism (part 1)

Topics:

- Demonstrating Polymorphism
  - Use direct and indirect superclass reference to subclass object
  - Use array of superclass reference to subclass objects
  - Send a subclass object to a method that receives a superclass reference

Exercise 1

Consider the 7 classes provided with this exercise which are: `Shape`, `TwoDimensionalShape`, `Circle`, `Rectangle`, `Square`, `Triangle`, and `Point`.

Analyze the definition of these classes and draw a simple class diagram (with only class names).

Note: Expressions used to compute the areas and the circumferences are:

Shape	Area	Perimeter
Circle	π * radius^2	2 * π * radius
Rectangle	length * width	2 (length + width)
Square	side^2	4 * side
Triangle	(base * height)/2	side a + base + side c

Exercise 2

Write a program to demonstrate the polymorphism aspects using the different shape classes. You can proceed as follow:

- Declare/instantiate an object of type `Shape`, call `toString` and `getArea`
- Declare/instantiate an object of type `TwoDimensionalShape`, call `toString` and `getArea` (using superclass reference), and call `getPerimeter`

- Declare/instantiate an object of type `Rectangle`, and call `toString`, `getArea`, and getPerimeter (using superclass reference only)
- Declare/instantiate objects of type `Circle`, `Sqaure`, and `Triangle`, call `toString` and `getArea` (using indirect superclass reference), and call `getPerimeter` (using direct superclass reference)

Exercise 3

Create a class called `Drawing`. This class contains a set (array) of shapes which initialized using its unique constructor. In addition, define the following methods:

- `getShapes`: returns the reference to the array of shapes.
- `updateShape`: takes a Shape object and an index, put the object in the array at the specified index, returns `true` if the index is correct, or `false` otherwise.
- `shapeAt`: takes an index, returns the `Shape` object from the array at the specified index if the index is correct, or `null` otherwise.
- `getShapeArea`: takes an index, returns the `Shape` object's area from the array at the specified index if the index is correct, or `0.0` otherwise.

Exercise 4

Write a program to demonstrate the polymorphism aspects using the different shape classes and `Drawing`. You can proceed as follow:

- Declare/instantiate an array of 4 `Shapes`.
- Create one object from each of the following types: `Shape`, `TwoDimensionalShape`, `Circle`, and `Triangle`.
- Insert the created objects into the array.

- Declare/instantiate an object of type `Drawing` using the previously defined array.
- Get the list of shapes from the `Drawing` object using `getShapes` and print their information (by calling `toString` and `getArea`)
- Using `updateShape`, create/put new object of type `Rectangle` in the position `0`, and another of type `Square` in the position `1`.
- Find the newly inserted objects using `shapeAt` and print their info

Exercise 5

Answer the following questions:

- Why we cannot call the `getPerimeter` method?
- If we change the array type will solve the problem completely?
- Is there any other way to solve the problem without any modification?

Tutorial 6 – Polymorphism (part 2)

Topics:

- Abstract Classes and Methods
  - Create and use abstract classes and methods
  - Perform downcasting
  - Use of the instanceof operator

Exercise 1

The abstract class is the class which we never intend (or it is not useful) to create objects of this class.

Considering the shape classes of the Tutorial 05, which are the classes that you qualified to be abstract? Justify your answer.

Exercise 2

Declare the classes Shape and TwoDimentionalShape as abstract classes and compile/execute the test programs (ShapeTest and DrawingTest).

Identify the eventual compile-time errors (if any) and try to resolve them.

Exercise 3

An abstract class normally contains one or more abstract methods. The abstract method is a method that do not provide implementation

Focus the classes Shape and TwoDimentionalShape, and try to identify which methods that you qualified to be abstract? Justify your answer.

Exercise 4

Declare the getArea() from Shape and getPerimeter() from TwoDimentionalShape as abstract methods and compile/execute the test programs (ShapeTest and DrawingTest).

Identify the eventual compile-time errors (if any) and try to resolve them.

Exercise 5

Update the DrawingTest program in order to print the perimeter by downcasting the Shape's instance.

Declare the getArea() from Shape and getPerimeter() from TwoDimentionalShape as abstract methods and compile/execute the test programs (ShapeTest and DrawingTest).

Identify the eventual compile-time errors (if any) and try to resolve them.

Problem (Homework)

Consider the following UML class diagram.

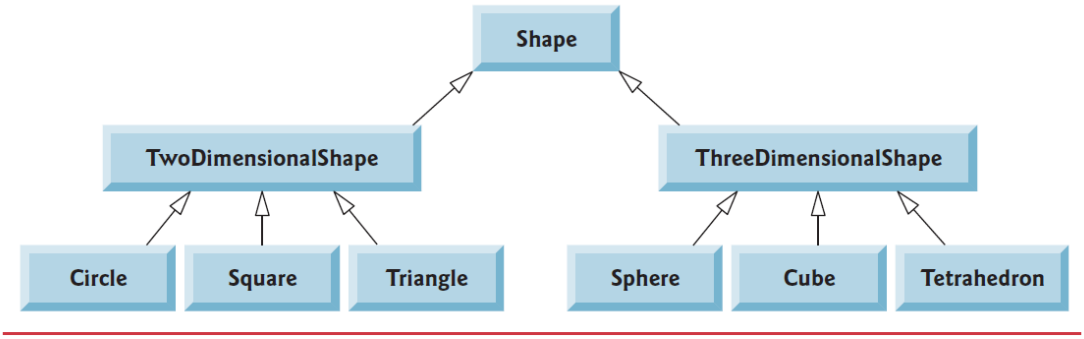


Fig. 9.3 | Inheritance hierarchy UML class diagram for Shapes.

1. Identify which classes are not yet implemented in Tutorial 5.
2. Create the missing classes starting by the abstract classes, then the concrete classes.
3. We need to define a method called getVolume to compute the volume of the 3D objects. Place this method in the most appropriate class and, when needed, declare it as abstract or concrete method.
4. Write a program to test your newly added shapes.
5. Update the DrawingTest program to test the specifities of the newly added classes and their methods.

Note: To compute the areas and volumes use the following expressions:

Shape	Area	Volume
Sphere	4 * π * radius^2	$\frac{4}{3}\pi r^3$
Cube	6 * side^2	side^3
Tetrahedron	$\sqrt{3}\ a^2$	$\frac{a^2}{6\sqrt{2}}$

Tutorial 7 – Polymorphism (part 3)

Topics:

- `final` Methods and Classes
- Creating and Using Interfaces

Exercise 1

A final class cannot have any subclass.

Considering the shape classes of the Tutorial 06 & 07, which are the classes that you qualified to be final? Justify your answer.

Exercise 2

Declare the class Point and all shapes classes that haven't subclasses as final. Then, compile/execute the test programs (ShapeTest, DrawingTest, Drawing2Test).

Identify the eventual compile-time errors (if any) and try to resolve them.

Exercise 3

A final method cannot be overridden by subclasses.

Focus the classes Shape, TwoDimentionalShape, ThreeDimensionalShape, and tell whether we can declare getArea, getPerimeter, getVolume as final or not. Justify your answer.

Exercise 4

Declare the method getPerimeter as final in the class Rectangle. Then, compile/execute the test programs (ShapeTest and DrawingTest).

Identify the eventual compile-time errors (if any) and try to resolve them.

Try to override getPerimeter in the Square class and read the error generated by the compiler.

Exercise 5

An interface defines a list of public abstract methods that have to be implemented in the subclasses. The presence of the keywords public and abstract is optional.

1. Define an interface Movable declaring 4 movements methods:
    - moveUp: subtract 1 from y coordinate
    - moveDown: add 1 to y coordinate
    - moveRight: add 1 to x coordinate
    - moveLeft: subtract 1 from x coordinate
2. Make all three-dimensional shape classes implements the interface Movable by adjusting the position point coordinates.

Exercise 6

1. Define an interface Resizable declaring 1 method:
    - resize(int percent): modifies the dimension (such as radius) by the given percentage.
2. Make all two-dimensional shape classes implements the Resizable interface by adjusting all of its measures.

Tutorial 8 – Exception Handling & Assertions

Topics:

- Exception handling
  - `try/catch/finally` blocks
  - `throws` and `throw` keywords
- Custom Exceptions: declare new Exception type
- Assertions

Exercise 1

Consider the provided class ReadTextFile that tries to read a file and prints its formatted content on the standard output.

Some of the used methods related to file processing are throwing exceptions. Try to identify the different exception wrap the concerned code with try/catch blocks and handle it conveniently.

This is the meaning of the exceptions that may be thrown (the last two are RuntimeException):

- IOException: error occurs when opening a file.
- NoSuchElementException: error occurs when file not formatted as expected
- IllegalStateException: error occurs when reading from file.

We want also to assure that the file to be closed even though if an uncaught unchecked exception is thrown. For that, wrap the concerned code with try/finally blocks.

Exercise 2

Create a class called HexToDec with a main method that asks the user to enter a string in hexadecimal format, then calculates and displays its decimal value. We want to throw a custom exception when the user enters a non-hexadecimal character.

You can proceed as follow (write only one file called HexToDec.java):

1. Create a class InvalidHexException, a subclass of RuntimeException
2. Create a class HexToDec that contains:
    a. A static method convertToInt that receives a string (supposed) in hexadecimal format, and returns an array containing the different integers representing each of the hexadecimal characters separately. This method throws an InvalidHexException if one of the string characters is not a hexadecimal character.
    b. A static method called convertToDec that receives an array of integers (the one produced by convertToInt) and compute the decimal value based on the elements of the received array.
    c. A main method that asks the user to enter string then passes it to convertToInt. After that, it calls convertToDec with the array of integers returned by convertToInt. Finally, display the resulting decimal value.

Wrap the code in the main method with try/catch blocks

Exercise 3

Assertions are primary used to detect bugs in a program. They should not be used to check the parameter of a public method. It is used to test/confirm developer assumptions about a program.

Examples where to use assertions*:

- Check the parameter of a private method
- In last else of nested if/else:

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
```

* https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

```
    ...
}
```

- In switch that has no default case:

```
switch (i % 3) {
   case 0:
    ...
   case 1:
    ...
   case 2:
    ...
   default: assert false : "Erroneous remainder.";
}
```

In this exercise, you are asked to write a program to check if a diagonal matrix (all its elements are zeros except of the diagonal) is an identity matrix (a diagonal matrix with only ones in the diagonal). Proceed as follow:

- Write a method called generateDiagonalMatrix that takes a one-dimensional array, creates a two-dimensional array, put the elements of the received array in the diagonal of the newly created array, and return this later.
- Write another method called isIdentityMatrix that takes a diagonal matrix and check whether the matrix is identity matrix or not.
- Write an assertion statement in the method isIdentityMatrix in the case of a non-diagonal element is different from zero.

Test your program and try to make the assertion statement to be executed

Tutorial 9 – Generic Classes and Methods

Topics

- Generic Methods
- Generic Classes
- Generic Methods and Bounded Type Parameters

Note

- The following exercises are inspired from Oracle tutorial on generics (https://docs.oracle.com/javase/tutorial/java/generics/index.html)
- As mentioned in the slides, a generic class can have multiple type parameters.
- Also, we've seen that the Comparable<T> interface is generic, so to declare a generic interface follow the same approach used to declare generic class.

Exercise 1

Declare a generic interface called Pair that takes two type parameters K and V. This interface has two methods getKey that returns a value of type K, and getValue that returns a value of type V.

Exercise 2

Write a class OrderedPair implementing the Pair interface. The class has two instance variables which are key (of type K) and value (of type V). Define the constructor of the class OrderedPair that initializes the instance variables.

Exercise 3

In a class named Util, define a generic method, compare, which compares two Pair objects. Based on equals method inherited from the Object class, returns true if the keys and values of the two pairs are equal.

Test your compare method.

Exercise 4

Write and test a generic method countGreaterThan that receives an array from any type that implements the Comparable interface and a parameter e of the same type. The method counts and returns the array elements that are greater than e.

Tutorial 10 – Graphical User Interface – Design

Topics

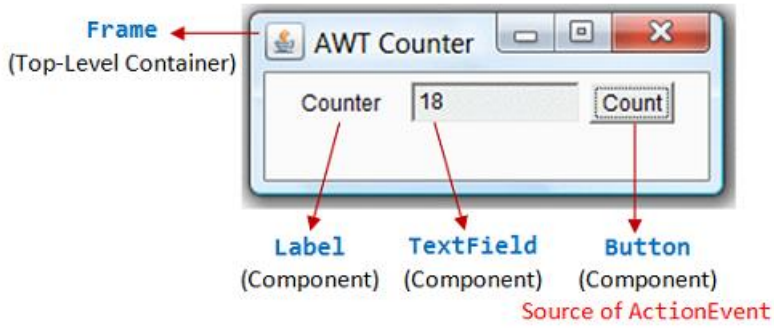
- Components
- Containers
- Layout Managers

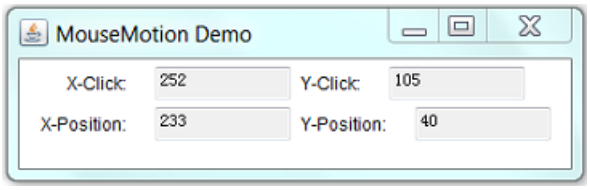
Note

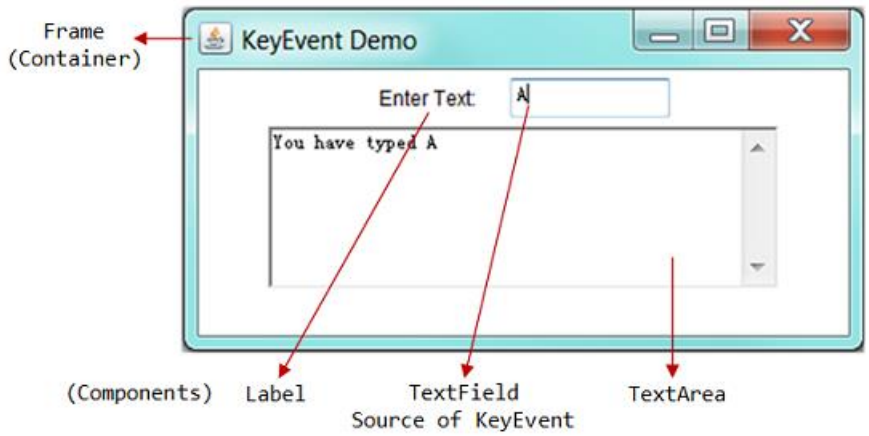
- The following exercises are inspired from: https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html
- For more details about Swing components and features, refer to: https://docs.oracle.com/javase/tutorial/uiswing/TOC.html
- For all of the exercises, you are just requested to design the GUI without Event Hanling

Exercise 1

Reproduce the same interfaces as follow:



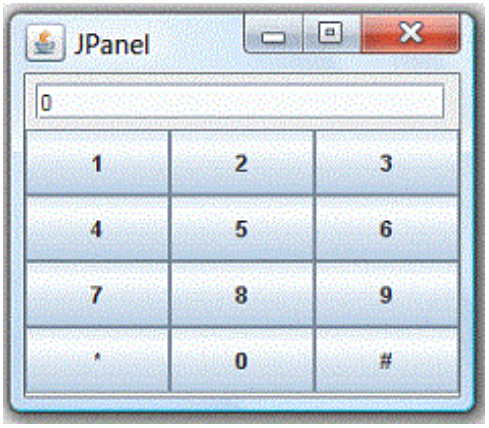




Exercise 2

A JPanel is a rectangular pane, which can be used as sub-container to organized a group of related components in a specific layout (e.g., FlowLayout, BorderLayout). JPanels are secondary containers, which shall be added into a top-level container (such as JFrame), or another JPanel.

Reproduce the same interface as in the following figure which shows a JFrame in BorderLayout containing two JPanels - panelResult in FlowLayout and panelButtons in GridLayout. panelResult is added to the NORTH, and panelButtons is added to the CENTER.



Exercise 3

Design a login interface (username, password, login button).

Tutorial 11 – Graphical User Interface - Event Handling

Topics

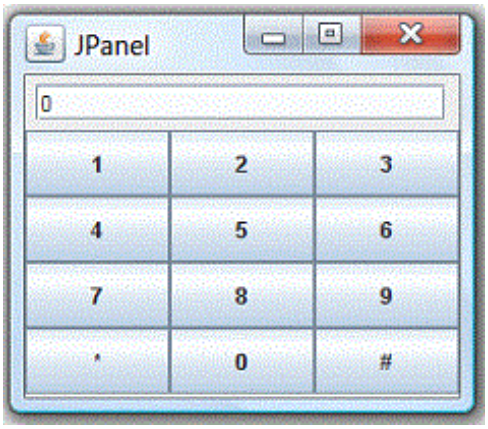
- **Event Handling**
  - **ActionListener**

Note

- The following exercises are inspired from: https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html
- For more details about Swing components and features, refer to: https://docs.oracle.com/javase/tutorial/uiswing/TOC.html

Exercise 1

Consider the designed interface from the Tutorial 09 exercise 2, which represent a telephone keypad.



Using the interface ActionListener on all buttons, write in the JTextField the typed number by accumulating the numbers.

As a second step, format the numbers according to the number of digit as follows:

- Local: 754-3010
- Domestic: (541) 754-3010

- International : +1-541-754-3010