COMP9103: Software Development in Java

W12: Final Exam Review

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY

Exercise: Questions and suggestions

- Please take 10m at the lab to complete the survey
  - Browse to https://student-surveys.sydney.edu.au/students/
  - Log in if you aren't already
  - Complete survey for COMP9103

Final Exam

- Format
  - Written examination
  - 1 x A4 page of own notes (double-sided) allowed
  - The exam will be 6 questions
  - It will draw from lectures and exercises
  - SIT policy: You must get 40% on the exam and 50% overall to pass COMP9103

Class Definition

- A class is defined in Java by using the class keyword and specifying a name for it: e.g. public class Customer{ }
- Inside a class it is possible to define:
  - Fields (sets of values, variables, characteristics)
  - Constructors (used to create and initialize new objects)
  - Methods (operations normally defined on the fields/variables)
- Each of these building blocks is qualified by an access modifier/specifier, such as public, private (to be covered next week)
- Syntax: class definition

```
accessSpecifier class ClassName
{
   fields/variables

   constructors

   methods
}
```

- Meaningful class name
- Class name should be noun
- Each word starts with capital letter

Static Fields

- static fields (with static as the specifier)

```
private static int customerNumber = 1000;
```

- A static field (also called class field) belongs to the class.
  - static fields can be used even when no object created
  - Only one copy and No duplication
  - Values in static fields are shared among all objects created from this class
  - Think of these as some kind of “global variable”, where changes are visible to all instances

ArrayList

- The ArrayList class is a generic class:
  `ArrayList<TypeParameter>` //an array list type
- For example:
  - **ArrayList<Customer>** // an array list of **Customer** type

```
ArrayList<Customer> customerList = new ArrayList< Customer >();
customerList.add(new Customer("Peter", -1276, 423));
customerList.add(new Customer("Mary", -254, 1765));
customerList.add(new Customer("Paul", -3124, 102));
```

  - You can replace *Customer* with any other **class** to get a different array list type
- When you construct an ArrayList object, it has an initial size of 0.
- An arraylist has a set of methods for common operations
  - You can use **add()** method to add an object to the end of the array list.
  - **size()** method returns the current size of the array list.

A code cliché: traversing a collection

– Traversing all elements of an ArrayList object:

```
ArrayList<Customer> customerList= . . . ;
int sum = 0;
for (Customer c : customerList) {
      sum = sum + c.getCreditCardBalance();
}
```

"For each Customer object ***c*** in the ***customerList***"

Finding the Maximum or Minimum

- Initialize a candidate with the starting element
- Compare candidate with remaining elements
- Update it if you find a larger or smaller value.

```
if (!customerList.isEmpty()){
    int max = customerList.get(0).wealth();
    String richestPerson = customerList.get(0).getName();
    for (Customer c : customerList ) {
       if (c.wealth() > max) {
                richestPerson = c.getName();
                max = c.wealth();
       }
    }
}

System.out.println("Richest person is " + richestPerson);
```

Get the starting object in customerList

Invoke its instance method

Finding a Value

- Check all elements until you find the value or reach the end of the array list

```
public Customer find(String name)
  {
     for (Customer c : customerList )
     {
        if (c.getName().equals(name))
              return c; // Found a match return c;
     }
     return null; // No match in the entire array list
  }
```

Using Iterators

```
  List<Product> list1;
...
list1 = new LinkedList<Product>();  // or ArrayList()
...

public void displayProducts()
{
   Iterator<Product> anIterator = list1.iterator();

   while (anIterator.hasNext())
   {
      Product curProduct = anIterator.next();
      // Do something with the element.....
      System.out.println( curProduct.getDescription() );
   }
}
```

Using Set with Comparable Interface

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
…..
@Override
  public int compareTo(Employee o) {
            return name.compareTo(o.name);}
  public String toString() {
            return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
[Employee [id=100, name=Bob],
Employee [id=400, name=Lui],
Employee [id=300, name=Alex],
Employee [id=200, name=Neymar]]
```

```
public static void main(String[] args) {
      HashSet<Employee> hashSet = new HashSet<Employee>();

       hashSet.add(new Employee ( 100, "Bob"));
       hashSet.add(new Employee ( 200, "Neymar"));
       hashSet.add(new Employee ( 300, "Alex"));
       hashSet.add(new Employee ( 400 , "Lui"));
       System.out.println(hashSet);

       TreeSet<Employee> treeSet = new TreeSet<Employee>(hashSet);
       System.out.println(treeSet);
}
```

```
[Employee [id=300, name=Alex],
Employee [id=100, name=Bob],
Employee [id=400, name=Lui],
Employee [id=200, name=Neymar]]
```

Using a Map with Comparable Interface

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
…..
@Override
  public int compareTo(Employee o) {
              return name.compareTo(o.name);}
  public String toString() {
              return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
{400=Employee [id=400, name=Lui],
100=Employee [id=100, name=Bob],
200=Employee [id=200, name=Neymar],
300=Employee [id=300, name=Alex]}
```

```
public static void main(String[] args) {
        HashMap<Integer,Employee> hashMap = new HashMap<Integer,Employee>();
        hashMap.put(100, new Employee ( 100, "Bob" ));
        hashMap.put(200, new Employee ( 200, "Neymar"));
        hashMap.put(300, new Employee ( 300,  "Alex"));
        hashMap.put(400, new Employee ( 400 , "Lui"));
        System.out.println(hashMap);
        TreeMap<Integer, Employee> treeMap = new TreeMap<Integer, Employee>(hashMap) ;
        System.out.println(treeMap);
}
```

```
{100=Employee [id=100, name=Bob],
200=Employee [id=200, name=Neymar],
300=Employee [id=300, name=Alex],
400=Employee [id=400, name=Lui]}
```

Sort by keys

Using a Map with Comparable Interface

```
class Employee implements Comparable<Employee> {
    private int id;
    private String name;
…..
@Override
  public int compareTo(Employee o) {
            return name.compareTo(o.name);}
  public String toString() {
            return "Employee [id=" + id + ", name=" + name + "]";}
}
```

```
300=Employee [id=300, name=Alex]
100=Employee [id=100, name=Bob]
400=Employee [id=400, name=Lui]
200=Employee [id=200, name=Neymar]
```

Sort by values

```
public static void main(String[] args) {
        Iterator<Entry<Integer, Employee>> iterator  =
                        hashMap.entrySet().stream().sorted(Map.Entry.comparingByValue()).iterator();
        while (iterator.hasNext()) {
            Map.Entry<Integer, Employee> entry = iterator.next();
            System.out.println(entry.getKey() + "=" + entry.getValue());
        }
//OR
hashMap.entrySet().stream().sorted(Map.Entry.comparingByValue()).forEach(System.out::println);}
```

Text Files – Reading

- We can use a **Scanner** object to read the contents of humanly readable text files
- Create a new Scanner object and set up its input resource as the file object

```
File file-obj = new File(file-name);
Scanner inputFile = new Scanner(file-obj);
```

- Example:

```
Scanner inputFile = new Scanner ( new File ( "myFile.txt" ) );
String firstLine = inputFile.nextLine();
System.out.println("Data from file: " + firstLine);
```

Text Files – Writing

- We can use a **PrintWriter** object to write contents to humanly readable text files
- Class **PrintWriter** has methods **println(), print() and printf()** that are used in the same way as the methods in **System.out**
- Construct an output stream for writing data to a file

```
File fw = new File(file_name);
E.g. File fw = new File("myFile.txt");
```

- Create a new PrintWriter from an existing File object

```
PrintWriter out = new PrintWriter(fw);
PrintWriter out = new PrintWriter(new FileWriter(fw, true));
```

append the new contents to the end of the file

Write to a Binary File

- An ObjectOutputStream object is wrapped around a FileOutputStream object to write data to a binary file.

```
FileOutputStream fstream = new    FileOutputStream("MyInfo.dat");
ObjectOutputStream outputFile = new  ObjectOutputStream(fstream);
```

- If the file that you are opening with the FileOutputStream object already exists, it will be erased and an empty file by the same name will be created.
- Once the ObjectOutputStream object has been created, you can use it to write binary data to the file.

Example:

```
import java.io.*;
public class WriteBinaryFile
{
        public static void main(String[] args)    throws IOException
        {
            Customer c = new Customer("Tom", 100,500);
            FileOutputStream fstream =   new FileOutputStream("customer.dat");
            ObjectOutputStream out =    new ObjectOutputStream(fstream);
            out.writeObject(c)
            out.close();
      }
}
```

Read from a Binary File

- To open a binary file for input, you wrap a ObjectInputStream object around a FileInputStream object.

```
FileInputStream fstream = new FileInputStream("MyInfo.dat");
ObjectInputStream inputFile = new ObjectInputStream(fstream);
```

- Once the ObjectInputStream object has been created, you can use it to read binary data from the file.

Example

```
import java.io.*;
public class ReadBinaryFile
{
        public static void main(String[] args)    throws IOException
        {
            Customer c = null;
            FileInputStream fstream =   new FileInputStream("customer.dat");
            ObjectInputStream in =    new ObjectInputStream(fstream);
            c = (Customer) in.readObject();
           in.close();
      }
      System.out.println( c ) ;
}
```

Example

- There is a text file named 'bankAccounts.txt' containing the details of several BankAccounts, where the name and balance are each on separate lines. The very first line of the file is a number, which says how many BankAccount objects are described in the file in the lines which follow it.

- Write code for a method named processTextFile which will open a file named 'bankAccounts.txt', from which it will read the data of a number of BankAccounts. It will create BankAccount objects using this data, placing them into an ArrayList.

```
public class Bank {

private ArrayList<BankAccount> accounts;
public Bank(String fileName) {
                accounts = new ArrayList<BankAccount>();
                loadAccounts(fileName);
}
public void loadAccounts(String  fileName) {

String name;
double balance;
Scanner fileScanner = null;
File file = new File(fileName);
 try {

fileScanner = new Scanner(file);
// Determine how many bank accounts there are to read...
int numberOfObjectsToRead = fileScanner.nextInt();
fileScanner.nextLine();// goes to the next line of file, after the int.

for (int index = 0; index < numberOfObjectsToRead; index ++)
        {
           name = fileScanner.nextLine();
           balance = fileScanner.nextDouble();
           fileScanner.nextLine();// goes to next line, after the double.
           // Create an object to represent that bank Account
           BankAccount bc = new BankAccount(name, balance);
           accounts.add(bc);
        }

} catch (FileNotFoundException e) {
e.printStackTrace();
}
 fileScanner.close();
}
```

```
5
Mark
1000.56
Paul
3221.41
John
155.11
Maria
50.2
Rima
3221.41
```

Inheritance (abstract classes and interface)

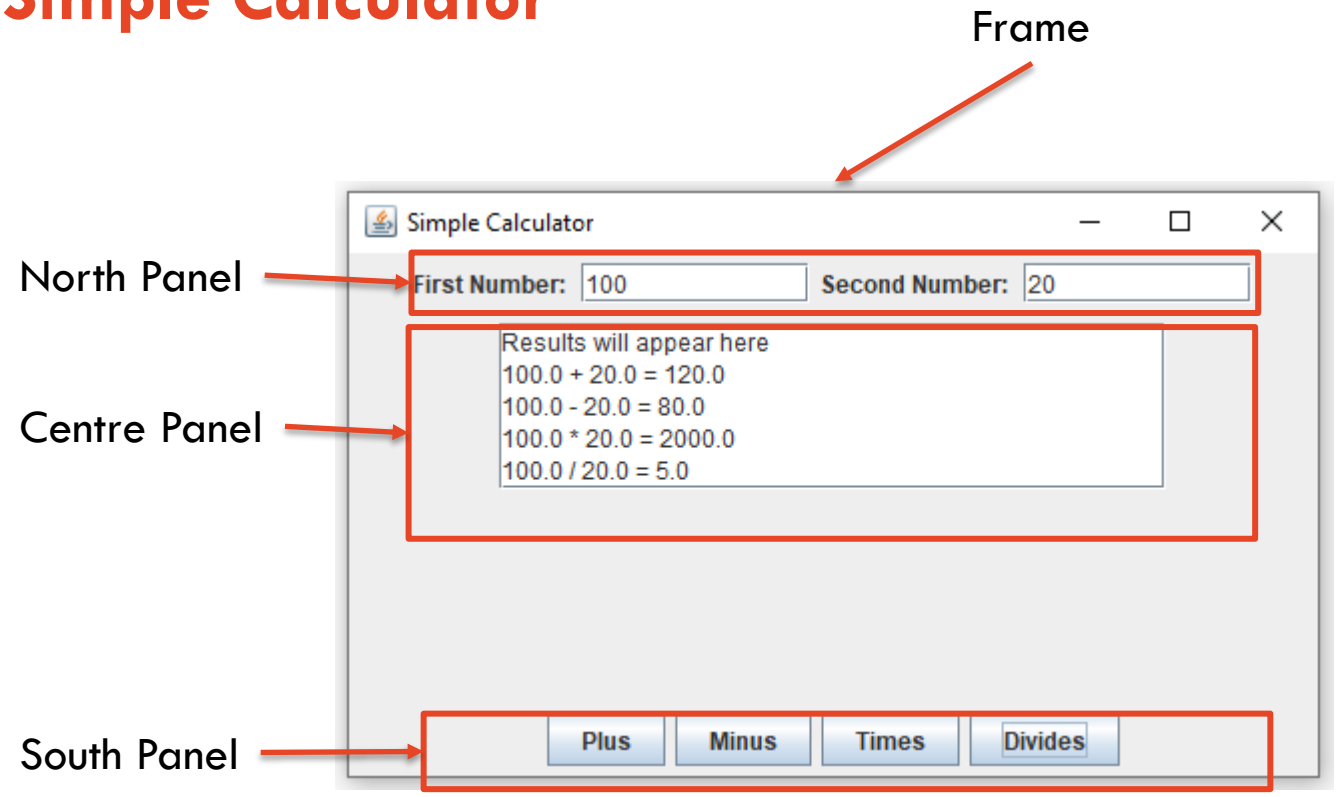
- ***Inheritance*** **is a mechanism by which a new class is derived from an existing class.**
  - The existing class that depicts common and general fields and methods is called as the **base class**, or **superclass**
    - E.g. the ***Person*** class
  - The classes defined as **extensions** of the superclass to **inherit all the fields and methods from the superclass** are called as **subclasses**, or **derived classes**
    - E.g. we define a ***Student*** class on the basis of ***Person*** class,

      then ***Student*** class is a subclass of ***Person*** class
- Inheritance is a powerful way to support **software reuse**.

Creating GUI applications

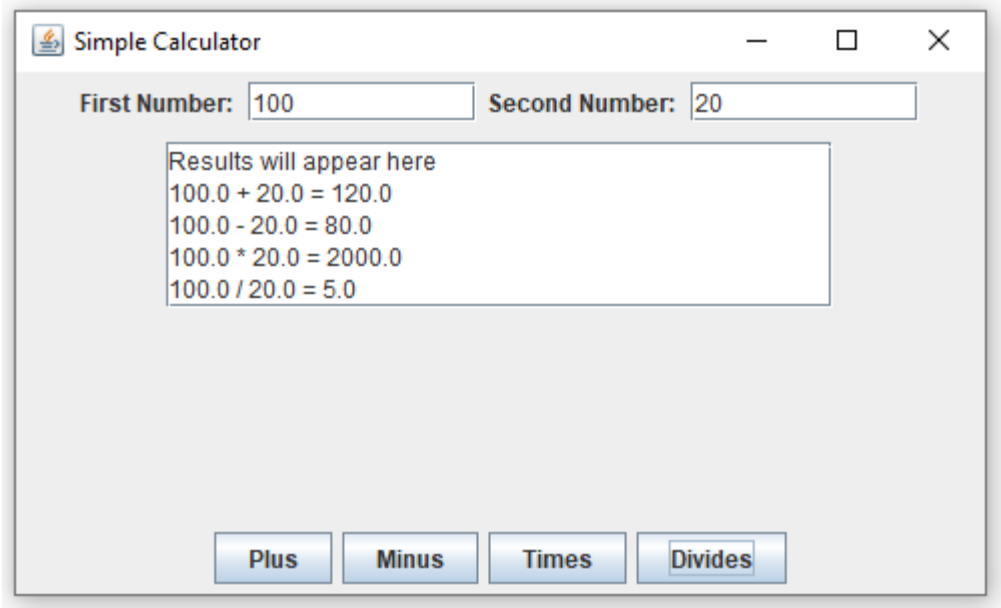
- We need to know
  - How to create components such as buttons, labels, checkboxes, radio buttons, etc.
  - How to assemble them inside a frame according to a preferred arrangement or order of the components.
  - How to perform actions on the components such as
    - *click a button*
    - *select an item from radio button*

Simple Calculator



Example

```
public class GUI {
    private JButton add, min, mul, div;
    private JLabel numL1, numL2;
    private JTextField numT1, numT2;
    private JTextArea outputTextArea;
    private JPanel northPanel, southPanel, centerPanel;
    private JFrame jf;
}
```

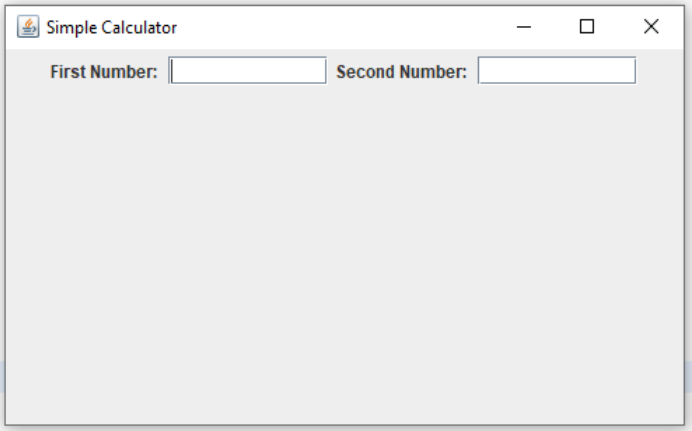


Example

```
public class GUI {
    ....
    public GUI() {
        jf = new JFrame("Simple Calculator");
        northPanel = new JPanel();
        numL1 = new JLabel("First Number: ");
        numL2 = new JLabel("Second Number: ");
        numT1 = new JTextField(10);
        numT2 = new JTextField(10);
        northPanel.add(numL1);
        northPanel.add(numT1);
        northPanel.add(numL2);
        northPanel.add(numT2);
        jf.add(northPanel, BorderLayout.NORTH);
        jf.setVisible(true);
        jf.setSize(500, 300);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

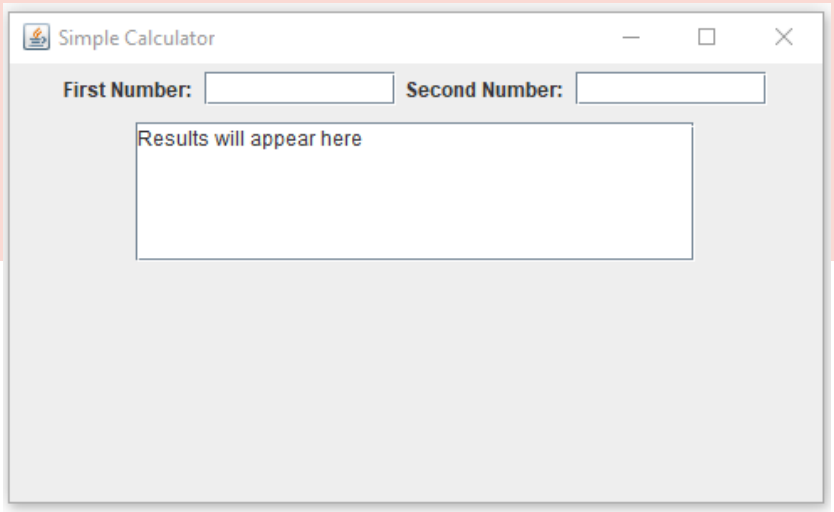
Default layout of JPanel is FlowLayout

```
public class Main {

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



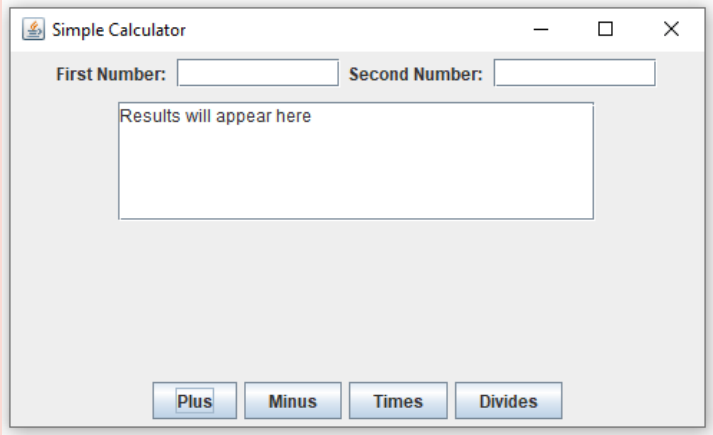
Example

```
public class GUI {
    ....
    public GUI() {
        ....
        centerPanel = new JPanel();
        outputTextArea =new JTextArea("Results will appear here");
        outputTextArea.setColumns(30);
        outputTextArea.setRows(5);
        JScrollPane scrollPane = new JScrollPane(outputTextArea);
        centerPanel.add(scrollPane);
        jf.add(centerPanel, BorderLayout.CENTER);
         ....

        }
}
```



Example

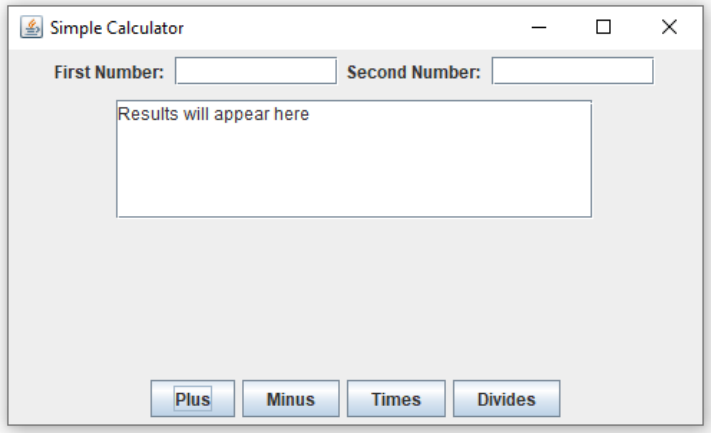
```
public class GUI {
    ....
    public GUI() {
        ....
        southPanel = new JPanel();
        plus = new JButton("Plus");
        minus = new JButton("Minus");
        times = new JButton("Times");
        divides = new JButton("Divides");
        southPanel.add(plus);
        southPanel.add(minus);
        southPanel.add(times);
        southPanel.add(divides);
        jf.add(southPanel, BorderLayout.SOUTH);
        ....
    }
}
```



Example

```
public class GUI  implements ActionListener {
        ……
      public GUI() {
          ……
          JPanel southPanel = new JPanel();
          plus = new JButton("Plus");
          minus = new JButton("Minus");
          times = new JButton("Times");
          divides = new JButton("Divides");
          plus.addActionListener(this);
          minus.addActionListener(this);
          times.addActionListener(this);
          divides.addActionListener(this);
          ……
      }
      @Override
        public void actionPerformed(ActionEvent ae) {

        }
}
```

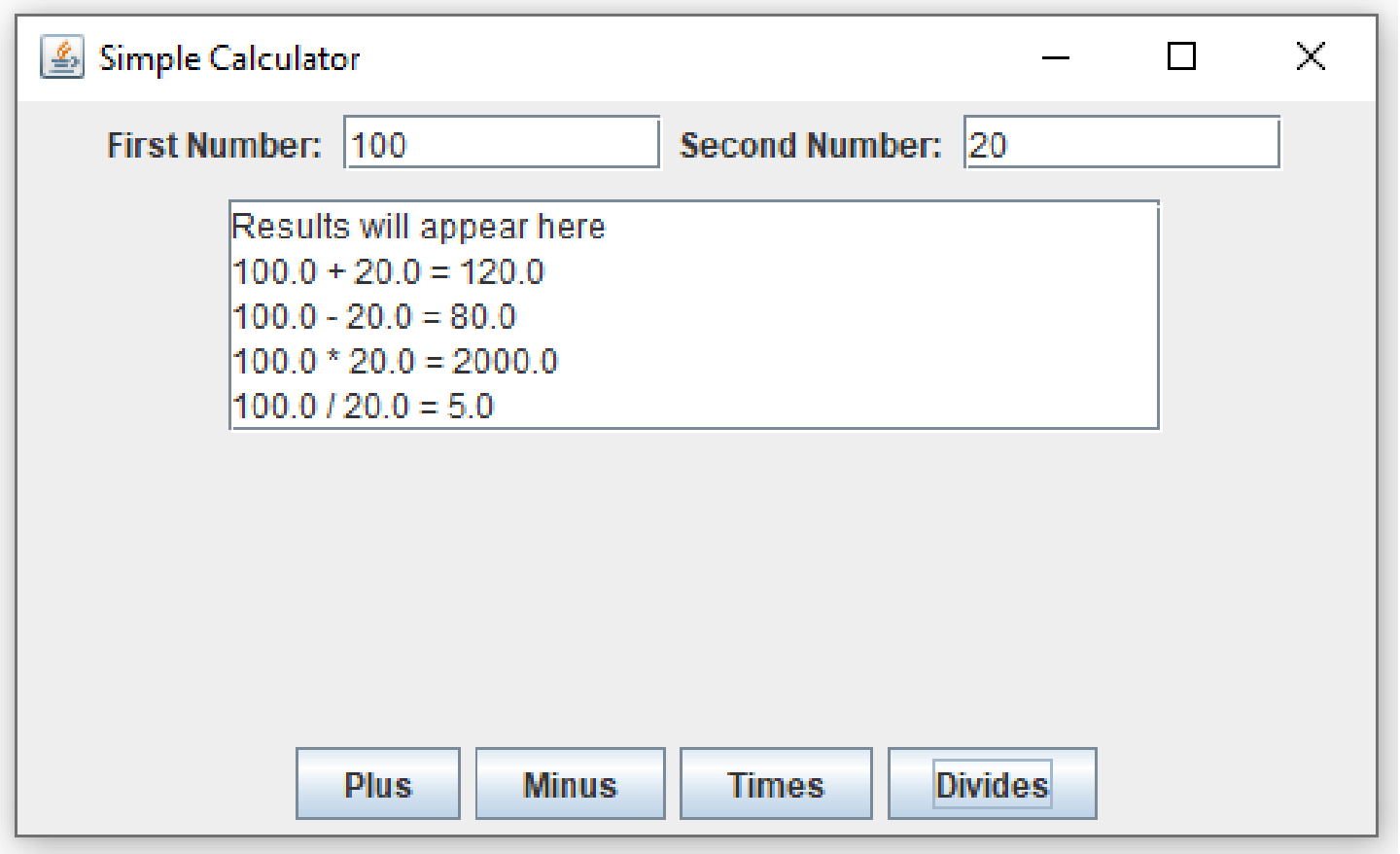
```
public class Main {

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



Example

```
public class GUI  implements ActionListener {
        ……
     @Override
      public void actionPerformed(ActionEvent ae) {
          double num1, num2;
        try {
             num1 = Double.parseDouble(numT1.getText());
             num2 = Double.parseDouble(numT2.getText());
             if(ae.getSource() == plus)
                    outputTextArea.append("\n"+ num1+ " + "+ num2 +" = "+String.valueOf(num1+num2));
             else if(ae.getSource() == minus)
                    outputTextArea.append("\n"+num1+ " - "+ num2 +" = "+String.valueOf(num1 - num2));
             else if(ae.getSource() == times)
                    outputTextArea.append("\n"+num1+ " * "+ num2 +" = "+String.valueOf(num1 * num2));
             else if(ae.getSource() == divides)
                    outputTextArea.append("\n"+num1+ " / "+ num2 +" = "+String.valueOf(num1 / num2));
          }
        catch(Exception e) {
              JOptionPane.showMessageDialog(jf, "Invalid values");
          }
      }
}
```

Example


Simple Calculator
First Number:
100
Second Number:
20
Results will appear here
100.0 + 20.0 = 120.0
100.0 - 20.0 = 80.0
100.0 * 20.0 = 2000.0
100.0 / 20.0 = 5.0
Plus
Minus
Times
Divides

Good Luck for your final exams

THE UNIVERSITY OF SYDNEY