소프트웨어 공학

IT대학 컴퓨터학부
권영우

Programming in Java

- In Java, a program generally has the following structure:
  - Declaring variables/ initializations
    - E.g. **`int`** `balance;`
  - Processing
    - **Read** inputs
    - **Process** inputs
    - **Show** output
  - Exit the program
    - return, exit

Data Types – Variables & Initializations

- **String** for strings of letters
  e.g. `String LastName = "Kwon";`

- **char** for characters
  e.g. `char alphabet = 'A';`

- **byte**, **short**, **int**, and **long** for integer numbers
  e.g. `int number = 3;`

- **float** & **double** for real numbers
  e.g. `double pi = 3.14159;`

- **boolean** for logical
  e.g. `boolean check = true;`

Expressions and Operators

- Expressions:

  ( ), * ,/ , +, -

- Operators:

&&	for	AND
||	for	OR
!	for	NOT
==	for	equal
!=	for	NOT equal

Selection Statements

- if statement ex)

```
if (A>B) {
        System.out.println("A>B");
}
```

- if…else statement ex)

```
if (A>B) {
        System.out.println("A>B");
} else {
        System.out.println("A<=B");
}
```

- switch statement ex)

```
switch (A) {
    case 1: {
        System.out.println("A=1");
        break;
    }
    case 2: {
        System.out.println("A=2");
        break;
    }
    …
    default: {
        System.out.println("A Undefined");
    }
}
```

Repetitions (while, do while, and for)

- Using While:

```
// Variable Declaration & Initialization
int sum = 0;
int count = 0;

// Processing
while (count <= 1000) {
        sum = sum + count;
        count = count +1; //same as count++
}

// Output
System.out.println("Sum: " + sum);
```

Repetitions(while, do while, and for)

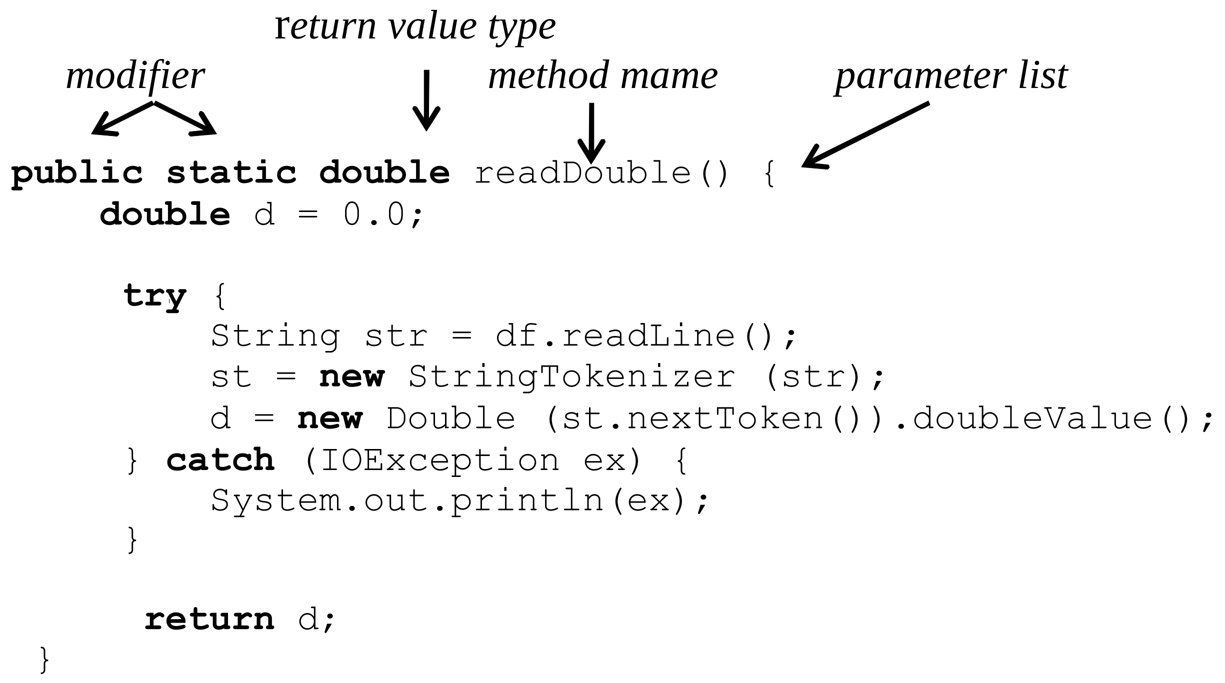
- Using do-while:

```
  // Variable Declaration & Initialization
  int sum =0;
  int count =0;

  // Processing
do {
          sum = sum + count;
          count = count +1; //same as count++
  } while (count <=1000);
```

- Using for:

```
  // Variable Declaration & Initialization
  int sum =0;

  // Processing
  for (int count =0; count <= 100; count++)  {
          sum = sum +count;
  }
```

Methods



```
public static double readDouble() {
    double d = 0.0;

     try {
         String str = df.readLine();
         st = new StringTokenizer (str);
         d = new Double (st.nextToken()).doubleValue();
     } catch (IOException ex) {
         System.out.println(ex);
     }

      return d;
 }
```

Declaring Methods

```
/**
 * This method returns the maximum of two numbers
 * @param num1 the first number
 * @param num2 the second number
 * @return either num1 or num2
 */
int max(int num1, int num2) {
    if (num1 > num2) {
          return num1;
    } else {
          return num2;
    }
}
```

Passing Parameters

```
/**
 * This method prints a message a specified number
 * of times.
 * @param message the message to be printed
 * @param n the number of times the message is to be
 * printed
 */
void nPrintln(String message, int n) {
    for (int i=0; i<n;  i++) {
          System.out.println(message);
    }
}
```

main method

- One of these methods is required for every program.
  - If your program does not have one, then you cannot run it.

- The main method has a very specific syntax:

```
public static void main(String[] args) {
    // Your code goes here!
}
```

An Example Program

```
class MyProgram {
   /**
    * This method prints a message a specified number
    * of times.
    * @param message the message to be printed
    * @param n the number of times the message is to be
    *          printed
    */
   void nPrintln(String message, int n) {
        for (int i=0; i<n;  i++) {
                System.out.println(message);
        }
   }

   public static void main(String[] args) {
        nPrintln("Hello World!", 10);
   }
}
```

Your turn

- Sum from 1 to 10
  - E.g., 1 + 2 + 3 + … + 10

1. Write main() method
2. Use for or while
3. Show an equation and its result
   - 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55

Class Declaration

```
class Circle {
  public double radius = 1.0;

   /**
    * This method calculates the area of the
   circle.
    * @return the area of the circle
    */
  public double findArea() {
    return radius*radius*3.14159;
  }
}
```

Data attribute

Method

Declaring Object Variables

- Syntax:

```
ClassName objectName;
```

- Example:

```
Circle myCircle;
```

Creating Objects

- Syntax:

```
objectName = new ClassName();
```

- Example:

```
myCircle = new Circle();
```

Declaring/Creating Objects in a Single Step

- Syntax:

```
ClassName objectName = new ClassName();
```

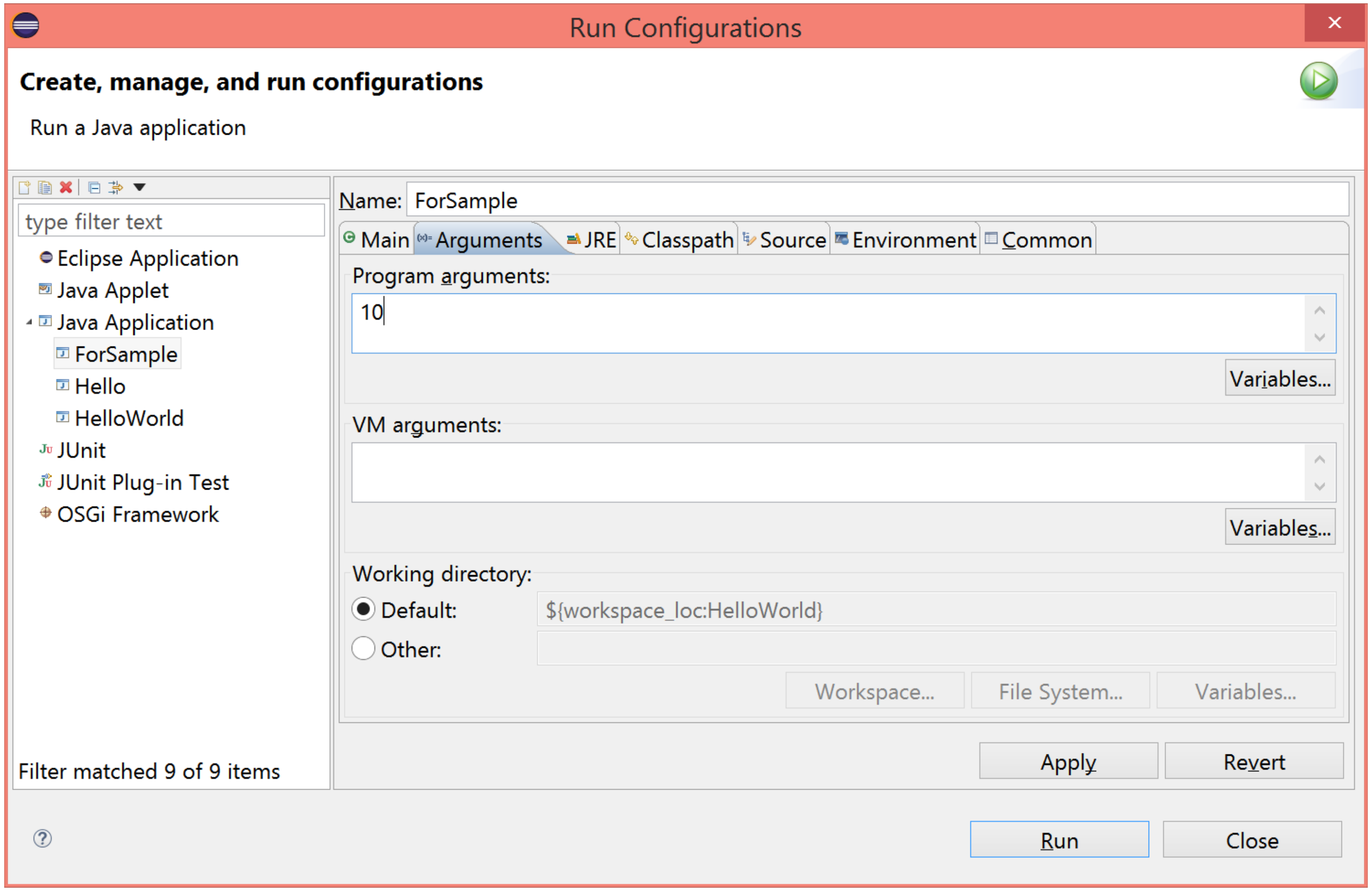
- Example:

```
Circle myCircle = new Circle();
```

Accessing Objects

- Referencing the object’s data:
  - Syntax:
    `objectName.data`
  - Example:
    `myCircle.radius`
- Referencing the object’s method (sending a message):
  - Syntax:
    `objectName.method(…)`
  - Example:
    `myCircle.findArea()`

Passing arguments to main()


Run Configurations

Create, manage, and run configurations

Run a Java application

type filter text

- Eclipse Application
- Java Applet
- Java Application
  - ForSample
  - Hello
  - HelloWorld
- JUnit
- JUnit Plug-in Test
- OSGi Framework

Filter matched 9 of 9 items

Name: ForSample

Main | Arguments | JRE | Classpath | Source | Environment | Common

Program arguments:

```
10
```

VM arguments:

Working directory:

Default: ${workspace_loc:HelloWorld}

Other:

4/29/2021

Type Conversion

- You can write your own function, but…

- String to Integer, Float, Double, etc.
  - Integer.parseInt(string)
  - Double.parseDouble(string)
- Integer, Float, Double to String
  - String.valueOf(number)
  - "" + number
  - Integer.toString(number)

Your turn

- Let’s extend the first program
- Sum from 1 to n
  - E.g., 1 + 2 + 3 + … + n

1. Define int sum(int n) method
2. main method takes the parameter *n* and passes it to the sum() method

Constructors

- Constructors are special methods that are called upon creation of an object.
  - Their purpose is to initialise objects as they are created.

- Syntax:

```
ClassName(…) {
    // Initialisation code goes here!
}
```

- Example:

```
Circle(double r) {
    radius = r;
}
```

Constructors

- Example (cont):

```
Circle() {
  radius = 1.0;
}
```

- We can specify which constructor to use when we create a new object.

- Example:

```
myCircle = new Circle(5.0);
```

Modifiers

By default, the class, variable, or data can be accessed by any class in the same package.

- `public`

  The class, data, or method is visible to any class in any package.

- `private`

  The data or methods can be accessed only by the declaring class.

Your turn

- Let’s extend the second program
  - Sum from 1 to n
    - E.g., 1 + 2 + 3 + … + n

1. Define Sum class and sum(int n) method
   - Sum is a public class
   - sum() is a public and static method

Packages

- **Packages** are used to organise Java classes.
  - Packages are like domain names – they are hierarchical (e.g. packages can have sub-packages, which can have sub-packages, and so on…).
  - There are five basic Java packages:
    java.lang: Classes that are part of the basic language (e.g. Strings)
    java.io: Classes that do input and output
    java.net: Classes for networking
    java.awt: Classes for creating Graphical Interfaces
    java.util: General Utility Classes
  - So, if I want a class that does file input, I look in the input/output (io) package!

Referencing Classes in Different Packages

- If we want to use a class that is in a package (with the exception of those in the java.lang package), we need to declare our intention at the top of the program.

- We do this with an **import** statement
  - For example:

```
import java.io.PrintStream;
```

  - If we want to use multiple classes from a single package we can use the following statement:

```
import java.io.*;
```

This means “any class”

Some Useful Java Classes

- One of the most useful sets of Java classes are the input/output (io) classes.
  - These classes are located in the java.io package and are part of the default Java API.
  - These classes allow programs to perform input/output in a variety of ways.
- There are many different types of io class.
  - You are not expected to know them all intimately, but you should know how to do a few basic things with them (e.g. User input, File input/output, etc.)

System.out

- You have already used an io class implicitly when you write information to the screen.

- System.out refers to a data attribute of the System class.
  - This attribute is an instance of the class **PrintStream**.
  - This class is used for output only.

- The PrintStream class includes a method, **println**, that prints a string to the specified output.
  - We don’t know where this output comes from!
  - We don’t need to know – we just need to know where it goes to (i.e. the screen).

Handling User Input

- To get input from the user, we use another data attribute of the System class:

  `System.in`

- This attribute is an instance of the **InputStream** class.
  - The InputStream class is not easy to use (it works only with bytes).
  - To make things easier, we can wrap the input stream in other io classes.

- In particular, we use two classes:
  - **InputStreamReader**. This class converts the byte stream into a character stream (this is much more useful – honestly!).
  - **BufferedReader**. This class buffers data from the enwrapped stream, allowing smoother reading of the stream (it doesn't wait for the program to ask for the next character in the stream).

Handling User Input

- To wrap an io object, we create a new io object, passing the old io object into the constructor.
  - For example, to wrap System.in within an InputStreamReader, we write:

```
new InputStreamReader(System.in)
```

- To use the BufferedReader (which, remember, is an optimisation), we wrap the InputStreamReader object we created above.
  - For example:

```
new BufferedReader(new InputStreamReader(System.in))
```

- Once we have a BufferedReader, we can use a nice method it provides called readLine() which reads a line of text!
  - This makes life much easier!

Handling User Input

- Now that we have a way of reading a line of text from the System input stream (i.e. the keyboard), how do we use it?

```
public String readString() {
    BufferedReader in = new BufferedReader(
                                   new InputStreamReader(System.in));
    String line = null;
    try {
        line = in.readLine();
        while (line == null) {
            line = in.readLine();
        }
    } catch (IOException ie) {
        System.out.println("The following problem occurred " +
                           "when reading input: " + ie);
    }
    return line;
}
```

Handling User Input

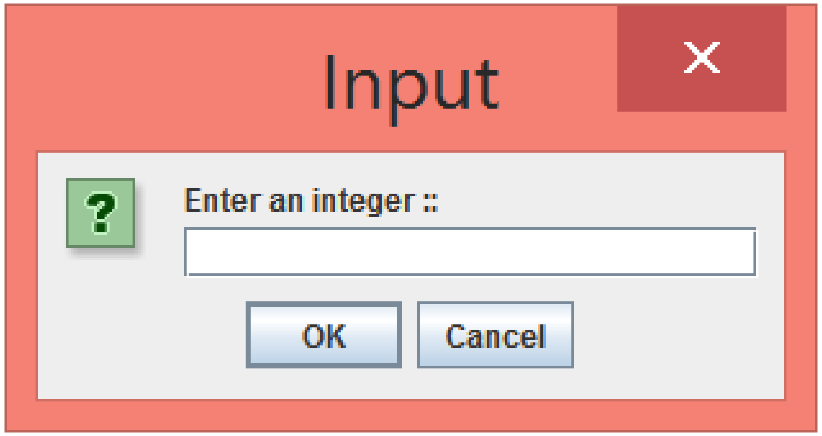
- Scanner
  - Read a line of input from its source
  - Example:

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
System.out.println("You entered" + i);
```

- This example reads a single `int` from `System.in` and outputs it to `System.out`. It does not check that the user actually entered an `int`.

Handling User Input

- Java Swing (GUI)
  - JOptionPane.*showInputDialog()*



Your turn

- Let’s extend the third program
  - Sum *n* integers

1. Define sum(int[])
2. Using one of user input mechanisms, take *n* integers from the console
   1. First, you will give the number of integers
   2. Then, you will give n integers
   3. They are stored into an integer array then passed to sum(int[])

I/O and File objects

- The `File` class in the `java.io` package represents files.
  - `import java.io.*;`
  - Create a `File` object to get information about a file on the disk. (Creating a `File` object doesn't create a new file on your disk.)

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000) {
    f.delete();
}
```

Method name	**Description**
`canRead()`	returns whether file is able to be read
`delete()`	removes file from disk
`exists()`	whether this file exists on disk
`getName()`	returns file's name
`length()`	returns number of characters in file
`renameTo(`*file*`)`	changes name of file

Scanners to read files

- To read a file, create a `File` object and pass it as a parameter when constructing a `Scanner`.

- Creating a `Scanner` for a file, general syntax:

  `Scanner` ***<name>*** `= new Scanner(new File("`***<file name>***`"));`

  Example:

  ```
  File f = new File("numbers.txt");
  Scanner input = new Scanner(f);
  ```

  or:

  ```
  Scanner input = new Scanner(new File("numbers.txt"));
  ```

File and path names

- **relative path**: does not specify any top-level folder
  - `"names.dat"`
  - `"input/kinglear.txt"`
- **absolute path**: specifies drive letter or top `"/"` folder
  - `"C:/Documents/smith/hw6/input/data.csv"`
  - Windows systems also use backslashes to separate folders. How would the above filename be written using backslashes?
- When you construct a `File` object with a relative path, Java assumes it is relative to the *current directory*.
  - `Scanner input = new Scanner(new File("data/readme.txt"));`
  - If our program is in: `H:/johnson/hw6`,
    Java will look for: `H:/johnson/hw6/data/readme.txt`.

Compiler error with files

- The following program does not compile:

```java
import java.io.*;      // for File
import java.util.*;    // for Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The following compiler error is produced:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or declared
to be thrown
        Scanner input = new Scanner(new File("data.txt"));
                        ^
```

Files and input cursor

- Consider a file `numbers.txt` that contains this text:

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

- A `Scanner` views all input as a stream of characters, which it processes with its *input cursor*:

  -
    ```
    308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
    ^
    ```
  - When you call the methods of the `Scanner` such as `next` or `nextDouble`, the `Scanner` breaks apart the input into *tokens*.

Input tokens

- **token**: A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).

- Example: If an input file contains the following:

```
23   3.14
  "John Smith"
```

  – The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
1. `23`	`int`, `double`, `String`
2. `3.14`	`double`, `String`
3. `"John`	`String`
4. `Smith"`	`String`

Consuming tokens

- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace.
  - We call this *consuming* input.

```
  input.nextDouble()
– 308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
       ^


  input.nextDouble()
– 308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n
                ^
```

Your turn

- Consider an input file named `numbers.txt` that contains the following text:

```
308.2   14.9 7.4  2.8 3.9 4.7 -15.4 2.8
```

- Write a program that reads the first 5 values from this file and prints them along with their sum. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
Sum = 337.19999999999993
```

File input answer

```
// Displays the first 5 numbers in the given file,
// and displays their sum at the end.

import java.io.*;    // for File, FileNotFoundException
import java.util.*;

public class Echo {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Test before read question

- Rewrite the previous program so that it reads the entire file. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```

Test before read answer

```
// Displays each number in the given file,
// and displays their sum at the end.

import java.io.*;
import java.util.*;

public class Echo2 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

File processing question

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
  - The program should skip any such tokens.

- For example, the program should produce the same output as before when given this input file:

```
308.2  hello
   14.9 7.4  bad stuff 2.8

3.9 4.7  oops  -15.4
:-)    2.8  @#*($&
```

File processing answer

```
// Displays each number in the given file,
// and displays their sum at the end.

import java.io.*;
import java.util.*;

public class Echo3 {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println("number = " + next);
                sum += next;
            } else {
                input.next();   // consume / throw away bad token
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

File processing question

- Write a program that accepts an input file containing integers representing daily high temperatures.

  Example input file:

  ```
  42 45 37 49 38 50 46 48 48 30 45 42 45 40 48
  ```

- Your program should print the difference between each adjacent pair of temperatures, such as the following:

  ```
  Temperature changed by 3 deg F
  Temperature changed by -8 deg F
  Temperature changed by 12 deg F
  Temperature changed by -11 deg F
  Temperature changed by 12 deg F
  Temperature changed by -4 deg F
  Temperature changed by 2 deg F
  Temperature changed by 0 deg F
  Temperature changed by -18 deg F
  Temperature changed by 15 deg F
  Temperature changed by -3 deg F
  Temperature changed by 3 deg F
  Temperature changed by -5 deg F
  Temperature changed by 8 deg F
  ```

File processing answer

```
import java.io.*;
import java.util.*;

public class Temperatures {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.dat"));
        int temp1 = input.nextInt();

        while (input.hasNextInt()) {
            int temp2 = input.nextInt();
            System.out.println("Temperature changed by " +
                               (temp2 - temp1) + " deg F");
            temp1 = temp2;
        }
    }
}
```

Conditionals and Loops

- Now we will examine programming statements that allow us to:
  - make decisions
  - repeat processing steps in a loop

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- The Java conditional statements are the:
  - *if statement*
  - *if-else statement*
  - *switch statement*

The if Statement

- The *if statement* has the following syntax:

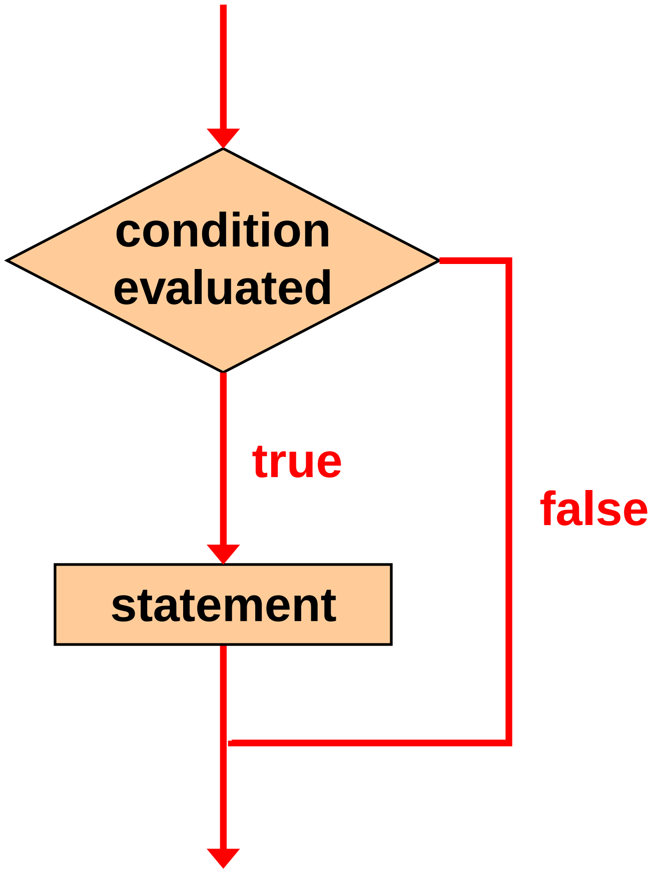
`if` is a Java reserved word

The *`condition`* must be a boolean expression. It must evaluate to either true or false.

```
if ( condition )
   statement;
```

If the *`condition`* is true, the *`statement`* is executed.
If it is false, the *`statement`* is skipped.

Logic of an if statement



Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

- Note the difference between the equality operator (==) and the assignment operator (=)

The if Statement

- An example of an `if` statement:

```
if (sum > MAX)
   delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

- **First the condition is evaluated -- the value of `sum` is either greater than the value of `MAX`, or it is not**
- **If the condition is true, the assignment statement i s executed -- if it isn't, it is skipped.**
- **Either way, the call to `println` is executed next**

The if Statement

- What do the following statements do?

```
if (top >= MAXIMUM)
   top = 0;
```

Sets `top` to zero if the current value of `top` is greater than or equal to the value of `MAXIMUM`

```
if (total != stock + warehouse)
   inventoryError = true;
```

Sets a flag to true if the value of `total` is not equal to the sum of `stock` and `warehouse`

- **The precedence of the arithmetic operators is higher than the precedence of the equality and relational operators**

Logical Operators

- Boolean expressions can also use the following *logical operators*:

`!`	Logical NOT
`&&`	Logical AND
`||`	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition `a` is true, then `!a` is false; if `a` is false, then `!a` is true
- Logical expressions can be shown using a *truth table*

`a`	`!a`
true	**false**
false	**true**

Logical AND and Logical OR

- The *logical AND* expression

  ```
  a && b
  ```

  is true if both `a` and `b` are true, and false otherwise

- The *logical OR* expression

  ```
  a || b
  ```

  is true if `a` or `b` or both are true, and false otherwise

Logical Operators

- Expressions that use logical operators can form complex conditions

```
      if (total < MAX+5 && !found)
         System.out.println ("Processing…");
```

- **All logical operators have lower precedence than t he relational operators**
- **Logical NOT has higher precedence than logical A ND and logical OR**

Logical Operators

- A truth table shows all possible true-false combinations of the terms
- Since `&&` and `||` each have two operands, there are four possible combinations of conditions `a` and `b`

`a`	`b`	`a && b`	`a || b`
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Boolean Expressions

- Specific expressions can be evaluated using truth tables

`total < MAX`	`found`	`!found`	`total < MAX && !found`
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

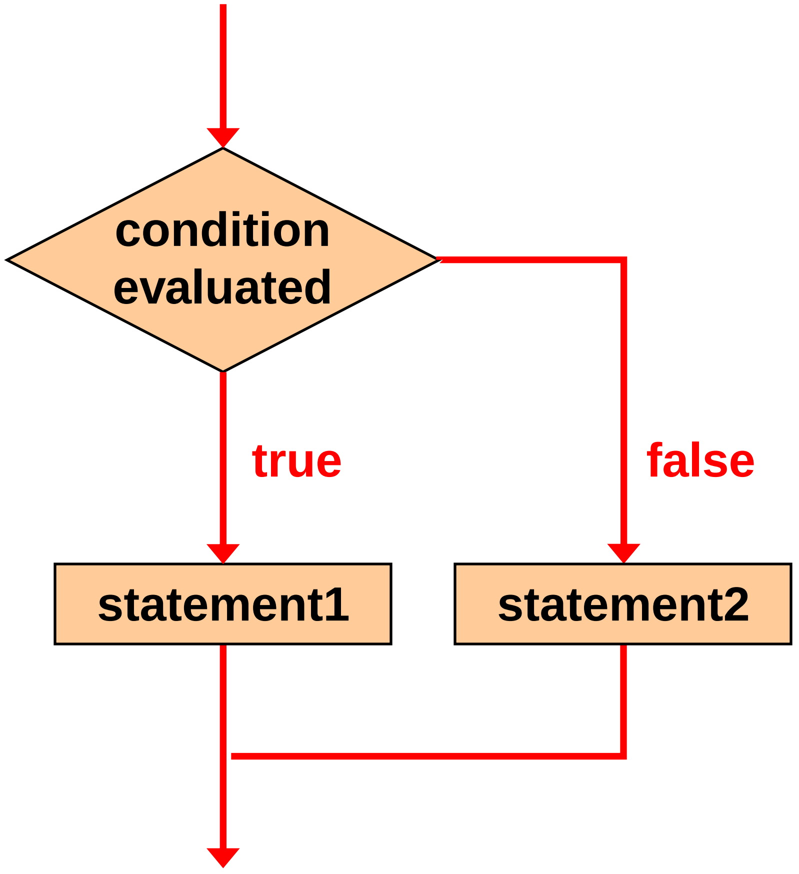
The if-else Statement

- An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if ( condition )
   statement1;
else
   statement2;
```

- **If the *`condition`* is true, *`statement1`* is executed; if the condition is false, *`statement2`* is executed**
- **One or the other will be executed, but not both**

Logic of an if-else statement



Fix Bugs!

- Syntax errors
- Logical error

The switch Statement

- The *switch statement* provides another way to decide which statement to execute next
- The `switch` statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- Each case contains a value and a list of statements
- The flow of control transfers to statement associated with the first case value that matches

The switch Statement

- The general syntax of a `switch` statement is:

```
switch ( expression )
{
   case value1 :
      statement-list1
   case value2 :
      statement-list2
   case value3 :
      statement-list3
   case  ...

}
```

`switch` **and** **`case`** **are reserved words**

If *`expression`* matches *`value2`*, control jumps to here

The switch Statement

- Often a *break statement* is used as the last statement in each case's statement list
- A `break` statement causes control to transfer to the end of the `switch` statement
- If a `break` statement is not used, the flow of control will continue into the next case
- Sometimes this may be appropriate, but often we want to execute only the statements associated with one case

The switch Statement

- An example of a switch statement:

```
switch (option)
{
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   case 'C':
      cCount++;
      break;
}
```

The switch Statement

- A `switch` statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- If there is no default case, and no other value matches, control falls through to the statement after the switch

The switch Statement

- The expression of a `switch` statement must result in an *integral type*, meaning an `int` or a `char`
- It cannot be a `boolean` value, a floating point value (`float` or `double`), or another integer type
- The implicit boolean condition in a `switch` statement is equality
- You cannot perform relational checks with a `switch` statement

Your turn

- Grade Report
  - Reads a score from the user and prints a grade
  - E.g., 89 = B+, 91 = A-, …

Example: Grade Report

```
public class GradeReport
{
   //-----------------------------------------------------------------
-   //  Reads a grade from the user and prints comments
   //  accordingly.
   //-----------------------------------------------------------------
-
   public static void main (String[] args)
   {
      int grade, category;

      Scanner scan = new Scanner (System.in);

      System.out.print ("Enter a numeric grade (0 to 100): ");
      grade = scan.nextInt();

      category = grade / 10;

      System.out.print ("That grade is ");
```

```
        switch (category)
        {
           case 10:
              System.out.println ("a perfect score. Well done.");
              break;
           case 9:
              System.out.println ("well above average. Great.");
              break;
           case 8:
              System.out.println ("above average. Nice job.");
              break;
           case 7:
              System.out.println ("average.");
              break;
           case 6:
              System.out.println ("below average.");
              System.out.println ("See the instructor.");
              break;
           default:
              System.out.println ("not passing.");
        }
     }
  }
```

Your turn

- Grade Report
  - Read students' scores from a file
  - And show their statistics
    - Total student number
    - Average, variance, standard deviation, etc
    - Write the results into a file

Graphical User Interface (GUI) Components

- View inputs and outputs simultaneously.
- One graphical window.
- Input values in any order.
- Change input values in window.
- Click buttons to get output.

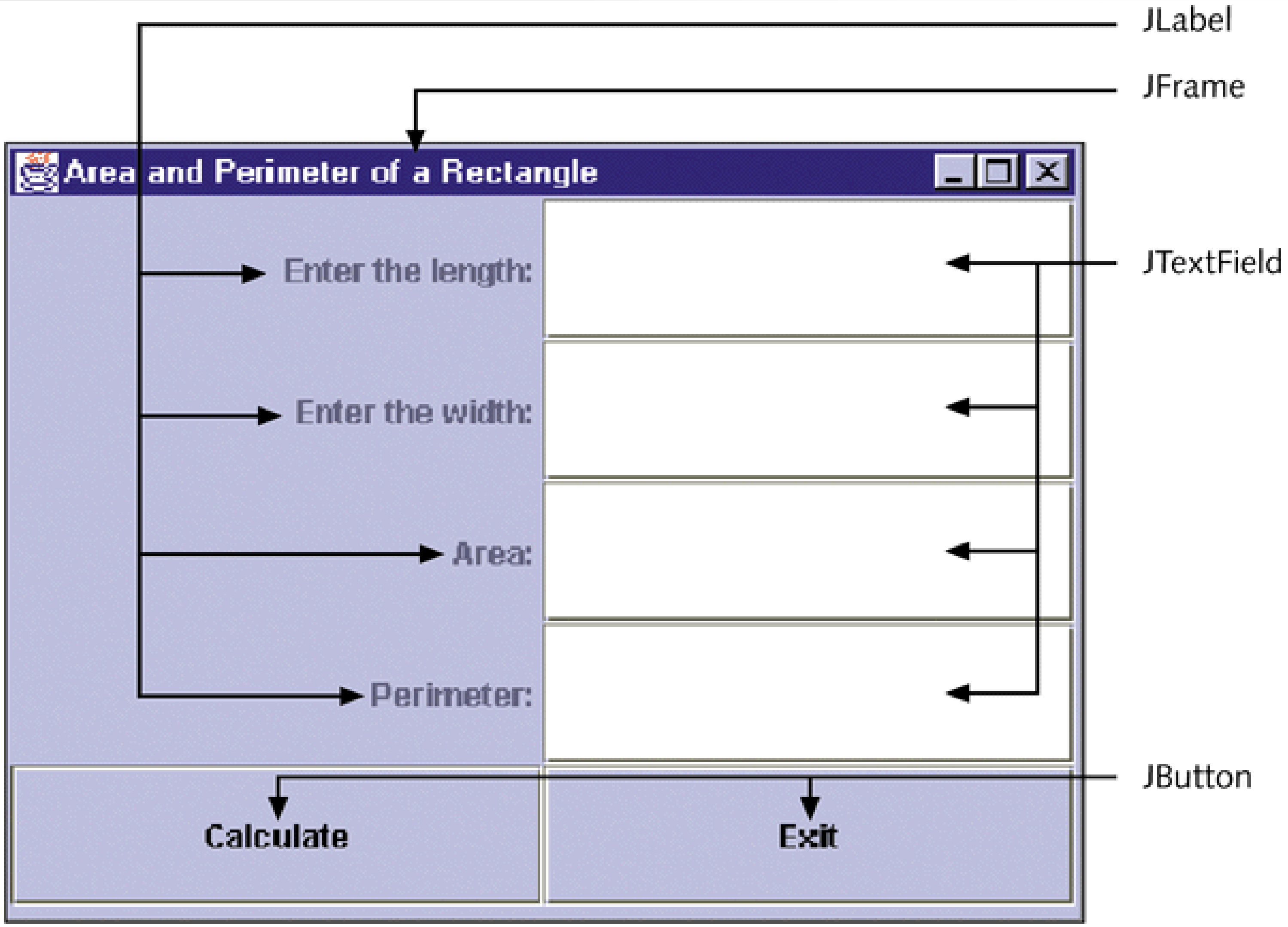
Swing vs AWT

- AWT is Java's original set of classes for building GUIs
  - Uses peer components of the OS; heavyweight
  - Not truly portable: looks different and lays out inconsistently on different OSs
    - Due to OS's underlying display management system
- Swing is designed to solve AWT's problems
  - 99% java; lightweight components
    - Drawing of components is done in java
    - Uses AWTs components
  - Lays out consistently on all OSs
  - Uses AWT event handling

Implementing a Swing GUI

- Import javax.swing.*, java.io.*, java.awt.*
- Make a specific class to do GUI functions
- Specify all the GUI functions/components in the class’s constructor (or methods / classes called by the constructor)
- Run the GUI by instantiating the class in the class’s main method

Java GUI Components



Graphical User Interface (GUI) Components

- GUI components placed in content pane.
- GUI components:
  - Windows
  - Labels
  - Text areas
  - Buttons

GUI Components

- Added to content pane of window.
- Not added to window itself.
- Pixel: A picture element.

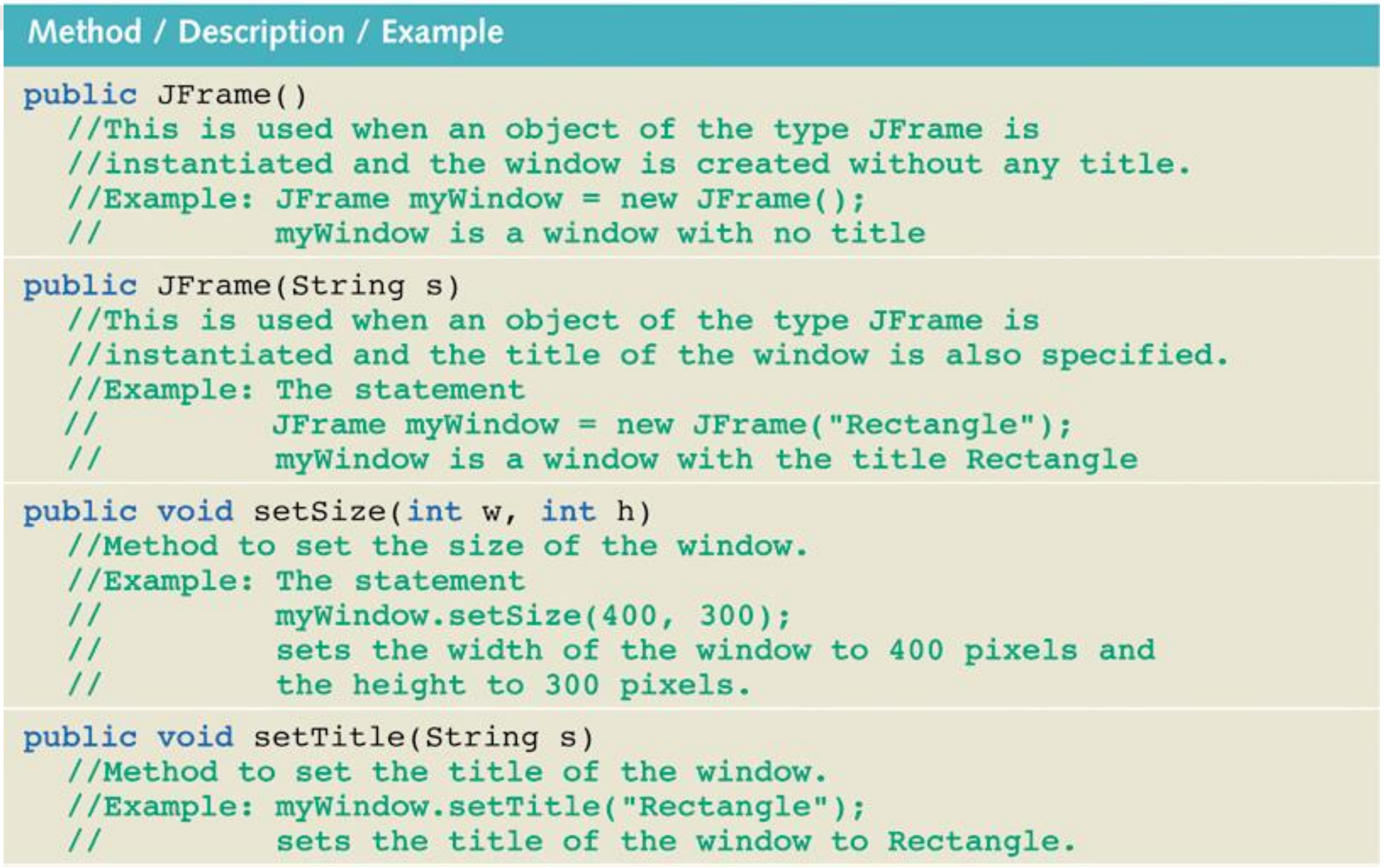
Windows

- Can be created using a Frame object.
- The class Frame provides various methods to control attributes of a window.
- Measured in pixels of height and width.
- Attributes associated with windows:
  - Title
  - Width
  - Height

`class JFrame`

- GUI window instance created as instance of Frame.
- Provides various methods to control window attributes.

Methods Provided by the `class JFrame`



Method / Description / Example

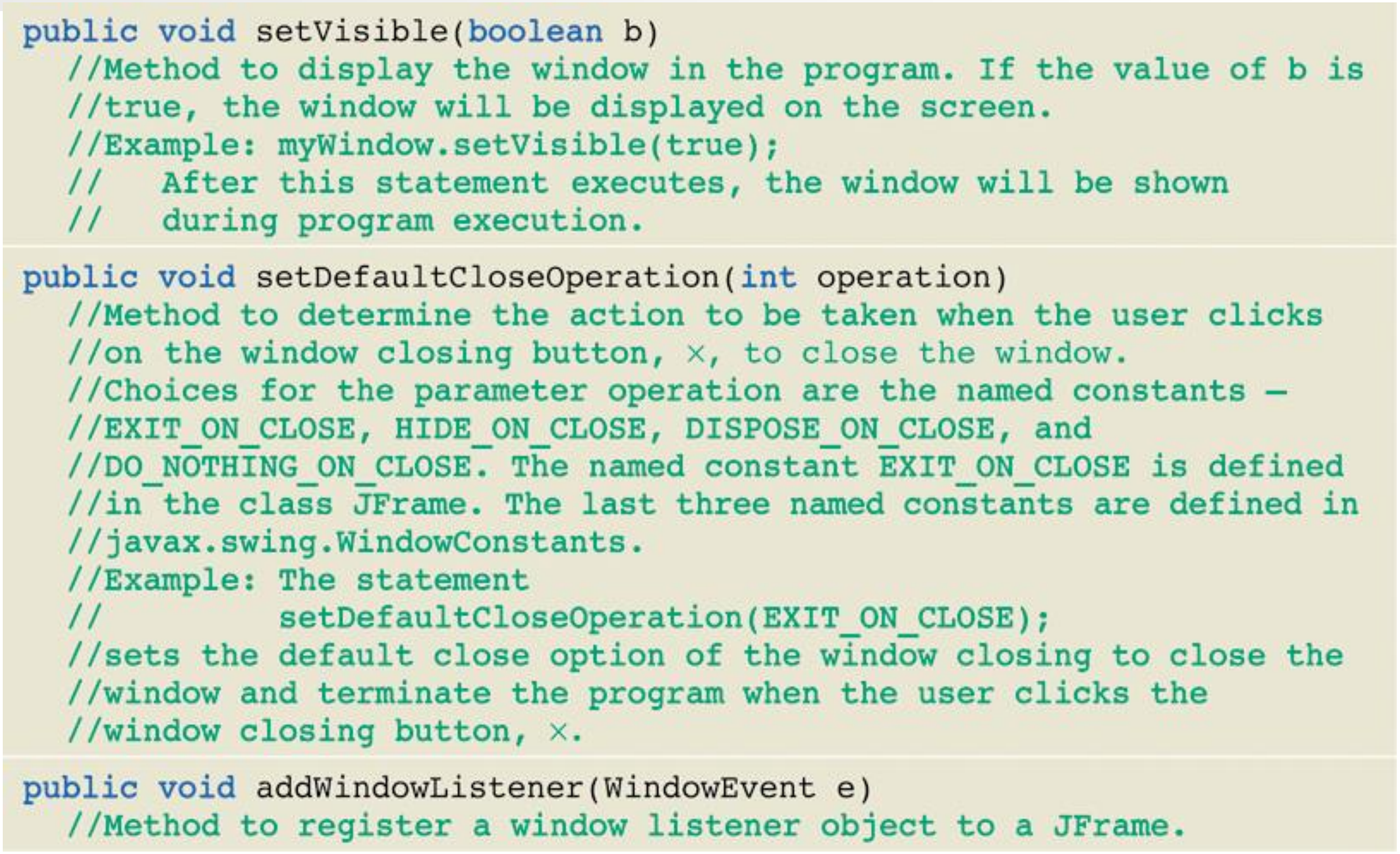
```
public JFrame()
   //This is used when an object of the type JFrame is
   //instantiated and the window is created without any title.
   //Example: JFrame myWindow = new JFrame();
   //          myWindow is a window with no title
```

```
public JFrame(String s)
   //This is used when an object of the type JFrame is
   //instantiated and the title of the window is also specified.
   //Example: The statement
   //          JFrame myWindow = new JFrame("Rectangle");
   //          myWindow is a window with the title Rectangle
```

```
public void setSize(int w, int h)
   //Method to set the size of the window.
   //Example: The statement
   //          myWindow.setSize(400, 300);
   //          sets the width of the window to 400 pixels and
   //          the height to 300 pixels.
```

```
public void setTitle(String s)
   //Method to set the title of the window.
   //Example: myWindow.setTitle("Rectangle");
   //          sets the title of the window to Rectangle.
```

Methods Provided by the `class JFrame`



```
public void setVisible(boolean b)
   //Method to display the window in the program. If the value of b is
   //true, the window will be displayed on the screen.
   //Example: myWindow.setVisible(true);
   //    After this statement executes, the window will be shown
   //    during program execution.
```

```
public void setDefaultCloseOperation(int operation)
   //Method to determine the action to be taken when the user clicks
   //on the window closing button, ×, to close the window.
   //Choices for the parameter operation are the named constants —
   //EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE, and
   //DO_NOTHING_ON_CLOSE. The named constant EXIT_ON_CLOSE is defined
   //in the class JFrame. The last three named constants are defined in
   //javax.swing.WindowConstants.
   //Example: The statement
   //          setDefaultCloseOperation(EXIT_ON_CLOSE);
   //sets the default close option of the window closing to close the
   //window and terminate the program when the user clicks the
   //window closing button, ×.
```

```
public void addWindowListener(WindowEvent e)
   //Method to register a window listener object to a JFrame.
```

Two Ways to Create a Window

- First way:
  - Declare object of type `JFrame`.
  - Instantiate object.
  - Use various methods to manipulate window.
- Second way:
  - Create class-containing application program by extending definition of `class JFrame`.
  - Utilize mechanism of inheritance.

Create a JFrame

```
frame = new JFrame();
frame.setSize(400, 300);
frame.setTitle("Area and Perimeter of a rec
tangle");
```

Content Pane

- Inner area of GUI window (below title bar, inside border).
- To access content pane:
  - Declare reference variable of type `Container`.
  - Use method `getContentPane` of `class JFrame`.

Methods Provided by the class Container

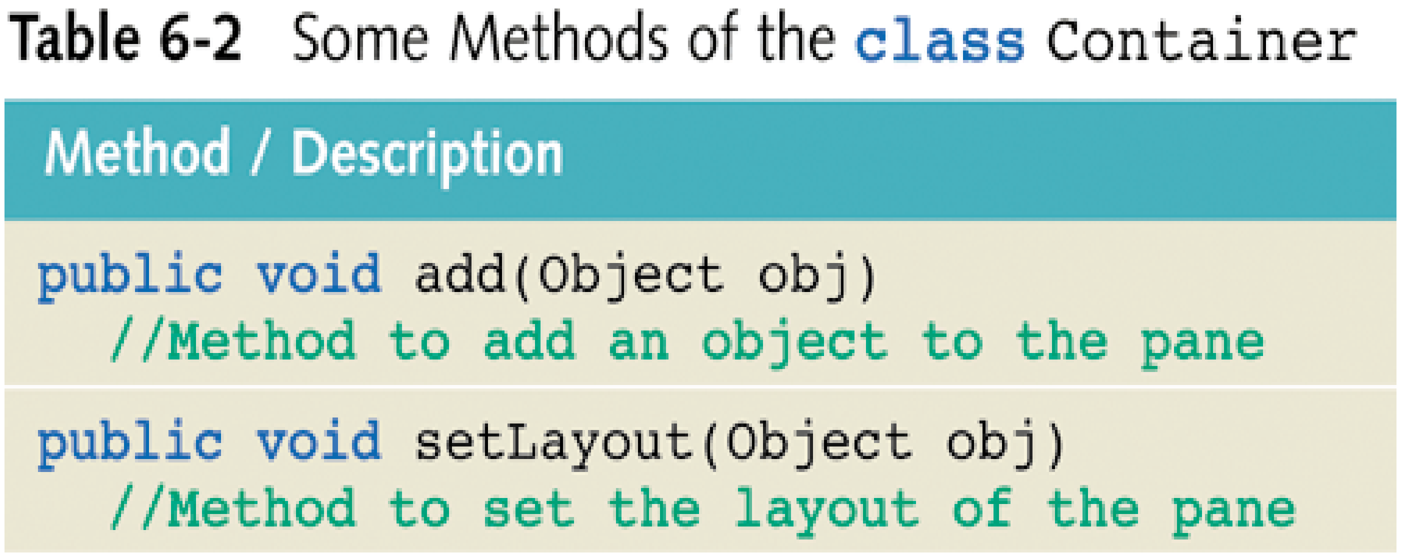


Table 6-2 Some Methods of the class Container

Method / Description

`public void add(Object obj)`
`   //Method to add an object to the pane`

`public void setLayout(Object obj)`
`   //Method to set the layout of the pane`

`class JLabel`

- Labels: Objects of a particular class type.
- `class JLabel`: Used to create labels.
- Label attributes:
  - Title
  - Width
  - Height
- To create a label:
  - Instantiate object of type `JLabel`.
  - Modify attributes to control display of labels.

Methods Provided by the class JLabel

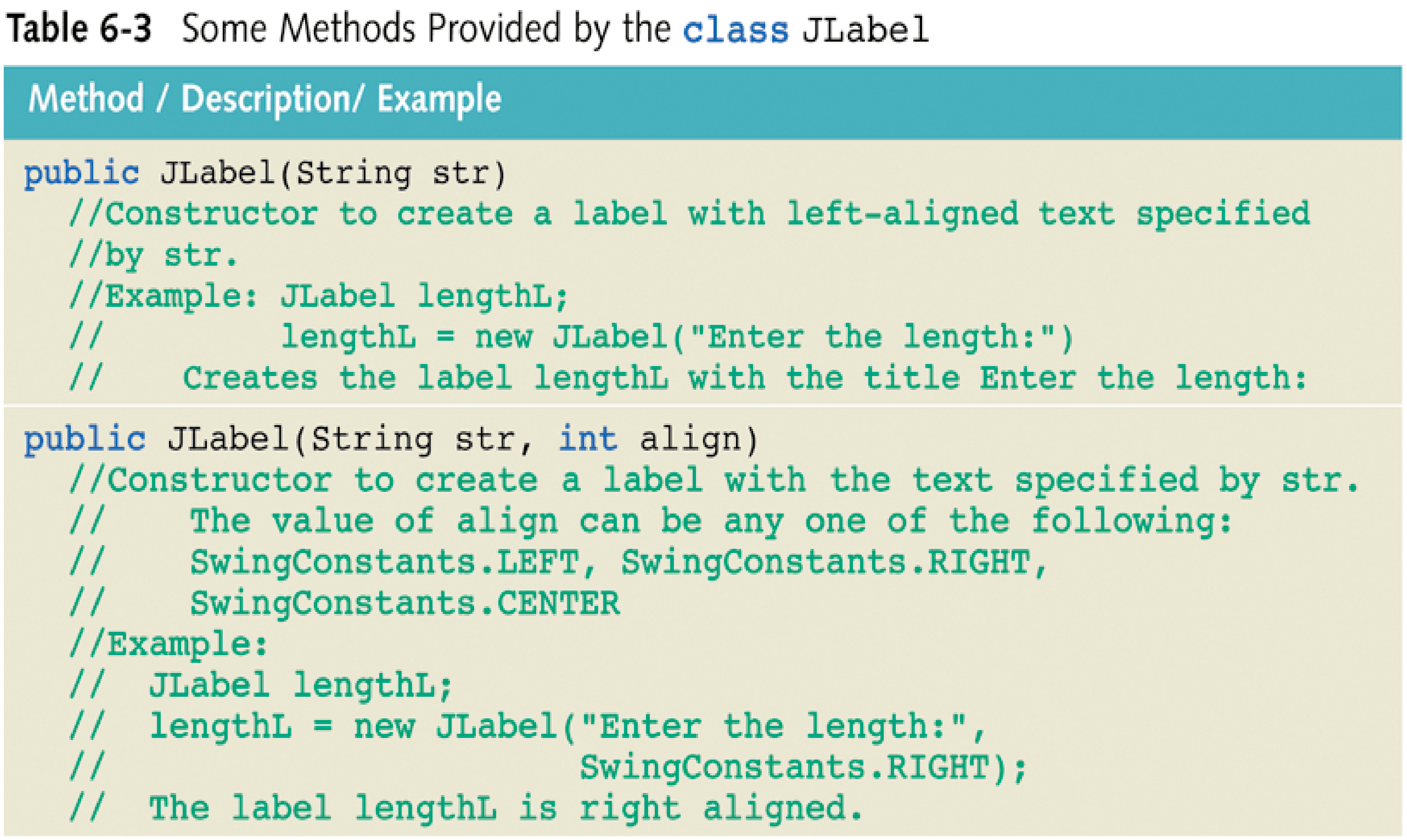


Table 6-3 Some Methods Provided by the class JLabel

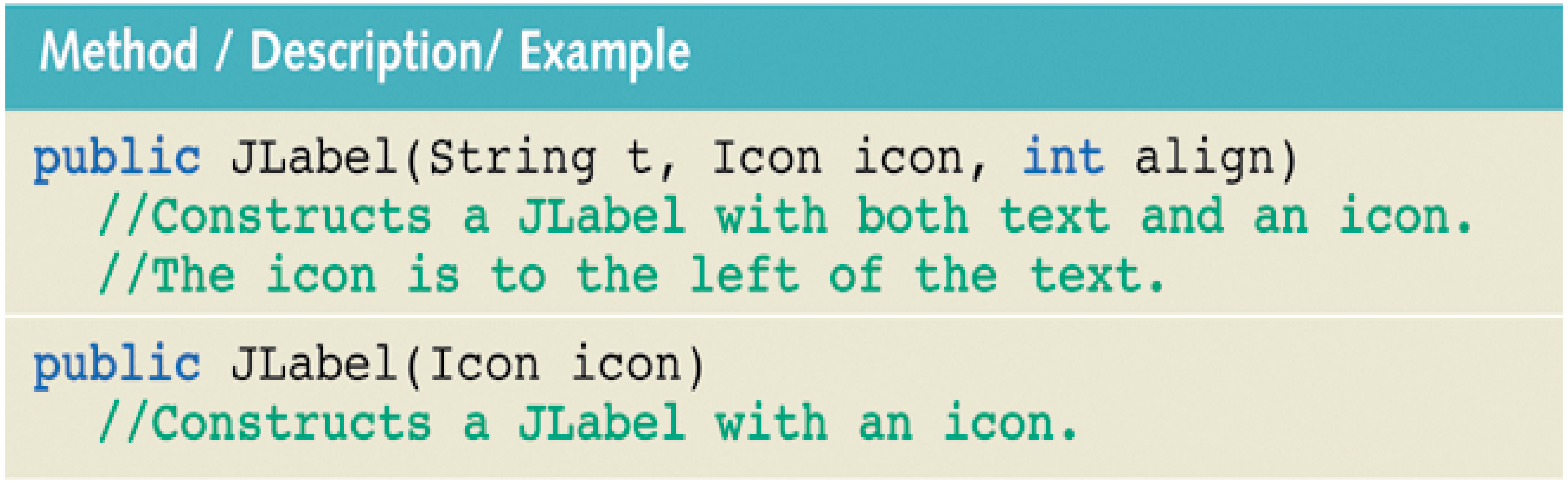
Method / Description/ Example

```
public JLabel(String str)
   //Constructor to create a label with left-aligned text specified
   //by str.
   //Example: JLabel lengthL;
   //         lengthL = new JLabel("Enter the length:")
   //    Creates the label lengthL with the title Enter the length:
```

```
public JLabel(String str, int align)
   //Constructor to create a label with the text specified by str.
   //    The value of align can be any one of the following:
   //    SwingConstants.LEFT, SwingConstants.RIGHT,
   //    SwingConstants.CENTER
   //Example:
   //  JLabel lengthL;
   //  lengthL = new JLabel("Enter the length:",
   //                       SwingConstants.RIGHT);
   //  The label lengthL is right aligned.
```

Methods Provided by the class JLabel

Table 6-3 Some Methods Provided by the class JLabel (continued)



Method / Description/ Example

`public JLabel(String t, Icon icon, int align)`
`   //Constructs a JLabel with both text and an icon.`
`   //The icon is to the left of the text.`

`public JLabel(Icon icon)`
`   //Constructs a JLabel with an icon.`

Add 4 Lables

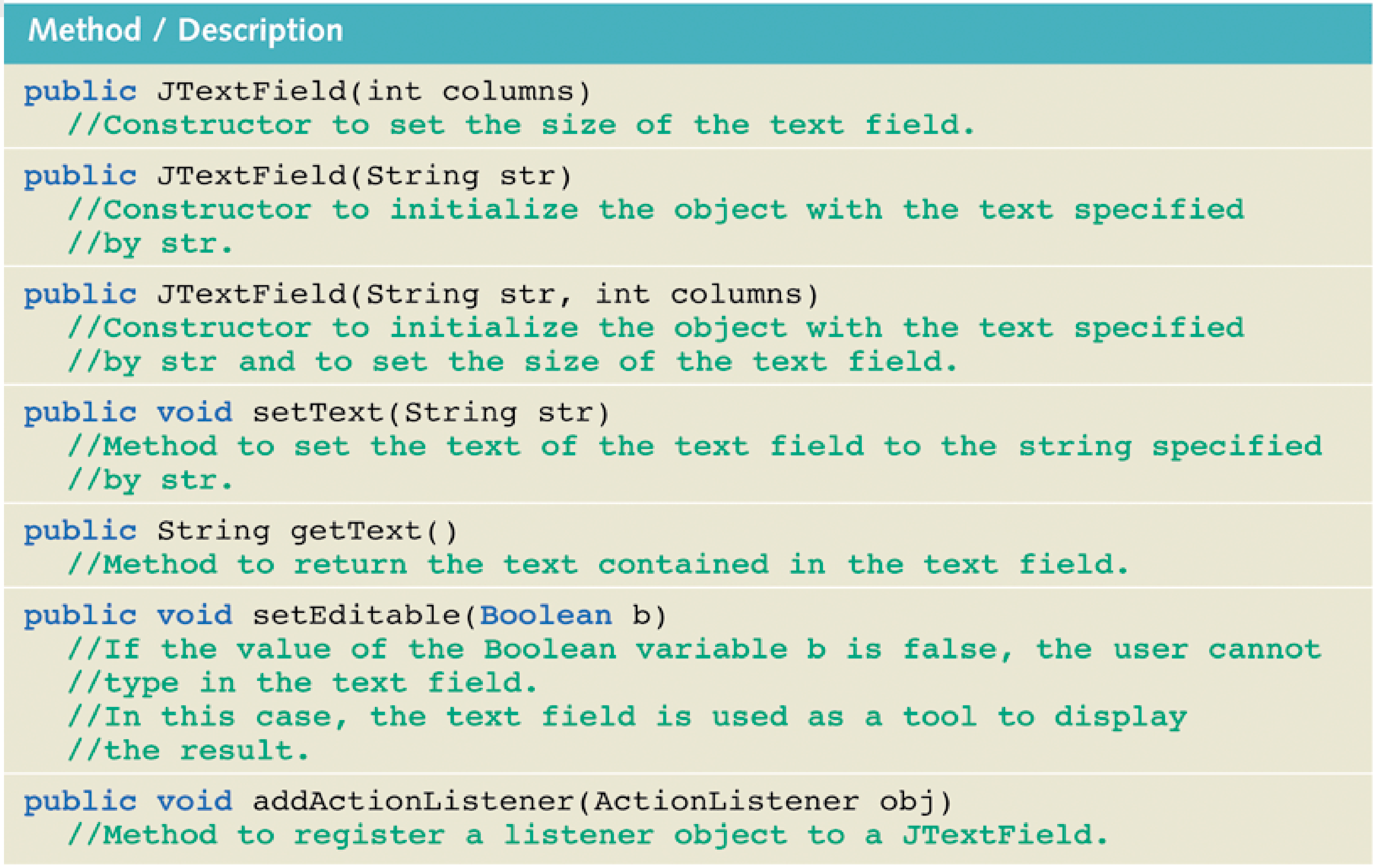
```
lengthL = new JLabel("Enter the length: ", SwingCons
nts.RIGHT);
widthL = new JLabel("Enter the width: ", SwingConsta
s.RIGHT);
areaL = new JLabel("Area: ", SwingConstants.RIGHT);
perimeterL = new JLabel("Perimeter: ", SwingConstant
RIGHT);
```

`class JTextField`

- Text fields: Objects belonging to `class JTextField`.
- To create a text field:
  - Declare reference variable of type `JTextField`.
  - Instantiate object.

Methods Provided by the `class JTextField`

Table 6-4 Some Methods of the `class JTextField`



Method / Description

`public JTextField(int columns)`
`   //Constructor to set the size of the text field.`

`public JTextField(String str)`
`   //Constructor to initialize the object with the text specified`
`   //by str.`

`public JTextField(String str, int columns)`
`   //Constructor to initialize the object with the text specified`
`   //by str and to set the size of the text field.`

`public void setText(String str)`
`   //Method to set the text of the text field to the string specified`
`   //by str.`

`public String getText()`
`   //Method to return the text contained in the text field.`

`public void setEditable(Boolean b)`
`   //If the value of the Boolean variable b is false, the user cannot`
`   //type in the text field.`
`   //In this case, the text field is used as a tool to display`
`   //the result.`

`public void addActionListener(ActionListener obj)`
`   //Method to register a listener object to a JTextField.`

Add 4 Text Fields

```
lengthTF = new JTextField(
10);
widthTF = new JTextField(1
0);
areaTF = new JTextField(10
);
perimeterTF = new JTextFie
ld(10);
```

`class JButton`

- Provided to create buttons in Java.
- To create a button:
  - Use the same technique that is used to create `JLabel` and `JTextField`.

Methods Provided by the `class JButton`

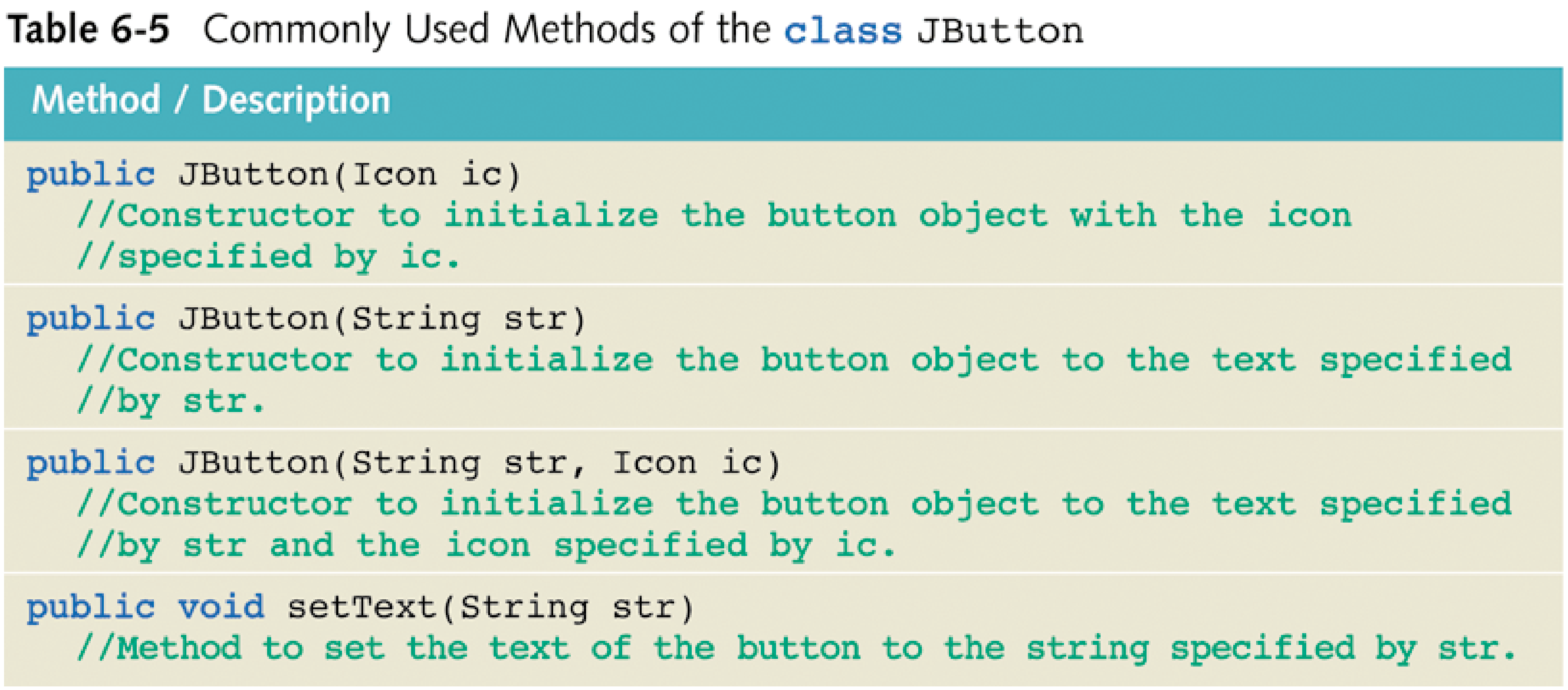
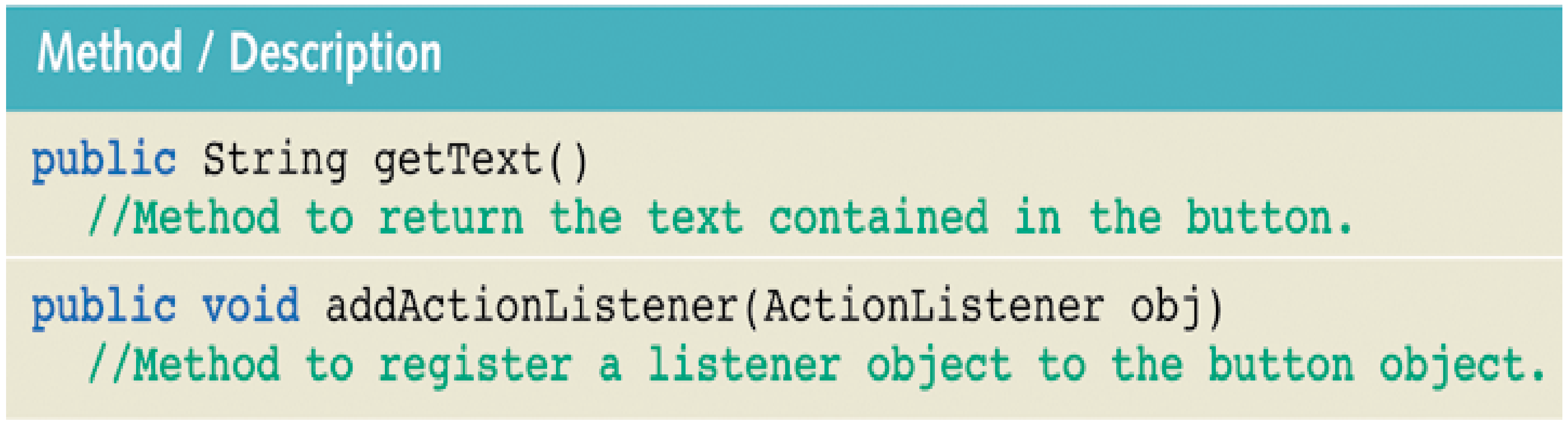


Table 6-5 Commonly Used Methods of the `class JButton`

Method / Description
`public JButton(Icon ic)` `//Constructor to initialize the button object with the icon` `//specified by ic.`
`public JButton(String str)` `//Constructor to initialize the button object to the text specified` `//by str.`
`public JButton(String str, Icon ic)` `//Constructor to initialize the button object to the text specified` `//by str and the icon specified by ic.`
`public void setText(String str)` `//Method to set the text of the button to the string specified by str.`

Methods Provided by the `class JButton`

Table 6-5 Commonly Used Methods of the `class JButton` (continued)



Method / Description

`public String getText()`
`   //Method to return the text contained in the button.`

`public void addActionListener(ActionListener obj)`
`   //Method to register a listener object to the button object.`

Add 2 Buttons

```
JButton calculateB = new JButton("Calcula
te");
JButton exitB = new JButton("Exit");
```

Handling an Event

- Action event: An event created when `JButton` is clicked.
- Event listener: An object that receives a message when `JButton` is clicked.
- In Java, you must register the listener.

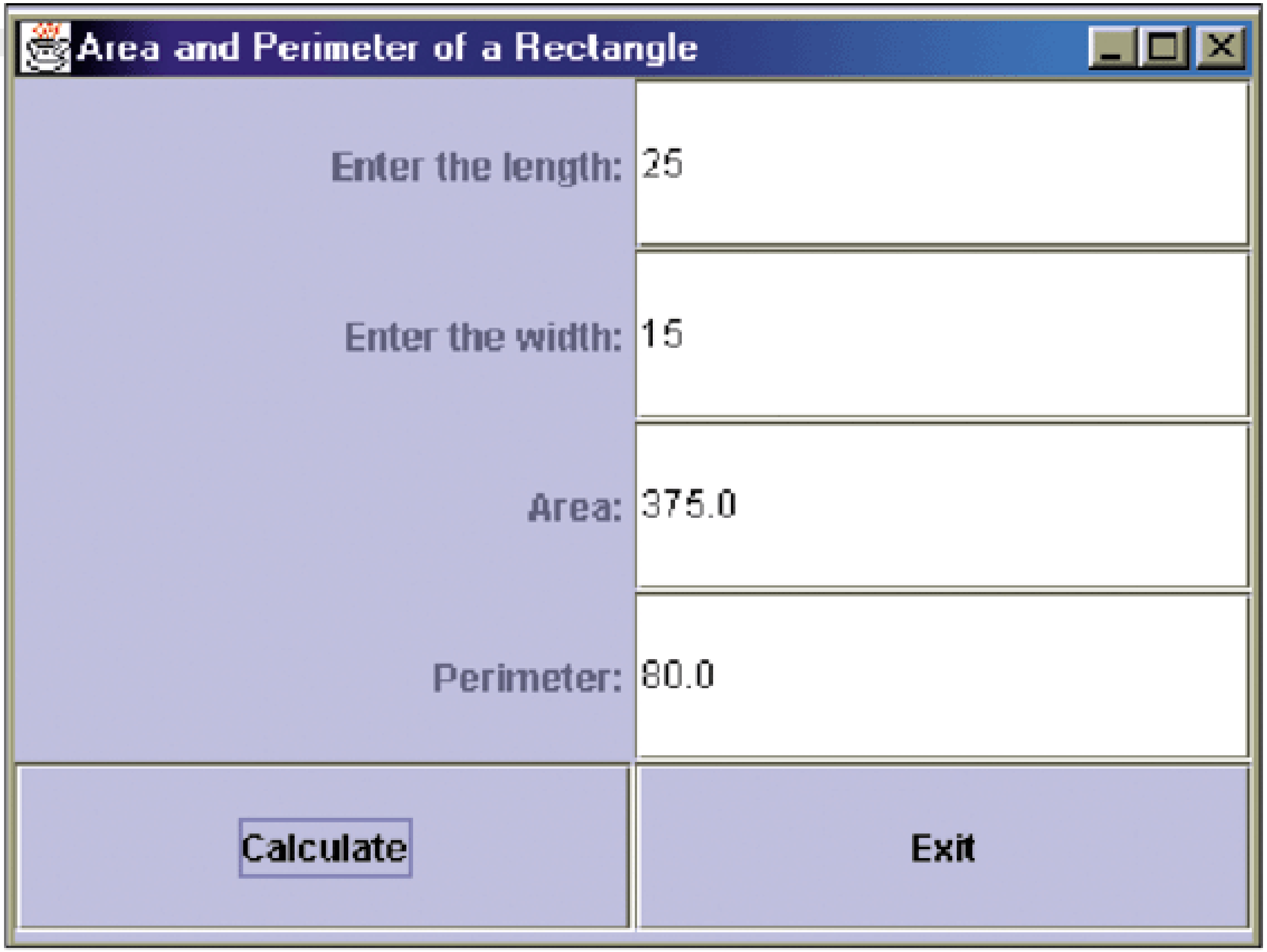
Handling an Event

- `class ActionListener`
  - Handles action event.
  - Part of package `java.awt.Event`.
  - The `class ActionListener` is a special type of class (interface).
  - Must contain the `actionPerformed` method.

Add an Event Handler

```
class CalculateButtonHandler implements Ac
tionListener {

   @Override
   public void actionPerformed(ActionEvent
arg0) {
   areaTF.setText("TEST");;
   }

}
```

Rectangle Program: Sample Run



Area and Perimeter of a Rectangle

Enter the length: 25

Enter the width: 15

Area: 375.0

Perimeter: 80.0

Figure 6-8 Sample run for the final `RectangleProgram`

Programming Example: Temperature Conversion

- Input: Temperature in Fahrenheit or Celsius.
- Output: Temperature in Celsius if input is Fahrenheit; temperature in Fahrenheit if input is Celsius.

Programming Example: Temperature Conversion

Solution:

1. Create the appropriate `JLabels`, `JTextFields`, `JButtons`.
2. Add them to the created content pane.
3. Calculate the appropriate conversions when the buttons are clicked and an event is triggered.

Sample Run for TempConversion

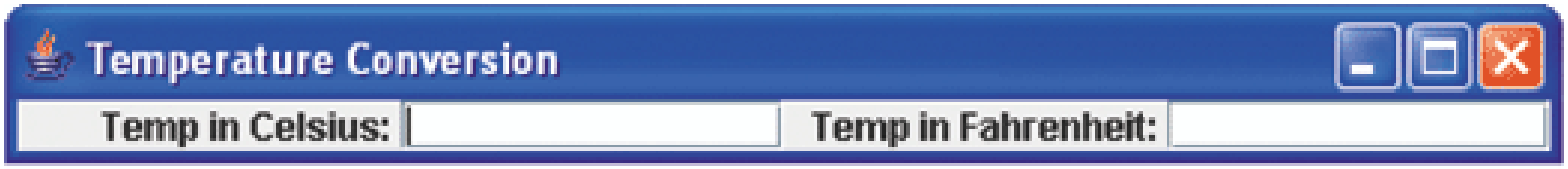
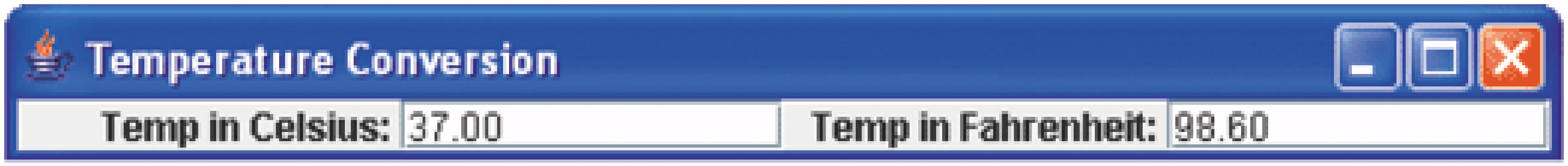


Figure 6-9 GUI for the temperature conversion program

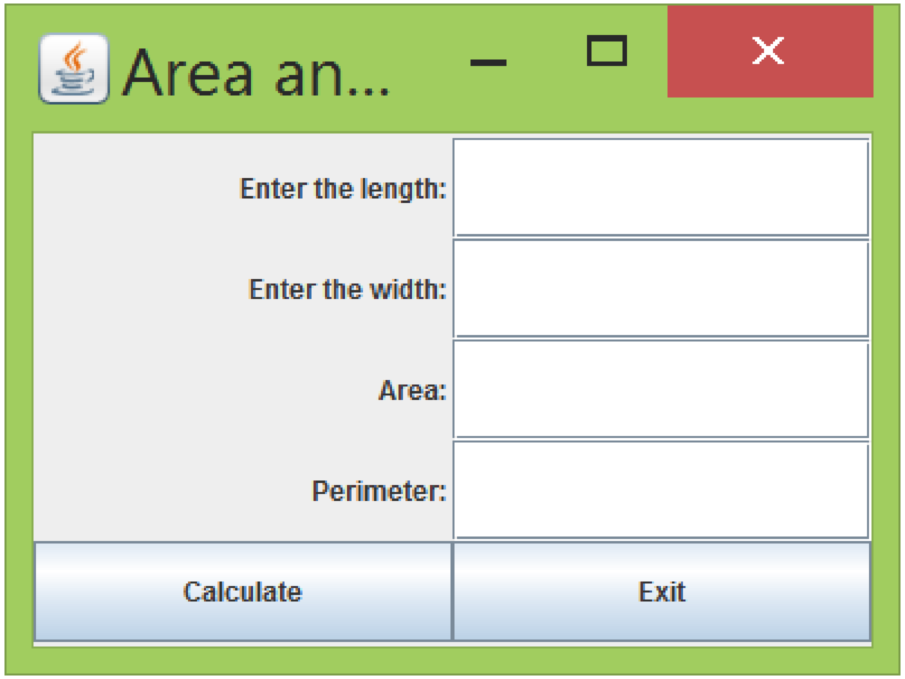


Temp in Celsius: 37.00
Temp in Fahrenheit: 98.60

Figure 6-10 Sample run for `TempConversion`

JFrame, JLabel, JTextField, JButton

- Copy code from the Canvas page and paste



Event-driven Programming

- program's execution is indeterminate
- on-screen components cause *events* to occur when they are clicked / interacted with
- events can be handled, causing the program to respond, *driving* the execution thru events (an "event-driven" program)

Java Event Hierarchy

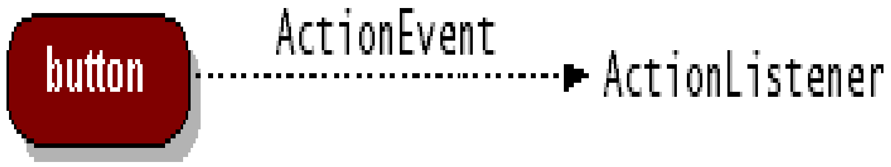
```
java.lang.Object
  +--java.util.EventObject
        +--java.awt.AWTEvent
              +--java.awt.event.ActionEvent
              +--java.awt.event.TextEvent
              +--java.awt.event.ComponentEvent
                    +--java.awt.event.FocusEvent
                    +--java.awt.event.WindowEvent
                    +--java.awt.event.InputEvent
                          +--java.awt.event.KeyEvent
                          +--java.awt.event.MouseEvent
```

Action events: `ActionEvent`

- most common / simple event type in Swing
- represent an action occurring on a GUI component
- created by:
  - button clicks
  - check box checking / unchecking
  - menu clicks
  - pressing Enter in a text field
  - etc.

Listening for events

- attach *listener* to component
- listener's appropriate method will be called when event occurs (e.g. when the button is clicked)
- for Action events, use `ActionListener`



Handling an Event

- Action event: An event created when `JButton` is clicked.
- Event listener: An object that receives a message when `JButton` is clicked.
- In Java, you must register the listener.

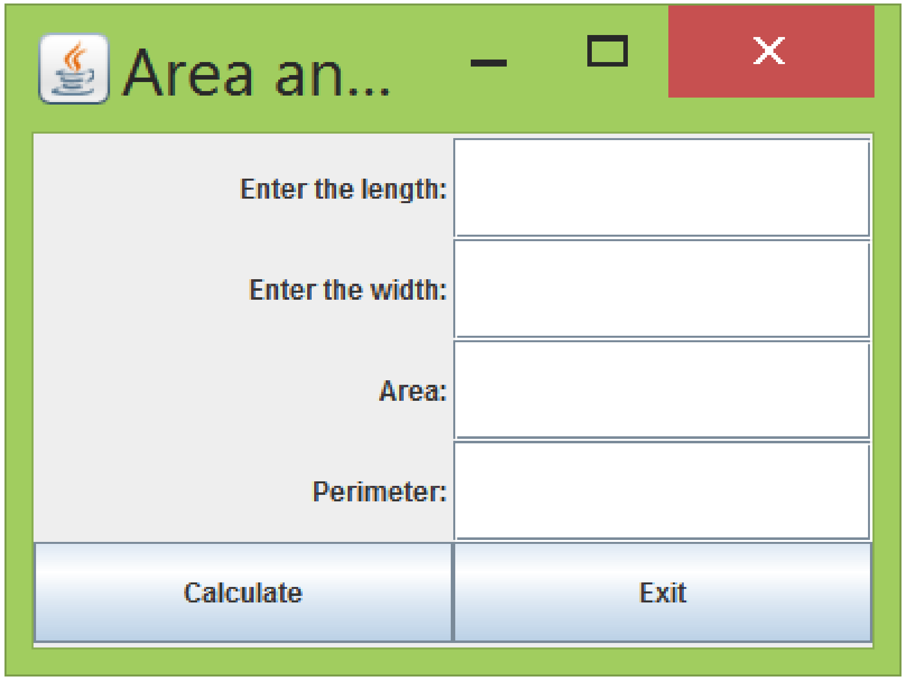
Handling an Event

- `class ActionListener`
  - Handles action event.
  - Part of package `java.awt.Event`.
  - The `class ActionListener` is a special type of class (interface).
  - Must contain the `actionPerformed` method.

Add an Event Handler

```
class CalculateButtonHandler implements Ac
tionListener {

   public void actionPerformed(ActionEvent
arg0) {
        areaTF.setText("TEST");
   }

}
```

Run your program



What is Java Interface?

- Similar to Java classes
- An interface is a collection of method declarations.
- An interface has no variable declarations or method bodies
- Why does Java have these interfaces?
  - Java does not support multiple inheritance
  - A class can be derived from only one class

Interface Examples

```
interface Bicycle {
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement)
;
}
```

```
class MyBicycle implements Bicycl
e {

    void changeGear(int newValue)
{
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement)
{
        speed = speed - decrement;
```

ActionListener

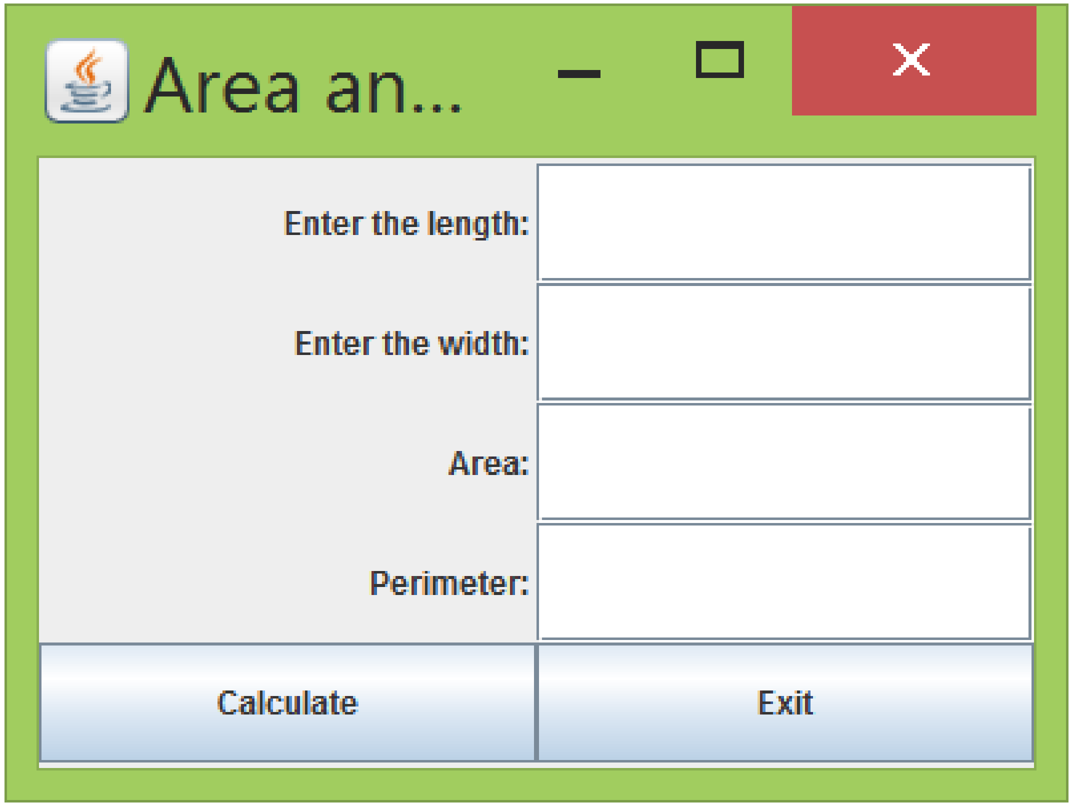
```
interface ActionListener {
        public void actionPerformed(ActionEvent e);
}

class MyActionEventHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                System.out.println(“Hello”);
}
class YourActionEventHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                System.out.println(“Hi!”);
}
```

Get Text from TextField

- JTextField
  - setText()
  - getText()

```
double width, length, area, perimeter;

length = Double.parseDouble(lengthTF.getTex
t());
width = Double.parseDouble(widthTF.getText(
));
area = length * width;
perimeter = 2 * length * width;

areaTF.setText(area + "");
```

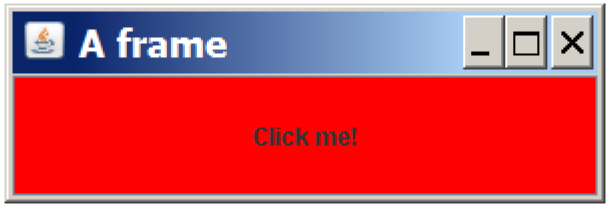
Run your program



Layout

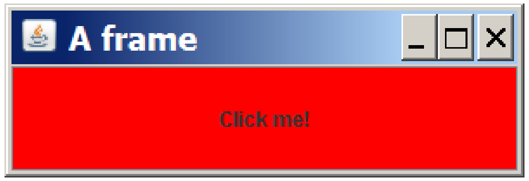
- The following code attempts to show two buttons:

```
JButton button1 = new JButton();
button1.setText("I'm a button.");
button1.setBackground(Color.BLUE);
this.add(button1);

JButton button2 = new JButton();
button2.setText("Click me!");
button2.setBackground(Color.RED);
this.add(button2);
```



Layout problems

- The preceding program added two buttons to the frame, but only one appeared on the screen.



- **layout manager**: An object contained inside frames and other graphical containers that decides the position and size of the components inside the container.

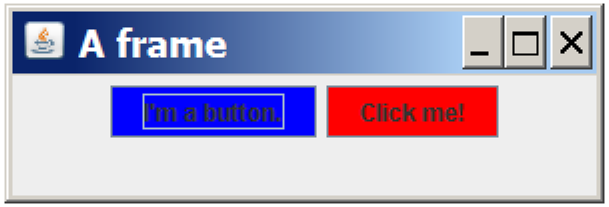
- The default layout manager sizes each component added to occupy the full window space.

Changing layouts

- We can correct the program's appearance by changing the frame's layout manager.
- Change the layout by calling the `setLayout` method on the frame and passing a layout manager object.
  - We will see several layout managers later.
  - We'll use one called a `FlowLayout`, which sizes each component to its preferred size and positions them in left-to-right rows.

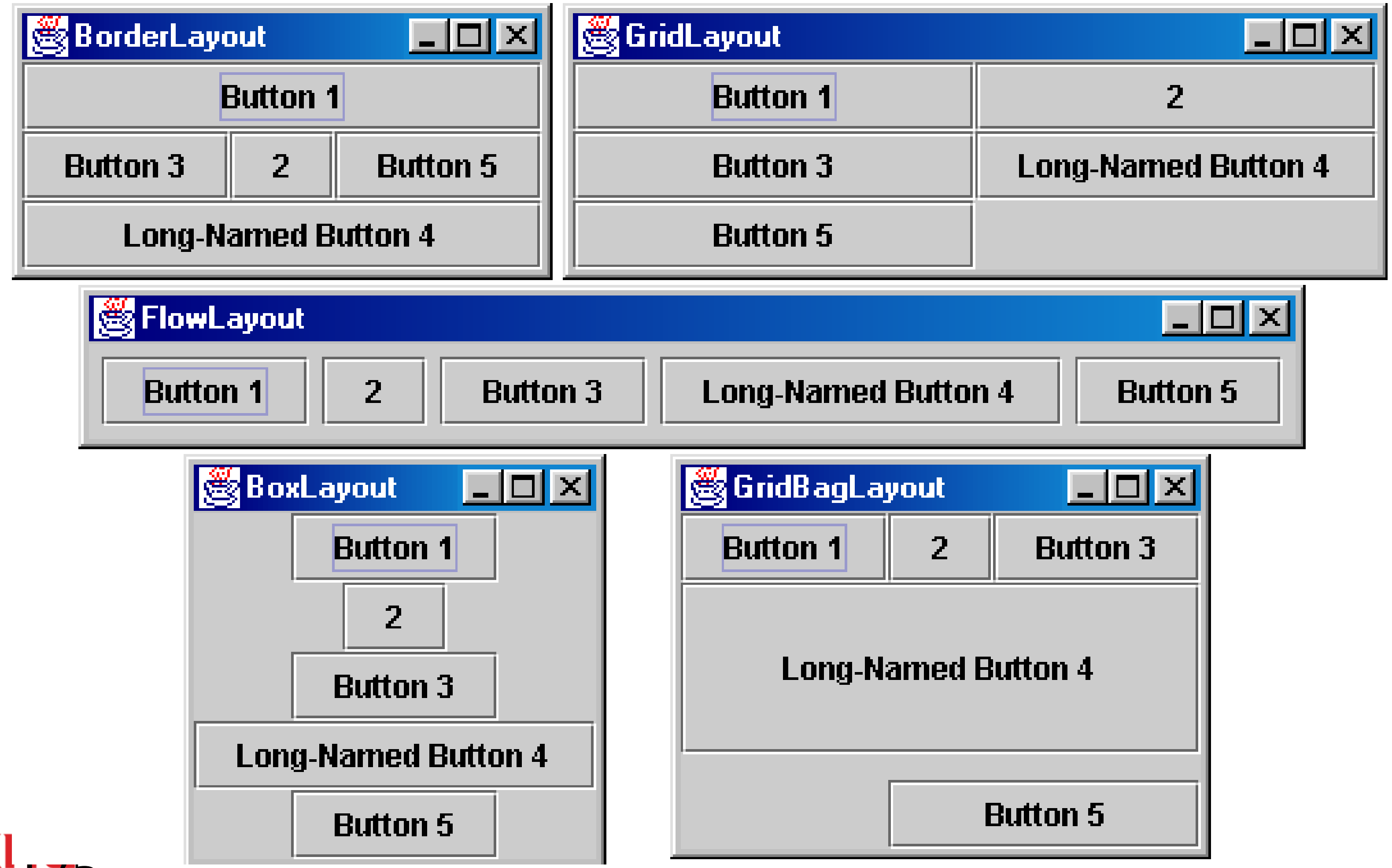
  - If the following line is added to the preceding program just before calling `setVisible(true)`, its appearance will be:

```
frame.setLayout(new FlowLayout());
```



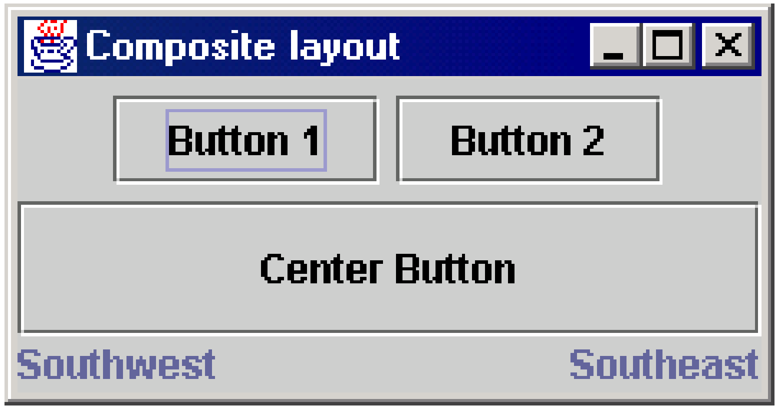
Layout managers

- Java layout managers:



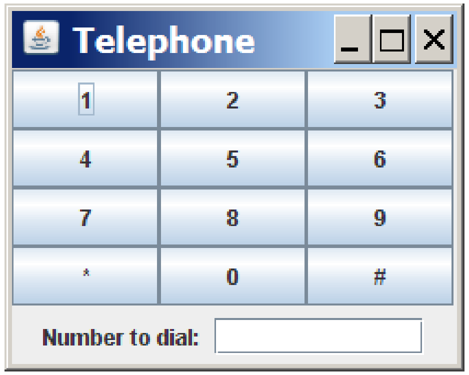
Composite layout

- create panels within panels
- each panel has a different layout, and by combining the layouts, more complex / powerful layout can be achieved
- example:
  - how many panels?
  - what layout in each?

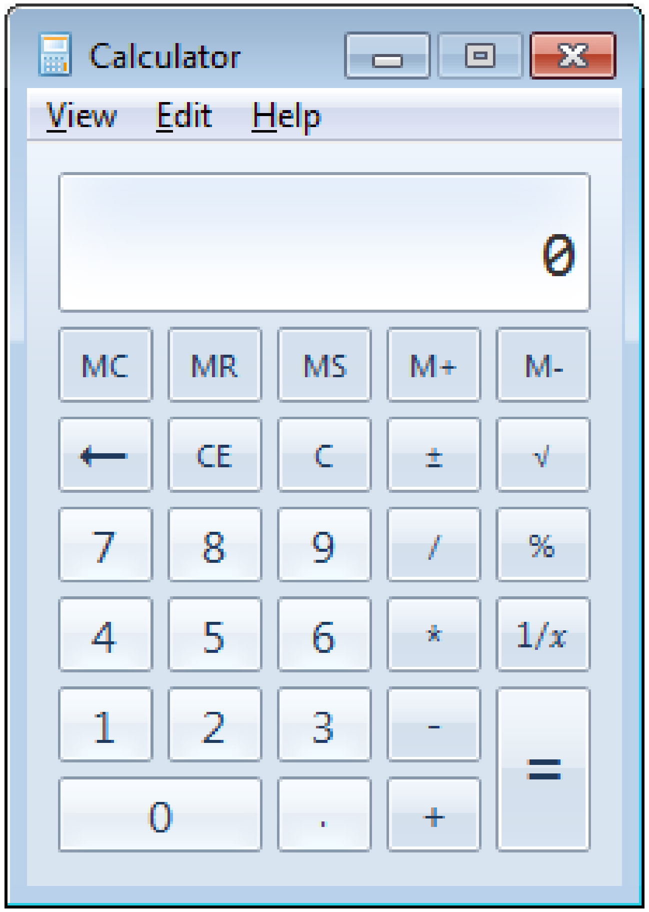


Composite layout example

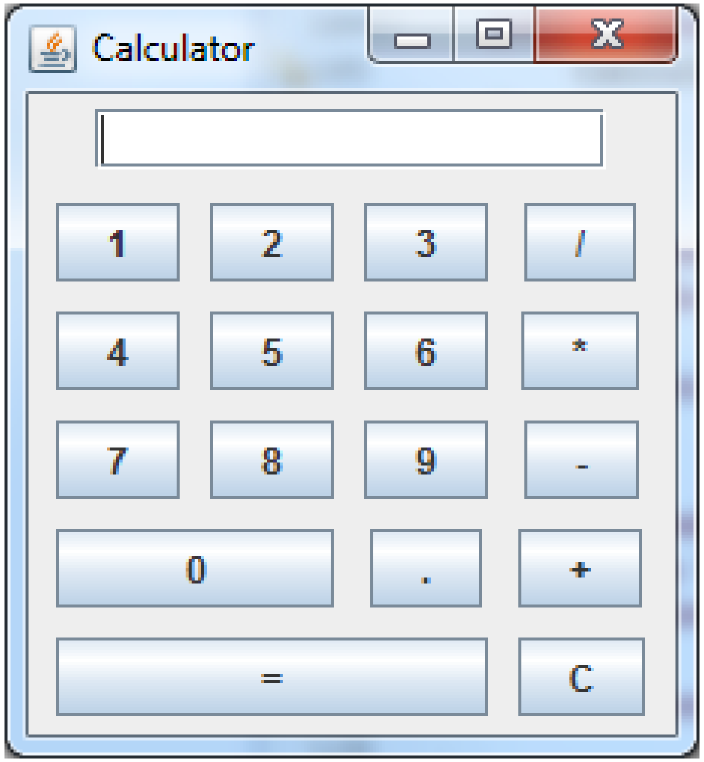
```
this.setLayout(new BorderLayout());

JPanel centerPanel = new JPanel(new GridLayout(4, 3));
for (int i = 1; i <= 9; i++) {
   centerPanel.add(new JButton("" + i));
}
centerPanel.add(new JButton("*"));
centerPanel.add(new JButton("0"));
centerPanel.add(new JButton("#"));

this.add(centerPanel, BorderLayout.CENTER);

JPanel southPanel = new JPanel(new FlowLayout());
southPanel.add(new JLabel("Number to dial: "));
southPanel.add(new JTextField(10));

this.add(southPanel, BorderLayout.SOUTH);
```

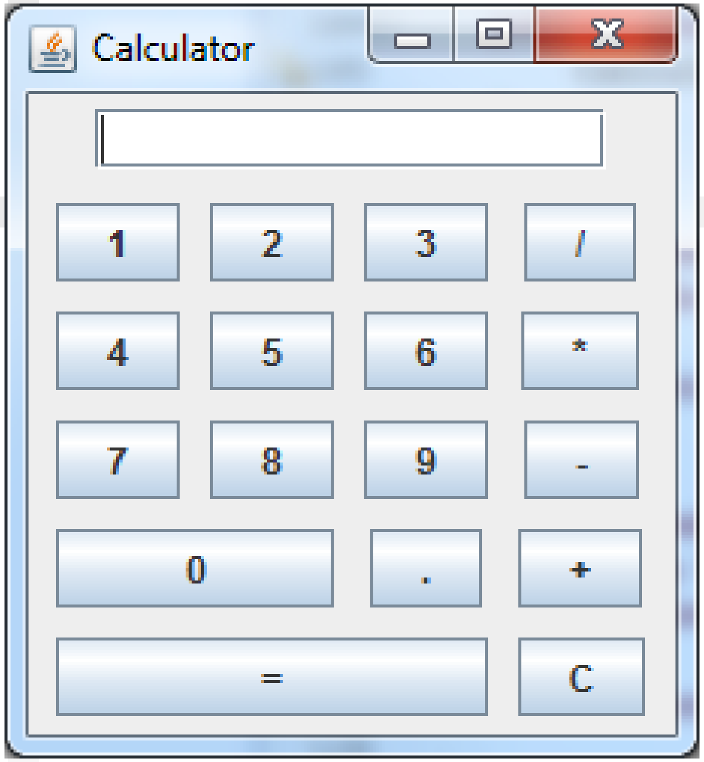


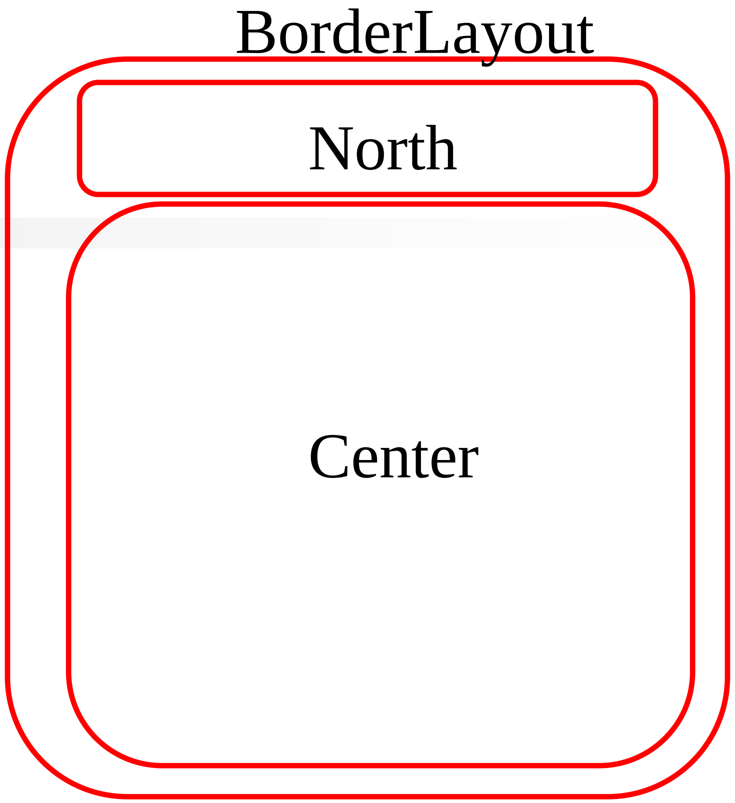
Calculator

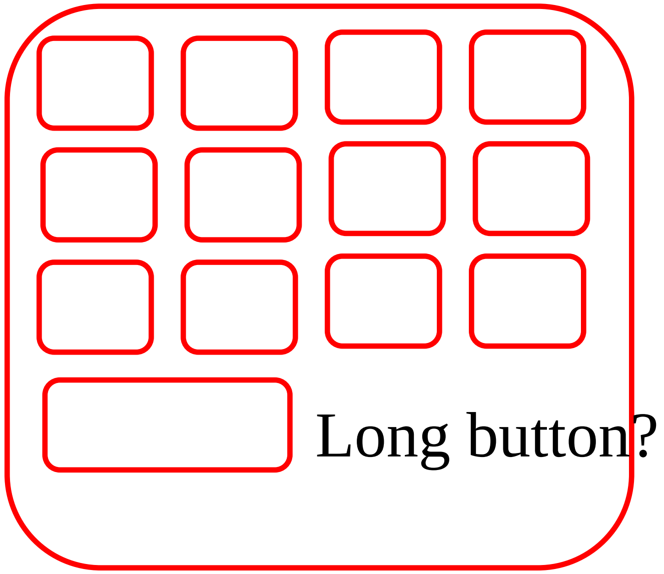


Windows calculator




Calculator
1
2
3
/
4
5
6
*
7
8
9
-
0
.
+
=
C


BorderLayout
North
Center


Long button?

GridLayout?

GridBagLayout

- Behavior
  - Divides the window into grids, without requiring the components to be the same size
  - Each component managed by a grid bag layout is associated with an instance of **`GridBagConstraints`**
    - The `GridBagConstraints` specifies:
      - How the component is laid out in the display area
      - In which cell the component starts and ends
      - How the component stretches when extra room is available
      - Alignment in cells

GridBagLayout: Basic Steps

- Set the layout, saving a reference to it

```
GridBagLayout layout = new GridBagLayout();
setLayout(layout);
```

- Allocate a `GridBagConstraints` object

```
GridBagConstraints constraints =
  new GridBagConstraints();
```

- Set up the `GridBagConstraints` for component 1

```
constraints.gridx = x1;
constraints.gridy = y1;
constraints.gridwidth = width1;
constraints.gridheight = height1;
```

- Add component 1 to the window, including constraints

```
add(component1, constraints);
```

- Repeat the last two steps for each remaining component

GridBagConstraints

- Can reuse the `GridBagConstraints`

```
GridBagConstraints constraints = new GridBagConstraints();
constraints.gridx = x1;
constraints.gridy = y1;
constraints.gridwidth = width1;
constraints.gridheight = height1;
add(component1, constraints);
constraints.gridx = x1;
constraints.gridy = y1;
add(component2, constraints);
```

GridBagConstraints Fields

- gridx, gridy
  - Specifies the top-left corner of the component
  - Upper left of grid is located at (gridx, gridy)=(0,0)
  - Set to **`GridBagConstraints.RELATIVE`** to auto-increment row/column

```
GridBagConstraints constraints = new GridBagConstraints();
constraints.gridx = GridBagConstraints.RELATIVE;
container.add(new Button("one"), constraints);
container.add(new Button("two"), constraints);
```

GridBagConstraints Fields (Continued)

- **gridwidth, gridheight**
  - Specifies the number of columns and rows the Component occupies

    ```
    constraints.gridwidth = 3;
    ```

  - **`GridBagConstraints.REMAINDER`** lets the component take up the remainder of the row/column
- **weightx, weighty**
  - Specifies how much the cell will stretch in the x or y direction if space is left over

    ```
    constraints.weightx = 3.0;
    ```

  - Constraint affects the cell, not the component (use `fill`)
  - Use a value of 0.0 for no expansion in a direction
  - Values are relative, not absolute

GridBagConstraints Fields (Continued)

- **fill**
  - Specifies what to do to an element that is smaller than the cell size

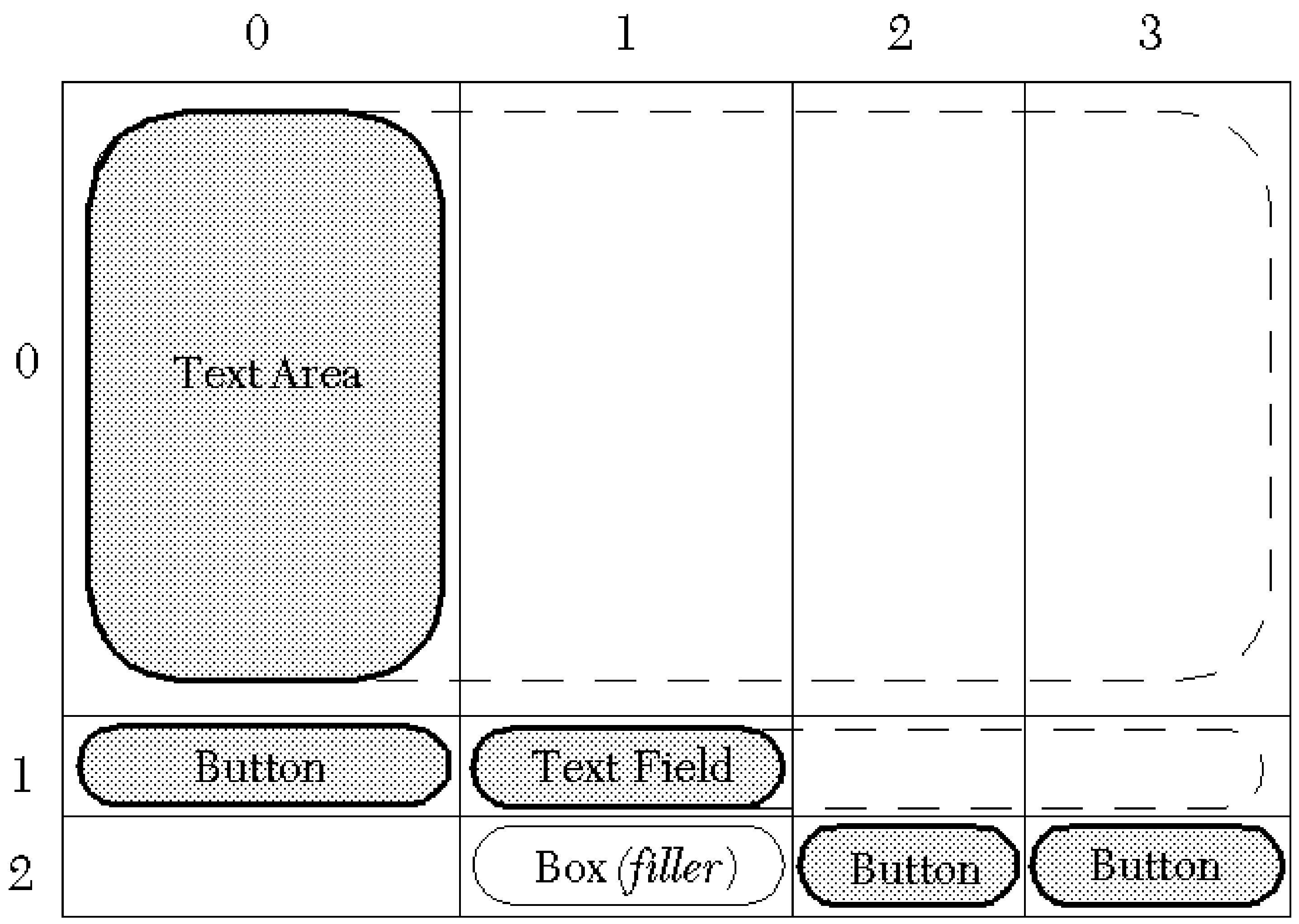
```
constraints.fill = GridBagConstraints.VERTICAL;
```

  - The size of row/column is determined by the widest/tallest element in it
  - Can be `NONE`, `HORIZONTAL`, `VERTICAL`, or `BOTH`
- **anchor**
  - If the fill is set to `GridBagConstraints.NONE`, then the `anchor` field determines where the component is placed

```
constraints.anchor = GridBagConstraints.NORTHEAST;
```

  - Can be `NORTH`, `EAST`, `SOUTH`, `WEST`, `NORTHEAST`, `NORTHWEST`, `SOUTHEAST`, or `SOUTHWEST`

GridBagLayout: Example

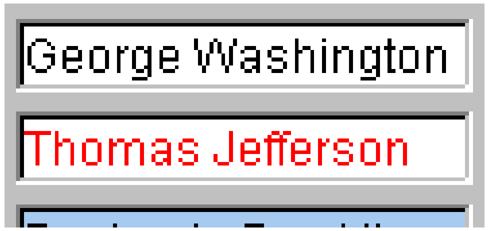


More GUI Stuff

- JFrame, JLabel, JButton
- JComboBox
- JRadioButton
- JTextArea
- JScrollBar
- JSlider
- Etc.

JTextField, JTextArea

A text field is like a label, except that the text in it can be edited and modified by the user. Text fields are commonly used for user input, where the user types information in the field and the program reads it

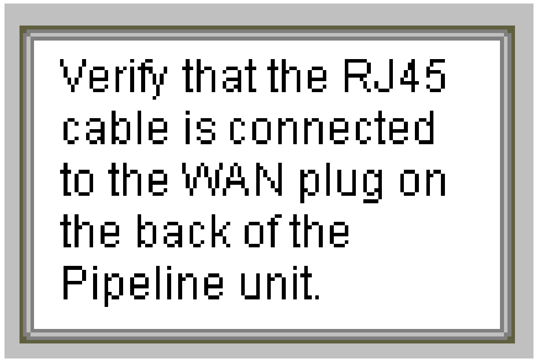

George Washington
Thomas Jefferson

A text area is a multi-line text field

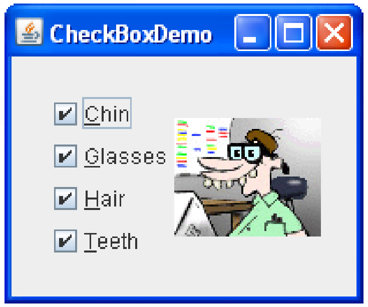
- `public JTextField(int columns)`
- `public JTextArea(int lines, int columns)`
  Creates a new text field that is the given number of columns (letters) wide.

- `public String getText()`
  Returns the text currently in the field.

- `public void setText(String text)`
  Sets field's text to be the given string.


Verify that the RJ45 cable is connected to the WAN plug on the back of the Pipeline unit.

JCheckBox, JRadioButton



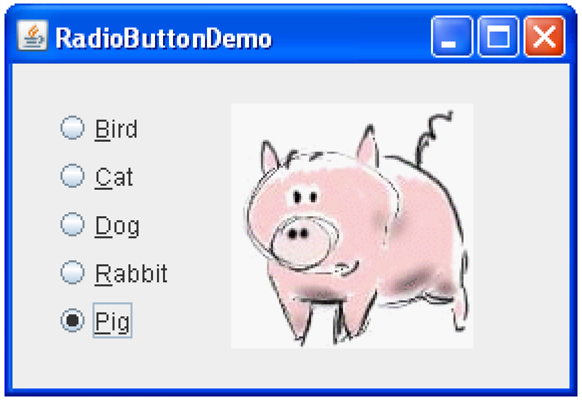
A check box is a toggleable button with two states:
checked and unchecked

A radio button is a button that can be selected; usually part of a group of mutually-exclusive radio buttons (1 selectable at a time)

- `public JCheckBox / JRadioButton(String text)`
  `public JCheckBox(String text, boolean isChecked)`
  Creates checked/unchecked check box with given text.

- `public boolean isSelected()`
  Returns true if check box is checked.

- `public void setSelected(boolean selected)`
  Sets box to be checked/unchecked.



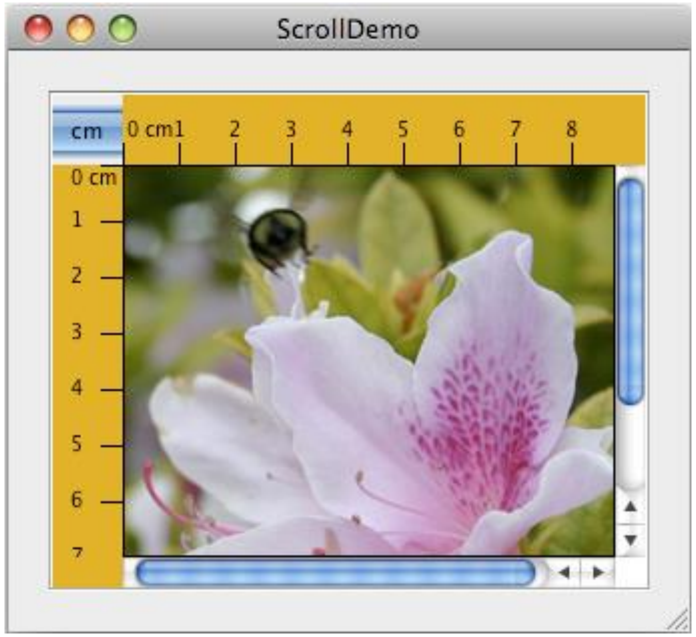
JScrollPane

A special container that holds a component,
using scrollbars to allow that component to be seen

- `public JScrollPane(Component comp)`
  Wraps the given component with scrollbars.

  After constructing the scroll pane, add the scroll pane to the container, not the original component.

  ```
  frame.add(new JScrollPane(area),
  BorderLayout.CENTER);
  ```



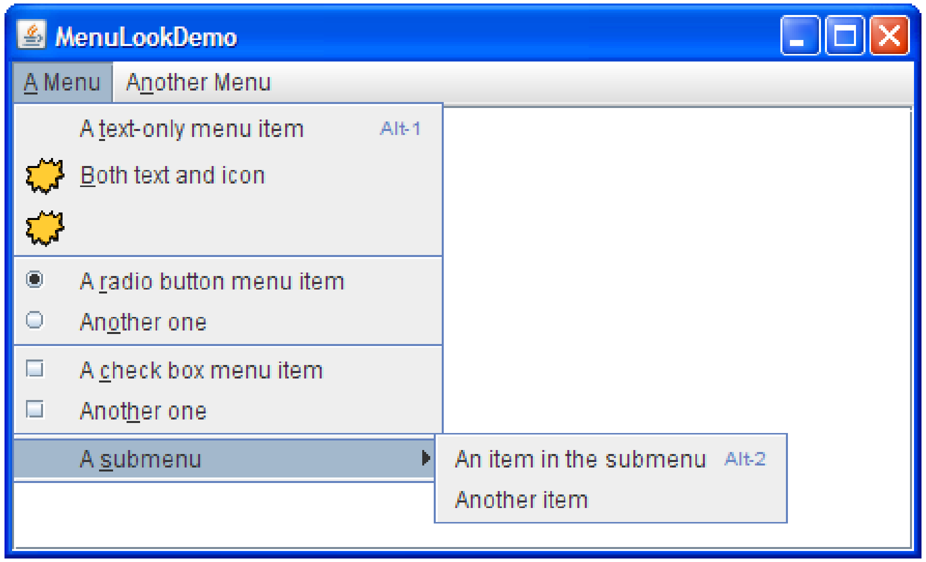
JMenuBar

The top-level container that holds menus; can be attached to a frame

- `public JMenuBar()`
- `public void add(JMenu menu)`

Usage: in `JFrame`, the following method exists:

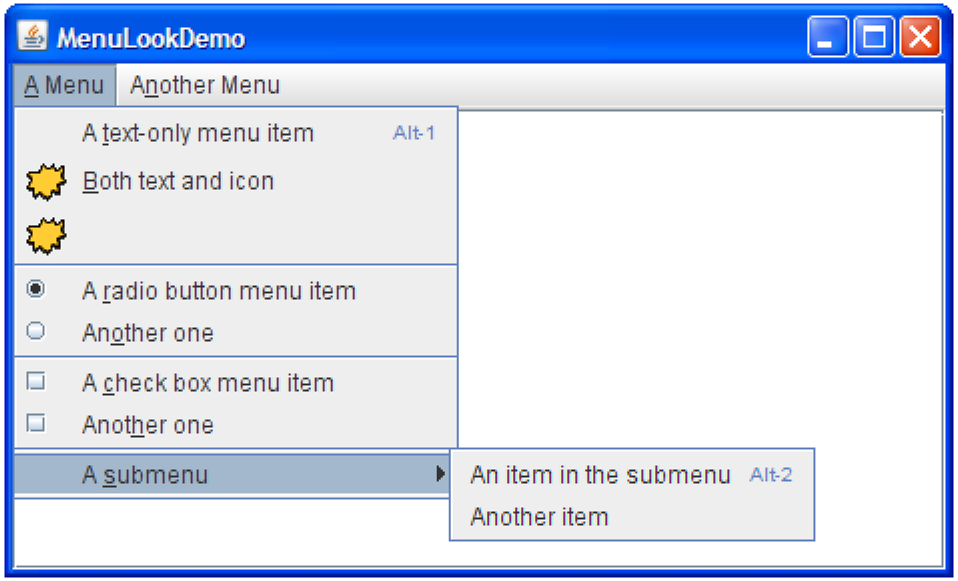
- `public void setJMenuBar(JMenuBar bar)`



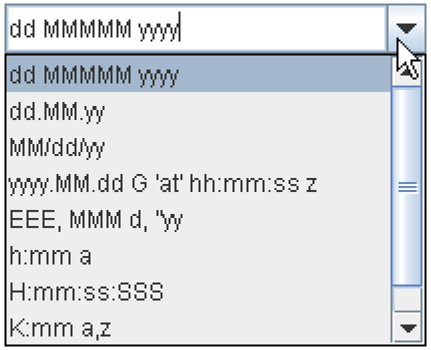
JMenu

A menu to hold menu items; menus can contain other menus

- `public JMenu(String text)`
- `public void add(JMenuItem item)`
- `public void addSeparator()`



JComboBox

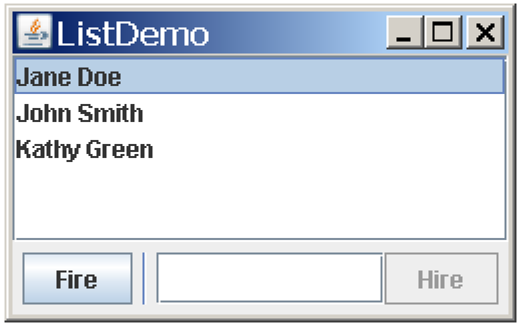


- `public JComboBox()`
- `public JComboBox(Object[] items)`
- `public JComboBox(Vector items)`
- `public JComboBox(ComboBoxModel model)`
  Constructs a combo box. Can optionally pass a vector or model of items. (See `DefaultComboBoxModel` for a model implementation.)

- `public void addActionListener(`
  `ActionListener al)`
  Causes an action event to be sent to listener al when the user selects or types a new item in the combo box.

JList

A list of selectable pre-defined items



- `public JList()`
  Constructs an empty JList.

- `public JList(ListModel model)`
- `public JList(Object[] data)`
- `public JList(Vector data)`
  Constructs a JList that displays the given data.

- `public void addListSelectionListener(`
  `  ListSelectionListener lsl)`
  Adds the given listener to be informed when the selected index / indices change for this list.

Event Handling

- Add ActionListener to all the buttons
  - One action listener for all buttons?
  - Separate action listener for each button?
  - It’s your design choice

Formula Evaluation

- Parsing the formula
- **Grammar:**

  expression = term | expression `+` term | expression `-` term

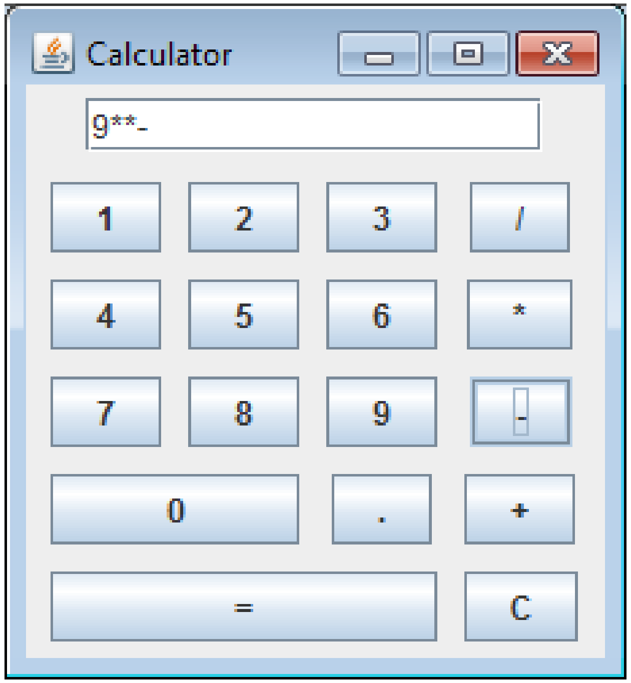
  term = factor | term `*` factor | term `/` factor | term brackets

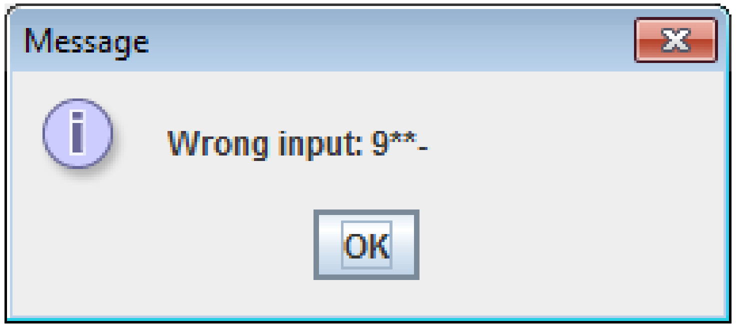
  factor = brackets | number | factor `^` factor

  brackets = `(` expression `)`

Exception Handling

- Show an error message
  - Use JOptionPane.showMessageDialog()

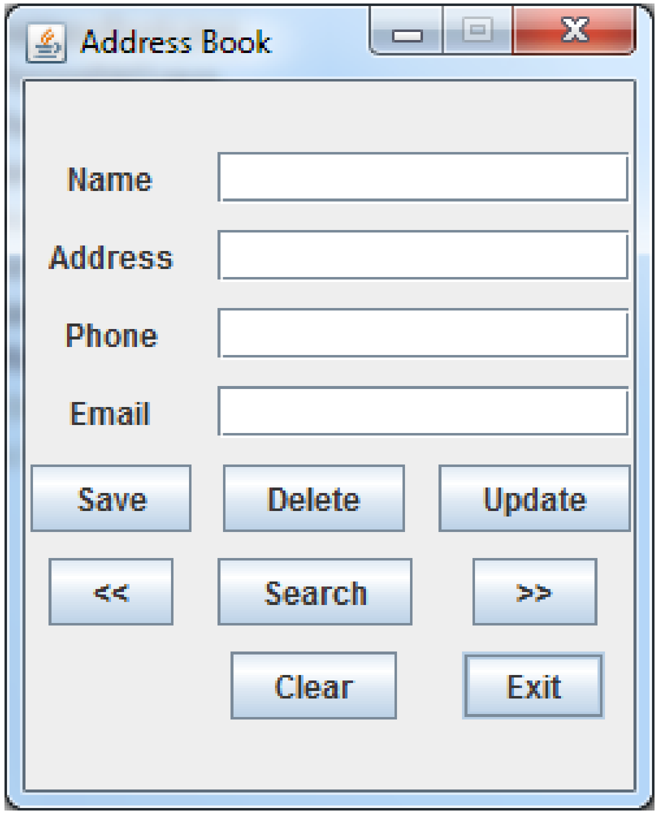




Wrong input: 9**-

Next Lab and H/W: Address Book

- File operations
- Layout
- Event handling


Address Book
Name
Address
Phone
Email
Save
Delete
Update
<<
Search
>>
Clear
Exit