Laboratory 4

Being Classy

Objective

Objects are the basic units of programming in object-oriented languages like Java. We define new types of objects by designing our own classes. A class is the "blueprint" from which objects can be created. This lab introduces us to designing and working with our own classes. We will build a program by creating two custom classes. We will explore information hiding and data abstraction. We will learn about some of the kinds of methods that classes can have.

Key Concepts

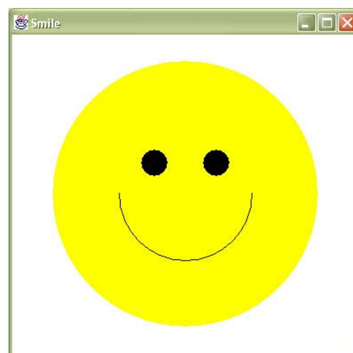
- Class design
- Data abstraction
- Information hiding
- Instance variables
- Accessor methods
- Mutator methods
- Constructors
- Facilitator methods
- **`Graphics.drawArc()`** and **`Graphics.fillArc()`** methods

4.1 GETTING STARTED

- Using the procedures in the introductory laboratory handout, create the working directory **`\javalab`** on the appropriate disk drive and obtain a copy of self-extracting archive **`lab04.exe`**. The copy should be placed in the **`javalab`** directory. Execute the copy to extract the files needed for this laboratory.

4.2 DESIGN

For this lab, we are going to design and create a program that will display a "smiley face" in a window, as in the figure below.



The size of the smiley face is dynamic and dependent upon the size of the window. That is, our program will automatically scale the smiley face to fit properly in whatever size window is created.

- The first step in creating a program to satisfy the specified requirements is the design phase. We need to develop the overall framework for the program. First, how will the user see the program? The user should be able to enter the size of a window and have a window of that size appear with a smiley face in the window. The user is not expected to provide the features of the face or the positions of these features. These will be calculated by our program.
- What variables should the user be able to access? What variables should the user be able to modify?

  ____

  ____

  ____

  ____

- In order to preserve encapsulation, a user should be able to specify only the size of the window. The user does not need to have any understanding of how the smiley face is drawn or scaled to the window.
- However, what do we as programmers need to know to draw a smiley face? List below the variables that we will need in order to render a smiley face in accordance with the specifications of our program.

  ____

  ____

  ____

  ____

  ____

- How many classes should we create? We could create one class **`SmileyWindow`** that reads input from the user, creates a window, and draws a smiley face in the window. In object-oriented programming, it is best to break tasks into easily managed pieces. Let's begin by creating a class **`SmileyFace`** that will do the actual rendering of the face. If we create a separate class, we can reuse that class in another application. For example, perhaps we will want to draw a button with a smiley face on it in a later program. We can create a button and then access the **`SmileyFace`** class to draw the face, instead of creating a completely new class from scratch.

4.3 MORE DESIGN AND CODE PRODUCTION

- Now let's begin building the **`SmileyFace`** class. Our program will compute the location of the eyes and smile on the face automatically. Therefore, we need to know only the diameter of the face and the location of the face.
- Open the file **`SmileyFace.java`**.

```
import java.awt.*;
```

```
public class SmileyFace {
   private int diameter;
   private int x;
   private int y;

   // constructors


   // accessor methods


   // mutator methods


   // facilitator methods
   public void draw(Graphics g) {

   }

   . . .
}
```

- Notice that our **`SmileyFace`** class contains three *instance variables*: **`diameter`**, **`x`**, and **`y`**. Why are the variables declared as **`private`**?
- It is good programming practice to create *accessor* and *mutator* methods for each of our instance variables. Accessor methods typically begin with **`get`** followed by the variable name. Their return type is the same as the variable type. Add accessor methods to your program for each instance variable. We create the first one for you:

```
public int getDiameter() {
       return diameter;
}
```

- Compile your program. Although there is nothing to run right now, compiling your program helps find syntax errors early. After every code revision, it is a good idea to compile the code to check for and to correct syntax errors, such as a missed semicolon or a misspelled variable name.
- Mutator methods are methods that change the value of the variable. Mutator methods typically begin with **`set`** followed by the variable name. Mutator methods are passed a parameter of the same type as the variable. Mutator methods do not return a value, so their return type is **`void`**. Add mutator methods to your program for each instance variable. We create the first one for you:

```
public void setDiameter(int d) {
      diameter = d;
}
```

- In general, it is best to name the parameters in mutators with a name different from the instance variable. However, Java will allow you to have parameters that have the same name as instance variables. How does Java tell which variable is which?

```
public void setDiameter(int diameter) {
   diameter = diameter; // the instance variable diameter is not set!!
}
```

The parameter name has precedence over the instance variable name. In the code above, the *parameter* **`diameter`** is set to the value of the *parameter* **`diameter`**; hardly a useful activity! To set the *instance variable* **`diameter`**, we need to use the following code:

```
   public void setDiameter(int diameter) {
      this.diameter = diameter;
}
```

We will learn more about **`this`** in later chapters. For now, **`this`** refers to the class and **`this.diameter`** refers to the instance variable **`diameter`**.

- Although providing these methods may seem to be unnecessary, the methods are important for achieving *information hiding,* an important software engineering principle. Properly hiding information gives the program, not the user, control over the circumstances under which variables can be accessed or modified. You can also choose to modify the internal representation of the smiley face, but the user will not know the difference because he will still call the same accessors and mutators to set the diameter and coordinates of the face. Separating the internal representation from the user interface, including the accessors and mutators, is another important software engineering principle.

- Now we will add constructors to the code. Add a constructor that initializes the **`diameter`**, **`x`** and **`y`** variables with user-supplied values. To preserve information hiding, make sure that your constructor calls the appropriate mutator methods rather than accessing the variables directly. We provide the header for the constructor:

```
public SmileyFace(int d, int xPos, int yPos) {
}
```

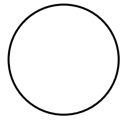
- Do we need any other constructors? We may need to create a **`SmileyFace`** object before we know the diameter and position of the object. In this case, we need a default constructor that has no parameters and simply initializes the object. We can add constants to our object that provide the default value of each instance variable, or we can simply allow the compiler to initialize each **`int`** variable to **`0`** automatically. For now, since we will be setting the diameter and coordinates, we will choose to allow the compiler perform a default initialization for each variable. Add the following default constructor to your code

```
public SmileyFace() {
}
```

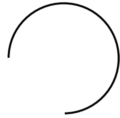
- If you do not have any constructors in your class, the compiler automatically adds a constructor like the one that you just added to your code. However, if you have any constructors in your class, the complier will not add a default constructor and you will need to add it yourself if you want one for your class.

- Now that we have created our accessor, mutator and constructor methods, we can concentrate on the *facilitator* methods. Facilitator methods are methods that help the object perform its intended task. In this case, we need to provide a **`SmileyFace`** object, including its **`diameter`** and **`x`** and **`y`** positions so that it can calculate the coordinates of its features and draw itself. Because of time constraints in the lab, we've provided the calculations for you in the file.

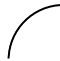
- Let's learn how we draw circles and arcs. Open up the Java API specification and find the method detail for **`Graphics.drawArc()`**, located in the **`java.awt`** package. Fill in the



```
g.drawArc(x, y, width, height, _______, _______ );
```



```
g.drawArc(x, y, width, height, _______, _______ );
```



```
g.drawArc(x, y, width, height, _______, _______ );
```

blanks below to produce Java code to draw the following arcs:

- The method **`Graphics.fillArc()`** works the same way as **`Graphics.drawArc`**, except that it fills the arc with the current Graphics color setting.
- Review the methods **`drawFace`**, **`drawEyes`** and **`drawMouth`**. Do you understand how the methods work? It may help to work through the code with pencil and paper.
- You will notice that instead of calculating the facial feature coordinates and drawing them in a single method, we distributed the tasks across multiple methods. In programming, it is generally recommended that you break down a task into easily manageable pieces. We could have placed all the code to draw the face in the **`draw`** method. But, it is easier to read and debug code that is broken down into manageable pieces, each represented by a separate method. Add the following lines to your object's **`draw`** method:

```
drawFace(g);
drawEyes(g);
drawSmile(g);
```

- Compile your **`SmileyFace`** class. If you get any compiler errors, try to determine where you made any mistakes. If you cannot figure out what is wrong, ask the lab instructor for assistance.

4.4 PUTTING IT ALL TOGETHER

- Open the file **`SmileyWindow.java`**.

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

public class SmileyWindow {
   JFrame window;

   public SmileyWindow() {
      window = new JFrame("Smile");
```

```
      window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   public void setSize(int width, int height) {
      window.setSize(width, height);
   }

   public void paint() {
      window.setVisible(true);
   }

   public static void main(String[] args) throws IOException {
      SmileyWindow me = new SmileyWindow();

      System.out.print(
         "Enter window size : ");
      BufferedReader stdin =
         new BufferedReader(new InputStreamReader(System.in));

      int size = Integer.parseInt(stdin.readLine());

      me.setSize(size, size);
      me.paint();
   }
}
```

- You may have noticed that the window we created is square. Until we learn to write a program that can make decisions with the **`if`** statement, it is easier to determine the diameter of the face using a square window.
- Compile and run the program. It should prompt you for a size and then display an empty window with width and height equal to the size which you provided as input. Do you understand how the program works?
- Now we will add our **`SmileyFace`** to the window. Add an instance variable named **`face`** of type **`SmileyFace`** to your program.

  ```
  SmileyFace face;
  ```

- When we call our **`SmileyWindow`** constructor, it initializes the **`JFrame`** for our window and sets its default close operation to **`EXIT_ON_CLOSE`**. Add a line of code in the constructor to initialize the face variable by calling its default constructor.

  ```
  face = new SmileyFace();
  ```

- When the window size is set in the **`setSize()`** method, we need to set the size of our **`SmileyFace`** object. We will set our **`SmileyFace`** diameter to ¾^{ths} of the window width. The x position of the face will be ½ the distance remaining after the diameter is subtracted from the width of the window. The y position is the same as the x position except that we allowed 10 pixels to accommodate the window title bar. Add the following lines of code to **`setSize()`**:

  ```
  // calculate size of smiley face, 3/4 of the window width
  int diameter = width / 4 * 3;

  int x = (width - diameter ) / 2;
  int y = x + 10;
  ```

- Notice the first line of code:

  ```
  int diameter = width / 4 * 3;
  ```

Why wouldn't we use the following?

```
int diameter = 3 / 4 * width;
```

If we used the preceding line of code the value for diameter would always be set to **`0`**. Why? (*Hint: This has something to do with* **`int`** *arithmetic*.)

____

____

____

____

____

- Use our calculated diameter and x and y positions to set these values in **`face`**. Add the following lines in your **`setSize()`** method following the code you just added to calculate these variables.

```
face.setDiameter(diameter);
face.setX(x);
face.setY(y);
```

- Now our final step is to call **`SmileyFace.draw()`**. In the **`paint()`** method, add the following lines of code:

```
Graphics g = window.getGraphics();
face.draw(g);
```

- We want **`face`** to draw itself in **`window`**, so we call **`window.getGraphics()`** to get the graphics context for **`window`**. We pass this graphics context to **`face`**.
- Compile and run the program. If it does not compile or run correctly, try to determine what the problem is. If you cannot figure it out, ask your lab instructor for assistance.
- When your program runs successfully, try entering different sizes to see if your face is drawn correctly. It is possible to enter sizes that are too small or sizes that are too large. In later chapters we will learn how to address this problem.
- Show your completed program to the lab instructor.

4.5 FINISHING UP

- Copy any files you wish to keep to your own drive.
- Delete the directory **`\javalab`**.
- Hand in your check-off sheet.

Congratulations! You have now finished the fourth laboratory.