STUDENT OUTLINE

Lesson 1 - Introduction to Object Oriented Programming

INTRODUCTION: Before we begin to write actual programs, we need to introduce a few basic concepts of *object-oriented programming*, the style of programming you will learn throughout this curriculum guide. The purpose of this lesson is to give you a feel for object-oriented programming and to introduce a conceptual foundation of object-oriented programming.

The key topics for this lesson are:

A. Classes and Objects
B. Messages and Methods
C. Objects in Software
D. Compiling and Running a Program

VOCABULARY:

OBJECT	CLASS
INSTANCE	MESSAGE
METHOD	ARGUMENT

DISCUSSION: A. <u>Classes and Objects</u>

1. Object-Oriented Programming (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In object-oriented programming, a program consists of a collection of interacting objects. To write such a program you need to describe different types of objects: what they can do, how they are created, and how they interact with other objects.

2. The world in which we live is filled with objects. For example, an object we are all familiar with is a drawing tool such as a pencil or pen. A drawing tool is an object, which can be described in terms of its state and behaviors. The attributes (state) of a pencil are its drawing color, width of the line it draws, its location on the drawing surface, etc. Its behaviors consist of drawing a circle, drawing a line in a forward or backward direction, changing its drawing direction, changing the color, etc.

3. An object in programming is an abstraction for a real-world object. For example, a drawing tool is an attempt to model the attributes and behaviors of a pencil or pen.

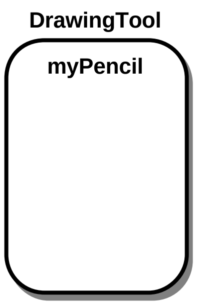


Figure 1.1 - A `DrawingTool` object named `myPencil`

4. To create an object inside the computer program, we must provide a definition for objects - how they behave and what kinds of information they maintain is called a *class*. A class is a kind of mold or template that the computer uses to create objects.

5. A class is like a rubber stamp that can be used many times to make many imprints. Each imprint is an object and each one has its own individual properties such as "size" and "position." Different stampings may have different characteristics, even though they were all made with the same rubber stamp.

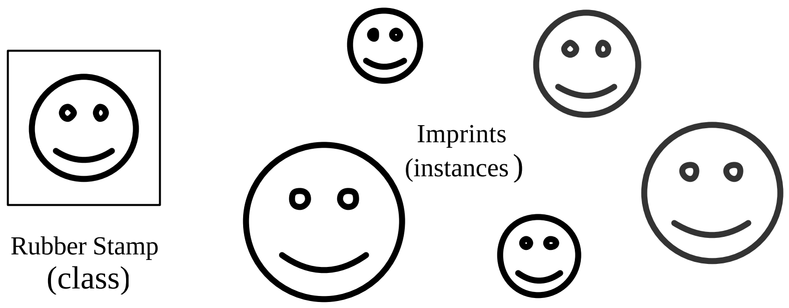


Figure 1.2 – A class and five instances having different values for instance variables "`size`" and "`position`."

6. In OOP terminology, we say the `DrawingTool` object `pencil` is an *instance* of the `DrawingTool` class. An object can only be an instance of one class. In effect, an instance of the class *belongs to* the class.

B. Messages and Methods

1. In writing object-oriented programs we first define classes, and while the program is running, we create objects from these classes to accomplish tasks. A task can range from drawing in a paint program, to adding numbers, to depositing money in a bank account. To instruct a class or an object to perform a task, we send a message to it.

2. You can send a message only to the classes and objects that understand the message. For an object to process the message it receives, it must possess a matching method, which is a sequence of instructions an object follows to perform a task.

3. For example, consider what kind of operations you can carry out with a pencil. You can

   - draw a line in the forward direction
   - change the drawing direction by turning left
   - get the current drawing color

4. Suppose you have an object `myPencil` of type `DrawingTool`. You could represent the behaviors of the `DrawingTool` class with the methods

   - `forward`
   - `turnleft`
   - `getcolor`

5. To draw a line of a specified length, we send the message `forward` along with the distance to move the pencil. A value we pass to an object is called an *argument* of a message. A diagram of sending a message is shown below in Figure 1.3.

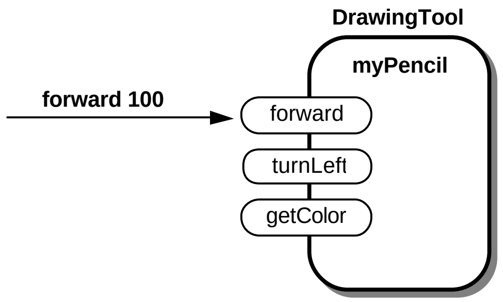


Figure 1.3 - Sending a `forward` message to a `DrawingTool` object

6. The diagram shown in Figure 1.3 illustrates a situation in which an object carries out a request (it draws a line 100 units long) but does not respond to the message sender. In many situations, we need an object to respond by returning a value to the message sender. For example, suppose we want to know the current color that is being used for drawing. We can use the `getColor` message to return the value. A method that returns a value to a message sender is illustrated in Figure 1.4 below.

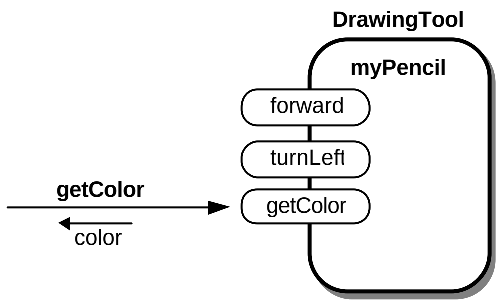


Figure 1.4 - The result of `getColor` is returned to the sender of the message

C. <u>Objects in Software</u>

1. A program is a collection of instructions that, when performed, cause a particular task to be performed on a computer. Individuals who write programs are therefore called programmers. The terms software and code refer to a collection of one or more programs, so programmers are also referred to as software developers.

2. Today, the strategy most often employed by software developers is called *object-oriented-programming (OOP)*. A programmer using an object-oriented strategy begins by selecting objects that can collectively solve the given problem.

3. To illustrate how a particular program might be developed in an OOP fashion, the software developer begins with a set of *program requirements* that specifies the desired task for a program. For example:

   Write a program to draw a square on a piece of paper with a pencil.

4. The program requirements suggest that there are two objects, namely a pencil and piece of paper. One way to determine the objects needed in a program is to search for the nouns of the problem. In our draw square problem, the pencil and paper are examples of such nouns.

5. Once a programmer identifies the objects in the program, the next step is to find or create a class corresponding to each object. Classes are essential because they serve as the places where the code of an object-oriented program resides.

6. Ideally, a programmer *reuses* an existing class, as opposed to writing code for a new class. For the purposes of our drawing example, we will use the preexisting `DrawingTool` and `SketchPad` classes for the pencil and paper objects.

7. Programming languages are like other foreign languages – the first exposure to a written example is bound to seem pretty mysterious. You don't have to understand the details of the program shown below, we'll go over them in the next lesson.

```
import apcslib.*;

public class DrawSquare
{
  public static void main(String[] args)
  {
    DrawingTool pencil;
    SketchPad paper;                          object declarations

    paper = new SketchPad(300, 300);
    pencil = new DrawingTool(paper);

    pencil.forward(100);
    pencil.turnLeft(90);
    pencil.forward(100);                                instructions
    pencil.turnLeft(90);
    pencil.forward(100);
    pencil.turnLeft(90);
    pencil.forward(100);
  }
}
```

Program 1.1 – `DrawSquare.java`

8. The execution of an object-oriented program begins with an initial object. This initial object serves as the starting point for the entire program. For the program in Program 1.1, the initial object belongs to the `DrawSquare` class.

9. The state of an object depends on its components (objects). The `DrawSquare` object includes one `DrawingTool` object declared in the line that begins with the word `DrawingTool` and a `SketchPad` object declared in the line that begins with `SketchPad`. The `DrawingTool` object is given the name `pencil` and the `SketchPad` object is given the name `paper`.

10. An object's behavior is determined by *instructions*. When a program executes, the program's instructions are performed. There are nine instructions for the `DrawSquare` object that are found following the object declaration lines.

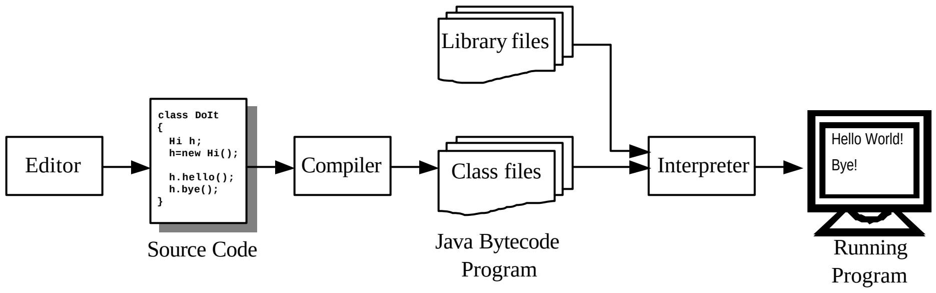
    a. The first instruction will construct a new `SketchPad` object named `paper` with dimensions of 300 by 300.
    b. The next instruction will cause a new `DrawingTool` object named `pencil` to be constructed on the `SketchPad` object named `paper`.
    c. The next line of code will cause the `pencil` to move forward 100 units drawing a line as it goes.
    d. The next line of code will cause the `pencil` to turn to the left 90 degrees.
    e. The remaining 5 steps repeat the process of steps c and d to draw the remaining three sides of the square.

11. The `DrawSquare` example illustrates the tools that a programmer uses to write a program. A program is built from classes that a programmer writes or reuses. Classes are composed from instructions, and these instructions are used in such a way that they manipulate objects to perform the desired tasks.

D. Compiling and Running a Program

1. A programmer writes the text of a program using a software program called an *editor.* The text of a program in a particular programming language is referred to as *source code*, or simply *source*. The source code is stored in a file called the *source file*. For example in the `DrawSquare` example given above, source program would be created and saved in a file named `DrawSquare.java`.

2. Compiling is the process of converting a program written in a high-level language into the *bytecode* language the Java interpreter understands. A Java compiler will generate a *bytecode file* from a source file if there are no errors in the source file. In the case of `DrawSquare`, the source statements in the `DrawSquare.java` source file would be compiled to generate the bytecode file `DrawSquare.class`.



```
class DoIt
{
  Hi h;
  h=new Hi();

  h.hello();
  h.bye();
}
```

Figure 1.5 – From Source Code to Running Program

3. Errors detected by the compiler are called *compilation errors*. Compilation errors are actually the easiest type of errors to correct. Most compilation errors are due to the violation of syntax rules.

4. The Java interpreter will process the bytecode file and execute the instructions in it.

5. If an error occurs while running the program, the interpreter will catch it and stop its execution. Errors detected by the interpreter are called *run-time errors.*

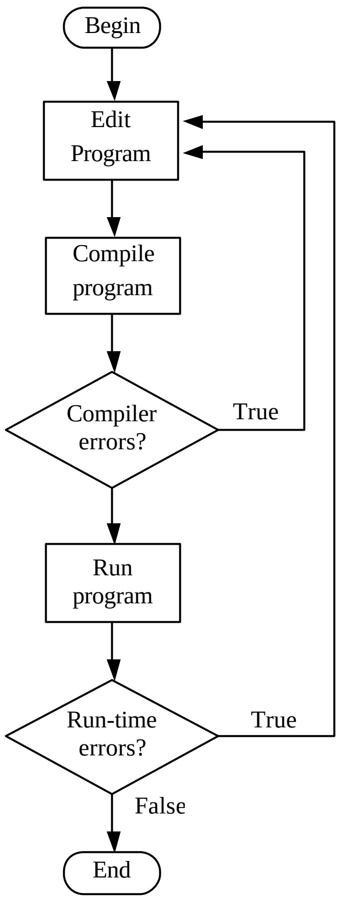


Figure 1.6 – Edit-Compile-Run Cycle for a Java Program

SUMMARY/ REVIEW: One can think of an OOP application as a simulated world of active objects. Each object has a set of methods that can process messages of certain types, send messages to other objects, and create new objects. A programmer creates an OOP application by defining classes of objects.

ASSIGNMENT: Lab Exercise, L.A.1.1, *DrawHouse*

Lab Exercise

DrawHouse

Background:

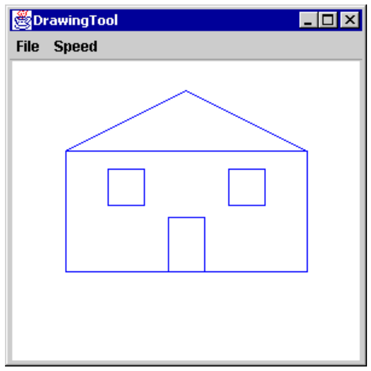
You will be provided with a file named `apcslib.jar`, which contains the code needed to implement the graphics tools to draw objects. The specifications of the drawing tools are provided in *Handout H.A.1.1 – DrawingTool*. Simply place the `apcslib.jar` file in the appropriate folder location so the Java compiler can find it. Then add this line of code

```
import apcslib.*;
```

at the top of your program and the drawing tools are available for use.

Assignment:

Write a program that creates a drawing area of appropriate size (try 500 x 500) and draws a house similar to the one shown below and with these specifications:



1. The house should fill up most of the drawing area, i.e. draw it big.
2. The house should be centered horizontally on the screen.
3. The house must have a sloped roof. It can be of any slope or configuration. But you cannot have a flat roof on the house.
4. Adding a door (centered) and windows is optional.

Instructions:

1. Include your name as a documentation statement and also a brief description of the program.
2. You will need to turn in (either on paper or electronically) a copy of your code and a picture of the house that resulted.

DRAWINGTOOL CLASS SPECIFICATIONS

These classes are not part of Java but are available through the library named `apcslib`. You must have the file `apcslib.jar` in the appropriate directory where Java can access it. To have these classes available in your program, use this command:

```
import apcslib.*;
```

Other features of apcslib will be covered in later lessons.

```
                    DrawingTool
-----------------------------------------------------
   protected double
   protected xPos
   protected yPos
   protected direction;
   protected int width;
   protected boolean isDown;
   protected Color color;
   ...
-----------------------------------------------------
<<constructors>>
   DrawingTool()
   DrawingTool(SketchPad)
   ...

<<accessors>>
   public Color getColor()
   public double getDirection()
   public Point2D.Double getPosition()
   public int getWidth()
   public String toString()

<<modifiers>>
   public void backward(double)
   public void down()
   public void drawString(String)
   public void drawCircle(double)
   public void forward(double)
   public void home()
   public void move(double)
   public void move(double, double)
   public void setColor(Color)
   public void setDirection(double)
   public void setWidth(int)
   public String toString()
   public void turn (double)
   public void turnLeft(double)
   public void turnRight(double)
```

```
      public void up()
      ...
```

Invariant

A DrawingTool object

- Appears in a SketchPad Window (this window is 250 pixels wide and 250 pixels high initially, but can be constructed with different dimensions.)
- The origin (0, 0) is at the center of the drawing window.
- Is directed either up, down, left, or right.
- Is either in drawing mode or in moving mode.

Constructor Methods

`public` `DrawingTool()`

postcondition
- A new `DrawingTool` is created and placed in the center (0, 0) of a `SketchPad` window that is 250 pixels wide and 250 pixels high.
- This object is set to drawing mode.
- The direction for this object is up (90º).
- The DrawingTool color is set to blue.
- The DrawingTool width is 1.

`public` `DrawingTool(SketchPad win)`

postcondition
- A new `DrawingTool` is created and placed in the center (0, 0) of the `SketchPad` window `win`.
- This object is set to drawing mode.
- The direction for this object is up (90º).
- The DrawingTool color is set to blue.
- The DrawingTool width is 1.

Accessor Methods

`public` `String toString();`

postcondition
result = color

`public` `Color getColor();`

postcondition
result = color

`public double` `getDirection();`

postcondition
result = direction

`public int` `getWidth();`

postcondition
result = width

Modifier Methods

`public` `backward (double distance);`

postcondition
- This `DrawingTool` object is moved backward from current direction by `distance` pixels from the old (previous) location.
- If this object is in drawing mode, a line segment is drawn across the `distance` path just traversed.
- The direction is unchanged.
- A 0.5 second delay occurs following this method's execution.

`public` `void down();`

postcondition
- This object is set to drawing mode.

`public` `drawString(String text);`

postcondition
- The string `text` is drawn at the current location using the current `color`.

`public` `drawCircle (double r);`

postcondition
- If the object is in drawing mode, a circle of radius `r` is drawn around the current location using the current `width` and `color`.

`public` `forward(double distance);`

postcondition
- This `DrawingTool` object is moved in the current direction by `distance` pixels from the old (previous) location.
- If this object is in drawing mode, a line segment is drawn across the `distance` path just traversed.
- A 0.5 second delay occurs following this method's execution.

`public` `home();`

postcondition
- The location of the `DrawingTool` object is set to the center of the `SketchPad` window.
- The drawing direction of the object is ups.

`public` `move(double d);`

postcondition
- This `DrawingTool` object is moved in the current direction by `d` pixels from the old (previous) location.
- If this object is in drawing mode, a line segment is drawn across the `d` path just traversed.

`public` `move(double x, double y);`

postcondition

- This `DrawingTool` object is moved from the current position to the position specified by the coordinates `x` and `y`.
- If this object is in drawing mode, a line segment is drawn from the old (previous) position to the absolute position specified by `x` and `y`.

`public` `setColor(Color c);`

precondition

- `c` is a valid Color

postcondition

- The color of the `DrawingTool` object is set to `c`.

`public` `setDirection(double d);`

postcondition

- Sets the direction to `d` degrees. The orientation is `d` degrees counterclockwise from the positive x-axis

`public` `setWidth(int w);`

precondition

- `w` is >= 1

postcondition

- The width of the `DrawingTool` object is set to `w` pixels.

`public` `turn(double d);`

postcondition

- Changes the current direction counterclockwise by `d` degrees from the current direction.

`public` `turnLeft(double degrees);`

postcondition

- Changes the current direction counterclockwise by `d` degrees from the current direction.

`public` `turnRight(double degrees);`

postcondition

- Changes the current direction clockwise by `d` degrees from the current direction.

`public` `up();`

postcondition

- This object is set to moving mode.