Lab 02:
Getting Familiar with ACM_JTF

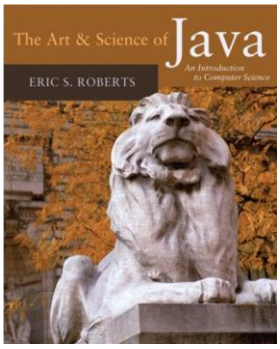
PART1: Introduction to the JTF Packages

In early 2004, the ACM created the Java Task Force (JTF) to review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

This lab is designed to give you a gentle introduction into how to use the JTF materials in the context of an introductory programming course.

Use following resources (especially free book, java api and acm jtf tutorial). If you are not clear about a certain class and its functionality **at any point during the semester**, the tutorial and the book will give you better explanations and examples than api page.

Textbook support for ACM libraries:



http://people.reed.edu/~jerry/121/materials/artsciencejava.pdf

ACM JTF url:

http://cs.stanford.edu/people/eroberts/jtf/

ACM Java API :

http://cs.stanford.edu/people/eroberts/jtf/javadoc/student/index.html

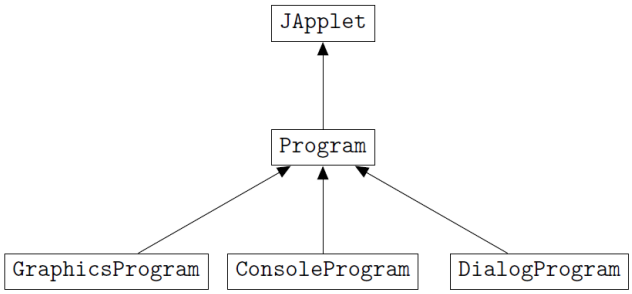
ACM JTF Tutorial

http://cs.stanford.edu/people/eroberts/jtf/tutorial/Tutorial.pdf

SECTION1.1: Getting Started

The program that you would be writing using ACM libraries are *applets*(called JApplet), which are programs that run in the context of a network browser. These are going to be one of three types: GraphicsProgram, ConsoleProgram and DalogProgram.

GraphicsProgram: Program whose window is used for drawing graphics.
ConsoleProgram: Program that installs a console in the window.
DialogProgram: Program that takes its input from a IODialog object



Go and explore these 3 classes in ACM Java API documentation:

The ACM Java Libraries

Packages	
acm.graphics	This package provides a set of classes that support the creation of simple, object-oriented graphica
acm.gui	This package provides a set of classes that support the creation of simple, interactive programs.
acm.io	This package includes two classes that simplify I/O operations.
acm.program	This package provides a set of classes that simplify the creation of programs.
acm.util	This package includes several classes that are common to the ACM package suite.

Package acm.program

This package provides a set of classes that simplify the creation of programs.

See:
Description

Class Summary	
CommandLineProgram	This class simulates the functionality of a ConsoleProgram in an environment that lacks
ConsoleProgram	This class is a standard subclass of Program that installs a console in the window.
DialogProgram	This class is a standard subclass of Program that takes its input from a IODialog object.
GraphicsProgram	This class is a standard subclass of Program whose principal window is used for drawing
Program	This class is the superclass for all executable programs in the acm.program package.
ProgramMenuBar	This class standardizes the menu bars used in the ACM program package.

Importing ACM Library classes

Before you use these classes in your programs you need to import them from the class library. Classes in ACM Java library are grouped into packages

- E.g. acm.programs package contains 6 classes:

acm.program
Classes
CommandLineProgram
ConsoleProgram
DialogProgram
GraphicsProgram
Program
ProgramMenuBar

- To import a specific class into your program, put an import statement at the beginning of the program
  Syntax : `import pacakagename.className;`
- E.g. importing DialogProgram in acm.program package:
  `import acm.program.DialogProgram;`

- To import all the classes contained in a particular package, use *:
- E.g. import all classes in acm.program package:
  `E.g. import acm.program.*;`

Writing first Program

The only way to learn a new programming language is by writing programs in it. Let's write the first program which prints "hello, World" in the console.

1. **ConsoleProgram**
   Program that installs a console in the window. This is like a terminal window where all your user interactions with program(input/output) will be taking place.

   Syntax:

```
import acm.program.ConsoleProgram;

public class YourClassName extends ConsoleProgram
{
    public void init()
    {
            //your initialization code here
    }
    public void run()
    {
            //your code here
    }



}
```

   When you compile and run your program, the main method will run. It automatically calls init() method first and run method next. Init() method is optional. This is where you place your initialization code. For example, code for creating objects, initializing some values etc. will go here. run method is where you place major part of your code that you want to run. Let us start with an example:

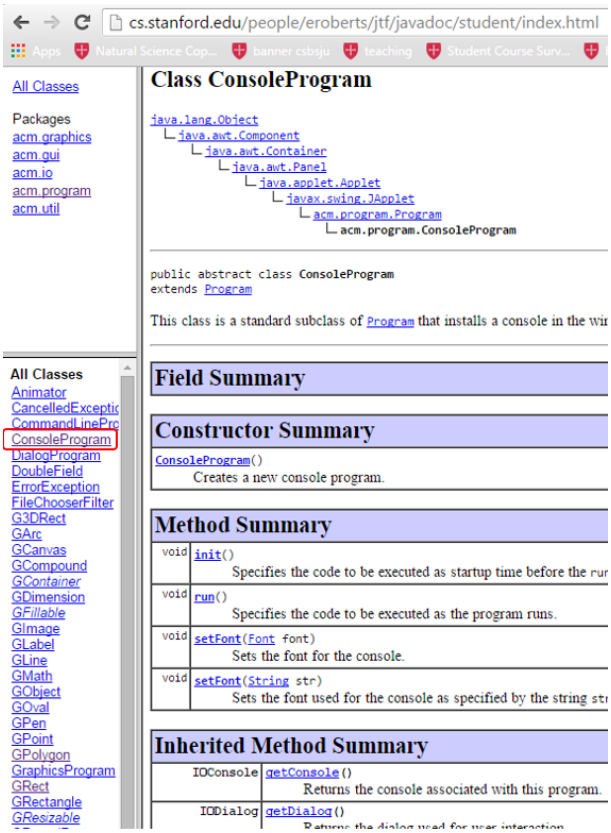
   Write the following program in DrJava:

```
/*
 * File: HelloConsole.java
 * -----------------------
 * This program displays the message "hello, world" and is inspired
 * by the first program "The C Programming Language" by Brian
 * Kernighan and Dennis Ritchie. This version displays its message
 * using a console window.
```

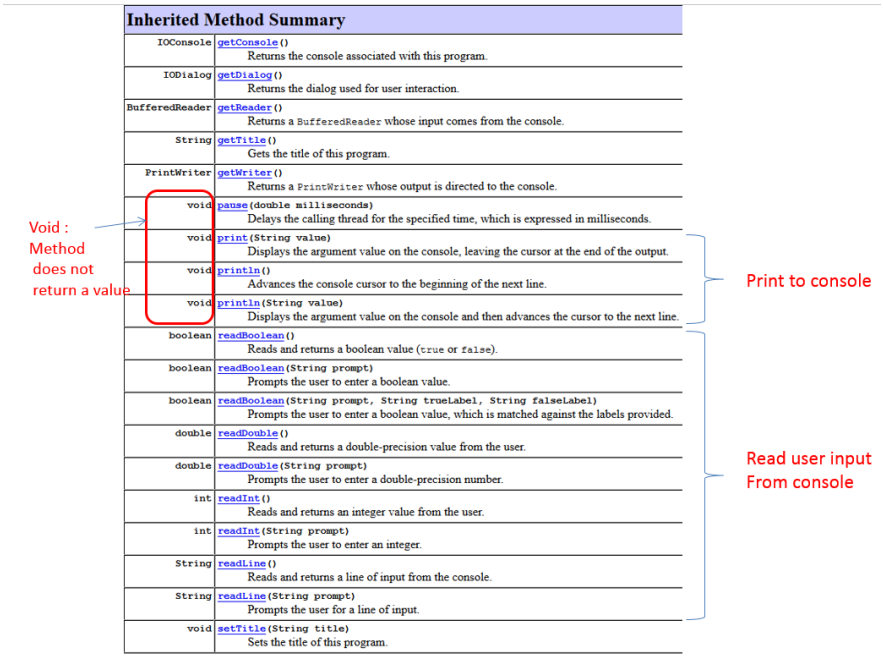
```
 */
import acm.program.ConsoleProgram;

public class HelloConsole extends ConsoleProgram
{
	public void run()
	{
		this.println("hello, world");
	}
}
```

Save it as HelloConsole.java in your Lab2 folder in your home directory. Compile and run the program. If everything is working, the computer should pop up a console window that looks something like this:

This is a simple program that prints “Hello, world” to console. Since no initialization takes place, we omitted the init() method. The statement this.`println("hello, world");` calls the println() method defined in ConsoleProgram class. Any public **method/inherited method** defined in ConsoleProgram can be called in our program. To figure out which methods are defined in a java library class, you need to look at the its API page for that class. Since ConsoleProgram is a ACM library class (not a standard java library class) you need to open it in ACM Java library documentation (http://cs.stanford.edu/people/eroberts/jtf/javadoc/student/index.html) Locate and click on ConsoleProgram class under “AllClasses” tab in left-hand side.



The methods you can call in your program (that extends ConsoleProgram) are listed under Method Summary and Inherited method summary. All of the input (read methods) and output (print methods) set of methods are listed under inherited methods:



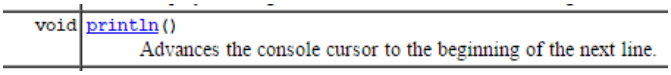
You can call any of these methods in your program. To call a method of a class, say X, we generally need to create an object from this class and call the method on it (we will cover this on part 2). However, you can directly call a method of X without creating objects of X in another class (e.g. HelloConsole.java) if it contains the suffix "extends X" in its class definition.

For example, note that in our HelloConsole program this was the case (it extends ConsoleProgram):

```
public class HelloConsole extends ConsoleProgram
{
```

To call a method, you simply give the method name (optional list of parameters if specified). If there is a return value, you can capture it with a matching data type variable.

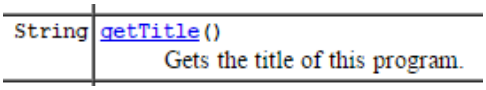
Example1: no input parameters, no return value(void)



Example use :

```
this.println();
```

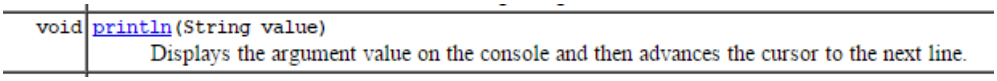
Example2: no input parameters, has return value



Example use :

```
String title = this.getTitle();
```

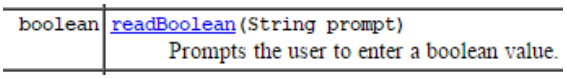
Example3: has parameters, no return value(void)



Example use :

```
this.println("testing123");
```

Example4: has parameters, has return value



Example use :

```
booelan continue = this.readBoolean("Do you wish to continue?");
```

This basic rule of calling methods applies when you want to call a method of any class X (say ConsoleProgram) in another class Y (HelloConsole) that extends X.

2. DialogProgram

Program that takes its input from a IODialog object

Syntax:

Same as Console program. Only difference is it extends DialogProgram

```
import acm.program.DialogProgram;

public class YourClassName extends DialogProgram
{
    public void run()
    {
            //your code here
    }




}
```

Write and save following program as HelloDialog.java.

```
/*
 * File: HelloDialog.java
 * ----------------------
 * This program displays the message "hello, world" and is inspired
 * by the first program "The C Programming Language" by Brian
 * Kernighan and Dennis Ritchie.  This version displays the message
 * in a dialog box.
 */

import acm.program.*;

public class HelloDialog extends DialogProgram
{
        public void run()
        {
                this.println("hello, world");
        }
}
```

If you compile and run the HelloDialog.java program in precisely the same way that you ran HelloConsole.java, the "hello, world" message won't appear in a console window. In fact, the program doesn't create a program frame at all. Instead the program pops up an interactive dialog box that looks something like this, although the precise format of the display will vary depending on what operating system you are using and what "look and feel" it defines for Java applications:

3. GraphicsProgram

Display graphics on a canvas (labels, rectangles etc.)

Syntax:

```
import acm.graphics.*;
import acm.program.GraphicsProgram;

public class YourClassName extends GraphicsProgram
{
  public void run()
  {
    //your code here
  }



}
```

Note that in addition to importing GraphicsProgram class, we are also importing all classes in acm.graphics package. Go and explore classes in this package in acm api. You will see these are classes representing graphical objects (e.g. GLabel for labels, GRect for rectangles) you may want to place on the canvas when running your graphics program.

Write and save the following program as HelloGraphics.java.

```
/*
 * File: HelloGraphics.java
 * ------------------------
 * This program displays the message "hello, world" and is inspired
 * by the first program "The C Programming Language" by Brian
 * Kernighan and Dennis Ritchie.  This version displays the message
 * graphically.
 */

import acm.graphics.*;
import acm.program.*;

public class HelloGraphics extends GraphicsProgram
{
	public void run()
	{
		GLabel label = new GLabel("hello, world");
		label.setFont("SansSerif-100");
		double x = (this.getWidth() - label.getWidth()) / 2;
		double y = (this.getHeight() + label.getAscent()) / 2;
		this.add(label, x, y);
	}
}
```

The HelloGraphics.java file uses the facilities of the acm.graphics package to display the message in large, friendly letters across the window:

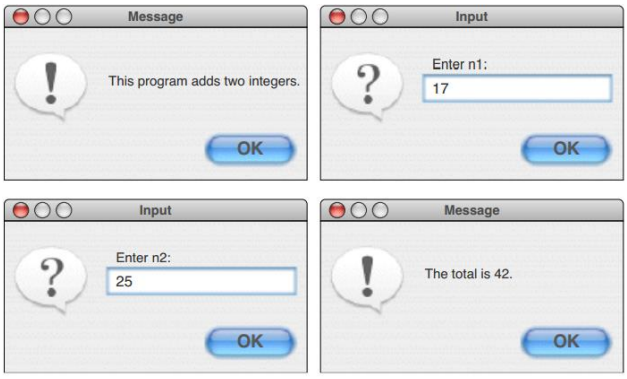
The details of the HelloGraphics the program is not important at this point. Even so, the basic idea is likely to be clear, even if you could not have generated the code as it stands. The first line creates a GLabel object with the message text, the second line gives it a larger font, and the last three lines take care of adding the label so that it is centered in the window. What is important to notice is that the HelloGraphics class extends GraphicsProgram, which is yet another category of program. These three classes—ConsoleProgram, DialogProgram, and GraphicsProgram—are the building blocks for Java applications built using the acm.program package, which is introduced in the following section.

Show the output of three programs to the TA before you move to next section.

Given below is a console program Add2Console.java, that prompts the user for two numbers and prints the total:

```
/*
 * File: Add2Program.java
 * ----------------------
 * This program adds two numbers and prints their sum.  Because
 * this version is a Program, input and output are assigned to
 * System.in and System.out.
 */
import acm.program.*;

public class Add2Program extends ConsoleProgram
{
	public void run()
	{
		this.println("This program adds two numbers.");
		int n1 = this.readInt("Enter n1: ");
		int n2 = this.readInt("Enter n2: ");
		int total = n1 + n2;
		this.println("The total is " + total + ".");
	}
}
```

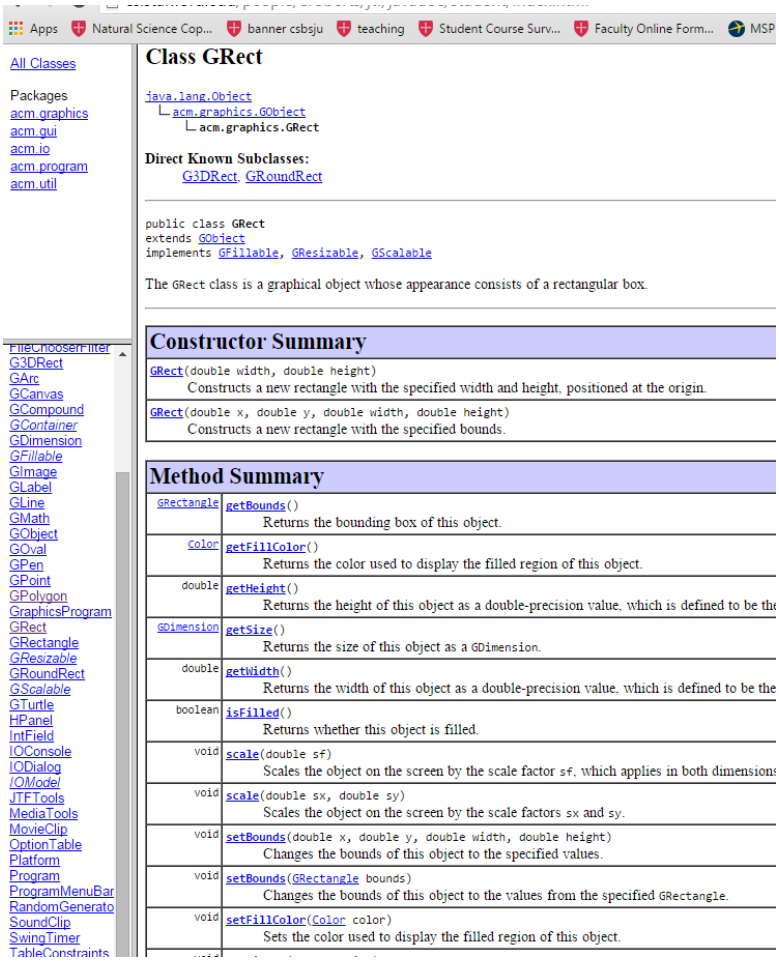
Write, compile and run the program. Now write a program called **Add2Dialog.java** that extends DialogProgram to do the same. The program should produce output like the following:

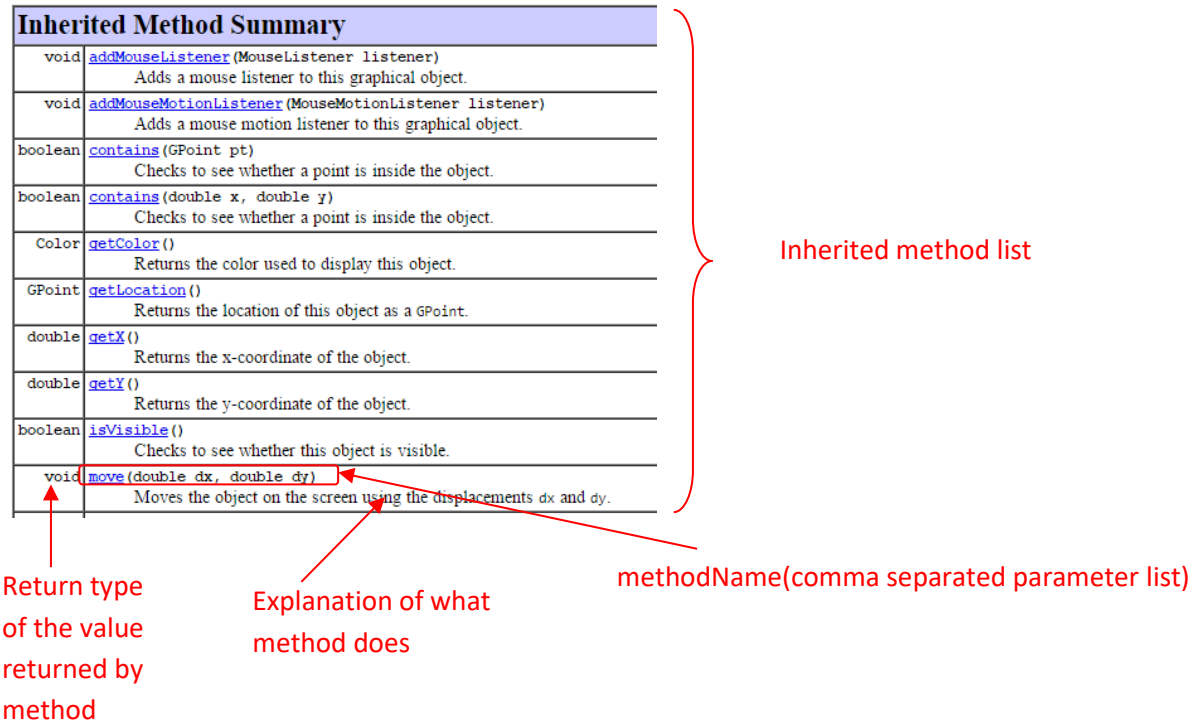


show the output of both Add2Console.java and Add2Dialog.java to TA before proceeding to next section.

PART2: Creating objects from classes

Java ACM library offers many predefined classes for you to use. These classes typically contain 1 or more methods which represent the functionalities or behaviors of the class. For example, GRect class represents a graphical rectangle that you can display in a graphics program and has many methods that allow us to manipulate a rectangle object. Open the acm java api documentation for GRect class and explore:





You will notice that there are 3 important areas:

1. Constructors
2. Methods
3. Inherited methods

Constructors let you create rectangle objects from GRect class and Methods and inherited methods let you manipulate those created objects (e.g. move, scale, fill with color). We are next going to study these 3.

1. Creating objects

To use the methods defined in a class you need to create object from those classes first. To use a predefined class in your program you should import that class. For example, if you need to use GRect class you should import it as follows:

```
import acm.graphics.GRect;
```

you can look at API page to figure out fully qualified name (packagename.classname):

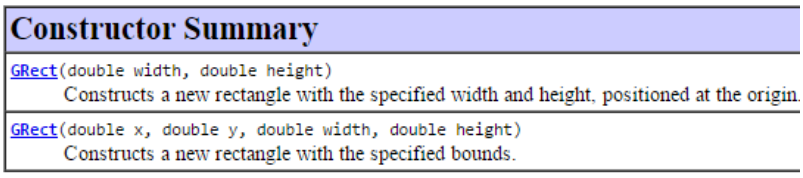
To create an object, we need to use one of the available constructors from the constructor list of the class.

The syntax for creating an object from a class is as follows:

```
ClassName objectname = new ClassName(list of parameters)
```

ClassName is the class which you are trying to create an object of. `objectname` is any variable name you give to the object. You should give some meaningful name. `new` is a keyword. List of parameters are decided based on which constructor you plan to use.

For example, let us say we want to create an object of GRect class. We have 2 constructors available:



Constructor Summary

GRect(double width, double height) Constructs a new rectangle with the specified width and height, positioned at the origin.
GRect(double x, double y, double width, double height) Constructs a new rectangle with the specified bounds.

If we use first constructor to create the object, we are required to specify values for width and height. The types of the values you give should match the types specified in the constructor and the order you specify them should be the same order in API documentation. For example, let us create a rectangle object of width 2.0 and height 5.0 as follows:

```
Grect rectangle1 = new GRect(2.0, 5.0);
```

example with constructor2(rectangle object at x=100, y=50 of width 2.0 and height 5.0):

```
Grect rectangle2 = new GRect(100.0, 50.0, 2.0, 5.0);
```

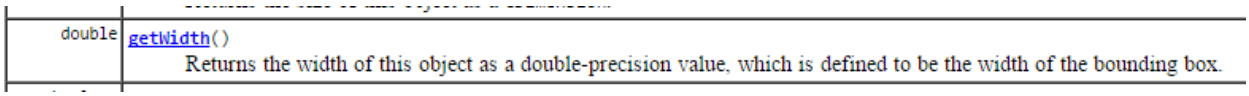
2. Manipulating objects

Now that we created an object we can manipulate it by calling methods/inherited methods defined for that class.

Syntax for calling a method on an object is as follows:

```
objectName.methodName(parameterList)
```

For example, let us say we want to find the width of the `rectangle1` object. Then what we need to do is go to ACM Java API page for GRect and read method/inherited method explanations in there to find a matching one. I found that the getWidth() method will get our job done:

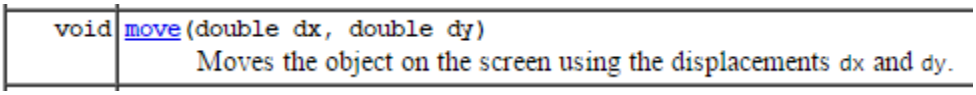


double	getWidth() Returns the width of this object as a double-precision value, which is defined to be the width of the bounding box.

So, we can call it as follows (since there is a return value I will capture it using a variable with matching data type. If the return type is void you do not need to do this)

```
double width = rectangle1.getWidth();
```

Another example: say we want to move our rectangle1 by 10 pixels in the x direction and 20 pixels in the y direction. Browsing the API page again I found following method will get our job done:



void	move(double dx, double dy) Moves the object on the screen using the displacements dx and dy.

So we can call it as follows (notice that the return type is void, so there is no need to capture a return value)

```
rectangle1.move(10.0, 20.0);
```

The HelloGraphics class you wrote earlier does exactly this. It creates a GLabel object called label with text "Hello, World" in it. Then it calls 3 methods on this object: setFont(), getWidth(), getAcent() to manipulate that object:

```
import acm.graphics.*;
import acm.program.*;

public class HelloGraphics extends GraphicsProgram {

 public void run() {
  GLabel label = new GLabel("hello, world");
  label.setFont("SansSerif-100");
  double x = (getWidth() - label.getWidth()) / 2;
  double y = (getHeight() + label.getAscent()) / 2;
  add(label, x, y);
 }

/* Standard Java entry point */
/* This method can be eliminated in most Java environments */
 public static void main(String[] args) {
  new HelloGraphics().start(args);
 }
}
```

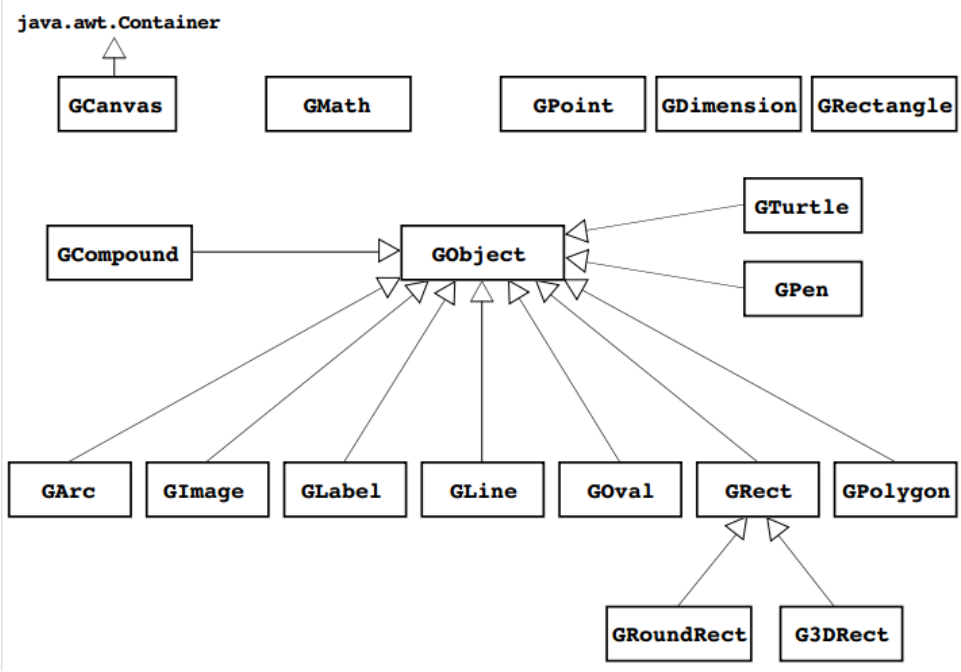
Create Glabel object

Part3 of this tutorial contains another similar example(FeltBoalrd.java) that creates a FeltBoard from graphics objects.

Most of the classes work this way, in that you can create objects from them and manipulate those objects by calling methods (with a few exceptions that you will learn later). So, if you completed Part1 and 2 successfully, you should be able to use ACM Java API to create objects of any ACM library classes you want and call methods on them with no issues.

PART3: Using the acm.graphics package

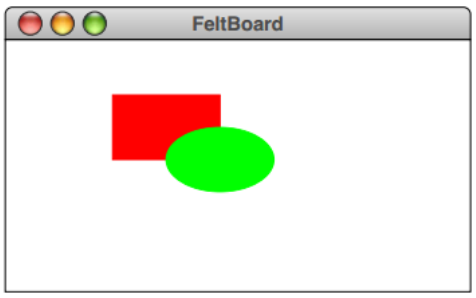
The class structure of acm.graphics package appears in following figure:



Conceptually, GObject represents the universal class of graphical objects that can be displayed (GLine, GOval, etc). When you use acm.graphics, you assemble a picture by constructing various graphical objects (GObjects such as GLine, GOval, etc). and adding them to a canvas (GCanvas) at the appropriate locations.

FeltBoard Example

Let us build a GraphicProgram modeling a felt board—the sort one might find in an elementary school classroom. A child creates pictures by taking shapes of colored felt and sticking them onto a large felt board that serves as the background canvas for the picture. The pieces stay where the child puts them because felt fibers interlock tightly enough for the pieces to stick together. The following figure shows a graphical model of a felt board.



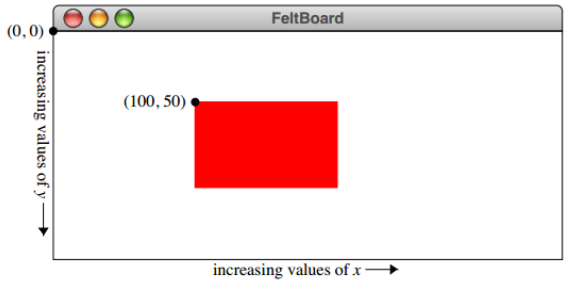
To create the picture, you would need to create two graphical objects—a red rectangle and a green oval—and add them to the graphical canvas that forms the background. The code for the FeltBoard example appears below:

```
/*
 * File: FeltBoard.java
```

```
 * --------------------
 * This program offers a simple example of the acm.graphics package
 * that draws a red rectangle and a green oval. The dimensions of
 * the rectangle are chosen so that its sides are in proportion to
 * the "golden ratio" thought by the Greeks to represent the most
 * aesthetically pleasing geometry.
 */
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class FeltBoard extends GraphicsProgram
{
  /** Constant representing the golden ratio */
  public static final double PHI = 1.618;

  /** Runs the program */
  public void run()
  {
    GRect rect = new GRect(100, 50, 100, 100 / PHI);
    rect.setFilled(true);
    rect.setColor(Color.RED);
    this.add(rect);
    GOval oval = new GOval(150, 50 + 50 / PHI, 100, 100 / PHI);
    oval.setFilled(true);
    oval.setColor(Color.GREEN);
    this.add(oval);
  }
}
```

Coordinate System

The coordinate system The acm.graphics package uses the same basic coordinate system that traditional Java programs do. Coordinate values are expressed in terms of pixels, which are the individual dots that cover the face of the screen. Each pixel in a graphics window is identified by its x and y coordinates, with x values increasing as you move rightward across the window and y values increasing as you move down from the top. The point (0, 0)—which is called the origin—is in the upper left corner of the window. Java coordinate system is shown below:



Lookup and study Javadoc pages for GCanvas, GRect, and GOval classes. For the most part, however, you won't use the GCanvas class directly but will instead use the GraphicsProgram class, which

automatically creates a GCanvas and installs it in the program window, as illustrated in several preceding examples.

GLabels:

Now that you know about these methods, you are finally in a position to understand the details of how the code in the HelloGraphics example centers the label on the canvas. The relevant lines look like this:

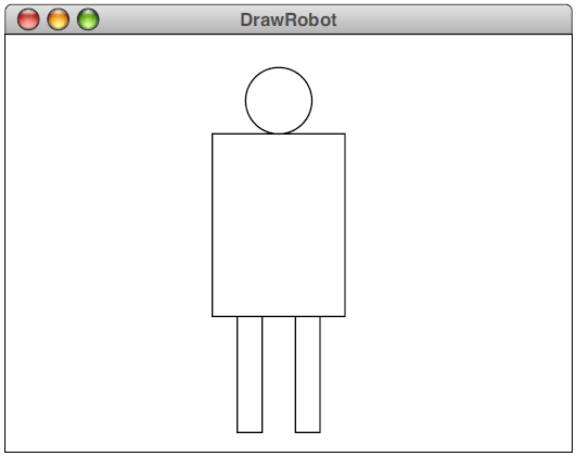
```
GLabel label = new GLabel("hello, world");
double x = (getWidth() - label.getWidth()) / 2;
double y = (getHeight() + label.getAscent()) / 2;
this.add(label, x, y);
```

GPolygon

You can define the diamond GPolygon relative to its center using the following code:

```
GPolygon diamond = new GPolygon();
diamond.addVertex(-30, 0);
diamond.addVertex(0, 40);
diamond.addVertex(30, 0);
```

Write a GraphicsProgram subclass DrawRobot.java that generates the following picture of a robot. Play with the coordinates until you get something that looks more or less correct. Show your work to TA.



Reading Exercise

•Read chapters 1,2 of ACM Java tutorial.

–http://cs.stanford.edu/people/eroberts/jtf/tutorial/Tutorial.pdf

•Try all examples and understand.