Chapter 2
ACM Java Graphics

The ACM Library provides a big toolkit that makes cool programming accessible to the beginner. But there’s a lot to learn. This chapter shows you the most useful tools and techniques, and you’ll need to refer to it as your work progresses. You can find more in the Quick Reference appendix.

2.1 The Graphics Window
ACM Java graphics programs run in a window called the graphics window. Some of the characteristics of this window may be set by the programmer.

2.1.1 Setting the Window Size
The size of the graphics window may be set by giving values to two constants, APPLICATION_WIDTH and APPLICATION_HEIGHT. These constants are predefined as part of the ACM library, but we can give them other values to set the size of the window, measured in pixels, as we prefer. Here’s an example:

```java
public static final int APPLICATION_WIDTH = 800;
pUBLIC static final int APPLICATION_HEIGHT = 500;
```

You can choose your own size of course, but otherwise these statements must be written exactly as shown to work correctly, and placed outside the run() method, as demonstrated below.

**HelloWorldGraphics2**

```java
//HelloWorldGraphics2.java
import acm.program.*;
import acm.graphics.*;

public class HelloWorldGraphics2 extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 700;
    public static final int APPLICATION_HEIGHT = 250;

    public void run() 
    {
        GLabel label1 = new GLabel("Hello world");
        add(label1 , 100, 75);
    }
}
```

©Daniel L. Schuster, 2001, all rights reserved
For now don't worry about the meaning of `public static final`. Consider it to be required boilerplate.

### 2.1.2 Setting the Background Color

The default color of the interior of the graphics window is white, but you can change that with the `setBackground()` method. Here's an example:

```java
setBackground(Color.GREEN);
```

You can use any of the defined colors earlier described, or the custom colors we'll learn to create later.

### 2.1.3 Adding andRemoving Objects

Graphic objects are added to the window with the `add()` method, of which there are two versions.

Here's an example of the first version.

```java
add(rect1); ← This statement adds the `rect1` at its currently set location.
```

And the second version.

```java
add(rect1, 100, 100); ← `rect1` is added at (100, 100)
```

If the location is outside of the graphics window then the object is not visible, but it still exists and can be manipulated or later moved to the window.

A graphic object may be removed from the screen with the `remove()` method.

```java
remove(rect1); ← remove `rect1` from the graphics window
```

It is important to note that removing an object is *not the same* action as making it invisible, which we will discuss later. Further, a removed object still exists, it's just not in the window. You can still manipulate the removed object, perhaps setting its color or other characteristics, and then bring it back to the window later in the program. Like zombies, removed objects can rise from the dead.

*All* of the graphics objects currently in the graphics window may be removed in one step with

```java
removeAll();
```
Note the empty parenthesis, which are required.

2.1.4 Slowing Things Down a Bit
We have already seen how execution can be slowed with the `pause()` method, which freezes execution for the specified time, measured in milliseconds.

`pause(1000);` ← pause for 1 second
`pause(2500);` ← pause for 2½ seconds
`pause(300);` ← pause for one-third of a second

We can also halt execution with

`waitForClick();`

which stops the program until the user clicks on the program window. Execution then resumes. Clicks outside the window are ignored.

2.2 More About Rectangles and Ovals
There are several shapes available to us, but for now let's concentrate on rectangles and ovals for now, as they are often the most useful.

2.2.1 Creating Graphics Objects and Putting Them in the Window
In object-oriented programming languages such as Java the methods used to create objects are called constructors. You must create an object using a constructor before you use it.

In ACM Java, rectangles are known as `GRects`. There are two methods for constructing them.

**Method 1**

`GRect rect1 = new GRect(100, 50);`

This statement makes `rect1` a `GRect` with a width of 100 pixels and a height of 50 pixels. Since we didn't specify a location, it has the default location of (0, 0), meaning that the upper left corner of `rect1` will be at the upper left corner of the window. However, it is not actually in the window yet as we haven't added it.

**Method 2**

`GRect rect2 = new GRect(400, 30, 200, 10);`

This statement makes `rect2` a `GRect` with a width of 200 and a height of 10. It is not on screen yet, but its location is (400, 30).

`GRects` can be put in the graphics window with the `add()` method.
add(rect1); \leftarrow \text{puts rect1 in the window at the current location, in this case (0, 0)}
add(rect1, 230, 300); \leftarrow \text{rect1 in the window at (230, 300)}
add(rect2); \leftarrow \text{rect2 in the window at its current location (400, 300)}
add(rect2, 80, 10); \leftarrow \text{rect2 at (80, 10). Its location is now (80, 10) even though it was originally (400, 30)}

GOvals are created similarly.

GOval headOfFigure = new GOval(70, 80); \leftarrow 70x80, location is (0, 0)
GOval nose = new GOval(41, 20, 18, 15); \leftarrow 18x15, location is (41, 20)

A circular object may be created with the GOval method by setting the width and height to the same value, as shown here.

GOval circle = new GOval(50, 50); \leftarrow 50x50 circle

Adding a GOval is exactly the same as adding a GRect and we refer you to those examples.

Memory allocation
Memory allocation for any object, including ACM graphic objects, is more complicated than it is for primitive data types. The statement

GRect r = new GRect (100, 50);

says "Find enough memory for a GRect, get the address of that memory location, put the address of that memory location in a second memory location, and call that second memory location r.

Conceptually we have

\[
\begin{array}{ccc}
\text{memory location of GRect object} & \text{contents of GRect object} & r \\
\end{array}
\]

For the sake of discussion assume the GRect object is at memory location 10,000. We have

\[
\begin{array}{ccc}
\text{memory location 10,000} & \text{contents of GRect object} & r \\
10,000 & & \\
\end{array}
\]

The important thing to note is that the memory location r doesn't contain the GRect itself, it contains the address of the GRect.
A common way to visually represent the relationship between an object and its location in memory is

\[ r \rightarrow \text{contents of GRect object} \]

and we say "r points to the object" which is a short way of saying "r contains the memory address of the object."

The statement

\[ r\text{.setColor(Color.RED);} \]

which colors \( r \) red, says "apply the \text{setColor(\ )} \) method to the object located at the memory location stored in \( r \)."

### 2.2.2 Setting Characteristics of Rectangles and Ovals

\text{GRects} and \text{GOvals} have several characteristics that can be set after the object is created, and can be changed again later as needed. Use is simple and a few examples should be sufficient.

**Coloring and filling in rectangles and ovals**

The default color of a graphics object is black, but a \text{GRect} or \text{GOval} may be given another color. They also may be outlined or filled in.

\[ \text{rect1.setColor(Color.BLUE);} \leftarrow \text{rect1 will be blue} \]
\[ \text{rect1.setFilled(true);} \leftarrow \text{rect1 will be a filled in object instead of just an outline} \]
\[ \text{rect1.setFilled(false);} \leftarrow \text{rect1 will be an outlined object} \]

Recall that you must use \text{import java.awt.*}; to use the Java colors.

**Moving rectangles and ovals**

A \text{GRect} or \text{GOval} may be moved relative to its current location or by jumping to a specific location in the graphics window.

To move a \text{GRect} or \text{GOval relative to its current position}, use the \text{move(\ )} \) method.

\[ \text{nose.move(10, 5);} \leftarrow \text{nose will be moved 10 pixels right and 5 pixels down from its current location} \]
\[ \text{rect1.move(-8, 15);} \leftarrow \text{rect1 will be moved 8 pixels left and 15 pixels down} \]
\[ \text{nose.move(-5, -9);} \leftarrow \text{nose will be moved 5 pixels left and 9 pixels up} \]

To move a \text{GRect} or \text{GOval} to a specific location, use the \text{setLocation(\ )} \) method.
nose.setLocation(240, 100); ← nose will be moved to (240, 100)
nose.setLocation(-100, 0); ← nose will be moved to (-100, 0), which is off the window. An object that is off the window can still be manipulated however.

Changing the size of a rectangle or oval
The horizontal and vertical size of a GRect or GOval may be changed, either to a specified size or relative to the current size. Either way the upper left corner of the object stays fixed and the rest of the object grows as required by the changed size.

To set the specific size of a GRect or GOval, use the setSize( ) method.

nose.setSize(20, 30); ← set the size of nose to 20x30

To change the size of a GRect or GOval, relative to its current size, use the scale( ) method.

nose.scale(1.2); ← scale nose to 120% of its current size. The nose get larger.
nose.scale(.75); ← scale nose to 75% of its current size. The nose gets smaller.
rect1.scale(3); ← scale rect1 to 300% of its current size, tripling its size.

Invisible objects
You may change the visibility of a rectangle or oval with the setVisible( ) method.

nose.setVisible(true); ← nose is now visible. Remember that you will still not be able to see it if it is covered by something else or is off the window.
nose.setVisible(false); ← nose is invisible. You can't see it, but it can still be manipulated and the program can still react to it. It's just that you can't see it. Tricky, eh!

2.2.3 Manipulating the Stacking Order
If you have more than one rectangle or oval object in the window, one of them may lie partially or totally on top of the other. The order of objects that lie on top of each other is called the stacking order and can be manipulated.

Let’s begin with the assumption that we have three rectangles r1, r2, and r3, that they appear in the graphics window as shown below and that r3 is partly on top of r1 which is partly on top of r2.

©Daniel L. Schuster, 20011, all rights reserved
Now consider this series of manipulations of the stacking order.

\( r_2.\text{sendForward}() \); \( \leftarrow \) send \( r_2 \) forward one layer in the stacking order

\( r_1.\text{sendToFront}() \); \( \leftarrow \) send \( r_1 \) all the way forward to the top of the stacking order

\( r_3.\text{sendBackward}() \); \( \leftarrow \) send \( r_3 \) back one layer in the stacking order

\( r_1.\text{sendToBack}() \); \( \leftarrow \) send \( r_1 \) all the way backward to the bottom of the stacking order

### 2.2.3 Getting the Coordinates of a GRect or GOval

The location of a GRect or GOval, defined as the coordinates of the upper left corner of the object, may be retrieved using the \( \text{getX}() \) and \( \text{getY}() \) methods. These methods ‘reach into’ the object they are applied to, get the requested information and then return it for further use.
double \( xLoc = rect1.getX() \); \( \leftarrow xLoc \) now has the value of \( rect1 \)'s x coordinate
double \( yLoc = rect1.getY() \); \( \leftarrow yLoc \) now has the value of \( rect1 \)'s y coordinate

These methods will be very useful when we start animating objects and need to know where
they are in the graphics window.

\textbf{int vs. double coordinates}

Notice that the \( \texttt{getX()} \) and \( \texttt{getY()} \) methods return \texttt{double} values, even though all of our
examples use \texttt{int}s for setting coordinates. In fact, though we have treated coordinates in the
window as integers, the ACM library works with them as real numbers. When we get them with
the \( \texttt{getX()} \) or \( \texttt{getY()} \) methods, we are getting \texttt{doubles} and so must store them as \texttt{doubles} as
shown above.

However, typically we want to work with window coordinates as \texttt{int}s, and so use a \texttt{type cast}
to convert the \texttt{double} via truncation to an \texttt{int}.

\texttt{int }\texttt{xLoc }= \texttt{(int) rect1.getX();}

\subsection*{2.3 An Example Program}

Now we know how to set up the window, to make rectangles and oval objects, and to change
those objects. In this example we put this knowledge to work, basically fooling around with
objects in the graphics window.

\begin{verbatim}
BasicGraphicsDemo
://BasicGraphicsDemo.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
public class BasicGraphicsDemo extends GraphicsProgram {
    //BasicGraphicsDemo.java
    public static final int APPLICATION_WIDTH = 600;
    public static final int APPLICATION_HEIGHT = 500;

    final int ONE_SECOND = 1000;
    final int BALL_SIZE = 50;
    final int RECT_WIDTH = 50;
    final int RECT_HEIGHT = 10;

    public void run( )
    {
        setBackground(Color.YELLOW);
        //make the objects and put them on screen
        GOval ball = new GOval(BALL_SIZE, BALL_SIZE);
    }
}
\end{verbatim}
Now let’s dissect the program.

import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class BasicGraphicsDemo extends GraphicsProgram {

    public static final int APPLICATION_WIDTH = 600;
    public static final int APPLICATION_HEIGHT = 500;

    final int ONE_SECOND = 1000;
    final int BALL_SIZE = 50;
    final int RECT_WIDTH = 50;
    final int RECT_HEIGHT = 10;

    public void run() {
        setBackground(Color.YELLOW);

        //make the objects and put them on screen
        GOval ball = new GOval(BALL_SIZE, BALL_SIZE);

        ball.setColor(Color.RED);
        ball.setFilled(true);
        add(ball);

        GRect rect1 = new GRect(RECT_WIDTH, RECT_HEIGHT);
        rect1.setColor(Color.GRAY);
        add(rect1, APPLICATION_WIDTH-RECT_WIDTH, 0);

        waitForClick();

        //manipulate the objects
        ball.move(0, 100);
        pause(ONE_SECOND);
        rect1.setVisible(false);
        rect1.setLocation(APPLICATION_WIDTH/2-RECT_WIDTH/2,
                          APPLICATION_HEIGHT/2-RECT_HEIGHT/2);
        pause(ONE_SECOND);
        rect1.setVisible(true);
    }
} //BasicGraphicsDemo
Let's take a look at a simple animation of a solid green circle that starts at the upper left corner and moves down and to the right.

**MoveGreenBall1**

```java
//MoveGreenBall1
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

class MoveGreenBall1 extends GraphicsProgram {
    public static final int APPLICATION_WIDTH = 600;
    //manipulate the objects
    ball.move(0, 100);
    pause(ONE_SECOND);
    rect1.setVisible(false);
    rect1.setLocation(APPLICATION_WIDTH/2-RECT_WIDTH/2,
                      APPLICATION_HEIGHT/2-RECT_HEIGHT/2);
    pause(ONE_SECOND);
    rect1.setVisible(true);
}
```
The animation comes from the `while(true)` loop, which endlessly repeats these steps:

1) move the ball
2) wait a little bit

The result is that the ball moves right and down, eventually moving out of the window. Certainly nothing fancy, but it's a start and all of our animations and game programs will have a similar structure. We call the idea of cutting the behavior of an object, in this case movement, into discrete pieces time slicing. Time slicing will be key to our game playing programs.

Here's another animation which simulates movement of a foreground object by moving the background underneath it. Many very simple computer games, often called scrollers, use this basic technique. 'Helicopter in a Cave' is a classic example of a scroller game.

**RollGreenBall**

```java
//RollGreenBall.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
public class RollGreenBall extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 600;
    public static final int APPLICATION_HEIGHT = 500;
    public void run( )
    {
        setBackground(Color.BLACK);
        G Oval ball = new G Oval(50, 50);
        ball. set Color(Color.GREEN);
        ball. set Filled(true);
        add(ball);
        waitForClick( );
        while (true)
        {
            ball.move(1, 1);
            pause(50);
        } //move the ball
    } //run
} //MoveGreenBall
```
Finally, let’s look at another animation example. This program displays lights that blink, much as you might see on a Christmas tree.

**Blinkers**

```java
//Blinkers.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class Blinkers extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 300;
    public static final int APPLICATION_HEIGHT = 200;

    public void run()
    {
        final int WAIT = 400;

        setBackground(Color.BLACK);
        GOval light1 = new GOval(100, 100, 20, 20);
        GOval light2 = new GOval(100, 150, 20, 20);

        setBackground(Color.LIGHT_GRAY);
        light1.setFilled(true);
        light1.setColor(Color.BLACK);
        setBackground(Color.WHITE);
        light2.setFilled(true);
        light2.setColor(Color.BLACK);

        while (true)
        {
            light1.move(-1, 0);
            light2.move(-1, 0);
            pause(WAIT);
        }
    }
}
```

create two buildings that are off the window
create the green ball on the window
move the buildings to the left, but don’t move the green ball. The result is a ball that seems to move with respect to the buildings.
light1.setFilled(true);
l1ight1.setColor(Color.BLUE);

GOval light2 = new GOval(200, 40, 20, 20);
l1ight2.setFilled(true);
l1ight2.setColor(Color.RED);

add(light1);
pause(WAIT);
add(light2);
pause(WAIT);

while(true) {
    remove(light1);
pause(WAIT);
    add(light1);
pause(WAIT);
    remove(light2);
pause(WAIT);
    add(light2);
pause(WAIT);
} // blink the lights

start the lights blinking

The important thing about the Blinkers example is the way we’ve interlaced the visual behaviors of the two lights. Basically, the algorithm is:

light 1 on
light 2 on
loop:
light 1 off
light 1 on
light 2 off
light 2 on

By turning on one light before we turn off the other, we ensure that the light displays overlap, creating a decent blinker effect. Interlacing behaviors of objects is a key idea for animating multiple objects in a game.

### 2.5 More About Labels

Labels provide us with the ability to put text such as game instructions or scores on the screen.
Constructing labels
Labels are created with the GLabel() constructor.

GLabel label1 = new GLabel("Hello world");

This creates a label with the default baseline location of (0, 0), in the default font (the font is the shape of the letters) and the default size of the letters. Note that if we add the label at the default location very little of the text will actually show due to the location of the base line

Glabel label2 = new GLabel("Hello world", 50, 80);

This creates a label with the location of (50, 80), default font and default size

Putting labels in the window
Labels are added to the window with the add() method as described for other graphics objects, except that the location is actually the location of the lower left corner of the first letter on the baseline as described earlier.

add(label1, 460, 500); ← put label1 at (460, 500) in the window
add(label2); ← put label2 at its location of (50, 80)

Changing the text of a label
The text of the label may be set when the label is created with the GLabel constructor and later changed with the setLabel() method, as in

label1.setLabel("Bye bye");

Setting the general shape, style and size of the characters in a label
The general shape of the characters is called the font family. We consider only the font families known as Serif, where the letters have decorative ‘tails’ and SansSerif, where the letters do not have decorative tails.

Serif – EXAMPLE of Serif FONT
SansSerif – EXAMPLE of SansSerif FONT

Additionally, characters may have a style. Our style possibilities are plain, bold, italic, and a combination of bold and italic. The styles are specified with the Java constants PLAIN, BOLD, ITALIC and BOLDITALIC.

Finally, the size of characters may be set by specifying a positive whole number, with larger numbers representing larger text. Font size is measured in points.
The shape, style and size of the characters in a label are set with the `setFont( )` method, which takes as an argument a **String** representing the font family, the style and the size.

```java
label1.setFont("Serif-PLAIN-100"); /* Serif font family, plain style and 100 point size */
label1.setFont("SansSerif-BOLD-50"); /* SansSerif font family, bold, 50 point size */
label1.setFont("*-ITALIC-60"); /* current font family (whatever it is), italic, 60 point */
label1.setFont("Serif-*-80"); /* Serif font family, current style, 80 point */
label1.setFont("*-BOLDITALIC-*"); /* current font family, bold and italic, current size */
```

**Other label manipulations**
The following methods all work with labels in the appropriate manner .

- `move( )`
- `sendForward( )`
- `getX( )`
- `setLocation( )`
- `sendBackward( )`
- `getY( )`
- `setVisible( )`
- `sendToFront( )`
- `setColor( )`
- `sendToBack( )`

**Things you can’t do with labels**
You can’t set the size of a label with `setSize( )` or `scale( )` or fill a label with `setFilled( )`.

### 2.6 Working with Lines

The ACM library provides several tools for working with straight lines.

**Constructing lines**
Lines are constructed with `GLine( )`, which creates a line of approximately 1 pixel thickness by specifying the starting and ending coordinates.

```java
GLine l1 = new GLine(10, 10, 100, 20);
```

This statement creates a line from the start point at (10, 10) to the end point at (100, 20).

Note that *almost* the same line may be created with

```java
GLine l2 = new GLine(100, 20, 10, 10);
```

but this sets the start point at (100, 20) and the end point at (10, 10). This makes an important difference when use the `setLocation( )` method as described below.

**Moving lines**
Lines may be moved with `move( )` or `setLocation( )`. Both move the start point, and the rest of the line is moved so that the length and orientation of the line are preserved.

There is one subtle point to consider when using `setLocation( )`. Remember that `l1` as created above goes from (10, 10) to (100, 20). `l1.setLocation(0, 0);` moves the *start point* (10, 10) to (0, 0). The *end point* moves to (90, 10). We would have
Recall $l_2$, a line from $(100, 20)$ to $(10, 10)$. $l_2\text{.}\text{setLocation}(0,0)$; moves the start point $(100, 20)$ to $(0, 0)$, which moves the rest of the line off the window. We would have

The important thing to remember is that the \texttt{setLocation()} method moves the start point, and the rest of the line moves with it.

\textbf{Setting the endpoints}

The start and end points of a line may be individually changed with \texttt{setStartPoint()} and \texttt{setEndPoint()}. 

$l_1\text{.}\text{setStartPoint}(80, 40)$; \leftarrow sets the start of \textit{l1} to $(80, 40)$ and the end point remains at $(100, 20)$.

$l_1\text{.}\text{setEndPoint}(400, 420)$; \leftarrow sets the end point at $(400, 420)$, and now \textit{l1} is a line from $(80, 40)$ to $(400, 420)$

\textbf{Other line manipulations}

The following methods all work with lines in the obvious manner.

\begin{verbatim}
setColor() sendForward() getX() \leftarrow return x coordinate of start
setVisible() sendBackward() getY() \leftarrow return y coordinate of start
sendToFront() sendToBack() \end{verbatim}

A line also may be scaled with the \texttt{scale()} method, which keeps the starting point in position and changes the length of the line by the scale factor, preserving the orientation.
Things that can’t be done with lines
You can’t resize a line with the \texttt{setSize()} method, and you can’t fill a line using the \texttt{setFilled()} method.

### 2.7 Methods for Graphics Images

*Existing* jpg or gif images, such as found on the web or drawn with a painting program, may be used in an ACM graphics program. In the ACM library images are known as \texttt{GImages}.

We begin with the assumption that the graphics files airplane.jpg and boat.gif exist \textit{in the same folder} as the Java program being written. \textit{This assumption is important!}

**Constructing an image for use in our program**
Existing images may be brought into the program with the \texttt{GImage()} constructor and added with the \texttt{add()} method.

\begin{verbatim}
GImage img1 = new GImage("airplane.jpg");
add(img1, 100, 200);
GImage img2 = new GImage("boat.gif", 240, 30);
add(img2);
\end{verbatim}

**Other image manipulations**
The following methods all work with graphics images in the appropriate manner.

\begin{table}
\begin{tabular}{llll}
move() & sendForward() & getX() & \\
setLocation() & sendBackward() & getY() & \\
setVisible() & sendToFront() & & \\
scale() & sendToBack() & & \\
\end{tabular}
\end{table}

Note that scaling an image so that it is larger will cause pixilation of the image, where the image appears ‘blocky’ as if the individual pixels have gotten bigger, and in fact that’s basically what happened.

**Things you can’t do with images**
You can’t use \texttt{setFill()} or \texttt{setColor()} with an image.

### 2.8 More About Graphics Objects
You may have noticed that the graphics objects have a lot in common. For example, they all have the methods

\begin{verbatim}
move() & getX() & \\
setLocation() & getY() & \\
setVisible() & & \\
\end{verbatim}
Why do all of the objects share so much in common? The answer is that by design all of these objects share many characteristics – they all can be placed on the screen in various locations, they can be hidden, they can be ordered front to back and so on.

So the ACM library designers created a general class GObject, and specified that GObject must have these methods defined. But the actual implementations, the code for the particular objects, are left to the specific classes such as G Oval or G Rect that are based on the GObject class. GObject is called a parent or base class, and G Oval and the other graphics object classes are children or subclasses of the GObject class. In Java-speak we say G Oval extends GObject, and of course the other graphics object classes also extend GObject.

When a class such as GObject specifies a set of methods but doesn’t implement them it is called an abstract class.

On the other hand, some methods don’t really apply to every object. For example, the size of a label is specified with point values in the setFont( ) method, not by setting the size with setSize( ) or scale( ). Similarly, setSize( ) doesn’t really make sense for lines, and setFilled( ) certainly doesn’t make sense for lines. setFilled( ) and setColor( ) can’t be applied to images, which already have these characteristics defined.

Since some methods are not appropriate for all graphical objects, they are left out of the GObject class, which describes only the common features of the objects. However, we do want these methods to be implemented where appropriate. In the ACM library this is handled with interfaces. Implementing an interface is sort of a very weak form of extending an object.

2.9 But Wait! There’s More

In the interest of brevity and clarity, we have covered only a small fraction of the methods available for graphics objects. There are many additional methods and objects. A web search of “ACM GObject” will quickly take you to pages of information, probably more than you want to know.

For now, we’ve covered almost all of the key graphics methods for writing simple games. We’ll see a bit more in Chapter 6 Game Programming I, where we’ll put these objects and methods to work, creating simple but real and playable graphics games.
You can also learn more by reading the Quick Reference appendix.

It might be interesting to note that there are additional subclasses of the GObject class including:

- **GCompound** – for creating more complex graphics objects with more than one component, such as this UFO with the bug-eyed alien pilot, pulling a sign across the sky.

  ![Clearly art is not my strong point](image)

  We’ll use GCompounds a lot later in our work.

- **Garc** – for creating curves.
- **GPen** – simulates drawing with a pen in the window.
- **GPolygon** – creating objects with many sides.
- **GTurtle** – a ‘turtle’ that can move around with simple commands.

And of course there are methods galore.

**Objective questions**

1. **TRUE/FALSE** The statement `final int APPLICATION_WIDTH = 500;` sets the width of the window to 500 pixels.
2. Write the statement to set the height of the window to 700 pixels.
3. **TRUE/FALSE** `setBackGround(Color.BLUE)` colors the window to blue.
4. Pick the statement that will remove `blob1` from the graphics window.
   a. `blob1.remove();`
   b. `blob1.remove();`
   c. `window.remove(blob1);`
   d. `remove(blob1);`
   e. none of these
5. **TRUE/FALSE** `pause(2.5);` freezes the program for 2½ seconds.
6. Write the statement to add `blob1` to the window at the location (250, 300).
7. Assume `blob1` is an object that measures 10x10 pixels. Write a statement that puts `blob1` just barely off the left side of the window.
8. **TRUE/FALSE** Assume `blob1` is in the window but not visible. Write the statement to make `blob1` visible again.
9. Write the statement to retrieve the x coordinate of `blob1` and store it in the variable `xBlob`.
10. **TRUE/FALSE** PLAINBOLD is not a valid style for fonts.
11. Assume `sign` is a valid label. Write the statement that will change the points of the font to 30, without changing any other characteristics.
12. Assume $o_1, o_2, o_3$ and $o_4$ are graphics objects, and that the top to bottom order is $o_1, o_2, o_3, o_4$. Write the statements to change the top to bottom order to $o_4, o_3, o_1, o_2$.

13. TRUE/FALSE Assume `sign` is a valid label. The statement `sign.scale(2)` doubles the font size of the label.

14. To change the end point of a line $l_1$ to the bottom left corner of the line, use
   a. $l_1.setEndPoint(0, 0)$;
   b. $l_1.setEndPoint(0, APPLICATION_HEIGHT)$;
   c. $l_1.setEndPoint(0, APPLICATION_HEIGHT - 1)$;
   d. none of these. The correct answer is ________________________________

**Short Answer Problems**

1. Assume `blob1` is in the window and has been set to be visible, yet `blob1` does not show on the screen. There are two possible causes for this. What are they?

2. Write the statements to create a label “The Dark Knight” in a large, serifed, bold and italicized font, then place in on the screen.

3. Write the statements to create a 100x85 blue rectangles, add it to the window at (30, 33) and make it invisible.

4. Write the statement to create a solid green circle of radius 40, but don't put it in the window.

5. Assume `b1` is a GOval that is already added to the window. Write the statements to perform each task.
   a. set its color to yellow and make it be filled, pause, then make it be outlined again
   b. change its size to 80x50
   c. double its size using the scale method
   d. put it in front of anything that might be covering it
   e. display its x and y coordinates in a label
   f. put `b1` in the upper left corner of the window and move it straight down
   g. put `b1` in the upper right corner of the window and move it straight left

6. Assume `bigLine` is a line in the window. Write the statements to move its start point exactly 1/2 halfway towards the upper left corner. This moves the entire line of course.

7. Assume "pinkPig.jp" is an image. Write the statements to bring pinkPig into your program, triple its size and place it in the window.

**Graphics Program Problems**

1) **BigAndSmall** Create and display a circle that jumps around the screen several times, grows in several steps and then shrinks down to nothing. Slow down the action enough that it is visible.

2) **StarryNight** Create a star field of white, yellow, blue and red stars, complete with a flickering effect, against a black background.

3) **FlashingUFO** Using a loop, create a UFO that flies across the window. Add lights that flash.

4) **DrawStar** Use GLines to draw a five pointed star.
5) **DropAndRoll** Put a ball on top of a block, then roll the ball off the block, falling to the bottom of the window, and then continue to roll in the same direction till it’s off the window. Here’s some snapshots of the action.

![DropAndRoll](image)

Note that you can do this without loops. Just use quite a few move statements.

6) **FlyUFO1** Create a UFO, then using a loop fly it across and then off the window.

![FlyUFO1](image)

7) **FlashingUFO1** Add flashing lights to the UFO in FlyUFO1.

8) **BigAndSmall** Place a rectangle in the approximate center of the window. Make the rectangle grow bigger and then smaller. The change in scale must be from the center, not from the upper left corner. To do this, you need to change the location of the rectangle each time you grow or shrink it.

9) **FlightAtNight** Airplanes at night can only be seen by the blinking lights on their nose, wing tips and tail. Write a program that simulates the view of such an airplane from ground level as it flies directly overhead. The nose light is red, the wing tips are blue and the tail light is white. The nose light is on and off for ¼ second at a time, the wing tip lights on and off for ½ a second at a time, and the tail light is on and off for 1 second at a time.

10) **ChristmasTree** Use GLines and GOvals to create a Christmas tree, complete with flashing lights.

11) **SlideShow** Write a program that creates a simple slide show, displaying 4 images (jpeg or gifs), each showing for 3 seconds on screen before the next appears. The slide show repeats endlessly.

12) **SlideShow2** Repeat SlideShow but now the image advanced only occurs after the user clicks in the window.

13) **StarWars** Create an opening scene for the classic movie Star Wars, where the words

    A Long Time Ago

    **In A Galaxy Far, Far Away**

    **Star Wars**

©Daniel L. Schuster, 20011, all rights reserved
scroll up, but appear to move farther way, getting small and smaller. The words should remain horizontally centered in the window.