**Chapter XX**

**Advanced Graphics**

**Chapter XX Topics**

20.1 Introduction

20.2 Drawing Individual Pixels

20.3 Using the Font class

20.4 Mathematics and Polygons

20.5 Using Mouse Routines with Graphics

20.6 Understanding Graphics Displays

20.7 Virtual Memory & Video Buffering

20.8 Buffering Program Examples

20.9 The Magic of Method update

20.10 Importing Images

20.11 Multi-Page Output

20.12 Program Input/Output with GUI Windows

20.13 Summary

**20.1 Introduction**

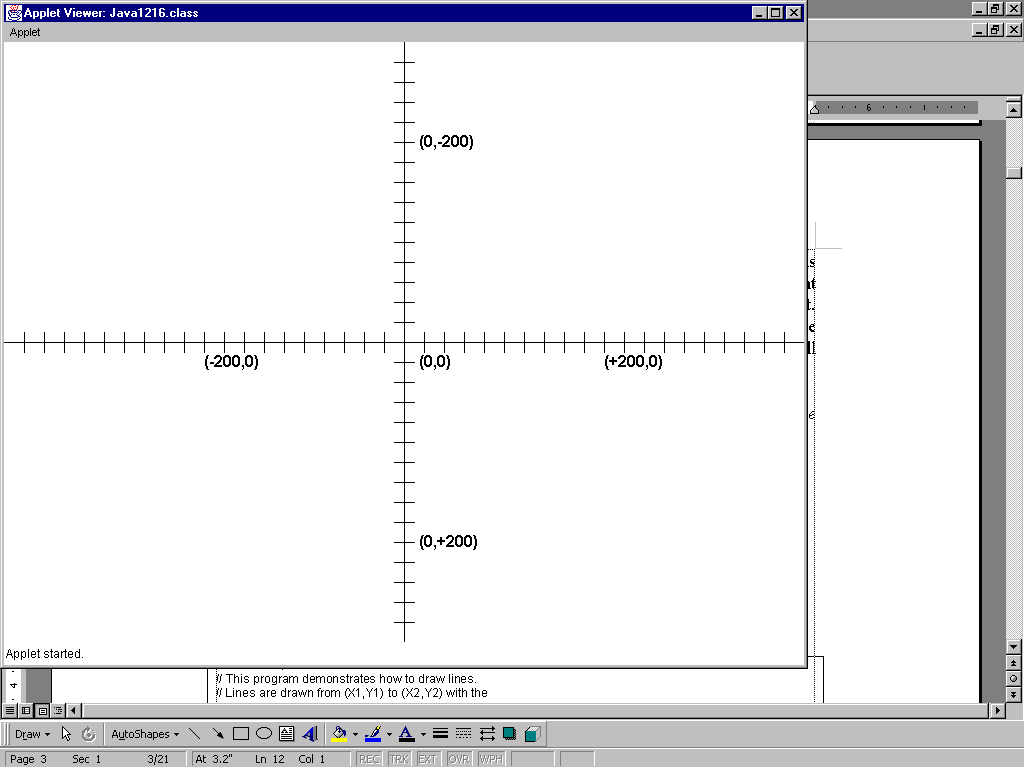
The primary purpose of Exposure Java is to support students in a college-prep or AP Computer Science course. A secondary purpose is to learn Java, the current language used on the AP Computer Science Examination. The College Board Test Development Committee has decided not to test students on input or output topics with the Java programming language. There are many different ways to handle input/output. You have already seen that "input" can be as tedious as hard coding variable values in a program. It can be a little better by entering values at the command line prompt. Input can also be interactively done entered as a text string or entered in some GUI window. This list is not even complete since there are many more ways that Java allows data to be entered into a program. You have also seen that output can be done rather simply with text output or in a more complex manner in a Windows environment. It is precisely because of the large variety of input/output options that the Test Development Committee decided not to ask specific questions in this area. You will therefore see some warning boxes from time to time, as the one below, which indicates that the current topic is optional. It is optional in the sense that you will not be asked questions on the AP Computer Science Examination, but many optional topics are very important nevertheless.

|  |
| --- |
| **AP Computer Science Examination Warning** |
| This is an optional chapter in the AP Computer Science curriculum. The graphics concepts in this chapter and future input/output chapters will not be tested on the AP Computer Science Examination. |

So just why is there a need to do topics that are optional? A simple answer is that computer science students like doing graphics. I do believe that you will like the lab assignments at the conclusion of this chapter better than the interest programs of an earlier chapter. A more accurate answer is that graphics is an excellent tool to learn and practice computer science concepts. In this chapter you will learn a large variety of Java graphics methods. Each one of these methods requires the precise use of parameters. The display of many graphics images involves the use of many control structures and a solid knowledge of Boolean logic. Graphics provides instant feedback. If you handle the parameters incorrectly, if you use wrong logic in your control structures conditions, the output will be bizarre. At the conclusion of this chapter you might be drawing a star and the US flag, or you may create your own paint program. Any mistake in your program will stare at you in the form of some odd-looking star or stranger-looking flag. The interest program did not provide a real convenient mechanism to instantly let you know that your program logic is flawed.

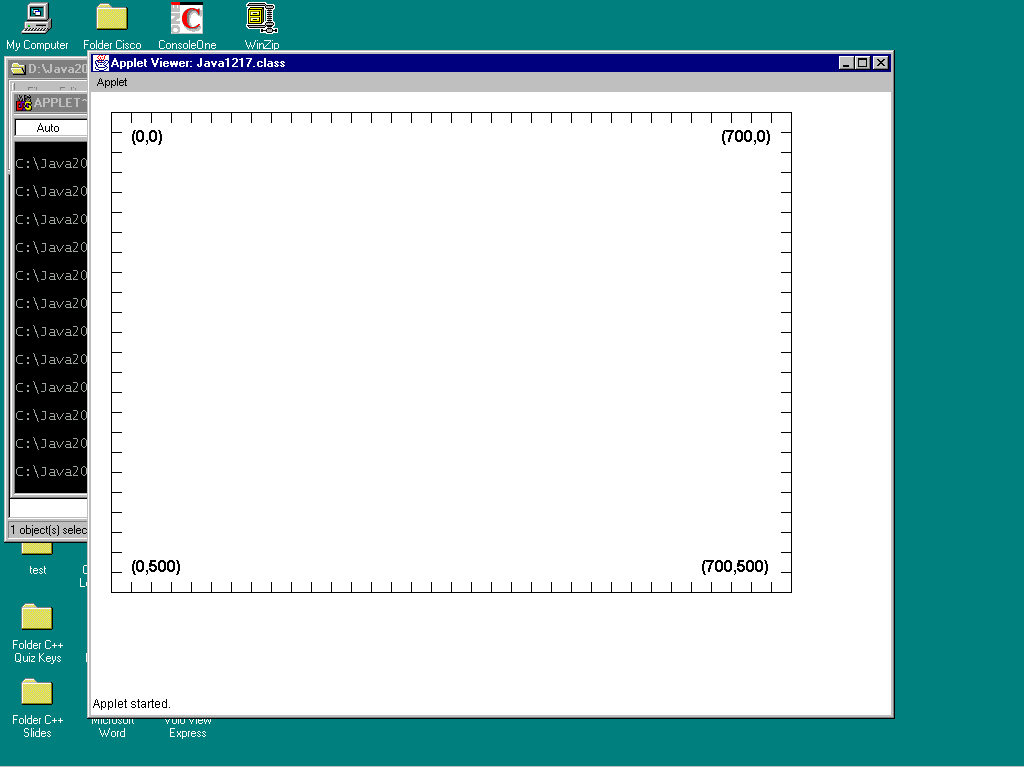
The foundation of Java graphics starts with a package called **awt**, which means *Abstract Windowing Toolkit*. This package provides many graphical features that can be displayed in a GUI (Graphics User Interface) Windows environment. Recently Java has improved its graphics capabilities with a new, and more advanced graphics package, called Java Swing. The advanced **Swing** graphics features will not be addressed in this chapter.

All graphics output is basically on a coordinate system that resembles the *Coordinate Geometry* you learned in many mathematics courses, yet it is also different. Recall that in your mathematics classes you worked with a *Cartesian* system. In this system, shown in figure 20.1, the coordinate plane is divided into four quadrants by two intersecting lines. The intersection of these lines, called the X-Axis and the Y-Axis, is the origin coordinate **(0,0)**. Extending from the origin, **X** values increase to the right and decrease to the left. **Y** values increase downward and decrease upward. Any point on the two-dimensional Cartesian plane can be defined with a coordinate of an X value and a Y value.

**Figure 20.1**

There is great similarity between the Cartesian coordinate system and the computer model. Like the mathematical model, the computer monitor also represents a two-dimensional plane of (X,Y) coordinates. Each coordinate on the monitor is called a **Pixel**. The size of a pixel determines the resolution of the graphics displays. The more - and smaller - pixels, the sharper the graphics image becomes. This means that graphics screens have settings that are specified as a resolution of (**640 X 480)**, (**800 X 600)**, **(1024 X 768)** and so on. The resolution of a monitor depends on the type of monitor, video card and the settings of a particular computer. Most of the program examples in this chapter require a minimum resolution of **(800 X 600)** and one example does require the higher **(1024 X 768)** resolution.

Mathematics and computers graphics may have similarities, but there is a significant difference. The computer graphics screen does not have the **(0,0)** origin at the center of the screen. The **(0,0)** coordinate is at the top-left corner of the monitor. Like the Cartesian system, X values increase to the right and decrease to the left. However Y values on a computer screen increase downwards and decrease upwards. This is opposite of the Cartesian system. Figure 20.2 shows a computer graphics screen with **(700 X 500)** resolution.

**Figure 20.2**

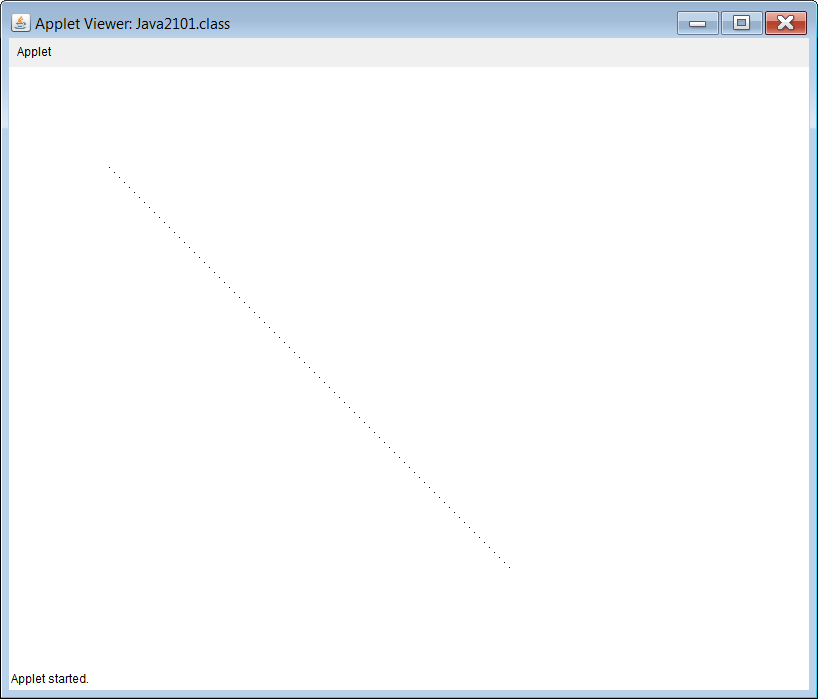
**20.2 Drawing Individual Pixels**

Java has an incredible quantity of graphics methods. We are only looking in one little area of the available graphics features. Somewhere along the line it must have seemed that nobody would want to draw a skinny, single pixel. After all, there are methods for every conceivable shape you want to draw. The reality is that it is important to draw single pixels and it is not too difficult. Pure, single pixels can be drawn using the **drawLine** method using identical starting and ending coordinates. Larger pixels can be drawn with the **fillRect** method. Program **Java2001.java**, in figure 20.3, displays a dotted line of pixels.

**Figure 20.3**

|  |
| --- |
| // Java2001.java  // This program draws a series of pixels with the <drawLine> method.  // This is accomplished by using the same start and end coordinates.  import java.awt.\*;  public class Java2001 extends java.applet.Applet  {  public void paint(Graphics g)  {  for (int x=100, y=100; x <= 500; x+=5, y+=5)  g.drawLine(x,y,x,y);  }  } |

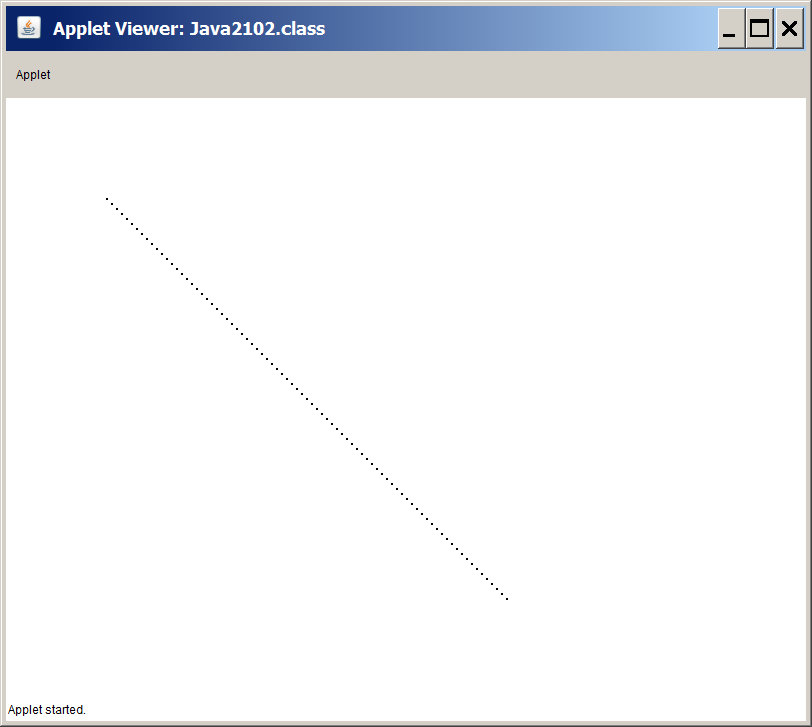
.



The pixels in the previous program are difficult to see. It may be desirable to draw "fatter" pixels like program **Java2002.java**, in figure 20.4. In this example the **fillRect** method is used to draw a pixel that is actually the size of four pixels.

**Figure 20.4**

|  |
| --- |
| // Java2002.java  // This program draws a series of pixels with the <fillRect> method.  // This approach may be better if large pixels are necessary.  import java.awt.\*;  public class Java2002 extends java.applet.Applet  {  public void paint(Graphics g)  {  for (int x=100, y=100; x <= 500; x+=5, y+=5)  g.fillRect(x,y,2,2);  }  } |



**20.3 Using the Font class**

The **drawString** method of the **Graphics** class was briefly introduced in an earlier chapter. The earlier introduction only showed that graphics characters could change color with the **setColor** method. The font style was fixed as Arial and only a small default font was available. There certainly are many more options available when graphics characters are displayed.

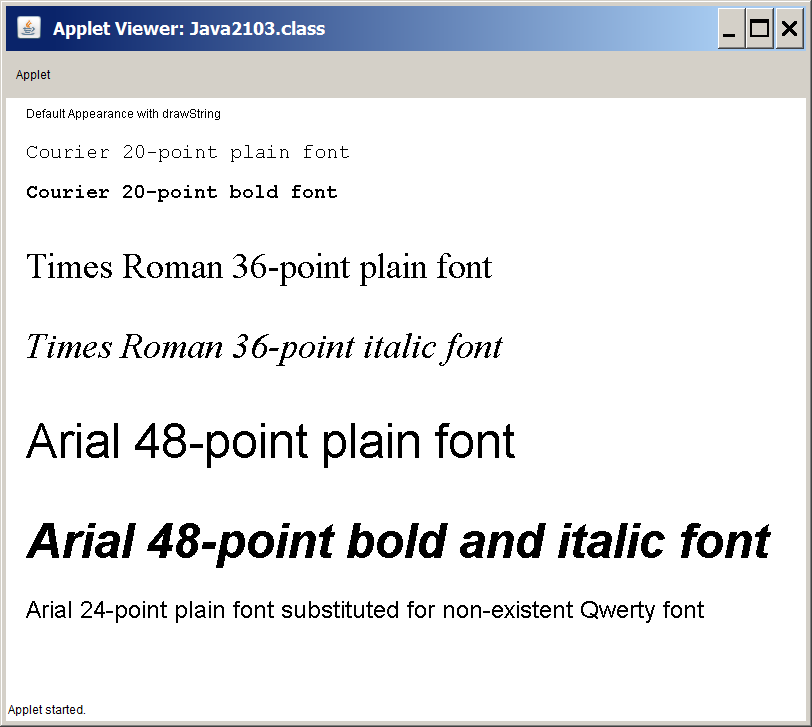
Method **drawString** will display the default Arial font with font size **12**. There is a **setFont** method, which alters the appearance of the default font style and size. Program **Java2003.java**, in figure 20.5, demonstrates four different character strings. Each string has a different font style and size. Additionally, some strings are displayed in **bold**, some are in *italic* and others are plain.

Method **setFont** is a method of the **Graphics** class, but it requires a parameter, which is an object of the **Font** class. Objects of the **Font** class are instantiated with information for *Font Style*, *Font Appearance* and *Font size*. Program **Java2003.java** displays four different styles, which are Courier, TimesRoman, Arial and Qwerty. My guess us that you are not familiar with the **Qwerty** font. The Qwerty font is used intentionally, because it does not exist. This gives us the opportunity to see how Java reacts when an unknown font style is requested. Will such a program even compile? Will it display garbage? Let's see.

**Figure 20.5**

|  |
| --- |
| // Java2003.java  // This program introduces the <setFont(new Font(Type,Style,Size))> method.  // Type is either "Courier","TimesRoman", "Arial", or any other available font.  // Style is either BOLD, ITALIC or PLAIN.  // Size is the point value of the font.  import java.awt.\*;  public class Java2003 extends java.applet.Applet  {  public void paint(Graphics g)  {  g.drawString("Default Appearance with drawString",20,20);    g.setFont(new Font("Courier",Font.PLAIN,20));  g.drawString("Courier 20-point plain font",20,60);  g.setFont(new Font("Courier",Font.BOLD,20));  g.drawString("Courier 20-point bold font",20,100);    g.setFont(new Font("TimesRoman",Font.PLAIN,36));  g.drawString("Times Roman 36-point plain font",20,180);  g.setFont(new Font("TimesRoman",Font.ITALIC,36));  g.drawString("Times Roman 36-point italic font",20,260);    g.setFont(new Font("Arial",Font.PLAIN,48));  g.drawString("Arial 48-point plain font",20,360);    g.setFont(new Font("Arial", Font.BOLD+Font.ITALIC, 48));  g.drawString("Arial 48-point bold and italic font",20,460);    g.setFont(new Font("Qwerty",Font.PLAIN,24));  g.drawString("Arial 24-point plain font substituted for non-existent Qwerty font",20,520);  }  } |

**Figure 20.5 Continued**



The three known fonts are handled nicely by Java. You see Courier, TimesRoman and Arial fonts displayed in **bold**, *italic* and plain with the requested size. The **Qwerty** font is another matter. Java is actually very civilized about the nonsense font. It does display the font in the requested 24-point size and then calmly uses the default Arial style.

It is possible to display a very large quantity of fonts with a provided file of font styles. The mechanics of using a font file will not be shown here. My experience is that with Courier, TimesRoman and Arial you can do quite a good job with your graphics program

|  |
| --- |
| **setFont Method Class: Graphics** |
| g.setFont(new Font("Courier",Font.BOLD,20));  alters the default font to Courier, Bold and size 20.  An object of the **Font** class is used to change the font. |

**20.4 Mathematics and Polygons**

If you do not like mathematics prepare to suffer. Suffering is alright since it builds character. You will thank me when you are older. You may have noticed that so far we have drawn lovely rectangles, squares, ovals and circles. They were all regular. You did not draw some odd-looking quadrilateral. Now you are drawing polygons and Java cares little about the appearance of your polygons. The polygons may be regular or weird, Java just connects the points. In this case it is easy to draw some irregular polygon. Now what happens if you want a regular pentagon, hexagon or other regular polygon? Java has no simple solution to accomplish such a feat. We can still achieve nice regular polygons with the help of our friend mathematics.

I have somewhat of dilemma here. A true understanding of what is to come involves understanding trigonometric or circular functions. Frankly, I hate to take such a side-tour and I believe it is possible to use these functions in a manner that makes it possible to get benefits without too much math, but some math will be necessary.

In the **Math** class is a **sin** method and a **cos** method. Both these methods require some *radian*  value as parameter. This is terrific because radians was always your favorite math topic. I am sure you know that a circle has 360 degrees. It is possible to travel around the circle in a different manner by using radians. You will be pleased to know that there are **2PI** radians to go all around the circle. This means that there are roughly **6.28** radians in a complete circle.

Program example **Java2004.java**, in figure 20.6, draws a circle. Notice the loop structure, which uses **radian** as a loop control variable. Radian values start at **0** and continue until **2\*PI**. This means that it hits all the Radian values on a complete circle. The increment is **0.01** to make the circle more accurate.

That is a lovely loop, but how does it create a circle? It turns out that **sin** and **cos** functions travel through a *unit circle*with values ranging from **-1** to **+1**. Now think of the cos value as providing an **X-coordinate** and the sin value as providing a **Y-coordinate**. In other words, as you go through the "two times PI radians loop" you end up generating a set of coordinates that draws a circle with radius **1**. Such a small circle is not very practical, but we can increase the size of the circle by multiplying the values times the **radius**. Suppose that a certain **radian** value returns **0.05**, when this value is multiplied times **100** you get **5**.

Now do keep in mind that we can only draw integer values. This means that after the sin and cos values are computed, and multiplied times the radius, they must be rounded to the nearest integer to be practical for a graphics program. Java has a **floor** method and a **ceil** method, which are not actually rounding functions, but it also has **Math.round**, which does the required job very nicely.

Well we are almost home free, but something else needs to be addressed. Circular functions travel around the unit circle, which creates a circle around the origin. In mathematics this means origin as in **(0,0)** on a Cartesian coordinate system. Simple enough perhaps, except you are working with a graphics coordinate system where the **(0,0)** coordinate is located at the left-top corner of the monitor.

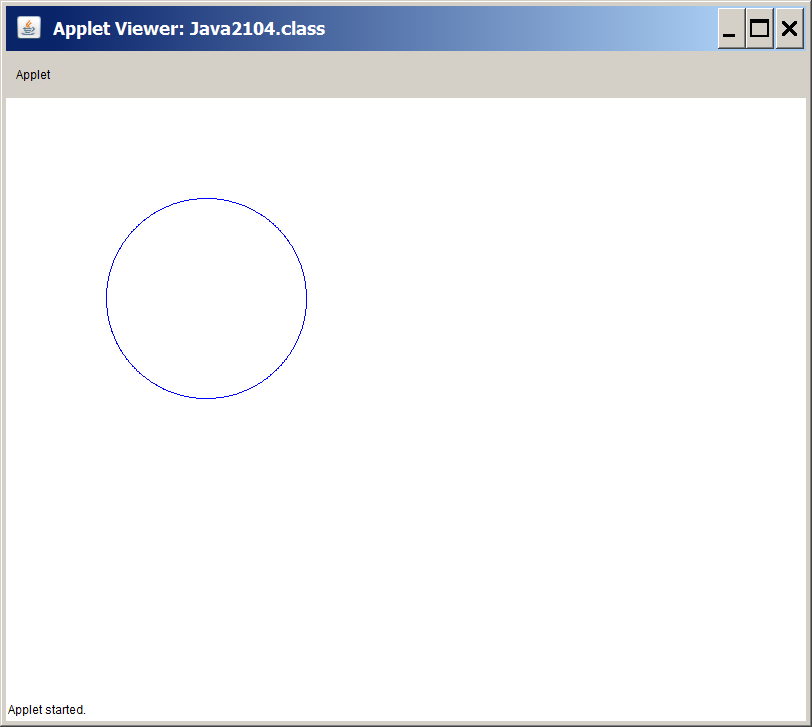
This problem is solved by adding an **xCenter** value and an **yCenter** value to the final outcome of the processed sin and cos results. You are so pleased that I decided not to explain too much mathematics, since this explanation probably has insulted your intelligence. Please consult your computer science teacher or math teacher if you need additional explanation on this topic.

**Figure 20.6**

|  |
| --- |
| // Java2004.java  // This program draws a circle using the <cos> and <sin> methods of the  // <Math> class. It is simpler to use the <drawOval> method. This program  // helps to explain the next program.  import java.awt.\*;  public class Java2004 extends java.applet.Applet  {  public void paint(Graphics g)  {  int x,y;  int radius = 100;  int centerX = 200;  int centerY = 200;  g.setColor(Color.blue);  for (double radian = 0; radian <= 2 \* Math.PI; radian += 0.01)  {  x = (int) Math.round(Math.cos(radian) \* radius) + centerX;  y = (int) Math.round(Math.sin(radian) \* radius) + centerY;  g.drawLine(x,y,x,y);  }  }  } |

Right about now there is some quiet or not so quiet grumbling happening. If you understand this correctly, the illustrious author of this stuff you are reading has drawn a circle in a very convoluted way. Earlier in the chapter you learned how to display a perfectly civilized circle with **drawOval**, and now you are tortured through **sin** and **cos** functions combined with radians and other strange details to draw a circle.

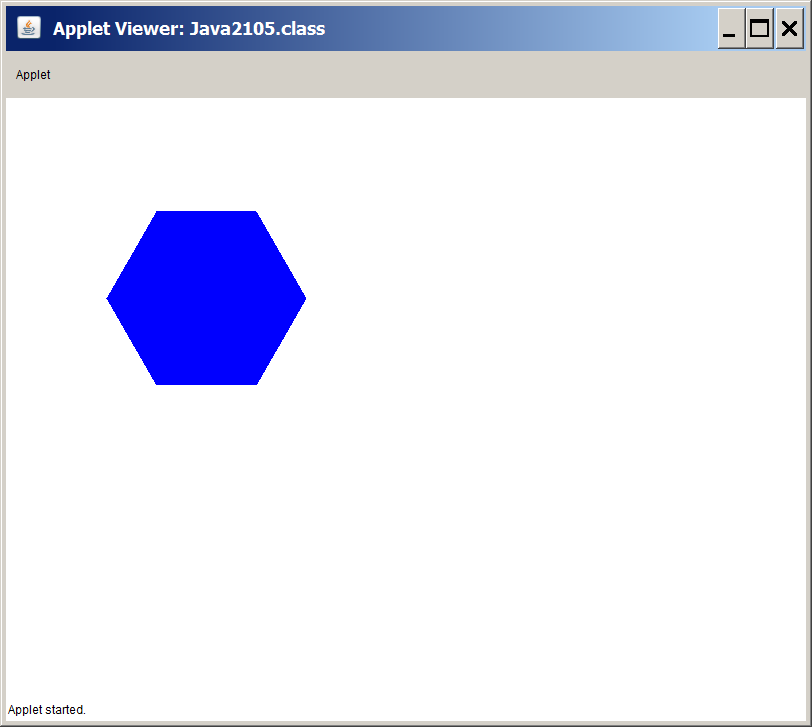
**Figure 20.6 Continued**



It may seem now like a long time ago, but our original aim in this section is to draw regular polygons. The previous program drew a lovely circle. Every regular polygon can be circumscribed by a circle. We can also say that the corners of any regular polygon can be found on a circle. In other words if we draw six points on a circle equidistant from each other, and connect those points, you will have a regular hexagon.

Now imagine **2\*PI** radians in a complete circle, which you can divide by five or six or seven and get the intervals, which you need, to draw a regular polygon. We will look at the program source code shortly, but in figure 20.7 you will see a lovely *regular* hexagon. Now let us see how this was done. This hexagon is not drawn with some special **Hexagon** class. It is drawn with the **Polygon** class using the **sin** and **cos** methods shown in the previous program.

**Figure 20.7**



Program **Java2005.java**, in figure 20.8, does not use a loop to process a circular set of values. A loop structure was necessary to draw a circle, because there were hundreds of points between 0 and 2PI. Now our mission is to find only six points on the circle and connect them to draw a hexagon. This task is accomplished by using values, like **1/6 \* 2PI**, and **2/6 \* 2PI** and **3/6 \* 2PI** and so on. The key difference is that you do not draw the points as the values are computed. You add the points to a Polygon object and then draw the regular hexagon. We used the same program logic that was used with a circle. The result is a regular hexagon because every point is 1/6 \* 2PI radians away from two adjacent points. The same exact logic can be used for a polygon of any size. There certainly are many program statements to compute the six hexagon coordinates and you may prefer to put this computation in a loop.

**Figure 20.8**

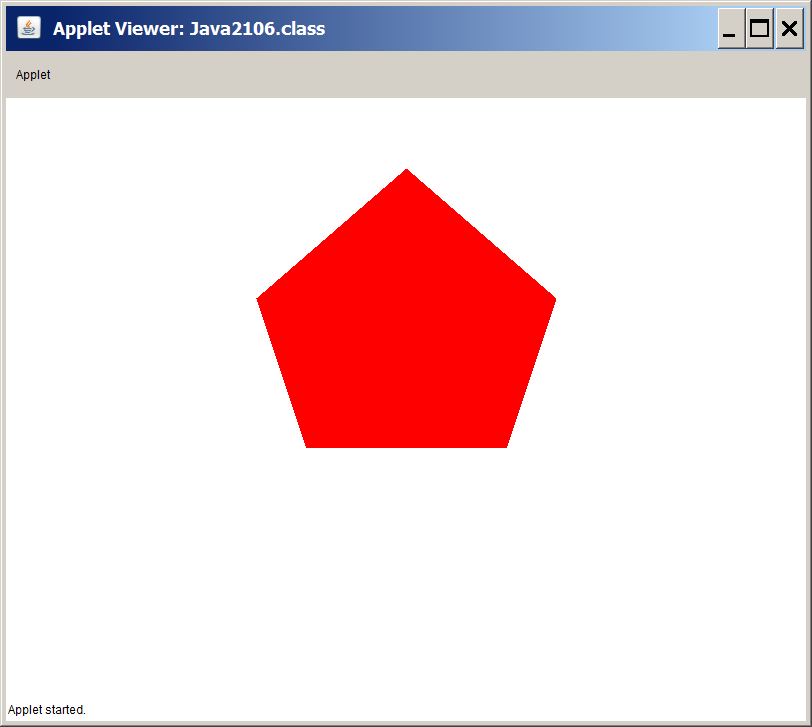
|  |
| --- |
| // Java2005.java  // This program draws a regular hexagon using the <cos> and <sin>  // methods of the <Math> class.  import java.awt.\*;  public class Java2005 extends java.applet.Applet  {    public void paint(Graphics g)  {  int radius = 100;  int centerX = 200;  int centerY = 200;  double twoPI = 2 \* Math.PI;  g.setColor(Color.blue);  int x1 = (int) Math.round(Math.cos(twoPI \* 1/6) \* radius) + centerX;  int y1 = (int) Math.round(Math.sin(twoPI \* 1/6) \* radius) + centerY;  int x2 = (int) Math.round(Math.cos(twoPI \* 2/6) \* radius) + centerX;  int y2 = (int) Math.round(Math.sin(twoPI \* 2/6) \* radius) + centerY;  int x3 = (int) Math.round(Math.cos(twoPI \* 3/6) \* radius) + centerX;  int y3 = (int) Math.round(Math.sin(twoPI \* 3/6) \* radius) + centerY;  int x4 = (int) Math.round(Math.cos(twoPI \* 4/6) \* radius) + centerX;  int y4 = (int) Math.round(Math.sin(twoPI \* 4/6) \* radius) + centerY;  int x5 = (int) Math.round(Math.cos(twoPI \* 5/6) \* radius) + centerX;  int y5 = (int) Math.round(Math.sin(twoPI \* 5/6) \* radius) + centerY;  int x6 = (int) Math.round(Math.cos(twoPI) \* radius) + centerX;  int y6 = (int) Math.round(Math.sin(twoPI) \* radius) + centerY;  Polygon hex = new Polygon();  hex.addPoint(x1,y1);  hex.addPoint(x2,y2);  hex.addPoint(x3,y3);  hex.addPoint(x4,y4);  hex.addPoint(x5,y5);  hex.addPoint(x6,y6);  g.fillPolygon(hex);  }    } |

Anybody with some sense of good program design will frown at the previous program. It required twelve program statements to generate six coordinate points for the hexagon. A program can become very tedious if you require a polygon with 24 sides.

There is a much shorter way to draw polygons. The **drawPolygon** and its cousin the **fillPolygon** methods can be overloaded and do not always require a **Polygon** object. It is also possible to provide an array of x-coordinates, along with an array of y-coordinates and the number of sides in the polygon. This approach was not shown in an earlier chapter, because you did not know how to handle arrays yet. Program **Java2006.java**, in figure 20.9, shows that a pentagon can be drawn with far fewer statements than the previous approach.

**Figure 20.9**

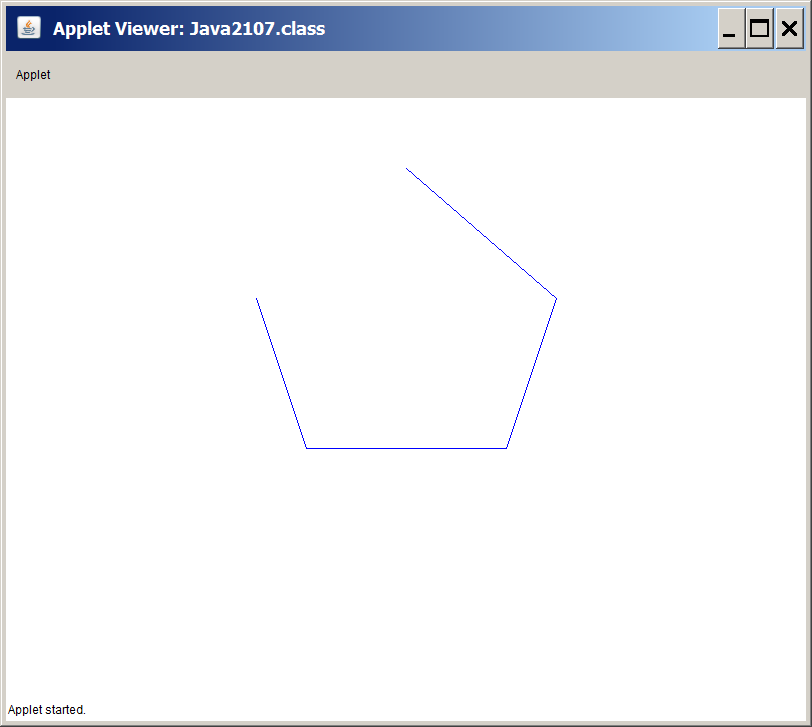
|  |
| --- |
| // Java2006.java  // This program draws a pentagon with the <fillPolygon> method.  // This example uses arrays of x and y coordinates with the fillPolygon method.  import java.awt.\*;  import java.applet.\*;  public class Java2006 extends Applet  {  public void paint(Graphics g)  {  g.setColor(Color.red);  int xCoord[] = {400,550,500,300,250};  int yCoord[] = {70,200,350,350,200};  g.fillPolygon(xCoord,yCoord,5);  }  } |



Program **Java2007.java**, in figure 20.10, shows that two arrays of coordinate values can also be used with the **drawPolyline** method. There is no method to fill in any space, like **fillPolyline**. Keep in mind that the sequence of coordinate points in the x-array and y-array still determine the display precisely as the sequence of coordinates used with the **addPoint** method.

**Figure 20.10**

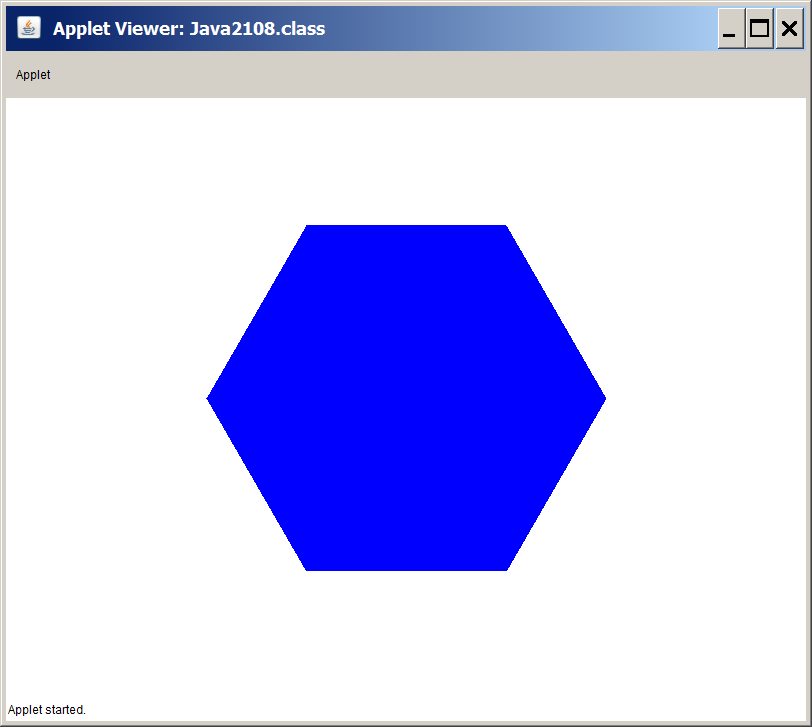
|  |
| --- |
| // Java2007.java  // This program draws a sequence of connected lines with the <drawPolyline> method.  import java.awt.\*;  import java.applet.\*;  public class Java2007 extends Applet  {  public void paint(Graphics g)  {  g.setColor(Color.blue);  int xCoord[] = {400,550,500,300,250};  int yCoord[] = {70,200,350,350,200};  g.drawPolyline(xCoord,yCoord,5);  }  } |



Program **Java2007.java** did not draw a regular pentagon. It is still possible to combine the **sin** and **cos** lessons from earlier sections to this current array stuff and draw regular polygons in an efficient manner. Program **Java2008.java**, in figure 20.11, draws a lovely regular hexagon using loops and arrays.

**Figure 20.11**

|  |
| --- |
| // Java2008.java  // This program demonstrates how to draw a regular hexagon efficiently by using coordinates  // arrays inside a loop control structure.  import java.awt.\*;  import java.applet.\*;  public class Java2008 extends Applet  {  public void paint(Graphics g)  {  int centerX = 400;  int centerY = 300;  int radius = 200;  double twoPI = 2 \* Math.PI;  int xCoord[] = new int[6];  int yCoord[] = new int[6];  g.setColor(Color.blue);  for (int k = 0; k < 6; k++)  {  xCoord[k] = (int) Math.round(Math.cos(twoPI \* k/6) \* radius) + centerX;  yCoord[k] = (int) Math.round(Math.sin(twoPI \* k/6) \* radius) + centerY;  }  g.fillPolygon(xCoord,yCoord,6);  }  } |

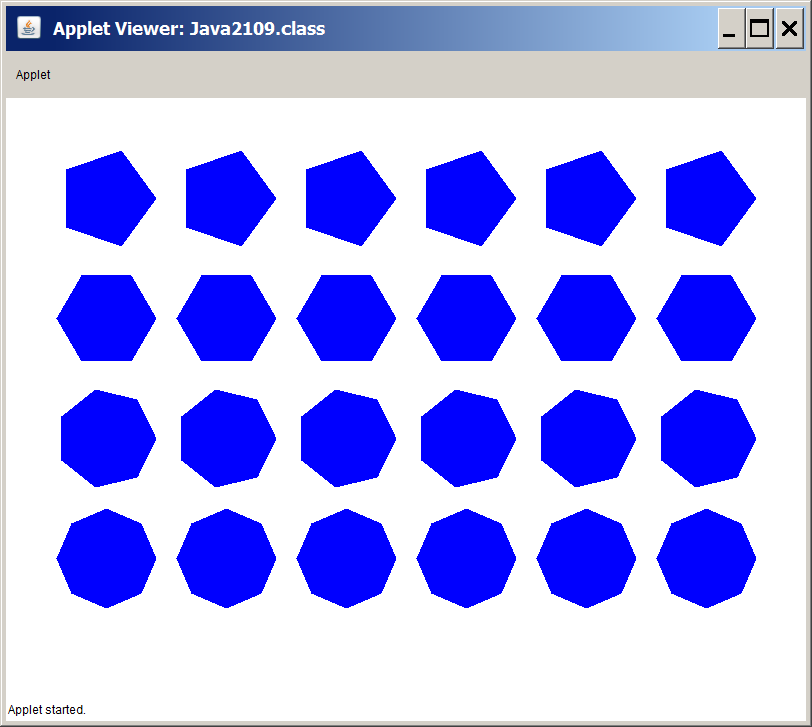


We have not finished with the efficiency business. You may have noticed that the previous program used a loop and used arrays in a manner that is specific for a regular hexagon. Did you notice the number **6** used many times? Imagine that you take the code necessary to create a regular polygon and place this code in a method. Furthermore, substitute a variable in place of the constant 6. What do you get? You get program **Java2009.java**, in figure 20.12, which is capable of drawing any size regular polygon. You will now see an applet filled with regular polygons of many different sizes.

**Figure 20.12**

|  |
| --- |
| // Java2009.java  // This program uses the regular hexagon code and creates a general regular Polygon method.  import java.awt.\*;  import java.applet.\*;  public class Java2009 extends Applet  {    public void paint(Graphics g)  {  int sides = 5;  for (int y = 100; y <= 500; y+=120)  {  for (int x = 100; x <= 800; x+=120)  regPolygon(g,50,x,y,sides);  sides++;  }  }    public void regPolygon(Graphics g, int radius, int centerX, int centerY,int sides)  {  double twoPI = 2 \* Math.PI;  int xCoord[] = new int[sides];  int yCoord[] = new int[sides];  g.setColor(Color.blue);  for (int k = 0; k < sides; k++)  {  xCoord[k] = (int) Math.round(Math.cos(twoPI \* k/sides) \* radius) + centerX;  yCoord[k] = (int) Math.round(Math.sin(twoPI \* k/sides) \* radius) + centerY;  }  g.fillPolygon(xCoord,yCoord,sides);  }  } |

**Figure 20.12 Continued**



**20.5 Mouse Routines with Graphics**

This chapter is titled *Advanced Graphics*. You may argue that the mathematical fun stuff of the previous section counts as advanced, but probably you are not very impressed at this stage. Hopefully, this section will change your perspective. You are about to learn how to use a mouse to interact with graphics displays. In the spirit of *Exposure Java*, which means learning repeatedly with small pieces of new information, you will see many different programs using mouse features. In many cases one program will be used to address one mouse feature. By the time that you are finished, you may be surprised that you can do some sophisticated graphics programming with this information. At the end of this chapter, you should be able to complete a lab assignment, which creates a paint program somewhat similar in capabilities to the MS-Windows Paint program.

Java is constantly improving its class and method selection. When new and improved classes are available, older classes and method are considered so "yesterday" and when you use the older stuff you get warning messages, not error messages, that you are using old beat-up Java methods. I find that much of the older graphics routines are easier to teach than the newer goodies, and they work very nicely for my purposes. You can either ignore warning messages when they arrive or you can turn off the warning messages. Each year with a new edition of Exposure Java I add new features, but I do not instantly change to some new feature if I feel that the older approach is very adequate.

|  |
| --- |
| **Steps to Turn Off Java Warning Messages in JCreator** |
| **1. Click Configure**  **2. Click Options**  **3. Click JDK Tools**  **4. Select Compiler in the tool type window**  **5. Click Edit**  **6. Click Parameters tab**  **7. Remove check from Warnings check box**  **8. Click OK twice** |

Fine, the warning business is settled and we can start to learn some serious mouse interaction in our Java applets. You will find that there is not much explanation with each program. Running the programs, observing the results and taking inventory of the methods necessary to perform the given action will teach you the majority of the mouse features. Now let us get the ball rolling with program **Java2010.java**, in figure 20.20. This program will count and display the number of times that the mouse is clicked. Program **Java2010.java** requires some explanation, because it includes a variety of new features that will be puzzling at first introduction.

You have already seen the **paint** method for some time. Applets use the **paint** method to display graphics patterns on the monitor. You have also learned that the **paint** method is called automatically by the web browser that executes the bytecode file of an applet. Method **paint** is not the only method that is called by the web browser. Method **init** is executed before **paint**, if it exists. True to its name, method **init** usually performs the job of initializing variables. In program **Java2010.java** the **init** method sets the **numClicks** integer field to **0**. After method **init** you see the familiar **paint** method. Apparently, method **paint** is only in charge of displaying a graphics string indicating how many times the mouse has been clicked.

**Counting Mouse Clicks**

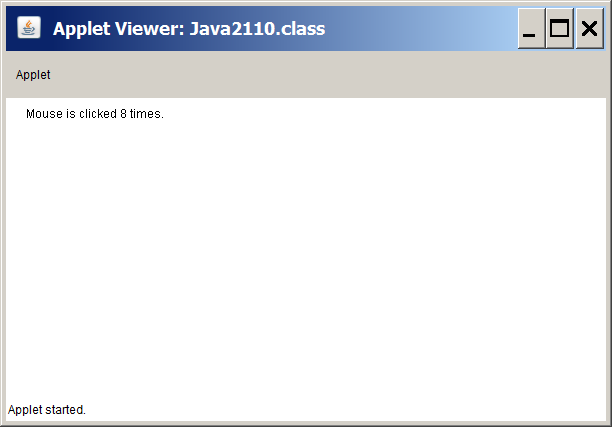
Below method **paint** appears to be another method called **mouseDown**. There is no evidence that **mouseDown** is called anywhere. Is this yet another method that is called automatically? Well **mouseDown** is not called like **init** or **paint** and it y is not a regular method. It is an event. Events are routines that are not called by using the identifier of the event, but the events are executed when some special *event* occurs. Mouse-type-events occur when the mouse is clicked, released, moved or dragged. Java has created a special behind-the-scene interface with the mouse that knows when something new is happening with the mouse.

In the case of **Java2010.java** something new means that either mouse button is clicked. It results in calling **mouseDown** and then **mouseDown** changes the **numClicks** value, as well as calling **repaint**. A call to method **repaint** results in executing method **paint** again. The bottom line is that every time you click, the number increases.

**Figure 20.13**

|  |
| --- |
| // Java2010.java  // This program counts the number of times a mouse is clicked.  // Both left-clicks and right-clicks are counted.  // using the <mouseDown> event. Ignore the "deprecated API" warning.  import java.applet.Applet;  import java.awt.\*;  public class Java2010 extends Applet  {  int numClicks;    public void init()  {  numClicks = 0;  }  public void paint(Graphics g)  {  g.drawString("Mouse is clicked " + numClicks + " times.",20,20);  }    public boolean mouseDown(Event e, int x, int y)  {  numClicks++;  repaint();  return true;  }    } |

**Figure 20.13 Continued**



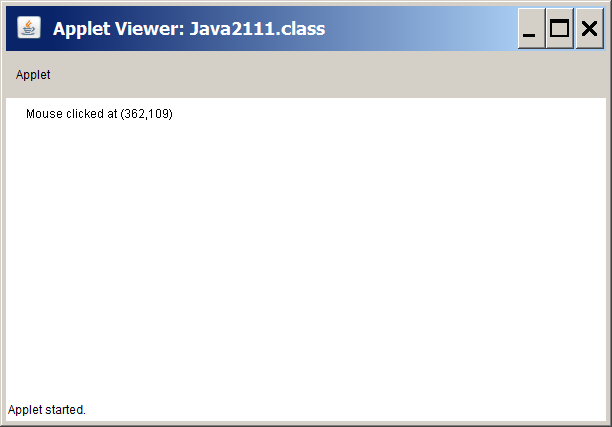
**Detecting Mouse Click Coordinates**

Each one of the programs that follows will demonstrate some new mouse feature. A brief explanation will detail the actions of the new mouse feature. By now you should realize that "playing" with the program is the most effective way to learn new program language concepts. Program **Java2011.java**, in figure 20.14 shows that the **mouseDown** event records the *X-coordinate* and *Y-Coordinate* values when the mouse is clicked in the **x** and **y** parameters of the **mouseDown** event.

**Figure 20.14**

|  |
| --- |
| // Java2011.java  // This program displays the position of the mouse every time it is clicked  // using the <mouseDown> method.  import java.applet.Applet;  import java.awt.\*;  public class Java2011 extends Applet  {  int xCoord, yCoord;    public void paint(Graphics g)  {  g.drawString("Mouse clicked at (" + xCoord + "," + yCoord + ")",20,20);  }    public boolean mouseDown(Event e, int x, int y)  {  xCoord = x;  yCoord = y;  repaint();  return true;  }  } |

**Figure 20.14 Continued**



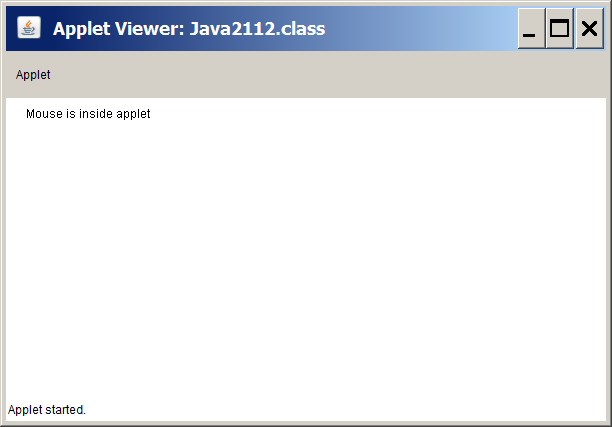
**Determining if the Mouse is In the Applet Window**

Program **Java2012.java**, in figure 20.15 uses both the **mouseEnter** event and **mouseExit** event to determine when the mouse enter or exits the applet window.

**Figure 20.15**

|  |
| --- |
| // Java2012.java  // This program demonstrates how to determine if the mouse is inside or  // outside the Applet window using the <mouseEnter> and <mouseExit> methods.  import java.applet.Applet;  import java.awt.\*;  public class Java2012 extends Applet  {  boolean insideApplet;    public void paint(Graphics g)  {  if (insideApplet)  g.drawString("Mouse is inside applet",20,20);  else  g.drawString("Mouse is outside applet",20,20);  }    public boolean mouseEnter(Event e, int x, int y)  {  insideApplet = true;  repaint();  return true;  }    public boolean mouseExit(Event e, int x, int y)  {  insideApplet = false;  repaint();  return true;  }  } |

**Figure 20.15 Continued**



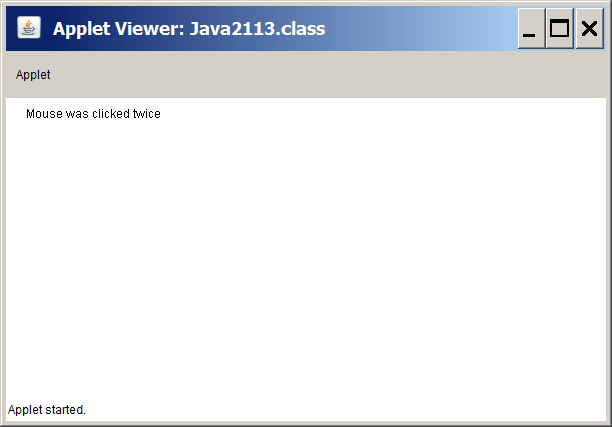
**Checking if the Button is Clicked Once or Twice**

The **e** object in the parameter heading of the **mouseDown** event has a **clickCount** method, which returns the number of times that a mouse is clicked. Program **Java2020.java**, in figure 20.16, demonstrates this concept. Keep in mind that the time delay between first and second click is based on Windows mouse settings.

**Figure 20.16**

|  |
| --- |
| // Java2020.java  // This program determines if a mouse is clicked once or twice using the  // <clickCount> method. This method works for the left or right button.  import java.applet.Applet;  import java.awt.\*;  public class Java2013 extends Applet  {  boolean singleClick,doubleClick;    public void paint(Graphics g)  {  if (singleClick)  g.drawString("Mouse was clicked once",20,20);  if (doubleClick)  g.drawString("Mouse was clicked twice",20,20);  }    public boolean mouseDown(Event e, int x, int y)  {  switch (e.clickCount)  {  case 1:  singleClick = true; doubleClick = false;  break;  case 2:  doubleClick = true; singleClick = false;  }  repaint();  return true;  }  } |

**Figure 20.16 Continued**



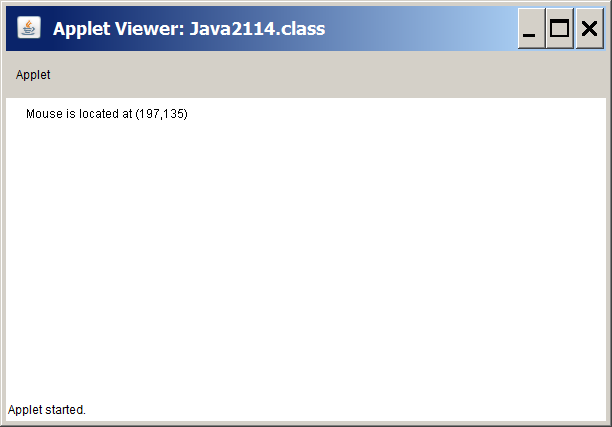
**Displaying the Mouse Position Continuously**

A few programs back the program displayed the mouse position whenever the mouse was clicked. Program **Java2014.java**, in figure 20.17, constantly checks and updates the location of the mouse using the **mouseMove** event. Any movement of the mouse triggers the **mouseMove** event, which stores the latest coordinate values in the **x** and **y** parameters.

**Figure 20.17**

|  |
| --- |
| // Java2014.java  // This program displays the position of the mouse every time it moves  // using the <mouseMove> method.  import java.applet.Applet;  import java.awt.\*;  public class Java2014 extends Applet  {  int xCoord, yCoord;    public void paint(Graphics g)  {  g.drawString("Mouse is located at (" + xCoord + "," + yCoord + ")",20,20);  }    public boolean mouseMove(Event e, int x, int y)  {  xCoord = x;  yCoord = y;  repaint();  return true;  }  } |

**Figure 20.17 Continued**



**The Start of a Simple Draw Program**

Program **Java2015.java**, in figure 20.18, does not introduce any new mouse routines. Rather this program example uses the **mouseDown** event in a more practical manner to draw small squares. One popular Windows program is the **Paint** program. With clicking and dragging a person can create pictures on the computer screen that are only limited by the individual's creativity. This drawing style depends heavily on the use of the mouse. This next program example is an very primitive paint program. Each time that the mouse is clicked a small, red square is drawn in the applet window.

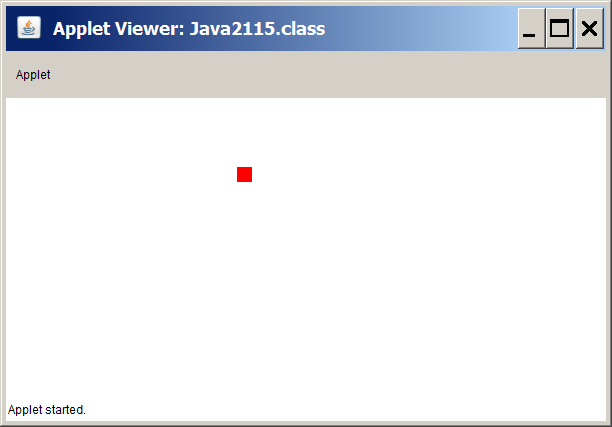
You learned earlier that the **mouseDown** event stores the coordinate values each time that a mouse button is clicked. The coordinates are placed in the **x** and **y** parameters of the **mouseDown** event. Whenever the **mouseDown** event is triggered the current mouse position is recorded in the **xCoord** and **yCoord** variables and method **repaint** is called to return to the **paint** method. In method **paint** a small, red square is drawn.

Another issue needs to be addressed. Method **paint** is called before the mouse is ever touched. This would mean that one initial square would be drawn in the applet, even though the mouse is not clicked. This problem is solved with the use of a **boolean** variable **firstPaint**. Variable **firstPaint** is initialized to **true** in the **init** method and prevents drawing a rectangle until the second time that method **paint** is visited.

This program example and many other mouse routine programs cannot be fully comprehended by reading the text alone. You must execute the program examples to understand the actual actions of the mouse routines.

**Figure 20.18**

|  |
| --- |
| // Java2015.java  // This program draws a small square at every mouse click position.  import java.applet.Applet;  import java.awt.\*;  public class Java2015 extends Applet  {  int xCoord;  int yCoord;  boolean firstPaint;    public void init()  {  firstPaint = true;  }    public void paint(Graphics g)  {  if (firstPaint)  firstPaint = false;  else  {  g.setColor(Color.red);  g.fillRect(xCoord,yCoord,15,15);  }  }    public boolean mouseDown(Event e, int x, int y)  {  xCoord = x;  yCoord = y;  repaint();  return true;  }  } |

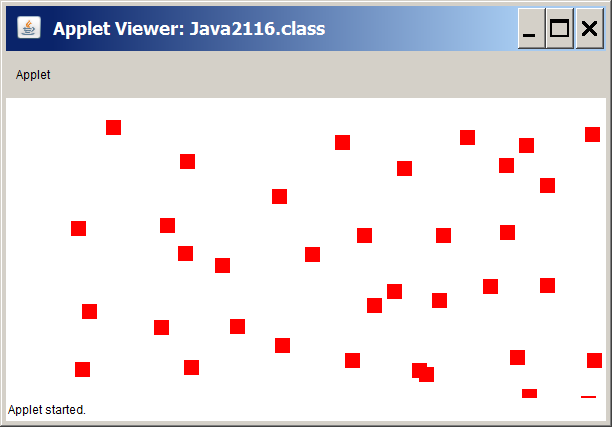


Let us assume that you understand the previous program perfectly. There is an interesting problem that must be solved. How do you draw anything and keep the drawing on the screen? The previous program drew many fat pixels, but each time that **paint** is called, it clears the screen and only displays the newest pixel. Program **Java2016.java**, in figure 20.19, shows one solution. Two **int** arrays are declared. Each array stores a coordinate value. The arrays can store up to 100 coordinate values. Each time that **paint** is called, the two coordinate arrays are used to display each one of the previous pictures. The illusion is that each pixel is added individually. The reality is that the screen is "repainted" from scratch every time that the **paint** method is called. The arrays are constructed with 100 **int** elements. Any attempt to draw more than 100 pixels will crash the program,

**Figure 20.19**

|  |
| --- |
| // Java2016.java  // This program draws small squares at every mouse click position.  // The program will crash if you try to draw more than 100 squares.  import java.applet.Applet;  import java.awt.\*;  public class Java2016 extends Applet  {  int xCoord[];  int yCoord[];  int numSquare;  boolean firstPaint;    public void init()  {  xCoord = new int[100];  yCoord = new int[100];  numSquare = 0;  firstPaint = true;  }    public void paint(Graphics g)  {  g.setColor(Color.red);  for (int k = 0; k < numSquare; k++)  g.fillRect(xCoord[k],yCoord[k],15,15);  }    public boolean mouseDown(Event e, int x, int y)  {  xCoord[numSquare] = x;  yCoord[numSquare] = y;  numSquare++;  repaint();  return true;  }  } |

**Figure 20.19 Continued**



**Determining if the Mouse Clicked Inside an Area**

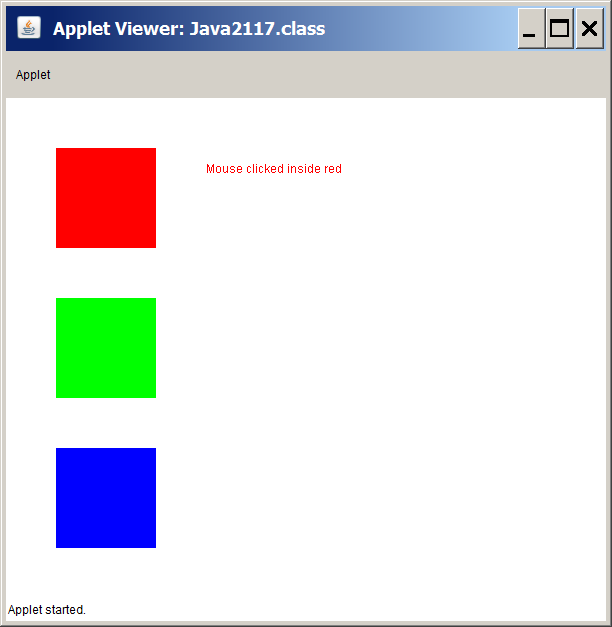
There is something that computer mouse users take for granted. It is the ability of the program to know that a mouse has clicked inside some area. We click on a color in a paint program. We click anywhere on some icon. We click on the top bar of a window. In each case clicking is not equal to some specified coordinate location. Rather some action results from clicking anywhere in a specified area.

Program **Java2017.java**, in figure 20.20, displays three colored squares. The mission of the program is to see if the program can tell if the mouse is clicked anywhere in one of the three squares or anywhere outside the squares. This program includes some new concepts that will be shown separately. Look at the program in its entirety first with its execution and then more explanation follows.

**Figure 20.20**

|  |
| --- |
| // Java2017.java  // This uses the <Rectangle> class with the <inside> method to determine if a certain square has been clicked.  import java.applet.Applet;  import java.awt.\*;  public class Java2017 extends Applet  {  Rectangle red, green, blue;  int numColor;    public void init()  {  red = new Rectangle(50,50,100,100);  green = new Rectangle(50,200,100,100);  blue = new Rectangle(50,350,100,100);  numColor = 0;  }  public void paint(Graphics g)  {  g.setColor(Color.red);  g.fillRect(50,50,100,100);  g.setColor(Color.green);  g.fillRect(50,200,100,100);  g.setColor(Color.blue);  g.fillRect(50,350,100,100);  switch (numColor)  {  case 1:  g.setColor(Color.red);  g.drawString("Mouse clicked inside red", 200,75);  break;  case 2:  g.setColor(Color.green);  g.drawString("Mouse clicked inside green", 200,225);  break;  case 3:  g.setColor(Color.blue);  g.drawString("Mouse clicked inside blue", 200,375);  break;  case 4:  g.setColor(Color.black);  g.drawString("Mouse is clicked outside the colored squares", 50,20);  break;  }  }  public boolean mouseDown(Event e, int x, int y)  {  if(red.inside(x,y))  numColor = 1;  else if(green.inside(x,y))  numColor = 2;  else if(blue.inside(x,y))  numColor = 3;  else  numColor = 4;  repaint();  return true;  }  } |

**Figure 20.20 Continued**



Now that you have had a chance to see what the program looks like and how it behave, let us break it apart and start understanding the separate pieces. A good place to start is the **init** method, shown in figure 20.20.

**Figure 20.21**

|  |
| --- |
| **public void init()**  **{**  **red = new Rectangle(50,50,100,100);**  **green = new Rectangle(50,200,100,100);**  **blue = new Rectangle(50,350,100,100);**  **numColor = 0;**  **}** |

The **Rectangle** class is a new class. Do not confuse this class with methods **drawRect** or **fillRect**. The **Rectangle** class constructs objects with parameters that are identical to displaying a rectangle, but the dimensions of the objects are abstract. In other words, each **Rectangle** object is dimensioned without actually displaying any rectangle. The abstract dimension is used in combination with mouse routines to determine if the mouse has clicked inside the rectangle area.

Figure 20.22 helps to clarify the purpose of the three **Rectangle** objects. The **Rectangle** class provides the very useful method **inside**, which determines if a set of coordinates are inside the dimensions of the **Rectangle** object. You see yet another example of the **mouseDown** event. In this case the coordinate values provided when the event is triggered determine if the mouse is clicked inside any of the three squares or outside any of the three squares. Variable **numColor** is assigned a color appropriate to the mouse-click location that will be used when **paint** method is called.

**Figure 20.22**

|  |
| --- |
| **public boolean mouseDown(Event e, int x, int y)**  **{**  **if(red.inside(x,y))**  **numColor = 1;**  **else if(green.inside(x,y))**  **numColor = 2;**  **else if(blue.inside(x,y))**  **numColor = 3;**  **else**  **numColor = 4;**  **repaint();**  **return true;**  **}** |

Finally, we look at the **paint** method in figure 20.23, to see the exciting conclusion of the **Rectangle** class program. You may easily say that the three rectangles are hardly abstract. You can see them in three colors. That is true, but you are not seeing any of the **Rectangle** objects. Method **paint** starts with three calls to the **fillRect** method, which results in displaying rectangles with the same dimensions as the **red**, **green** and **blue** objects. The **switch** control structure is used to display the appropriate message based on the color value of **numColor**.

**Figure 20.23**

|  |
| --- |
| public void paint(Graphics g)  {  g.setColor(Color.red);  g.fillRect(50,50,100,100);  g.setColor(Color.green);  g.fillRect(50,200,100,100);  g.setColor(Color.blue);  g.fillRect(50,350,100,100);  switch (numColor)  {  case 1:  g.setColor(Color.red);  g.drawString("Mouse clicked inside red", 200,75);  break;  case 2:  g.setColor(Color.green);  g.drawString("Mouse clicked inside green", 200,225);  break;  case 3:  g.setColor(Color.blue);  g.drawString("Mouse clicked inside blue", 200,375);  break;  case 4:  g.setColor(Color.black);  g.drawString("Mouse is clicked outside the colored squares", 50,20);  break;  }  } |

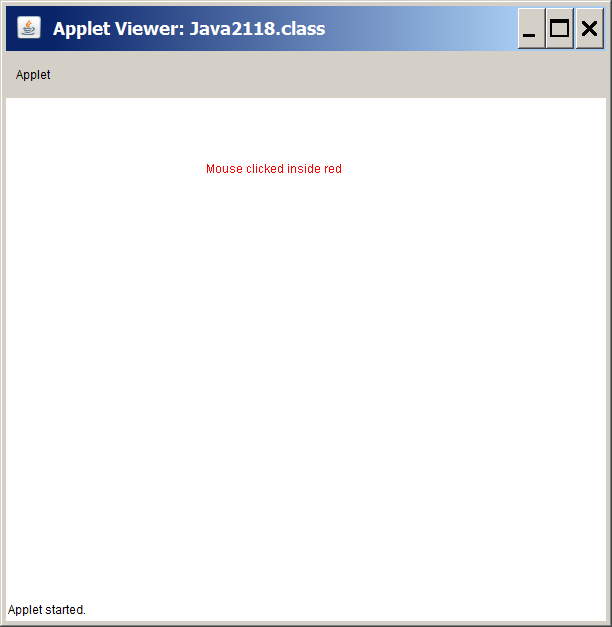
You may not really quite get this abstract business. Perhaps with program **Java2018.java**, in figure 20.24, you will appreciate that objects **red**, **green** and **blue** are true abstract rectangles. The next program example is almost identical to program **Java2017.java**, but the three **drawRect** calls are not made inside the **paint** method.

This program simply cannot be appreciated by reading the text book. You must execute the program and see that it is still possible to get a response, but this response is based on estimating where you expect the rectangles to be.

**Figure 20.24**

|  |
| --- |
| // Java2018.java  // This program proves that objects of the <Rectangle> class are abstract to the viewer and  // not visible. The three square are intentionally not displayed. The program still works  // like the previous program, but you must guess at the location of the squares.  import java.applet.Applet;  import java.awt.\*;  public class Java2018 extends Applet  {  Rectangle red, green, blue;  int numColor;    public void init()  {  red = new Rectangle(50,50,100,100);  green = new Rectangle(50,200,100,100);  blue = new Rectangle(50,350,100,100);  numColor = 0;  }    public void paint(Graphics g)  {  switch (numColor)  {  case 1:  g.setColor(Color.red);  g.drawString("Mouse clicked inside red", 200,75);  break;  case 2:  g.setColor(Color.green);  g.drawString("Mouse clicked inside green", 200,225);  break;  case 3:  g.setColor(Color.blue);  g.drawString("Mouse clicked inside blue", 200,375);  break;  case 4:  g.setColor(Color.black);  g.drawString("Mouse is clicked outside the colored squares", 50,20);  break;  }  }    public boolean mouseDown(Event e, int x, int y)  {  if(red.inside(x,y))  numColor = 1;  else if(green.inside(x,y))  numColor = 2;  else if(blue.inside(x,y))  numColor = 3;  else  numColor = 4;  repaint();  return true;  }  } |

**Figure 20.24 Continued**



**Drawing "Rubber band" Straight Lines**

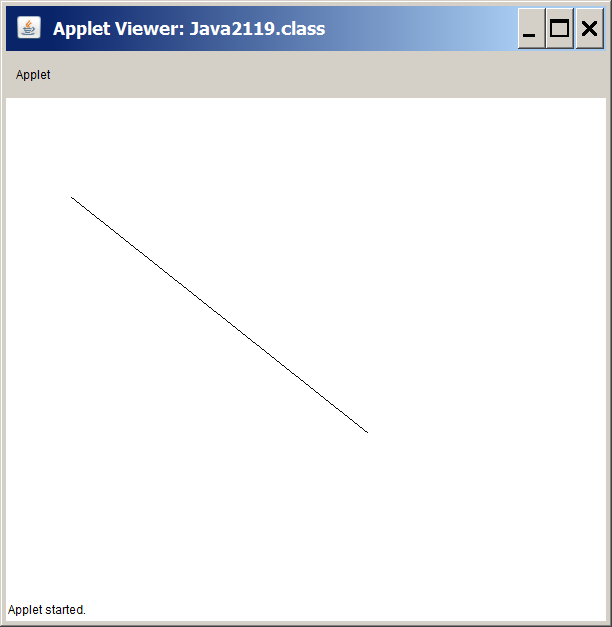
In all "Paint" type programs it is possible the draw a straight line from a starting point to a given end point. This is done by clicking on the start point and then dragging to the end point. As you drag the mouse the line is constantly changing as if you were moving a rubber band that is attached on one end.

Program **Java2019.java**, in figure 20.25, will not exactly give you this feeling of a "rubber band" since the line is not visible until you release the mouse button. Just wait, because this is the first in a series of programs to achieve the line drawing skills.

You have seen the **mouseDown** event used in many program examples. This program also uses the **mouseUp** event, which triggers an event the moment the mouse button is released. These two methods are very useful working together. This program records the coordinates where the line starts in the **mouseDown** event and then records a second set of coordinates where the line should end with the **mouseUp** event. The **paint** method is one of the shortest methods in this mouse program section. There is one simple statement to draw a straight line.

**Figure 20.25**

|  |
| --- |
| // Java2019.java  // This program draws a straight line from the point where the mouse is clicked  // to the point where the mouse is released.  import java.applet.Applet;  import java.awt.\*;  public class Java2019 extends Applet  {    int startX,startY,endX,endY;    public void paint(Graphics g)  {  g.drawLine(startX,startY,endX,endY);  }  public boolean mouseDown(Event e, int x, int y)  {  startX = x;  startY = y;  return true;  }    public boolean mouseUp(Event e, int x, int y)  {  endX = x;  endY = y;  repaint();  return true;  }    } |



The last program was not very satisfying. Basically, you are moving a mouse in an attempt to draw a line, but you do not see where you came from, where you are or where you are going. Pretty much you are drawing in the blind. Program **Java2020.java**, in figure 20.26, will make some major improvements and you will see a true "rubber band" drawing approach.

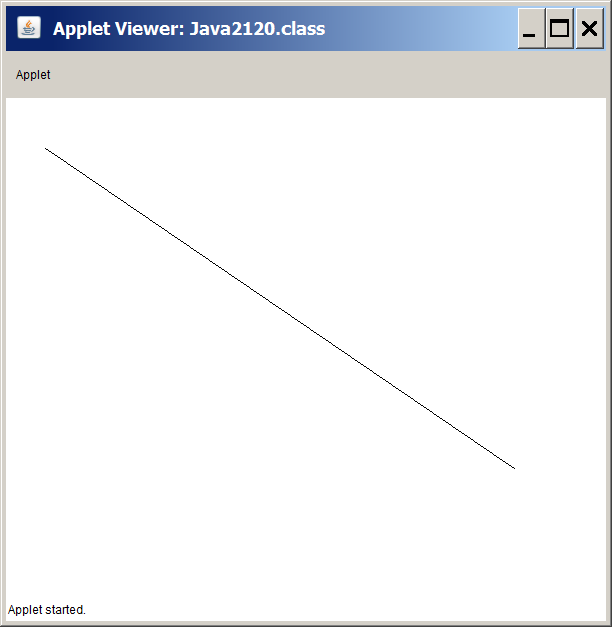
You will note that the **mouseUp** method is now gone and in its place is the new **mouseDrag** event. Event **mouseDrag** is triggered anytime that the mouse ball moves with a button clicked. Each time that the mouse moves the **mouseDrag** event records new **endX** and **endY** values, call the **paint** method and a new line is drawn. Calling method **paint** starts by clearing the screen and drawing a brand-new line each time gives the illusion of the "rubber band" stretching and contracting.

The sample execution in the textbook is pretty useless. It looks identical to the execution of the previous program since it only shows the end result of the straight line. I do hope that you realize more and more that learning computer science is very dynamic and you need to observe programs during execution with different values and different actions to understand the new concepts.

**Figure 20.26**

|  |
| --- |
| // Java2020.java  // This program draws a straight line from the point where the mouse is clicked  // to the point where the mouse is released. In this example the line is  // constantly visible.  import java.applet.Applet;  import java.awt.\*;  public class Java2020 extends Applet  {  int startX,startY,endX,endY;    public void paint(Graphics g)  {  g.drawLine(startX,startY,endX,endY);  }  public boolean mouseDown(Event e, int x, int y)  {  startX = x;  startY = y;  return true;  }    public boolean mouseDrag(Event e, int x, int y)  {  endX = x;  endY = y;  repaint();  return true;  }    } |

**Figure 20.26 Continued**



The "rubber band" drawing requires one more program. It is a program that draws straight lines in a nice "rubber band" style and then somehow stores the previous lines. If we want to draw multiple straight lines it simply does not make sense to only see a single line.

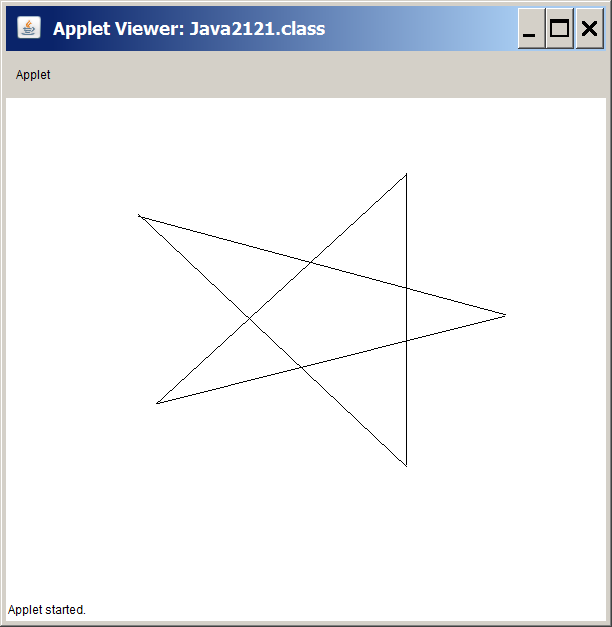
In a previous program you saw how it was possible to draw big pixels and display all the previous pixels on the screen. This was accomplished by storing the coordinate values of each pixel in two coordinate arrays. The same technique can store multiple lines. It does require four arrays, because you need to store two coordinate values for the start of each line and two more values for the end points.

There is one new twist. The **paint** method has a dual job. First, it needs to draw all the previous lines. Second, it has the task of drawing the "rubber bend" line as it is drawn. Drawing the previous lines is done by using the four coordinate value arrays, which are used to get all the start-points and end-points of all the previous lines. After that job is done the current line is drawn. The current line is the line that behaves like a "rubber band" and is not yet stored in the arrays. A line that is currently moving cannot be added to the line-value arrays until the current line is finished.

**Figure 20.27**

|  |
| --- |
| // Java2020.java  // This program draws a straight line from the point where the mouse is clicked  // to the point where the mouse is released. Additionally, the program stores the  // lines coordinate information in arrays so that all the lines are visible.  import java.applet.Applet;  import java.awt.\*;  public class Java2021 extends Applet  {    int[] startX,startY,endX,endY;  int currentStartX,currentStartY,currentEndX,currentEndY;  boolean currentLineDone;  int lineCount;    public void init()  {  startX = new int[100];  startY = new int[100];  endX = new int[100];  endY = new int[100];  lineCount = 0;  currentLineDone = false;  }    public void paint(Graphics g)  {  for (int k = 0; k < lineCount; k++)  g.drawLine(startX[k],startY[k],endX[k],endY[k]);  if (!currentLineDone)  g.drawLine(currentStartX,currentStartY,currentEndX,currentEndY);  currentLineDone = false;  }    public boolean mouseDown(Event e, int x, int y)  {  currentStartX = x;  currentStartY = y;  return true;  }    public boolean mouseDrag(Event e, int x, int y)  {  currentEndX = x;  currentEndY = y;  repaint();  return true;  }    public boolean mouseUp(Event e, int x, int y)  {  addLine(x,y);  currentLineDone = true;  return true;  }    public void addLine(int x, int y)  {  startX[lineCount] = currentStartX;  startY[lineCount] = currentStartY;  endX[lineCount] = x;  endY[lineCount] = y;  lineCount++;  repaint();  }  } |

**Figure 20.27 Continued**



**Mouse Drawing Without Straight Lines**

We now come to the last program example in our mouse routines section. There is one final program that demonstrates how to draw continuously with a mouse, but not in the straight-line-rubber-band approach. This program will demonstrate strictly free-hand drawing. What you see is what you get. It is like holding a pencil in your hand and you control the pencil.

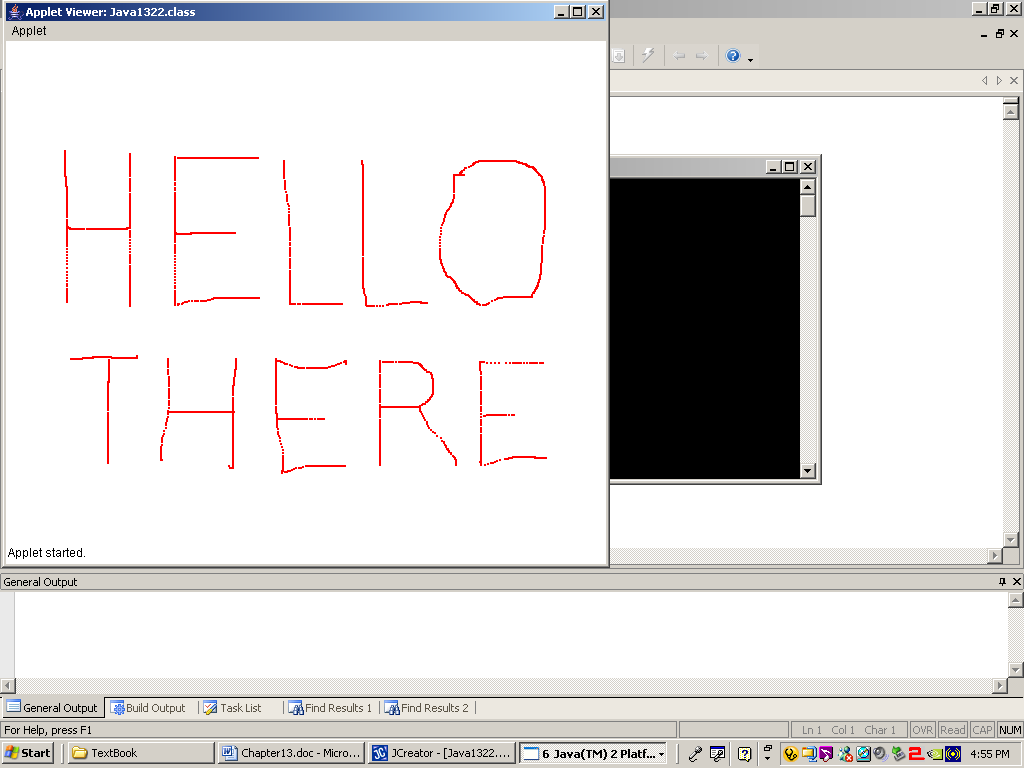
It turns out that creating a program that draws pixels continuously is not so difficult. In an earlier program you were able to draws pixels as the mouse was clicked. That mouse program used the **mouseDown** event and stored the pixel coordinate values of previous pixels. A small change to using the **mouseDrag** event changes the program and draws every pixel along the way.

If you move the mouse slowly the lines appear solid and neat. If you move the mouse quickly, the program will not be able to keep up with each coordinate change and the lines will appear as a series of disconnected pixels.

Program **Java2022.java**, in figure 20.28, attempts to draw **HELLO THERE** with the mouse. I am not much of an artist. I can certainly write programs much better than I can draw with a pencil, a brush or a mouse.

**Figure 20.28**

|  |
| --- |
| // Java2022.java  // This program demonstrates a simple mouse draw approach.  import java.applet.Applet;  import java.awt.\*;  public class Java2022 extends Applet  {    int[] xCoord,yCoord;  int pointCount;    public void init()  {  xCoord = new int[10000];  yCoord = new int[10000];  pointCount = 0;  }    public void paint(Graphics g)  {  g.setColor(Color.red);  for (int k = 0; k < pointCount; k++)  g.fillRect(xCoord[k],yCoord[k],2,2);  }    public boolean mouseDrag(Event e, int x, int y)  {  xCoord[pointCount] = x;  yCoord[pointCount] = y;  pointCount++;  repaint();  return true;  }  } |



**20.6 Understanding Graphics Displays**

It is possible that you noticed some annoying flickering with the last mouse routine program. The program worked correctly, but it sure seemed nervous and was constantly jumping around. Perhaps you did not care and took the casual attitude that this is just a first-year or second-year course and we are not yet creating a Jurassic Park type movie. That is true enough, but you will learn how to solve the flicker problem before the chapter is finished.

The mouse routines are behind us and the emphasis will now shift to another aspect of graphics programming. Before we return to program examples, you need to learn some computer science concepts related to graphics programming that will help to explain the next set of programs. In particular how does a certain colored pixel end up on a monitor location? I will try to answer this question using as little technical stuff as possible. My aim is to give you enough information that will help to clarify a new set of programs that are designed to show graphics animation.

There is an interaction between the *monitor*, *video memory* and *video card* that brings about some desired graphics display. First we need to briefly understand the function of each one of these three components.

You have learned that the computer monitor behaves like a coordinate geometry system with the (0,0) at the top-left corner of the monitor. The number of coordinate values where individually colored dots, or pixels, can be displayed depends on the monitor resolution. A monitor with an 800 X 600 resolution will have a total of 480,000 pixels. A monitor with a 1024 X 768 resolution will have 786,432 pixels. The greater the resolution, the more and smaller pixels on the monitor and the better the graphics image looks. However, do keep in mind that higher resolutions require more memory and computer processing, which can potentially slow down your graphics program.

Modern computers have millions of bytes of **RAM**, which is the **R**andom **A**ccess **M**emory that stores data during computer applications. One part of the **RAM** is the *video memory*. The video memory contains numerical values which are color codes. The location in the video memory corresponds to a location on the monitor. The first value at the start of the video memory corresponds to the (0,0) monitor location. In other words, if the first value in the video memory is **4** and color value **4** means **red** then the pixel on the monitor at coordinate (0,0) will be displayed red.

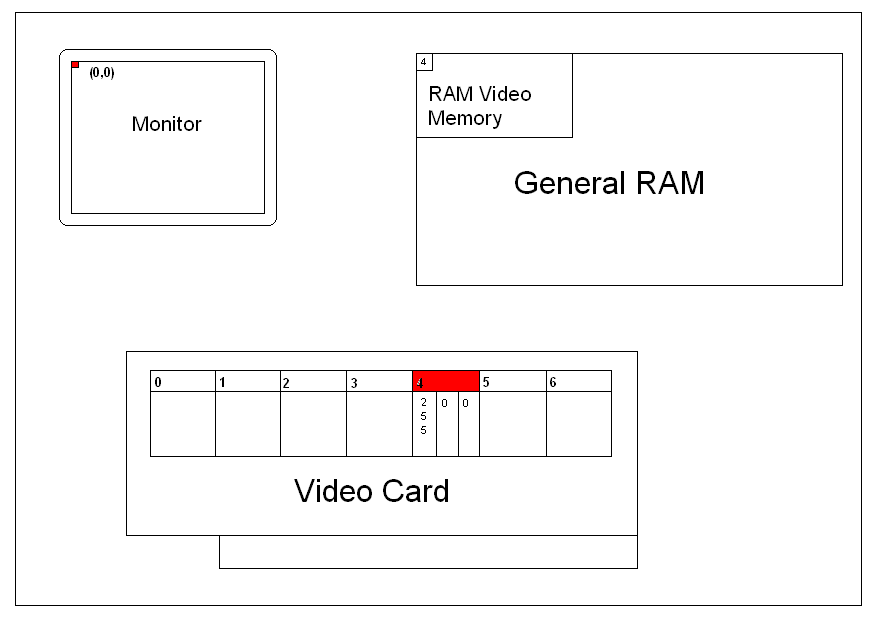
The video card is responsible for the available colors. A color shade is formed with a combination of red, green and blue values in the range of 0..255. You have seen these RGB values used with the **Color** class. Today there are many video cards with all kinds of capabilities, but let us keep it simple. Imagine that the video card has a bunch of color code boxes. Box code number **4** contains the RGB value of (**255,0,0**) which is bright red. This gives us the following sequence of operations illustrated by figure 20.29.

▪ Some method call places the color code 4 in the video memory.

▪ The video card identifies that code 4 is bright red.

▪ A bright-red pixel is now displayed on the monitor at coordinate (0,0).

**Figure 20.29**



Please realize that this picture in figure 20.29 is very rough and totally out of scale. The pixel at (0,0) should be smaller. The memory location in the RAM and the video card should also be much smaller. Furthermore, there are no values of **4** or **255** stored anywhere in memory. In reality there are only bits with the binary codes of **1**s and **0**s. This is a symbolic representation to help explain the interaction between the monitor, memory and video card to produce a picture.

For you, the graphics programmer, this means that you need to manipulate the video memory in such a way that it displays desired images. Still images are not a challenge if you ignore the creative talents required. The real challenge comes with animation. The next section will explain some special tricks that can be used to create animation.

**20.7 Virtual Memory & Video Buffering**

You have seen several early graphics programs that introduced animation. These programs used a simple animation approach that is called *draw and erase*. The concept is quite simple. You draw a picture, wait a short time and then erase the picture, followed by a picture in a new location. In this section, *draw and erase* will be repeated primarily to demonstrate that there are some serious problems with this approach.

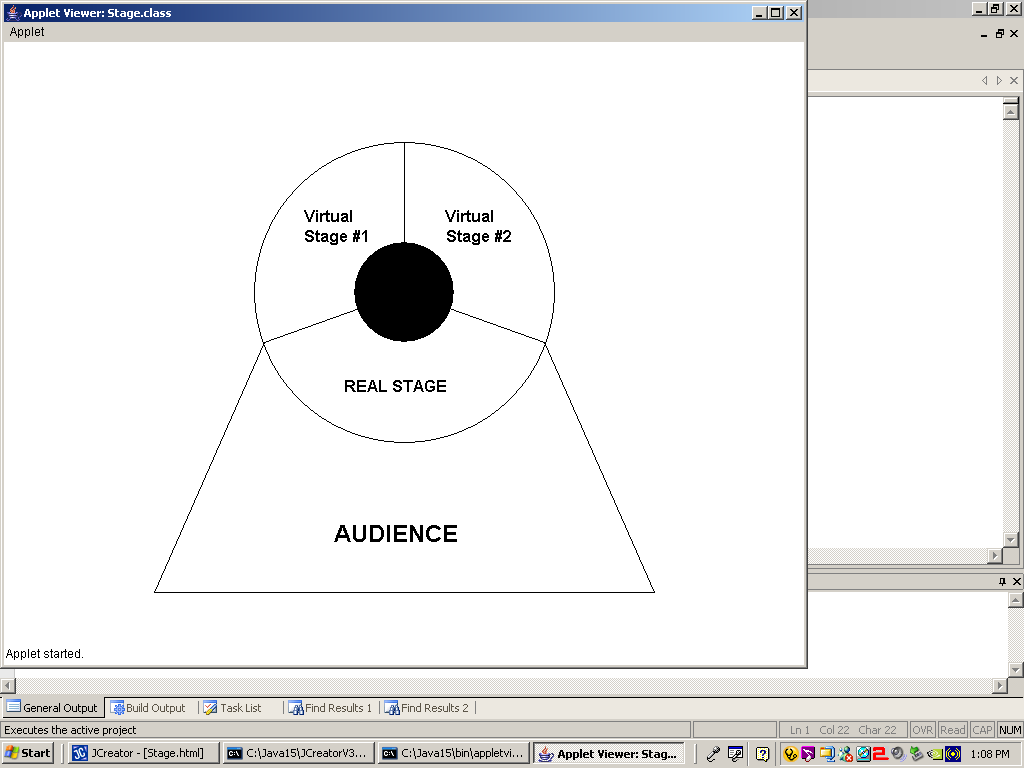
This chapter is titled *Advanced Graphics* and that certainly means looking at animation in a more advanced manner using virtual memory and buffer techniques. Right now you do not know virtual memory, from page flipping from double buffering from mushroom pizza. This is fine, because we will go slowly. For starters let us go to a theater and observe how the stage is prepared.

One approach is to use a single stage with a curtain. Act One is done and the curtain drops. Tech crews work furiously to remove the props from Act One and replace them with the new props needed for Act Two. The curtain raises and there is a new stage. This approach is not bad as long at the stage appearances between Act One and Act Two are not too drastically different.

There is a better technique that requires a large rotating stage and yes a bunch more money. Now we use a circular stage divided up into three segments like slices of a pizza. One-third of this stage is facing the audience. The stage facing the audience is the real stage that displays the props needed for the current Act. On the other side are two additional, virtual stages, called Virtual Stage #1 and Virtual Stage #2. These stages are called virtual, because they are not real viewable stages, at least not at this moment. However, when Act One is finished, the curtain drops and with the push of a button the stage rotates and Virtual Stage #1 now becomes the real stage.

The efficiency of the rotating stage is achieved by the fact that crews can work on the virtual stage while the real stage is being used. When the time comes, the stage rotates and very rapidly the next act is ready. Figure 20.30 shows a picture of such a stage configuration.

**Figure 20.30**



You find this stage talk fascinating. Your general education is improving and you are eager to discuss virtual stage techniques at dinner tonight. However, you were under the impression that this course was titled *Computer Science* and the topic at hand is *Graphics Animation*. You are correct and now the stage (pun intended) is set for you to understand virtual memory better.

In the last section you learned that color codes stored in the video memory correspond to locations on the monitor. Anytime you change a value in the video memory, you change the appearance of the monitor display.

Now imagine the following. Memory is memory and memory stores values. You can grab a block of memory and make it equivalent to the memory required for the video display. Right now ignore how this is all accomplished. In this block of memory, called *Virtual Memory*, you place values as if this were the real video memory. While you are putting values in the virtual memory the monitor is busy displaying values that are stored in video memory.

At the correct moment you dump, literally in electronic fashion instantaneously dump, all the values stored by the virtual memory into the video memory. This is like rotating the stage and you will instantly change the monitor display.

This technique is also called page flipping. Consider reading a book. The book contains many pages. Only the pages that are currently exposed to your eyes are readable. All other pages are hidden until you flip the page and now two new pages are ready for reading.

Certainly there must be more ways to label this technique, which is also called double buffering, but that name will not make any sense until we get to the next section and look at the Java classes, methods and program statements necessary to achieve the "page flipping" technique.

Do remember that some early animation techniques will be reviewed to help illustrate some fundamental problems with *draw and erase* animation. I think you will like this type of animation and you have taken the first step to learning how to create your own video game. Do not get too excited, because there are other programming techniques that you must master before you are ready for the video game big time, but you are on track and do not be surprised if you create a very nice video game in the future.

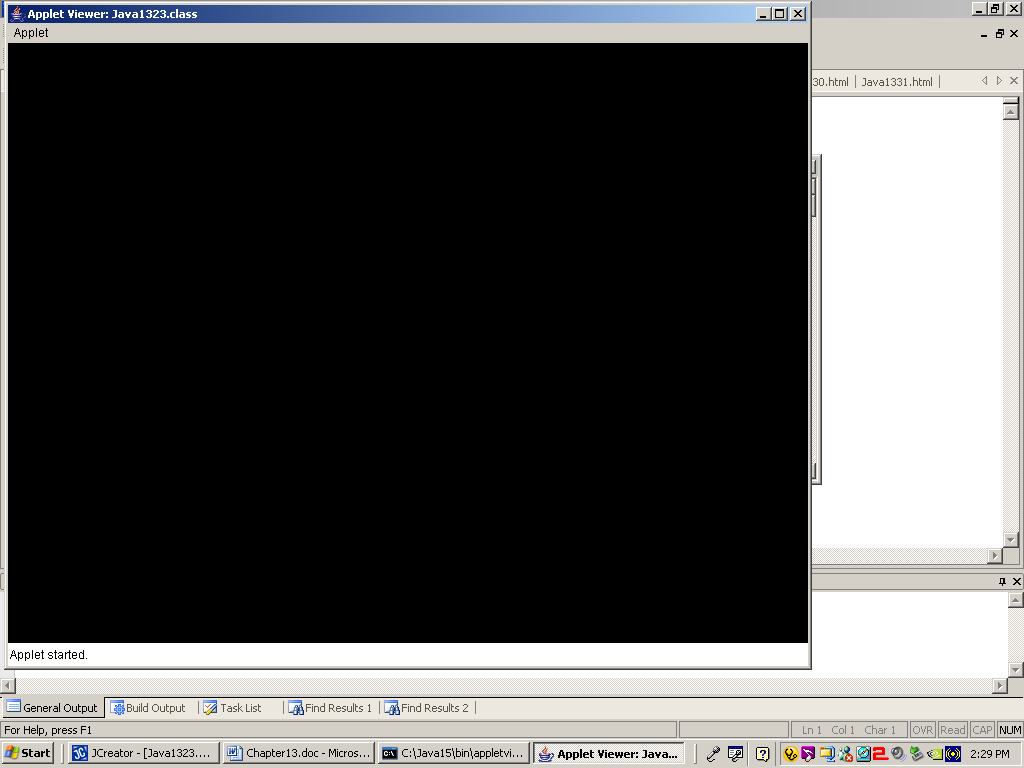
**20.8 Buffering Program Examples**

This has been a chapter with lots of information and we are not done yet. However, this is the last section and the end is in sight. Hopefully, the previous two sections of graphics and animation theory will help to clarify the coming program examples.

Program **Java2020.java**, in figure 20.31, starts nice and simple by creating an applet that displays a black background. There is nothing new or complicated introduced here.

**Figure 20.31**

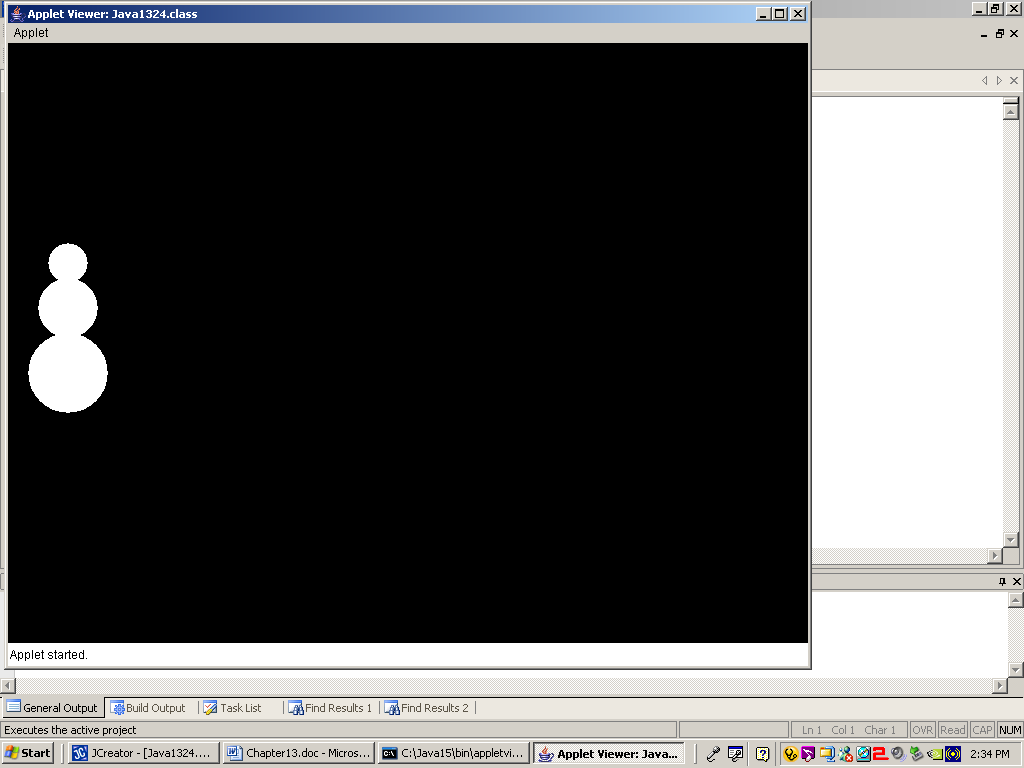
|  |
| --- |
| // Java2020.java  // This program creates an applet window with a black background.  import java.awt.\*;  import java.applet.Applet;  public class Java2023 extends Applet  {  int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  }      public void paint(Graphics g)  {  g.setColor(Color.BLACK);  g.fillRect(0,0,appletWidth,appletHeight);  }  } |



Program **Java2024.java**, in figure 20.32, adds a **drawSnowman** method to the previous program. A lovely snowman with three white snowballs is now drawn on the black background of the applet.

**Figure 20.32**

|  |
| --- |
| // Java2024.java  // This program introduces the <drawSnowman> method.  import java.awt.\*;  import java.applet.\*;  public class Java2024 extends Applet  {  int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  }    public void paint(Graphics g)  {  g.setColor(Color.BLACK);  g.fillRect(0,0,appletWidth,appletHeight);  drawSnowman(g,20,200);  }    public void drawSnowman(Graphics g, int x, int y)  {  g.setColor(Color.WHITE);  g.fillOval(x+20,y,40,40);  g.fillOval(x+10,y+35,60,60);  g.fillOval(x,y+90,80,80);  }  } |



Let us see if we can make the snowman move. Program **Java2025.java**, in figure 20.33, adds an **eraseSnowman** method. The program intentionally makes the *draw and erase*mistake of going too fast. There is no delay and you see nothing.

**Figure 20.33**

|  |
| --- |
| // Java2025.java  // This program adds the <eraseSnowman> method to the <drawSnowman> method.  // The snowman is not visible because it is erased as quickly as it is drawn.  import java.awt.\*;  import java.applet.\*;  public class Java2025 extends Applet  {    int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  }    public void paint(Graphics g)  {  g.setColor(Color.BLACK);  g.fillRect(0,0,appletWidth,appletHeight);  for (int x = 20; x < 700; x += 100)  {  drawSnowman(g,x,200);  eraseSnowman(g,x,200);  }  }    public void drawSnowman(Graphics g, int x, int y)  {  g.setColor(Color.WHITE);  g.fillOval(x+20,y,40,40);  g.fillOval(x+10,y+35,60,60);  g.fillOval(x,y+90,80,80);  }  public void eraseSnowman(Graphics g, int x, int y)  {  g.setColor(Color.BLACK);  g.fillRect(x,y,80,170);  }  } |

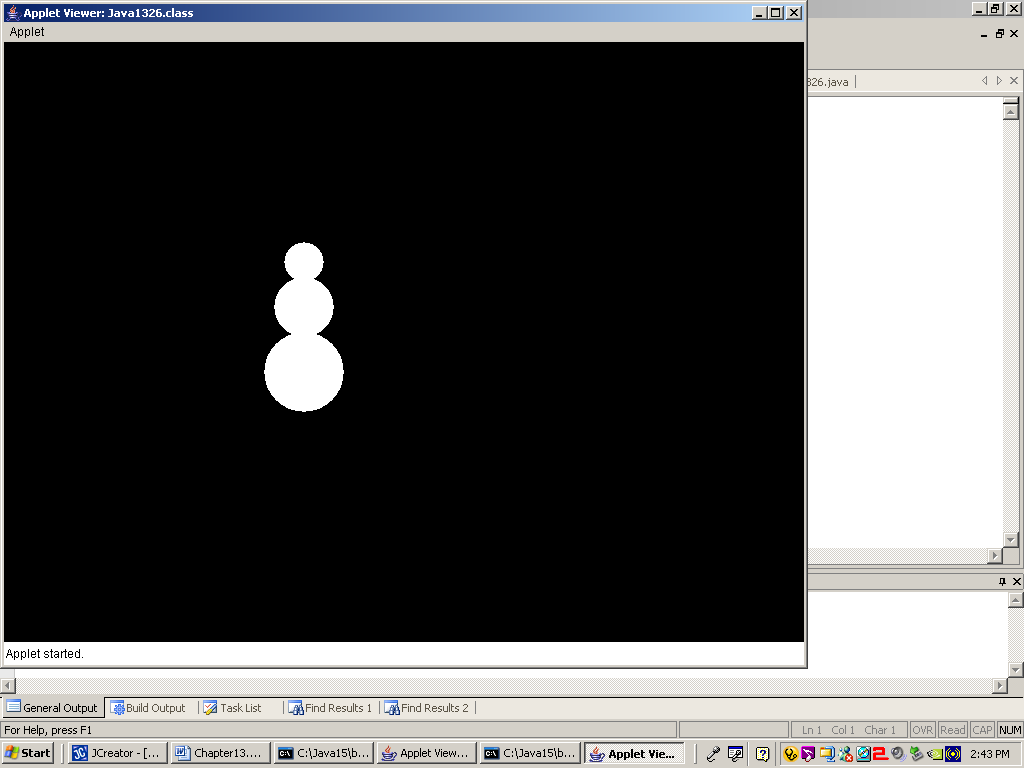
The output of program **Java2025.java** is strictly a black applet window. If you had fast enough eyes you would see a white snowman zooming along the screen, but human eyes cannot handle such a speed and the snowman simply cannot not be seen anywhere.

Program **Java2026.java**, in figure 20.34, adds a neat **delay** method. This method uses the **System.currentTimeMillis** method to measure milli-seconds. This time you will be able to see the snowman move before it disappears at the end.

**Figure 20.34**

|  |
| --- |
| // Java2026.java  // This program adds a <delay> method, which makes it possible  // to see the snowman movement.  import java.awt.\*;  import java.applet.\*;  public class Java2026 extends Applet  {    int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  }    public void paint(Graphics g)  {  g.setColor(Color.BLACK);  g.fillRect(0,0,appletWidth,appletHeight);  for (int x = 20; x < 700; x += 10)  {  drawSnowman(g,x,200);  delay(50);  eraseSnowman(g,x,200);  }  }    public void drawSnowman(Graphics g, int x, int y)  {  g.setColor(Color.WHITE);  g.fillOval(x+20,y,40,40);  g.fillOval(x+10,y+35,60,60);  g.fillOval(x,y+90,80,80);  }  public void eraseSnowman(Graphics g, int x, int y)  {  g.setColor(Color.BLACK);  g.fillRect(x,y,80,170);  }    public void delay(int n)  {  long startDelay = System.currentTimeMillis();  long endDelay = 0;  while (endDelay - startDelay < n)  endDelay = System.currentTimeMillis();  }    } |

**Figure 20.34 Continued**

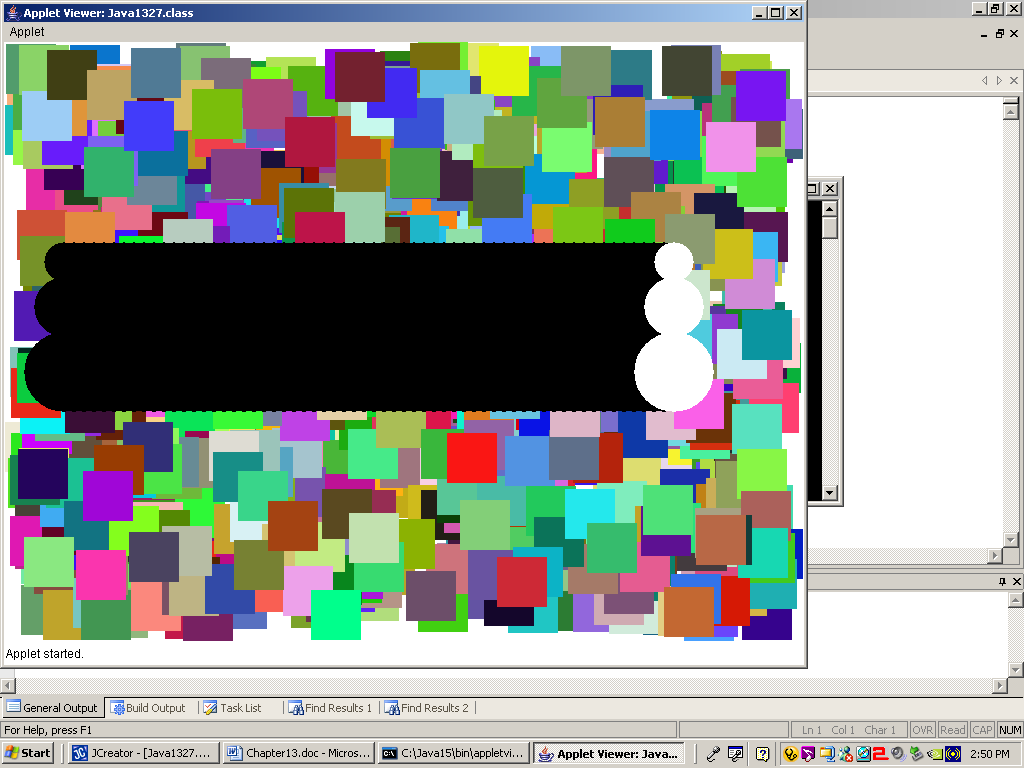
****

The *draw and erase* approach has some serious limitations that will be shown by program **Java2027.java**, in figure 20.35. The moving snowman, which moved so nicely - if you ignore the flicker - in the last program, is used again. This time a multi-colored background of random squares provides the background. You will now detect a serious flaw. The moving snowman chews up the background.

**Figure 20.35**

|  |
| --- |
| // Java2027.java  // This program demonstrates that the "draw and erase" animation approach  // does not work on a multi-colored background.  import java.awt.\*;  import java.applet.\*;  import java.util.\*;  public class Java2027 extends Applet  {  int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  }    public void paint(Graphics g)  {  g.setColor(Color.BLACK);  createBackGround(g);  for (int x = 20; x < 700; x += 10)  {  drawSnowman(g,x,200);  delay(50);  eraseSnowman(g,x,200);  }  }    public void createBackGround(Graphics g)  {  Random rnd = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int rndX = rnd.nextInt(750);  int rndY = rnd.nextInt(550);  g.setColor(new Color(rnd.nextInt(256),rnd.nextInt(256),rnd.nextInt(256)));  g.fillRect(rndX,rndY,50,50);  }  }    public void drawSnowman(Graphics g, int x, int y)  {  g.setColor(Color.WHITE);  g.fillOval(x+20,y,40,40);  g.fillOval(x+10,y+35,60,60);  g.fillOval(x,y+90,80,80);  }  public void eraseSnowman(Graphics g, int x, int y)  {  g.setColor(Color.BLACK);  g.fillOval(x+20,y,40,40);  g.fillOval(x+10,y+35,60,60);  g.fillOval(x,y+90,80,80);  }    public void delay(int n)  {  long startDelay = System.currentTimeMillis();  long endDelay = 0;  while (endDelay - startDelay < n)  endDelay = System.currentTimeMillis();  }  } |

**Figure 20.35 Continued**

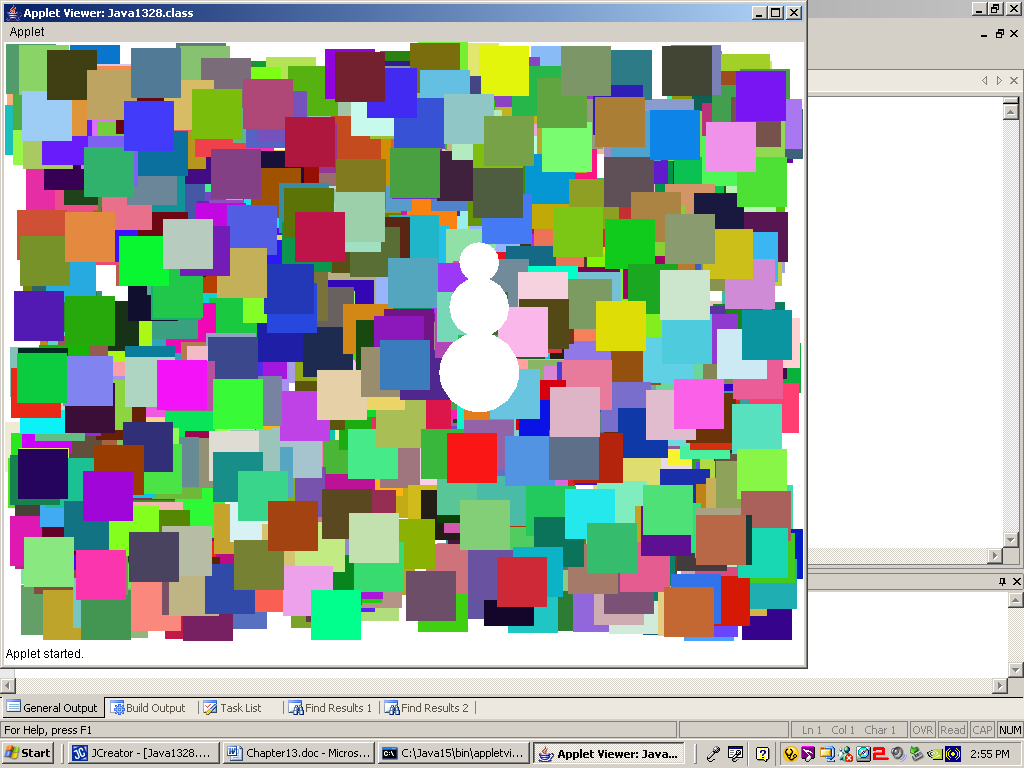


Program **Java2028.java**, in figure 20.36, cures the erase problem of the previous program example. The problem is not solved with a simple set of new steps. There are some serious new goodies presented in this program. Take a quick glance at code and then execute the program to observe the difference. After that it is time to carefully take this program apart into small pieces.

**Figure 20.36**

|  |
| --- |
| // Java2028.java  // This program cures the animation problem on multi-colored backgrounds with  // virtual memory, buffer methods.  import java.awt.\*;  import java.applet.\*;  import java.util.\*;  public class Java2028 extends Applet  {    int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    Image virtualMem;  Graphics gBuffer;    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  virtualMem = createImage(appletWidth,appletHeight);  gBuffer = virtualMem.getGraphics();  }    public void paint(Graphics g)  {  for (int x = 20; x < 700; x += 5)  {  drawSnowman(g,x,200);  delay(10);  }  }    public void drawSnowman(Graphics g, int x, int y)  {  createBackGround();  gBuffer.setColor(Color.WHITE);  gBuffer.fillOval(x+20,y,40,40);  gBuffer.fillOval(x+10,y+35,60,60);  gBuffer.fillOval(x,y+90,80,80);  g.drawImage (virtualMem,0,0, this);  }    public void createBackGround()  {  Random rnd = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int rndX = rnd.nextInt(750);  int rndY = rnd.nextInt(550);  gBuffer.setColor(new Color(rnd.nextInt(256),rnd.nextInt(256),rnd.nextInt(256)));  gBuffer.fillRect(rndX,rndY,50,50);  }  }    public void delay(int n)  {  long startDelay = System.currentTimeMillis();  long endDelay = 0;  while (endDelay - startDelay < n)  endDelay = System.currentTimeMillis();  }    } |

**Figure 20.36 Continued**

****

You have now arrived at the first program that uses *page flipping*, *virtual memory* or *double buffer* techniques. Get used to the three names, because you will hear all three terms used interchangeably.

Figure 20.37 starts with the **init** method. You will see a new class used, which is the **Image** class. The **Image** class lets us manipulate computer memory by setting up a block of **RAM** for virtual memory. The memory that is needed will be based on the size of the applet window. We will use the window width and height to construct an **Image** object, called **virtualMem**. Take a look at the sequence.

**Figure 20.37**

|  |
| --- |
| int appletWidth; // width of the Applet window  int appletHeight; // height of the Applet window    Image virtualMem;  Graphics gBuffer;    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  virtualMem = createImage(appletWidth,appletHeight);  gBuffer = virtualMem.getGraphics();  } |

For readability variable **appletWidth** and **appletHeight** are declared. These two variables get values from methods **getWidth** and **getHeight**.

With the applet window dimensions in our pocket we can construct the new object of the **Image** class, called **virtualMem** using the **createImage** method along with parameters **appletWidth** and **appletHeight**. We now have a block of virtual memory set aside that is equal to the number of pixels in our applet window.

Computer information frequently needs to transfer from one location to another and wait in a temporary holding place. At these transfer places there is also the opportunity to change formats. For instance a print buffer stores print job information and changes RAM code to **PCL** or **P**rinter **C**ontrol **L**anguage. There are keyboard buffers, file buffers and video buffers.

Normally, when the regular video memory is used, only a single buffer is used. You do not even realize that a buffer is anywhere in sight. Inside the **paint** method you make a call, like **g.drawLine(x1,y1,x2,y2);** and a line is drawn. The **g** object transfers information to the video display via the video buffer. This direct approach is not called *single buffering*, but it would be an appropriate name.

When virtual memory is used, there exists a second **Graphics** object, which transfers information to the virtual video display via a second buffer. In our program this second object is called **gBuffer**. You must carefully pay attention when information is handled by the **g** object and when it is handled by the **gBuffer** object.

The last statement in the **init** method, **gBuffer = virtualMem.getGraphics();**, creates a graphics context for drawing to an off-screen image. This means that any method used with **gBuffer** will send information to the virtual memory.

The next method to place under the microscope is the **paint** method, which is shown in figure 20.38. The **paint** method looks surprisingly simple. It contains a loop that repeats in such a way that a snowman is drawn every 5 pixels. There is also a call to the **delay** method. Missing is any evidence of an **eraseSnowman** method. This should not be a surprise. You are now in virtual memory land and the old *draw and erase* is no longer acceptable.

**Figure 20.38**

|  |
| --- |
| public void paint(Graphics g)  {  for (int x = 20; x < 700; x += 5)  {  drawSnowman(g,x,200);  delay(10);  }  } |

Figure 20.39 shows two **createBackGround** methods. The first method is the one used by the current program, **Java2028.java**. The second method was used in the earlier **Java2027.java** program. Do you understand the difference?

**Figure 20.39**

|  |
| --- |
| public void createBackGround()  {  Random rnd = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int rndX = rnd.nextInt(750);  int rndY = rnd.nextInt(550);  gBuffer.setColor(new Color(rnd.nextInt(256),rnd.nextInt(256),rnd.nextInt(256)));  gBuffer.fillRect(rndX,rndY,50,50);  }  } |
| public void createBackGround(Graphics g)  {  Random rnd = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int rndX = rnd.nextInt(750);  int rndY = rnd.nextInt(550);  g.setColor(new Color(rnd.nextInt(256),rnd.nextInt(256),rnd.nextInt(256)));  g.fillRect(rndX,rndY,50,50);  }  } |

The older **createBackground** method used a **g Graphics** object. This means that all graphics information is transferred to the video memory for immediate display to the memory. The newer **createBackGround** method uses the **gBuffer** object, which results in all graphics information going to the virtual memory block.

Now we are ready for the finale with the **drawSnowman** method. The method starts by calling **createBackGround**, which was just explained. Four method calls in sequence use **gBuffer** and all four methods are used to draw a snowman. In other words, after the background is drawn to the virtual memory, it is followed by drawing a snowman to the same virtual memory.

The last program statement, **g.drawImage**, goes directly to the video display with the **g** object. Method **drawImage** takes the contents of the **virtualMem** and deposits this in the video memory starting at location (0,0). The last parameter, **this**, will not be explained in this course. For now, use **this**, and do not worry about the meaning of parameter **this**.

**Figure 20.40**

|  |
| --- |
| public void drawSnowman(Graphics g, int x, int y)  {  createBackGround();  gBuffer.setColor(Color.WHITE);  gBuffer.fillOval(x+20,y,40,40);  gBuffer.fillOval(x+10,y+35,60,60);  gBuffer.fillOval(x,y+90,80,80);  g.drawImage (virtualMem,0,0, this);  } |

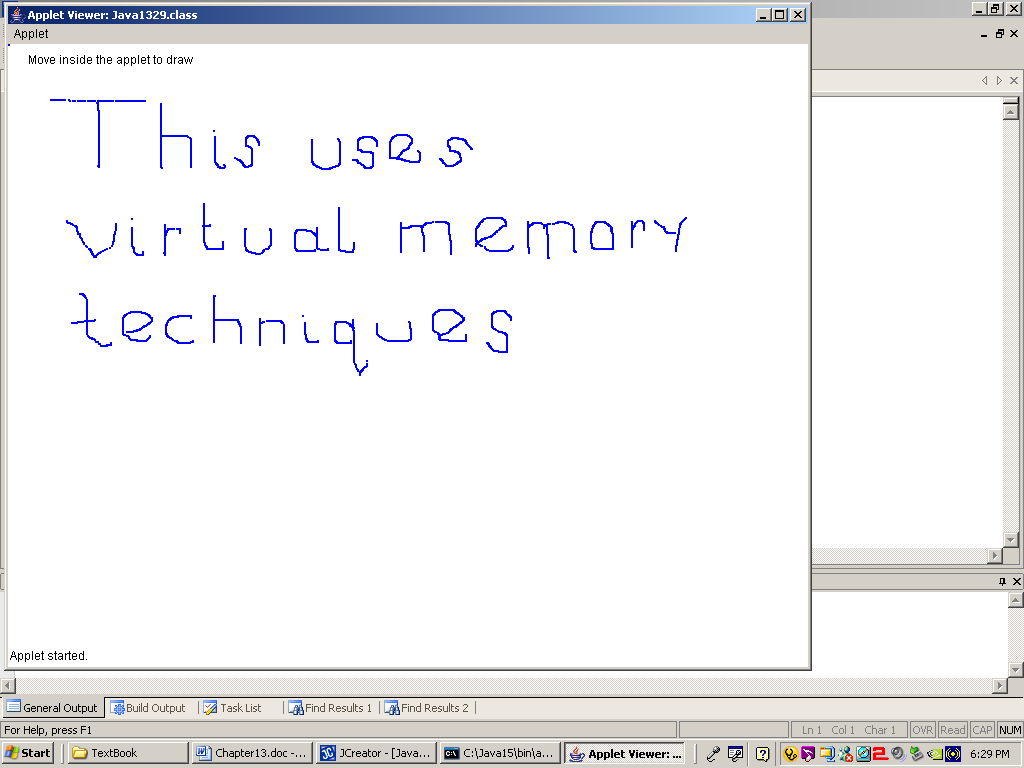
The next program is easy to explain, with one little assumption that you understood the explanation of the last program. It may be necessary to read the buffer-business-graphics a number of times and allow it to soak in. Program **Java2029.java**, in figure 20.41, returns to the mouse draw program. It behaves in a manner, which is very similar to the mouse draw program earlier in this chapter.

This time we operate in double buffer land and that means there are no arrays to store information. All graphics information is stored in virtual memory and each time the **mouseMove** method is triggered, method **paint** is called. Method **paint** starts by updating the virtual memory with the latest mouse information using **gBuffer**. Like the earlier **drawSnowman** method, the last statement in **paint** calls method **drawImage** and the virtual image display is now flipped to the real video display.

**Figure 20.41**

|  |
| --- |
| // Java2029.java  // This program uses virtual memory for a simple draw program.  // The program will flicker while drawing.  import java.applet.Applet;  import java.awt.\*;    public class Java2029 extends Applet  {    Image virtualMem;  Graphics gBuffer;  int oldX, oldY, newX, newY;  int appletWidth;  int appletHeight;    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  virtualMem = createImage(appletWidth,appletHeight);  gBuffer = virtualMem.getGraphics();  gBuffer.setColor(Color.white);  gBuffer.fillRect(0,0,appletWidth,appletHeight);  }    public void paint(Graphics g)  {  gBuffer.setColor(Color.black);  gBuffer.drawString("Move inside the applet to draw", 20,20);  gBuffer.setColor(Color.blue);  gBuffer.fillRect(oldX,oldY,2,2);  g.drawImage(virtualMem,0,0,this);  }    public boolean mouseDown(Event e, int x, int y)  {  newX = x;  newY = y;  oldX = newX;  oldY = newY;  repaint();  return true;  }    public boolean mouseDrag(Event e, int x, int y)  {  newX = x;  newY = y;  oldX = newX;  oldY = newY;  repaint();  return true;  }    } |

**Figure 20.41 Continued**

****

This virtual memory business may be pretty advanced, but you know the bottom line, and the reality persists that the draw program flickers in an annoying manner. You had probably hoped that the buffer business would cure the flicker. Program **Java2030.java**, in figure 20.42, will cure the flicker stuff by adding a small little method, called **update**. Method **update** will be explained in the next section. Right now I want you to be satisfied that the flicker is finally gone and it appears that you have a program that operates properly.

**Figure 20.42**

|  |
| --- |
| // Java2030.java  // This program adds the <update> methods which stabilizes the flicker of the  // Java2022.java and Java2029.java programs.  import java.applet.Applet;  import java.awt.\*;    public class Java2030 extends Applet  {    Image virtualMem;  Graphics gBuffer;  int oldX, oldY, newX, newY;  int appletWidth;  int appletHeight;    public void init()  {  appletWidth = getWidth();  appletHeight = getHeight();  virtualMem = createImage(appletWidth,appletHeight);  gBuffer = virtualMem.getGraphics();  gBuffer.setColor(Color.white);  gBuffer.fillRect(0,0,appletWidth,appletHeight);  }    public void paint(Graphics g)  {  gBuffer.setColor(Color.black);  gBuffer.drawString("Move inside the applet to draw", 20,20);  gBuffer.setColor(Color.blue);  gBuffer.drawLine(oldX,oldY,newX,newY);  g.drawImage(virtualMem,0,0,this);  }    public boolean mouseDown(Event e, int x, int y)  {  newX = x;  newY = y;  oldX = newX;  oldY = newY;  repaint();  return true;  }    public boolean mouseDrag(Event e, int x, int y)  {  newX = x;  newY = y;  oldX = newX;  oldY = newY;  repaint();  return true;  }    **public void update(Graphics g)**  **{**  **paint**  **(g);**  **}**    } |

There is no program output sample with this program. The whole intention of the program is to demonstrate that the flicker has disappeared with the use of the **update** method. Only running the program will properly demonstrate that fact. On paper there is absolutely no difference between the last program and this latest program example.

**20.9 The Magic of Method update**

Every time that the **paint** method is called, the video displayed is completely cleared and the display is created from scratch. This continuous clearing and redrawing causes the annoying flicker. The flicker will pretty much disappear if any new drawings to the monitor are accomplished without first clearing the screen. In other words you want to *update* the display, not redraw everything.

This is the mission of method **update**. The method does appear a little mysterious, because it does not get called in any direct sense. Method **update** works together with method **repaint**. Method **repaint** arranges for method **paint** to be revisited. The presence of the **update** method alters the manner in which **paint** executes. Without **update**, **paint** clears the screen and redraws the entire video display. With **update**, **paint** only "updates" the video display.

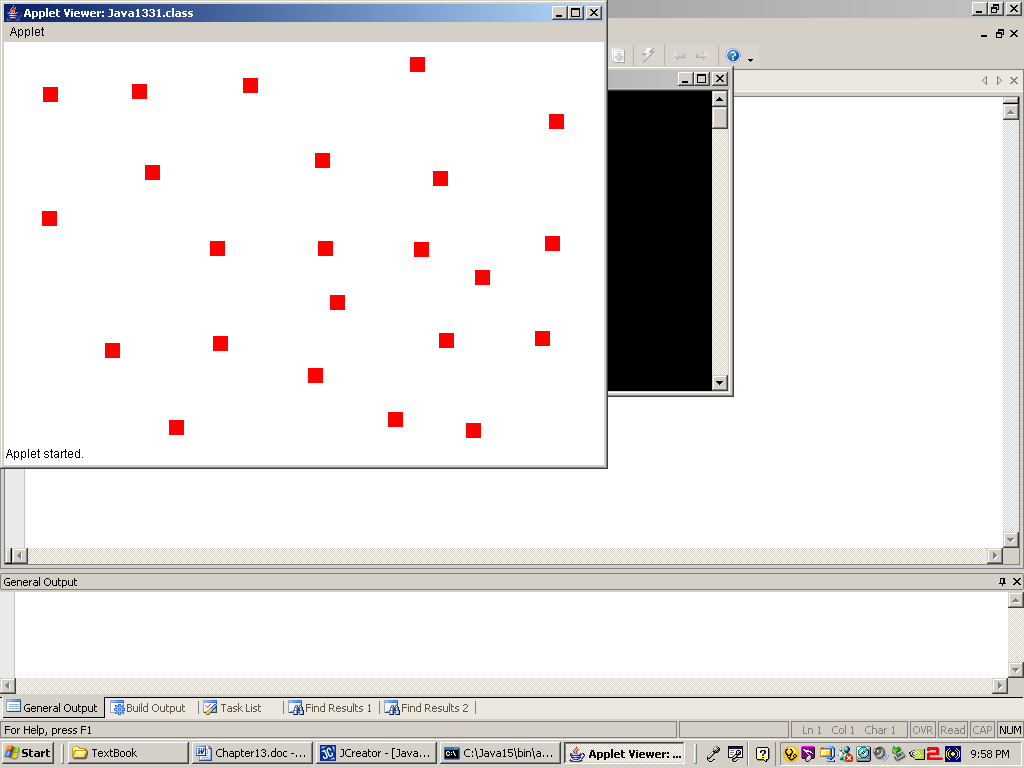
The function of **update** is demonstrated very well with program **Java2031.java**, in figure 20.43, which is a program presented earlier in the mouse routine section. It is the program that draws a small square each time the mouse is clicked. The earlier program showed only one small square at a time. The display of any new square erased the display of the previous square. This "single square" program was followed by a similar program that used an array to store coordinates for each one of the prior squares. This altered program did repaint the display every time, but the array of coordinates guaranteed that all previous squares were drawn. The result was an illusion that the squares never left.

Now program **Java2031.java** will achieve the same result. I mean the result of not erasing prior squares, but this time there is no array in sight. This time the modest **update** method accomplishes the desired result. Since **update** alters the paint method to not clear the screen, but only add new displays, the previous squares are preserved.

**Figure 20.43**

|  |
| --- |
| // Java2031.java  // This program draws small squares using the <update> method rather than  // using arrays to store coordinate values.  import java.applet.Applet;  import java.awt.\*;  public class Java2031 extends Applet  {  int xCoord;  int yCoord;  boolean firstPaint;      public void init()  {  firstPaint = true;  }    public void paint(Graphics g)  {  if (firstPaint)  firstPaint = false;  else  {  g.setColor(Color.red);  g.fillRect(xCoord,yCoord,15,15);  }  }    public boolean mouseDown(Event e, int x, int y)  {  xCoord = x;  yCoord = y;  repaint();  return true;  }    public void update(Graphics g)  {  paint(g);  }  } |

**Figure 20.43 Continued**



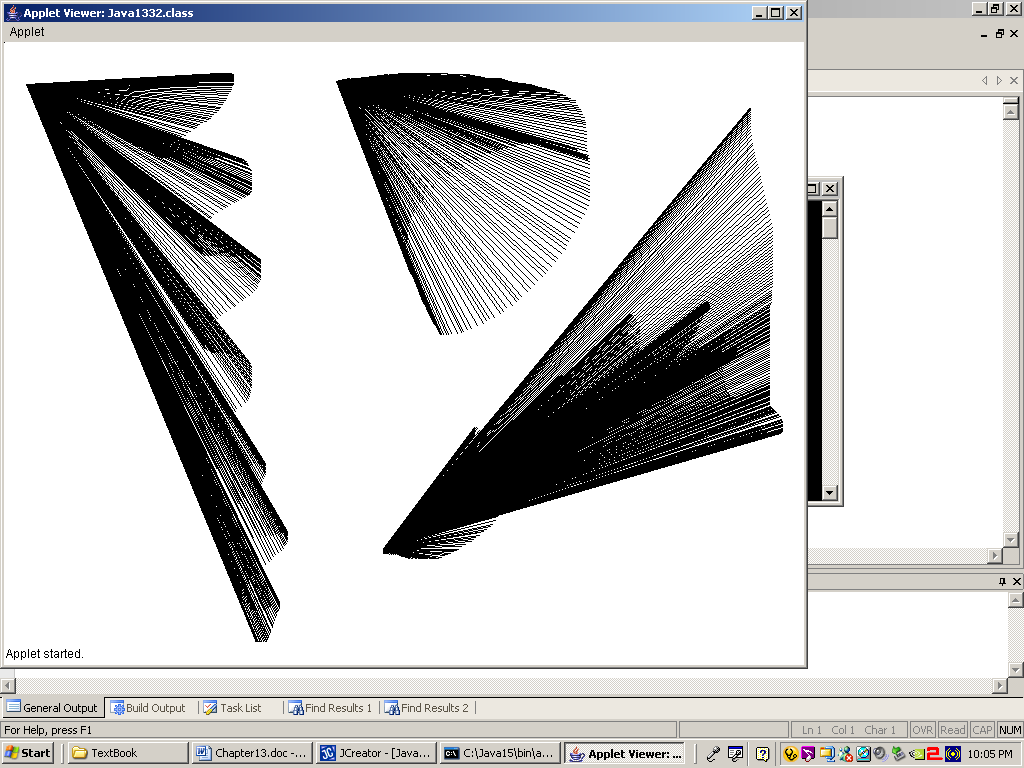
Now you might wonder why there were program examples with arrays. It appears that using method **update** does the job of preserving the monitor display very nicely without the complexity of storing information in an array. That is true for a program like **Java2031.java**, but it is possible to preserve too much.

Consider program **Java2032.java**, in figure 20.44, which is the "rubber band" straight line program revisited. Observe what happens when method **update** is attached to this program. You may find the program output interesting, but it is not exactly what you might desire.

**Figure 20.44**

|  |
| --- |
| // Java2032.java  // This program revisits the earlier Java2020.java program using the <update> method.  // It shows that <update> can cause an unwanted side effect.  import java.applet.Applet;  import java.awt.\*;  public class Java2032 extends Applet  {    int startX,startY,endX,endY;    public void paint(Graphics g)  {  g.drawLine(startX,startY,endX,endY);  }  public boolean mouseDown(Event e, int x, int y)  {  startX = x;  startY = y;  return true;  }    public boolean mouseDrag(Event e, int x, int y)  {  endX = x;  endY = y;  repaint();  return true;  }    public void update(Graphics g)  {  paint(g);  }    } |

**Figure 20.44 Continued**

****

The use of the double buffering graphics approach solves the problem of stuffing multiple arrays with information that is drawn again and again. It is now possible to keep a copy of the previous graphics display neatly hidden in virtual memory. At the right moment this virtual memory moves front center to the real video memory and the monitor display changes. Using virtual memory combined with the **update** method makes some impressive graphics animation possible.

**20.10 Importing Images**

At some point in the near future you will be working on a graphics project in this class. The vast majority of output in any graphics project is expected to be created by the Java program with **Graphics** methods like **drawLine**, **drawRect**, and **drawOval**. Sometimes it is nice to add a little personalizing touch to a project. This is similar to what my son does with the *PowerPoint* slides he creates for *Exposure Java*. You have probably noticed that the left picture from figure 20.45 is on the first slide of every *PowerPoint* presentation. The picture on the right was created by a former student and will be used in the next program.

**Figure 20.45**

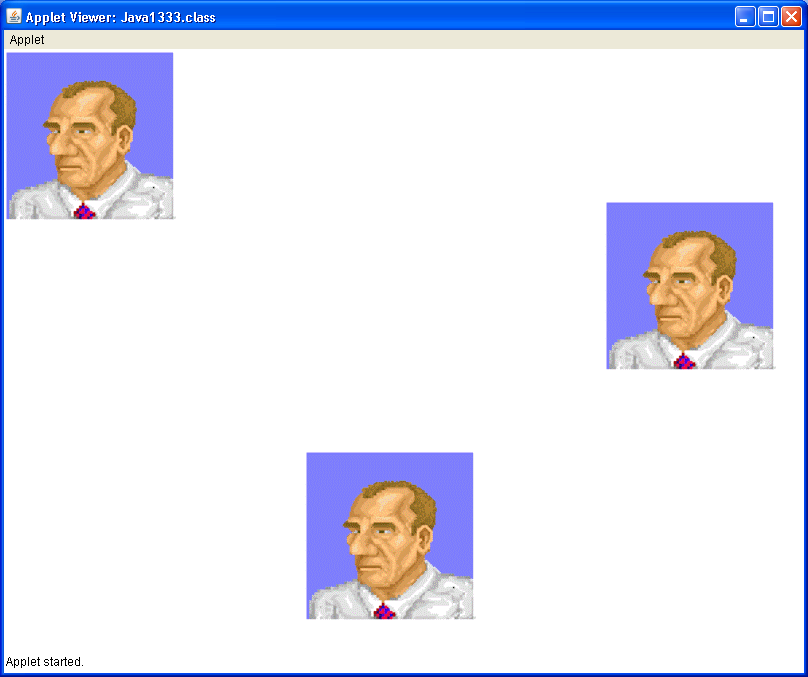
 

Pictures like these can be imported into a Java program. Before I show you how to do this, remember to keep one thing in mind. In a graphics project, the vast majority of output is expected to be created with **Graphics** methods. Imported images should be used sparingly. Program **Java2033.java** in figure 20.46 shows how an image file can be imported and displayed at any location on the screen.

**Figure 20.46**

|  |
| --- |
| // Java2033.java  // This program demonstrates how to display a graphics image from a gif file  // at any x,y location on the screen.  import java.awt.\*;  public class Java2033 extends java.applet.Applet  {  Image picture;    public void init()  {  picture = getImage(getDocumentBase(),"LSchram.gif");  }    public void paint(Graphics g)  {  g.drawImage(picture,0,0,this);  g.drawImage(picture,600,150,this);  g.drawImage(picture,300,400,this);  }  } |

**Figure 20.46 Continued**

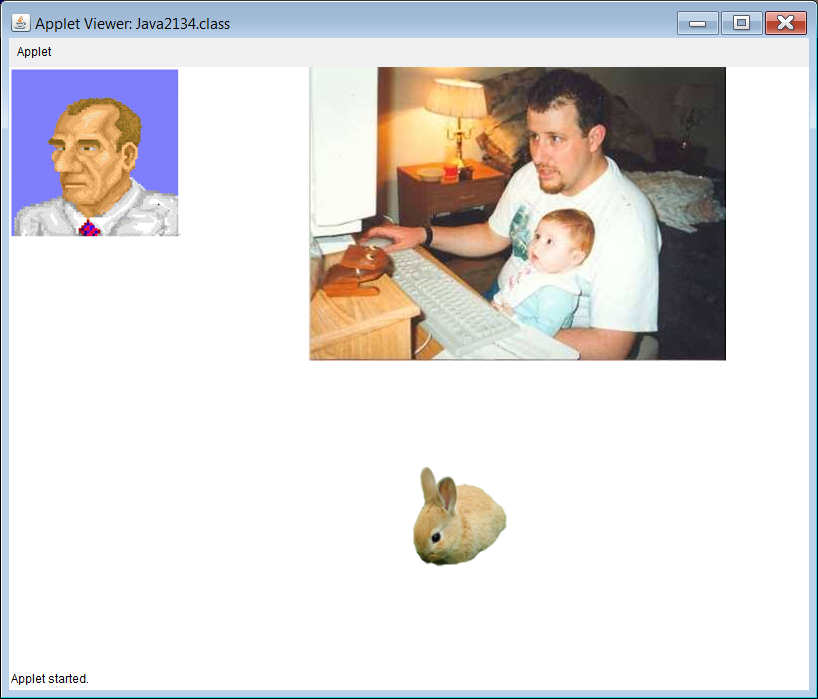
****

OK, we know we can display **gif** images, but what about **jpg** (jpeg) or **bmp** (bitmap) images or **png** files? Program **Java2034.java** in figure 20.47 examines this by trying to display a **gif**, a **jpg**, a **png** and a **bmp** image.

**Figure 20.47**

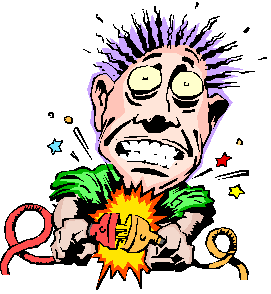
|  |
| --- |
| // Java2034.java  // This program demonstrates displaying different images of different types.  // It also shows that gif, jpg and png files can be displayed.  // The bmp file will not display.  import java.awt.\*;  public class Java2034 extends java.applet.Applet  {  Image picture1, picture2, picture3, picture4;  public void init()  {  picture1 = getImage(getDocumentBase(),"LSchram.gif");  picture2 = getImage(getDocumentBase(),"JSchram.jpg");  picture3 = getImage(getDocumentBase(),"ShortCircuit.bmp");  picture4 = getImage(getDocumentBase(),"bunny.png");  }  public void paint(Graphics g)  {  g.drawImage(picture1,0,0,this);  g.drawImage(picture2,300,0,this);  g.drawImage(picture3,50,300,this);  g.drawImage(picture4,400,400,this);  }  } |

**Figure 20.47 Continued**



The gif and the jpeg images were displayed. The bitmap was not. Remember that a Java applet is designed to execute as part of a web page. If you work with web pages, you learn quickly that only certain types of images can be display on a web page. Gifs, jpegs and png files can be displayed. Bitmaps cannot. Just in case you are curious about the bitmap file that was not displayed; it is shown in figure 20.48. Unlike Java programs, *Word* documents have no problems with bitmap images.

**Figure 20.48**

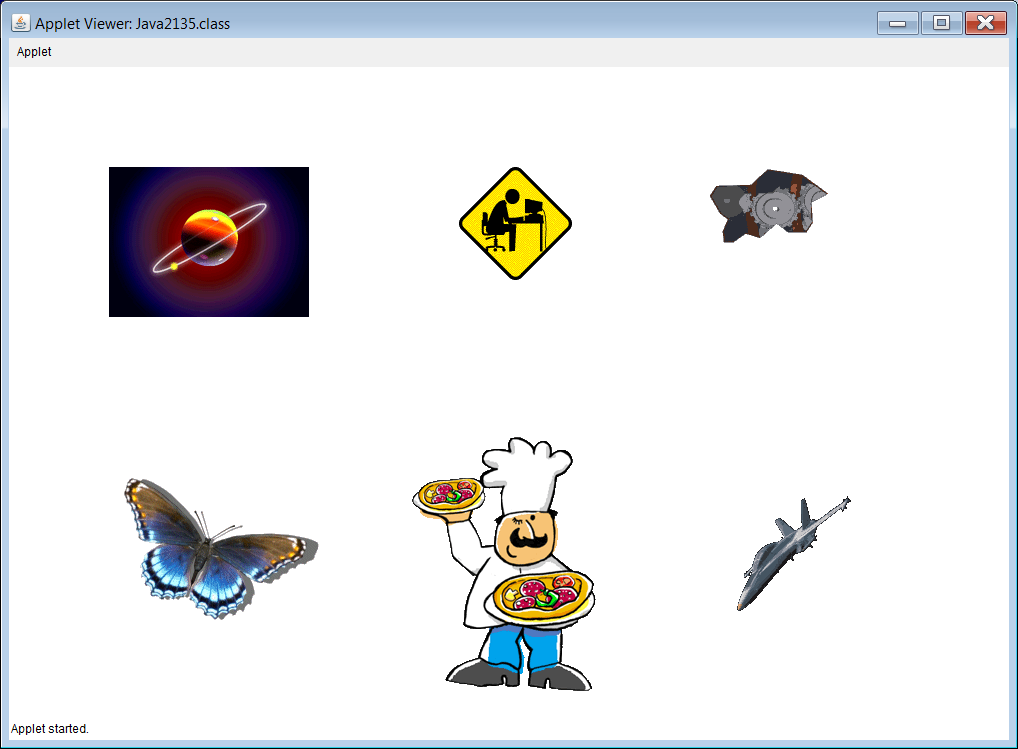


Since we know that gifs work, the next question might be, *"What about animated gifs?"* Good question. The best possible way to find the answer is to test it. This is exactly what is done in program **Java2035.java** in figure 20.49.

**Figure 20.49**

|  |
| --- |
| // Java2035.java  // This program demonstrates that animated gifs are inserted in the same way as  // normals gifs. Animated gifs can cause a flickering side-effect on the screen.  import java.awt.\*;  public class Java2035 extends java.applet.Applet  {  Image picture1, picture2, picture3, picture4, picture5, picture6;  public void init()  {  picture1 = getImage(getDocumentBase(),"space.gif");  picture2 = getImage(getDocumentBase(),"computer.gif");  picture3 = getImage(getDocumentBase(),"gears.gif");  picture4 = getImage(getDocumentBase(),"butterfly.gif");  picture5 = getImage(getDocumentBase(),"pizza.gif");  picture6 = getImage(getDocumentBase(),"jet.gif");  }  public void paint(Graphics g)  {  g.drawImage(picture1,100,100,this);  g.drawImage(picture2,450,100,this);  g.drawImage(picture3,700,100,this);  g.drawImage(picture4,100,400,this);  g.drawImage(picture5,400,350,this);  g.drawImage(picture6,700,400,this);  }  } |

**Figure 20.49 Continued**



Here is one example that shows that there is a limit to what can be shown in a *Word* document. Program **Java2035.java**, in figure 20.49, displays 6 animated gifs. All 6 animated gifs are displayed. However, the screenshot in figure 20.49 does not show the *animating* part of the animated gifs, nor does it show the annoying flickering effect that accompanies them. The flickering is being caused by the animated gifs themselves. Every time the gifs change their appearance, they cause the **paint** method to be called again. This causes everything on the screen to be erased and then redrawn, which causes the flicker.

Now, if you have been paying attention, you may remember the previous section titled *The Magic of Method Update*. Just having the **update** method in your program prevents the screen from erasing when the **paint**method is called. Will it work for animated gifs? Again, the best way to find this out is to test it, which is what is done in program **Java2036.java** in figure 20.50.

**Figure 20.50**

|  |
| --- |
| // Java2036.java  // This program attempts to cure the flickering caused by the ANIMATED gifs by using the update method.  // The output shows that for some animated gifs this will work, and for others it will make things worse.  // NOTE: Do not expect to get any "animation bonus points" for using ANIMATED gifs in any graphics project.  // You will lose points instead!  import java.awt.\*;  public class Java2036 extends java.applet.Applet  {  Image picture1, picture2, picture3, picture4, picture5, picture6;  public void init()  {  picture1 = getImage(getDocumentBase(),"space.gif");  picture2 = getImage(getDocumentBase(),"computer.gif");  picture3 = getImage(getDocumentBase(),"gears.gif");  picture4 = getImage(getDocumentBase(),"butterfly.gif");  picture5 = getImage(getDocumentBase(),"pizza.gif");  picture6 = getImage(getDocumentBase(),"jet.gif");  }  public void paint(Graphics g)  {  g.drawImage(picture1,100,100,this);  g.drawImage(picture2,450,100,this);  g.drawImage(picture3,700,100,this);  g.drawImage(picture4,100,400,this);  g.drawImage(picture5,400,350,this);  g.drawImage(picture6,700,400,this);  }  **public void update(Graphics g) { paint(g); }**  } |

A screen shot will not come close to showing what happens with this program. You need to execute it yourself and see what happens. What you should notice is that the flickering is gone. Hooray! However, while the **update** method works for some of the animated gifs, it will actually make mess up others.

|  |
| --- |
| **Animated Gif Warning** |
| Do not insult your teacher's intelligence by trying to pass off an animated gif as an animated graphics project!  Your teacher will be neither impressed, nor amused and this will be demonstrated in your grade. |

**20.11 Multi-Page Output**

When students work on graphics projects there are two questions I get on a regular basis:

1) *How do I import images?*

2) *How do I create multiple pages of graphics output?*

The previous section dealt with the first question. This one deals with the second. Creating multiple pages of output may sound simple, but Java makes things a little complicated with its **paint** method. Even if your **paint** method creates different pages in sequence, it will automatically start from the beginning whenever the **paint** method is called again, and this can happen just by clicking the mouse or dragging a window.

Program **Java2037.java** in figure 20.51 attempts to use a **switch** in the **mouseDown** event to cause different strings to be displayed based on how many times the mouse has been clicked.

**Figure 20.51**

|  |
| --- |
| // Java2037.java  // This program attempts to display multiple pages of graphics output.  // The program does not compile because there is no access to the graphics object g.  import java.awt.\*;  public class Java2037 extends java.applet.Applet  {  int numClicks;    public void init() { numClicks = 0; }    public void paint(Graphics g)  {  }    public boolean mouseDown(Event e, int x, int y)  {  numClicks++;  switch (numClicks)  {  case 0: g.drawString("TITLE PAGE",200,100); break;  case 1: g.drawString("PAGE 2",200,300); break;  case 2: g.drawString("PAGE 3",200,500); break;  }  repaint();  return true;  }  } |

**Figure 20.51 Continued**

|  |
| --- |
| **Java2037.java Output**  --------------------Configuration: <Default>--------------------  C:\Users\JohnSchram\Documents\)\APCS-TeachingUnits\APCS-21-Advanced Graphics\Programs21\Java2037.java:29: error: cannot find symbol  case 0: g.drawString("TITLE PAGE",200,100); break;  ^  symbol: variable g  location: class Java2037  C:\Users\JohnSchram\Documents\ExpoCS2012\AP-ComputerScience(A)\TeachAPCS(A)\APCS-TeachingUnits\APCS-21-Advanced Graphics\Programs21\Java2037.java:30: error: cannot find symbol  case 1: g.drawString("PAGE 2",200,300); break;  ^  symbol: variable g  location: class Java2037  C:\Users\JohnSchram\Documents\ExpoCS2012\AP-ComputerScience(A)\TeachAPCS(A)\APCS-TeachingUnits\APCS-21-Advanced Graphics\Programs21\Java2037.java:31: error: cannot find symbol  case 2: g.drawString("PAGE 3",200,500); break;  ^  symbol: variable g  location: class Java2037  Note: Java2037.java uses or overrides a deprecated API.  Note: Recompile with -Xlint:deprecation for details.  3 errors |

This does not work. The reason it does not work is that a **mouseDown** event has no access to the **Graphics** object *g*. I have seen many students have this problem, and I have also seen many of them try to fix the problem by adding **Graphics** *g*to the parameter list. This is shown in program **Java2038.java** in figure 20.52.

**Figure 20.52**

|  |
| --- |
| // Java2038.java  // This program attempts to display multiple pages of graphics output.  // While adding the "Graphics g" in the mouseDown parameter list allows  // the program to compile, it also makes it no longer the mouseDown EVENT.  // Now you have a mouseDown METHOD that has no way of being called.  import java.awt.\*;  public class Java2038 extends java.applet.Applet  {  int numClicks;  public void init() { numClicks = 0; }  public void paint(Graphics g)  {  }    public boolean mouseDown(Event e, int x, int y, Graphics g)  {  numClicks++;  switch (numClicks)  {  case 0: g.drawString("TITLE PAGE",200,100); break;  case 1: g.drawString("PAGE 2",200,300); break;  case 2: g.drawString("PAGE 3",200,500); break;  }  repaint();  return true;  }  } |

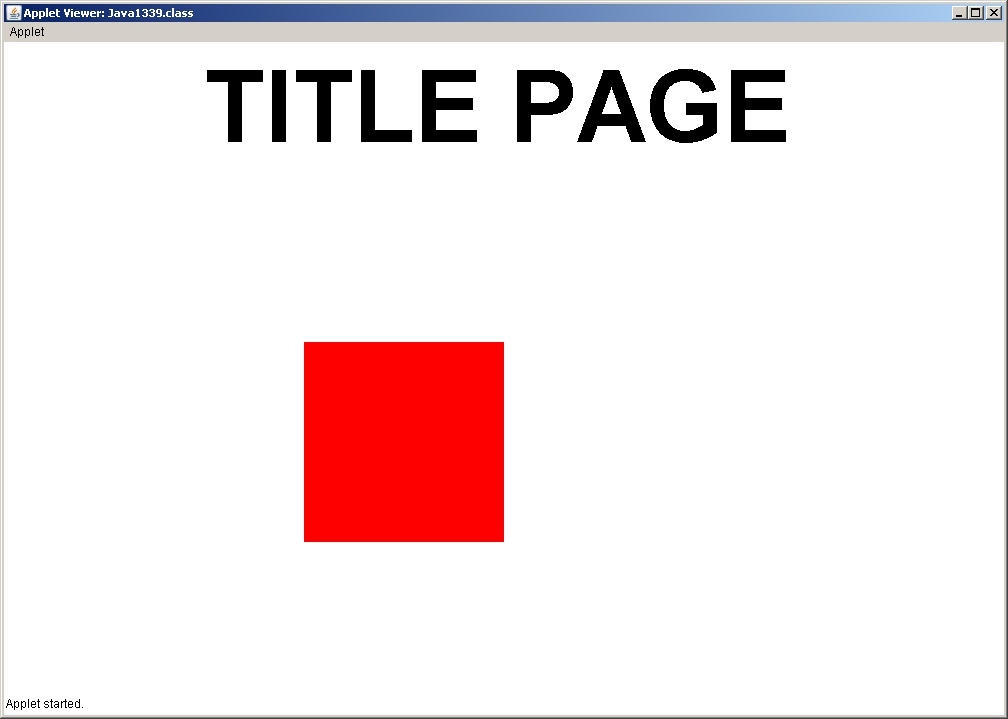
This does not work either. While the program does compile and execute, it has no output. The problem is that adding the extra parameter to the **mouseDown** event made it no longer an *event*. It is now a *method*. To be more specific, it is a method that has no way of being called.

You will be shown one more program example. Program **Java2039.java** in figure 20.53 demonstrates the proper way to do multi-page output. The three pages being displayed are not very advanced at all. They simply display some squares in different colors. What is displayed on each page is not really important. You can put any graphics commands in the different methods being called from the **paint** method.

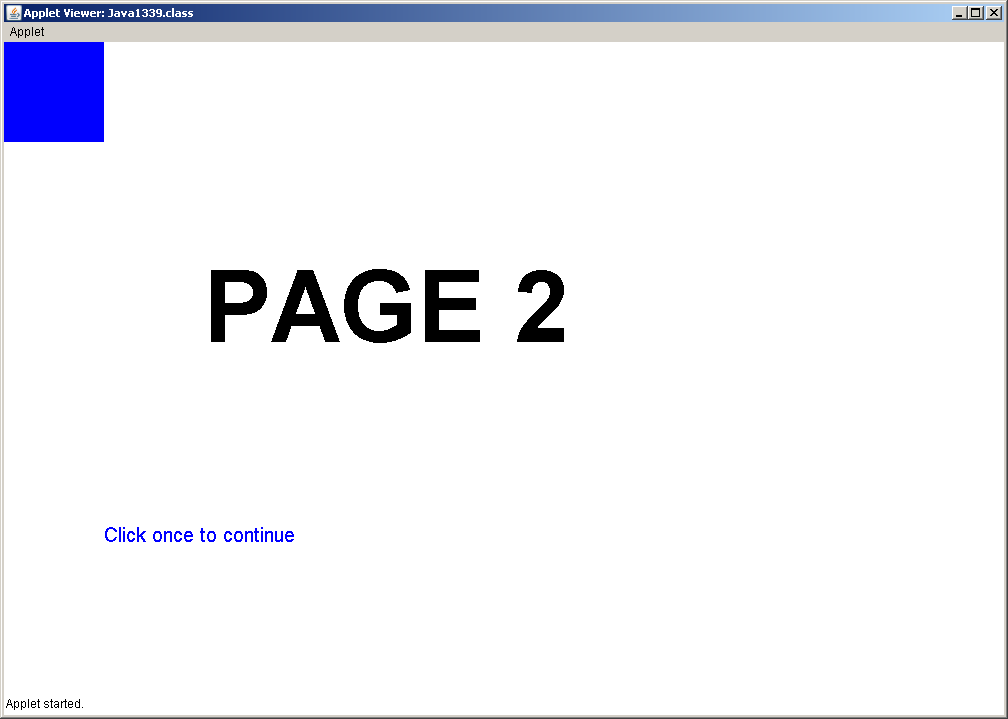
**Figure 20.53**

|  |
| --- |
| // Java2039.java  // This program shows the correct way to display multiple pages of graphics output.  // Remember... graphics commands CANNOT be executed by EVENTS.  // They can only be executed by the paint method, or by methods called from the paint method.  // Events are used to manipulate the values of variables used by the paint method.  import java.awt.\*;  public class Java2039 extends java.applet.Applet  {    int numClicks;    public void init()  {  numClicks = 0;  }    public void paint(Graphics g)  {  switch (numClicks)  {  case 0: page1(g); break;  case 1: page2(g); break;  case 2: page3(g); break;  }  }    public boolean mouseDown(Event e, int x, int y)  {  numClicks++;  repaint();  return true;  }    public void page1(Graphics g)  {  g.setFont(new Font("Arial",Font.BOLD,100));  g.drawString("TITLE PAGE",200,100);  g.setColor(Color.red);  g.fillRect(300,300,200,200);  g.setFont(new Font("Times Roman",Font.PLAIN,20));  g.drawString("Click once to continue",100,500);  }    public void page2(Graphics g)  {  g.setFont(new Font("Arial",Font.BOLD,100));  g.drawString("PAGE 2",200,300);  g.setColor(Color.blue);  g.fillRect(0,0,100,100);  g.setFont(new Font("Times Roman",Font.PLAIN,20));  g.drawString("Click once to continue",100,500);  }    public void page3(Graphics g)  {  g.setColor(Color.green);  g.fillRect(100,100,300,300);  g.setFont(new Font("Arial",Font.BOLD,100));  g.drawString("PAGE 3",200,500);  g.setFont(new Font("Times Roman",Font.PLAIN,20));  g.drawString("Click to exit",400,600);  }  } |

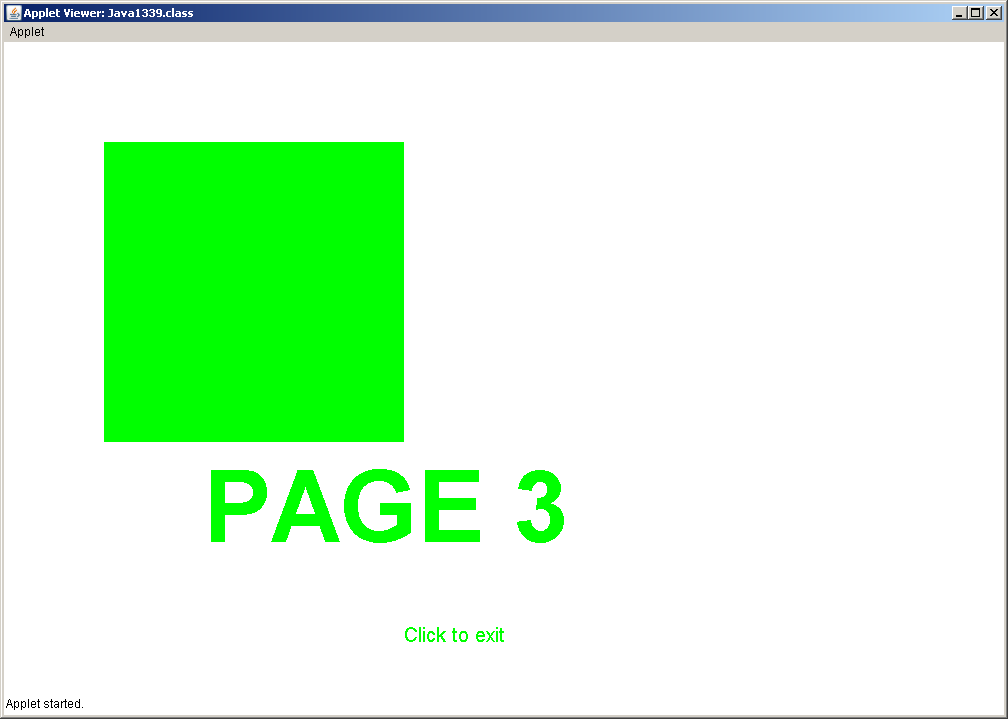
**Figure 20.53 Continued - Page 1**

****

**Figure 20.53 Continued - Page 2**

****

**Figure 20.53 Continued - Page 3**

****

Program **Java2039.java** also shows the proper relationship between the **paint** method and any *events* you might have in your program. **Graphics** methods cannot be called from *events*. They can be called from the **paint** method. They can also be called from methods which are called from the **paint** method. The purpose of the *events* is to alter the value of some variable. Based on the value of this variable, the **paint** method can execute different commands or call different methods.

|  |
| --- |
| **The Relationship Between Events and the paint Method** |
| **Graphics** methods cannot be called from *events*. They can only be called from the **paint** method or from methods which are called from the **paint** method. The purpose of the *events* is to alter the value of some variable. Based on the value of this variable, the **paint** method can execute different commands or call different methods. |

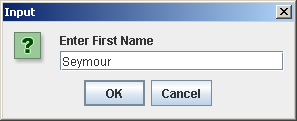
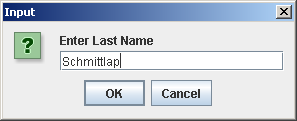
**20.12 Program Input/Output with GUI Windows**

The last section in this chapter does not fit the chapter title. The graphics here are not nearly as advanced as some of what has been shown in this chapter. However, you have written programs that perform input and output on a black and white text screen. This ability is also useful in a graphics program, but neither **System.out.println**,nor **input.nextLine** will work on a graphics screen.

GUI input and output is made possible by the **JOptionPane** class. It has a method for input called **showInputDialog** and another method for output called **showMessageDialog**. Both of the GUI methods are shown by **Java2040.java**, in figure 20.54. The program output shows three boxes that appear in sequence.

**Figure 20.54**

|  |
| --- |
| // Java2040.java  // This program introduces GUI program input/output using Swing dialog boxes.  import javax.swing.JOptionPane;  public class Java2040  {  public static void main (String args[])  {  String firstName = JOptionPane.showInputDialog("Enter First Name");  String lastName = JOptionPane.showInputDialog("Enter Last Name");  String fullName = firstName + " " + lastName;    JOptionPane.showMessageDialog(null,"Your name is " + fullName);    System.exit(0);  }  } |

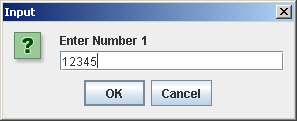
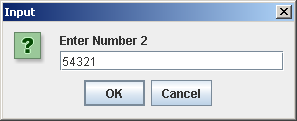
** **

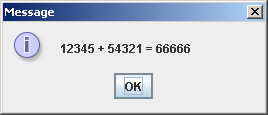
****

The last program demonstrated GUI input entering a first name string of characters and a last name string of characters. You learned that with the text input style it was possible to enter string, integer or double data types. This same process is not available with GUI input. You will need to use **Integer.parseInt** or **Double.parseDouble** to convert the entered **String** value into either an integer or real number respectively. Program **Java2041.java**, in figure 20.55, demonstrates entering 2 integers.

**Figure 20.55**

|  |
| --- |
| // Java2041.java  // This program shows that numerical program input with dialog boxes requires  // a conversion process in the same manner as text window input.  import javax.swing.JOptionPane;  public class Java2041  {  public static void main (String args[])  {  String strNbr1 = JOptionPane.showInputDialog("Enter Number 1");  String strNbr2 = JOptionPane.showInputDialog("Enter Number 2");  int intNbr1 = Integer.parseInt(strNbr1);  int intNbr2 = Integer.parseInt(strNbr2);  int sum = intNbr1 + intNbr2;  JOptionPane.showMessageDialog(null,intNbr1 + " + " + intNbr2 + " = " + sum);    System.exit(0);  }  } |

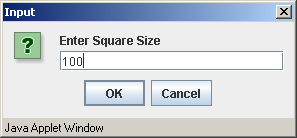
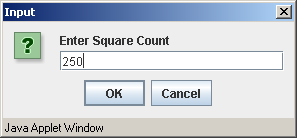
** **

****

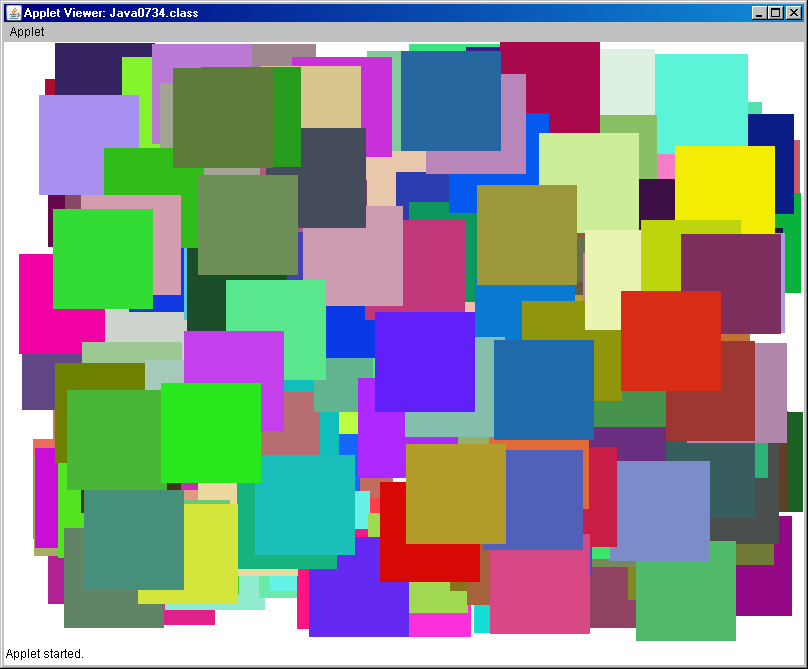
We will conclude with Program **Java2042.java**, in figure 20.56. This program uses GUI input in an applet. It is very important to realize that the calls to **showInputDialog** need to be in the **init** method, so they are only called once at the beginning of the program. If you were to call **showInputDialog** from inside the **paint** method, it would cause the input windows to show up repeatedly.

**Figure 20.56**

|  |
| --- |
| // Java2042.java  // This program demonstrates how to use the <init> method for GUI input  // followed by the <paint> method for graphical output.  import javax.swing.\*;  import java.awt.\*;  import java.util.\*;  public class Java2042 extends JApplet  {  int size; // size of squares to be displayed  int count; // the quantity of squares to be displayed    public void init()  {  String str1 = JOptionPane.showInputDialog("Enter Square Size");  String str2 = JOptionPane.showInputDialog("Enter Square Count");  size = Integer.parseInt(str1);  count = Integer.parseInt(str2);  }    public void paint(Graphics g)  {  Random rnd = new Random(12345);  for (int k = 1; k <= count; k++)  {  int x = rnd.nextInt(800-size);  int y = rnd.nextInt(600-size);  int red = rnd.nextInt(256);  int green = rnd.nextInt(256);  int blue = rnd.nextInt(256);  g.setColor(new Color(red,green,blue));  g.fillRect(x,y,size,size);  }  }  } |

** **

**Figure 20.56 Continued**

****

**20.13 Summary**

This chapter introduced a variety of advanced **awt** graphics features. The use of any class and method in the **awt** (abstract windowing toolkit)package starts with the use of the **import java.awt.\*;** statement.In review it is important to realize which of the graphics identifiers represent classes and which represent methods. We have used the **Graphics** class to draw lines, rectangles, ovals and strings. The **Font** class manipulates the size and style of the fonts used with text graphics. The **Color** class contains constants and methods to control the color display. A special **Polygon** class allows the display of any type of polygon shape.

Using the methods of the **Graphics**, **Color**, **Font,** **Polygon** and the **JOptionPane** classes is not complicated. The biggest concern is to know the parameter requirements and to make sure that parameter values are presented in a correct sequence. All the **Graphics** methods previously introduced and any new **Graphics** methods introduced in this chapter will be summarized in their own tables.

|  |
| --- |
| **drawLine Method Class: Graphics** |
| drawLine(X1,Y1,X2,Y2)  Draws a line from coordinate (X1,Y1) to coordinate (X2,Y2) |

|  |
| --- |
| **drawRect and fillRect Methods Class: Graphics** |
| drawRect(X,Y,Width,Height)  Draws a rectangle with top-left corner at coordinate (X,Y) using Width and Height dimensions.    fillRect uses identical parameters, but fills in the rectangle. |

|  |
| --- |
| **drawRoundRect and fillRoundRect Methods** |
| drawRoundRect(X,Y,Width,Height,CornerW,CornerH)  Draws a rectangle with top-left corner at coordinate (X,Y) using Width and Height dimensions. CornerW and CornerH are the Width and Height values of the rectangle that dimensions the size of the rounded corner.  FillRoundRect uses the same parameters, but fills the rounded rectangle completely. |

|  |
| --- |
| **drawOval and fillOval Methods Class: Graphics** |
| drawOval(X,Y,Width,Height)  Draws an oval that is circumscribed by the rectangle with  top-left corner at coordinate (X,Y) using Width and Height dimensions.    fillOval uses identical parameters, but fills in the oval. |

|  |
| --- |
| **drawArc and fillArc Methods Class: Graphics** |
| Draws an arc that is circumscribed by the rectangle with top-left corner at coordinate (X,Y) using Width and Height dimensions. Start indicates the degree location of the beginning of the arc and Degrees indicates the degrees traveled by the arc. **0** **degrees** is at the **3:00 o’clock** position and increases counter clockwise to **360 degrees**. |

|  |
| --- |
| **drawString Method Class: Graphics** |
| drawString(S,X,Y)  Draws a string S starting at the coordinate (X,Y). |

|  |
| --- |
| **setFont Method Class: Graphics** |
| setFont(Style,Appearance,Size)  Sets the font type for the next text graphics display. **Style** can be specified as **Courier**, **TimesRoman** or **Arial**. An unknown style will default to **Arial**. Appearance can be **PLAIN**, **BOLD** or **ITALIC**, and **Size** is the font size in points. |

|  |
| --- |
| **setColor Method Class: Graphics** |
| **Color.constant Class: Color** |
| setColor(Color.constant)  Sets the graphics display color of the following graphics output to the specified constant of the **Color** class. Color constants are combinations of Red, Green and Blue (RGB) values. The following constants are defined for the Color class:  red 255, 0, 0  green 0, 255, 0  blue 0, 0, 255  orange 255, 200, 0  cyan 0, 255, 255  magenta 255, 0, 255  yellow 255, 255, 0  gray 128, 128, 128  lightgray 192, 192, 192  darkgray 64, 64, 64  pink 255, 175, 175  black 0, 0, 0  white 255, 255, 255 |
| setColor(ColorObject);  The setColor method can also set the color of some specified color by instantiating a new color object with some RGB values.  setColor(new Color(100,0,0)); |

|  |
| --- |
| **addPoint Method Class: Polygon** |
| **drawPolygon & fill Polygon Methods Class: Graphics** |
| PolyObject.addPoint(X,Y);  Adds a coordinate to PolyObject.  drawPolygon(PolyObject);  Draws a polygon by connecting the coordinate stored in the Polygon object passed as parameter (*PolyObject*). |

The **Graphics** methods provided by Java are not sufficient for every display situation. This chapter introduced some mathematical concepts that are used with graphics. Sophisticated graphics, and especially animation, requires knowledge of very complex mathematics. Drawing still objects is already difficult, but rotating objects and moving objects involves use of massive mathematical libraries. The field of graphical animation is exciting, rewarding and it is a high-paying field. You also need to realize that it requires advanced computer science knowledge build on a solid base of sophisticated mathematical concepts.

One example of using mathematics was introduced to draw regular polygons. With the aid of the **sin** and **cos** methods of the **Math** class you saw how it is possible to draw a circle. The circle is drawn with hundreds of points. It is possible to compute a small number of points on this circle with regular intervals. Connecting these points with lines will draw a regular polygon.

You also need to be aware of the difference between mathematical reality, as you learned on a Cartesian system, and the display reality of GUI coordinates. You need to translate the mathematical coordinates according to the screen resolution that you are using. Keep in mind that mathematical Y-values decrease downward.

Graphics displays become far more interesting with the addition of mouse routines that allow interaction with a graphics program. Many mouse routines were introduced during this chapter. These routines will be reviewed in the same manner that they were introduced and used.

In this chapter you learned a new type of module, called an *event*. An event appears to have the same format as any other method with a method heading, parameters and a method body. The key difference is that a method is called by using the method identifier and a mouse event is called when some change in the mouse triggers the appropriate event. Event triggers are button clicking, button release, mouse moving and mouse dragging.

There is an **Event** class the trigger to some mouse event creates an object, called **e** of the **Event** class. The **e** object contains several methods that provide useful information about the behavior of the mouse.

The event examples that follow are examples. Keep in mind that the heading of a mouse event is fixed, as is the information that is provided. The flexible part of the event is the set of program statements that will be executed when the specific event is triggered.

|  |
| --- |
| **mouseDown Event** |
| **Event mouseDown is triggered when either one of the two mouse buttons are clicked. Three examples follow that use the mouseDown event.** |
| **In this example each time that either button is clicked, the numClicks variable is incremented and paint is called.**  **public boolean mouseDown(Event e, int x, int y)**  **{**  **numClicks++;**  **repaint();**  **return true;**  **}** |
| **In this example the mouse location, at the moment that either button is clicked, is recorded in the x,y parameters and assigned to variables xCoord and yCoord.**  **public boolean mouseDown(Event e, int x, int y)**  **{**  **xCoord = x;**  **yCoord = y;**  **repaint();**  **return true;**  **}** |
| **In this example the clickCount method of the e object is used to determine if a single-click or double-click was performed.**  **public boolean mouseDown(Event e, int x, int y)**  **{**  **switch (e.clickCount)**  **{**  **case 1:**  **singleClick = true; doubleClick = false;**  **break;**  **case 2:**  **doubleClick = true; singleClick = false;**  **}**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **mouseEnter Event** |
| **Event mouseEnter is triggered when the mouse cursor enters the applet window.**  **public boolean mouseEnter(Event e, int x, int y)**  **{**  **insideApplet = true;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **mouseExit Event** |
| **Event mouseExit is triggered when the mouse cursor exits the applet window.**  **public boolean mouseExit(Event e, int x, int y)**  **{**  **insideApplet = false;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **mouseUp Event** |
| **Event mouseUp is triggered whenever the mouse button is released after being clicked.**  **public boolean mouseUp(Event e, int x, int y)**  **{**  **endX = x;**  **endY = y;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **mouseMove Event** |
| **Event mouseMove is triggered whenever the mouse moves to a new location.**  **public boolean mouseMove(Event e, int x, int y)**  **{**  **xCoord = x;**  **yCoord = y;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **mouseDrag Event** |
| **Event mouseMove is triggered whenever the mouse moves to a new location with a button held down.**  **public boolean mouseDrag(Event e, int x, int y)**  **{**  **endX = x;**  **endY = y;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **Rectangle Class** |
| **A Rectangle class object, combined with the inside method determines if the mouse is located inside an abstract rectangular area. This abstract area should be made visible with the fillRect method.**    **public void init()**  **{**  **red = new Rectangle(50,50,100,100);**  **green = new Rectangle(50,200,100,100);**  **blue = new Rectangle(50,350,100,100);**  **numColor = 0;**  **}**  **public boolean mouseDown(Event e, int x, int y)**  **{**  **if(red.inside(x,y))**  **numColor = 1;**  **else if(green.inside(x,y))**  **numColor = 2;**  **else if(blue.inside(x,y))**  **numColor = 3;**  **else**  **numColor = 4;**  **repaint();**  **return true;**  **}** |

|  |
| --- |
| **Methods getWidth and getHeight** |
| **Method getWidth returns the width of the applet window.**    **Method getHeight returns the height of the applet window.**    **public void init()**  **{**  **appletWidth = getWidth();**  **appletHeight = getHeight();**  **}** |

|  |
| --- |
| **Method currentTimeMillis** |
| **Method currentTimeMillis returns the value, in milli-seconds, that is maintained by the system clock. This value can be used to compute elapsed time accurate to a milli-second.**    **public void delay(int n)**  **{**  **long startDelay = System.currentTimeMillis();**  **long endDelay = 0;**  **while (endDelay - startDelay < n)**  **endDelay = System.currentTimeMillis();**  **}** |

|  |
| --- |
| **Method update** |
| **Method update alters the performance of method paint. With the presence of the update method, a call to method paint will not clear the video display, but only update new information.**    **public void update(Graphics g)**  **{**  **paint(g);**  **}** |

|  |
| --- |
| **Methods createImage, getGraphics and drawImage** |
| **An object of the Image class creates a virtual memory space used for double buffering with the createImage method. Method getGraphics associated a Graphics buffer object with the Image object. Method drawImage places the graphics display, stored in the virtual memory, into the real-display video memory.**    **public void init()**  **{**  **appletWidth = getWidth();**  **appletHeight = getHeight();**  **virtualMem = createImage(appletWidth,appletHeight);**  **gBuffer = virtualMem.getGraphics();**  **}**  **public void drawSnowman(Graphics g, int x, int y)**  **{**  **createBackGround();**  **gBuffer.setColor(Color.WHITE);**  **gBuffer.fillOval(x+20,y,40,40);**  **gBuffer.fillOval(x+10,y+35,60,60);**  **gBuffer.fillOval(x,y+90,80,80);**  **g.drawImage (virtualMem,0,0, this);**  **}** |

|  |
| --- |
| **Animated Gif Warning** |
| Do not insult your teacher's intelligence by trying to pass off an animated gif as an animated graphics project!  Your teacher will be neither impressed, nor amused and this will be demonstrated in your grade. |

|  |
| --- |
| **The Relationship Between Events and the paint Method** |
| **Graphics** methods cannot be called from *events*. They can only be called from the **paint** method or from methods which are called from the **paint** method. The purpose of the *events* is to alter the value of some variable. Based on the value of this variable, the **paint** method can execute different commands or call different methods. |