CHAPTER 19

Java Never Ends

19.1 MULTITHREADING 958

Example: A Nonresponsive GUI 959

Thread.sleep 959

The getGraphics Method 963

Fixing a Nonresponsive Program Using Threads 964

Example: A Multithreaded Program 965

The Class Thread 965

The Runnable Interface ✜ 968

19.2 JAVABEANS 971

The Component Model 971

The JavaBeans Model 972

19.3 JAVA AND DATABASE CONNECTIONS 973

SQL 973

JDBC 975

CHAPTER SUMMARY 975

ANSWERS TO SELF-TEST EXERCISES 976

PROGRAMMING PROJECTS 976

19 Java Never Ends

And thick and fast they came at last,
And more, and more, and more—

Lewis Carroll, *Through the Looking-Glass*

Introduction

Of course there is only a finite amount of Java, but when you consider all the standard libraries and other accompanying software, the amount of power and the amount to learn seem to be endless. In this chapter we give you a brief introduction to three topics to give you a flavor of some of the directions you can take in extending your knowledge of Java. The three topics are multithreading, JavaBeans, and the interaction of Java with database systems.

Prerequisites

You really should cover most of the book before covering this chapter. However, Section 19.1 requires only Chapters 16 and 18 and their prerequisites. Sections 19.2 and 19.3 require only Chapters 1 through 6. The Sections 19.1, 19.2, and 19.3 are independent of each other and may be read in any order.

19.1 Multithreading

"Can you do two things at once?"
"I have trouble doing one thing at once."

Part of a job interview

thread

A **thread** is a separate computation process. In Java, you can have programs with multiple threads. You can think of the threads as computations that execute in parallel. On a computer with enough processors, the threads might indeed execute in parallel. In most normal computing situations, the threads do not really execute in parallel. Instead, the computer switches resources between threads so that each thread in turn does a little bit of computing. To the user this looks like the processes are executing in parallel.

You have already experienced threads. Modern operating systems allow you to run more than one program at the same time. For example, rather than waiting for your virus scanning program to finish its computation, you can go on to, say, read your E-mail while the virus scanning program is still executing. The operating system is using threads to make this happen. There may or may not be some work being done in parallel depending on your computer

and operating system. Most likely the two computation threads are simply sharing computer resources so that they take turns using the computer's resources. When reading your E-mail, you may or may not notice that response is slower because resources are being shared with the virus scanning program. Your E-mail reading program is indeed slowed down, but since humans are so much slower than computers, any apparent slowdown is likely to be unnoticed.

Example

A Nonresponsive GUI

Display 19.1 contains a very simple action GUI. When the `"Start"` button is clicked, the GUI draws circles one after the other until a large portion of the window is filled with circles. There is 1/10 of a second pause between the drawing of each circle. So, you can see the circles appear one after the other. If you're interested in Java programming, this can be pretty exciting for the first few circles, but it quickly becomes boring. You are likely to want to end the program early, but if you click the close-window button, nothing will happen until the program is finished drawing all its little circles. We will use threads to fix this problem, but first let's understand this program, which does not really use threads in any essential way, despite the occurrence of the word `Thread` in the program. We explain this Swing program in the next few subsections.

`Thread.sleep`

In Display 19.1 the following method invocation produces a 1/10 of a second pause after drawing each of the circles:

```
doNothing(PAUSE);
```

which is equivalent to

```
doNothing(100);
```

The method `doNothing` is a private helping method that does nothing except call the method `Thread.sleep` and take care of catching any thrown exception. So, the pause is really created by the method invocation

```
Thread.sleep(100);
```

`Thread.sleep`

This is a static method in the class `Thread` that pauses whatever thread includes the invocation. It pauses for the number of milliseconds (thousandths of a second) given as an argument. So, this pauses the computation of the program in Display 19.1 for 100 milliseconds or 1/10 of a second.

"Wait a minute," you may think, "the program in Display 19.1 was not supposed to use threads in any essential way." That is basically true, but every Java program uses threads in some way. If there is only one stream of computation, as in Display 19.1,

Display 19.1 Nonresponsive GUI *(Part 1 of 3)*

```
1   import javax.swing.JFrame;
2   import javax.swing.JPanel;
3   import javax.swing.JButton;
4   import java.awt.Container;
5   import java.awt.BorderLayout;
6   import java.awt.FlowLayout;
7   import java.awt.Graphics;
8   import java.awt.event.ActionListener;
9   import java.awt.event.ActionEvent;

10  /**
11  Packs a section of the frame window with circles, one at a time.
12  */
13  public class FillDemo extends JFrame implements ActionListener
14  {
15      public static final int WIDTH = 300;
16      public static final int HEIGHT = 200;
17      public static final int FILL_WIDTH = 300;
18      public static final int FILL_HEIGHT = 100;
19      public static final int CIRCLE_SIZE = 10;
20      public static final int PAUSE = 100; //milliseconds

21      private JPanel box;

22      public static void main(String[] args)
23      {
24          FillDemo gui = new FillDemo();
25          gui.setVisible(true);
26      }

27      public FillDemo()
28      {
29          setSize(WIDTH, HEIGHT);
30          setTitle("FillDemo");
31          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

32          Container contentPane = getContentPane();
33          contentPane.setLayout(new BorderLayout());
```

Display 19.1 Nonresponsive GUI *(Part 2 of 3)*

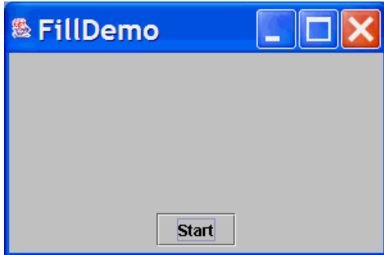
```
34          box = new JPanel();
35          contentPane.add(box, "Center");

36          JPanel buttonPanel = new JPanel();
37          buttonPanel.setLayout(new FlowLayout());
38          JButton startButton = new JButton("Start");
39          startButton.addActionListener(this);
40          buttonPanel.add(startButton);
41          contentPane.add(buttonPanel, "South");
42      }

43      public void actionPerformed(ActionEvent e)
44      {
45          fill();
46      }

47      public void fill()
48      {
49          Graphics g = box.getGraphics();

50          for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
51              for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
52              {
53                  g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
54                  doNothing(PAUSE);
55              }
56      }

57      public void doNothing(int milliseconds)
58      {
59          try
60          {
61              Thread.sleep(milliseconds);
62          }
63          catch(InterruptedException e)
64          {
65              System.out.println("Unexpected interrupt");
66              System.exit(0);
67          }
68      }
69  }
```

Nothing else can happen until `actionPerformed` returns, which does not happen until `fill` returns.

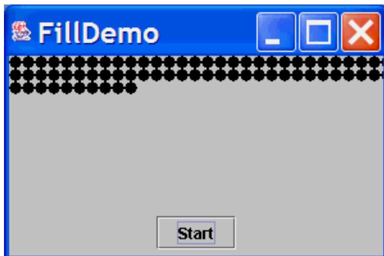
Everything stops for 100 milliseconds (1/10 of a second).

Display 19.1 **Nonresponsive GUI *(Part 3 of 3)***

RESULTING GUI (When started)

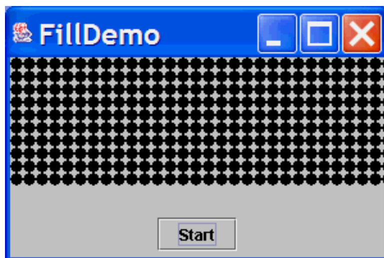


RESULTING GUI (While drawing circles)



If you click the close-window button while the circles are being drawn, the window will not close until all the circles are drawn.

RESULTING GUI (After all circles are drawn)



then that is treated as a single thread by Java. So, threads are always used by Java, but not in an interesting way until more than one thread is used.

You can safely think of the invocation of

```
Thread.sleep(milliseconds);
```

as a pause in the computation that lasts (approximately) the number of milliseconds given as the argument. (If this invocation is in a thread of a multithreaded program, then the pause, like anything else in the thread, applies only to the thread in which it occurs.)

The method `Thread.sleep` can sometimes be handy even if you do not do any multithreaded programming. The class `Thread` is in the package `java.lang` and so requires no import statement.

The method `Thread.sleep` can throw an `InterruptedException`, which is a checked exception—that is, it must be either caught in a `catch` block or declared in a `throws` clause. We do not discuss `InterruptedException` in this book, leaving it for more advanced books on multithreaded programming, but it has to do with one thread interrupting another thread. We will simply note that an `InterruptedException` may be thrown by `Thread.sleep` and so must be accounted for, in our case by a simple `catch` block. The class `InterruptedException` is in the `java.lang` package and so requires no import statement.

Thread.sleep

`Thread.sleep` is a static method in the class `Thread` that pauses the thread that includes the invocation. It pauses for the number of milliseconds (thousandths of a second) given as an argument.

The method `Thread.sleep` may throw an `InterruptedException`, which is a checked exception and so must be either caught in a `catch` block or declared in a `throws` clause.

The classes `Thread` and `InterruptedException` are both in the package `java.lang`, so neither requires any import statement.

Note that `Thread.sleep` can be invoked in an ordinary (single thread) program of the kind we have seen before this chapter. It will insert a pause in the single thread of that program.

SYNTAX:

```
Thread.sleep(Number_Of_Milliseconds);
```

EXAMPLE:

```
try
{
    Thread.sleep(100); //Pause of 1/10 of a second
}
catch(InterruptedException e)
{
    System.out.println("Unexpected interrupt");
}
```

THE getGraphics METHOD

The other new method in Display 19.1 is the `getGraphics` method, which is used in the following line from the method `fill`:

```
Graphics g = box.getGraphics();
```

getGraphics

The `getGraphics` method is almost self-explanatory. As we already noted in Chapter 18, almost every item displayed on the screen (more precisely, every `JComponent`) has an associated `Graphics` object. The method `getGraphics` is an accessor method that returns the associated `Graphics` object (of the calling object for `getGraphics`), in this case, the `Graphics` object associated with the panel `box`. This gives us a `Graphics` object that can draw circles (or anything else) in the panel `box`.

We still need to say a bit more about why the program in Display 19.1 makes you wait before it will respond to the close-window button, but otherwise this concludes our explanation of Display 19.1. The rest of the code consists of standard things we have seen before.

getGraphics

Every `JComponent` has an associated `Graphics` object. The method `getGraphics` is an accessor method that returns the associated `Graphics` object of its calling object.

SYNTAX:

```
Component.getGraphics();
```

EXAMPLE (SEE DISPLAY 19.1 FOR CONTEXT):

```
Graphics g = box.getGraphics();
```

FIXING A NONRESPONSIVE PROGRAM USING THREADS

Now that we have explained the new items in the program in Display 19.1, we are ready to explain why it is nonresponsive and to show you how to use threads to write a responsive version of that program.

Recall that when you run the program in Display 19.1, it draws circles one after the other to fill a portion of the frame. Although there is only a 1/10 of a second pause between drawing each circle, it can still seem like it takes a long time to finish. So, you are likely to want to abort the program and close the window early. But, if you click the close-window button, the window will not close until the GUI is finished drawing all the circles.

Here is why the close-window button is nonresponsive: The method `fill`, which draws the circles, is invoked in the body of the method `actionPerformed`. So, the method `actionPerformed` does not end until after the method `fill` ends. And, until the method `actionPerformed` ends, the GUI cannot go on to do the next thing, which is probably to respond to the close-window button.

Here is how we fixed the problem: We have the method `actionPerformed` create a new (independent) thread to draw the circles. Once `actionPerformed` has created this new thread, the new thread is an independent process that proceeds on its own. The method `actionPerformed` has nothing more to do with this new thread; the work of

`actionPerformed` is ended. So, the main thread (the one with `actionPerformed`) is ready to move on to the next thing, which will probably be to respond promptly to a click of the close-window button. At the same time, the new thread draws the circles. So, the circles are drawn, but at the same time a click of the close-window button will end the program. The program that implements this multithreaded solution is given in the next Programming Example.

Example

A MULTITHREADED PROGRAM

Display 19.2 contains a program that uses a main thread and a second thread to implement the technique discussed in the previous subsection. The general approach was outlined in the previous subsection, but we need to explain the Java code details. We do that in the next few subsections.

THE CLASS Thread

In Java, a thread is an object of the class `Thread`. The normal way to program a thread is to define a class that is a derived class of the class `Thread`. An object of this derived class will be a thread that follows the programming given in the definition of the derived (thread) class.

Thread

Where do you do the programming of a thread? The class `Thread` has a method named `run`. The definition of the method `run` is the code for the thread. When the thread is executed, the method `run` is executed. Of course, the method defined in the class `Thread` and inherited by any derived class of `Thread` does not do what you want your thread to do. So, when you define a derived class of `Thread`, you override the definition of the method `run` to do what you want the thread to do.

run()

In Display 19.2 the inner class `Packer` is a derived class of the class `Thread`. The method `run` for the class `Packer` is defined to be exactly the same as the method `fill` in our previous, unresponsive GUI (Display 19.1). So, an object of the class `Packer` is a thread that will do what `fill` does, namely draw the circles to fill up a portion of the window.

The method `actionPerformed` in Display 19.2 differs from the method `actionPerformed` in our older, nonresponsive program (Display 19.1) in that the invocation of the method `fill` is replaced with the following:

start()

```
Packer packerThread = new Packer();
packerThread.start();
```

This creates a new, independent thread named `packerThread` and starts it processing. Whatever `packerThread` does, it does as an independent thread. The main thread can then allow `actionPerformed` to end and the main thread will be ready to respond to any click of the close-window button.

We only need to explain the method `start` and we will be through with our explanation. The method `start` initiates the computation (process) of the calling thread. It

Display 19.2 Threaded Version of `FillDemo` *(Part 1 of 2)*

The GUI produced is identical to the GUI produced by Display 19.1 except that in this version the close-window button works even while the circles are being drawn, so you can end the GUI early if you get bored.

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JButton;
4  import java.awt.Container;
5  import java.awt.BorderLayout;
6  import java.awt.FlowLayout;
7  import java.awt.Graphics;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;

10 public class ThreadedFillDemo extends JFrame implements ActionListener
11 {
12     public static final int WIDTH = 300;
13     public static final int HEIGHT = 200;
14     public static final int FILL_WIDTH = 300;
15     public static final int FILL_HEIGHT = 100;
16     public static final int CIRCLE_SIZE = 10;
17     public static final int PAUSE = 100; //milliseconds

18     private JPanel box;

19     public static void main(String[] args)
20     {
21         ThreadedFillDemo gui = new ThreadedFillDemo();
22         gui.setVisible(true);
23     }

24     public ThreadedFillDemo()
25     {
26         setSize(WIDTH, HEIGHT);
27         setTitle("Threaded Fill Demo");
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

29         Container contentPane = getContentPane();
30         contentPane.setLayout(new BorderLayout());

31         box = new JPanel();
32         contentPane.add(box, "Center");

33         JPanel buttonPanel = new JPanel();
34         buttonPanel.setLayout(new FlowLayout());
```

Display 19.2 Threaded Version of FillDemo *(Part 2 of 2)*

```
35            JButton startButton = new JButton("Start");
36            startButton.addActionListener(this);
37            buttonPanel.add(startButton);
38            contentPane.add(buttonPanel, "South");
39        }

40        public void actionPerformed(ActionEvent e)
41        {
42                Packer packerThread = new Packer();
43                packerThread.start();
44        }

45        private class Packer extends Thread
46        {
47            public void run()
48            {
49                Graphics g = box.getGraphics();
50                for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
51                  for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
52                  {
53                      g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
54                      doNothing(PAUSE);
55                  }
56            }

57            public void doNothing(int milliseconds)
58            {
59                try
60                {
61                    Thread.sleep(milliseconds);
62                }
63                catch(InterruptedException e)
64                {
65                    System.out.println("Unexpected interrupt");
66                    System.exit(0);
67                }
68            }
69      } //End Packer inner class

70    }
```

You need a thread object, even if there are no instance variables in the class definition of `Packer`.

`start` "starts" the thread and calls `run`.

`run` is inherited from `Thread` but needs to be overridden. This definition of `run` is identical to that of `fill` in Display 19.1.

run()

performs some overhead associated with starting a thread and then it invokes the run method for the thread. As we have already seen, the run method of the class Packer in Display 19.2 draws the circles we want, so the invocation

```
packerThread.start();
```

does draw the circles we want, because it calls run. Note that you do not invoke run directly. Instead, you invoke start, which does some other needed things and invokes run.

This ends our explanation of the multithreaded program in Display 19.2, but there is still one, perhaps puzzling, thing about the class Packer that we should explain. The definition of the class Packer includes no instance variables. So, why do we need to bother with an object of the class Packer? Why not simply make all the methods static and call them with the class name Packer? The answer is that the only way to get a new thread is to create a new Thread object. The things inherited from the class Thread are what the object needs to be a thread. Static methods do not a thread make. In fact, not only will static methods not work, the compiler will not even allow you to define run to be static. This is because run is inherited from Thread as a nonstatic method and that cannot be changed to static when overriding a method definition. The compiler will not let you even try to do this without creating an object of the class Packer.

THE Thread CLASS

A thread is an object of the class Thread. The normal way to program a thread is to define a class that is a derived class of the class Thread. An object of this derived class will be a thread that follows the programming given in the definition of the derived (thread) class's method named run.

Any thread class inherits the method start from the class Thread. An invocation of start by an object of a thread class will start the thread and invoke the method run for that thread.

See Display 19.2 for an example.

THE Runnable INTERFACE ✜

There are times when you would rather not make a thread class a derived class of the class Thread. The alternative to making your class a derived class of the class Thread is to have your class instead implement the Runnable interface. The Runnable interface has only one method heading:

```
public void run()
```

A class that implements the Runnable interface must still be run from an instance of the class Thread. This is usually done by passing the Runnable object as an argument to the thread constructor. The following is an outline of one way to do this:

```
public class ClassToRun extends SomeClass implements Runnable
{
    ....
    public void run()
```

```
    {
        //Fill this just as you would if ClassToRun
         //were derived from Thread.
     }
     ....
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.run();
    }
     ....
}
```

The above method `startThread` is not compulsory, but it is one way to produce a thread that will in turn run the `run` method of an object of the class `ClassToRun`. In Display 19.3 we have rewritten the program in Display 19.2 using the `Runnable` interface. The program behaves exactly the same as the one in Display 19.2.

Self-Test Exercises

1. Since `sleep` is a static method, how can it possibly know what thread it needs to pause?
2. Where was polymorphism used in the program in Display 19.2? (Hint: We are looking for an answer involving the class `Packer`.)

Display 19.3 The Runnable Interface *(Part 1 of 3)*

```
1   import javax.swing.JFrame;
2   import javax.swing.JPanel;
3   import javax.swing.JButton;
4   import java.awt.Container;
5   import java.awt.BorderLayout;
6   import java.awt.FlowLayout;
7   import java.awt.Graphics;
8   import java.awt.event.ActionListener;
9   import java.awt.event.ActionEvent;

10  public class ThreadedFillDemo2 extends JFrame
11                                 implements ActionListener, Runnable
12  {
13      public static final int WIDTH = 300;
14      public static final int HEIGHT = 200;
15      public static final int FILL_WIDTH = 300;
16      public static final int FILL_HEIGHT = 100;
17      public static final int CIRCLE_SIZE = 10;
18      public static final int PAUSE = 100; //milliseconds
```

Display 19.3 The Runnable Interface *(Part 2 of 3)*

```
19      private JPanel box;

20      public static void main(String[] args)
21      {
22          ThreadedFillDemo2 gui = new ThreadedFillDemo2();
23          gui.setVisible(true);
24      }

25      public ThreadedFillDemo2()
26      {
27          setSize(WIDTH, HEIGHT);
28          setTitle("Threaded Fill Demo");
29          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

30          Container contentPane = getContentPane();
31          contentPane.setLayout(new BorderLayout());

32          box = new JPanel();
33          contentPane.add(box, "Center");

34          JPanel buttonPanel = new JPanel();
35          buttonPanel.setLayout(new FlowLayout());

36          JButton startButton = new JButton("Start");
37          startButton.addActionListener(this);
38          buttonPanel.add(startButton);
39          contentPane.add(buttonPanel, "South");
40      }

41      public void actionPerformed(ActionEvent e)
42      {
43          startThread();
44      }

45      public void run()
46      {
47          Graphics g = box.getGraphics();
48          for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
49            for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
50            {
51                g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
52                doNothing(PAUSE);
53            }
54      }
```

Display 19.3 **The Runnable Interface** ***(Part 3 of 3)***

```
55      public void startThread()
56      {
57          Thread theThread = new Thread(this);
58          theThread.start();
59      }
60      public void doNothing(int milliseconds)
61      {
62          try
63          {
64              Thread.sleep(milliseconds);
65          }
66          catch(InterruptedException e)
67          {
68              System.out.println("Unexpected interrupt");
69              System.exit(0);
70          }
71      }
72  }
```

19.2 JavaBeans

Insert tab A into slot B.

Common assembly instruction

JavaBeans refers to a framework that facilitates software building by connecting software components from diverse sources. Some of the components might be standard existing pieces of software. Some might be designed for the particular application. Typically, the various components were designed and coded by different teams. If the components are all designed within the JavaBeans framework, that simplifies the process of integrating the components and means that the components produced can more easily be reused for future software projects. JavaBeans have been widely used. For example, the AWT and Swing packages were built within the JavaBeans framework.

JavaBeans

THE COMPONENT MODEL

You are most likely to have heard the word "component" when shopping for a home entertainment system. The individual pieces, such as a receiver/amplifier, DVD player, speakers, and so forth, are called "components." You connect the components to produce a working system, but you do not connect them in just any way. You must connect them following the interface rules for each component. The speaker wire must

connect to the correct plug, and there better be a plug for it to connect to. You may think it is obvious that a receiver/amplifier needs to have connections for speakers; it is obvious if the receiver/amplifier design is going to be used to make many identical units for use by many different people. However, if you were only making one receiver/amplifier for one home entertainment system, you might just "open the box" and connect the wire inside. Software systems, unfortunately, are often constructed using the "open the box" approach. The component model says that components should always have well-defined connections for other components, which in our case will be other software components, not speakers, but the idea is the same.

THE JAVABEANS MODEL

A component model specifies how components interact with one another. In the case of JavaBeans, the software components (classes) are required to provide at least the following interface services or abilities:

1. **Rules to Ensure Consistency in Writing Interfaces:**
   For example, the rules say, among other things, that the name of an accessor method must begin with `get` and that the name of a mutator method must start with `set`. This same rule has always been a style rule for us in this book, but it was a "should do." In the JavaBeans framework it becomes a "must do." Of course, there are other rules as well. This is just a sample rule.

event handling

2. **An Event Handling Model:**
   This is essentially the event-handling model we presented for the AWT and Swing. (Remember the AWT and Swing were done within the JavaBeans framework.)

persistence

3. **Persistence:**
   Persistence means that an object has an identity that extends beyond one session. For example, a `JFrame` of the kind we have seen so far may be used, and when you are finished using it, it goes away. The next time you use it, it starts out completely new, born again just as it started before. Persistence means the `JFrame` or other component can retain information about its former use; its state is saved, for example, in a database someplace.

introspection

4. **Introspection:**
   Introspection is an enhancement of simple accessor and mutator methods. It includes facilities to find out what access to a component is available as well as providing access.

5. **Builder Support:**
   These are primarily IDEs (Integrated Development Environments) designed to connect JavaBean components to produce a final application. Some examples are Sun's Bean Builder, Symantec's Visual Cafe, and Borland's JBuilder.

Self-Test Exercises

3. What is meant by *persistence*?

4. What event-handling model is used with JavaBeans?

WHAT IS A JAVABEAN?

A **JavaBean** (often called a *JavaBean component* or simply a *Bean*) is a reusable software component (Java class or classes) that satisfies the requirements of the JavaBeans framework and that can be manipulated in an IDE designed for building applications out of Beans.

WHAT ARE ENTERPRISE JAVABEANS?

The **Enterprise JavaBean** framework extends the JavaBeans framework to more readily accommodate business applications.

19.3 Java and Database Connections

It is a capital mistake to theorize before one has data.

Sir Arthur Conan Doyle (*Sherlock Holmes*), *Scandal in Bohemia*

As an example of how Java has been extended to interact with other software systems, in this section, we will briefly describe how Java interacts with database management systems. This is not a complete enough introduction to allow you to immediately start writing Java code for database manipulations. The intent of this section is to let you know what kinds of things are available to you for database programming in Java.

SQL

SQL

SQL is a standard for database access that has been adopted by virtually all database vendors. The initials SQL stand for Structured Query Language. SQL is pronounced either by saying the letters or by saying the word "sequel." SQL is a language for formulating queries for a relational database. SQL is not part of Java, but Java does have a library (and accompanying implementation software) known as JDBC that allows you to embed SQL commands in your Java code.

SQL works with relational databases. Most commercially available database management systems are relational databases. A relational database can be thought of as a collection of named tables with rows and columns, such as those shown in Display 19.4. In this brief introduction we will not go into the details of the constraints on tables, but to see that there are some constraints, note that in the three tables in Display 19.4, no relationship is repeated. For example, if we had one entry for each book with all the information—title, author, ISBN number[1], and author's URL—then there would be two entries giving Dan Simmons' URL, since he has two books in our database.

Display 19.4 Relational Database Tables

Names

AUTHOR	AUTHOR_ID	URL
Adams, Douglas	1	http:// ...
Simmons, Dan	2	http:// ...
Stephenson, Neal	3	http:// ...

Titles

TITLE	ISBN
Snow Crash	0-553-38095-8
Endymion	0-553-57294-6
The Hitchhiker's Guide to the Galaxy	0-671-46149-4
The Rise of Endymion	0-553-57298-9

Books **Authors**

ISBN	AUTHOR_ID
0-553-38095-8	3
0-553-57294-6	2
0-671-46149-4	1
0-553-57298-9	2

The following is a sample SQL command:

```
SELECT Titles.Title, Titles.ISBN, BooksAuthors.Author_ID
FROM Titles, BooksAuthors
WHERE Titles.ISBN = BooksAuthors.ISBN
```

This will produce the table shown in Display 19.5. That table contains all titles with matching ISBN number and author ID, given in Display 19.5. The `ISBN` number is the bridge that connects the tables `Titles` and `BooksAuthors`.

[1] The ISBN number is a unique identification number assigned to (almost) every book published.

Display 19.5 **Result of SQL Command in Text**

Result

TITLE	ISBN	AUTHOR_ID
Snow Crash	0-553-38095-8	3
Endymion	0-553-57294-6	2
The Hitchhiker's Guide to the Galaxy	0-671-46149-4	1
The Rise of Endymion	0-553-57298-9	2

JDBC

According to most authorities JDBC stands for "Java Database Connectivity." It allows you to insert SQL commands in your Java code. In order to use Java's JDBC you need to have JDBC and a database system compatible with JDBC installed. JDBC is available from Sun. Various Microsoft and Oracle database systems are among the many commercially available database systems that are compatible with JDBC. Typically, you need to download and install a JDBC driver for your database system.

Conceptually JDBC is simple: You establish a connection to a database system (either on your computer or over the Internet) and execute SQL commands, and you do this all within your Java code. The details are not trivial, but hopefully you now know enough about JDBC to know whether you wish to study it further.

Self-Test Exercises

5. Give an SQL command to produce a table of book titles with corresponding author and author ID from the table `Result` in Display 19.5 and one of the tables in Display 19.4. Follow the example of an SQL command given in the text.

Chapter Summary

- A thread is a separate computation process. A Java program can have multiple threads.
- You use the class `Thread` to produce multiple threads.
- The static method `Thread.sleep` inserts a pause into the thread in which it is invoked.
- JavaBeans refers to a framework for producing Java components that are easy to combine with other JavaBean components to produce new applications.
- Using JDBC, you can insert SQL database commands in your Java code.

Answers to Self-Test Exercises

1. The invocation of `Thread.sleep` takes place inside a thread. Every action in Java takes place inside some thread. Any action performed by a method invocation in a specific thread takes place in that thread. Even if it does not know where it is, a method's action takes place where the method invocation is; if you were lost and yelled out, you might know where you are but the yell would still be wherever it is you are.

2. The class `Packer` inherits the method `start` from the base class `Thread` and it is not overridden. The method `start` invokes the method `run`, but when `start` is invoked by an object of the class `Packer`, it is the definition of `run` that is given in the class `Packer` that is used, not the definition of `run` given in the definition of `Thread`. That is exactly what is meant by late binding or polymorphism.

3. *Persistence* means that a component's state can be saved so that the next time it is used it remembers what state it was in.

4. The same one that is used for Swing and AWT.

5.
```
SELECT Result.Title, Names.Author, Result.Author_ID
FROM Result, Names
WHERE Result.Author_ID = Names.Author_ID
```

Programming Projects

1. Modify the GUI in Display 19.2 so that the circles are alternately red, white, and blue and they fill the area from bottom to top instead of top to bottom.

2. Produce a GUI similar to the one in Display 19.2 except that instead of filling an area with circles, it launches a ball (which is just a circle) and the ball bounce around inside a rectangle. You get a moving ball by repeatedly erasing the ball and redrawing it at a location a little further along its path. Look up the method `setXORMode` in the class `Graphics`. It will take care of the erasing.