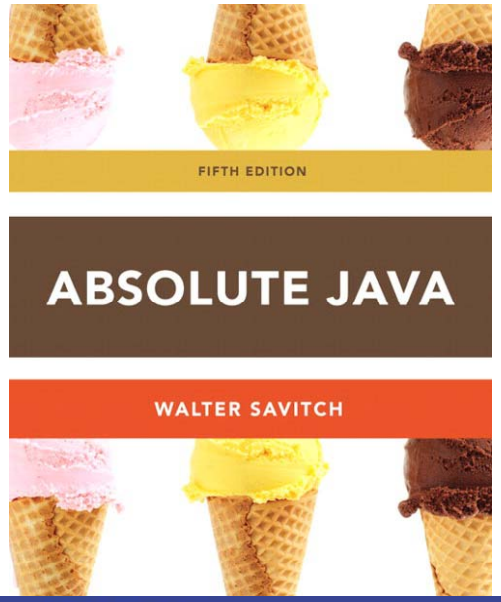


Chapter 19

Java
Never Ends

Slides prepared by Rose Williams,
Binghamton University

Kenrick Mock, *University of Alaska Anchorage*

Multithreading

- In Java, programs can have multiple threads
  - A *thread* is a separate computation process
- Threads are often thought of as computations that run in parallel
  - Although they usually do not really execute in parallel
  - Instead, the computer switches resources between threads so that each one does a little bit of computing in turn
- Modern operating systems allow more than one program to run at the same time
  - An operating system uses threads to do this

`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 given as an argument
  - Note that it may be invoked in an ordinary program to insert a pause in the single thread of that program
- It may throw a checked exception, **`InterruptedException`**, which must be caught or declared
  - Both the **`Thread`** and **`InterruptedException`** classes are in the package **`java.lang`**

The `getGraphics` Method

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

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

A Nonresponsive GUI

- The following program contains a simple GUI that draws circles one after the other when the "Start" button is clicked
  - There is a 1/10 of a second pause between drawing each circle
- If the close-window button is clicked, nothing happens until the program is finished drawing all its circles
- Note the use of the **`Thread.sleep`** (in the method **`doNothing`**) and **`getGraphics`** (in the method **`fill`**) methods

Nonresponsive GUI (Part 1 of 9)

Nonresponsive GUI

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

(continued)

Nonresponsive GUI (Part 2 of 9)

Nonresponsive GUI

```
9  /**
10 Packs a section of the frame window with circles, one at a time.
11 */
12 public class FillDemo extends JFrame implements ActionListener
13 {
14     public static final int WIDTH = 300;
15     public static final int HEIGHT = 200;
16     public static final int FILL_WIDTH = 300;
17     public static final int FILL_HEIGHT = 100;
18     public static final int CIRCLE_SIZE = 10;
19     public static final int PAUSE = 100; //milliseconds


20     private JPanel box;
```

(continued)

Nonresponsive GUI (Part 3 of 9)

Nonresponsive GUI

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


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


31        setLayout(new BorderLayout());
```

(continued)

Nonresponsive GUI (Part 4 of 9)

Nonresponsive GUI

```
32          box = new JPanel();
33          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          add(buttonPanel, "South");
40      }
```

(continued)

Nonresponsive GUI (Part 5 of 9)

Nonresponsive GUI

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

45      public void fill()
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)
```

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

(continued)

Nonresponsive GUI (Part 6 of 9)

Nonresponsive GUI

```
50              {
51                  g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
52                  doNothing(PAUSE);
53              }
54      }

55      public void doNothing(int milliseconds)
56      {
57          try
58          {
59              Thread.sleep(milliseconds);
60          }
61          catch(InterruptedException e)
62          {
63              System.out.println("Unexpected interrupt");
64              System.exit(0);
65          }
66      }
67  }
```

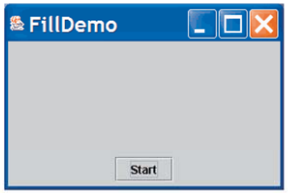
Everything stops for 100 milliseconds (1/10 of a second).

(continued)

Nonresponsive GUI (Part 7 of 9)

Nonresponsive GUI

RESULTING GUI (When started)

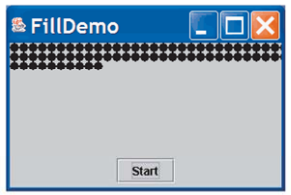


(continued)

Nonresponsive GUI (Part 8 of 9)

Nonresponsive GUI

RESULTING GUI (While drawing circles)



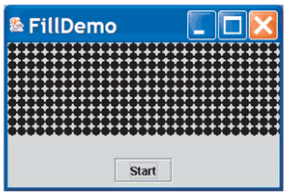
If you click the close-window button while the circles are being drawn, the window will not close until all the circles are drawn.

(continued)

Nonresponsive GUI (Part 9 of 9)

Nonresponsive GUI

RESULTING GUI (After all circles are drawn)



Fixing a Nonresponsive Program Using Threads

- This is why the close-window button does not respond immediately:
  - Because the method `fill` is invoked in the body of the method `actionPerformed`, the method `actionPerformed` does not end until after the method `fill` ends
  - Therefore, the method `actionPerformed` does not end until after the method `fill` ends
  - Until the method `actionPerformed` ends, the GUI cannot respond to anything else

Fixing a Nonresponsive Program Using Threads

- This is how to fix the problem:
  - Have the `actionPerformed` method create a new (independent) thread to draw the circles
  - Once created, the new thread will be an independent process that proceeds on its own
  - Now, the work of the `actionPerformed` method is ended, and the main thread (containing `actionPerformed`) is ready to respond to something else
  - If the close-window button is clicked while the new thread draws the circles, then the program will end

The Class **Thread**

- In Java, a thread is an object of the class **Thread**
- Usually, a derived class of **Thread** is used to program a thread
  - The methods **run** and **start** are inherited from **Thread**
  - The derived class overrides the method **run** to program the thread
  - The method **start** initiates the thread processing and invokes the **run** method

A Multithreaded Program that Fixes a Nonresponsive GUI

- The following program uses a main thread and a second thread to fix the nonresponsive GUI
  - It creates an inner class **Packer** that is a derived class of **Thread**
  - The method **run** is defined in the same way as the previous method **fill**
  - Instead of invoking **fill**, the **actionPerformed** method now creates an instance of **Packer**, a new independent thread named **packerThread**
  - The **packerThread** object then invokes its **start** method
  - The **start** method initiates processing and invokes **run**

Threaded Version of **FillDemo** (Part 1 of 6)

Threaded Version of FillDemo

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

(continued)

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.

Threaded Version of **FillDemo** (Part 2 of 6)

Threaded Version of FillDemo

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

17      private JPanel box;

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

(continued)

Threaded Version of **FillDemo** (Part 3 of 6)

Threaded Version of FillDemo

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

28          setLayout(new BorderLayout());

29          box = new JPanel();
30          add(box, "Center");

31          JPanel buttonPanel = new JPanel();
32          buttonPanel.setLayout(new FlowLayout());
```

(continued)

Threaded Version of **FillDemo** (Part 4 of 6)

Threaded Version of FillDemo

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

38      public void actionPerformed(ActionEvent e)
39      {
40          Packer packerThread = new Packer();
41          packerThread.start();
42      }

43      private class Packer extends Thread
```

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`.

(continued)

Threaded Version of **FillDemo** (Part 5 of 6)

Threaded Version of FillDemo

```
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          }
```

`run` *is inherited from* `Thread` *but needs to be overridden.*

(continued)

Threaded Version of **FillDemo** (Part 6 of 6)

Threaded Version of FillDemo

```
55          public void doNothing(int milliseconds)
56          {
57              try
58              {
59                  Thread.sleep(milliseconds);
60              }
61              catch(InterruptedException e)
62              {
63                  System.out.println("Unexpected interrupt");
64                  System.exit(0);
65              }
66          }
67      } //End Packer inner class

68  }
```

The **Runnable** Interface

- Another way to create a thread is to have a class implement the **Runnable** interface
  - The **Runnable** interface has one method heading:

```
public void run();
```

- A class that implements **Runnable** must still be run from an instance of **Thread**
  - This is usually done by passing the **Runnable** object as an argument to the thread constructor

The **Runnable** Interface: Suggested Implementation Outline

```
public class ClassToRun extends SomeClass implements
   Runnable
{ . . .
  public void run()
  {
    // Fill this as if ClassToRun
    // were derived from Thread
  }
  . . .
  public void startThread()
  {
    Thread theThread = new Thread(this);
    theThread.run();
  }
  . . .
}
```


The Runnable Interface (Part 1 of 5)

The Runnable Interface

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

9  public class ThreadedFillDemo2 extends JFrame
10                              implements ActionListener, Runnable
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
```

(continued)

The Runnable Interface (Part 2 of 5)

The Runnable Interface

```
18     private JPanel box;

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

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

29         setLayout(new BorderLayout());
```

(continued)

The Runnable Interface (Part 3 of 5)

The Runnable Interface

```
30          box = new JPanel();
31          add(box, "Center");

32          JPanel buttonPanel = new JPanel();
33          buttonPanel.setLayout(new FlowLayout());

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

(continued)

The Runnable Interface (Part 4 of 5)

The Runnable Interface

```
39      public void actionPerformed(ActionEvent e)
40      {
41          startThread();
42      }

43      public void run()
44      {
45          Graphics g = box.getGraphics();
46          for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
47            for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
48            {
49                g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
50                doNothing(PAUSE);
51            }
52      }
```

(continued)

The Runnable Interface (Part 5 of 5)

The Runnable Interface

```
53      public void startThread()
54      {
55          Thread theThread = new Thread(this);
56          theThread.start();
57      }

58      public void doNothing(int milliseconds)
59      {
60          try
61          {
62              Thread.sleep(milliseconds);
63          }
64          catch(InterruptedException e)
65          {
66              System.out.println("Unexpected interrupt");
67              System.exit(0);
68          }
69      }
70  }
```


Race Conditions

- When multiple threads change a shared variable it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value.
- This is called a race condition because the final value depends on the sequence in which the threads access the shared value.
- We will use the Counter class to demonstrate a race condition.

Counter Class

Display 19.4 The `Counter` Class

```
1  public class Counter
2  {
3    private int counter;
4    public Counter()
5    {
6          counter = 0;
7    }
8    public int value()
9    {
10         return counter;
11   }
12   public void increment()
13   {
14         int local;
15         local = counter;
16         local++;
17         counter = local;
18   }
19 }
```


Race Condition Example

1. Create a single instance of the Counter class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the Counter class.
3. Each thread runs and invokes the increment() method.
4. Wait for each thread to finish and then output the value of the counter. If there were no race conditions then its value should be 30,000. If there were race conditions then the value will be less than 30,000.

Race Condition Test Class (1 of 3)

Display 19.5 The `RaceConditionTest` Class

```
1  public class RaceConditionTest extends Thread
2  {
3    private Counter countObject;

4    public RaceConditionTest(Counter ctr)
5    {
6        countObject = ctr;
7    }
```

Stores a reference to a single `Counter` object.

Race Condition Test Class (2 of 3)

```
8    public void run()
9    {
10     countObject.increment();
11   }

12   public static void main(String[] args)
13   {
14     int i;
15     Counter masterCounter = new Counter();
16     RaceConditionTest[] threads = new RaceConditionTest[30000];

17     System.out.println("The counter is " + masterCounter.value());
18     for (i = 0; i < threads.length; i++)
19     {
20         threads[i] = new RaceConditionTest(masterCounter);
21         threads[i].start();
22     }
```

Invokes the code in Display 19.4 where the race condition occurs.

The single instance of the `Counter` object.

Array of 30,000 threads.

Give each thread a reference to the single `Counter` object and start each thread.

Race Condition Test Class (3 of 3)

```
23        // Wait for the threads to finish
24        for (i = 0; i < threads.length; i++)
25        {
26             try
27             {
28               threads[i].join();
29             }
30             catch (InterruptedException e)
31             {
32               System.out.println(e.getMessage());
33             }
34        }
35        System.out.println("The counter is " + masterCounter.value());
37     }
38  }
```

Waits for the thread to complete.

Sample Dialogue (output will vary)

```
The counter is 0
The counter is 29998
```


Thread Synchronization

- The solution is to make each thread wait so only one thread can run the code in increment() at a time.
- This section of code is called a **critical region** . Java allows you to add the keyword **synchronized** around a critical region to enforce that only one thread can run this code at a time.

Synchronized

- Two solutions:

```
public synchronized void increment()
{
  int local;
  local = counter;
  local++;
  counter = local;
}
```

```
public void increment()
{
  int local;
  synchronized (this)
  {
   local = counter;
   local++;
   counter = local;
  }
}
```


Networking with Stream Sockets

- Transmission Control Protocol – **TCP**
  - Most common network protocol on the Internet
  - Called a reliable protocol because it guarantees that data sent from the sender is received in the same order it is sent
- **Server**
  - Program waiting to receive input
- **Client**
  - Program that initiates a connection to the server

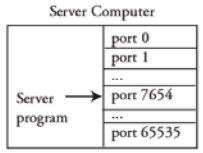
Sockets

- A socket describes one end of the connection between two programs over the network. It consists of:
  - An address that identifies the remote computer, e.g. IP Address
  - A port for the local and remote computer
    - Number between 0 and 65535
    - Identifies the program that should handle data received by the network
    - Only one program may bind to a port
    - Ports 0 to 1024 are reserved for the operating system

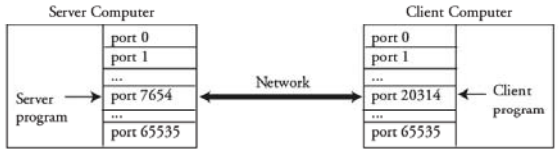
Client/Server Socket Example

Display 19.4 Client/Server Network Communication through Sockets

1. The server listens and waits for a connection on port 7654.



2. The client connects to the server on port 7654. It uses a local port that is assigned automatically, in this case, port 20314.



The server program can now communicate over a socket bound locally to port 7654 and remotely to the client's address at port 20314

The client program can now communicate over a socket bound locally to port 20314 and remotely to the server's address at port 7654

Sockets Programming

- Very similar to File I/O using a `FileOutputStream` but instead we substitute a `DataOutputStream`
- We can use `localhost` as the name of the local machine
- Socket and stream objects throw checked exceptions
  - We must catch them

Date and Time Server (1 of 2)

```
1       import java.util.Date;
2       import java.net.ServerSocket;
3       import java.net.Socket;
4       import java.io.DataOutputStream;
5       import java.io.BufferedReader;
6       import java.io.InputStreamReader;
7       import java.io.IOException;

8       public class DateServer
9       {
10        public static void main(String[] args)
11        {
12              Date now = new Date( );

13              try
14              {
15                System.out.println("Waiting for a connection on port 7654.");
16                ServerSocket serverSock = new ServerSocket(7654);
17                Socket connectionSock = serverSock.accept( );

18                BufferedReader clientInput = new BufferedReader(
19                      new InputStreamReader(connectionSock.getInputStream( )));
20                DataOutputStream clientOutput = new DataOutputStream(
21                      connectionSock.getOutputStream( ));
```

Date and Time Server (2 of 2)

```
22                System.out.println("Connection made, waiting for client " +
23                      "to send their name.");
24                String clientText = clientInput.readLine( );
25                String replyText = "Welcome, " + clientText +
26                      ", Today is " + now.toString( ) + "\n";
27                clientOutput.writeBytes(replyText);
28                System.out.println("Sent: " + replyText);

29                clientOutput.close( );
30                clientInput.close( );
31                connectionSock.close( );
32                serverSock.close( );
33              }
34              catch (IOException e)
35              {
                  System.out.println(e.getMessage( ));
36              }
37        }
38      }
```

SAMPLE DIALOGUE (AFTER CLIENT CONNECTS TO SERVER)
Waiting for a connection on port 7654.
Connection made, waiting for client to send their name.
Sent: Welcome, Dusty Rhodes, Today is Fri Oct 13 03:03:21 AKDT 2006

Date and Time Client (1 of 2)

```
1       import java.net.Socket;
2       import java.io.DataOutputStream;
3       import java.io.BufferedReader;
4       import java.io.InputStreamReader;
5       import java.io.IOException;

6       public class DateClient
7       {
8         public static void main(String[] args)
9         {
10              try
11              {
12                String hostname = "localhost";
13                int port = 7654;

14                System.out.println("Connecting to server on port " + port);
15                Socket connectionSock = new Socket(hostname, port);

16                BufferedReader serverInput = new BufferedReader(
17                      new InputStreamReader(connectionSock.getInputStream( )));
18                DataOutputStream serverOutput = new DataOutputStream(
19                      connectionSock.getOutputStream( ));
```


Date and Time Client (2 of 2)

```
20                System.out.println("Connection made, sending name.");
21                serverOutput.writeBytes("Dusty Rhodes\n");

22                System.out.println("Waiting for reply.");
23                String serverData = serverInput.readLine( );
24                System.out.println("Received: " + serverData);

25                serverOutput.close( );
26                serverInput.close( );
27                connectionSock.close( );
28              }
29              catch (IOException e)
30              {
31                System.out.println(e.getMessage( ));
32              }
33        }
34      }
```

SAMPLE DIALOGUE (AFTER CLIENT CONNECTS TO SERVER)
Connecting to server on port 7654
Connection made, sending name.
Waiting for reply.
Received: Welcome, Dusty Rhodes, Today is Fri Oct 13 03:03:21 AKDT 2006

Sockets and Threading

- The server waits, or blocks, at the `serverSock.accept()` call until a client connects.
- The client and server block at the `readLine()` calls if data is not available.
- This can cause an unresponsive network program and difficult to handle connections from multiple clients on the server end
- The typical solution is to employ threading

Threaded Server

- For the server, the `accept()` call is typically placed in a loop and a new thread created to handle each client connection:

```
while (true)
{
        Socket connectionSock = serverSock.accept( );
        ClientHandler handler = new ClientHandler(connectionSock);
        Thread theThread = new Thread(handler);
        theThread.start( );
}
```


JavaBeans

- *JavaBeans* is a framework that facilitates software building by connecting software components from different sources
  - Some may be standard
  - Others may be designed for a particular application
- Components built using this framework are more easily integrated and reused

The JavaBeans Model

- Software components (i.e., classes) that follow the JavaBeans model are required to provide the following interface services or abilities:
  1. Rules to ensure consistency in writing interfaces:
     - For example, all accessor methods must begin with **`get`**, and all mutator methods must begin with **`set`**
     - This is required, not optional
  2. An event handling model:
     - Essentially, the event-handling model for the AWT and Swing

The JavaBeans Model

3. Persistence:
   - A component (such as a **`JFrame`**) can save its state (e.g., in a database), and therefore retain information about its former use
4. Introspection:
   - An enhancement of simple accessor and mutator methods that includes means to find what access to a component is available, as well as providing access
5. Builder support:
   - Integrated Development Environments (IDEs) designed to connect JavaBean components to produce a final application (e.g., Sun's Bean Builder)

JavaBeans and Enterprise JavaBeans

- A JavaBean (often called a *JavaBean component* or just a *Bean*) is a reusable software component that satisfies the requirements of the JavaBeans framework
  - It can be manipulated in an IDE designed for building applications out of Beans
- The *Enterprise JavaBean* framework extends the JavaBeans framework to more readily accommodate business applications

Java and Database Connections: SQL

- Structured Query Language (SQL) is a language for formulating queries for a relational database
  - SQL is not a part of Java, but Java has a library (JDBC) that allows SQL commands to be embedded in Java code
- SQL works with relational databases
  - Most commercial database management systems are relational databases

Java and Database Connections: SQL

- A relational database can be thought of as a collection of named tables with rows and columns
  - Each table relates together certain information, but the same relationship is not repeated in other tables
  - However, a piece of information in one table may provide a bridge to another

Relational Database Tables (Part 1 of 3)

Relational Database Tables

Names

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

(continued)

Relational Database Tables (Part 2 of 3)

Relational Database Tables

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

(continued)

Relational Database Tables (Part 3 of 3)

Relational Database Tables

BooksAuthors

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

A Sample SQL Command

- The following is a sample SQL command that can be used in conjunction with the tables from the previous slide:

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

- The above command will produce the table shown on the following slide

Result of SQL Command in Text

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

Common SQL Statements (1 of 2)

`CREATE TABLE`	Create a new table named `newtable` with fields `field1`, `field2`, etc. Data types are similar to Java and include: `int`, `bigint`, `float`, `double`, and `var(size)` which is equivalent to a String of maximum length `size`.	`CREATE TABLE newtable (field1 datatype, field2 datatype, ...)`
`INSERT`	Insert a new row into the table `tableName` where `field1` has the value `field1Value`, `field2` has the value `field2Value`, etc. The data types for the values must match those for the corresponding fields when the table was created. String values should be enclosed in single quotes.	`INSERT INTO tableName VALUES (field1Value, field2Value, ...)`

Common SQL Statements (2 of 2)

`UPDATE`	Change the specified fields to the new values for any rows that match the `WHERE` clause. `Op` is a comparison operator such as =, <> (not equal to), <, >, etc.	`UPDATE tableName SET field1 = newValue, field2 = newValue, ... WHERE fieldName Op someValue`
`SELECT`	Retrieve the specified fields for the rows that match the `WHERE` clause. The `*` may be used to retrieve all fields. Omit the `WHERE` clause to retrieve all rows from the table.	`SELECT field1, field2 FROM tableName WHERE fieldname Op someValue`

SQL Examples

- CREATE TABLE names(author varchar(50), author_id int, url varchar(80))
- INSERT INTO names VALUES ('Adams, Douglas', 1, 'http://www.douglasadams.com')
- UPDATE names SET url = 'http://www.douglasadams.com/dna/bio.html' WHERE author_id = 1
- SELECT author, author_id, url FROM names
- SELECT author, author_id, url FROM names WHERE author_id > 1

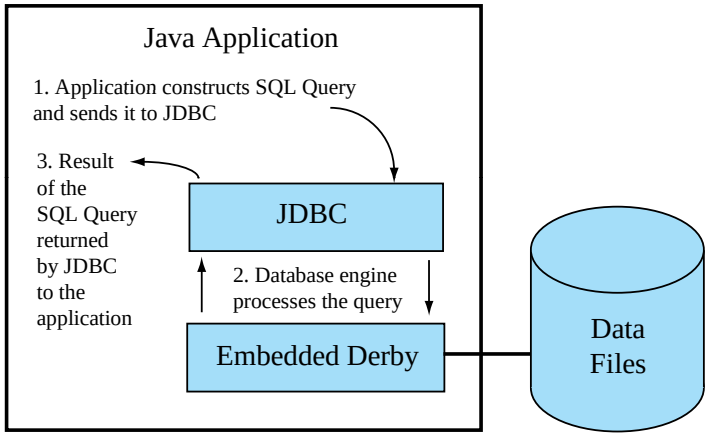
JDBC

- *Java Database Connectivity (JDBC)* allows SQL commands to be inserted into Java code
  - In order to use it, both JDBC and a database system compatible with it must be installed
  - A JDBC driver for the database system may need to be downloaded and installed as well
- Inside the Java code, a connection to a database system is made, and SQL commands are then executed

Java DB

- In the following examples we will use *Java DB*
  - Packaged with version 6 or higher of the Java SDK
  - Based on the open source database known as *Apache Derby*
  - See http://www.oracle.com/technetwork/java/javadb/index.html
  - Installation may require some configuration
  - See instructions that come with Java DB and more detail in the book
- Runs in Network Mode or Embedded Mode
  - We only use embedded mode here

Data Flow of an Embedded Derby Application



Derby Database Connection and Creation

- Steps in accessing a Derby database
  - Load the driver

```
String driver = "org.apache.derby.jdbc.EmbeddedDriver";
Class.forName(driver).newInstance( );
```

  - Connect to the database using a Connection String

```
Connection conn = null;
conn =
DriverManager.getConnection("jdbc:derby:BookDatabase;create=true");
```

  - Issue SQL commands to access or manipulate the database

```
Statement s = conn.createStatement();
s.execute(SQLString);
```

  - Close the connection when done

Derby Database Creation Example (1 of 3)

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  import java.sql.SQLException;
4  import java.sql.Statement;

5  public class CreateDB
6  {
7    private static final String driver = "org.apache.derby.jdbc.EmbeddedDriver";
8    private static final String protocol = "jdbc:derby:";

9    public static void main(String[] args)
10   {
11     try
12     {
13      Class.forName(driver).newInstance();
14      System.out.println("Loaded the embedded driver.");
15     }

16    catch (Exception err)
17    {
18      System.err.println("Unable to load the embedded driver.");
19      err.printStackTrace(System.err);
20      System.exit(0);
21    }
```

Loads embedded Derby driver

Must catch ClassNotFoundException, InstantiationException, IllegalAccessException.

Derby Database Creation Example (2 of 3)

Connection String to create the database. Remove ";create=true" if connecting to an existing database.

```
22  String dbName = "BookDatabase";
23  Connection conn = null;
24  try
25  {
26       System.out.println("Connecting to and creating the database...");
27       conn = DriverManager.getConnection(protocol + dbName + ";create=true");
28       System.out.println("Database created.");

29       Statement s = conn.createStatement();
30       s.execute("CREATE TABLE names" +
31                 "(author varchar(50), author_id int, url varchar(80))");
32       System.out.println("Created 'names' table.");
```

Create a table called "names" with three fields, 50 characters for an author, an integer author ID, and 80 characters for a URL

Derby Database Creation Example (3 of 3)

Insert sample data

```
33       System.out.println("Inserting authors.");
34       s.execute("INSERT INTO names " +
35         "VALUES ('Adams, Douglas', 1, 'http://www.douglasadams.com')");
36       s.execute("INSERT INTO names " +
37         "VALUES ('Simmons, Dan', 2, 'http://www.dansimmons.com')");
38       s.execute("INSERT INTO names " +
39         "VALUES ('Stephenson, Neal', 3, 'http://www.nealstephenson.com')");
40       System.out.println("Authors inserted.");

41       conn.close();
42   }
43   catch (SQLException err)
44   {
45       System.err.println("SQL error.");
46       err.printStackTrace(System.err);
47       System.exit(0);
48   }
49  }
50 }
```

Catch SQL Error Exceptions

SAMPLE DIALOGUE

```
Loaded the embedded driver.
Connecting to and creating the database.
Database created.
Created 'names' table.
Inserting authors.
Authors inserted.
```


Retrieving Data from Derby

- The SELECT statement is used to retrieve data from the database
  - Invoke the `executeQuery( )` method of a `Statement` object.
  - Returns an object of type `ResultSet` that maintains a cursor to each matching row in the database.
    - Can iterate through the set with a loop

Processing a ResultSet

- Initially, the cursor is positioned before the first row.
- The `next( )` method advances the cursor to the next row. If there is no next row, then `false` is returned. Otherwise, `true` is returned.
- Use one of following methods to retrieve data from a specific column in the current row :

```
intVal = resultSet.getInt("name of int field");
lngVal = resultSet.getLong("name of bigint field");
strVal = resultSet.getString("name of varchar field");
dblVal = resultSet.getDouble("name of double field");
fltVal = resultSet.getFloat("name of float field");
```

Reading from a Derby Database

```
// Code to connect to the database

Statement s = conn.createStatement();
ResultSet rs = null;

rs = s.executeQuery("SELECT author, author_id FROM names");

while (rs.next())
{
        int id = rs.getInt("author_id");
        String author = rs.getString("author“);
        System.out.println(id + " " + author);
}

rs.close();

// Above should be in a try/catch block
```

SQL to retrieve the ID and Author for all records

Loop through and print all records that match the query

SAMPLE DIALOGUE
1 Adams, Douglas
2 Simmons, Dan
3 Stephenson, Neal

Update Query

- Use the execute command for UPDATE queries
- Example to change the URL to the contents of the variable newURL for author with ID 1

```
Statement s = conn.createStatement();
s.execute("UPDATE names SET URL = '" + newURL +
            "' WHERE author_id = 1");
```

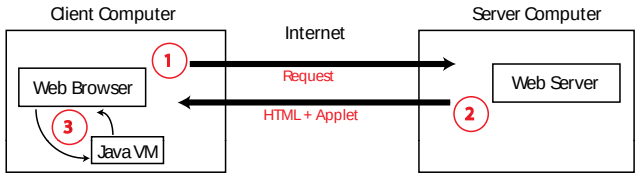

More SQL

- We have just scratched the surface of what is possible to do with SQL, JDBC, Java DB, etc.
- This section covered the basics about how to integrate a database with a Java application
  - Refer to database and more advanced Java textbooks to learn more

Web Programming with Java Server Pages

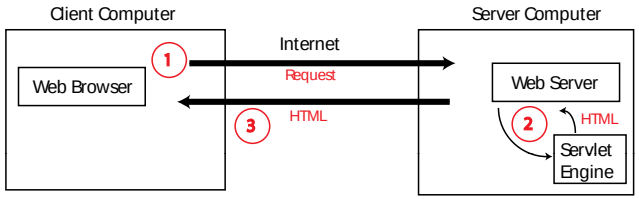
- Many technologies exist that allow programs to run within a web browser when visiting a website
- Applets
  - Run on the client
- Servlets
  - Compiled Java programs on the server
- JSP
  - Dynamically compiles to Servlets and integrated with the server

Running a Java Applet



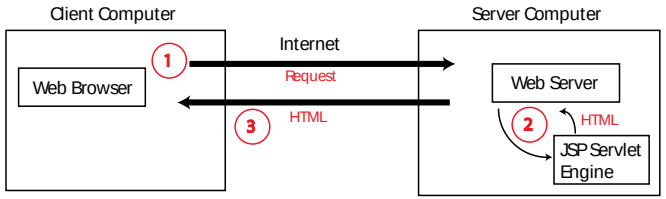
1. The client's web browser sends a request to the server for a web page with a Java Applet.
2. The server sends the HTML for the web page and applet class files to the client.
3. The client runs the applet using the Java Virtual Machine and displays its output in the web browser.

Running a Java Servlet



1. The client's web browser sends a request to the server for a web page that runs a Java servlet.
2. The web server instructs the Servlet engine to execute the requested servlet, which consists of running precompiled Java code. The servlet outputs HTML that is returned to the web server.
3. The web server sends the servlet's HTML to the client's web browser to be displayed.

Running a Java Server Page (JSP) Program



1. The client's web browser sends a request to the server for a web page that contains JSP code.
2. The JSP Servlet engine dynamically compiles the JSP source code into a Java servlet if a current, compiled servlet doesn't exist. The servlet runs and outputs HTML that is returned to the web server.
3. The web server sends the servlet's HTML to the client's web browser to be displayed.

JSP Requirements

- Web server capable of running JSP servlets
- Here we use the Sun GlassFish Enterprise Server, previously known as the Sun Java System Application Server
  - Part of the Java Enterprise Edition SDK
  - See instructions that come with the software for installation
    - Documents go in <glassfish_home>\domains\domain1\docroot
    - Default URL is http://localhost:8080

HTML Forms

- The information you enter into an HTML form is transmitted to the web server using a protocol called the *Common Gateway Interface* (CGI)
- Syntax for HTML Form

```
<FORM ACTION="Path_To_CGI_Program" METHOD="GET or POST">
Form_Elements
</FORM>
```

- ACTION identifies the program to execute
  - In our case, a JSP program
- GET or POST identify how data is transmitted
  - GET sends data as the URL, POST over the socket

Some HTML Form Elements

- Input Textbox

```
<INPUT TYPE="TEXT" NAME="Textbox_Name" VALUE="Default_Text"
 SIZE="Length_In_Characters"
 MAXLENGTH="Maximum_Number_Of_Allowable_Characters">
```

- Submission Button

```
<INPUT TYPE="SUBMIT" NAME="Name" VALUE="Button_Text">
```

- Many others form elements exist
  - E.g. radio buttons, drop down list, etc.

Example HTML Form Document (Display 19.16)

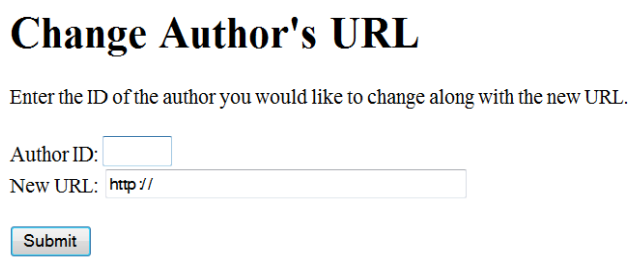
```
<html>
<head>
<title>Change Author's URL</title>
</head>

<body>
<h1>Change Author's URL</h1>
<p>
Enter the ID of the author you would like to change
along with the new URL.
</p>

<form ACTION = "EditURL.jsp" METHOD = POST>
Author ID:
<input TYPE = "TEXT" NAME = "AuthorID"
 VALUE = "" SIZE = "4" MAXLENGTH = "4">
<br />
New URL:
<input TYPE = "TEXT" NAME = "URL"
 VALUE = "http://" SIZE = "40" MAXLENGTH = "200">

<p>
<INPUT TYPE="SUBMIT" VALUE="Submit">
</p>
</form>

</body>
</html>
```

Invokes the JSP program named EditURL.jsp. If this program does not exist you will see an error message upon clicking the Submit button.

Creates a TextBox named AuthorID that is empty, displays 4 characters at once, and accepts at most 4 characters.

Creates a submission button

Browser View of HTML Form Document



JSP Tags - Declarations

- Declarations
  - Use to define variables and methods
  - The variables and methods are accessible from any scriptlets and expressions on the same page
  - Variable declarations are compiled as instance variables for a class that corresponds to the JSP page
  - Syntax:

```
<%!
        Declarations
 %>
```

```
<%!
   private int count = 0;
   private void incrementCount()
   {
         count++;
   }
%>
```

Defines an instance variable named `count` and a method named `incrementCount` that increments the `count` variable

JSP Tags - Expressions

- Expressions
  - Use to access variables defined in declarations
  - Syntax:

```
<%=
        Expression
 %>
```

```
The value of count is <b> <%= count %> </b>
```

Outputs the value of the `count` variable in bold type

JSP Tags - Scriptlet

- Scriptlet
  - Use to embed blocks of Java Code
  - Syntax:

```
<%
        Java Code
 %>
```

  - Use `out.println()` to output to the browser

```
<%
  incrementCount();
  out.println("The counter's value is " + count + "<br />");
%>
```

Invokes the `incrementCount( )` method and then outputs the value in `count`

JSP Example To Display Heading Levels

```
<html>
<title>
Displaying Heading Tags with JSP
</title>

<body>
<%!
   private static final int LASTLEVEL = 6;
%>
<p>
This page uses JSP to display Heading Tags from
Level 1 to Level <%= LASTLEVEL %>
</p>
<%
   int i;
   for (i = 1; i <= LASTLEVEL; i++)
   {
         out.println("<H" + i + ">" +
           "This text is in Heading Level " + i +
           "</H" + i + ">");
   }
%>

</body>
</html>
```

JSP Declaration

JSP Expression that evaluates to 6

JSP Scriptlet

HTML Generated by JSP Example

```
<html>
<title>
Displaying Heading Tags with JSP
</title>
<body>
<p>
This page uses JSP to display Heading Tags from
Level 1 to Level 6
</p>
<H1>This text is in Heading Level 1</H1>
<H2>This text is in Heading Level 2</H2>
<H3>This text is in Heading Level 3</H3>
<H4>This text is in Heading Level 4</H4>
<H5>This text is in Heading Level 5</H5>
<H6>This text is in Heading Level 6</H6>
</body>
</html>
```


Browser View of JSP Page

This page uses JSP to display Heading Tags from Level 1 to Level 6

This text is in Heading Level 1

This text is in Heading Level 2

This text is in Heading Level 3

This text is in Heading Level 4

This text is in Heading Level 5

This text is in Heading Level 6

Reading HTML Form Input

- The **`request.getParameter`** method takes a String parameter as input that identifies the name of an HTML form element and returns the value entered by the user for that element on the form.
  - For example, if there is a textbox named AuthorID then we can retrieve the value entered in that textbox with the scriptlet code:

```
String value = request.getParameter("AuthorID");
```

- If the user leaves the field blank then getParameter returns an empty string.

JSP Program To Echo Input From the HTML Form in Display 19.16

This file should be named "EditURL.JSP" and match the entry in the ACTION tag of the HTML form.

```
<html>
<title>Edit URL: Echo submitted values</title>
<body>
<h2>Edit URL</h2>

<p>
This version of EditURL.jsp simply echoes back to the
user the values that were entered in the textboxes.
</p>

<%
   String url = request.getParameter("URL");
   String stringID = request.getParameter("AuthorID");
   int author_id = Integer.parseInt(stringID);
   out.println("The submitted author ID is: " + author_id);
   out.println("<br/>");
   out.println("The submitted URL is: " + url);
%>
</body>
</html>
```

The `getParameter` method calls return as Strings the values entered by the user in the *URL* and *AuthorID* textboxes from Display 19.16.

Sample Dialogue for EditUrl.JSP

SUBMITTED ON THE WEB BROWSER WHEN VIEWING DISPLAY 19.16

```
Author ID:
2
New URL:
http://www.dansimmons.com/about/bio.htm
```

WEB BROWSER DISPLAY AFTER CLICKING SUBMIT

```
Edit URL
This version of EditURL.jsp simply echoes back to the user
the values that were entered in the textboxes.
The submitted author ID is: 2
The submitted URL is:
http://www.dansimmons.com/about/bio.htm
```


JSP Tags - Directive

- Directives
  - Instruct the compiler how to process a JSP program. Examples include the definition of our own tags, including the source code of other files, and importing packages.
  - Syntax:

```
<%@
        Directives
%>
```

```
<%@
   page import="java.util.*,java.sql.*"
%>
```

Import libraries so we could use SQL code. Multiple packages separated by a comma.

More JSP

- Although we have covered enough JSP to write fairly sophisticated programs, there is much more that we have not covered.
  - For example, beans can be used as a convenient way to encapsulate data submitted from a HTML form.
  - Sessions, tag libraries, security, and numerous other topics are important in the construction of JSP pages.
  - Refer to a textbook dedicated to JSP to learn more.