ADVANCE JAVA -SCOPE INTERNATIONAL

TOPIC OF CONTENTS

1.JDBC

2.Generics

3.Collections Framework

4.MultiThreading

5.Concurrency

6.Fork-join Framework

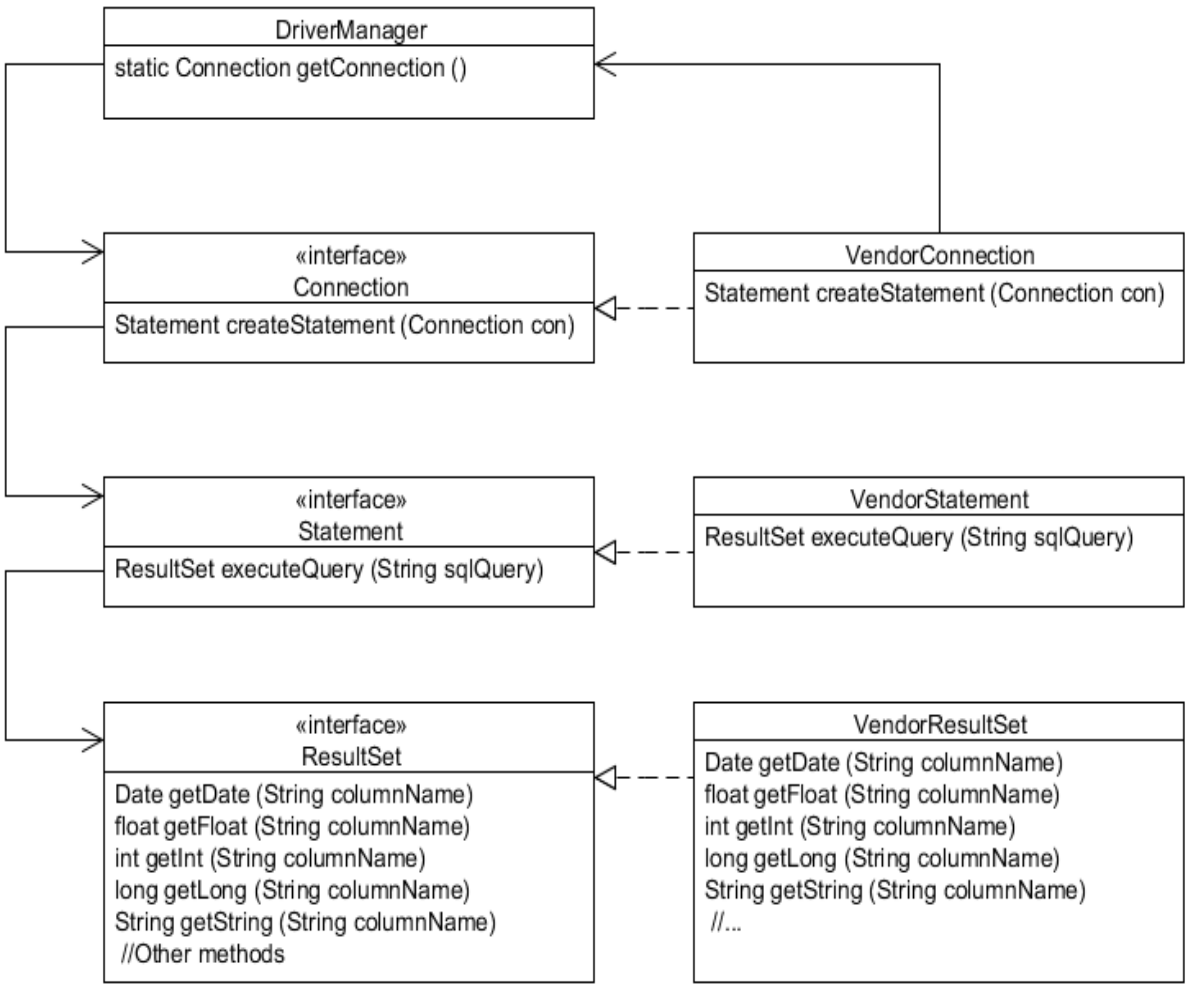
7.Localization

8.networking

Building Database Applications with JDBC

Using the JDBC API

Using a Vendor's Driver Class



The DriverManager class is used to get an instance of a Connection object, using the JDBC driver named in the JDBC URL:

String url = "**jdbc:derby:**//localhost:1527/EmployeeDB";

Connection con = DriverManager.getConnection (url);

The URL syntax for a JDBC driver is:

jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]

Each vendor can implement its own subprotocol.

The URL syntax for an Oracle Thin driver is:

jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE

Example:

jdbc:oracle:thin:@//myhost:1521/orcl

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

java.sql.Connection : A connection that represents the session between your Java application and the database

```
Connection con = DriverManager.getConnection(url, username, password);
```

java.sql.Statement : An object used to execute a static SQL statement and return the result

```
Statement stmt = con.createStatement();
```

java.sql.ResultSet : A object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Using a ResultSet Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Putting It All Together

```
package com.example.text;

import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

public class SimpleJDBCTest {

  public static void main(String[] args) {
    String url = "jdbc:derby://localhost:1527/EmployeeDB";
    String username = "public";
    String password = "tiger";
    String query = "SELECT * FROM Employee";
    try (Connection con =
      DriverManager.getConnection (url, username, password);
      Statement stmt = con.createStatement ();
      ResultSet rs = stmt.executeQuery (query)) {
```

Putting It All Together

```
        while (rs.next()) {
           int empID = rs.getInt("ID");
           String first = rs.getString("FirstName");
           String last = rs.getString("LastName");
           Date birthDate = rs.getDate("BirthDate");
           float salary = rs.getFloat("Salary");
           System.out.println("Employee ID:   " + empID + "\n"
           + "Employee Name: " + first + " " + last + "\n"
           + "Birth Date:    " + birthDate + "\n"
           + "Salary:        " + salary);
        } // end of while
     } catch (SQLException e) {
        System.out.println("SQL Exception: " + e);
     } // end of try-with-resources
  }
}
```

Writing Portable JDBC Code

The JDBC driver provides a programmatic "insulating" layer between your Java application and the database. However, you also need to consider SQL syntax and semantics when writing database applications.

Most databases support a standard set of SQL syntax and semantics described by the American National Standards Institute (ANSI) SQL-92 Entry-level specification.

You can programmatically check for support for this specification from your driver:

```
Connection con = DriverManager.getConnection(url, username, password);
DatabaseMetaData dbm = con.getMetaData();
if (dbm.supportsANSI92EntrySQL()) {
  // Support for Entry-level SQL-92 standard
}
```

The SQLException Class

SQLException can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the SQLException s thrown:

```
catch(SQLException ex) {
   while(ex != null) {
      System.out.println("SQLState:  " + ex.getSQLState());
      System.out.println("Error Code:" + ex.getErrorCode());
      System.out.println("Message:   " + ex.getMessage());
      Throwable t = ex.getCause();
      while(t != null) {
         System.out.println("Cause:" + t);
         t = t.getCause();
      }
```

```
        ex = ex.getNextException();
    }
}
```

Closing JDBC Objects

The try -with-resources Construct

Given the following try -with-resources statement:

```
try (Connection con =
    DriverManager.getConnection(url, username, password);
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery (query)){
```

The compiler checks to see that the object inside the parentheses implements java.lang.AutoCloseable .

This interface includes one method: void close().

The close method is automatically called at the end of the try block in the proper order (last declaration to first).

Multiple closeable resources can be included in the try block, separated by semicolons.

try -with-resources: Bad Practice

It might be tempting to write try -with-resources more compactly:

```
try (ResultSet rs = DriverManager.getConnection(url, username,
password).createStatement().executeQuery(query)) {
```

However, only the close method of ResultSet is called, which is not a good practice.

Always keep in mind which resources you need to close when using try -with- resources.

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the Statement object.

```
Statement stmt = con.createStatement();
```

Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery (query);
```

Note that there are three Statement execute methods:

Method	Returns	Used for
executeQuery(sqlString)	ResultSet	SELECT statement
executeUpdate(sqlString)	int (rows affected)	INSERT, UPDATE, DELETE, or a DDL
execute(sqlString)	boolean (true if there was a ResultSet)	Any SQL command or commands

ResultSetMetaData

There may be a time where you need to dynamically discover the number of columns and their type.

```
int numCols = rs.getMetaData().getColumnCount();
String [] colNames = new String[numCols];
String [] colTypes = new String[numCols];
for (int i= 0; i < numCols; i++) {
   colNames[i] = rs.getMetaData().getColumnName(i+1);
   colTypes[i] = rs.getMetaData().getColumnTypeName(i+1);
}
System.out.println ("Number of columns returned: " + numCols);
System.out.println ("Column names/types returned: ");
for (int i = 0; i < numCols; i++) {
   System.out.println (colNames[i] + " : " + colTypes[i]);
}
```

Getting a Row Count

A common question when executing a query is: "How many rows were returned?"

```
public int rowCount(ResultSet rs) throws SQLException{
   int rowCount = 0;
   int currRow = rs.getRow();
   // Valid ResultSet?
   if (!rs.last()) return -1;
   rowCount = rs.getRow();
   // Return the cursor to the current position
   if (currRow == 0) rs.beforeFirst();
   else rs.absolute(currRow);
  return rowCount;
}
```

To use this technique, the ResultSet must be scrollable.

Controlling ResultSet Fetch Size

By default, the number of rows fetched at one time by a query is determined by the JDBC driver. You may wish to control this behavior for large data sets.

For example, if you wanted to limit the number of rows fetched into cache to 25, you could set the fetch size:

```
rs.setFetchSize(25);
```

Calls to rs.next() return the data in the cache until the 26^{th} row, at which time the driver will fetch another 25 rows.

Using PreparedStatement

PreparedStatement is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.

PreparedStatement is useful when you have a SQL statements that you are going to execute multiple times.

Using CallableStatement

A CallableStatement allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
      = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
       " Employees over the age of " + age);
```

Stored procedures are executed on the database.

What Is a Transaction?

A transaction is a mechanism to handle groups of operations as though they were one.

Either all operations in a transaction occur or none occur at all.

The operations involved in a transaction might rely on one or more databases.

ACID Properties of a Transaction

A transaction is formally defined by the set of properties that is known by the acronym ACID.

Atomicity: A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.

Consistency: A transaction transforms a system from one consistent state to another consistent state.

Isolation: Each transaction occurs independently of other transactions that occur at the same time.

Durability: Completed transactions remain permanent, even during system failure.

Transferring Without Transactions

Successful transfer (A)

Unsuccessful transfer (Accounts are left in an inconsistent state.) (B)

Successful Transfer with Transactions

Changes within a transaction are buffered. (A)

If a transfer is successful, changes are committed (made permanent). (B)

Unsuccessful Transfer with Transactions

Changes within a transaction are buffered. (A)

If a problem occurs, the transaction is rolled back to the previous consistent state. (B)

JDBC Transactions

By default, when a Connection is created, it is in auto-commit mode.

Each individual SQL statement is treated as a transaction and automatically committed after it is executed.

To group two or more statements together, you must disable auto-commit mode.

```
con.setAutoCommit (false);
```

You must explicitly call the commit method to complete the transaction with the database.

```
con.commit();
```

You can also programmatically roll back transactions in the event of a failure.

```
con.rollback();
```

RowSet 1.1: RowSetProvider and RowSetFactory

The JDK 7 API specification introduces the new RowSet 1.1 API. One of the new features of this API is RowSetProvider .

javax.sql.rowset.RowSetProvider is used to create a RowSetFactory object:

```
myRowSetFactory = RowSetProvider.newFactory();
```

The default RowSetFactory implementation is: com.sun.rowset.RowSetFactoryImpl

RowSetFactory is used to create one of the RowSet 1.1 RowSet object types.

Using RowSet 1.1 RowSetFactory

RowSetFactory is used to create instances of RowSet implementations:

Example: Using JdbcRowSet

```
try (JdbcRowSet jdbcRs =
    RowSetProvider.newFactory().createJdbcRowSet()) {
   jdbcRs.setUrl(url);
   jdbcRs.setUsername(username);
   jdbcRs.setPassword(password);
   jdbcRs.setCommand("SELECT * FROM Employee");
   jdbcRs.execute();
   // Now just treat JDBC Row Set like a ResultSet object
   while (jdbcRs.next()) {
```

```
      int empID = jdbcRs.getInt("ID");
      String first = jdbcRs.getString("FirstName");
      String last = jdbcRs.getString("LastName");
      Date birthDate = jdbcRs.getDate("BirthDate");
       float salary = jdbcRs.getFloat("Salary");
   }
  //... other methods
}
```

Quiz

Which Statement method executes a SQL statement and returns the number of rows affected?

stmt.execute(query);

stmt.executeUpdate(query);

stmt.executeQuery(query);

stmt.query(query);

Quiz

When using a Statement to execute a query that returns only one record, it is not necessary to use the ResultSet 's next() method.

True

False

Quiz

The following try -with-resources statement will properly close the JDBC resources:

```
 try (Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery(query)){
    //...
 } catch (SQLException s) {
 }
```

True

False

Quiz

Given:

```
 String[] params = {"Bob", "Smith"};
 String query = "SELECT itemCount FROM Customer " +
            "WHERE lastName='?' AND firstName='?'";
 try (PreparedStatement pStmt = con.prepareStatement(query)) {
    for (int i = 0; i < params.length; i++)
       pStmt.setObject(i, params[i]);
    ResultSet rs = pStmt.executeQuery();
    while (rs.next()) System.out.println (rs.getInt("itemCount"));
 } catch (SQLException e){ }
```

Assuming there is a valid Connection object and the SQL query will produce at least one row, what is the result?

Each itemCount value for customer Bob Smith

Compiler error

A run time error

A SQLException

Generics

Provide flexible type safety to your code

Move many common errors from runtime to compile time

Provide cleaner, easier-to-write code

Reduce the need for casting with collections

Are used heavily in the Java Collections API

Simple Cache Class Without Generics

```
public class CacheString {
   private String message = "";

   public void add(String message){
      this.message = message;
   }

   public String get(){
      return this.message;
   }
}
```

Generic Cache Class

```
public class CacheAny <T>{

   private T t;

   public void add(T t){
      this.t = t;
   }

   public T get(){
      return this.t;
   }
}
```

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
public static void main(String args[]){
  CacheString myMessage = new CacheString(); // Type
  CacheShirt myShirt = new CacheShirt();     // Type

  //Generics
  CacheAny<String> myGenericMessage = new CacheAny<String>();
  CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

  myMessage.add("Save this for me"); // Type
  myGenericMessage.add("Save this for me"); // Generic

}
```

Generics with Type Inference Diamond

Syntax

There is no need to repeat types on the right side of the statement.

Angle brackets indicate that type parameters are mirrored.

Simplifies generic declarations

Saves typing

Quiz

Which of the following is *not* a conventional abbreviation for use with generics?

T: Table

E: Element

K: Key

V: Value

Collections

A collection is a single object designed to manage a group of objects.

Objects in a collection are called *elements*.

Primitives are not allowed in a collection.

Various collection types implement many common data structures:

Stack, queue, dynamic array, hash

The Collections API relies heavily on generics for its implementation.

ADVANCE JAVA -SCOPE INTERNATIONAL

Collection Types

List Interface

List is an interface that defines generic list behavior.

An ordered collection of elements

List behaviors include:

Adding elements at a specific index

Adding elements to the end of the list

Getting an element based on an index

Removing an element based on an index

Overwriting an element based on an index

Getting the size of the list

Use List as a reference type to hide implementation details.

ArrayList Implementation Class

Is a dynamically growable array

The list automatically grows if elements exceed initial size.

Has a numeric index

Elements are accessed by index.

Elements can be inserted based on index.

Elements can be overwritten.

Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);
partList.add(new Integer(1111));
partList.add(new Integer(2222));
partList.add(new Integer(3333));
partList.add(new Integer(4444)); // ArrayList auto grows
System.out.println("First Part: " + partList.get(0)); // First item
partList.add(0, new Integer(5555)); // Insert an item by index
```

ArrayList Without Generics

Generic ArrayList

Generic ArrayList : Iteration and Boxing

The enhanced for loop, or for-each loop, provides cleaner code.

No casting is done because of autoboxing and unboxing.

Autoboxing and Unboxing

Simplifies syntax

Produces cleaner, easier-to-read code

```
public class AutoBox {
  public static void main(String[] args){
     Integer intObject = new Integer(1);
     int intPrimitive = 2;

     Integer tempInteger;
     int tempPrimitive;

     tempInteger = new Integer(intPrimitive);
     tempPrimitive = intObject.intValue();

     tempInteger = intPrimitive; // Auto box
     tempPrimitive = intObject;  // Auto unbox
```

Quiz

Assuming a valid Employee class, and given this fragment:

```
List<Object> staff = new ArrayList<>(3);
staff.add(new Employee(101, "Bob Andrews"));
staff.add(new Employee(102, "Fred Smith"));
staff.add(new Employee(103, "Susan Newman"));
staff.add(3, new Employee(104, "Tim Downs"));
Iterator<Employee> elements = staff.iterator();
while (elements.hasNext())
  System.out.println(elements.next().getName());
```

What change is required to allow this code to compile?

Line 1: The (3) needs to be (4)

Line 8: Need to cast elements.next() to Employee before invoking getName()

Line 1: Object needs to be Employee

Line 5: The 3 needs to be 4

Set Interface

A set is a list that contains only unique elements.

A set has no index.

Duplicate elements are not allowed in a set.

You can iterate through elements to access them.

TreeSet provides sorted implementation.

Set Interface: Example

A set is a collection of unique elements.

```
public class SetExample {
  public static void main(String[] args){
    Set<String> set = new TreeSet<>();

    set.add("one");
    set.add("two");
    set.add("three");
    set.add("three"); // not added, only unique

    for (String item:set){
      System.out.println("Item: " + item);
    }
  }
}
```

Map Interface

A collection that stores multiple key-value pairs

Key: Unique identifier for each element in a collection

Value: A value stored in the element associated with the key

Called “associative arrays” in other languages

Map Types

Map Interface: Example

```
public class MapExample {
  public static void main(String[] args){
    Map <String, String> partList = new TreeMap<>();
    partList.put(“S001", "Blue Polo Shirt");
    partList.put(“S002", "Black Polo Shirt");
    partList.put(“H001", "Duke Hat");

    partList.put(“S002", "Black T-Shirt"); // Overwrite value
    Set<String> keys = partList.keySet();

    System.out.println("=== Part List ===");
    for (String key:keys){
      System.out.println("Part#: " + key + " " +
                  partList.get(key));
    }
  }
}
```

Deque Interface

A collection that can be used as a stack or a queue

Means “double-ended queue” (and is pronounced “deck”)

A queue provides FIFO (first in, first out) operations

add(e) and remove() methods

A stack provides LIFO (last in, first out) operations

push(e) and pop() methods

Stack with Deque: Example

```
public class TestStack {
  public static void main(String[] args){
    Deque<String> stack = new ArrayDeque<>();
    stack.push("one");
    stack.push("two");
    stack.push("three");

    int size = stack.size() - 1;
    while (size >= 0 ) {
      System.out.println(stack.pop());
      size--;
    }
  }
}
```

Ordering Collections

The Comparable and Comparator interfaces are used to sort collections.

Both are implemented using generics.

Using the Comparable interface:

Overrides the compareTo method

Provides only one sort option

Using the Comparator interface:

Is implemented by using the compare method

Enables you to create multiple Comparator classes

Enables you to create and use numerous sorting options

Comparable Interface

Using the Comparable interface:

Overrides the compareTo method

Provides only one sort option

Comparable : Example

```
public class ComparableStudent implements Comparable<ComparableStudent>{
  private String name; private long id = 0; private double gpa = 0.0;

  public ComparableStudent(String name, long id, double gpa){
    // Additional code here
  }
  public String getName(){ return this.name; }
    // Additional code here

  public int compareTo(ComparableStudent s){
    int result = this.name.compareTo(s.getName());
    if (result > 0) { return 1; }
    else if (result < 0){ return -1; }
    else { return 0; }
  }
}
```

Comparable Test: Example

Comparator Interface

Using the Comparator interface:

Is implemented by using the compare method

Enables you to create multiple Comparator classes

Enables you to create and use numerous sorting options

Comparator : Example

```
public class StudentSortName implements Comparator<Student>{
  public int compare(Student s1, Student s2){
    int result = s1.getName().compareTo(s2.getName());
    if (result != 0) { return result; }
    else {
      return 0;  // Or do more comparing
    }
  }
}

public class StudentSortGpa implements Comparator<Student>{
  public int compare(Student s1, Student s2){
    if (s1.getGpa() < s2.getGpa()) { return 1; }
    else if (s1.getGpa() > s2.getGpa()) { return -1; }
    else { return 0; }
```

```
  }
}
```

Comparator Test: Example

```
public class TestComparator {
  public static void main(String[] args){
    List<Student> studentList = new ArrayList<>(3);
    Comparator<Student> sortName = new StudentSortName();
    Comparator<Student> sortGpa = new StudentSortGpa();

    // Initialize list here

    Collections.sort(studentList, sortName);
    for(Student student:studentList){
      System.out.println(student);
    }

    Collections.sort(studentList, sortGpa);
    for(Student student:studentList){
      System.out.println(student);
    }
  }
}
```

Quiz

Which interface would you use to create multiple sort options for a collection?

Comparable

Comparison

Comparator

Comparinator

Threading

Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications. There are two types of tasks that can be scheduled for execution:

Processes: A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.

Thread: A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Why Threading Matters

To execute a program as quickly as possible, you must avoid performance bottlenecks. Some of these bottlenecks are:

Resource Contention: Two or more tasks waiting for exclusive use of a resource

Blocking I/O operations: Doing nothing while waiting for disk or network data transfers

Underutilization of CPUs: A single-threaded application uses only a single CPU

The Thread Class

The Thread class is used to create and start threads. Code to be executed by a thread must be placed in a class, which does either of the following:

Extends the Thread class

Simpler code

Implements the Runnable interface

More flexible

extends is still free.

Extending Thread

Extend java.lang.Thread and override the run method:

```
public class ExampleThread extends Thread {
   @Override
   public void run() {
      for(int i = 0; i < 100; i++) {
         System.out.println("i:" + i);
      }
   }
}
```

Starting a Thread

After creating a new Thread , it must be started by calling the Thread 's start method:

```
public static void main(String[] args) {
   ExampleThread t1 = new ExampleThread();
   t1.start();
}
```

Implementing Runnable

Implement java.lang.Runnable and implement the run method:

```
public class ExampleRunnable implements Runnable {
   @Override
   public void run() {
      for(int i = 0; i < 100; i++) {
         System.out.println("i:" + i);
```

```
    }
  }
}
```

Executing Runnable Instances

After creating a new Runnable , it must be passed to a Thread constructor. The Thread 's start method begins execution:

```
public static void main(String[] args) {
   ExampleRunnable r1 = new ExampleRunnable();
   Thread t1 = new Thread(r1);
   t1.start();
}
```

A Runnable with Shared Data

Static and instance fields are potentially shared by threads.

```
public class ExampleRunnable implements Runnable {
   private int i;

   @Override
   public void run() {
      for(i = 0; i < 100; i++) {
         System.out.println("i:" + i);
      }
   }
}
```

One Runnable: Multiple Threads

An object that is referenced by multiple threads can lead to instance fields being concurrently accessed.

```
public static void main(String[] args) {
   ExampleRunnable r1 = new ExampleRunnable();
   Thread t1 = new Thread(r1);
   t1.start();
   Thread t2 = new Thread(r1);
   t2.start();
}
```

Quiz

Creating a new thread requires the use of:

java.lang.Runnable

java.lang.Thread

java.util.concurrent.Callable

Problems with Shared Data

Shared data must be accessed cautiously. Instance and static fields:

Are created in an area of memory known as heap space

Can potentially be shared by any thread

Might be changed concurrently by multiple threads

There are no compiler or IDE warnings.

"Safely" accessing shared fields is your responsibility.

The preceding slides might produce the following:

i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...

Nonshared Data

Some variable types are never shared. The following types are always thread-safe:

Local variables

Method parameters

Exception handler parameters

Quiz

Variables are thread-safe if they are:

local

static

final

private

Atomic Operations

Atomic operations function as a single operation. A single statement in the Java language is not always atomic.

i++;

Creates a temporary copy of the value in i

Increments the temporary copy

Writes the new value back to i

l = 0xffff_ffff_ffff_ffff;

64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?

Out-of-Order Execution

Operations performed in one thread may not appear to execute in order if you observe the results from another thread.

Code optimization may result in out-of-order operation.

Threads operate on cached copies of shared variables.

To ensure consistent behavior in your threads, you must synchronize their actions.

You need a way to state that an action happens before another.

You need a way to flush changes to shared variables back to main memory.

Quiz

Which of the following cause a thread to synchronize variables?

Reading a volatile field

Calling isAlive() on a thread

Starting a new thread

Completing a synchronized code block

The volatile Keyword

A field may have the volatile modifier applied to it:

```
public volatile int i;
```

Reading or writing a volatile field will cause a thread to synchronize its working memory with main memory.

volatile does not mean atomic.

If i is volatile , i++ is still not a thread- safe operation.

Stopping a Thread

A thread stops by completing its run method.

```
public class ExampleRunnable implements Runnable {
   public volatile boolean timeToQuit = false;

   @Override
   public void run() {
      System.out.println("Thread started");
      while(!timeToQuit) {
         // ...
      }
      System.out.println("Thread finishing");
   }
}
```

Stopping a Thread

```
public static void main(String[] args) {
   ExampleRunnable r1 = new ExampleRunnable();
   Thread t1 = new Thread(r1);
```

```
    t1.start();
    // ...
    r1.timeToQuit = true;
}
```

The synchronized Keyword

The synchronized keyword is used to create thread-safe code blocks. A synchronized code block:

Causes a thread to write all of its changes to main memory when the end of the block is reached

Similar to volatile

Is used to group blocks of code for exclusive execution

Threads block until they can get exclusive access

Solves the atomic problem

synchronized Methods

```
public class ShoppingCart {
    private List<Item> cart = new ArrayList<>();
    public synchronized void addItem(Item item) {
        cart.add(item);
    }
    public synchronized void removeItem(int index) {
        cart.remove(index);
    }
    public synchronized void printCart() {
        Iterator<Item> ii = cart.iterator();
        while(ii.hasNext()) {
            Item i = ii.next();
            System.out.println("Item:" + i.getDescription());
        }
    }
}
```

synchronized Blocks

```
public void printCart() {
    StringBuilder sb = new StringBuilder();
    synchronized (this) {
        Iterator<Item> ii = cart.iterator();
        while (ii.hasNext()) {
            Item i = ii.next();
            sb.append("Item:");
            sb.append(i.getDescription());
            sb.append("\n");
        }
```

```
    }
    System.out.println(sb.toString());
}
```

Object Monitor Locking

Each object in Java is associated with a monitor, which a thread can lock or unlock.

synchronized methods use the monitor for the this object.

static synchronized methods use the classes' monitor.

synchronized blocks must specify which object's monitor to lock or unlock.

```
synchronized ( this ) { }
```

synchronized blocks can be nested.

Detecting Interruption

Interrupting a thread is another possible way to request that a thread stop executing.

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread started");
        while(!Thread.interrupted()) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Interrupting a Thread

Every thread has an interrupt() and isInterrupted() method.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    t1.interrupt();
}
```

Thread.sleep()

A Thread may pause execution for a duration of time.

```
long start = System.currentTimeMillis();
try {
   Thread.sleep(4000);
} catch (InterruptedException ex) {
   // What to do?
}
long time = System.currentTimeMillis() - start;
System.out.println("Slept for " + time + " ms");
```

Quiz

A call to Thread.sleep(4000) will cause the executing thread to always sleep for exactly 4 seconds

True

False

High-Level Threading Alternatives

Traditional Thread related APIs can be difficult to use properly. Alternatives include:

java.util.concurrent.ExecutorService , a higher level mechanism used to execute tasks

It may create and reuse Thread objects for you.

It allows you to submit work and check on the results in the future.

The Fork-Join framework, a specialized work-stealing ExecutorService new in Java 7

java.util.concurrent.ExecutorService

An ExecutorService is used to execute tasks.

It eliminates the need to manually create and manage threads.

Tasks **might** be executed in parallel depending on the ExecutorService implementation.

Tasks can be:

java.lang.Runnable

java.util.concurrent.Callable

Implementing instances can be obtained with Executors .

```
ExecutorService es = Executors.newCachedThreadPool();
```

java.util.concurrent.Callable

The Callable interface:

Defines a task submitted to an ExecutorService

Is similar in nature to Runnable , but can:

Return a result using generics

Throw a checked exception

```
package java.util.concurrent;
public interface Callable<V> {
   V call() throws Exception;
}
```

java.util.concurrent.Future

The Future interface is used to obtain the results from a Callable 's V call() method.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
   V result = future.get();
} catch (ExecutionException|InterruptedException ex) {

}
```

Shutting Down an ExecutorService

Shutting down an ExecutorService is important because its threads are nondaemon threads and will keep your JVM from shutting down.

```
es.shutdown();

try {
   es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
   System.out.println("Stopped waiting early");
}
```

Parallelism

Modern systems contain multiple CPUs. Taking advantage of the processing power in a system requires you to execute tasks in parallel on multiple CPUs.

Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.

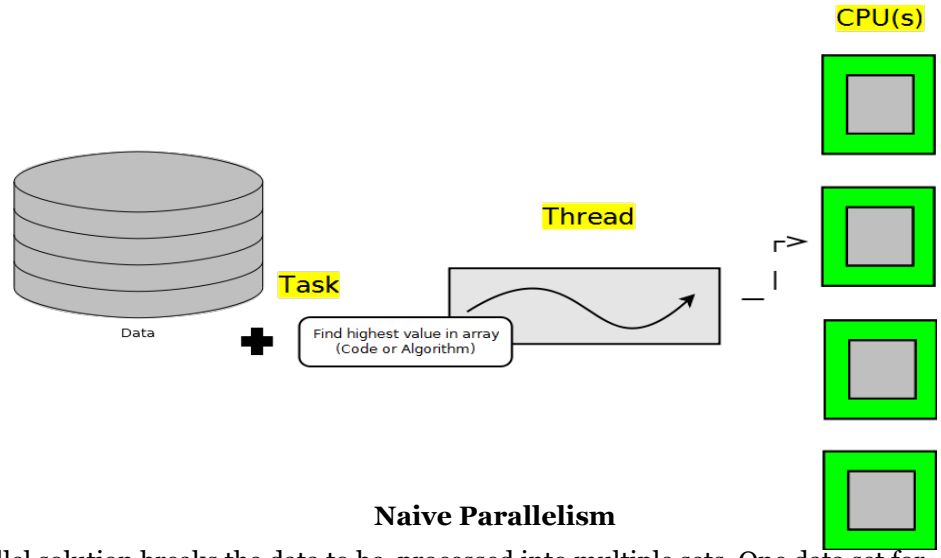
Some problems can be difficult to execute as parallel tasks.

Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.

Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.

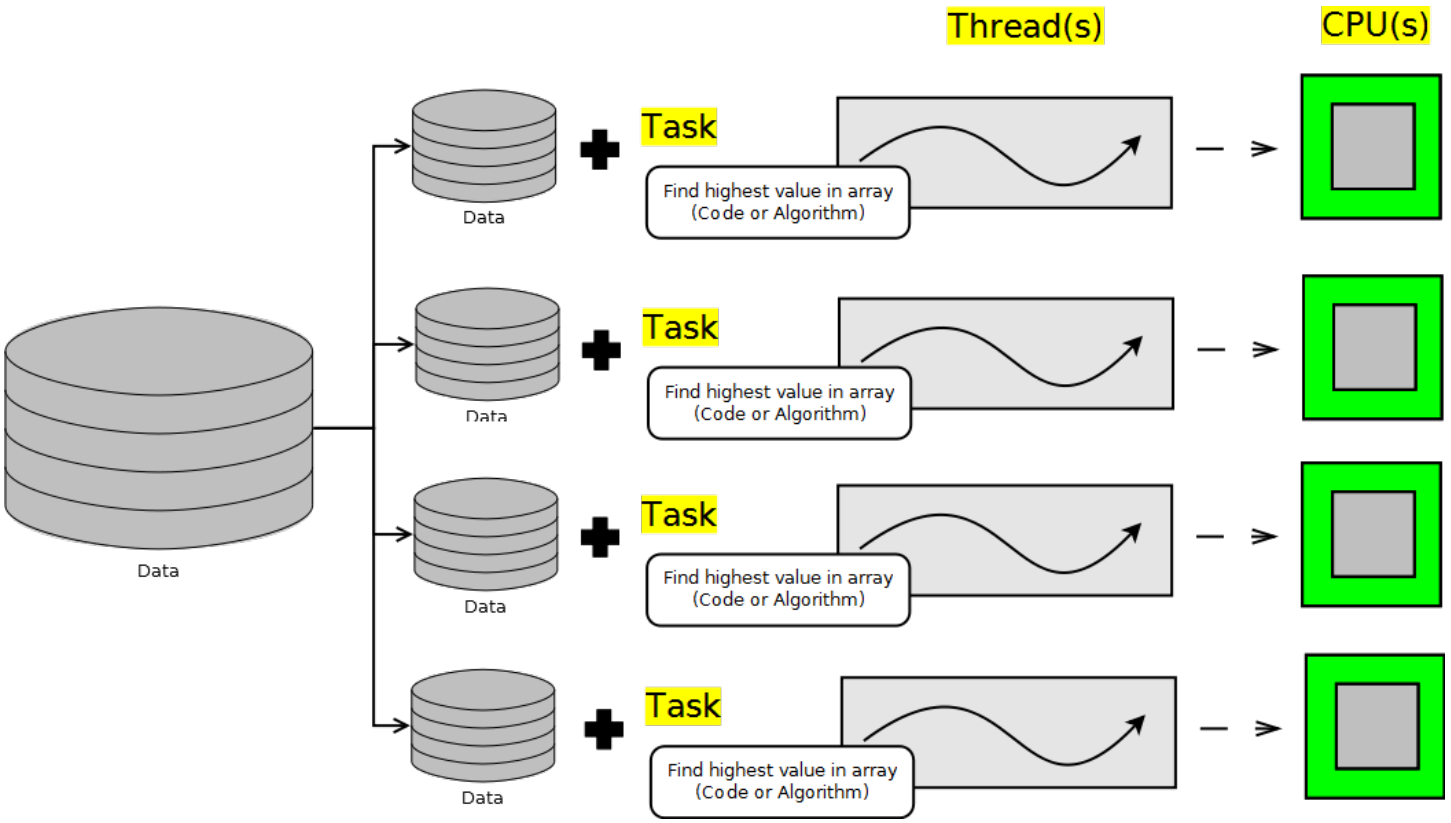
Without Parallelism

Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.



Naive Parallelism

A simple parallel solution breaks the data to be processed into multiple sets. One data set for each CPU and one thread to process each data set.



The Need for the Fork-Join Framework

Splitting datasets into equal sized subsets for each thread to process has a couple of problems. Ideally all CPUs should be fully utilized until the task is finished but:

CPUs may run a different speeds

Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU

Work-Stealing

To keep multiple threads busy:

Divide the data to be processed into a large number of subsets

Assign the data subsets to a thread's processing queue

A Single-Threaded Example

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
   data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
   if (value > max) {
      max = value;
   }
}
System.out.println("Max value found:" + max);
```

java.util.concurrent.ForkJoinTask<V>

A ForkJoinTask object represents a task to be executed.

A task contains the code and data to be processed. Similar to a Runnable or Callable .

A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.

A ForkJoinTask typically creates more ForkJoinTask instances until the data to processed has been subdivided adequately.

Developers typically use the following subclasses:

RecursiveAction : When a task does not need to return a result

RecursiveTask : When a task does need to return a result

RecursiveTask Example

```
public class FindMaxTask extends RecursiveTask<Integer> {
```

```
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end, int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

compute Structure

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

compute Example (Below Threshold)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
```

```
}
```

compute Example (Above Threshold)

```
protected Integer compute() {
   if (end - start < threshold) {
      // find max
   } else {
      int midway = (end - start) / 2 + start;
      FindMaxTask a1 =
   new FindMaxTask(myArray, start, midway, threshold);
      a1.fork();
      FindMaxTask a2 =
   new FindMaxTask(myArray, midway + 1, end, threshold);
      return Math.max(a2.compute(), a1.join());
   }
}
```

ForkJoinPool Example

A ForkJoinPool is used to execute a ForkJoinTask . It creates a thread for each CPU in the system by default.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

Fork-Join Framework Recommendations

Avoid I/O or blocking operations.

Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.

Know your hardware.

A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.

Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.

Know your problem.

Many problems have additional overhead if executed in parallel (parallel sorting, for example).

Quiz

Applying the Fork-Join framework will always result in a performance benefit.

True

False

ADVANCE JAVA -SCOPE INTERNATIONAL

Localization

The decision to create a version of an application for international use often happens at the start of a development project.

Region- and language-aware software

Dates, numbers, and currencies formatted for specific countries

Ability to plug in country-specific data without changing code

A Sample Application

Localize a sample application:

Text-based user interface

Localize menus

Display currency and date localizations

=== Localization App ===
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:

Locale

A Locale specifies a particular language and country:

Language

An alpha-2 or alpha-3 ISO 639 code

“en” for English, “es” for Spanish

Always uses lowercase

Country

Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code

"US" for United States, "ES" for Spain

Always uses uppercase

See The Java Tutorials for details of all standards used

Resource Bundle

The ResourceBundle class isolates locale-specific data:

Returns key/value pairs stored separately

Can be a class or a .properties file

Steps to use:

Create bundle files for each locale.

Call a specific locale from your application.

Resource Bundle File

Properties file contains a set of key/value pairs.

Each key identifies a specific application component.

Special file names use language and country codes.

Default for sample application:

Menu converted into resource bundle

MessageBundle.properties

menu1 = Set to English
menu2 = Set to French
menu3 = Set to Chinese
menu4 = Set to Russian
menu5 = Show the Date
menu6 = Show me the money!
menuq = Enter q to quit

Sample Resource Bundle Files

Samples for French and Chinese

MessagesBundle_fr_FR.properties

menu1 = Régler à l'anglais
menu2 = Régler au français
menu3 = Réglez chinoise
menu4 = Définir pour la Russie
menu5 = Afficher la date
menu6 = Montrez-moi l'argent!
menuq = Saisissez q pour quitter

MessagesBundle_zh_CN.properties

menu1 = □置□英□
menu2 = □置□法□
menu3 = □置□中文
menu4 = □置到俄□斯
menu5 = □示日期
menu6 = □示我的□□
menuq = □入q退出

Quiz

Which bundle file represents a language of Spanish and a country code of US?

MessagesBundle_ES_US.properties

MessagesBundle_es_es.properties

MessagesBundle_es_US.properties

MessagesBundle_ES_us.properties

Initializing the Sample Application

```
PrintWriter pw = new PrintWriter(System.out, true);
    // More init code here

    Locale usLocale = Locale.US;
    Locale frLocale = Locale.FRANCE;
    Locale zhLocale = new Locale("zh", "CN");
    Locale ruLocale = new Locale("ru", "RU");
    Locale currentLocale = Locale.getDefault();

    ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);

      // more init code here

    public static void main(String[] args){
       SampleApp ui = new SampleApp();
       ui.run();
    }
```

Sample Application: Main Loop

The printMenu Method

Instead of text, resource bundle is used.

messages is a resource bundle.

A key is used to retrieve each menu item.

Language is selected based on the Locale setting.

```
    public void printMenu(){
       pw.println("=== Localization App ===");
       pw.println("1. " + messages.getString("menu1"));
       pw.println("2. " + messages.getString("menu2"));
       pw.println("3. " + messages.getString("menu3"));
       pw.println("4. " + messages.getString("menu4"));
       pw.println("5. " + messages.getString("menu5"));
       pw.println("6. " + messages.getString("menu6"));
       pw.println("q. " + messages.getString("menuq"));
       System.out.print(messages.getString("menucommand")+" ");
```

```
}
```

Changing the Locale

To change the Locale :

Set currentLocale to the desired language.

Reload the bundle by using the current locale.

```
public void setFrench(){
   currentLocale = frLocale;
   messages = ResourceBundle.getBundle("MessagesBundle", currentLocale);
}
```

Sample Interface with French

After the French option is selected, the updated user interface looks like the following:

```
=== Localization App ===
1. Régler à l'anglais
2. Régler au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```

Format Date and Currency

Numbers can be localized and displayed in their local format.

Special format classes include:

DateFormat

NumberFormat

Create objects using Locale .

Initialize Date and Currency

The application can show a local formatted date and currency. The variables are initialized as follows:

```
// More init code precedes
NumberFormat currency;
Double money = new Double(1000000.00);

Date today = new Date();
```

```
DateFormat df;
```

Displaying a Date

Format a date:

Get a DateFormat object based on the Locale .

Call the format method passing the date to format.

```
public void showDate(){

   df = DateFormat.getDateInstance(DateFormat.DEFAULT, currentLocale);
   pw.println(df.format(today) + " " + currentLocale.toString());
}
```

Sample dates:

```
20 juil. 2011 fr_FR
20.07.2011 ru_RU
```

Customizing a Date

DateFormat constants include:

SHORT: Is completely numeric, such as 12.13.52 or 3:30pm

MEDIUM: Is longer, such as Jan 12, 1952

LONG: Is longer, such as January 12, 1952 or 3:30:32pm

FULL: Is completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST

SimpleDateFormat:

A subclass of a DateFormat class

Displaying Currency

Format currency:

Get a currency instance from NumberFormat .

Pass the Double to the format method.

```
public void showMoney(){
   currency = NumberFormat.getCurrencyInstance(currentLocale);
   pw.println(currency.format(money) + " " + currentLocale.toString());
}
```

Sample currency output:

```
1 000 000 руб. ru_RU
1 000 000,00 € fr_FR
￥1,000,000.00 zh_CN
```

Quiz

Which date format constant provides the most detailed information?

LONG

FULL

MAX

COMPLETE

Networking

The following section describes the concept of networking by using sockets.

Sockets

Socket is the name given, in one particular programming model, to the endpoints of a communication link between processes. Because of the popularity of that particular programming model, the term socket has been reused in other programming models, including Java technology. When processes communicate over a network, Java technology uses the streams model. A socket can hold two streams: one input stream and one output stream. A process sends data to another process through the network by writing to the output stream associated with the socket. A process reads data written by another process by reading from the input stream associated with the socket.

After the network connection is set up, using the streams associated with that connection is similar to using any other stream.

Setting Up the Connection

To set up the connection, one machine must run a program that is waiting for a connection, and a second machine must try to reach the first. This is similar to a telephone system, in which one party must make the call, while the other party is waiting by the telephone when that call is made.

A description of the TCP/IP network connections is presented in this module. An example network connection is shown in Figure 13-1.

Figure 13-1 Diagram of Example Network Connections

The same port on the server is reused by multiple client connections. A *connection* is identified by four numbers: client IP address, client port number, server IP address, and server port number. Consequently, this scheme enables the server to remain in place and handle multiple connections from the same client, because the client will allocate different port numbers for each connection that it initiates. This client-port number allocation is handled by the OS. The programmer does not specify the client port, but specifies only the server port. The analogy with the

telephone system works well in this situation. The caller (client) must know both the phone number and the extension of the party that the caller wishes to speak with, but the client's own phone number (IP address) and extension (port) are not important to the person on the phone (although the underlying system uses this information).

Networking With Java Technology

This section describes the concept of networking by using Java technology.

Addressing the Connection

When you make a telephone call, you need to know the telephone number to dial. When you make a network connection, you need to know the address or the name of the remote machine. In addition, a network connection requires a port number, which you can think of as a telephone extension number. After you connect to the proper computer, you must identify a particular purpose for the connection. So, in the same way that you can use a particular telephone extension number to talk to the accounts department, you can use a particular port number to communicate with the accounting program.

Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and the values range from 0–65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless communicating with one of those services (such as `telnet`, Simple Mail Transport Protocol [SMTP] mail, `ftp`, and so on). Client port numbers are allocated by the host OS to something not in use, while server port numbers are specified by the programmer, and are used to identify a particular service. Both client and server must agree in advance on which port to use. If the port numbers used by the two parts of the system do not agree, communication does not occur.

Java Networking Model

In the Java programming language, TCP/IP socket connections are implemented with classes in the `java.net` package.

- The server assigns a port number. When the client requests a connection, the server opens the socket connection with the `accept()` method.
- The client establishes a connection with *`host`* on port *`port#`*.
- Both the client and server communicate by using an `InputStream` and an `OutputStream`.

Minimal TCP/IP Server

TCP/IP server applications rely on the `ServerSocket` and `Socket` networking classes provided by the Java programming language. The `ServerSocket` class takes most of the work out of establishing a server connection.

```
1 import java.net.*;
2 import java.io.*;
3
4 public class SimpleServer {
5 public static void main(String args[]) {
6 ServerSocket s = null;
7
8 // Register your service on port 5432
9 try {
10 s = new ServerSocket(5432);
11 } catch (IOException e) {
12 e.printStackTrace();
13 }
14
15 // Run the listen/accept loop forever
16 while (true) {
17 try {
18 // Wait here and listen for a connection
19 Socket s1 = s.accept();
20
21 // Get output stream associated with the socket
22 OutputStream s1out = s1.getOutputStream();
23 BufferedWriter bw = new BufferedWriter(
24 new OutputStreamWriter(s1out));
25
26 // Send your string!
27 bw.write("Hello Net World!\n");
28
29 // Close the connection, but not the server
socket
30 bw.close();
```

```
31 s1.close();
32 } catch (IOException e) {
33 e.printStackTrace();
34 } // end of try-catch
35 } // end of while(true)
36 } // end of main method
37 } //end of SimpleServer program
```

Minimal TCP/IP Client

The client side of a TCP/IP application relies on the `Socket` class. Again, much of the work involved in establishing connections is performed by the `Socket` class. The client attaches to the server, as presented in ‘‘Minimal TCP/IP Server’’ and then prints everything sent

by the server to the console.

```
1 import java.net.*;
2 import java.io.*;
3
4 public class SimpleClient {
5 public static void main(String args[]) {
6 try {
7 // Open your connection to a server, at port 5432
8 // localhost used here
9 Socket s1 = new Socket("127.0.0.1", 5432);
10
11 // Get an input stream from the socket
12 InputStream is = s1.getInputStream();
13 InputStreamReader irs = new InputStreamReader(is);
14 BufferedReader br = new BufferedReader(isr);
15 // Read the input and print it to the screen
16 System.out.println(br.readLine());
17
18 // When done, close the steam and connection
19 br.close();
20 s1.close();
21 } catch (ConnectException connExc) {
22 System.err.println("Could not connect.");
23 } catch (IOException e) {
```

```
24 // ignore
25 } //end of try-catch
26 } // end of main method
27 } // end of SimpleClient program
```

The URL Class

Java programs can use the `URL` class to connect to a `URL` address to access content pointed to by the URL. This class is part of the `java.net` package. The `URL` class provides several features to connect and retrieve information on the Internet.

The `URL` class can be used to create and parse a URL address, open a connection pointed by the URL address, and read and write to the connection.

Accessing Information From a URL

The following steps are usually associated with connecting to a URL target and accessing the URL target data.

- Create a `URL` object
- Parse the URL
- Connect to the URL target
- Read or write content to the URL connection
- Close the URL connection

The following sections provide more details about these steps.

Creating a URL

The `URL` class provides several constructors for creating the URL. Depending on the information available, any of these constructors can be used to create URLs.

The simplest option is to use an absolute URL address in the form of a string. For example,

```
URL sunAddress = new URL(“http://www.sun.com/products/index.jsp”);
```

The alternative to this is to specify the relative path rather than the absolute path.

```
URL baseAddress = new URL(“http://www.sun.com/”);
URL sunAddress = new URL(baseAddress,”products/index.jsp”);
```

The advantage with this mechanism is that the same `baseAddress` can be used to access other pages in that `baseAddress`. For example

```
URL sunAddress = new URL(baseAddress,”software/index.jsp”);
```

The other forms of creating a `URL` object are useful when the protocol, host name, and the required file name are known. If the port number for the protocol is different from the default port number, another form of URL constructor that takes the port number as the parameter can be used.

Parsing the URL

The `URL` class can be used to retrieve information about the `URL`. For example, Table 13-1 lists some of the methods of the `URL` class and the values returned when these methods are applied to the following URL:
`http://onesearch.sun.com/search/onesearch/index.jsp?qt=JAVA&charset=UTF-8`

The following example uses the `openStream` method to open the connection stream and read from the stream. Code 13-1 shows an example of establishing the connection and reading the content.

Code 13-1 `URLExample` Class

```
1 import java.net.*;
2 import java.io.*;
3
4 public class URLExample {
5 public static void main(String[] args) throws Exception {
6 String oneLine;
7 URL url_address = new URL("http://www.sun.com/");
8 BufferedReader br= new BufferedReader(
9 new InputStreamReader(url_address.openStream()));
10 while ((oneLine = br.readLine()) != null) {
11 System.out.println(oneLine);
12 }
13 br.close();
14 }
15 }
```

The `readLine()` method reads each line from the opened stream and stores in the local variable `oneLine`. The `println` statement on line 11 prints each line of the HTML code.