Intermediate Java Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

INTERMEDIATE JAVA PROGRAMMING

Contributing Authors: John Crabtree, Danielle Hopkins, Julie Johnson, Channing Lovely, Mike Naseef, Jamie Romero, Rob Roselius, Rob Seitz, and Rick Sussenbach.

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Jan Waleri

Editorial Staff: Danielle North

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Jimmy Ball, Larry Burley, Roger Jones, Joe McGlynn, Jim McNally, Mike Naseef, Richard Raab, and Todd Wright, and the many students who have offered comments, suggestions, criticisms, and insights.

CONTENTS

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Use the Java Collections Framework to work with groups of objects.
- Use the **java.awt** and **javax.swing** packages to create GUI applications.
- Write Java programs that interface with databases via JDBC.
- Match patterns in Java programs with regular expressions.

COURSE OVERVIEW

- **Audience**: This course is designed for programmers who wish to expand their knowledge of Java Programming. You will write many programs in this class.

- **Prerequisites**: *Introduction to Java* or equivalent experience is required.

- **Classroom Environment:**

  - One Java development environment per student.

  - DBMS Server.

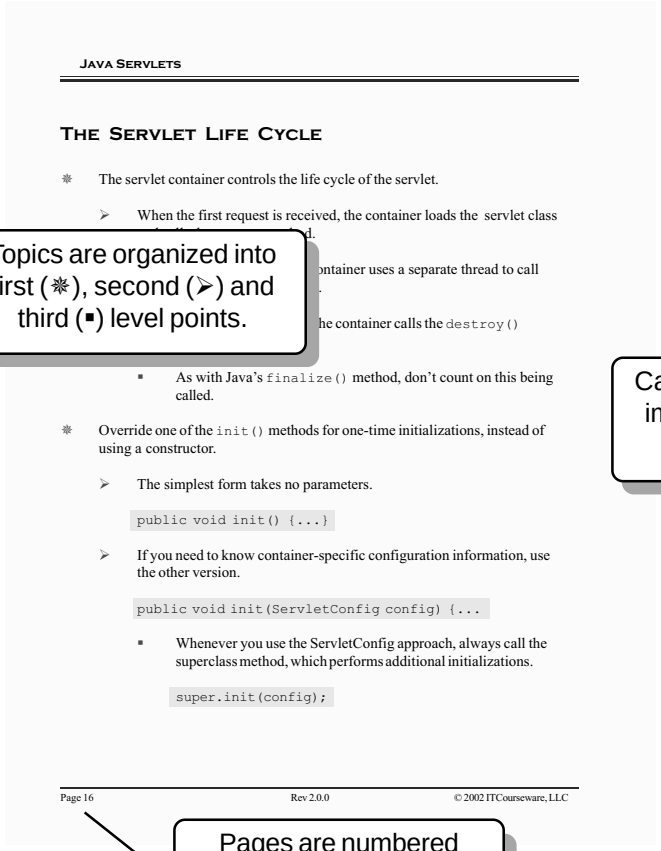
USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. *Hands On* sections provide opportunities for practical application of key concepts. *Try It* and *Investigate* sections help direct individual discovery.

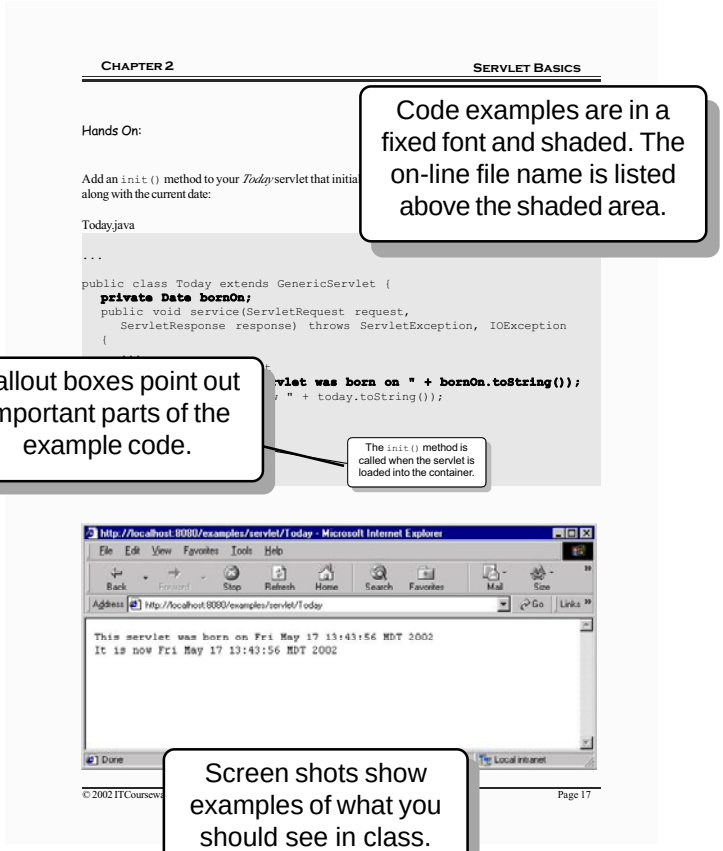
In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.



Topics are organized into first (❋), second (➢) and third (▪) level points.



Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Arnold, Ken, James Gosling, and David Holmes. 2013. ***The Java Programming Language (5th Edition)***. Addison-Wesley, Reading, MA. ISBN 978-0132761680.

Bloch, Joshua. 2008. ***Effective Java (2nd Edition)***. Addison-Wesley, Reading, MA. ISBN 978-0321356680.

Cadenhead, Rogers. 2012. ***Sams Teach Yourself Java in 21 Days (6th Edition)***. Sams, Indianapolis, IN. ISBN 978-0672335747.

Eckel, Bruce. 2006. ***Thinking in Java (4th Edition)***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0131872486.

Horstmann, Cay and Gary Cornell. 2012. ***Core Java 2, Volume I: Fundamentals (9th Edition)***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081899.

Horstmann, Cay and Gary Cornell. 2013. ***Core Java 2, Volume II: Advanced Features (9th Edition)***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081608.

Schildt, Herbert. 2011. ***Java, A Beginner's Guide (5th Edition)***. McGraw Hill, New York, NY. ISBN 978-0071606325.

Schildt, Herbert. 2011. ***Java The Complete Reference (8th Edition)***. McGraw Hill, New York, NY. ISBN 978-0070435926.

Sierra, Kathy and Bert Bates. 2005. ***Head First Java (2nd Edition)***. O'Reilly & Associates, Sebastopol, CA. ISBN 978-0596009205.

http://stackoverflow.com/questions/tagged/java
http://www.javaworld.com
http://www.javaranch.com
http://www.oracle.com/technetwork/java

CHAPTER 2 - COLLECTION SORTING AND TUNING

OBJECTIVES

- Use the **Comparable** and **Comparator** interfaces to sort your collections.
- Use the various methods available in the **Collections** class.
- Tune **ArrayList**, **HashSet**, and **HashMap** using various constructors.

SORTING WITH COMPARABLE

- The **java.util** package defines two classes that allow you to create sorted collections.
  - The **TreeSet** implementation represents a sorted **Set**.
  - The **TreeMap** implementation sorts a **Map** based on the key.
- Both classes rely on each member of the collection implementing the **java.lang.Comparable** interface.

```
public class Planet implements Comparable<Planet> {
```

  - The **compareTo()** method defines whether the given object is greater than, less than, or equal to a passed-in object.

```
public int compareTo(Planet p) {
```

    - Return a positive integer for greater than, a negative integer for less than, and zero for equal.
  - Each time an element is added to the collection, its **compareTo()** method is called and it is stored based on its relation to other members of the collection.
- Many of the Java SE library classes implement **Comparable**, including **String** and the primitive wrapper classes.

Planet.java

```
...
public class Planet implements Comparable<Planet> {
    private final String name;
    private final long orbit;
    private final int diameter;
    ...
    @Override
    public int compareTo(Planet other) {
        if (this.orbit < other.orbit) {
            return -1;
        }
        else if (this.orbit > other.orbit) {
            return 1;
        }
        else {
            return 0;
        }
    }
    ...
}
```

Sort based on closest orbit around the Sun.

SortPlanets.java

```
...
public class SortPlanets {

    public static void main(String[] args) {
        Set<Planet> planets = new TreeSet<>();

        planets.add(new Planet("Earth", 149_600_000, 12_756));
        planets.add(new Planet("Jupiter", 778_330_000, 142_984));
        planets.add(new Planet("Mars", 227_940_000, 6_794));
        planets.add(new Planet("Mercury", 57_910_000, 4_880));
        planets.add(new Planet("Neptune", 4_504_000_000L, 49_532));
        planets.add(new Planet("Saturn", 1_429_400_000, 120_536));
        planets.add(new Planet("Uranus", 2_870_990_000L, 51_118));
        planets.add(new Planet("Venus", 108_200_000, 12_103));

        for (Planet planet : planets) {
            System.out.println(planet);
        }
    }
}
```

TreeSet sorts elements as you add them.

Try It:

Run *SortPlanets.java* to list the planets from closest to farthest from the Sun.

SORTING WITH COMPARATOR

- Use **Comparable** if you can easily modify the source code for the objects you wish to sort.
  - If you don't have access to the source code, you can instead create a subclass of **java.util.Comparator** to specify the sorting behavior.

```
public class String implements Comparator<String> {
```

  - You may also choose to write a **Comparator** in situations where you'd like to provide an alternative implementation for an existing **Comparable**.

- The class that implements the **Comparator** interface must define the **compare()** method.

```
public int compare(String a, String b) {
```

  - The method takes two parameters representing the two objects that need to be compared against each other and returns an **int**.
  - The method defines whether the first object is greater than, less than, or equal to the second object.
    - Returns a positive integer for greater than, a negative integer for less than, and zero for equal.

- When working with a **TreeSet** or a **TreeMap**, you can provide an additional argument to the constructor to specify the **Comparator** to be used for sorting.

```
Comparator<String> comp = new StringComparator();
Set<String> set = new TreeSet<>(comp);
```


StringComparator.java

```
package examples;

import java.util.Comparator;

public class StringComparator implements Comparator<String> {
    public int compare(String a, String b) {
        return a.toUpperCase().compareTo(b.toUpperCase());
    }
}
```

Case-insensitive sort.

SortStrings.java

```
package examples;

import java.util.Set;
import java.util.TreeSet;

public class SortStrings {

    public static void main(String[] args) {
        Set<String> students = new TreeSet<>();
        // Set<String> students = new TreeSet<>(new StringComparator());
        students.add("James");
        students.add("Jack");
        students.add("joseph");
        students.add("Jim");
        students.add("Juan");

        for (String student : students) {
            System.out.println(student);
        }
    }
}
```

Uncomment this line to use the **Comparator.**

Try It:

Run *SortStrings.java* to see the names sorted in alphabetical order. Notice how lowercase letters sort after uppercase letters. Modify the code to use the **StringComparator** and run it again to see case-insensitive sorting.

Sorting Lists and Arrays

- Use **TreeSet** or **TreeMap** to sort **Set**s or **Map**s.
- To sort a **List**, pass it to the **sort()** method of the **Collections** class.
  - Use the one-parameter version of the **sort()** method if all elements of the **List** implement **Comparable**.
    - Otherwise, pass a **Comparator** as the second parameter.
- The **Arrays** class has multiple **sort()** methods defined to allow you to sort Java language arrays.

PlanetDiameterComparator.java

```
...
public class PlanetDiameterComparator implements Comparator<Planet> {
    public int compare(Planet a, Planet b) {
        if (a.getDiameter() < b.getDiameter())
            return -1;
        else if (a.getDiameter() > b.getDiameter())
            return 1;
        else
            return a.getName().compareTo(b.getName());
    }
}
```

Sort based on planet diameter.

SortPlanets2.java

```
...
public class SortPlanets2 {

    public static void main(String[] args) {
        List<Planet> planets = new ArrayList<>();

        planets.add(new Planet("Mercury", 57_910_000, 4_880));
        planets.add(new Planet("Venus", 108_200_000, 12_103));
        planets.add(new Planet("Mars", 227_940_000, 6_794));
        planets.add(new Planet("Earth", 149_600_000, 12_756));
        planets.add(new Planet("Jupiter", 778_330_000, 142_984));
        planets.add(new Planet("Saturn", 1_429_400_000, 120_536));
        planets.add(new Planet("Uranus", 2_870_990_000L, 51_118));
        planets.add(new Planet("Neptune", 4_504_000_000L, 49_532));

        PlanetDiameterComparator comparator =
                new PlanetDiameterComparator();
        Collections.sort(planets, comparator);
        for (Planet planet : planets) {
            System.out.println(planet);
        }
    }
}
```

Pass the **Comparator** to the **sort()** method.

Try It:

Run *SortPlanets2.java* to list the planets from the smallest to the largest.

COLLECTIONS UTILITY METHODS

- In addition to **sort()**, the **Collections** class has many other utility methods defined for working with lists.
  - **reverse(*list*)** — reverses the order of the elements of the passed-in ***list***.
  - **rotate(*list, dist*)** — rotates the elements of the passed-in ***list*** by the specified distance.
  - **shuffle(*list*)** — randomly shuffles the order of the elements of the passed-in ***list***.
  - **binarySearch(*list, object*)** — uses a binary search algorithm to locate the given ***object*** within the ***list***.
    - The **int** return value represents the index of the first match; **-1** indicates no match was found.
    - For the algorithm to work properly, make sure to **sort()** the ***list*** before calling **binarySearch()**.
  - **fill(*list, object*)** — replaces all elements within the given ***list*** with the passed-in ***object***.
- Other **Collections** methods are available for working with all of the collection types.
  - **synchronizedList(*list*)** — returns a thread-safe version of ***list***.
    - Versions of this method exist for **Map** and **Set** as well.
  - **unmodifiableList(*list*)** — returns a read-only version of ***list***.
    - Versions of this method also exist for **Map** and **Set**.

The **Arrays** class contains many similar methods for use with Java language arrays.

ListPlanets.java

```
...
public class ListPlanets {
    public static void main(String args[]) {
        new ListPlanets();
    }

    public ListPlanets() {
        List<Planet> planets = new ArrayList<>();

        planets.add(new Planet("Mercury", 57_910_000, 4_880));
        planets.add(new Planet("Venus", 108_200_000, 12_103));
        planets.add(new Planet("Mars", 227_940_000, 6_794));
        planets.add(new Planet("Earth", 149_600_000, 12_756));
        planets.add(new Planet("Jupiter", 778_330_000, 142_984));
        planets.add(new Planet("Saturn", 1_429_400_000, 120_536));
        planets.add(new Planet("Uranus", 2_870_990_000L, 51_118));
        planets.add(new Planet("Neptune", 4_504_000_000L, 49_532));

        System.out.println("\n**Original List**");
        printPlanets(planets);

        System.out.println("\n**After Reverse**");
        Collections.reverse(planets);
        printPlanets(planets);

        System.out.println("\n**After Rotate**");
        Collections.rotate(planets, 2);
        printPlanets(planets);

        System.out.println("\n**After Shuffle**");
        Collections.shuffle(planets);
        printPlanets(planets);
    }

    private void printPlanets(List<Planet> planets) {
        for (Planet planet : planets) {
            System.out.println(planet);
        }
    }
}
```

Note:
The **Collections** class is different from the **Collection** interface.

TUNING ARRAYLIST

- When an **ArrayList** is instantiated, a Java language array is created to store the list elements.

- The no-arg constructor creates an array of size ten by default.

  - Each time the size of the **ArrayList** exceeds the length of the underlying array, the array must grow.

    1. Memory is allocated for a new array based on the formula: newCapacity = (oldCapacity * 3)/2 + 1.

    2. A **System.arraycopy()** is performed to move the elements to the new array.

    3. The old array is eligible for Garbage Collection.

- Pass an **int** to the **ArrayList** constructor to specify an initial size of the array to minimize the growth.

```
List<String> states = new ArrayList<>(50);
```

  - You can also use the **ensureCapacity(int)** method to grow the array independent of the constructor.

- The **trimToSize()** method can be used to shrink the underlying array to the actual size of the **ArrayList** in order to conserve memory.

  - A new array is allocated to the size of the **ArrayList** and an **arraycopy()** is performed.

Rainfall.java

```
package examples;

import java.util.ArrayList;
import java.util.List;

public class Rainfall {
    public static void main(String[] args) {
        List<Float> monthlyRainfall = new ArrayList<>(12);

        monthlyRainfall.add(5.41F);
        monthlyRainfall.add(4.78F);
        monthlyRainfall.add(6.39F);
        monthlyRainfall.add(3.91F);
        monthlyRainfall.add(4.38F);
        monthlyRainfall.add(6.37F);
        monthlyRainfall.add(7.99F);
        monthlyRainfall.add(6.60F);
        monthlyRainfall.add(5.83F);
        monthlyRainfall.add(3.96F);
        monthlyRainfall.add(4.46F);
        monthlyRainfall.add(3.92F);

        float total = 0.0F;
        for (float amount : monthlyRainfall) {
            total += amount;
        }
        System.out.println("Pensacola, FL");
        System.out.printf("Avg monthly rainfall  = %.2f\n",
                total / 12.0);
        System.out.printf("Total yearly rainfall = %.2f\n", total);
    }
}
```

Give an initial size equal to what will be used.

TUNING HASHMAP AND HASHSET

- When instantiating a **HashMap** you can provide an *initial capacity* to the constructor.
  - The capacity of the **HashMap** determines how many "buckets" are stored in the underlying hash table implementation; the default is **16**.
- You can also specify a load factor, as a **float**, in addition to the initial capacity.
  - The *load factor* determines how full the hash table can be before the capacity is increased.
  - **HashMap** retrievals perform best when most of the buckets contain no more than one object.
    - The higher the capacity, the less the chance of collisions.
  - The default value for load factor is **0.75**.
    - This usually provides a good balance between speed and memory usage.
    - Higher values decrease the memory overhead — you will have less empty buckets, more collisions, and slower retrievals.
- Try to anticipate the total number of entries in the map and add extra capacity to allow for the load factor.
  - A **HashMap** that will hold 500 elements with a load factor of 0.75 should have an initial capacity of at least 667.
- **HashSet** uses the keys of an underlying **HashMap** to store its unique elements.
  - Tuning characteristics of a **HashMap** apply to a **HashSet** as well.

Hash table algorithms typically disperse elements into an underlying array of linked lists. Each array index corresponds to a computed hash for the objects that are added in. If two objects both have the same hash calculated (a collision), they will both be accessible from the same index. This is achieved by creating a linked list of objects that all have the same hash value. Each array position is typically called a *bucket*.

Hash tables are at their fastest when each bucket contains one object. This allows for maximum retrieval speeds, because accessing an object is equivalent to retrieving an element based on an index.

The more objects that are stored in the same bucket, the slower the overall retrieval performance. Not only is an indexed lookup performed, but also each linked list is iterated over, one element at a time.

Increasing the capacity results in better retrieval performance. The more buckets, the fewer contain more than one object. Lower capacities tend to decrease performance, because there are more chances for collisions.

LABS

❶ *Card.java, Deck.java, Rank.java,* and *Suit.java* contain code that simulates a deck of playing cards. Modify *Card.java* and *Deck.java* so that the cards are printed in a sorted order, with rank sorting before suit.
(Solutions: *Card2.java, Deck2.java*)

❷ Modify *Deck.java* again so that in addition to printing the sorted cards, you also print out the same cards after they have been shuffled.
(Solution: *Deck3.java*)

❸ The *access_log* file contains records of hits to a web site. Consider each line equivalent to one hit. Write a Java application that counts the number of hits from each unique visitor. Then display the unique visitors along with the corresponding number of total hits. (Hint: Use **String**'s **split()** method to get the first field of each line, which represents the hostname of the visitor.)
(Solution: *ShowHits.java*)

❹ (Optional) Modify your *ShowHits* program to sort the displayed records according to the total number of hits.
(Solutions: *ShowSortedHits.java, AccessLogComparator.java*)

CHAPTER 5 - INTRODUCTION TO JDBC

OBJECTIVES

- Describe the Java Database Connectivity model.
- Write a simple Java program that uses JDBC.

THE JDBC CONNECTIVITY MODEL

- JDBC provides a simple but complete programming interface for accessing Database Management Systems (DBMSs).

- Conceptually, JDBC resembles Microsoft's Open Database Connectivity (ODBC) API.

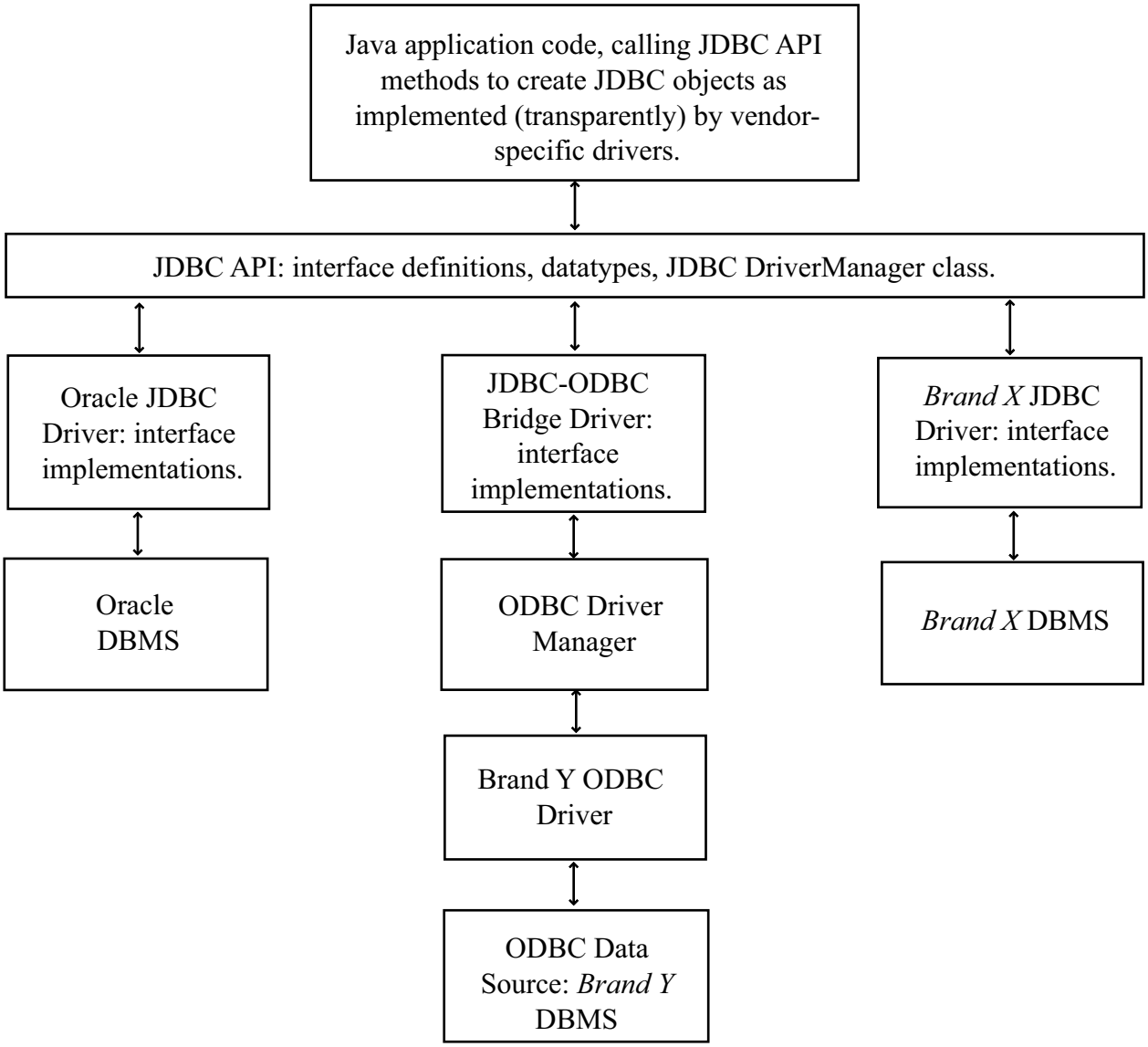
  - Application programmers access a generic API, and a driver manager controls connections.

  - Vendor/product-specific drivers implement the connection and interaction with specific DBMSs.

- The **java.sql** and **javax.sql** packages define the JDBC interfaces and datatypes for dealing with database connections, statements, query results, etc.

  - Normally, you will work with just a few relatively simple interfaces and methods.

  - JDBC also provides features for tool builders and driver writers.



JDBC supports four different types of drivers:

Type 1 — A JDBC-ODBC bridge uses an ODBC driver to talk to the DBMS server.
Type 2 — A native driver uses a platform-specific library (*.dll*, etc.) to access the DBMS server.
Type 3 — A pure Java middleware driver uses a standard protocol to communicate with the DBMS server.
Type 4 — A pure Java "thin" driver uses a vendor-specific protocol to access the DBMS server.

DATABASE PROGRAMMING

- Programmers writing applications that interact with a DBMS typically have four basic steps to complete:

    1. Create variables for database data.

    2. Connect to the DBMS.

    3. Do whatever it is you need to do with the database data.

    4. Disconnect from the DBMS.

- If what you need to do with the database data is to retrieve a set of records, the basic steps are:

    3.1. Declare a *cursor* — an object associated with a specific DBMS query statement.

    3.2. Open the cursor — this executes the query statement, and identifies its *result set*.

    3.3. Iterate through the records that make up the result set.

    3.4. Close the cursor.

- All of these standard database programming tasks are implemented simply in JDBC.

Hands On:

Create a new Java program, *JDBCTest.java*. Import **java.sql.*** so that you can use the **JDBC API**. Give it a **main()** method. It will use methods that throw **SQLException**s. For now, rather than catching it, just declare **main()** as throwing it.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
  public static void main(String args[]) throws SQLException {
  }
}
```

CONNECTING TO THE DATABASE

- To connect your Java program to the database invoke the **DriverManager.getConnection()** static method.
  - **getConnection()** takes a **String** argument (the URL for a connection) and returns a **Connection** object.

```
String URL = "jdbc:derby://localhost:1527/java";
Connection conn = DriverManager.getConnection(URL);
```

    - Authentication information (username, password) may be embedded in the URL, or may be passed as separate arguments to **getConnection()**.

Hands On:

In *JDBCTest.java*, declare a **String** variable named **URL** and initialize it with the correct connection URL for your classroom database. Use the URL to obtain a connection from the **DriverManager**.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
   public static void main(String args []) throws SQLException {
      String URL = "jdbc:derby://localhost:1527/java";
      Connection conn = DriverManager.getConnection(URL);
      conn.close();
   }
}
```

Note:

This code uses the Apache Derby driver. Other drivers will require other URL strings, and may require a username and password to get the connection. Your instructor will provide the specific connection URL for your environment as well as instructions for starting the database.

You may also need to modify your **CLASSPATH** environment variable to include the location of your JDBC driver classes:

UNIX:

```
CLASSPATH=.:/oracle_dir/classes12.zip:$CLASSPATH
export CLASSPATH
```

Windows:

```
set CLASSPATH=.;c:\oracle_dir\classes12.zip;%CLASSPATH%
```

If you are using Eclipse, rather than setting your **CLASSPATH**, ask your instructor how to configure the **Java Build Path** so that the driver is available to your project.

CREATING A SQL QUERY

- To execute SQL statements, you must create a **Statement** object.
  - Use **Connection**'s **createStatement()** method to do so.
- The **Statement** interface provides methods for executing SQL queries and updates.
  - The methods we'll use for now take a single **String** argument: a SQL statement.
- You can reuse a **Statement** object to execute additional SQL statements.
  - SQL executed by a **Statement** object is parsed and compiled by the DBMS on each execution.
  - You can use a single **Statement** object to execute any number of SQL statements at different points in your program.

Hands On:
Declare a **String** variable. This will be the text of the SQL statement. Use **Connection.createStatement()** to create a **Statement** object.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
   public static void main(String args []) throws SQLException {
      String URL = "jdbc:derby://localhost:1527/java";
      Connection conn = DriverManager.getConnection(URL);

      String sqltxt;
      sqltxt = "SELECT ID, FIRSTNAME, LASTNAME FROM EMPLOYEE";
      Statement stmt = conn.createStatement();
      stmt.close();

      conn.close();
   }
}
```

We'll execute our SQL statement between these lines.

GETTING THE RESULTS

- Executing a SQL query (that is, a SQL **SELECT** statement) produces a *result set* — a "virtual table" consisting of the rows and column values retrieved by the **SELECT** statement.

- If you are making a SQL query, use **Statement**'s **executeQuery()** method.

  - **executeQuery()** takes a **String** argument, the SQL **SELECT** statement.

  - **executeQuery()** returns a **ResultSet** object.

  - The **ResultSet** object represents the opened cursor for the SQL query.

- Use your **ResultSet** object's **next()** method to retrieve each row, in turn, of the result set.

  - **next()** returns **false** after the last row has been returned.

- **ResultSet**'s **getString(*col*)** method returns the **String** representation of the column whose name or number (left-to-right, starting with **1**, as listed in the **SELECT** clause) is ***col***.

  - Other **ResultSet get*XXX*()** methods return DBMS values as different Java datatypes, performing any necessary conversion.

- The **close()** method of either the **ResultSet** or its **Statement** closes the cursor and frees resources.

Hands On:
Declare a **ResultSet** variable and use **executeQuery()** to get a **ResultSet** object, passing your SQL string to **executeQuery()**. Use a **while** loop to iterate through the **ResultSet**'s rows, calling **ResultSet.next()**. Get and print out the values of all columns retrieved.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
   public static void main(String args []) throws SQLException {
      String URL = "jdbc:derby://localhost:1527/java";
      Connection conn = DriverManager.getConnection(URL);

      String sqltxt;
      sqltxt = "SELECT ID, FIRSTNAME, LASTNAME FROM EMPLOYEE";
      Statement stmt = conn.createStatement();

      ResultSet rs = stmt.executeQuery(sqltxt);
      while(rs.next()) {
         System.out.println(rs.getString(1) + " " +
                            rs.getString(2) + " " +
                            rs.getString(3));
      }
      rs.close();

      stmt.close();
      conn.close();
   }
}
```

The **ResultSet** interface also allows for scrolling and absolute positioning. These methods require a scrollable **ResultSet**. Create one by specifying scrollability when you create the **Statement**:

```
Statement stmt = conn.createStatement(resultSetType, concurrency);
```

Where *resultSetType* is one of three values:

```
ResultSet.TYPE_FORWARD_ONLY          // non-scrollable but fast
ResultSet.TYPE_SCROLL_INSENSITIVE    // scrollable but doesn't see
                                     // others' changes
ResultSet.TYPE_SCROLL_SENSITIVE      // scrollable and does see
                                     // others' changes
```

To move around in a scrollable **ResultSet**, **previous()**, **first()**, **last()**, and other methods have been provided.

UPDATING DATABASE DATA

- Executing a Data Manipulation Language (DML) statement (that is, a SQL **INSERT**, **UPDATE**, or **DELETE** statement) modifies data in the DBMS.

- If you are doing a DML statement, use **Statement**'s **executeUpdate()** method.

  - **executeUpdate()** takes a **String** argument, the SQL statement.

- **executeUpdate()** returns an **int** — the number of database rows affected by the statement.

  - This update count may be zero.

Hands On:

Change the SQL string to an **update** statement which will update one row. Execute the statement with **executeUpdate()**, capturing and printing the update count.

JDBCTest.java

```
package examples;

import java.sql.*;

public class JDBCTest {
   public static void main(String args []) throws SQLException {
      String URL = "jdbc:derby://localhost:1527/java";
      Connection conn = DriverManager.getConnection(URL);

      String sqltxt;
      sqltxt = "SELECT ID, FIRSTNAME, LASTNAME FROM EMPLOYEE";
      Statement stmt = conn.createStatement();

      ResultSet rs = stmt.executeQuery(sqltxt);
      while(rs.next()) {
         System.out.println(rs.getString(1) + " " +
                            rs.getString(2) + " " +
                            rs.getString(3));
      }
      rs.close();

      sqltxt = "UPDATE EMPLOYEE SET LASTNAME = 'Smithers' " +
               " WHERE id = 9883";
      int uc = stmt.executeUpdate(sqltxt);
      System.out.println("\n" + uc + " row(s) updated.");

      stmt.close();
      conn.close();
   }
}
```

FINISHING UP

- Typically, the open cursor associated with a **ResultSet** object consumes DBMS resources (locks, shared memory, temporary table space, etc.).
- **Connection** and **Statement** objects also may hold DBMS resources.
- When a **Connection**, **Statement**, or **ResultSet** object goes out of scope, it is subject to Garbage Collection.
  - Garbage Collection should free the resources associated with a Java database object, but . . .
  - Don't count on Garbage Collection; free cursor and other resources explicitly when your program is finished with them.
  - Each interface provides a **close()** method to do this.
- When a **Statement** is closed, any **ResultSet**s associated with it are automatically closed.
- Closing a **Connection** normally disconnects you from the database.
- You can use the Java 7 try-with-resources syntax to automatically close your **Connection**, **Statement**, and/or **ResultSet**.
  - Embed the declarations of the closeable resources within a set of parenthesis after the **try** keyword.

```
try (Connection c = DriverManager.getConnection(URL);
     Statement stmt = c.createStatement();
     ResultSet rs = stmt.executeQuery(sqltxt);) {
   ...
}
```

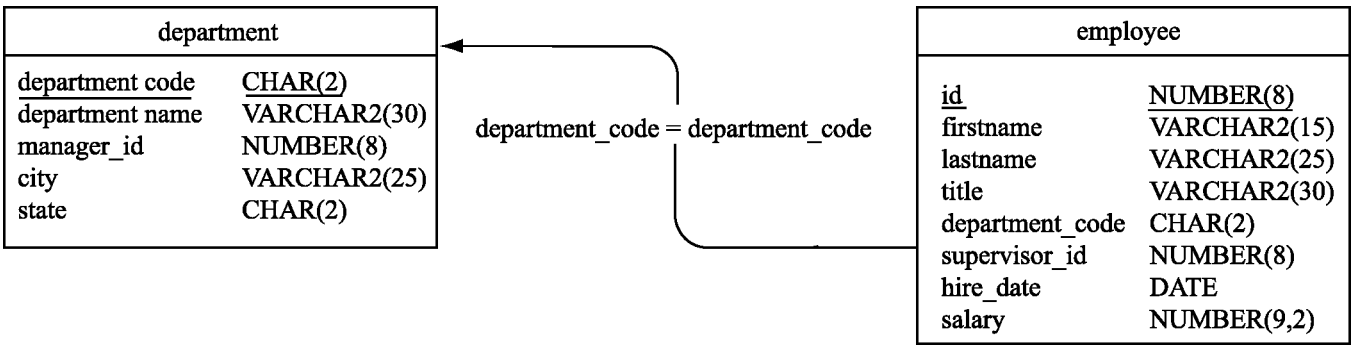
  - Omit the explicit calls to the **close()** methods on these resources.

EmployeeQuery.java

```
package examples;

import java.sql.*;

public class EmployeeQuery {
    public static void main(String args[]) {
         String URL = "jdbc:derby://localhost:1527/java";
         String sqltxt = "SELECT ID, FIRSTNAME, LASTNAME FROM EMPLOYEE";

         try (Connection conn = DriverManager.getConnection(URL);
                 Statement stmt = conn.createStatement();
                 ResultSet rs = stmt.executeQuery(sqltxt);) {

             while (rs.next()) {
                 System.out.println(rs.getString(1) + " "
                         + rs.getString(2) + " " + rs.getString(3));
             }
         }
         catch (SQLException e) {
             System.err.println(e);
         }
    }
}
```

LABS

❶ Write a program to insert a record with your name and all other information into the **employee** table. If it works, try it again. Does it work a second time? If not, why not?
(Solution: *PutMeIn.java*)



Example SQL **INSERT** statement:

```
INSERT INTO EMPLOYEE
   ( ID, FIRSTNAME, LASTNAME, TITLE, DEPARTMENT_CODE, SUPERVISOR_ID,
     HIRE_DATE, SALARY )
  VALUES ( 9999, 'Rob', 'Roselius', 'Lord High Coder', 'RD', NULL,
     '2012-11-30', 450000.00 )
```