
Java Programming Book 2



7400 E. Orchard Road, Suite 1450 N, Greenwood Village, CO 80111
303-302-5234 | 800-292-3716
SkillDistillery.com

JAVA PROGRAMMING

BOOK 2

Student Workbook



For Skill Distillery student use only.
Unauthorized distribution or reproduction is prohibited.

JAVA PROGRAMMING - BOOK 2

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., 7400 E. Orchard Rd, Suite 1450N, Greenwood Village, CO 80111

Editor: Jan Waleri

Editorial Staff: Ginny Jaranowski

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.

Copyright © 2016 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photo-copying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC., 7400 E. Orchard Rd, Suite 1450N, Greenwood Village, Colorado, 80111. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

Chapter 1 - Course Introduction	9
Course Objectives	10
Course Overview	12
Using the Workbook	13
Suggested References	14
Chapter 2 - Generics and Lists	17
Collections	18
Generics	20
The List Interface	22
List Implementation Classes	24
Labs	26
Chapter 3 - Input/Output Streams	29
Overview of Streams	30
Bytes vs. Characters	32
File Object	34
Binary Input and Output	36
PrintWriter Class	38
Reading and Writing Objects	40
Closing Streams	42
Labs	44
Chapter 4 - Regular Expressions	47
Pattern Matching and Regular Expressions	48
Regular Expressions in Java	50
Regular Expression Syntax	52
Special Characters	54
Quantifiers	56
Assertions	58
The Pattern Class	60
The Matcher Class	62
Capturing Groups	64
Labs	66

Chapter 5 - Core Collection Classes	69
The Collections Framework	70
The Set Interface	72
Set Implementation Classes	74
The Map Interface	76
Map Implementation Classes	78
Labs	80
Chapter 6 - Collection Sorting and Tuning	83
Sorting with Comparable	84
Sorting with Comparator	86
Sorting Lists and Arrays	88
Collections Utility Methods	90
Tuning ArrayList	92
Tuning HashMap and HashSet	94
Labs	96
Chapter 7 - Inner Classes	99
Inner Classes	100
Member Classes	102
Local Classes	104
Anonymous Classes	106
Instance Initializers	108
Labs	110
Chapter 8 - Introduction to Swing	113
AWT and Swing	114
Displaying a Window	116
GUI Programming in Java	118
Handling Events	120
Arranging Components	122
A Scrollable Component	124
Configuring Components	126
Menus	128
Using the JFileChooser	130
Labs	132

Chapter 9 - Introduction to Threads	135
Non-Threaded Applications	136
Threaded Applications	138
Creating Threads	140
Thread States	142
Runnable Threads	144
Coordinating Threads	146
Interrupting Threads	148
Runnable Interface	150
Labs	152
 Chapter 10 - Thread Synchronization and Concurrency	 155
Race Conditions	156
Synchronized Methods	158
Deadlocks	160
Synchronized Blocks	162
Synchronized Collections	164
Thread-Aware Collections	166
Executor	168
Callable	170
Labs	172
 Chapter 11 - Introduction to JDBC	 175
The JDBC Connectivity Model	176
Database Programming	178
Connecting to the Database	180
Creating a SQL Query	182
Getting the Results	184
Updating Database Data	186
Finishing Up	188
Labs	190

Chapter 12 - JDBC SQL Programming	193
Error Checking and the SQLException Class	194
JDBC Types	196
Executing SQL Queries	198
ResultSetMetaData	200
Executing SQL Updates	202
The execute() Method	204
Using a PreparedStatement	206
Parameterized Statements	208
Transaction Management	210
Labs	212
Index	215

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

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

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:** *Java Programming - Book 1.*
- * **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.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
 - When the first request is received, the container loads the servlet class
 - The container uses a separate thread to call
 - The container calls the destroy()
- As with Java's finalize() method, don't count on this being called.
- * Override one of the init() methods for one-time initializations, instead of using a constructor.
 - The simplest form takes no parameters.


```
public void init() {...}
```
 - If you need to know container-specific configuration information, use the other version.


```
public void init(ServletConfig config) {...}
```
 - Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.


```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

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

CHAPTER 2 **SERVLET BASICS**

Hands On:

Add an init() method to your *Today* servlet that initializes along with the current date:

Today.java

```
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
        ServletResponse response) throws ServletException, IOException
    {
        ...
        Servlet was born on " + bornOn.toString();
        " + today.toString();
    }
}
```

The init() method is called when the servlet is loaded into the container.

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.

© 2002 ITCourseware, LLC

Page 17

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 - GENERICS AND LISTS

OBJECTIVES

- * Create classes that use generics to eliminate downcasting.
- * Use an **ArrayList** as a resizable alternative to a Java language array.

COLLECTIONS

* Collections are objects that hold and manipulate other objects, in a well-defined way.

➤ You can build your own collection classes or use the ones provided for you in the **java.util** package.

* A collection class can be defined to contain data of type **Object**, and can then hold objects of any given class type, since all objects can be implicitly cast to type **Object**.

```
public class Pair {
    private Object first;
    private Object second;
```

➤ A **Pair** can represent and hold two of anything.

```
Pair p1 = new Pair("ABC", "XYZ");
Pair p2 = new Pair(new Integer(1), new Integer(2));
```

* Downcasting is necessary to retrieve the original type from the collection.

```
String st = (String)p1.getFirstElement();
Integer i = (Integer)p2.getFirstElement();
```

➤ Downcasting is prone to **ClassCastException**, unless you check the datatype with the **instanceof** operator.

```
if (p2.getFirstElement() instanceof String)
    String s = (String)p2.getFirstElement();
```

Pair.java

```
...
public class Pair {
    private Object first;
    private Object second;

    public Pair(Object one, Object two) {
        first = one;
        second = two;
    }
    public Object getFirstElement() {
        return first;
    }
    public Object getSecondElement() {
        return second;
    }
    public void setFirstElement(Object obj) {
        first = obj;
    }
    public void setSecondElement(Object obj) {
        second = obj;
    }
}
```

PairTester.java

```
...
public class PairTester {
    public static void main(String[] args) {
        Pair pres = new Pair("George", "Washington");

        String first = (String) pres.getFirstElement();
        String last = (String) pres.getSecondElement();

        System.out.println(first);
        System.out.println(last);
    }
}
```

Try It:

Run *PairTester.java* to try out downcasting with our **Pair** collection.

GENERICS

- * Java introduced *generics* (also called *parameterized types*) to simplify a developer's code when using collections.

- The collection object is passed the datatype of the objects it will store when it is instantiated.

```
GenericPair<String> p1 = new GenericPair<>("ABC", "XYZ");
GenericPair<Integer> p2 = new
    GenericPair<>(new Integer(1), new Integer(2));
```

- Using generics, you no longer have to downcast when retrieving elements from a collection.

```
String s = p1.getFirstElement();
Integer i = p2.getFirstElement();
```

- The compiler checks the datatype that you will retrieve, so there is no danger of a **ClassCastException**.

```
String s = p2.getFirstElement(); // compiler error
```

- * When declaring a class that will store generics, add a simple identifier within angle brackets to the class declaration.

```
public class GenericPair<T> {}
```

- Use the same identifier for the datatypes of fields and parameters that are of the passed-in type.

```
public T getFirstElement() {}
```

- The identifier is replaced throughout the code with the actual datatype that is specified when the object is used.

```
GenericPair<String> p1 = new GenericPair<>("A", "B");
```

GenericPair.java

```
...
public class GenericPair<T> {
    private T first;
    private T second;

    public GenericPair(T one, T two) {
        first = one;
        second = two;
    }
    public T getFirstElement() {
        return first;
    }
    public T getSecondElement() {
        return second;
    }
    public void setFirstElement(T obj) {
        first = obj;
    }
    public void setSecondElement(T obj) {
        second = obj;
    }
}
```

All of the Ts will be replaced with the correct datatype name.

GenericPairTester.java

```
...
public class GenericPairTester {
    public static void main(String[] args) {
        GenericPair<String> pres =
            new GenericPair<>("George", "Washington");

        String first = pres.getFirstElement();
        String last = pres.getSecondElement();

        System.out.println(first);
        System.out.println(last);
    }
}
```

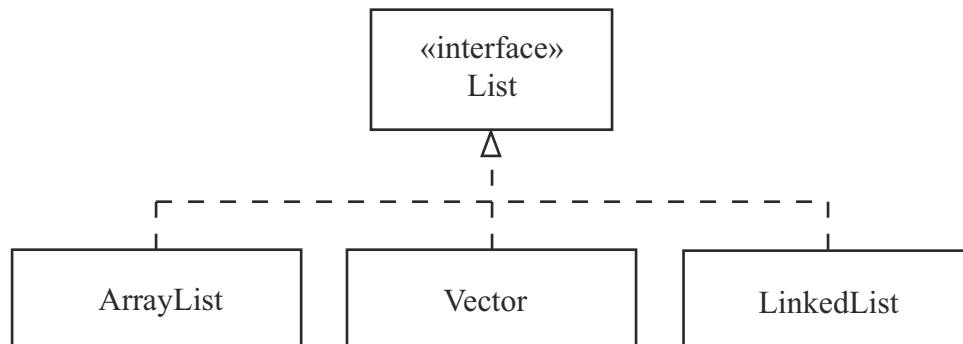
Prior to Java 7, you were required to repeat the datatype within the <> braces.

No downcast necessary.

THE LIST INTERFACE

- * One of the collection types defined in the **java.util** package is **List**.
 - **List** guarantees insertion-order; the order you add elements into a list is the same order that you will iterate through them.
 - **List** allows duplicate elements.
- * Use the **size()** method to determine the number of elements in the **List**.
- * Use the **add(*element*)** method to add an element to the end of the **List**.
- * **List** also provides methods for indexed access to the list elements.
 - **add(*index*, *element*)** inserts an element into a specific position within the **List**.
 - **set(*index*, *element*)** overwrites an element at the specific position within the **List**.
 - **get(*index*)** retrieves an element from a specific position within the **List**.
 - All indexing is zero-based.
- * **List** is an interface.
 - You will actually instantiate one of its implementing classes like **ArrayList**, **Vector**, or **LinkedList**.

```
List<String> myList = new ArrayList<>();
```



LIST IMPLEMENTATION CLASSES

- * An **ArrayList** is a resizable array that implements the **List** interface.
 - An **ArrayList** follows an indexed ordering scheme similar to arrays.
 - It resembles the legacy container class **Vector**.
 - **Vector** methods are **synchronized** for thread safety, while **ArrayList** methods are not.
 - Use an **ArrayList** when an application frequently needs to retrieve an element from the middle of the **List**.
 - Positional access is fast, because each element in an **ArrayList** is associated with an index.
- * The **LinkedList** class implements a doubly-linked list.
 - Like **ArrayList** objects, **LinkedList** objects can have duplicates and maintain the order of elements.
 - **LinkedLists** are useful when an application needs to frequently insert elements into the beginning or the end of the list.
 - Positional access is slower with **LinkedList** than with **ArrayList**, because only head and tail references are maintained.

Planet.java

```
...
public class Planet {
    private final String name;
    private final long orbit;
    private final int diameter;

    public Planet(String name, long orbit, int diameter) {
        this.name = name;
        this.orbit = orbit;
        this.diameter = diameter;
    }
    ...
}
```

PlanetTest.java

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

public class PlanetTest {
    public void printPlanets() {
        List<Planet> planets = new ArrayList<>();
//        List<Planet> planets = new LinkedList<>();

        planets.add(new Planet("Mercury", 57_910_000, 4_880));
        planets.add(new Planet("Venus", 108_200_000, 12_103));
        planets.add(new Planet("Earth", 149_600_000, 12_756));
        planets.add(new Planet("Mars", 227_940_000, 6_794));
        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("Num Planets: " + planets.size());
        for (Planet planet : planets) {
            System.out.println(planet);
        }
    }
    ...
}
```

Try It:

Run this **ArrayList** example. Uncomment the code that references the **LinkedList** to use that implementation rather than the **ArrayList**.

LABS

- ① Write a program that contains an **ArrayList** of the names of the students in your class. Use a `foreach` loop to print out each name.
(Solution: *Students.java*)
- ② Write a program that uses **Math.random()** to build an **ArrayList** of 10 numbers between 0 and 1. Use a `standard for` loop to print out each number to 3 decimal places.
(Solution: *Numbers.java*)
- ③ Modify your previous solution to sort the numbers and print them. Don't try to develop a sort algorithm on your own, rather search the Internet for existing algorithms like bubble sort or insertion sort. If you have time, try multiple sort algorithms and be prepared to discuss the details of each with a partner.
(Solution: *SortNumbers.java*)

CHAPTER 3 - INPUT/OUTPUT STREAMS

OBJECTIVES

- * Use streams to read and write data to files in binary and text formats.
- * Serialize an object to a stream.
- * Avoid resource leaks by closing streams.

OVERVIEW OF STREAMS

- * All Java I/O is conducted through *streams*, which are ordered sequences of data.
 - A stream is either an input stream, an output stream, or both at the same time.
- * Streams aren't just for file I/O; their uses include:
 - Communication across sockets.
 - I/O to or from a **URL** object, including a servlet.
 - Storing and retrieving data based on a **String**, **char[]**, or **byte[]**.
 - Reading and writing Large Objects in a database with JDBC.
 - Serialization of objects.
- * Almost all of the methods in Java's I/O classes throw **java.io.IOException** (or one of its children).

ReadIt.java

```
package examples;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadIt {
    public static void main(String[] args) {
        BufferedReader bufIn = null;
        try {
            bufIn = new BufferedReader(new FileReader("input.txt"));

            String line;
            while ((line = bufIn.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println(e);
        }
        ...
    }
}
```

The **BufferedReader** wraps the **FileReader**.

Write each line from the file to standard output.

BufferedReader

The **BufferedReader** class allows us to read more efficiently from a device like the hard drive. Along with buffering input, it also provides the ability to read one line at a time. Without a **BufferedReader**, we would have to read a **byte** at a time, or read a set of **bytes** and convert them to a **String** manually.

BYTES VS. CHARACTERS

- * We can characterize any Java I/O stream as either a byte stream or a character stream.
- * *Byte streams* always deal with data as streams of 8-bit bytes.
 - Byte streams are referred to in Java as *streams*, with class names like **InputStream** and **OutputStream**.
 - Any time binary, non-textual data is being manipulated, you will use a stream.
 - An **int** written with an output stream is written as 4 bytes, not as readable text.
- * *Character streams* use 16-bit Unicode characters.
 - Character streams are referred to in Java as *readers* and *writers*, with class names like **FileReader** and **FileWriter**.
 - Any time textual data is being manipulated, you will use a reader or writer.
 - An **int** written with a writer is written as a sequence of characters; this is a readable format.
- * For every input stream there is usually a corresponding reader class; for every output stream there is usually a corresponding writer class.

Animal.java

```
...
public class Animal {
    private String species;
    private String name;
    private boolean friendly;
    private int weight;
    ...
    public void writeBinary(OutputStream out) throws IOException {
        DataOutputStream dataOut = new DataOutputStream(out);
        dataOut.writeChars(species);
        dataOut.writeChars(name);
        dataOut.writeBoolean(friendly);
        dataOut.writeInt(weight);
    }

    public void writeText(Writer out) {
        PrintWriter printOut = new PrintWriter(out);
        printOut.println(species);
        printOut.println(name);
        printOut.println(friendly);
        printOut.println(weight);
    }

    public static void main(String[] args) {
        Animal scrappy = new Animal("canine", "Scrappy", true, 112);
        Animal hobbles = new Animal("tiger", "Hobbles", true, 425);
        Animal scar = new Animal("lion", "Scar", false, 600);
        try {
            FileOutputStream dat = new FileOutputStream("Animal.dat");
            scrappy.writeBinary(dat);
            hobbles.writeBinary(dat);
            scar.writeBinary(dat);
            dat.close();

            FileWriter txt = new FileWriter("Animal.txt");
            scrappy.writeText(txt);
            hobbles.writeText(txt);
            scar.writeText(txt);
            txt.close();
            System.out.println("New files written to file system");
        }
        ...
    }
}
```

Try It:

After running *Animal.java*, look at the data stored in *Animal.dat* and *Animal.txt*.

FILE OBJECT

- ✧ You can use a **File** object to represent the path and filename of a given file.

```
File f = new File("./input.txt");
```

- The file's length, full directory location, and associated information can be retrieved from the **File** object.

- **getName()**
- **getCanonicalPath()**
- **length()**

- The platform-specific path separator can also be used for more generic naming of paths.

```
File f = new File(".") + File.separator + "input.txt");
```

- A **File** does not need to represent a file stream that is currently open.

FileInfo.java

```
package examples;

import java.io.File;
import java.io.IOException;

public class FileInfo {
    public static void main(String[] args) {
        File f = new File("input.txt");
        System.out.printf("The length of %s is %d bytes.%n",
            f.getName(), f.length();

        try {
            System.out.printf("%s has a full path of %s.%n",
                f.getName(), f.getCanonicalPath();
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Try It:

Run *FileInfo.java* to see extra information about *input.txt*.

BINARY INPUT AND OUTPUT

- * Classes such as **FileInputStream** and **FileOutputStream** are used for binary I/O, but only provide **read()** and **write()** methods that expect **bytes** or **byte** arrays.

- This is usually too cumbersome to deal with.

- * You can create a **DataInputStream** or **DataOutputStream** object to convert Java primitive datatypes into sequences of **bytes**.

- * Create a **DataInputStream** by passing an **InputStream** to its constructor.

```
FileInputStream finput = new FileInputStream( filename );
DataInputStream dinput = new DataInputStream( finput );
```


- * **DataInputStream** and **DataOutputStream** are typically used in pairs to read and write a binary file, or send and receive binary data through a socket.

WriteBinary.java

```
...
public class WriteBinary {
    public static void main(String[] args) {
        try {
            FileOutputStream fout = new FileOutputStream("test.dat");
            DataOutputStream dout = new DataOutputStream(fout);

            dout.writeInt(12);
            dout.writeDouble(12.5);
            dout.writeBoolean(false);
            dout.writeChar('a');
            dout.writeUTF("A String");

            dout.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```



ReadBinary.java

```
...
public class ReadBinary {
    public static void main(String[] args) {
        try {
            FileInputStream fin = new FileInputStream("test.dat");
            DataInputStream din = new DataInputStream(fin);

            System.out.println(din.readInt());
            System.out.println(din.readDouble());
            System.out.println(din.readBoolean());
            System.out.println(din.readChar());
            System.out.println(din.readUTF());

            din.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Try It:

Run *WriteBinary.java* to create *test.dat*, then run *ReadBinary.java* to read it back in.

PRINTWRITER CLASS

- * Use a **PrintWriter** object when you need to output data as text.
 - This data may be primitive types, or objects from your own classes.
- * You can construct a **PrintWriter** from either an **OutputStream** or another **Writer**.

```
FileWriter fw = new FileWriter("test.txt");  
PrintWriter pw = new PrintWriter(fw);
```

- * **PrintWriter** provides several forms of **print()** and **println()** that take each of the primitive types and convert them to **Strings** before printing them.

```
pout.print( appDimensions.width );
```

- **print()** and **println()** also have versions that take **Object**, and call **toString()** on that object.

```
System.out.println(user1); //calls user1.toString()
```

- Use **PrintWriter**'s **printf()** method for formatted output.
- * The **PrintStream** class has all of the same methods, but works with binary data instead of character data.
 - **System.out** and **System.err** are **PrintStream** objects.

WriteText.java

```
...
public class WriteText {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("test.txt");
            PrintWriter pw = new PrintWriter(fw);

            pw.println(12);
            pw.printf("%1$.2f %n", 12.5);
            pw.println(false);
            pw.println('a');
            pw.println("A String");

            pw.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

The **PrintWriter** wraps
the **FileWriter**.

ReadText.java

```
...
public class ReadText {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("test.txt");
            BufferedReader buf = new BufferedReader(fr);

            String line;
            while ((line = buf.readLine()) != null) {
                System.out.println(line);
            }

            buf.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Try It:

Run *WriteText.java* to create *test.txt*, then run *ReadText.java* to read it back in.

READING AND WRITING OBJECTS

✴ You can read and write an object to a stream using the **ObjectInputStream** and **ObjectOutputStream** classes.

- These classes contain **readObject()** and **writeObject()** methods, respectively.
 - The object is written in a binary format and is called a *serialized* object.
 - The **readObject()** method throws **ClassNotFoundException** if it cannot find the class definition (.class file) for the serialized object.
- Any object that you send to the **readObject()** or **writeObject()** method needs to implement the **Serializable** interface.

```
public class Book implements Serializable {
```

- There are no methods defined in this interface — you simply say **implements Serializable** and you're done!
 - These classes contain additional methods that are used for customizing the serialization process.
- ✴ Objects can be saved in files, written to a database, or sent across a socket or pipe — all using the **ObjectInputStream** and **ObjectOutputStream** classes.

Book.java

```
...
import java.io.Serializable;
public class Book implements Serializable {
    ...
}
```

WriteBook.java

```
...
public class WriteBook {
    public static void main(String[] args) {
        Book b = new Book("Animal Farm", "George Orwell", 1945, 144);
        try {
            FileOutputStream fout = new FileOutputStream("Book.ser");
            ObjectOutputStream out = new ObjectOutputStream(fout);
            out.writeObject(b);
            out.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

The **ObjectOutputStream**
wraps the
FileOutputStream.

ReadBook.java

```
...
public class ReadBook {
    public static void main(String[] args) {
        Object o = null;
        try {
            FileInputStream fin = new FileInputStream("Book.ser");
            ObjectInputStream in = new ObjectInputStream(fin);
            try {
                o = in.readObject();
            }
            catch (ClassNotFoundException e) {
                System.err.println(e.getMessage());
            }
            in.close();
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
        }
        System.out.println(o);
    }
}
```

What type of thing is o?

CLOSING STREAMS

- * Upon opening a stream, your program may acquire resources such as file handles or sockets.

- You must remember to close your stream to release these resources.

```
FileReader fr= new FileReader("input.txt");
BufferedReader bReader = new BufferedReader(fr);
...
bReader.close();
```

- * Use a **finally** block to ensure that a stream is closed whether or not an exception has occurred.

- Since the **close()** method itself may throw an exception, your code may end up unwieldy with a **try/catch** block embedded within a **finally** block.

- * Any object that implements the **java.lang.AutoCloseable** interface can take advantage of the *try-with-resources* syntax introduced in Java 7.

- A stream declared within parenthesis after the **try** keyword will automatically be closed for you whether or not an exception occurs.

```
try (BufferedReader bufIn = new BufferedReader(
    new FileReader("input.txt"))) {
    ...
}
```

- Any **catch** or **finally** blocks associated with a try-with-resources statement will run after the resources are closed.

ReadIt.java

```

...
public class ReadIt {
    public static void main(String[] args) {
        BufferedReader bufIn = null;
        try {
            bufIn = new BufferedReader(new FileReader("input.txt"));

            String line;
            while ((line = bufIn.readLine()) != null) {
                System.out.println(line);
            }
        }
        catch (IOException e) {
            System.err.println(e);
        }
        finally {
            if (bufIn != null) {
                try {
                    bufIn.close();
                }
                catch (IOException e) {
                    System.err.println(e);
                }
            }
        }
    }
}

```

If you wrap one stream object within another, you can simply close the outermost wrapper.

ReadIt2.java

```

...
public class ReadIt2 {
    public static void main(String[] args) {
        try (BufferedReader bufIn =
            new BufferedReader(new FileReader("input.txt"))) {

            String line;
            while ((line = bufIn.readLine()) != null) {
                System.out.println(line);
            }
        }
        catch (IOException e) {
            System.err.println(e);
        }
    }
}

```

In Java 7, you can declare the **BufferedReader** within the parenthesis after the **try** so it will close automatically.

LABS

❶ Create a class named **Order** which contains a customer id and name. Provide an appropriate constructor.

- a. Write the contents of an **Order** object to the file *order.txt* in a readable text format.
- b. Write the contents of the same object to the file *order.dat* in a binary format.
- c. Write the same object to the file *order.ser* as a serialized object.

Make sure your **Order** class implements **Serializable**.

(Solution: *Order.java*)

❷ Create a class named **Search** that searches through the file *employee.txt* for the pattern "manager" and prints out any lines that match the pattern. (Hint: You can search through a string for a substring using the **indexOf()** method in the **String** class.)

(Solution: *Search.java*)

❸ Modify the search program to obtain the pattern followed by the filename from the command line, such as:

```
java Search manager employee.txt
```

Try searching through your source code file, *Search.java*, for a pattern such as "**= new**" or "**Reader.**"

(Solution: *Search2.java*)

CHAPTER 4 - REGULAR EXPRESSIONS

OBJECTIVES

- * Describe the concept of pattern matching.
- * Construct regular expressions to search strings.
- * Use regular expressions in Java programs.

PATTERN MATCHING AND REGULAR EXPRESSIONS

- * Pattern matching is a method of determining if a specified text pattern occurs in a string.
- * A *Regular Expression* (RE) is a specification of a text pattern.
- * An RE is used to search for occurrences of the pattern in strings.
- * Example (the following are all characters, not numbers):

- Regular expression: **"ERROR"**
- A string: **"Operation 235: SUCCESS"**
- Another string: **"Operation 236: ERROR"**

Question: Does the pattern **ERROR** occur in the first string?

Question: Does the pattern **ERROR** occur in the second string?

- * A common use of REs is to identify all lines in a text file that contain a certain pattern, then have your program perform an operation on those lines.
 - Extract all lines in a file that have the area code **303**.
 - Display all lines in a file that contain the pattern **ERROR**.
 - Check to see if a file has any blank lines in it.
 - In a file, change all area codes from the format **(nnn)** to **nnn-**.
 - In a memo, change all occurrences of **Mrs** or **Miss** to **Ms**.
- * The strings, or lines of text, that are searched by REs may come from files, HTML forms, sockets, database queries, the command line, etc.
- * REs themselves are strings of text.

This is an introduction to the use of regular expressions in Java. You will study several RE specifications, but not all of them, and you will learn how to use the most important methods in the RE classes.

REGULAR EXPRESSIONS IN JAVA

- ✧ In Java, the tools for using REs are found primarily in two classes, both in **java.util.regex**.
 - The **Pattern** class.
 - The **Matcher** class.
- ✧ The **Pattern** and **Matcher** classes work together.
 1. The **Pattern** class takes an RE string and compiles it into an internal representation contained in a **Pattern** object.
 2. Next, a **Matcher** object is created by the **Pattern** object.
 3. Finally, strings are sent to the **Matcher** object to see if they match the pattern.
- ✧ Here is a snippet that shows a basic example of using these classes:

```
// Determine if the pattern ERROR occurs in a line
Pattern p = Pattern.compile("ERROR");
Matcher m = p.matcher("Abort: ERROR 339 has occurred");
boolean result = m.find();
```

- The **m.find()** method searches the string held in the **Matcher** object **m** to see if the pattern "**ERROR**" occurs in the string.
- If the pattern is found in the string, it is called a *match*.
- **find()** returns **true** if it finds a match, otherwise **false**.

RE1.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RE1 {
    public static void main(String[] args) {
        String re = "303";
        String string1 = "(303) 555-1212";
        String string2 = "(720) 555-1212";

        Pattern p = Pattern.compile(re);
        Matcher m = p.matcher(string1);

        boolean result = m.find();
        System.out.println("find() is " + result + ".");

        // change the search String
        m.reset(string2);
        result = m.find();
        System.out.println("find() is " + result + ".");
    }
}
```

reset() allows us to reuse the **Matcher** on a new **String**.

Try It:

Run *RE1.java* to find the pattern.

REGULAR EXPRESSION SYNTAX

- * REs provide a comprehensive syntax for specifying patterns of text.
- * The most simple REs are literals, which are searched for by direct string comparison.

- The RE **"A"** will match each of the following strings:

```
"An apple a day ..."  
"We skied Aspen for several days."
```

- The RE **"bro"** will match each of these strings:

```
"Bottles were broken all over the place."  
"I called her sis, and him bro."  
"I called my sister sis, and my brother bro."
```

- * REs can also contain special characters, or *metacharacters*, that have special meaning.
- * The **^** metacharacter means a pattern must start at the beginning of the string.

- The RE **"^A"** matches the first string but not the second:

```
"An apple a day ..."  
"We skied Aspen for several days."
```

- * The period metacharacter matches any single character.

- The RE **"12."** matches any three-character sequence starting with **"12"**:

```
"125"  
"As simple as 123"  
"The address is 7412 Main Street."    What matches!  
"Today I turned 12"                    Why does this not match?
```

RETest.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RETest {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(args[1]);
        Matcher m = p.matcher(args[0]);

        // Search args[0] for pattern in args[1]
        boolean result = m.find();
        System.out.println("find() is " + result + ".");
    }
}
```

Try It:

The program *RETest.java* lets you experiment with different RE patterns and different strings.

Run it with the string to be searched in **args[0]** and the RE in **args[1]**:

```
java RETest hello e
java RETest Aspen "^A"
java RETest "The A is not first" "^A"
```

Experiment away!

SPECIAL CHARACTERS

- * In an RE, a `.` matches any single character (except a newline).
- * You can narrow things down by using other special characters.
- * `\w` (*word* character) matches any character in the set **A-Z, a-z, 0-9**, or `_`.
 - `\W` matches any simple character not in that set.
- * `\s` (*whitespace* character) matches a newline, carriage return, formfeed, tab, or space character.
 - `\S` matches any non-space character.
- * `\d` (*digit* character) matches a digit, **0-9**.
 - `\D` matches any non-digit character.
- * The above are *character classes*; the generalized form of an RE character class is a list of characters in brackets, `[]`.
 - `[A-Za-z_0-9]` is the same as `\w`.
 - `[\n\r\f\t]` is the same as `\s`.
 - `[0-9]` is the same as `\d`.
 - If the first character after the opening `[` is a caret, `^`, then the character class matches any character not listed.
 - `[^A-Za-z_0-9]` is the same as `\W` — any character other than these.
 - `[^AEIOU]` matches any simple character that isn't a capital vowel.

Perl RE special characters used in Java

.	Matches any character except newline
[a-z0-9]	Matches any single character of set
[^a-z0-9]	Matches any single character not in set
\d	Matches a digit, same as [0-9]
\D	Matches a non-digit, same as [^0-9]
\w	Matches an alphanumeric (word) character [a-zA-Z0-9_]
\W	Matches a non-word character [^a-zA-Z0-9_]
\s	Matches a whitespace char (space, tab, newline . . .)
\S	Matches a non-whitespace character
\n	Matches a newline
\r	Matches a return
\t	Matches a tab
\f	Matches a formfeed
\b	Matches a backspace (inside [] only)
\.	Matches period
*	Matches asterisk
\	Matches pipe
\0	Matches a null character
\000	Also matches a null character
\0nn	Matches an ASCII character of that octal value
\xnn	Matches an ASCII character of that hexadecimal value
\cX	Matches an ASCII control character <ctrl>X
\metachar	Matches the character itself (\ , \., * ...)

POSIX character classes (US-ASCII only)

\p{Lower}	A lowercase alphabetic character: [a-z]
\p{Upper}	An uppercase alphabetic character: [A-Z]
\p{ASCII}	All ASCII: [\x00-\x7F]
\p{Alpha}	An alphabetic character: [\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character: [\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of: !"#\$%&'()*+,-./:;<=>?@[^\^_`{ }~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]

QUANTIFIERS

- * *Quantifiers* match a sequence of a single character, of members of a character class, or of a subexpression.
- * *** is the least specific, matching any number, even zero, of the preceding RE character.
 - `\d*` Matches any sequence of digits, or none at all.
 - `.*` Is the general-purpose, match-anything-at-all RE.
- * *+* matches one or more of the preceding RE character.
 - `\d+` Matches a sequence of digits.
- * *?* matches either zero or one of the preceding RE character.
- * You can specify any quantity of an RE character by using `{}`.
 - `{n}` Matches exactly *n* occurrences of the preceding RE character.
 - `\w\d{4}\s` Matches a word character, followed by exactly four digits, followed by a whitespace character.
 - `{n,}` Matches *n* or more occurrences of the preceding RE character.
 - `\w\d{4,}\s` Matches a word character, followed by at least four digits, followed by a whitespace character.
 - `{n,m}` Matches at least *n* but no more than *m* occurrences of the preceding RE character.
 - `\w\d{4,7}\s` Matches a word character, followed by four, five, six, or seven digits, followed by a whitespace character.

Here are some more examples using quantifiers. Because the backslash is the escape character in Java, remember to precede a backslash with another backslash in a literal string.

`"\\d{2,3}\\s\\w+"` Matches two or three digits, followed by a space character, followed by one or more word characters.

`"\\d{5,}"` Matches a string that contains at least five digits.

`"animals?\\s"` Matches "animal " or "animals " — note the trailing space.

RE2.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RE2 {
    public static void main(String[] args) {

        String[] words = { "IT", "1265478654", " 3453453 ", "wow",
                           "some spaces", "101 dalmations", "agent007", ".",
                           "another example" };

        String[] regExs = { "\\d{7,}", "\\s+\\w+\\s+", "... " };

        for (String re : regExs) {
            Pattern p = Pattern.compile(re);

            for (String word : words) {
                Matcher m = p.matcher(word);
                boolean result = m.find();
                System.out.println("Pattern \"" + re + "\" -> \""
                                   + word + "\": " + result);
            }
        }
    }
}
```

Try It:

Before running *RE2.java*, predict which patterns in the **regExs** array will be found in which strings in the **words** array.

ASSERTIONS

- * Assertions anchor a pattern to a specific location in a string.
- * `^` and `$` anchor a pattern to the beginning or ending, respectively, of a string.
 - `^s\d` Matches a line starting with `s` followed by a digit.
 - `sh$` Matches a line ending with `sh`.
- * `\b` anchors a pattern to a word boundary.
 - A *word boundary* is simply a position with a word character (`\w`) on one side but not on the other.
 - `\B` matches a position other than a word boundary — a position with word characters (`\w`) on either side or with non-word characters (`\W`) on either side.

Here are some examples that use assertions:

- `"^\\d+$"` Matches a string that contains only digits.
- `"^...$"` Matches a string with exactly three characters.
- `"\\.\\.$"` Matches a string that ends with a period.
- `"\\bis\\b"` Matches a string that contains the word **is**.

RE3.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RE3 {
    public static void main(String[] args) {
        String s = "This is a test.";
        Pattern p = Pattern.compile("\\bis\\b");
        Matcher m = p.matcher(s);

        boolean result = m.find();

        if (result) {
            System.out.println("Found \\bis\\b at index "
                + m.start() + " for string \"" + s + "\"");
        }
        else {
            System.out.println("No match.");
        }
    }
}
```

Try It:

Run *RE3.java* to find the word **is**.

THE PATTERN CLASS

- ✴ Use **Pattern.compile(String *regex*)** or **Pattern.compile(String *regex*, int *flags*)** to create a pattern object.

- Some of the *flags* are:

CASE_INSENSITIVE Allows case-insensitive matching.

COMMENTS Allows white space and comments in a pattern.

MULTILINE Searches for patterns in a multiline string.

- The following will match "CAT", "Cat", "cat", etc. . .

```
Pattern p = Pattern.compile("cat", Pattern.CASE_INSENSITIVE);
```

- ✴ The **flags()** method returns which flags are set for a pattern.

```
int fl = p.flags();
```

- ✴ The **pattern()** method returns the regular expression.

```
String re = p.pattern();
```

- ✴ You may split a string with the **split(CharSequence *c*)** method.

```
Pattern p = Pattern.compile("[|]");
String[] words = p.split("This|is|a|test.");
```

- ✴ For a single-use pattern:

```
boolean b=Pattern.matches("a*b", "aaaab");
```

- This works like the **matches()** method in the **Matcher** class.

RE4.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

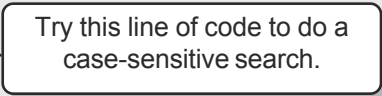
public class RE4 {
    public static void main(String args[]) {
        String s = "TEXT IN ALL CAPS";

        // Pattern p = Pattern.compile("all");
        Pattern p = Pattern.compile("all", Pattern.CASE_INSENSITIVE);
        Matcher m = p.matcher(s);

        boolean result = m.find();
        System.out.println("*** Case sensitive test***");
        System.out.println("\"" + p.pattern() + "\" matches \"" + s
            + "\" --> " + result);

        s = "Perhaps, this, is, a, flatfile, database";
        p = Pattern.compile(",");
        String[] words = p.split(s);

        System.out.println("*** Split test ***");
        for (String word : words) {
            System.out.println(word);
        }
    }
}
```

**Try It:**

Run *RE4.java* to try a case-insensitive search and to split a string.

THE MATCHER CLASS

- * The **find()** method searches for a substring that matches the pattern.
 - Successive calls to **find()** will start searching at the next character after the last character of the previous match.
 - The **reset()** method sets the starting search position to the first character in the string.
 - The **reset(String s)** replaces the search text.
- * The **matches()** method is like the **find()**, but infers a beginning ^ and ending \$ anchor on the pattern.
- * The **lookingAt()** method is like **find()**, except it infers a beginning anchor ^.
- * The **Matcher** class has methods that return data about a previous match.
 - **start()** returns the start index of the previous match.
 - **end()** returns the index of the last character matched, plus one.
 - **group()** returns the string which was previously matched by the pattern.
- * **replaceFirst(String s)** replaces the first matched pattern with *s*.
- * **replaceAll(String s)** replaces all occurrences of the pattern with *s*.

RE5.java

```
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RE5 {
    public static void main(String args[]) {
        String s = "My dog likes to eat. My dog's name is Fred.";

        Pattern p = Pattern.compile("[dD]og");
        Matcher m = p.matcher(s);

        String newString = m.replaceFirst("cat");
        System.out.println("*** replaceFirst() example ***\nString \"" +
            s + "\"\n is now: \n\"" + newString + "\"");

        newString = m.replaceAll("cat");
        System.out.println("\n** replaceAll() example ***\nString \"" +
            s + "\"\n is now: \n\"" + newString + "\"");
    }
}
```

Try It:

Run *RE5.java* to use the **replaceFirst()** and **replaceAll()** methods.

CAPTURING GROUPS

- ✧ Your RE can remember what it grabbed and use it for future reference.
- ✧ Use parentheses () in the pattern to denote a captured group.

```
Pattern p = Pattern.compile("(\\w+) (\\s+) (\\w+)");
```

- The matched string from each group is stored in the **Matcher** object.
- Subsequent replacement and append methods can refer to the first matched string as **\$1**, the second as **\$2**, etc.
- Retrieve the string for the corresponding group with the **group(int i)** method.

- ✧ You may also nest groups.

```
Pattern p = Pattern.compile("(^(\\w+) (\\s+) (\\w+))$");
```

- The outermost group will be given the lowest group number.
- ✧ **groupCount()** returns the number of capturing groups in the RE.

```
RE6.java
package examples;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RE6 {
    public static void main(String args[]) {
        String s = "These      are      spaces";

        // grab 1 or more word chars, 1 or more space chars...
        Pattern p = Pattern.compile("(\\w+) (\\s+) (\\w+)");
        Matcher m = p.matcher(s);

        String newString = m.replaceFirst("$3$2$1");

        System.out.println("Group 1: " + m.group(1) + "\nGroup 2: "
            + m.group(2) + "\nGroup 3: " + m.group(3));

        System.out.println("String \"" + s + "\"\nis now: \n\""
            + newString + "\"\n");

        p = Pattern.compile("^((\\w+) (\\s+) (\\w+))$");
        m = p.matcher("Agent 007");

        m.find();
        for (int i = 1; i <= m.groupCount(); i++) {
            System.out.println("Group " + i + ": " + m.group(i));
        }
    }
}
```

Try It:

Run *RE6.java* to work with groups.

Note:

To match an open parenthesis, use "(".

To match a close parenthesis, use ")"

LABS

- ① Create a program that opens a Java file, reads the lines in one at a time, and determines if the line contains only inline comments or is a blank line. Print all the lines that contain any real Java code. (Hint: Use **BufferedReader** and **FileReader**.)

```
BufferedReader in = new BufferedReader(new FileReader(args[0]));
```

(Solution: *ReadFile.java*)

- ② Generalize *ReadFile.java* so that it searches for any pattern (**arg[0]**) in any file (**arg[1]**). This is a slightly simplified version of the UNIX **grep** utility. (Solution: *JGrep.java*)

- ③ *pet.txt* is a flat file database consisting of pet data. The fields are delimited by a pipe (|), and the records are all on separate lines. Open the file, read the lines, and use the **split()** method to find the data. Then print out the information in a readable format. The first line of the file contains the field headings, so don't print that out! (Hint: You'll need to use a character class in your regular expression or escape the RE.) (Solution: *ReadPet.java*)

CHAPTER 5 - CORE COLLECTION CLASSES

OBJECTIVES

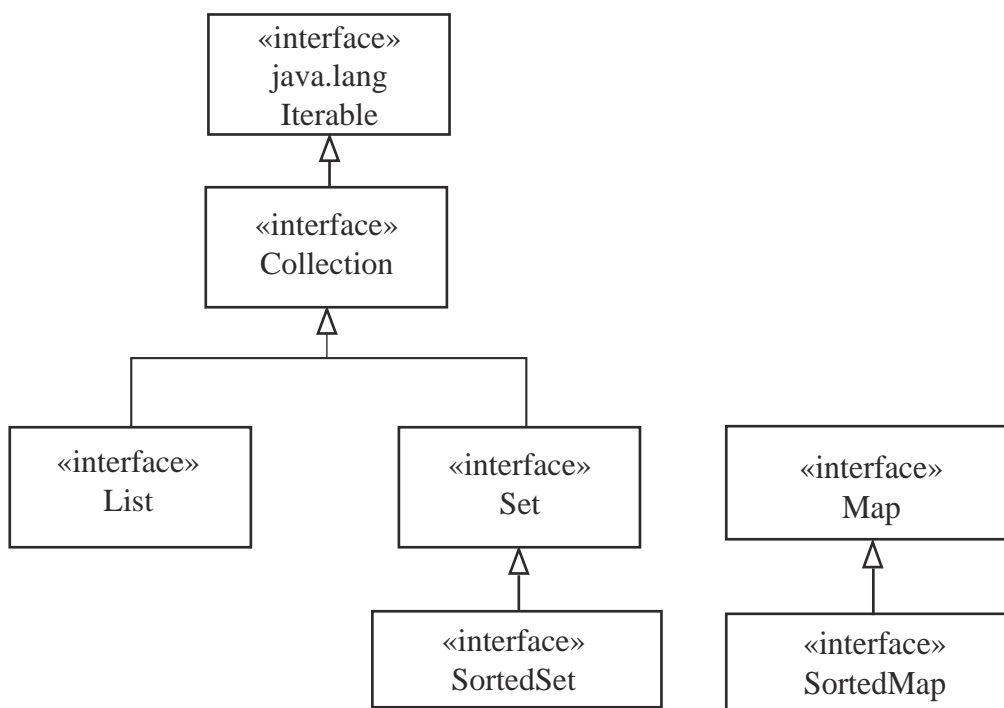
- * Use the implementation classes for the **Set**, **List**, and **Map** interfaces.
- * Choose the appropriate collection for a given application.
- * Iterate through the elements of a collection.

THE COLLECTIONS FRAMEWORK

- * A *collection* is an object that contains other objects.
 - The size of a collection can be altered after creation.
 - Some collection types order elements, while others do not.
- * Interfaces and classes related to collections make up the *Java Collections Framework*.
- * Interfaces determine common behavior or characteristics of various collection types.
 - **Collection** is the parent interface of **Set** and **List**.
 - The **Set** interface describes a group of unique elements.
 - The **List** interface describes an ordered group of elements.
 - The **Map** interface describes a group of key/value pairs, the keys being unique.
- * Abstract classes aid in the implementation of the interfaces.
- * Developers use implementation classes, including **HashSet**, **ArrayList**, and **HashMap**.
 - Legacy collections, such as **Vector**, are still supported.
- * Use the **Iterator** interface to access collection elements sequentially.

Prior to Java 2, Java provided a few basic container classes that all remain in the Java API. The **Hashtable**, **Properties**, **Stack**, and **Vector** classes have been retrofitted into the current Collections Framework. While **Enumeration** is still supported, developers are encouraged to use the **Iterator** interface for sequential access to collection elements. Consider the abstract class **Dictionary** obsolete in favor of the **Map** interface. No changes were made to **BitSet**.

All interfaces are members of the **java.util** package, unless otherwise noted.



Note:

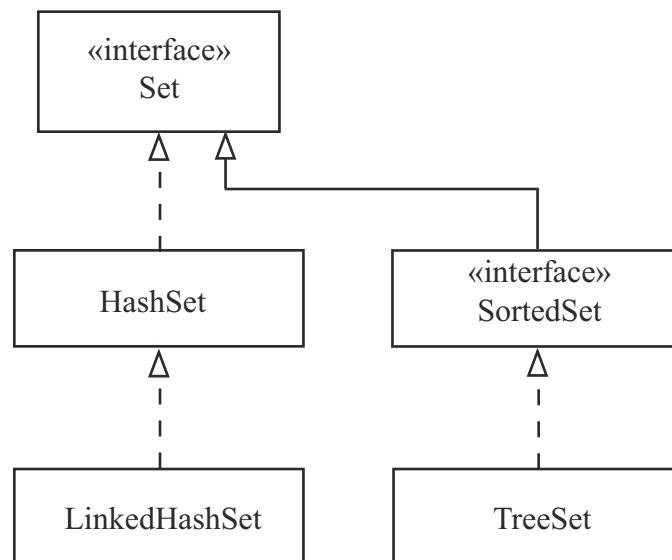
Beginning with Java 5.0, all **Collection** classes have been implemented as generics for ease of data access.

THE SET INTERFACE

- * **Sets** are not allowed to store duplicate elements.
 - No two elements in a **Set** can be **equal()** to each other.
 - At most, one **null** element may be placed in a **Set**.
 - Some implementations may disallow **null** elements.
- * Use the **add(*element*)** method to add an element to the **Set**.
 - The returned **boolean** may be used to determine whether a unique element was added.
 - **true** indicates the added element did not already exist in the **Set**.
 - **false** indicates that the **Set** was left unchanged because the element already existed.
- * Use the **iterator()** method to retrieve an **Iterator** object which you can use to step through a **Set**.
 - Use **hasNext()** to determine if there are more elements in a collection.
 - Use **next()** to then retrieve the next element.

```
Iterator<String> it = myset.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- * The **SortedSet** interface adds the guarantee that the **Set** will be ordered.



SET IMPLEMENTATION CLASSES

- ✧ **HashSet** is a general-purpose implementation of the **Set** interface.
 - A **HashSet** contains unique objects whose order is not guaranteed.
 - The **hashCode()** and **equals()** methods work in combination to ensure that only unique objects are in the set.
 - Your **hashCode()** method should return a unique hashcode, however you define "unique."
- ✧ **TreeSet** implements the **SortedSet** interface.
 - **TreeSet** extends the functionality of **HashSet** by sorting elements as they are added to and removed from the set.
- ✧ **LinkedHashSet** is a **HashSet** that implements the **Set** interface.
 - **LinkedHashSet** guarantees that the order of iteration will be insertion-order.
 - **HashSet** makes no guarantees about order.
 - **TreeSet** sorts on each insertion and removal: an expensive operation.

CD.java

```
...
public class CD {
    private int id;
    private String artist;
    private String title;

    public CD(int i, String a, String t) {
        id = i;
        artist = a;
        title = t;
    }
    ...
    @Override
    public int hashCode() {
        ...
    }

    @Override
    public boolean equals(Object obj) {
        ...
    }
}
```

SetTest.java

```
...
public class SetTest {
    public static void main(String args[]) {
        CD cd1 = new CD(1, "The Beatles", "The Beatles 1");
        CD cd2 = new CD(2, "Prince", "The Very Best of Prince");
        CD cd3 = new CD(3, "Garth Brooks", "The Ultimate Hits");
        CD cd4 = new CD(3, "Garth Brooks", "The Ultimate Hits");

        Set<CD> cdCollection = new HashSet<>();
        // Set<CD> cdCollection = new LinkedHashSet<>();

        cdCollection.add(cd1);
        cdCollection.add(cd2);
        cdCollection.add(cd3);
        cdCollection.add(cd4);

        Iterator<CD> it = cdCollection.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

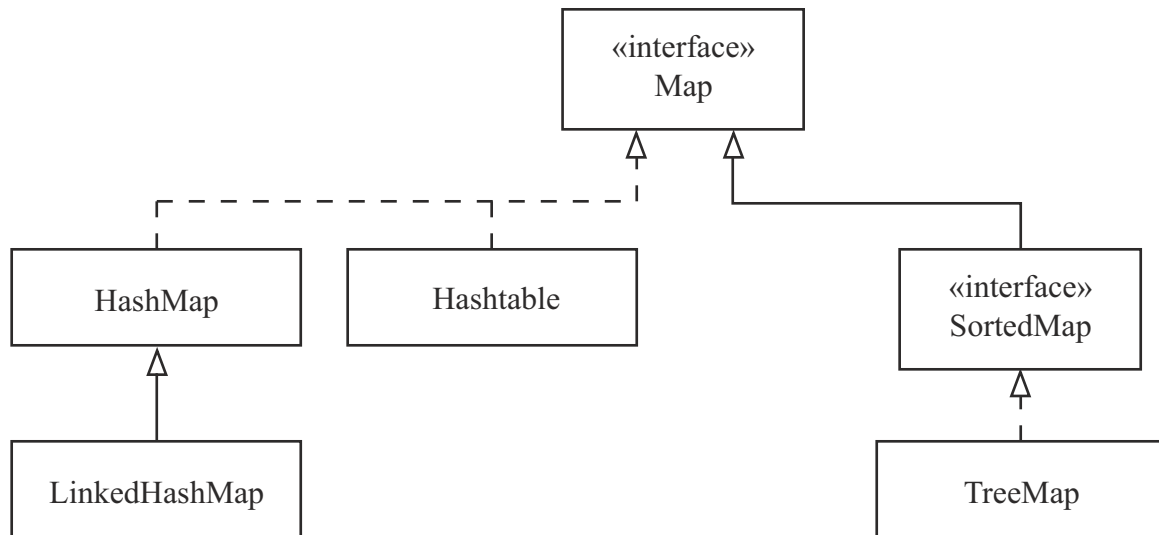
Uncomment to see the list in the order that the CDs were added.

THE MAP INTERFACE

- * **Maps** allow you to store key/value pairs.
 - The key must be unique.
 - A key can map to only a single value.
- * The **Map** interface makes no guarantees about order.
- * Use **Map**'s methods to work with the **Map**.
 - The **put(key, value)** method adds a key/value pair to the **Map**.
 - The **get(key)** method returns the value associated with the given key.
 - The **containsKey(key)** and **containsValue(value)** methods return **true** if the given parameter is in the **Map**.
- * **Map** does not define an **iterator()** method.
 - Use the **keySet()** method to retrieve a **Set** of keys that you can then iterate on.

```
Set<String> s = myMap.keySet();
Iterator<String> it = s.iterator();

while (it.hasNext()) {
    String key = it.next();
    System.out.print("Key: " + key);
    System.out.println(" Value: " + myMap.get(key));
}
```



MAP IMPLEMENTATION CLASSES

- * **HashMap** stores keys and values using an underlying hash algorithm.
 - Because it is based on a hash, **puts** and **gets** operate at constant-time performance.
 - The iteration order of a **HashMap** is not guaranteed.
 - Both keys and values are allowed to be **null**.
 - The legacy class **Hashtable** differs from **HashMap** in two ways:
 - **Hashtable** does not allow **null** keys.
 - **Hashtable**'s methods are **synchronized**, **HashMap**'s are not.
- * **LinkedHashMap** adds ordered iteration to **HashMap**.
 - The **keySet()** method will return a **Set** of keys that are in insertion order.
- * **TreeMap** implements the **SortedMap** interface.
 - **TreeMap** objects are sorted based upon the key at the time of insertion and removal.
 - **keySet()** returns a **Set** of keys in ascending order.

MapTest.java

```
package examples;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
//import java.util.LinkedHashMap;

public class MapTest {
    public static void main(String args[]) {
        CD cd1 = new CD(1, "The Beatles", "The Beatles 1");
        CD cd2 = new CD(2, "Prince", "The Very Best of Prince");
        CD cd3 = new CD(3, "Garth Brooks", "The Ultimate Hits");

        Map<String, CD> cdCollection = new HashMap<>();
        // Map<String,CD> cdCollection = new LinkedHashMap<>();

        cdCollection.put("B00004ZAV3", cd1);
        cdCollection.put("B00005M989", cd2);
        cdCollection.put("B000UVT3OI", cd3);

        Set<String> s = cdCollection.keySet();
        Iterator<String> it = s.iterator();

        while (it.hasNext()) {
            String key = it.next();
            System.out.print("Key: " + key);
            System.out.println(" Value: " + cdCollection.get(key));
        }
    }
}
```

Try It:

Run *MapTest.java*. What order are the **CD**'s printed in? Why? Rerun *MapTest.java* using the **LinkedHashMap** collection.

LABS

- ① Run the application *Guess.java* and guess how many elements are in the **HashSet**. Now, modify *Guess.java* to insert **StringBuilder** objects (initialized with identical values) instead of **String** objects into the **HashSet**. Recompile and run the application. How does the output differ? Why? (Solution: *Guess2.java*, *guess.txt*)
- ② The program *AreaCodeLister.java* reads in *abbreviations.txt* and *areacodes.txt*. Modify that program so that it loads the data from these files into two **HashMaps**. Then, print out each area code followed by the full name of the state or province associated with it. (Hint: Use **String.split()** to parse the key/value pairs separated by tab, '\t', characters.) (Solution: *AreaCodeLister2.java*)
- ③ Change your solution to the previous lab so that your output is sorted by area code. (Solution: *AreaCodeLister3.java*)

CHAPTER 6 - 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 StringComparator
    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:

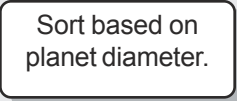
Run *SortStrings.java* to see the names sorted in alphabetical order. Notice how lowercase letters sort after uppercase letters. Modify the code to use the **StringComparator** and run it again to see case-insensitive sorting.

SORTING LISTS AND ARRAYS

- * Use **TreeSet** or **TreeMap** to sort **Sets** or **Maps**.
- * 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.

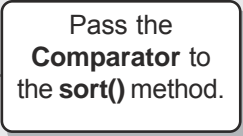
SortPlanets2.java

```
...
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:

$$\text{newCapacity} = (\text{oldCapacity} * 3) / 2 + 1.$$
 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

- ① *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*)
- ② 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*)
- ③ 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*)
- ④ (Optional) Modify your *ShowHits* program to sort the displayed records according to the total number of hits.
(Solutions: *ShowSortedHits.java*, *AccessLogComparator.java*)

CHAPTER 7 - INNER CLASSES

OBJECTIVES

- * Use member, local, and anonymous inner classes.
- * Create small classes when and where you need them.

INNER CLASSES

- * *Inner classes* were established in Java 1.1 as classes defined within other classes.
 - This syntax allows for quick and easy creation of classes for specific purposes.
- * The class that contains the inner class is called the *enclosing class* or the *enclosing instance* (depending on the type of inner class).
- * The enclosing class provides a namespace for its inner classes.
 - Two enclosing classes can each have an inner class with exactly the same name.
- * Inner classes have some restrictions:
 - They cannot contain any **static** declarations.
 - They cannot have the same name as any containing class (unlike fields and constructors).
- * There are three types of inner classes:
 - Member classes.
 - Local classes.
 - Anonymous classes.

The Java compiler compiles inner classes as normal classes, changing their names to ensure uniqueness and indicate scope, and usually adding some code to them to make them work as described.

The only thing that you, as a programmer, might notice are some strange *.class* files. Suppose that you have a *.java* file with a class called **Outer** that has an inner class called **Inner**. The compiler will create an *Outer.class* file and an *Outer\$Inner.class* file. Don't use this name in your code at all. The compiler takes care of everything!

PrintSortedPlanets.java

```
...
public class PrintSortedPlanets {

    public void printPlanets() {
        Set<Planet> planets = new TreeSet<>(new PlanetComparator());

        planets.add(new Planet("Mercury", 57_910_000, 4_880));
        planets.add(new Planet("Venus", 108_200_000, 12_103));
        planets.add(new Planet("Earth", 149_600_000, 12_756));
        ...
        for (Planet planet : planets) {
            System.out.println(planet);
        }
    }

    private class PlanetComparator 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());
        }
    }

    public static void main(String[] args) {
        PrintSortedPlanets p = new PrintSortedPlanets();
        p.printPlanets();
    }
}
```

Enclosing class.

Inner class.

MEMBER CLASSES

- ✧ A class defined within an enclosing class, but without the **static** keyword, is a *member class*.
 - It is defined at the same level as other fields and methods that are members of the class.

- ✧ Every instance of the member class has an internal reference to the enclosing object.

- A method in the member class can use its own data and the data in the enclosing instance (including **private** fields), implicitly, without any special syntax.

- ✧ Member class methods can explicitly refer to the enclosing instance's fields using the enclosing class name along with **this**.

```
int i = Outer.this.outerField;
```

- This syntax is only necessary for explicit access to shadowed fields or methods.

- ✧ You must specify an enclosing instance when creating a member instance from another class.

```
Outer out = new Outer();  
Outer.Inner in = out.new Inner();
```

- ✧ A member class may be declared as **private**, for use within its enclosing class, or as **protected**, **public**, or default, depending on how it will be used.

Outer.java

```
...
// Member Inner Class
public class Outer {
    private int outerField;

    public void aMethod() {
        Inner in = new Inner();
        System.out.println(in);
    }

    class Inner {
        private int innerField;
        public String toString() {
            // explicit access to outerField
            //return "o: " + Outer.this.outerField + " i: " + innerField;

            // implicit access to outerField
            return "o: " + outerField + " i: " + innerField;
        }
    }

    public static void main(String[] args) {
        Outer out = new Outer();
        out.aMethod();
    }
}
```

Inner can be used from any class within the same package.

Fields are initialized to zero by default.

TestInner.java

```
...
// Use a Member Inner Class
public class TestInner {
    public static void main(String[] args) {
        Outer out = new Outer();
        Outer.Inner in = out.new Inner();
        System.out.println(in);
    }
}
```


LOCAL CLASSES

- * A class defined within a method or constructor is a *local class*.
- * A local class has some interesting features:
 - It is visible only within the code block in which it is defined, just like a local variable.
 - It can only use **final** method parameters or **final** local variables that are within the same scope.
- * A local class has access to fields and methods of the enclosing class, just like a member class does.
 - When declared in a static block, local inner classes can access only the static fields and methods of the enclosing class.
- * Local classes cannot be declared **public**, **protected**, or **private**, since they are not members of a class.
- * Local classes are commonly used to implement event listeners and other interfaces.
 - The use of the class can be written right after the definition of the class, making the code more readable.

Outer2.java

```
...
// Local Inner Class
public class Outer2 {
    private int outerField;

    public void aMethod() {
        final int localVar = 0;

        class Inner {
            private int innerField;
            public String toString() {
                return "o: " + outerField + " i: " + innerField
                    + " l: " + localVar;
            }
        }

        Inner in = new Inner();
        System.out.println(in);
    }
    ...
}
```

PrintSortedPlanets2.java

```
...
public class PrintSortedPlanets2 {

    public void printPlanets() {

        class PlanetComparator 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());
            }
        }

        Set<Planet> planets = new TreeSet<>(new PlanetComparator());
        ...
    }
    ...
}
```

ANONYMOUS CLASSES

- * Anonymous classes are a kind of local inner classes that combine the class definition with the object instantiation.
 - They are used in expressions such as method calls or assignments.
- * Additional syntax for the **new** operator was introduced to allow for anonymous classes.

```
new Name( [arguments] ) {
    //class definition
}
```

- If *Name* is a class name, then the anonymous class **extends** that class and can provide constructor *arguments*.
 - If *Name* is an interface name, then the anonymous class **implements** that interface.
- * Find a consistent, readable indentation style for your anonymous classes.
- * Methods of local (including anonymous) classes can only use variables:
 - Declared in the method body.
 - Declared as parameters to the method.
 - Declared as **final** in the local class' enclosing scope (the enclosing method or block).
 - Declared as fields of the enclosing class or the inner class.

Outer3.java

```
...
// Anonymous Inner Class
public class Outer3 {
    private int outerField;

    public void aMethod() {
        final int localVar = 0;

        Object in = new Object() {
            private int innerField;
            public String toString() {
                return "o: " + outerField + " i: " + innerField
                    + " l: " + localVar;
            }
        };

        System.out.println(in);
    }
    ...
}
```

Note the semicolon to end the assignment statement.

PrintSortedPlanets3.java

```
...
public class PrintSortedPlanets3 {

    public void printPlanets() {

        Set<Planet> planets = new TreeSet<>(new 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());
            }
        });

        ...
    }
    ...
}
```

This open parenthesis...

...is closed here.

INSTANCE INITIALIZERS

- * You would typically initialize data members of a class with a constructor.
 - An anonymous class cannot provide constructors, since it has no class name.
- * Instance initializers allow an anonymous object to be properly initialized.
 - Any class can contain an instance initializer, though they are typically only used for anonymous classes.
- * A stand-alone code block inside a class definition is an *instance initializer*.
 - Multiple instance initializers are allowed; they are run from top to bottom.
 - They run after the superclass constructor, and before the current class' constructor (if it isn't anonymous).
- * The initialization of a data member can be performed immediately after the declaration of the variable.
 - This is helpful if a simple assignment won't accomplish the task.

Outer4.java

```
package examples;

// Anonymous Inner Class with instance initializer
public class Outer4 {
    private int outerField;

    public void aMethod() {
        final int localVar = 0;

        Object in = new Object() {
            private int innerField;

            // instance initializer
            {
                innerField = 12;
            }

            public String toString() {
                return "o: " + outerField + " i: " + innerField
                    + " l: " + localVar;
            }
        };

        System.out.println(in);
    }

    public static void main(String[] args) {
        Outer4 out = new Outer4();
        out.aMethod();
    }
}
```

Complex initialization
could be performed here
for the anonymous class.

LABS

Use *ShowSortedHits.java*, *AccessLogComparator.java*, and *access_log* as starter files for these labs.

- ① Modify *ShowSortedHits.java* so it uses a member inner class as a **Comparator**, rather than having *AccessLogComparator.java* as a separate file.
(Solution: *ShowSortedHits2.java*)
- ② Change your solution to the previous lab to use a local inner class instead of a member inner class.
(Solution: *ShowSortedHits3.java*)
- ③ Change your solution to the previous lab to use an anonymous inner class instead of a local inner class.
(Solution: *ShowSortedHits4.java*)

CHAPTER 8 - INTRODUCTION TO SWING

OBJECTIVES

- * Create a window using Java's graphical user interface packages.
- * Place components, such as **JButton** and **JTextArea**, in a GUI window.
- * Handle component events.

AWT AND SWING

- ✧ From the beginning, Java has supported development of graphical applications, both for pure two-dimensional drawing and for Graphical User Interfaces (GUIs).
- ✧ Java 1.0 introduced the Abstract Windowing Toolkit (AWT).
 - AWT components use the corresponding native widgets of the host windowing environment.
 - A given AWT component (such as a radio button) looks like a Windows radio button when run under Windows, and like a Motif radio button when run under Motif.
 - These are called *heavyweight* components because they actually use the widgets defined and compiled into the external, native toolkit API libraries (Win32 on Windows, *libXm.so* in Motif, etc.).
- ✧ Java 1.2 added the Swing user interface classes.
 - Most Swing components are *lightweight* (written entirely in Java).
 - Components look the same regardless of the native windowing environment.
 - Swing allows for *Pluggable Look and Feel* (PLAF), in which the overall appearance of an application can be customized and switched at runtime.
- ✧ Some Swing components, such as **JFrame** and **JWindow**, inherit their functionality from AWT base classes, since the native system must render those.

While you are learning to use Swing, keep your browser open to the *Java Platform Standard Edition API Specification* section of the Java documentation. Even after you've become proficient at GUI programming in Java, you will often need to browse the **java.awt** and **javax.swing** documentation to look up specific details for components.

PlafExample.java

```
...
public class PlafExample extends JFrame {
    ...
    private void addComponents() {
        JPanel panel = new JPanel();

        UIManager.LookAndFeelInfo[] laf = UIManager
            .getInstalledLookAndFeels();

        for (UIManager.LookAndFeelInfo info : laf) {
            String name = info.getName();
            final String className = info.getClassName();

            JButton button = new JButton(name);

            class LookAndFeelListener implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                    try {
                        UIManager.setLookAndFeel(className);
                        SwingUtilities.updateComponentTreeUI(
                            getContentPane());
                        pack();
                    }
                    catch (Exception ex) {
                        System.err.println(ex);
                    }
                }
            }

            button.addActionListener(new LookAndFeelListener());
            panel.add(button);
        }

        this.add(panel, BorderLayout.CENTER);
    }
    ...
}
```

Local Inner Class

Try It:

Run *PlafExample.java* to see the Pluggable Look and Feel in action.

DISPLAYING A WINDOW

- ✴ Your GUI applications will use classes defined in the **java.awt** and **javax.swing** packages, and some of their subpackages.

```
import java.awt.*;
import javax.swing.*;
```

- ✴ A typical GUI has one or more frames containing components.
 - A *frame* is a *top-level* window that is controlled by a window manager; it has a border, and decorations for minimizing, maximizing, resizing, etc.
- ✴ A **JFrame** provides a container for components (some of which might be containers themselves).
 - Containers can contain other containers.
 - Each container manages the layout of its contents independently of the others, and you define your overall layout by combinations of containers and components.
- ✴ To display a window, you must:
 1. Create a subclass of **JFrame** or an object of type **JFrame**.
 2. Give the **JFrame** a nonzero size.
 3. Make the **JFrame** object visible.

Notepad1.java

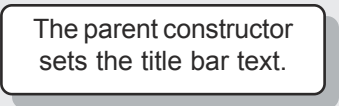
```
package examples;

import javax.swing.JFrame;

public class Notepad1 extends JFrame {

    public Notepad1() {
        super("Notepad");
        setSize(300, 400);
        setVisible(true);
    }

    public static void main(String[] arg) {
        new Notepad1();
    }
}
```



The parent constructor
sets the title bar text.

Try It:

When you run this example, you will find that your program does not stop when you close the window. Ask your instructor how to terminate the program until we fix this problem.

GUI PROGRAMMING IN JAVA

- * Components, such as buttons, labels, or text fields, make up the interface that the user can interact with.
 - Not all components are interactive; labels do not usually allow interaction, whereas buttons usually do.
- * To add a component to a window, you must first construct the component object, then add it to the window's container.
- * At practically any time, the user can use the mouse and keyboard to interact with a component; these interactions generate *events*.
 - To make your GUI interactive, you write methods which define what happens when a particular event occurs.
 - Define these methods in an *event listener* — an object you create and then add to a component that generates events.
- * So, setting up a GUI involves the same main steps each time:
 1. Create a top-level container, often by extending **JFrame**.
 2. Create component objects and lay them out in the container.
 3. Set up listeners for all components that you would like to handle events for, defining what happens when events occur.
 4. Size your window and then display it.

Notepad2.java

```
package examples;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Notepad2 extends JFrame {
    private JButton closeButton;

    public Notepad2() {
        super("Notepad");
        addComponents();
        setSize(300, 400);
        setVisible(true);
    }

    private void addComponents() {
        closeButton = new JButton("Close");
        this.add(closeButton);
    }

    public static void main(String[] arg) {
        new Notepad2();
    }
}
```

Try It:

Run this example. Can you see the button? How big is it? What happens when you click it? What happens when you resize the window? When you close the window?

HANDLING EVENTS

* An event listener, registered with a component (an *event source*), defines what happens when various kinds of events occur.

* To register an event listener, you must:

1. Create a *listener* class of the appropriate type.

- A **JButton** generates an **ActionEvent** when clicked, so you need an **ActionListener** to handle a button.

```
class MyButtonHandler implements ActionListener {
}
```

2. Implement the methods defined in the listener interface.

- **actionPerformed()** is the only method needed for an **ActionListener**.

```
public void actionPerformed(ActionEvent evt) {
    // Do something useful.
}
```

3. Register a listener object by adding it to the event source.

```
myButton.addActionListener(new MyButtonHandler());
```

* To tell Java that the application should exit when the window is closed, use the **setDefaultCloseOperation()** method.

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Let's create a listener for the event that is generated by clicking our **Close** button; the listener will close the window using JFrame's **dispose()** method.

Notepad3.java

```
package examples;
```

```
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
```

```
...
```

```
public class Notepad3 extends JFrame {  
    private JButton closeButton;
```

```
    public Notepad3() {  
        super("Notepad");  
        addComponents();  
        addEventHandlers();  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setSize(300, 400);  
        setVisible(true);  
    }  
    ...
```

```
    private void addEventHandlers() {  
        class CloseListener implements ActionListener {  
            public void actionPerformed(ActionEvent e) {  
                dispose();  
            }  
        }  
        ActionListener closeListener = new CloseListener();  
        closeButton.addActionListener(closeListener);
```

JFrame's **dispose()**
method closes the window.

```
        /*  
        // anonymous inner class  
        closeButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                dispose();  
            }  
        });  
        */
```

An anonymous class can be
used instead of a local class.

Try It:

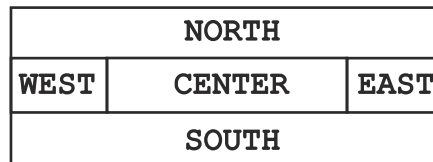
Test the button. The window should now close when you click it.

ARRANGING COMPONENTS

- * When you add multiple components to a container, you can choose how to manage their layout.
 - Explicitly specify the size and location of each component, making sure they don't overlap or hide one another.
 - Set a **LayoutManager** for the container, and let the **LayoutManager** arrange all the components.
- * Explicitly setting the size and location of components gives you total control over the appearance of your window.
 - **Component** methods like **setLocation()**, **setSize()**, and **setBounds()** explicitly shape and place components.
 - Some GUI-builder tools used to do this, when you drag and drop a component onto your window.
 - If the user resizes the window, though, the components won't automatically adjust and resize.
- * The Java libraries provide several **LayoutManager** classes for arranging components according to various layout schemes.
 - Commonly-used layout managers include **FlowLayout**, **BorderLayout**, **GridLayout**, **GroupLayout**, and **BoxLayout**.
 - Use the **Container** method **setLayout()** to specify which layout manager you want.

```
JPanel p = new JPanel();
p.setLayout(new GridLayout(2,4)); //2rowsX4cols
```

By default, a **JFrame**'s container uses a **BorderLayout** layout manager. A **BorderLayout** divides a container into five areas, each of which can hold only one component:



When you add a component to a container that uses a **BorderLayout**, the component will be placed in the **CENTER** area unless you specify otherwise. Unused areas are "squeezed out."

Let's put a new **JTextArea** component in the **CENTER**, and put the **Close** button in the **SOUTH**. Also, instead of presetting the size of the window, let's let the layout manager size it according to the size of the components it contains. The **pack()** method does this:

Notepad4.java

```
...
import java.awt.BorderLayout;
...
import javax.swing.JTextArea;

public class Notepad4 extends JFrame {
    private JButton closeButton;
    private JTextArea notesTextArea;

    public Notepad4() {
        ...
        // setSize(300,400);
        pack();
        setVisible(true);
    }

    private void addComponents() {
        notesTextArea = new JTextArea(24, 60);
        closeButton = new JButton("Close");

        this.add(notesTextArea, BorderLayout.CENTER);
        this.add(closeButton, BorderLayout.SOUTH);
    }
    ...
}
```

The **JTextArea** will have 24 rows and 60 columns.

Try It:

Run this application. Type several lines of gibberish, including some very long lines. How does the text area behave?

A SCROLLABLE COMPONENT

- ✴ When the content of a component is bigger than the area available to display it, you'll want to provide scrollbars.
- ✴ Swing has a **JScrollBar** component, but setting it up to scroll a component is complicated and tedious.
- ✴ Because scrollbars are almost always attached to another component, Swing provides a **JScrollPane** that wraps any component or container with a scrolling viewport.

```
JScrollPane sp = new JScrollPane(myComponent);
```

- **JScrollPane** is a container for a component (or for another container with multiple components).
- ✴ By default, the scrollpane will have vertical and horizontal scrollbars as necessary.
- You can set the scrollbar policy in the constructor:

```
JScrollPane sp = new JScrollPane(myComponent,
    ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

- Alternatively, you can set the scrollbar policy using **set** methods:

```
sp.setHorizontalScrollBarPolicy(
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
sp.setVerticalScrollBarPolicy(
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
```

Notepad5.java

```
...
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Notepad5 extends JFrame {
    private JButton closeButton;
    private JTextArea notesTextArea;
    ...
    private void addComponents() {
        notesTextArea = new JTextArea(24, 60);
        JScrollPane scrollPane = new JScrollPane(notesTextArea);
        closeButton = new JButton("Close");

        // contentPane.add(notesTextArea, BorderLayout.CENTER);
        this.add(scrollPane, BorderLayout.CENTER);
        this.add(closeButton, BorderLayout.SOUTH);
    }
    ...
}
```

Scroll bars will show
as needed by default.

Try It:

Run the application again to test the **JScrollPane**.

CONFIGURING COMPONENTS

- * Swing components are highly configurable.
 - Many standard properties apply to most Swing components.
 - Some properties apply only to certain classes of component.
- * You can specify some properties when you construct the component.

```
JButton b = new JButton("Exit");
```

- * **setProperty()** methods allow you to change component properties at any time.

```
b.setText("Quit");
```

- * There's usually a **getProperty()** method for determining a property value at runtime.

```
String lastname = lastNameTextField.getText();
```

Let's set the line wrap property of the **TextArea** so that lines of text wrap at word boundaries when they reach the margin.

Notepad6.java

```
...
public class Notepad6 extends JFrame {
    ...
    private void addComponents() {
        notesTextArea = new JTextArea(24, 60);
        notesTextArea.setLineWrap(true);
        notesTextArea.setWrapStyleWord(true);
        JScrollPane scrollPane = new JScrollPane(notesTextArea);
        closeButton = new JButton("Close");

        this.add(scrollPane, BorderLayout.CENTER);
        this.add(closeButton, BorderLayout.SOUTH);
    }
    ...
}
```

Try It:

Test the notepad by typing many long lines of text.

MENUS

- * In a **JFrame**, a menubar sits above the content pane.
- * Selecting a **JMenuItem** generates an **ActionEvent**, so you'll use **ActionListeners** to handle menu item interactions.
- * To give your **JFrame** a menubar:

1. Create a **JMenuBar**.

```
JMenuBar mb = new JMenuBar();
```

2. For each menu you want in the menubar, create a **JMenu**.

```
JMenu optionsMenu = new JMenu("Options");
```

3. Create a **JMenuItem** for each menu item.

```
JMenuItem colorMenuItem = new JMenuItem("Color...");
JMenuItem fontMenuItem = new JMenuItem("Font...");
```

4. Add an **ActionListener** to each **JMenuItem**.

```
colorMenuItem.addActionListener(myMenuListener);
```

5. Add the **JMenuItems** to their **JMenus**, and add the **JMenus** to the **JMenuBar**.

```
optionsMenu.add(colorMenuItem);
optionsMenu.add(fontMenuItem);
mb.add(optionsMenu);
```

6. Add the **JMenuBar** to the frame.

```
myframe.setJMenuBar(mb);
```

Notepad7.java

```
...
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Notepad7 extends JFrame {
    private JButton closeButton;
    private JMenuItem openMenuItem;
    private JMenuItem saveMenuItem;
    private JMenuItem exitMenuItem;
    ...
    private void addMenu() {
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu("File");
        openMenuItem = new JMenuItem("Open");
        saveMenuItem = new JMenuItem("Save");
        exitMenuItem = new JMenuItem("Exit");

        fileMenu.add(openMenuItem);
        fileMenu.add(saveMenuItem);
        fileMenu.add(exitMenuItem);
        fileMenu.insertSeparator(2); // put a line between save and exit
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);
    }

    private void addEventHandlers() {
        ...
        closeButton.addActionListener(closeListener);
        exitMenuItem.addActionListener(closeListener);

        openMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("ToDo: Add open code");
            }
        });

        saveMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("ToDo: Add save code");
            }
        });
    }
    ...
}
```

Anonymous Inner Classes

USING THE JFileChooser

- * Swing includes a pre-built, pure-Java file browser.
 - Any time you want the user to choose a filename, you can use the **JFileChooser**.
- * To get a filename from the user:
 1. Instantiate a **JFileChooser** object.


```
JFileChooser fc = new JFileChooser(".");
```
 2. Use **showOpenDialog()** or **showSaveDialog()** to display the chooser.


```
int retval = fc.showOpenDialog(parentFrame);
```

 - The integer return value of **showOpenDialog()** tells you whether the user chose a file or cancelled.
 3. Use **getSelectedFile()** to get the chosen file.


```
if (retval == JFileDialog.APPROVE_OPTION)
    File saveFile = fc.getSelectedFile();
```
- * **getSelectedFile()** returns a **File** object; the **File** class is defined in the **java.io** package, so you'll need to import that.

We'll use the **JFileChooser** to implement the **Save** option of our **File** menu.

Notepad8.java

```
...
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
...
import javax.swing.JFileChooser;
...
public class Notepad8 extends JFrame {
    ...
    private void addEventHandlers() {
        ...
        openMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("ToDo: Add open code");
            }
        });

        saveMenuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JFileChooser fc = new JFileChooser(".");
                int result = fc.showSaveDialog(Notepad8.this);
                if (result == JFileChooser.APPROVE_OPTION) {
                    File saveFile = fc.getSelectedFile();
                    try {
                        FileWriter fileOut = new FileWriter(saveFile);
                        fileOut.write(notesTextArea.getText());
                        fileOut.close();
                    }
                    catch (IOException ioe) {
                        System.err.println("I/O Error on Save.");
                    }
                }
            }
        });
    }
    ...
}
```

Open the
FileChooser in the
current directory.

Try It:

Run the application to test the save functionality.

LABS

- ① The **Close** button currently occupies the entire **SOUTH** area of your frame. Modify the notepad application: instead of adding the button directly to the frame's content pane, create a **JPanel**, add the button to the **JPanel**, and add the **JPanel** to the **SOUTH** area. Does the appearance of the button change? Where does it appear?
(Solution: *Notepad9.java*)

- ② In your **Notepad** class, declare a **JLabel** and a **JTextField** as instance fields. Create the **JLabel** with the text "**Filename:** ". Create the **JTextField** with the text **<Untitled>**. Add these to the **JPanel** you put in the **SOUTH** area of your frame so that the label is on the left, the text field is in the middle, and the button is on the right.

How big is each component? How are they aligned on the window? Change the text in the text field. Does the size of the field change? How about if you resize the window? What happens if you resize the window so it's too narrow for all three components in the **SOUTH** area?
(Solution: *Notepad10.java*)

- ③ Look up **JPanel** in the available documentation and find out what its default layout manager is. Find and read the description of that layout manager. Change the **JPanel** you created for the **SOUTH** area to a **Box**. A **Box** is just a container with a **BoxLayout** layout manager. A **BoxLayout** arranges its components in a single row, along either the vertical or horizontal axis. When you create a **BoxLayout** (or a **Box**), you must specify which axis it will use:

```
Box southBox = new Box(BoxLayout.X_AXIS);
```

Other than that, you use a **Box** just as you would use a **JPanel**.

How does the appearance and behavior of your frame's **SOUTH** area, and of the components it contains, change when you use a **Box** instead of a **JPanel**?
(Solution: *Notepad11.java*)

- ④ **JTextComponent** is the superclass of **JTextField**. Use the **JTextComponent** method **setEditable()** to make your **JTextField** read-only.
(Solution: *Notepad12.java*)

- ⑤ Modify the save menu listener so that when the text has been saved to a file, the **JTextField** is updated to display the filename.
(Solution: *Notepad13.java*)

- ⑥ Finish the **File** menu by implementing the **Open** item.

```
File openFile = fc.getSelectedFile();
try {
    FileReader fileIn = new FileReader(openFile);
    char[] readBuffer = new char[(int)openFile.length()];
    fileIn.read(readBuffer);
    notesTextArea.setText(new String(readBuffer));
    fileIn.close();
}
catch (IOException ioe) {...}
```

After you get this working, make it also update your **JTextField** with the name of the opened file.
(Solution: *Notepad14.java*)

- ⑦ (Optional) Add another **Menu** to the **MenuBar** with **MenuItems** that change the look and feel.
(Solution: *Notepad15.java*)

CHAPTER 9 - INTRODUCTION TO THREADS

OBJECTIVES

- * Explain what threads are and when to use them.
- * Create threads using the **Thread** class and the **Runnable** interface.
- * Use methods to place threads in different thread states.

NON-THREADED APPLICATIONS

- ✳ Normally, when you call a method, you wait until the method completes before continuing your code.
 - The sequential completion of tasks is a fundamental part of computer programming.
- ✳ An application runs in a single process.
 - Modern operating systems allow users to have multiple processes running at the same time.
 - Each process has its own code and data space.
 - The operating system allocates processor time to each process.

NonThread.java

```
package examples;

public class NonThread {
    public static void main(String[] args) {
        PrintNumbers p1 = new PrintNumbers(1, 2); // odds
        PrintNumbers p2 = new PrintNumbers(2, 2); // evens
        p1.print();
        p2.print();
    }
}

class PrintNumbers {
    private int start = 0;
    private int increment = 1;

    public PrintNumbers(int st, int inc) {
        start = st;
        increment = inc;
    }

    public void print() {
        int i, j;
        for (i = start, j = 0; j < 20; j++, i += increment) {
            System.out.println(i);
        }
    }
}
```

Try It:

Compile and run the above application. Watch the numbers as they are output.

THREADED APPLICATIONS

- * Most operating systems allow a single process to have multiple *threads* of execution.
 - A thread is also called a *lightweight process*, because it is quicker to use multiple threads than to use multiple processes.
- * In the same way that processes appear to run at the same time, threads can appear to run at the same time.
 - The processor time given to one process is divided up among its threads.
 - While one thread is waiting for some task to finish, such as printing or disk access, another thread can be using the CPU.
- * Every Java program has at least one thread running that is started by the Virtual Machine.
 - This thread is typically called the *main* or *application* thread.
 - If additional threads are not explicitly created, your program will behave just like a non-threaded application.
- * Additional threads are created and started by the program.
 - The system may change which thread it is running at any time.
 - You can help schedule and coordinate the threads in your application.
- * All of the threads in a process can share code and data.

ThreadApp.java

```
package examples;

public class ThreadApp {
    public static void main(String[] args) {
        PrintNumbers2 p1 = new PrintNumbers2(1, 2); // odds
        PrintNumbers2 p2 = new PrintNumbers2(2, 2); // evens
        p1.start();
        p2.start();
    }
}

class PrintNumbers2 extends Thread {
    private int start = 0;
    private int increment = 1;

    public PrintNumbers2(int st, int inc) {
        start = st;
        increment = inc;
    }

    public void print() {
        int i, j;
        for (i = start, j = 0; j < 20; j++, i += increment) {
            System.out.println(i);
        }
    }

    @Override
    public void run() {
        print();
    }
}
```

Try It:

Now, compile and run the multi-threaded version. Watch the numbers as they are output. The numbers might appear in different orders on different environments.

CREATING THREADS

- * Define your own threads by extending **java.lang.Thread** and overriding its **run()** method.
 - The **run()** method defines the life-cycle of the thread.
 - When the end of the **run()** method is reached, the thread is finished (or dead).
- * Send a **start()** message to a **Thread** object to activate it.
 - Once a thread has been started, the VM is able to call its **run()** method, whenever it has processor time.
 - The **run()** method is called directly by the VM; you never call **run()** directly.
- * You can give a **Thread** a name when constructed, or with the **setName()** method.

```
thread.setName("MainThread");
```

- **Thread** names are used for display purposes, and can be retrieved and used with the **getName()** method.
- If you do not give a **Thread** a name, a unique name is generated and given to it.

ThreadApp2.java

```
package examples;

public class ThreadApp2 {
    public static void main(String[] args) {
        PrintNumbers3 p1 = new PrintNumbers3(1, 2); // odds
        PrintNumbers3 p2 = new PrintNumbers3(2, 2); // evens
        p1.setName("Odds");
        p2.setName("Evens");
        p1.start();
        p2.start();
    }
}

class PrintNumbers3 extends Thread {
    private int start = 0;
    private int increment = 1;

    public PrintNumbers3(int st, int inc) {
        start = st;
        increment = inc;
    }

    public void print() {
        int i, j;
        for (i = start, j = 0; j < 20; j++, i += increment) {
            System.out.println(this.getName() + " " + i);
        }
    }

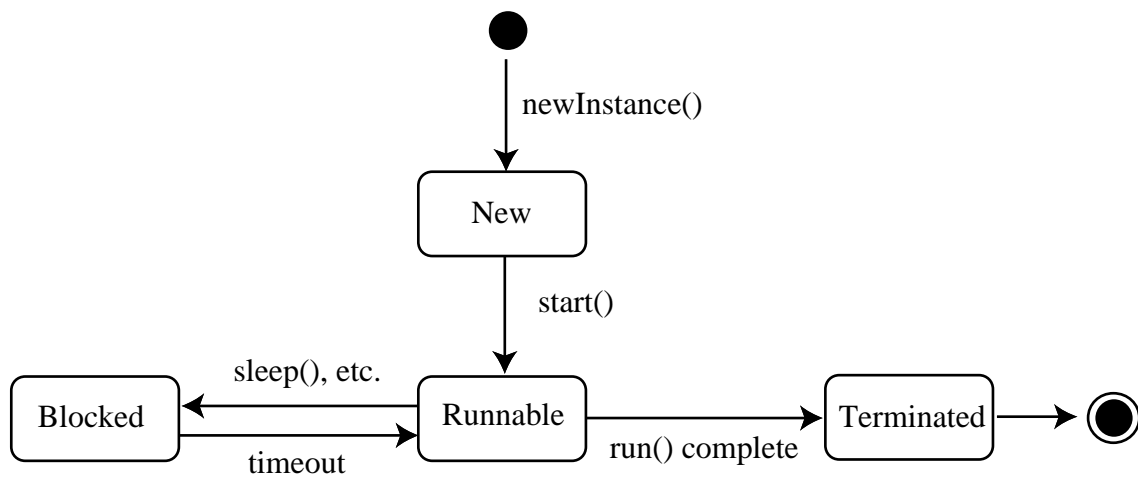
    @Override
    public void run() {
        print();
    }
}
```

THREAD STATES

- * A thread's state is defined by what has been done to it or what it is trying to do.
- * A thread is in a *new* state after it has been created, but before it has been sent a **start()** message.

```
Thread t1 = new Thread();
```

- * A thread is in a *terminated* (dead) state after it has completed its **run()**.
- * A thread is in a *blocked* state when it is waiting for some condition to be met.
 - A thread that is waiting on I/O is blocked until the action occurs.
 - A thread that is sleeping is blocked for a specified amount of time.
 - A thread that is waiting for another thread to finish or to release a resource is also blocked.
- * A thread is in a *runnable* state if:
 - It has been sent a **start()** message.
 - It is not terminated.
 - It is not blocked.
- * A runnable thread is not necessarily a "running" thread.



Beginning in Java 5, the blocked state is broken into three sub-states: **Waiting**, **Timed_Waiting**, and **Blocked**.

RUNNABLE THREADS

- ✴ The VM maintains a queue of all currently-runnable threads.
 - As time slices are allocated to a thread, its **run()** method will proceed until:
 - The thread is blocked.
 - The thread voluntarily gives up its time slice.
 - The VM ends the time slice.
 - The thread dies.
 - If the thread is still runnable, it is placed back in the queue and given additional time slices as they are available.
- ✴ The VM resumes the thread's **run()** method where it left off, as each time slice is awarded to a thread.

The *Thread Scheduler* keeps a queue of all threads in your application. When an actively-running thread either gives up its time, or its time slice is over, it is placed at the end of the thread queue. The first thread in the queue is then made the currently-running thread.

If a thread is in a blocked state, the VM will not try to run that thread until it is unblocked.

The thread queue is actually a priority queue, so a thread with a high priority will not be placed at the end of the queue, but will be placed based on its priority. Thread priorities can be set by calling **setPriority(int newPriority)** on a **Thread**. **static finals** exist to determine the max, min, and normal priorities for threads.

COORDINATING THREADS

✴ **Thread.yield()** causes the currently-executing thread to give up its time slice.

- The thread remains in a runnable state.
- Another runnable thread can start executing.

✴ **Thread.sleep(long *milliseconds*)** causes the currently-executing thread to block for the specified number of milliseconds.

- Once the time has elapsed, the thread returns to the runnable state.
 - It is returned to the thread queue and may be run at any time.

✴ Block the current thread until the death of another by joining the other thread:

```
otherThread.join();
```

- The currently-executing thread is blocked until **otherThread** completes its **run()** method.
- You can pass a **long** to the **join()** method to limit the block to a specified number of milliseconds.
 - If the other thread doesn't complete by that time, the current thread will become runnable again.

ThreadApp3.java

```
package examples;

public class ThreadApp3 {
    public static void main(String[] args) {
        PrintNumbers4 p1 = new PrintNumbers4(1, 2); // odds
        PrintNumbers4 p2 = new PrintNumbers4(2, 2); // evens
        p1.setName("Odds");
        p2.setName("Evens");
        p1.start();
        p2.start();
    }
}

class PrintNumbers4 extends Thread {
    private int start = 0;
    private int increment = 1;

    public PrintNumbers4(int st, int inc) {
        start = st;
        increment = inc;
    }

    public void print() {
        int i, j;
        for (i = start, j = 0; j < 20; j++, i += increment) {
            System.out.println(this.getName() + " " + i);
            Thread.yield();
        }
    }

    @Override
    public void run() {
        print();
    }
}
```

INTERRUPTING THREADS

- * **join()** and **sleep()** are **Thread** methods that do not return immediately.
 - You may need to "wake up" a **Thread** that is in the middle of one of these methods.
 - Perhaps this **Thread** is waiting for something that is never going to happen!
- * Calling **interrupt()** on a **Thread** will cause its **join()** or **sleep()** method to throw an **InterruptedException**.
 - You must handle or declare the **InterruptedException** when using these **Thread** methods.
- * If the thread is not currently blocked, an "interrupted" flag will be set for that thread.
 - You can determine if a **Thread** was interrupted by calling **isInterrupted()**.


```
if (thatThread.isInterrupted())
    // someone tried to interrupt thatThread
```
 - Test the currently-running **Thread** with the **interrupted()** **static** method.


```
if (Thread.interrupted())
    // someone tried to interrupt me,
    // since I am the currently running Thread
```


RUNNABLE INTERFACE

- * It is not always possible to extend the **java.lang.Thread** class.
 - Suppose you are subclassing **JPanel** for a GUI application.
- * Instead, you can implement the **java.lang.Runnable** interface.
 - The **Runnable** interface has a **run()** method, just like a **Thread**.
- * Your **Runnable** class must be associated with an active **Thread** using these steps:

1. Implement the **Runnable** interface:

```
class PrintNumbers5 implements Runnable
```

2. Write the **run()** method to define the life-cycle of your thread:

```
public void run() {  
    // do some neat stuff  
}
```

3. Add a **Thread** field to your class:

```
private Thread theThread;
```

4. Initialize the thread with your **Runnable** object:

```
theThread = new Thread(this);
```

5. Start the thread:

```
theThread.start();
```

- * When the **Thread** is started, the **Thread's run()** method will call the **Runnable** object's **run()** method automatically.

ThreadApp4.java

```
package examples;

public class ThreadApp4 {
    public static void main(String[] args) {
        PrintNumbers5 p1 = new PrintNumbers5(1, 2); // odds
        PrintNumbers5 p2 = new PrintNumbers5(2, 2); // evens
        p1.setThreadName("Odds");
        p2.setThreadName("Evens");
        p1.startThread();
        p2.startThread();
    }
}

class PrintNumbers5 implements Runnable {
    private int start = 0;
    private int increment = 1;
    private Thread theThread = null;

    public PrintNumbers5(int st, int inc) {
        start = st;
        increment = inc;
        theThread = new Thread(this);
    }

    public void print() {
        int i, j;
        for (i = start, j = 0; j < 20; j++, i += increment) {
            System.out.println(theThread.getName() + " " + i);
            Thread.yield();
        }
    }

    public void run() {
        print();
    }

    public void startThread() {
        theThread.start();
    }

    public void setThreadName(String name) {
        theThread.setName(name);
    }
}
```


LABS

- 1 Extend a class from **Thread** called **ReadFile**. The class should have a constructor that takes a filename and an object number (an **int**) as its arguments.

ReadFile's **run()** method should have a loop that reads five lines from the file, sends each line to the screen with this object's number, then calls **yield()**.

Add a main method to your class that instantiates three different **ReadFile** objects. Each one should be given a unique object number. Call **start()** on each object.
(Solution: *ReadFile.java*)

- 2 Change your solution to 1 to implement **Runnable** instead of extending **Thread**.
(Solution: *ReadFile2.java*)

CHAPTER 10 - THREAD SYNCHRONIZATION AND CONCURRENCY

OBJECTIVES

- ✧ Use the **synchronized** keyword to avoid race conditions.
- ✧ Store data in thread safe collections.
- ✧ Create thread pools with the **Executors** class.
- ✧ Implement the **Callable** interface to return a value from a thread.

RACE CONDITIONS

- * Threads share code and data space.
 - Two threads running in the same object can both modify the same field.
 - The timing of context switches may result in one thread being suspended in the middle of modifying a field.
 - Meanwhile another thread may modify the same field.
 - When the first thread runs again, it may be working with old data.
 - This behavior is called a *race condition*.
- * Each thread maintains its own stack.
 - Local variables are not susceptible to race conditions.
- * Beware: context switches can occur in the middle of a statement.
 - A single Java source code statement can be made up of multiple byte code instructions.

Account.java

```
...
public class Account {
    private double balance;

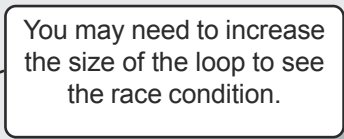
    public Account(double initialBalance) {
        balance = initialBalance;
    }

    public void deposit(double amount) {
        double tempBalance = balance;
        tempBalance = tempBalance + amount;
        balance = tempBalance;
    }
    ...
}
```

AccountTester.java

```
...
public class AccountTester extends Thread {
    private Account theAccount = null;
    private double depositAmount = 0.0;
    ...
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            theAccount.deposit(depositAmount);
        }
    }
    public static void main(String[] args) {
        Account account = new Account(0.0);
        AccountTester tester1 = new AccountTester(account, 2);
        AccountTester tester2 = new AccountTester(account, 3);

        tester1.start();
        tester2.start();
        ...
        System.out.println("End balance is: " + account.getBalance());
    }
}
```



Try It:

Compile and run *AccountTester.java* to see a race condition. If the end balance is 5,000, then run it again.

SYNCHRONIZED METHODS

- ✴ To avoid race conditions, mark any methods that modify the invoking object's fields as **synchronized**.

```
public synchronized void deposit(double amount) {...}
```

- ✴ The Virtual Machine maintains a single "lock" and "key" for each object.
 - In order to enter a **synchronized** block of code, a thread must lock the object and hold the key.
 - If a thread encounters a locked section of code, then it blocks until the key is available.
 - Upon exiting a **synchronized** block of code, the thread unlocks the object and returns the key.
- ✴ Only one thread at a time may be in any **synchronized** block within an object.
 - Synchronization is applied at the object level, not at the class level.
- ✴ Many threads can run in un-**synchronized**, unlocked sections of code simultaneously.

AccountSynchronized.java

```
package examples;

public class AccountSynchronized {
    private double balance;

    public AccountSynchronized(double initialBalance) {
        balance = initialBalance;
    }

    public synchronized void deposit(double amount) {
        double tempBalance = balance;
        tempBalance = tempBalance + amount;
        balance = tempBalance;
    }

    public synchronized void withdraw(double amount) {
        balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

AccountTester2.java

```
...
public class AccountTester2 extends Thread {
    ...
    public static void main(String[] args) {
        AccountSynchronized account = new AccountSynchronized(0.0);
        AccountTester2 tester1 = new AccountTester2(account, 2);
        AccountTester2 tester2 = new AccountTester2(account, 3);

        tester1.start();
        tester2.start();

        ...
        System.out.println("End balance is: " + account.getBalance());
    }
}
```

Try It:

Compile and run *AccountTester2.java*.

DEADLOCKS

- ✴ If making your code **synchronized** prevents race conditions, then why not synchronize all threaded code?
 - Code running in **synchronized** blocks is slower.
 - **synchronized** code is single-threaded.
 - Excessively marking code as **synchronized** may lead to deadlocks.
 - A deadlock occurs when one thread, which has locked *Object A*, wants to call a **synchronized** method on *Object B*.
 - However, *Object B* has already been locked by another thread that is currently waiting to acquire the key for *Object A*.
 - Neither thread can proceed until the other releases its lock.
- ✴ All it takes is two objects and two threads to have a deadlock.
- ✴ Minimize the amount of **synchronized** code to prevent deadlocks.
- ✴ Design your code to avoid deadlock conditions.

DeadLockExample.java

```
...
public class DeadLockExample {
    public static ClassA a;
    public static ClassB b;

    public static void main(String[] args) {
        a = new ClassA();
        b = new ClassB();
        a.start();
        b.start();
    }
}

class ClassA extends Thread {
    public synchronized void method1() {
        System.out.println(this.getName() + " acquired A's lock");
        Thread.yield(); // force deadlock
        System.out.println(this.getName() + " calling B's method");
        DeadLockExample.b.method4();
        System.out.println(this.getName() + " released A's lock");
    }

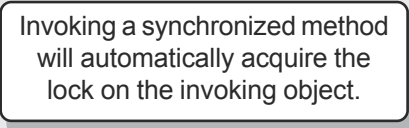
    public synchronized void method2() { }

    public void run() {
        this.method1();
    }
}

class ClassB extends Thread {
    public synchronized void method3() {
        System.out.println(this.getName() + " acquired B's lock");
        System.out.println(this.getName() + " calling A's method");
        DeadLockExample.a.method2();
        System.out.println(this.getName() + " released B's lock");
    }

    public synchronized void method4() { }

    public void run() {
        this.method3();
    }
}
```



Try It:

Compile and run *DeadLockExample.java* to cause a deadlock. At a command prompt, run the **jstack** command followed by the process id of your Java program to see a thread dump and find the deadlock.

SYNCHRONIZED BLOCKS

- * You should minimize the amount of **synchronized** code to avoid deadlocks.
- * Marking a method **synchronized** locks the invoking object for the duration of the method call.

```
public synchronized void deposit(double amount) {...}
```

- * You can also use the **synchronized** keyword to synchronize a block of code.

```
synchronized (this) {  
    // code to synchronize  
}
```

- The object you pass in the parentheses is the one whose lock is held.

AccountSynchronized2.java

```
package examples;

public class AccountSynchronized2 {
    private double balance;

    public AccountSynchronized2(double initialBalance) {
        balance = initialBalance;
    }

    public void deposit(double amount) {
        synchronized (this) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        synchronized (this) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

If you have studied concurrency before, then the terminology you may be used to is *mutex* instead of lock, *monitor* for the object that we are locking, and *critical section* for the portion of code that is wrapped by the **synchronized** block.

SYNCHRONIZED COLLECTIONS

- * Collections, such as **ArrayList** and **HashMap**, were written in a non thread-safe manner for performance purposes.
 - The legacy classes **Vector** and **Hashtable** are thread-safe, but performance suffers because their methods are synchronized.
 - **Collections'** **synchronizedCollection()**, **synchronizedList()**, and **synchronizedMap()** methods convert a non-synchronized collection into a synchronized version of it.
- * Even with the synchronized version of a collection, it may not be sufficiently safe.
 - Iteration, for example, may need additional synchronization since the size of the list can change if another thread is concurrently removing items from the list.

```
synchronized(list) {
    for (int i=0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```

- Use a **synchronized** block to lock the list itself while iterating.

States.java

```

...
public class States {
    public static void main(String[] args) {
        List<String> nonThreadSafeStates = getListOfStatesAndTerritories();

        List<String> states = Collections
            .synchronizedList(nonThreadSafeStates);

        removeTerritories(states);

        // Loop through the list, displaying each value
        synchronized (states) {
            Iterator<String> it = states.iterator();
            while (it.hasNext()) {
                System.out.print(it.next());
                if (it.hasNext())
                    System.out.print(", ");
                else
                    System.out.println();
            }
        }

        private static void removeTerritories(List<String> states) {
            new Thread(new Runnable() {
                public void run() {
                    synchronized (states) {
                        for (int i = 0; i < states.size(); i++) {
                            String state = states.get(i);
                            switch (state) {
                                case "AS": case "GU": case "PR": case "VI":
                                    states.remove(i);
                                    break;
                            }
                        }
                    }
                }
            }).start();
        }
    }
}

```

Create a "Thread Safe" version of the list.

Synchronize the list to avoid a **ConcurrentModificationException**.

In a separate thread, modify the list.

Try It:

Run *States.java* with and without the synchronized blocks to see why the additional synchronization is necessary even though the list itself is synchronized.

THREAD-AWARE COLLECTIONS

- ✴ **java.util.concurrent** provides additional versions of standard collection classes that can be accessed in a concurrent context.
 - Most of these classes use the **java.util.concurrent.locks**' locking mechanisms to avoid suffering from the performance bottleneck of the single object lock that **synchronized** uses.

- ✴ **CopyOnWriteArrayList** and **CopyOnWriteArraySet** perform well for read operations by allowing multiple, simultaneous reader threads.
 - Modifying the object involves the changes being written to a new copy of the underlying array that is later copied back into the original.
 - An **Iterator** retrieved before the copy was made will utilize a snapshot of the original array.
 - Any new **Iterators** will be based on the copy.
 - Copying the array is an expensive operation, so only use these classes when you do a lot more reading than writing.

- ✴ **ConcurrentHashMap** can be used instead of a **synchronized HashMap** for multi-threaded access to a **Map** without the overhead of synchronization.
 - **ConcurrentHashMap** does not synchronize reads.
 - A read operation will return the most recent update.
 - By default, up to 16 simultaneous writes can occur due to the way that the collection is segmented.

States2.java

```
...
public class States2 {

    public static void main(String[] args) {
        List<String> nonThreadSafeStates = getListOfStatesAndTerritories();

        CopyOnWriteArrayList<String> states = new CopyOnWriteArrayList<>(  
            nonThreadSafeStates);

        removeTerritories(states);

        // Loop through the list, displaying each value
        Iterator<String> it = states.iterator();
        while (it.hasNext()) {
            System.out.print(it.next());
            if (it.hasNext())
                System.out.print(", ");
            else
                System.out.println();
        }
    }

    private static void removeTerritories(List<String> states) {
        new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < states.size(); i++) {
                    String state = states.get(i);
                    switch (state) {
                        case "AS":
                        case "GU":
                        case "PR":
                        case "VI":
                            states.remove(i);
                            break;
                    }
                }
            }
        }).start();
    }
    ...
}
```

There is no need to synchronize the iteration because we are using a **CopyOnWriteArrayList**.

Since this is done in a **CopyOnWriteArrayList**, each call to **remove()** results in a new copy of the backing array. Any existing **Iterators** will use the old copy, all new **Iterators** will use the new copy.

Try It:Compile and run *States2.java* to use a **CopyOnWriteArrayList**.

EXECUTOR

- ✴ Use the **Executor** interface to launch a **Runnable**, rather than instantiate a **Thread** and call **start()** on it.

```
Executor ex = ...;
ex.execute(new Task1); // Task1 implements Runnable
ex.execute(new Task2); // Task2 implements Runnable
instead of
```

```
Thread t1 = new Thread(new Task1);
Thread t2 = new Thread(new Task2);
t1.start();
t2.start();
```

- **Executor** is an interface that defines a single **execute()** method.
- ✴ Use the **Executors** class as a factory for creating **Executor** implementations.
 - **Executors.newFixedThreadPool(int *poolSize*)** uses a fixed-size pool of threads to run your tasks in.


```
Executor ex = Executors.newFixedThreadPool(2);
ex.execute(new Task1);
ex.execute(new Task2);
```
 - Use **Executors.newCachedThreadPool()** to create a pool with as many threads as necessary to run your tasks.
 - In both cases, threads will be reused after they have completed their task.
 - Each method returns a specialized version of **Executor** called **ExecutorService**.
 - An **ExecutorService** waits for more tasks to invoke, so call the methods **shutdown()** or **shutdownNow()** to tell the **ExecutorService** to exit.
 - **shutdown()** will complete any pending tasks before exiting.
 - **shutdownNow()** will immediately end the **ExecutorService**, even if some threads are currently running.

ThreadApp5.java

```
package examples;

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ThreadApp5 {
    public static void main(String[] args) {
        ExecutorService ex = Executors.newFixedThreadPool(2);
        ex.execute(new PrintNumbers6(1, 2));
        ex.execute(new PrintNumbers6(2, 2));
        ex.shutdown();
        //ex.shutdownNow();
    }
}

class PrintNumbers6 implements Runnable {
    private int start=0;
    private int increment=1;

    public PrintNumbers6(int st, int inc) {
        start = st;
        increment = inc;
    }

    public void print() {
        int i, j;
        for (i=start, j=0; j<20; j++, i+=increment) {
            System.out.println(Thread.currentThread().getName() + " " + i);

            if (Thread.interrupted()) {
                break;
            }
            Thread.yield();
        }
    }

    public void run() {
        print();
    }
}
```

Check the **interrupted** flag to properly handle **shutdownNow()**.

Try It:

Compile and run *ThreadApp5.java* to use an **Executor**. Try **shutdownNow()** instead of **shutdown()** to see the **Executor** exit immediately. Try to run the code without either **shutdown()** method. What happens?

CALLABLE

- ✧ Whether you are extending **Thread** or implementing **Runnable**, the **run()** method you define always returns **void**.
- ✧ The **Callable** interface was introduced to substitute for **Runnable** when you want the thread to return back a value.
 - **Callable** defines a single **call()** method that you must implement in your class.
 - The **Callable** interface makes use of generics to define the return type of the **call()** method.

```
public class PiCalc implements Callable<Double> {  
    public Double call() {  
        // compute pi to lots of digits  
    }  
}
```

- Pass the **Callable** object to an **ExecutorService's submit()** method.
- **submit()** returns a **Future** object that can be later queried for the result using the blocking **get()** method.

```
ExecutorService ex = Executors.newCachedThreadPool();  
Future<Double> result = ex.submit(new PiCalc());  
// do other stuff  
Double pi = result.get();
```

PiCalc.java

```
package examples;

import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

import static java.lang.Math.*;

public class PiCalc implements Callable<Double> {
    public Double call() {
        // Machin's formula
        // pi/4 = 4 * arctan(1/5) - arctan(1/239);
        double pi = 16 * atan(1.0/5.0) - 4 * atan(1.0/239.0);

        try {
            // Sleep for 1 second to simulate a long running task
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        return pi;
    }

    public static void main(String[] args) throws Exception {
        ExecutorService ex = Executors.newCachedThreadPool();
        Future<Double> result = ex.submit(new PiCalc());
        // do other stuff
        Double pi = result.get();
        System.out.println(pi);
        ex.shutdown();
    }
}
```

The calculation can be done in a background thread while other tasks are accomplished.

LABS

- ① Compile and run *Race.java*. Do you see a race condition? If not, increase the size of the loop until you see a race condition. What is causing the race condition? (Hint: Look in `%JAVA_HOME%\src.zip` for the source code to *ArrayList.java*.)
(Solution: *Race.java*, *racecondition.txt*)
- ② Use the **synchronized** keyword to modify *Race.java* so that the race condition no longer exists.
(Solution: *SyncRace.java*)
- ③ Replace **ArrayList** with a different collection that will prevent the race condition.
(Solution: *CollectionRace.java*)
- ④ Modify your code again to implement **Callable** rather than **Runnable**. Have your **call()** method return the **String** "Done."
(Solution: *CallableRace.java*)

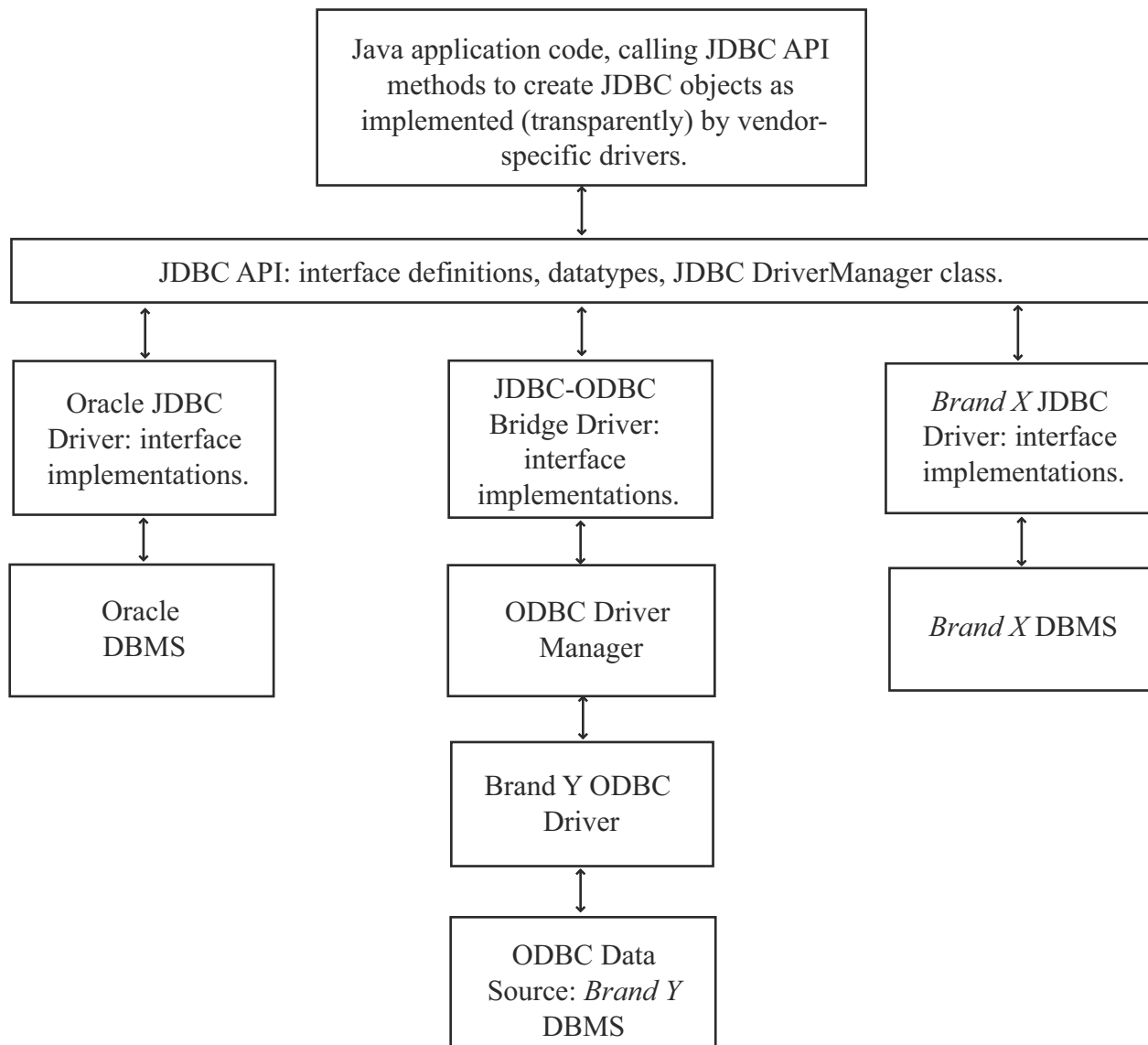
CHAPTER 11 - INTRODUCTION TO JDBC

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 **SQLExceptions**. 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:mysql://localhost:3306/companydb";
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 {
    private static String url = "jdbc:mysql://localhost:3306/companydb";
    public static void main(String args []) throws SQLException {
        String user = "student";
        String pword = "student";
        Connection conn = DriverManager.getConnection(url,user,pword);
        conn.close();
    }
}
```

Note:

This code uses the MySQL Connector 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.

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

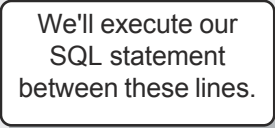
```
package examples;

import java.sql.*;

public class JDBCTest {
    private static String url = "jdbc:mysql://localhost:3306/companydb";
    public static void main(String args []) throws SQLException {
        String user = "student";
        String pword = "student";
        Connection conn = DriverManager.getConnection(url,user,pword);

        String sqltxt;
        sqltxt = "SELECT id, firstname, lastname FROM employees";
        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** **getXXX()** 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 {
    private static String url = "jdbc:mysql://localhost:3306/companydb";
    public static void main(String args []) throws SQLException {
        String user = "student";
        String pword = "student";
        Connection conn = DriverManager.getConnection(url,user,pword);

        String sqltxt;
        sqltxt = "SELECT id, firstname, lastname FROM employees";
        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 {
    private static String url = "jdbc:mysql://localhost:3306/companydb";
    public static void main(String args []) throws SQLException {
        String user = "student";
        String pword = "student";
        Connection conn = DriverManager.getConnection(url,user,pword);

        String sqltxt;
        sqltxt = "SELECT id, firstname, lastname FROM employees";
        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 employees SET lastname = 'Smithers' " +
                 " WHERE id = 1123";
        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 **ResultSets** 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 {
    private static String URL = "jdbc:mysql://localhost:3306/companydb";
    public static void main(String args[]) {
        String user = "student";
        String pword = "student";
        String sqltxt = "SELECT id, firstname, lastname FROM employees";

        try (Connection conn=DriverManager.getConnection(URL,user,pword);
            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

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

CHAPTER 12 - JDBC SQL PROGRAMMING

OBJECTIVES

- * Use **SQLExceptions** and **SQLWarnings** to detect and handle DBMS errors and warnings.
- * Describe the most important JDBC interfaces, and use their methods.
- * Use the JDBC datatypes to convert DBMS-specific data to Java data.
- * Use **ResultSetMetaData** to handle dynamically-generated SQL.
- * Use **PreparedStatement** objects to more efficiently run a SQL statement repeatedly.
- * Invoke DBMS stored procedures and functions from Java.
- * Manage DBMS transactions.

ERROR CHECKING AND THE **SQLException** CLASS

- * A runtime error from the DBMS triggers a **SQLException**.
- * The **SQLException**'s methods provide the standard DBMS error information:
 - **getSQLState()** returns the ANSI-standard SQLSTATE value (if your DBMS supports it).
 - **getErrorCode()** returns the vendor-specific SQLCODE error code.
 - **getMessage()** is overridden by the **SQLException** class to return the vendor-specific error message.
- * A single action might result in multiple database errors, so a **SQLException** may have one or more additional **SQLExceptions** chained to it.
 - **getNextException()** returns the next exception in the chain.
- * Most methods defined under **java.sql** throw **SQLException**; assume that your JDBC code needs to either catch or declare **SQLException**.

Database programmers rely on a simple mechanism for obtaining error information from the database engine: after every database operation, the DBMS sets an error value that the programmer can check. An error is typically identified by a numeric error code (SQLCODE) and a brief error message. By convention, DBMS error codes are negative numbers; a SQLCODE value of 0 means no error has occurred; and a special value, typically (but not always) +100, means there was no error, but no data was found to satisfy the operation.

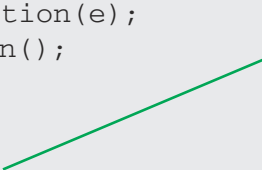
SQLCODE values are vendor-specific, however, so programmers must learn the specific code values for each product they program against. To relieve this situation, standards organizations introduced the SQLSTATE, a standardized, 5-character string encoding the error condition.

Some DBMSs now implement SQLSTATE, though most still also return their own, vendor-specific, SQLCODE values and messages.

With JDBC, when you call certain methods, Java automatically checks the SQLSTATE/SQLCODE and throws a **SQLException** when there's an error.

SQLUtils.java

```
...
public class SQLUtils {
    public static String formatSQLException(SQLException e) {
        StringBuilder msg = new StringBuilder("");
        if (e != null) {
            msg.append(" SQLState: " + e.getSQLState() + "\n");
            msg.append("      Code: " + e.getErrorCode() + "\n");
            msg.append(" Message: " + e.getMessage() + "\n\n");
        }
        return msg.toString();
    }
    public static String formatSQLExceptions(SQLException e) {
        String msg = "";
        while (e != null) {
            msg += formatSQLException(e);
            e = e.getNextException();
        }
        return msg;
    }
    public static void printSQLExceptions(SQLException e) {
        if (e != null) {
            System.err.println("SQL Error:\n" + formatSQLExceptions(e));
        }
    }
    ...
}
```



Call this method from a **catch** handler to print nice error messages.

JDBC TYPES

- * JDBC defines datatypes to separate the Java developer from the database implementation.
 - Database vendors often support different types of data or use different names for the datatypes.
 - The driver is responsible for converting DBMS types to JDBC types.
 - JDBC datatypes are based on standard SQL datatypes.

- * When you use a **getXXX()** method on a **ResultSet**, the **XXX** refers to the Java type that is returned.

```
int id = rs.getInt(1);
String lastName = rs.getString(2);
```

- There is a suggested **getXXX()** method for each JDBC datatype, but there is usually more than one choice.
 - **getObject()** works for any JDBC type.
- Look in the mapping table to see which JDBC types are supported by which **getXXX()** methods.
- * The **java.sql.Types** class defines a **static final int** for each JDBC datatype.
 - These constants are the way that JDBC references the type.

Mapping Table

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte()	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getShort()	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getInt()	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getLong()	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓												
getFloat()	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓												
getDouble()	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓												
getBigDecimal()	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓												
getBoolean()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓												
getString()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓						
getBytes()														✗	✗	✓									
getDate()											✓	✓	✓				✗		✓						
getTime()											✓	✓	✓					✗	✓						
getTimestamp()											✓	✓	✓				✓	✓	✗						
getAsciiStream()											✓	✓	✗	✓	✓	✓									
getBinaryStream()														✓	✓	✗									
getCharacterStream()											✓	✓	✗	✓	✓	✓									
getClob()																				✗					
getBlob()																					✗				
getArray()																						✗			
getRef()																							✗		
getObject()	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

✗ — Preferred

✓ — Optional (can implicitly convert)

EXECUTING SQL QUERIES

- ✴ Use **Statement**'s **executeQuery()** method to run a SQL **SELECT** statement, which will return a single **ResultSet**.
 - If the **Statement** already had an open **ResultSet**, it is automatically closed first.
- ✴ **executeQuery()** takes a single **String** argument: the SQL **SELECT** statement.
- ✴ Dynamic SQL is simple with JDBC.
 - Your program builds SQL query strings and passes them to **executeQuery()**.
 - The database parses and compiles the query string each time you call **executeQuery()**.

EmpSals.java

```
...
public class EmpSals {

    public static void main(String args[]) {
        String url = "jdbc:mysql://localhost:3306/companydb";
        String user = "student";
        String pword = "student";
        String sqltxt;
        sqltxt = "SELECT firstname, lastname, salary, departments.name"
            + " FROM employees JOIN departments "
            + " ON employees.department_id = departments.id";

        try (Connection conn=DriverManager.getConnection(url,user,pword);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sqltxt);) {

            String name;
            String dept;
            float salary;
            while (rs.next()) {
                name = rs.getString("firstname") + " "
                    + rs.getString("lastname");
                dept = rs.getString("name");
                salary = rs.getFloat("salary");
                System.out.println(name + "\t" + dept + "\t" + salary);
            }
        }
        catch (SQLException sqle) {
            System.err.println("SQL Error:");
            System.err.println("      Code: " + sqle.getErrorCode());
            System.err.println("  Message: " + sqle.getMessage());
        }
    }
}
```



RESULTSETMETADATA

- * For queries built at runtime for which you need to determine the number, types, names, etc. of columns of a **ResultSet**, get the **ResultSet**'s metadata.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

- * A **ResultSetMetaData** object's methods return information about the columns of its **ResultSet**.

```
int getColumnCount()
String getColumnName(int column)
String getColumnLabel(int column)
int getColumnType(int column)
String getColumnName(int column)
int getPrecision(int column)
int getScale(int column)
int getColumnDisplaySize(int column)
int isNullable(int column)
boolean isReadOnly(int column)
boolean isCurrency(int column)
etc...
```

- All of these methods may throw **SQLException**.

- * **getColumnType()** returns an **int** corresponding to a constant defined in **java.sql.Types**.

- This can be useful in determining which **getXXX()** method to call.

- * **ResultSetMetaData** is valuable when:

- **execute()** was used to run some arbitrary query.
- **executeQuery()** was used to run a query statement that was generated at runtime.

ExecuteFormat.java

```
...
public class ExecuteFormat {

    public static void main(String args[]) {
        ...
        ResultSetMetaData rsmd = rs.getMetaData();
        int cols = rsmd.getColumnCount();
        int colWidth[] = new int[cols];

        String resultLine = "";
        for (int col=1; col <= cols; col++) {
            colWidth[col-1] = getAppDisplayWidth(rsmd, col);
            resultLine += SQLUtils.rPadTrunc(rsmd.getColumnName(col),
                colWidth[col-1]);
            if (col < cols) {
                resultLine += " ";
            }
        }
        System.out.println(resultLine);
        ...
    }

    static int getAppDisplayWidth(ResultSetMetaData rsmd, int col)
        throws SQLException {

        switch (rsmd.getColumnType(col)) {
            case Types.NUMERIC:
            case Types.INTEGER:
                return 9;
            case Types.TIMESTAMP:
                return 22;
            case Types.DATE:
                return 10;
            case Types.TIME:
                return 8;
            default :
                return rsmd.getColumnDisplaySize(col);
        }
    }
}
```

Try It:

Run *ExecuteFormat.java* to see all **Department** data printed out in columns with headers.

EXECUTING SQL UPDATES

- * Use **Statement's executeUpdate()** method to run SQL statements that do not generate result sets.

- Data Manipulation Language (DML): **INSERT, UPDATE, DELETE.**

```
stmt.executeUpdate( "DELETE FROM employees " +
    " WHERE id = 1133");
```

- Data Definition Language (DDL): **CREATE, ALTER, DROP, GRANT, REVOKE.**

```
stmt.executeUpdate( "GRANT SELECT ON employees " +
    " TO '%@'localhost");
```

- * **executeUpdate()** returns an **int**, the *update count*, which is one of two values:

```
int uc;
uc = stmt.executeUpdate( "UPDATE employees "
    + " SET salary = salary * 1.1 "
    + "WHERE job_id = 9");
```

- The number of rows affected by a DML statement (may be **0**).
- **0** for DDL or other statements.

- * **executeUpdate()** throws an exception if you use it to run a **SELECT** or other statement that generates a result set.

The **Statement** interface's **execute()** method can execute an arbitrary SQL string that might generate either a result set or an update count. The price of this generality is some extra coding to retrieve the actual results. Usually, though, you will use **executeQuery()** for queries and **executeUpdate()** for DML and DDL.

THE EXECUTE() METHOD

- * **Statement's execute()** method is the most general way to execute SQL.
- * **execute()** executes any arbitrary SQL string.
 - It may generate either a result set, an update count, a combination of any number of either, or even no results at all; for example:
 - A statement built dynamically at runtime, which you can't restrict to either a query or an update.
 - A stored procedure yielding multiple result sets or update counts (if the DBMS supports this).
- * **execute()** returns a **boolean**.
 - **true** if the statement generated an initial result set.
 - **false** if the statement generated either an initial update count or no results at all.
- * Use **getResultSet()** and **getUpdateCount()** to retrieve the actual results of the statement.
 - **getResultSet()** returns a **ResultSet** object if one is available, otherwise **null**.
 - **getUpdateCount()** returns an **int**: **0** or greater for DML or DDL, **-1** if no update count is available.

After an **execute()**, your **Statement** object has either a **ResultSet**, an update count, or neither. In addition, after you are finished with that result (either the **ResultSet** or the update count), there may be another result to retrieve and process. If you have no idea what to expect, you can still process all of your results:

ExecuteExample.java

```
...
public class ExecuteExample {
    public static void main(String args[]) {
        String url = "jdbc:mysql://localhost:3306/companydb";
        String user = "student";
        String pword = "student";
        String sqlstring = readInSQLStatement();
        System.out.println("Executing SQL statement:\n" + sqlstring
            + "\n");
        try (Connection conn=DriverManager.getConnection(url,user,pword);
            Statement stmt = conn.createStatement();) {

            boolean haveResultSet = stmt.execute(sqlstring);
            if (haveResultSet) {
                ResultSet rs = stmt.getResultSet();
                ResultSetMetaData rsmd = rs.getMetaData();
                int cols = rsmd.getColumnCount();
                for (int col = 1; col <= cols; col++)
                    System.out.print(rsmd.getColumnName(col) + "\t");
                System.out.println("");
                while (rs.next()) {
                    for (int col = 1; col <= cols; col++) {
                        System.out.print(rs.getString(col) + "\t");
                    }
                    System.out.println("");
                }
                rs.close();
            }
            else { // No result set.
                int uc = stmt.getUpdateCount();
                System.out.println(uc + " row(s) updated.");
            }
        }
        ...
    }
    ...
}
```

Try It:

Run *ExecuteExample.java*. Enter a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** SQL statement when prompted.

USING A PREPAREDSTATEMENT

- ✴ If your program executes the same SQL statement repeatedly, perhaps just with different values in certain places, you may benefit by using a **PreparedStatement**.

- Use **Connection**'s **prepareStatement()** method to instantiate a **PreparedStatement** object.

```
PreparedStatement pst;
pst=conn.prepareStatement("SELECT name"
    + " FROM departments"
    + " WHERE id = 3");
ResultSet rs = pst.executeQuery();
```

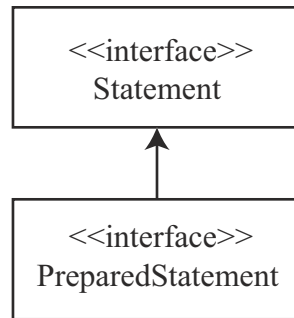
- You must supply the SQL syntax when you create the **PreparedStatement**, not when you execute it.

- ✴ Where a **Statement**'s *execute* methods (**execute()**, **executeQuery()**, **executeUpdate()**) cause the DBMS to parse, compile, and execute the SQL syntax each time, a **PreparedStatement**'s SQL syntax may be parsed and compiled by the DBMS only once.

- ✴ **PreparedStatement** descends from **Statement**.

- Note that a **PreparedStatement**'s *execute* methods don't take a SQL **String** argument — thus they don't override the **Statement**'s *execute(String sqlstr)* methods.

- Do not use the **Statement**'s *execute(String sqlstr)* methods with a **PreparedStatement** object.



PARAMETERIZED STATEMENTS

- * You can execute a prepared statement multiple times with different values inserted into the statement syntax.
- * When creating the statement, use a question mark, **?**, as a placeholder for any value you'll change later.

```
PreparedStatement pst;
pst = conn.prepareStatement( "SELECT name "
    + " FROM departments "
    + " WHERE id = ?");
```

- * You must set values for all placeholders before executing the statement.
- Use a **PreparedStatement**'s **setXXX()** methods, where **XXX** is the Java datatype (the JDBC driver implicitly converts to the appropriate JDBC datatype).

```
pst.setInt(1, 3);
ResultSet rs = pst.executeQuery();
```

- The **setXXX()** methods take the index of the placeholder, from left-to-right starting at **1**, as the first argument.

```
int dept = 3;
int mgrid = 1134;
pst = conn.prepareStatement("UPDATE departments "
    + " SET manager_id = ? "
    + " WHERE id = ?");
pst.setInt(1, mgrid);
pst.setInt(2, dept);
int uc = pst.executeUpdate();
System.out.println(uc + " row(s) updated.");
```

- * Parameters assigned to a **PreparedStatement** are sometimes referred to as *IN* parameters: values passed into the statement.

This example uses a prepared statement to retrieve the department name using the department code.

PrepParamExample.java

```
...
public class PrepParamExample {
    public PrepParamExample() {
        String url = "jdbc:mysql://localhost:3306/companydb";
        String user = "student";
        String pword = "student";
        String sql = "SELECT name FROM departments "
            + "WHERE id = ?";

        try (Connection conn = DriverManager.getConnection(url,user,pword);
            PreparedStatement pst = conn.prepareStatement(sql);) {

            System.out.println(getDeptName(pst, 9));
            System.out.println(getDeptName(pst, 3));
        }
        catch (SQLException e) {
            System.err.println(e);
        }
    }

    private String getDeptName(PreparedStatement pst, int deptid)
        throws SQLException {
        pst.setInt(1, deptid);
        ResultSet rs = pst.executeQuery();
        String dept = "";
        while (rs.next()) {
            dept += rs.getString(1) + "\n";
        }
        return dept;
    }

    public static void main(String args[]) {
        new PrepParamExample();
    }
}
```



Try It:

Run this program. The output will be the department names for the **"RD"** and **"HR"** departments.

TRANSACTION MANAGEMENT

- ✧ A DBMS logs all DML operations in a session as part of a *transaction*.
 - At your request, the DBMS will either:
 - *Commit* the entire transaction, making your operations permanent and visible to other users.
 - *Rollback* the entire transaction, completely undoing all DML operations.
 - When you commit or rollback a transaction, a new transaction begins.
- ✧ The default behavior of a **Connection** object is to automatically commit after executing each statement.
 - In this case, each statement is a single, independent transaction.
- ✧ To turn off autocommit mode, use **setAutoCommit(false)**.
 - You can then use the **commit()** or **rollback()** methods of your **Connection** object to manage your transactions.
 - The **getAutoCommit()** method of your **Connection** object returns **true** when autocommit mode is on.

In most cases, autocommit is not desirable — you don't have the ability to check errors and warnings and program appropriately, or to treat multiple individual statements as parts of a single transaction, or to set transaction savepoints (if your DBMS supports them).

Transaction Side Effects

- Some DBMSs, notably Oracle, automatically commit the current transaction any time you execute a DDL statement.
- Committing or rolling back the current transaction frees all locks.
- Committing the current transaction frees the update locks associated with an updatable cursor, thus forcing the cursor closed.

Isolation Level

Your transaction *isolation level* describes how your session behaves in regard to ongoing, uncommitted transactions performed by other users.

Connection.getTransactionIsolation() retrieves an integer value identifying your current isolation level, and **setTransactionIsolation()** sets it for your current session. The values for these are defined as **static finals** in the **Connection** interface:

Connection.TRANSACTION_NONE	Transactions are not supported.
Connection.TRANSACTION_READ_UNCOMMITTED	Dirty reads, non-repeatable reads, and phantom reads can occur.
Connection.TRANSACTION_READ_COMMITTED	Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
Connection.TRANSACTION_REPEATABLE_READ	Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
Connection.TRANSACTION_SERIALIZABLE	Dirty reads, non-repeatable reads, and phantom reads are prevented.

Note:

Your DBMS may not provide all isolation levels.

LABS

- ❶ Write a program to print a listing of all employee names and salaries, and the total of all salaries at the end.
(Solution: *SalRpt1.java*)
- ❷ Modify the salary report program from *SalRpt1.java* so that after the list of all employee names, it prints a list of the department names with salary totals for each. (Hint: Use a **HashMap** to accumulate the salary total for each department.)
(Solution: *SalRpt2.java*)
- ❸ (Optional) Modify the salary report program to use **rPadTrunc()**, found in the provided *SQLUtils.java*, to format the output.
(Solutions: *SalRpt3.java*, *SQLUtils.java*)
- ❹ Write a program that uses a **PreparedStatement** to retrieve the id, name, and title of all employees of a department. The statement should take the department code as its only parameter. Prompt the user for a department code and use the response to run the query and print the results.
(Solution: *DeptEmps.java*)
- ❺ Write a program that prompts the user to enter a SQL statement (using **System.in** or a simple GUI, as you prefer). The program should then determine if the statement is a query or not, and execute it using the appropriate method. If it is a query, just print out the number of rows found. If it is a DML or other statement, print the number of rows affected.
(Solution: *SQLExec.java*)
- ❻ Modify your solution to *SQLExec.java*: if the statement is executed as a query, use **ResultSetMetaData** to determine the number of columns retrieved by the query. For each row, print out all columns.
(Solution: *SQLExec2.java*)

INDEX

SYMBOLS

? 208

A

Abstract Windowing Toolkit (AWT) 114
ActionEvent class 120, 128
ActionListener interface 120, 128
anonymous class 100, 106, 108
 inner 106
application
 thread 138
ArrayList class 24, 92, 164
assertion 58
autocommit 210
AWT 114

B

BorderLayout class 122, 123
BoxLayout class 122
bucket 94
BufferedReader class 31
byte stream 32

C

Callable interface 170
captured group 64
character
 class 54
 stream 32
class
 ActionEvent 120, 128
 anonymous 100, 106, 108
 anonymous inner 106
 ArrayList 24, 92, 164
 BorderLayout 122, 123
 BoxLayout 122
 BufferedReader 31
 Collections 88, 90
 ConcurrentHashMap 166
 DataInputStream 36
 DataOutputStream 36
 DriverManager 176
 enclosing 100, 102
 FileReader 32

FileWriter 32
FlowLayout 122
GridLayout 122
HashMap 78, 94, 164
HashSet 74, 94
Hashtable 71, 78, 164
inner 100
InputStream 32
InterruptedException 148
IOException 30
JButton 120
JFileChooser 130
JFrame 116, 128
JMenu 128
JMenuBar 128
JMenuItem 128
JScrollBar 124
JTextArea 123
LinkedHashMap 78
LinkedHashSet 74
LinkedList 24
local 100, 104
Matcher **50**, 60, 62
member 100, 102
Object 38
ObjectInputStream 40
ObjectOutputStream 40
OutputStream 32
Pattern **50**, 60
PrintStream 38
PrintWriter 38
Thread 140, 150
TreeMap 78, 84, 86
TreeSet 74, 84, 86
Vector 24, 70, 71, 164
close 184, 188
close() 42
collection 18, 70
Collection interface 70
Collections class 88, 90
Collections Framework 70
commit 210
Comparable interface 84, 86
component 118, 122, 124, 126
ConcurrentHashMap class 166
Connection 188

Connection interface 182, 188
container 118, 122, 124
createStatement 182
cursor 178, 188

D

Data Manipulation Language (DML) 186
Database Management Systems (DBMS) 176, 178
DataInputStream class 36
DataOutputStream class 36
dead threads 142
deadlock 160, 162
DELETE 186
downcasting 18, 20
driver 176
DriverManager class 176
DriverManager.getConnection 180

E

enclosing
 class 100, 102
 instance 100, 102
Enumeration interface 71
event 118
 event listener 118, 120
 event source 120
execute 204
executeQuery 184, 198
executeUpdate 202
Executor interface 168
extend 106

F

File 34, 130
file
 browser 130
FileReader class 32
FileWriter class 32
final 106
finally 42
find 50, 62
FlowLayout class 122
frames 116

G

Garbage Collection 188
generics 20
getName 140
getProperty 126
getResultSet 204

getSelectedFile 130
getString 184
getUpdateCount 204
Graphical User Interface (GUI) 114, 116, 118
GridLayout class 122

H

HashMap class 78, 94, 164
HashSet class 74, 94
Hashtable class 71, 78, 164
heavyweight component 114

I

I/O 30
implement 106
inner class 100
input stream 30, 32
InputStream class 32
INSERT 186
instance
 enclosing 100, 102
 initializer 108
interface
 ActionListener 120, 128
 Callable 170
 Collection 70
 Comparable 84, 86
 Connection 182, 188
 Enumeration 71
 Executor 168
 Iterator 70, 71, 72
 LayoutManager 122
 List 22, 24, 70, 88
 Map 70, 71, 76
 ResultSet 184, 188
 Runnable 150, 168, 170
 Serializable 40
 Set 70, 72, 74
 Statement 182, 184, 186, 188
interrupt 148
interrupted 148
InterruptedException class 148
IOException class 30
isInterrupted 148
Iterator interface 70, 71, 72

J

Java 2 API 71
Java 7 188
Java Database Connectivity (JDBC) 176, 178
java.awt 116, 122

java.lang.Runnable 150
 java.lang.Thread 140
 java.sql 176
 java.sql.Types 196
 javax.swing 116
 JButton class 120
 JDBC type 196
 JFileChooser class 130
 JFrame class 116, 128
 JMenu class 128
 JMenuBar class 128
 JMenuItem class 128
 join 146, 148
 JScrollBar class 124
 JTextArea class 123

L

LayoutManager interface 122
 lightweight
 component 114
 process 138
 lightweight process 138
 LinkedHashMap class 78
 LinkedHashSet class 74
 LinkedList class 24
 List interface 22, 24, 70, 88
 load factor 94
 local class 100, 104
 lock 158, 160

M

main thread 138
 Map interface 70, 71, 76
 Matcher class 50, 60, 62
 matches 60
 member class 100, 102
 menubar 128
 metacharacter 52
 method
 execute 204
 executeQuery 198
 executeUpdate 202
 getResultSet 204
 getUpdateCount 204
 prepareStatement 206
 multi-tasking 136
 multi-threading 138, 139

N

new 106
 new threads 142

next 184

O

object
 serialized 40
 Object class 38
 ObjectInputStream class 40
 ObjectOutputStream class 40
 Open Database Connectivity (ODBC) 176
 output stream 30, 32
 OutputStream class 32

P

parameterized
 type 20
 pattern 48
 Pattern class 50, 60
 placeholder 208
 Pluggable Look and Feel (PLAF) 114
 PreparedStatement 206, 208
 prepareStatement 206
 primitive 36
 print 38
 printf 38
 println 38
 PrintStream class 38
 PrintWriter class 38
 private 102
 process 136
 properties 126
 protected 102
 public 102

Q

quantifier 56

R

race condition 156
 reader 32
 Regular Expression (RE) 48
 result set 178, 184
 ResultSet 200
 ResultSet interface 184
 object 184, 188
 ResultSetMetaData 200
 rollback 210
 run 140, 144
 Runnable interface 150, 168, 170
 runnable threads 142

S

- scrollbar 124
- SELECT 184
- Serializable interface 40
- Set interface 70, 72, 74
- setBounds 122
- setLayout 122
- setLocation 122
- setName 140
- setProperty 126
- setSize 122
- showOpenDialog 130
- showSaveDialog 130
- sleep 146, 148
- socket 30
- SortedSet 74
- split 60
- SQL 182
 - DELETE 186
 - INSERT 186
 - query 184
 - SELECT 184
 - statements 182
 - UPDATE 186
- SQLException 194
- stack 156
- start 140
- Statement 198, 202, 204
- Statement interface 184, 186, 188
 - object 182
- static 100, 102
- stream 30
 - byte 32
 - character 32
 - input 30
 - output 30
- Swing 114, 126
- synchronized 158, 160, 162, 166

T

- this 102
- thread 138
 - blocked 142, 146, 148
 - blocks 158
 - name 140
 - queue 144, 145
 - runnable 142, 146
 - state 142
 - terminated 142
- Thread class 140, 150
- top-level

- container 118
 - window 116
- toString 38
- transaction 210
- transaction isolation level 211
- TreeMap class 78, 84, 86
- TreeSet class 74, 84, 86
- try-with-resources 42, 188
- tuning 92, 94
- type
 - parameterized 20

U

- Unicode 32
- Uniform Resource Locator (URL) 181
 - JDBC connection 180
- UPDATE 186
- update
 - count 186

V

- Vector class 24, 70, 71, 164
- Virtual Machine (VM) 138

W

- writer 32

Y

- yield 146

Skill Distillery

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
303-302-5234
www.SkillDistillery.com