Java SE 7 Programming

Student Guide - Volume II

D67238GC20
Edition 2.0
June 2012
D74997

ORACLE®

Authors

Michael Williams

Tom McGinn

Matt Heimer

Technical Contributors and Reviewers

Peter Hall

Marnie Knue

Lee Klement

Steve Watts

Brian Earl

Vasily Strelnikov

Andy Smith

Nancy K.A.N

Chris Lamb

Todd Lowry

Ionut Radu

Joe Darcy

Brian Goetz

Alan Bateman

David Holmes

Editors

Richard Wallis

Daniel Milne

Vijayalakshmi Narasimhan

Graphic Designer

James Hans

Publishers

Syed Imtiaz Ali

Sumesh Koshy

Contents

9

Exceptions and Assertions

Objectives

After completing this lesson, you should be able to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions

ORACLE

Error Handling

Applications will encounter errors while executing. Reliable applications should handle errors as gracefully as possible. Errors:

- Should be the “exception” and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
  - Databases becoming unreachable
  - Hard drives failing

Returning a Failure Result

Some programming languages use the return value of a method to indicate whether or not a method completed successfully. For instance, in the C example `int x = printf("hi");`, a negative value in `x` would indicate a failure. Many of C’s standard library functions return a negative value upon failure. The problem is that the previous example could also be written as `printf("hi");` where the return value is ignored. In Java, you also have the same concern; any return value can be ignored.

When a method you are writing in the Java language fails to execute successfully, consider using the exception-generating and handling features available in the language instead of using return values.

Exception Handling in Java

When using Java libraries that rely on external resources, the compiler will require you to "handle or declare" the exceptions that might occur.

- Handling an exception means you must add in a code block to handle the error.
- Declaring an exception means you declare that a method may fail to execute successfully.

ORACLE

The Handle or Declare Rule

Many libraries that you use will require knowledge of exception handling. They include:

- File IO (NIO: `java.nio`)
- Database access (JDBC: `java.sql`)

Handling an exception means you use a `try-catch` statement to transfer control to an exception-handling block when an exception occurs. Declaring an exception means to add a `throws` clause to a method declaration, indicating that the method may fail to execute in a specific way. To state it another way, handling means it is your problem to deal with and declaring means that it is someone else's problem to deal with.

The `try-catch` Statement

The `try-catch` statement is used to handle exceptions.

```
try {
    System.out.println("About to open a file");
    InputStream in =
        new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

This line is skipped if the previous line failed to open the file.

This line runs only if something went wrong in the `try` block.

The `catch` Clause

When an exception occurs inside of a `try` block, execution will transfer to the attached `catch` block. Any lines inside the `try` block that appear after exception are skipped and will not execute. The `catch` clause should be used to:

- Retry the operation
- Try an alternate operation
- Gracefully exit or return

Avoid having an empty `catch` block. Silently swallowing an exception is a bad practice.

Exception Objects

A `catch` clause is passed a reference to a `java.lang.Exception` object. The `java.lang.Throwable` class is the parent class for `Exception` and it outlines several methods that you may use.

```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}
```

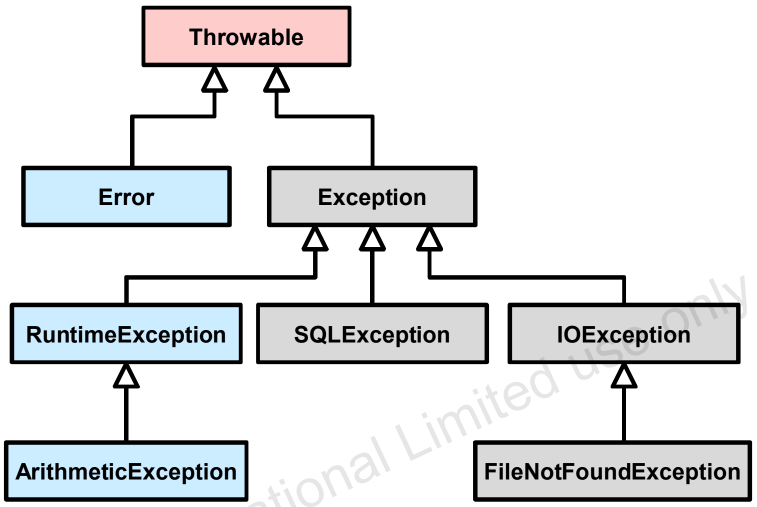
Logging Exceptions

When things go wrong in your application, you will often want to record what happened. Java developers have a choice of several logging libraries including Apache's Log4j and the built-in `java.util` logging framework. While these logging libraries are beyond the scope of this course, you may notice that IDEs such as NetBeans recommend that you should remove any calls to `printStackTrace()`. This is because production-quality applications should use a logging library instead of printing debug messages to the screen.

Exception Categories

The `java.lang.Throwable` class forms the basis of the hierarchy of exception classes. There are two main categories of exceptions:

- Checked exceptions, which must be "handled or declared"
- Unchecked exceptions, which are not typically "handled or declared"



Dealing with Exceptions

When an `Exception` object is generated and passed to a `catch` clause, it will be instantiated from a class that represents the specific type of problem that occurred. These exception-related classes can be divided into two categories: checked and unchecked.

Unchecked Exceptions

`java.lang.RuntimeException` and `java.lang.Error` and their subclasses are categorized as unchecked exceptions. These types of exceptions should not normally occur during the execution of your application. You may use a `try-catch` statement to help discover the source of these exceptions, but when an application is ready for production use, there should be little code remaining that deals with `RuntimeException` and its subclasses. The Error subclasses represent errors that are beyond your ability to correct, such as the JVM running out of memory. Common `RuntimeException`s that you may have to troubleshoot include:

- `ArrayIndexOutOfBoundsException`: Accessing an array element that does not exist
- `NullPointerException`: Using a reference that does not point to an object
- `ArithmeticException`: Dividing by zero

Quiz

A `NullPointerException` must be caught by using a `try-catch` statement.

a. True

b. False

Answer: b

Quiz

Which of the following types are all checked exceptions (`instanceof`)?

a. `Error`

b. `Throwable`

c. `RuntimeException`

d. `Exception`

Answer: b, d

Handling Exceptions

You should always catch the most specific type of exception. Multiple catch blocks can be associated with a single try.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
    in.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
} catch (IOException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
}
```

Order is important. You must catch the most specific exceptions first (that is, child classes before parent classes).

Checked Exceptions

Every class that is a subclass of `Exception` except `RuntimeException` and its subclasses falls into the category of checked exceptions. You must "handle or declare" these exceptions with a `try` or `throws` statement. The HTML documentation for the Java API (Javadoc) will describe which checked exceptions may be generated by a method or constructor and why.

Catching the most specific type of exception enables you to write `catch` blocks that are targeted at handling very specific types of errors. You should avoid catching the base type of `Exception`, because it is difficult to create a general purpose `catch` block that can deal with every possible error.

Note: Exceptions thrown by the Java Persistence API (JPA) extend `RuntimeException` and as such they are categorized as unchecked exceptions. These exceptions may need to be "handled or declared" in production-ready code, even though you are not required to do so by the compiler.

The `finally` Clause

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```

A `finally` clause runs regardless of whether or not an `Exception` was generated.

You always want to close open resources.

ORACLE

Closing Resources

When you open resources such as files or database connections, you should always close them when they are no longer needed. Attempting to close these resources inside the `try` block can be problematic because you can end up skipping the close operation. A `finally` block always runs regardless of whether or not an error occurred during the execution of the `try` block. If control jumps to a `catch` block, the `finally` block will execute after the `catch` block.

Sometimes the operation that you want to perform in your `finally` block may itself cause an `Exception` to be generated. In that case, you may be required to nest a `try-catch` inside of a `finally` block. You may also nest a `try-catch` inside of `try` and `catch` blocks.

The `try`-with-resources Statement

Java SE 7 provides a new `try`-with-resources statement that will autoclose resources.

```
System.out.println("About to open a file");
try (InputStream in =
       new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Closeable Resources

The `try-with-resources` statement can eliminate the need for a lengthy `finally` block. Resources opened using the `try`-with-resources statement are always closed. Any class that implements the `java.lang.AutoCloseable` can be used as a resource. If a resource should be autoclosed, its reference must be declared within the `try` statement's parentheses.

Multiple resources can be opened if they are separated by semicolons. If you open multiple resources, they will be closed in the opposite order in which you opened them.

Suppressed Exceptions

If an exception occurs in the `try` block of a `try`-with-resources statement *and* an exception occurs while closing the resources, the resulting exceptions will be suppressed.

```
} catch(Exception e) {
    System.out.println(e.getMessage());
    for(Throwable t : e.getSuppressed()) {
        System.out.println(t.getMessage());
    }
}
```

ORACLE

Resource Exceptions

If an exception occurs while creating the `AutoCloseable` resource, control will immediately jump to a `catch` block.

If an exception occurs in the body of the `try` block, all resources will be closed *before* the `catch` block runs. If an exception is generated while closing the resources, it will be suppressed.

If the `try` block executes with no exceptions, but an exception is generated during the closing of a resource, control will jump to a `catch` block.

The `AutoCloseable` Interface

Resource in a `try`-with-resources statement must implement either:

- `java.lang.AutoCloseable`
  - New in JDK 7
  - May throw an `Exception`
- `java.io.Closeable`
  - Refactored in JDK7 to extend `AutoCloseable`
  - May throw an `IOException`

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

AutoCloseable vs. Closeable

The Java API documentation has the following to say about `AutoCloseable`: "Note that unlike the close method of `Closeable`, this close method is *not* required to be idempotent. In other words, calling this close method more than once may have some visible side effect, unlike `Closeable.close`, which is required to have no effect if called more than once. However, implementers of this interface are strongly encouraged to make their close methods idempotent."

Catching Multiple Exceptions

Java SE 7 provides a new multi-`catch` clause.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
     cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Multiple exception types are separated with a vertical bar.

The Benefits of multi-`catch`

Sometimes you want to perform the same action regardless of the exception being generated. The new multi-`catch` clause reduces the amount of code you must write, by eliminating the need for multiple `catch` clauses with the same behaviors.

Another benefit of the multi-`catch` clause is that it makes it less likely that you will attempt to catch a generic exception. Catching `Exception` prevents you from noticing other types of exceptions that might be generated by code that you add later to a `try` block.

The type alternatives that are separated with vertical bars cannot have an inheritance relationship. You may not list both a `FileNotFoundException` and an `IOException` in a multi-`catch` clause.

File I/O and object serialization are covered in the lesson titled “Java I/O Fundamentals.”

Declaring Exceptions

You may declare that a method throws an exception instead of handling it.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Notice the lack of catch clauses. The try-with-resources statement is being used only to close resources.

Using the throws clause, a method may declare that it throws one or more exceptions during execution. If an exception is generated while executing the method, the method will stop executing and the exception will be thrown to the caller. Overridden methods may declare the same exceptions, fewer exceptions, or more specific exceptions, but not additional or more generic exceptions. A method may declare multiple exceptions with a comma-separated list.

```
public static int readByteFromFile() throws FileNotFoundException,
IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Technically you do not need to declare FileNotFoundException, because it is a subclass of IOException, but it is a good practice to do so.

Handling Declared Exceptions

The exceptions that methods may throw must still be handled. Declaring an exception just makes it someone else's job to handle them.

```
public static void main(String[] args) {
   try {
        int data = readByteFromFile();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Method that declared an exception

Handling Exceptions

Your application should always handle its exceptions. Adding a `throws` clause to a method just delays the handling of the exception. In fact, an exception can be thrown repeatedly up the call stack. A standard Java SE application must handle any exceptions before they are thrown out of the `main` method; otherwise, you risk having your program terminate abnormally. It is possible to declare that `main` throws an exception, but unless you are designing programs to terminate in a nongraceful fashion, you should avoid doing so.

Throwing Exceptions

You can rethrow an exception that has already been caught. Note that there is both a `throws` clause and a `throw` statement.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    } catch (IOException e) {
        e.printStackTrace();
        throw e;
    }
}
```

Precise Rethrow

Java SE 7 supports rethrowing the precise exception type. The following example would not compile with Java SE 6 because the `catch` clause receives an `Exception`, but the method throws an `IOException`. For more about the new precise rethrow feature, see http://download.oracle.com/javase/7/docs/technotes/guides/language/catch-multiple.html#rethrow.

```
public static int readByteFromFile() throws IOException {
    try {
        InputStream in = new FileInputStream("a.txt");
        System.out.println("File open");
        return in.read();
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
}
```

Custom Exceptions

You can create custom exception classes by extending `Exception` or one of its subclasses.

```
public class DAOException extends Exception {

    public DAOException() {
        super();
    }

    public DAOException(String message) {
        super(message);
    }
}
```

Custom exceptions are never thrown by the standard Java class libraries. To take advantage of a custom exception class, you must throw it yourself. For example:

```
throw new DAOException();
```

A custom exception class may override methods or add new functionality. The rules of inheritance are the same, even though the parent class type is an exception.

Because exceptions capture information about a problem that has occurred, you may need to add fields and methods depending on the type of information that needs to be captured. If a string can capture all the necessary information, you can use the `getMessage()` method that all `Exception` classes inherit from `Throwable`. Any `Exception` constructor that receives a string will store it to be returned by `getMessage()`.

Quiz

Which keyword would you use to add a clause to a method stating that the method might produce an exception?

a. `throw`

b. `thrown`

c. `throws`

d. `assert`

Answer: c

Wrapper Exceptions

To hide the type of exception being generated without simply swallowing the exception, use a wrapper exception.

```
public class DAOException extends Exception {
    public DAOException(Throwable cause) {
        super(cause);
    }

    public DAOException(String message, Throwable cause)
   {
        super(message, cause);
    }
}
```

ORACLE

Getting the Cause

The `Throwable` class contains a `getCause()` method that can be used to retrieve a wrapped exception.

```
try {
    //…
} catch (DAOException e) {
    Throwable t = e.getCause();
}
```

Revisiting the DAO Pattern

The DAO pattern uses abstraction (an interface) to allow implementation substitution. A file or database DAO must deal with exceptions. A DAO implementation may use a wrapper exception to preserve the abstraction and avoid swallowing exceptions.

```
public Employee findById(int id) throws DAOException {
    try {
        return employeeArray[id];
    } catch (ArrayIndexOutOfBoundsException e) {
     throw new DAOException("Error finding employee in DAO", e);
    }
}
```

DAO Exceptions

A file-based DAO must deal with `IOExceptions` and a JDBC-based DAO must deal with `SQLExceptions`. If these types of exceptions were thrown by a DAO, any clients would be tied to an implementation instead of an abstraction. By modifying the DAO interface and implementing classes to throw a wrapper exception (`DAOException`), you can preserve the abstraction and let clients know when a DAO implementation encounters a problem.

Assertions

- Use assertions to document and verify the assumptions and internal logic of a single method:
  - Internal invariants
  - Control flow invariants
  - Postconditions and class invariants
- Inappropriate uses of assertions

  Assertions can be disabled at run time; therefore:
  - Do not use assertions to check the parameters of a public method.
  - Do not use methods that can cause side effects in the assertion check.

Why Use Assertions

You can use assertions to add code to your applications that ensures that the application is executing as expected. Using assertions, you test for various conditions failing; if they do, you terminate the application and display debugging-related information. Assertions should not be used if the checks to be performed should always be executed because assertion checking may be disabled.

Assertion Syntax

- Syntax of an assertion is:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

- If `<boolean_expression>` evaluates `false`, then an `AssertionError` is thrown.
- The second argument is converted to a string and used as descriptive text in the `AssertionError` message.

The `assert` Statement

Assertions combine the exception-handling mechanism of Java with conditionally executed code. The following is a pseudo-code example of the behavior of assertions:

```
if (AssertionsAreEnabled) {
      if (condition == false) throw new AssertionError();
}
```

`AssertionError` is a subclass of `Error` and, therefore, falls in the category of unchecked exceptions.

Internal Invariants

- The problem is:

```
if (x > 0) {
  // do this
} else {
  // do that
}
```

- The solution is:

```
if (x > 0) {
  // do this
} else {
  assert ( x == 0 );
  // do that, unless x is negative
}
```

ORACLE

Control Flow Invariants

Example:

```
switch (suit) {
    case Suit.CLUBS: // ...
      break;
    case Suit.DIAMONDS: // ...
      break;
    case Suit.HEARTS: // ...
      break;
    case Suit.SPADES: // ...
      break;
    default: assert false : "Unknown playing card suit";
        break;
  }
```

Postconditions and Class Invariants

Example:

```
public Object pop() {
  int size = this.getElementCount();
  if (size == 0) {
    throw new RuntimeException("Attempt to pop from empty stack");
  }

  Object result = /* code to retrieve the popped element */ ;

  // test the postcondition
    assert (this.getElementCount() == size - 1);

  return result;
  }
```

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as if the check were never there.
- Assertion checks are disabled by default. Enable assertions with either of the following commands:

```
java -enableassertions MyProgram
```

```
java -ea MyProgram
```

- Assertion checking can be controlled on class, package, and package hierarchy basis. See: http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html

Quiz

Assertions should be used to perform user-input validation?

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants using assertions

Practice 9-1 Overview: Catching Exceptions

This practice covers the following topics:

- Adding `try-catch` statements to a class
- Handling exceptions

In this practice, you write code to deal with both checked and unchecked exceptions.

Practice 9-2 Overview: Extending `Exception`

This practice covers the following topics:

- Adding `try-catch` statements to a class
- Handling exceptions
- Extending the `Exception` class
- Creating a custom auto-closeable resource
- Using a `try`-with-resources statement
- Throwing exceptions using `throw` and `throws`

In this practice, you update a DAO pattern implementation to use a custom wrapper exception.

10

Java I/O Fundamentals

Objectives

After completing this lesson, you should be able to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use streams to read and write files
- Read and write objects by using serialization

Java I/O Basics

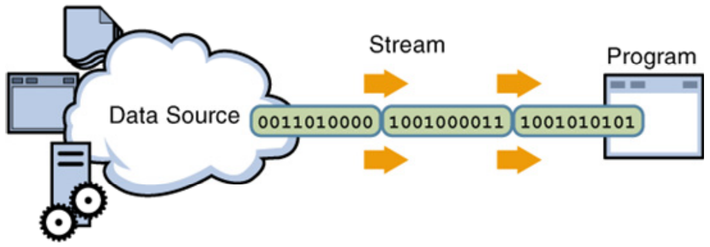
The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

- Java defines an I/O channel as a stream.
- An I/O Stream represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.

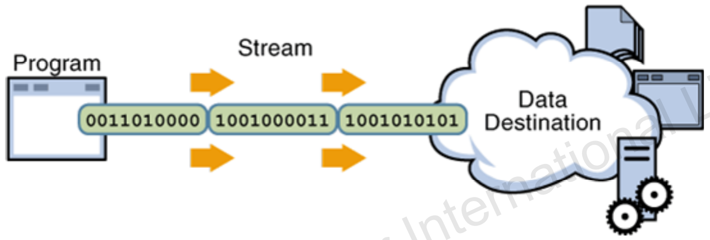
Some streams simply pass on data; others manipulate and transform the data in useful ways.

I/O Streams

- A program uses an input stream to read data from a source, one item at a time.



- A program uses an output stream to write data to a destination (sink), one item at time.



No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data.

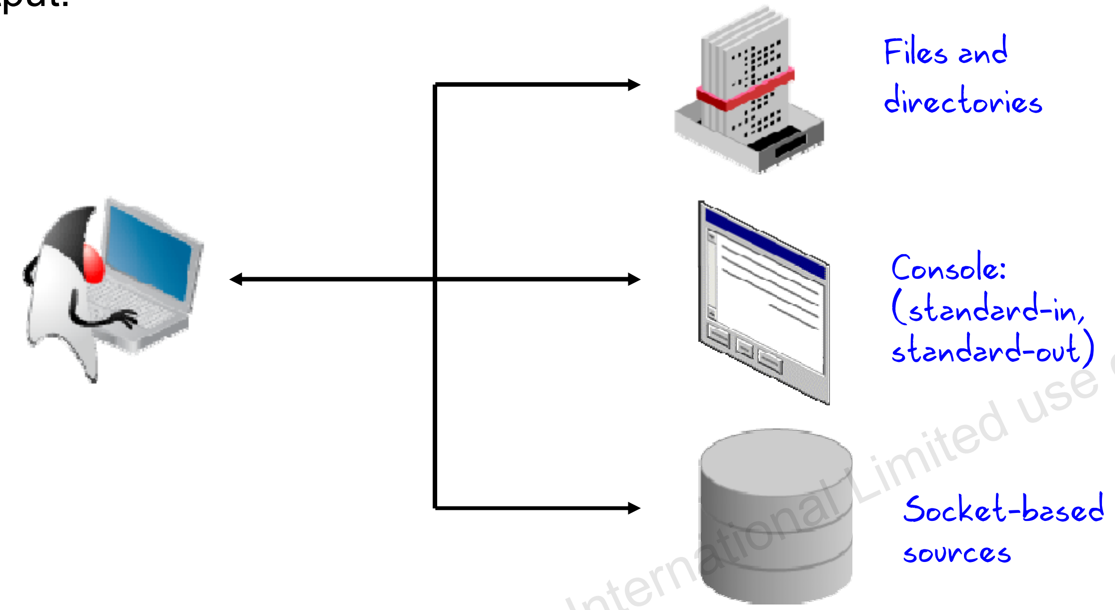
A stream is a flow of data. A stream can come from a source or can be generated to a sink.

- A source stream initiates the flow of data, also called an input stream.
- A sink stream terminates the flow of data, also called an output stream.

Sources and sinks are both node streams. Types of node streams are files, memory, and pipes between threads or processes.

I/O Application

Typically, there are three ways a developer may use input and output:



An application developer typically uses I/O streams to read and write files, to read and write information to and from some output device, such as the keyboard (standard in) and the console (standard out). Finally, an application may need to use a socket to communicate with another application on a remote system.

Data Within Streams

- Java technology supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

Stream	Byte Streams	Character Streams
Source streams	`InputStream`	`Reader`
Sink streams	`OutputStream`	`Writer`

Java technology supports two types of data in streams: raw bytes and Unicode characters. Typically, the term *stream* refers to byte streams and the terms *reader* and *writer* refer to character streams.

More specifically, byte input streams are implemented by subclasses of the `InputStream` class and byte output streams are implemented by subclasses of the `OutputStream` class. Character input streams are implemented by subclasses of the `Reader` class and character output streams are implemented by subclasses of the `Writer` class.

Byte streams are best applied to reading and writing or raw bytes (such as image files, audio files, and objects). Specific subclasses provide methods to provide specific support for each of these stream types.

Character streams are designed for reading characters (such as in files and other character-based streams).

Byte Stream `InputStream` Methods

- The three basic `read` methods are:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close();              // Close an open stream
int available();           // Number of bytes available
long skip(long n);         // Discard n bytes from stream

boolean markSupported();    //
void mark(int readlimit);   // Push-back operations
void reset();               //
```

`InputStream` Methods

The `read()` method returns an `int`, which contains either a byte read from the stream, or a `-1`, which indicates the end-of-file condition. The other two read methods read the stream into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array that needs to be filled.

Note: For efficiency, always read data in the largest practical block, or use buffered streams.

When you have finished with a stream, close it. If you have a stack of streams, use filter streams to close the stream at the top of the stack. This operation also closes the lower streams.

Note: In Java SE 7, `InputStream` implements `AutoCloseable`, which means that if you use an `InputStream` (or one of its subclasses) in a `try`-with-resources block, the stream is automatically closed at the end of the try.

The `available` method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

The `skip` method discards the specified number of bytes from the stream.

The `markSupported()`, `mark()`, and `reset()` methods perform push-back operations on a stream, if supported by that stream. The `markSupported()` method returns true if the `mark()` and `reset()` methods are operational for that particular stream. The `mark(int)` method indicates that the current point in the stream should be noted and a buffer big enough for at least the specified argument number of bytes should be allocated. The parameter of the `mark(int)` method specifies the number of bytes that can be re-read by calling `reset()`. After subsequent `read()` operations, calling the `reset()` method returns the input stream to the point you marked. If you read past the marked buffer, `reset()` has no meaning.

Byte Stream `OutputStream` Methods

- The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close(); // Automatically closed in try-with-resources
void flush(); // Force a write to the stream
```

`OutputStream` Methods

As with input, always try to write data in the largest practical block.

Byte Stream Example

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteStreamCopyTest {
    public static void main(String[] args) {
        byte[] b = new byte[128]; int bLen = b.length;
        // Example use of InputStream methods
        try (FileInputStream fis = new FileInputStream (args[0]);
             FileOutputStream fos = new FileOutputStream (args[1])) {
             System.out.println ("Bytes available: " + fis.available());
            int count = 0; int read = 0;
            while ((read = fis.read(b)) != -1) {
                if (read < bLen) fos.write(b, 0, read);
                else fos.write(b);
                count += read;
             }
             System.out.println ("Wrote: " + count);
         } catch (FileNotFoundException f) {
             System.out.println ("File not found: " + f);
         } catch (IOException e) {
             System.out.println ("IOException: " + e);
         }
     }
}
```

Note that you must keep track of how many bytes are read into the byte array each time.

This example copies one file to another by using a byte array. Note that `FileInputStream` and `FileOutputStream` are meant for streams of raw bytes like image files.

Note: The `available()` method, according to the Javadocs, reports "an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking."

Character Stream `Reader` Methods

- The three basic `read` methods are:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

`Reader` Methods

The first method returns an `int`, which contains either a Unicode character read from the stream, or a `-1`, which indicates the end-of-file condition. The other two methods read into a character array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array that needs to be filled.

Character Stream `Writer` Methods

- The basic `write` methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

`Writer` Methods

The methods are analogous to the `OutputStream` methods.

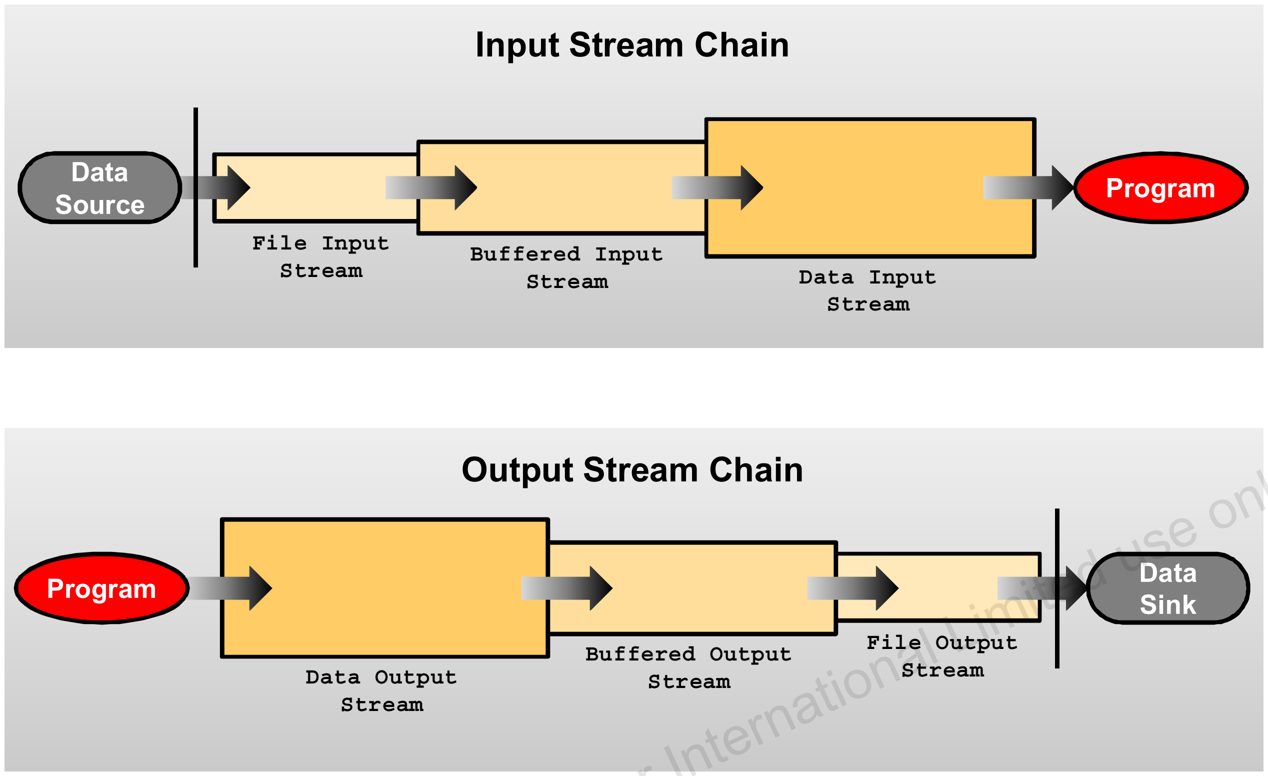
Character Stream Example

```
import java.io.FileReader; import java.io.FileWriter;
import java.io.IOException; import java.io.FileNotFoundException;

public class CharStreamCopyTest {
    public static void main(String[] args) {
        char[] c = new char[128]; int cLen = c.length;
        // Example use of InputStream methods
        try (FileReader fr = new FileReader(args[0]);
             FileWriter fw = new FileWriter(args[1])) {
             int count = 0;
             int read = 0;
             while ((read = fr.read(c)) != -1) {
                 if (read < cLen) fw.write(c, 0, read);
                 else fw.write(c);
                 count += read;
             }
             System.out.println("Wrote: " + count + " characters.");
         } catch (FileNotFoundException f) {
             System.out.println("File " + args[0] + " not found.");
         } catch (IOException e) {
             System.out.println("IOException: " + e);
         }
    }
}
```

Now, rather than a byte array, this version uses a character array.

Similar to the byte stream example, this application copies one file to another by using a character array instead of a byte array. `FileReader` and `FileWriter` are classes designed to read and write character streams, such as text files.

I/O Stream Chaining



A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. The first figure in the slide demonstrates an example of input stream; in this case, a file stream is buffered for efficiency and then converted into data (Java primitives) items. The second figure demonstrates an example of output stream; in this case, data is written, then buffered, and finally written to a file.

Chained Streams Example

```
import java.io.BufferedReader; import java.io.BufferedWriter;
import java.io.FileReader; import java.io.FileWriter;
import java.io.FileNotFoundException; import java.io.IOException;

public class BufferedStreamCopyTest {
   public static void main(String[] args) {
       try (BufferedReader bufInput
                = new BufferedReader(new FileReader(args[0]));
            BufferedWriter bufOutput
                 = new BufferedWriter(new FileWriter(args[1]))) {
            String line = "";
            while ((line = bufInput.readLine()) != null) {
                 bufOutput.write(line);
                 bufOutput.newLine();
            }
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

A `FileReader` chained to a `BufferedFileReader`: This allows you to use a method that reads a String.

The character buffer replaced by a `String`. Note that `readLine()` uses the newline character as a terminator. Therefore, you must add that back to the output file.

Here is the copy application one more time. This version illustrates the use of a `BufferedReader` chained to the `FileReader` that you saw before.

The flow of this program is the same as before. Instead of reading a character buffer, this program reads a line at a time using the line variable to hold the `String` returned by the `readLine` method, which provides greater efficiency. The reason is that each read request made of a `Reader` causes a corresponding read request to be made of the underlying character or byte stream. A `BufferedReader` reads characters from the stream into a buffer (the size of the buffer can be set, but the default value is generally sufficient.)

Processing Streams

Functionality	Character Streams	Byte Streams
Buffering (Strings)	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversion (byte to character)	InputStreamReader OutputStreamWriter	
Object serialization		ObjectInputStream ObjectOutputStream
Data conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

A processing stream performs a conversion on another stream. You choose the stream type that you want based on the functionality that you need for the final stream.

Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output.*
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input.*
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error.*

Console I/O Using System

- **`System.out`** is the "standard" output stream. This stream is already open and ready to accept output data. Typically, this stream corresponds to display output or another output destination specified by the host environment or user.
- **`System.in`** is the "standard" input stream. This stream is already open and ready to supply input data. Typically, this stream corresponds to keyboard input or another input source specified by the host environment or user.
- **`System.err`** is the "standard" error output stream. This stream is already open and ready to accept output data.
  Typically, this stream corresponds to display output or another output destination specified by the host environment or user. By convention, this output stream is used to display error messages or other information that should come to the immediate attention of a user even if the principal output stream, the value of the variable `out`, has been redirected to a file or other destination that is typically not continuously monitored.

java.io.Console

In addition to the `PrintStream` objects, `System` can also access an instance of the `java.io.Console` object:

```
Console cons = System.console();
if (cons != null) {
    String userTyped; String pwdTyped;
    do {
        userTyped = cons.readLine("%s", "User name: ");
        pwdTyped = new String(cons.readPassword("%s", "Password: "));
        if (userTyped.equals("oracle") && pwdTyped.equals("tiger")) {
           userValid = true;
        } else {
          System.out.println("Wrong user name/password. Try again.\n");
        }
    } while (!userValid);
}
```

`readPassword` does not echo the characters typed to the console.

- Note that you should pass the username and password to an authentication process.

The `Console` object represents the character-based console associate with the current JVM. Whether a virtual machine has a console depends on the underlying platform and also upon the manner in which the virtual machine is invoked.

NetBeans, for example, does not have a console. To run the example in the project `SystemConsoleExample`, use the command line.

Note: This example is just to illustrate the methods of the console class. You should ensure that the lifetime of the fields `userTyped` and `pwdTyped` are as short as possible and pass the received credentials to some type of authentication service. See the Java Authentication and Authorization Service (JAAS) API for more information: http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

Writing to Standard Output

- The `println` and `print` methods are part of the `java.io.PrintStream` class.
- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- The `print` and `println` methods are overloaded for most primitive types `(boolean`, `char`, `int`, `long`, `float`, and `double)` and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument.

Print Methods

Note that there is also a formatted print method, `printf`. You saw this method in the lesson titled “String Processing.”

Reading from Standard Input

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 public class KeyboardInput {
5     public static void main(String[] args) {
6     try (BufferedReader in =
7          new BufferedReader (new InputStreamReader (System.in))) {
8        String s = "";
9        // Read each input line and echo it to the screen.
10        while (s != null) {
11            System.out.print("Type xyz to exit: ");
12            s = in.readLine();
13            if (s != null) s = s.trim();
14            System.out.println("Read: " + s);
15            if (s.equals ("xyz")) System.exit(0);
16        }
17    } catch (IOException e) {
18        System.out.println ("Exception: " + e);
19    }
20}
```

Chain a buffered reader to an input stream that takes the console input.

The `try`-with-resources statement on line 6 opens `BufferedReader`, which is chained to an `InputStreamReader`, which is chained to the static standard console input `System.in`.

If the string read is equal to “`xyz`,” then the program exits. The purpose of the `trim()` method on the String returned by `in.readLine` is to remove any whitespace characters.

Note: A null string is returned if an end of stream is reached (the result of a user pressing Ctrl-C in Windows, for example) thus the test for null on line 13.

Channel I/O

Introduced in JDK 1.4, a channel reads bytes and characters in blocks, rather than one byte or character at a time.

```
import java.io.FileInputStream; import java.io.FileOutputStream;
import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
import java.io.FileNotFoundException; import java.io.IOException;

public class ByteChannelCopyTest {
    public static void main(String[] args) {
        try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
             FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
            ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
             fcIn.read(buff);
            buff.position(0);
            fcOut.write(buff);
        } catch (FileNotFoundException f) {
            System.out.println("File not found: " + f);
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

Create a buffer sized the same as the file size, and then read and write the file in a single operation.

In this example, a file can be read in its entirety into a buffer, and then written out in a single operation.

Channel I/O was introduced in the `java.nio` package in JDK 1.4.

Practice 10-1 Overview: Writing a Simple Console I/O Application

This practice covers the following topics:

- Writing a main class that accepts a file name as an argument.
- Using `System` console I/O to read a search string.
- Using stream chaining to use the appropriate method to search for the string in the file and report the number of occurrences.
- Continuing to read from the console until an exit sequence is entered.

In this practice, you will write the code necessary to read a file name as an application argument, and use the `System` console to read from standard input until a termination character is typed in.

Persistence

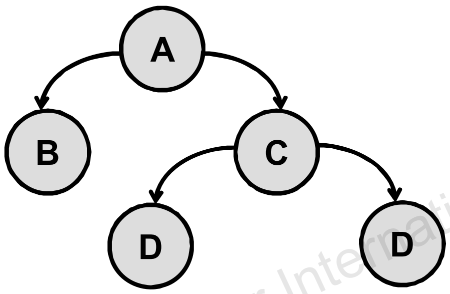
Saving data to some type of permanent storage is called persistence. An object that is persistent-capable, can be stored on disk (or any other storage device), or sent to another machine to be stored there.

- A nonpersisted object exists only as long as the Java Virtual Machine is running.
- Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
- To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

The `java.io.Serializable` interface defines no methods, and serves only as a marker to indicate that the class should be considered for serialization.

Serialization and Object Graphs

- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object's fields constitutes the *object graph*.

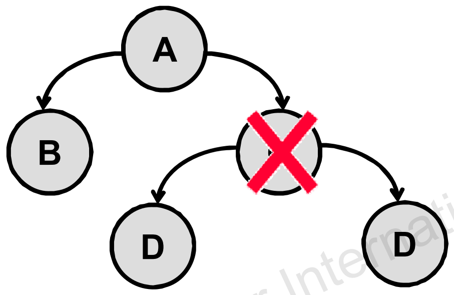


Object Graphs

Serialization traverses the object graph and writes that data to the file (or other output stream) for each node of the graph.

Transient Fields and Objects

- Some object classes are not serializable because they represent transient operating system–specific information.
- If the object graph contains a nonserializable reference, a `NotSerializableException` is thrown and the serialization operation fails.
- Fields that should not be serialized or that do not need to be serialized can be marked with the keyword `transient`.



Transient

If a field containing an object reference is encountered that is not marked as serializable (implement `java.io.Serializable`), a `NotSerializableException` is thrown and the entire serialization operation fails. To serialize a graph containing fields that reference objects that are not serializable, those fields must be marked using the keyword `transient`.

Transient: Example

```
public class Portfolio implements Serializable {
    public transient FileInputStream inputFile;
    public static int BASE = 100;
    private transient int totalValue = 10;
    protected Stock[] stocks;
}
```

`static` fields are not serialized.

Serialization will include all of the members of the `stocks` array.

- The field access modifier has no effect on the data field being serialized.
- The values stored in static fields are not serialized.
- When the object is deserialized, the values of static fields are set to the values declared in the class. The value of non-static transient fields is set to the default value for the type.

When an object is deserialized, the values of static transient fields are set to the values defined in the class declaration. The values of non-static fields are set to the default value of their type. So in the example shown in the slide, the value of BASE will be 100, per the class declaration. The value of non-static transient fields, `inputFile` and `totalValue`, are set to their default values, `null` and `0`, respectively.

Serial Version UID

- During serialization, a version number, serialVersionUID, is used to associate the serialized output with the class used in the serialization process.
- Upon deserialization, the serialVersionUID is checked to verify that the classes loaded are compatible with the object being deserialized.
- If the receiver of a serialized object has loaded classes for that object with different serialVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own serialVersionUID by explicitly declaring a field named `serialVersionUID` as a static final and of type long:

```
private static long serialVersionUID = 42L;
```

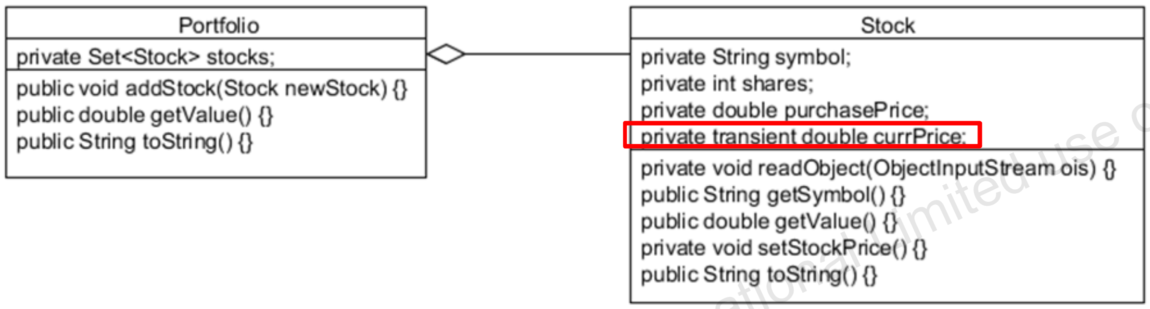
Note: The documentation for `java.io.Serializable` states the following:

If a serializable class does not explicitly declare a serialVersionUID, then the serialization run time will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members. Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes.

Serialization Example

In this example, a Portfolio is made up of a set of Stocks.

- During serialization, the current price is not serialized, and is therefore marked `transient`.
- However, we do want the current value of the stock to be set to the current market price upon deserialization.


Portfolio
private Set<Stock> stocks;
public void addStock(Stock newStock) {}
public double getValue() {}
public String toString() {}
Stock
private String symbol;
private int shares;
private double purchasePrice;
private transient double currPrice;
private void readObject(ObjectInputStream ois) {}
public String getSymbol() {}
public double getValue() {}
private void setStockPrice() {}
public String toString() {}

Writing and Reading an Object Stream

```
1 public static void main(String[] args) {
2     Stock s1 = new Stock("ORCL", 100, 32.50);
3     Stock s2 = new Stock("APPL", 100, 245);
4     Stock s3 = new Stock("GOGL", 100, 54.67);
5     Portfolio p = new   Portfolio(s1, s2, s3);
6     try (FileOutputStream fos = new FileOutputStream(args[0]);
7          ObjectOutputStream out = new ObjectOutputStream(fos)) {
8         out.writeObject(p);
9     } catch (IOException i) {
10         System.out.println("Exception writing out Portfolio: " + i);
11     }
12     try (FileInputStream fis = new FileInputStream(args[0]);
13          ObjectInputStream in = new ObjectInputStream(fis)) {
14         Portfolio newP = (Portfolio)in.readObject();
15     } catch (ClassNotFoundException | IOException i) {
16         System.out.println("Exception reading in Portfolio: " + i);
17 }
```

`Portfolio` is the root object.

The `writeObject` method writes the object graph of `p` to the file stream.

The `readObject` method restores the object from the file stream.

The `SerializeStock` class.

- **Line 6 – 8:** A `FileOutputStream` is chained to an `ObjectOutputStream`. This allows the raw bytes generated by the `ObjectOutputStream` to be written to a file through the `writeObject` method. This method walks the object's graph and writes the data contained in the non-transient and non-static fields as raw bytes.
- **Line 12 – 14:** To restore an object from a file, a `FileInputStream` is chained to an `ObjectInputStream`. The raw bytes read by the `readObject` method restore an `Object` containing the non-static and non-transient data fields. This `Object` must be cast to expected type.

Serialization Methods

An object being serialized (and deserialized) can control the serialization of its own fields.

```
public class MyClass implements Serializable {
    // Fields
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        // Write/save additional fields
        oos.writeObject(new java.util.Date());
    }
}
```

`defaultWriteObject` called to perform the serialization of this classes fields.

- For example, in this class the current time is written into the object graph.
- During deserialization a similar method is invoked:

```
private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException {}
```

The `writeObject` method is invoked on the object being serialized. If the object does not contain this method, the `defaultWriteObject` method is invoked instead.

- This method must also be called once and only once from the object's `writeObject` method.

During deserialization, the `readObject` method is invoked on the object being deserialized (if present in the class file of the object). The signature of the method is important.

```
private void readObject(ObjectInputStream ois) throws
                              ClassNotFoundException, IOException {
    ois.defaultReadObject();
    // Print the date this object was serialized
    System.out.println ("Restored from date: " +
                                (java.util.Date)ois.readObject()));
}
```

`readObject` Example

```java
public class Stock implements Serializable {
    private static final long serialVersionUID = 100L;
    private String symbol;
    private int shares;
    private double purchasePrice;
    private transient double currPrice;

    public Stock(String symbol, int shares, double purchasePrice) {
        this.symbol = symbol;
         this.shares = shares;
         this.purchasePrice = purchasePrice;
         setStockPrice();
     }

     // This method is called post-serialization
     private void readObject(ObjectInputStream ois)
                        throws IOException, ClassNotFoundException {
         ois.defaultReadObject();
         // perform other initialization
         setStockPrice();
     }
}
```

Stock `currPrice` is set by the `setStockPrice` method during creation of the Stock object, but the constructor is not called during deserialization.

Stock `currPrice` is set after the other fields are deserialized

In the Stock class, the `readObject` method is provided, to ensure that the stock's `currPrice` is set (by the `setStockPrice` method) after deserialization of the Stock object.

Note: The signature of the `readObject` method is critical for this method to be called during deserialization.

Summary

In this lesson, you should have learned how to:

- Describe the basics of input and output in Java
- Read data from and write data to the console
- Use streams to read and write files
- Write and read objects by using serialization

Quiz

The purpose of chaining streams together is to:

a. Allow the streams to add functionality
b. Change the direction of the stream
c. Modify the access of the stream
d. Meet the requirements of JDK 7

Answer: a

Chaining one stream to another allows you to add functionality to the stream (for example, converting data from bytes to characters, from characters to a buffered character stream, and so on).

Quiz

To prevent the serialization of operating system–specific fields, you should mark the field:

a. `private`

b. `static`

c. `transient`

d. `final`

Answer: c

a: Access modifiers have no effect on serialization of a field.

b: Although static fields are not serialized, this is not recommended, because marking a field `static` also changes the field's access.

d: `final` has no effect on the serialization of the field and also changes the meaning of the field, making it immutable.

Quiz

Given the following fragments:

```
public MyClass implements Serializable {
    private String name;
    private static int id = 10;
    private transient String keyword;
    public MyClass(String name, String keyword) {
        this.name = name; this.keyword = keyword;
    }
}
```

```
MyClass mc = new MyClass ("Zim", "xyzzy");
```

Assuming no other changes to the data, what is the value of `name` and `keyword` fields after deserialization of the `mc` object instance?

a. `Zim, ""`
b. `Zim, null`
c. `Zim, xyzzy`
d. `"", null`

Answer: b

The field `keyword` is marked transient, and thus will not be serialized. Upon deserialization, the value of the `keyword` is set to its default value of a `String`: `null`.

Quiz

Given the following fragment:

```
1 public class MyClass implements Serializable {
2     private transient String keyword;
3     public void readObject(ObjectInputStream ois)
4                            throws IOException, ClassNotFoundException {
5         ois.defaultReadObject();
6         String this.keyword = (String)ois.readObject();
7     }
8 }
```

What is required to properly deserialize an instance of `MyClass` from the stream containing this object?

a. Make the field `keyword static`
b. Change the field access modifier to `public`
c. Make the `readObject` method `private` (line 3)
d. Use `readString` instead of `readObject` (line 6)

ORACLE

Answer: c

The signature of the method must be:

```
private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException
```

in order to be invoked during serialization.

Practice 10-2 Overview: Serializing and Deserializing a `ShoppingCart`

This practice covers the following topics:

- Creating an application that serializes a `ShoppingCart` object that is composed of an `ArrayList` of `Item` objects
- Using the `transient` keyword to prevent the serialization of the `ShoppingCart` total. This will allow items to vary their cost.
- Use the `writeObject` method to store today's date on the serialized stream.
- Use the `readObject` method to recalculate the total cost of the cart after deserialization and print the date that the object was serialized.

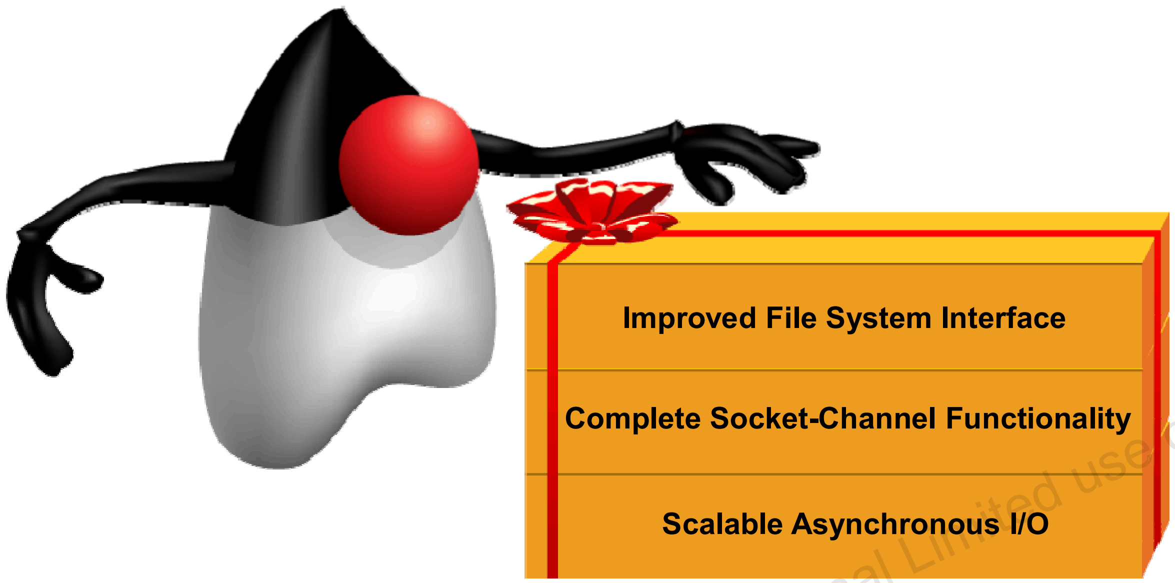
11 Java File I/O (NIO.2)

Objectives

After completing this lesson, you should be able to:

- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use `Files` class methods to read and write files using channel I/O and stream I/O
- Read and change file and directory attributes
- Recursively access a directory tree
- Find a file by using the `PathMatcher` class

New File I/O API (NIO.2)


Improved File System Interface
Complete Socket-Channel Functionality
Scalable Asynchronous I/O

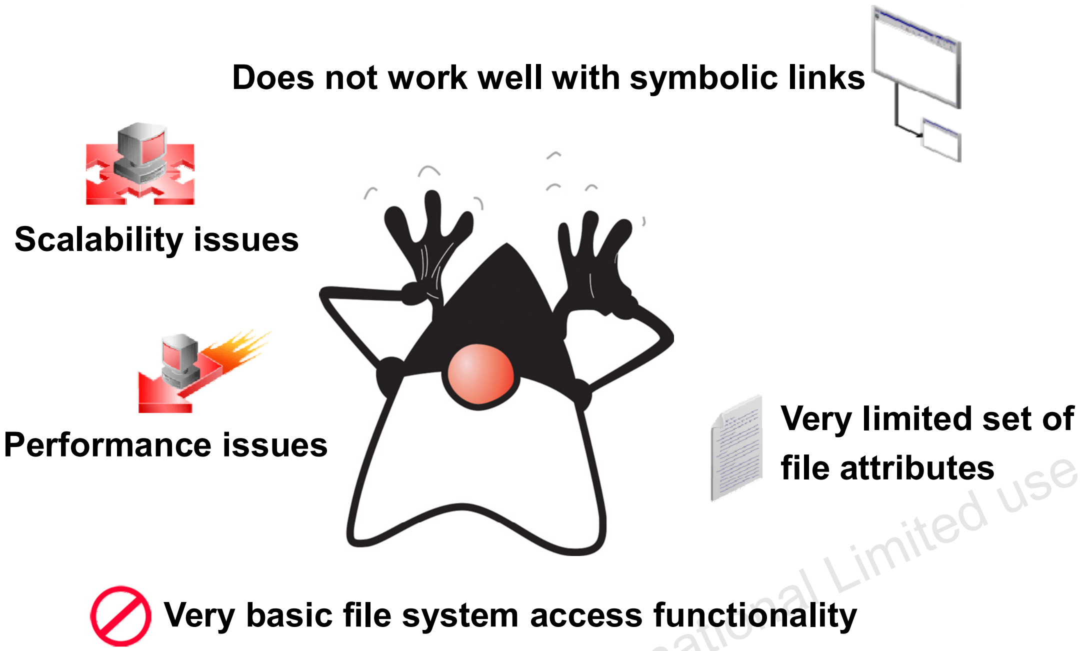
NIO API in JSR 51 established the basis for NIO in Java, focusing on buffers, channels, and charsets. JSR 51 delivered the first piece of the scalable socket I/Os into the platform, providing a non-blocking, multiplexed I/O API, thus allowing the development of highly scalable servers without having to resort to native code.

For many developers, the most significant goal of JSR 203 is to address issues with `java.io.File` by developing a new file system interface.

The new API:

- Works more consistently across platforms
- Makes it easier to write programs that gracefully handle the failure of file system operations
- Provides more efficient access to a larger set of file attributes
- Allows developers of sophisticated applications to take advantage of platform-specific features when absolutely necessary
- Allows support for non-native file systems, to be “plugged in” to the platform

Limitations of `java.io.File`


Does not work well with symbolic links
Scalability issues
Performance issues
Very limited set of file attributes
Very basic file system access functionality

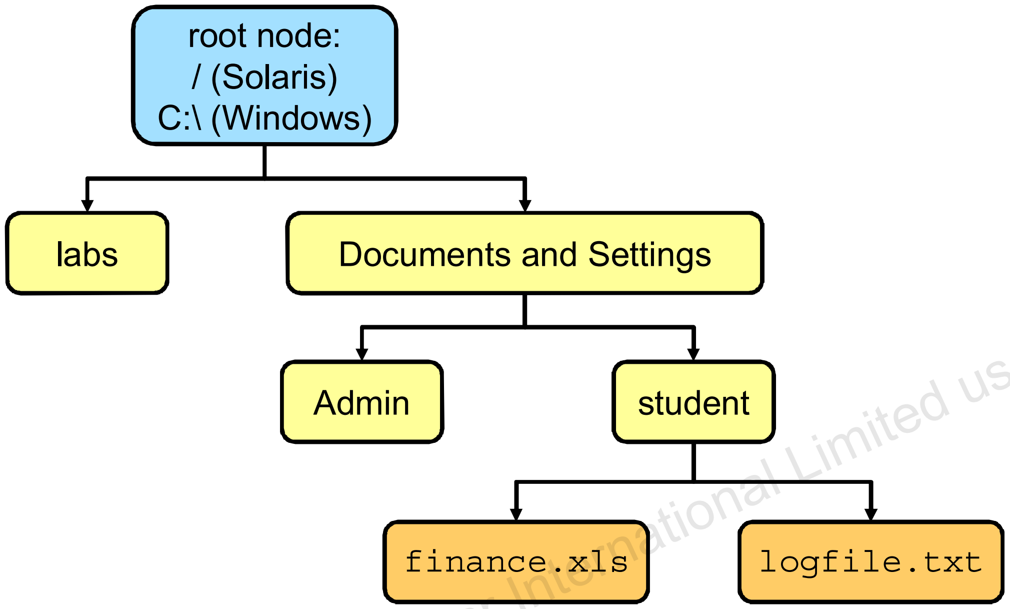
The Java I/O File API (`java.io.File`) presented challenges for developers.

- Many methods did not throw exceptions when they failed, so it was impossible to obtain a useful error message.
- Several operations were missing (file copy, move, and so on).
- The rename method did not work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient—every call for metadata resulted in a system call, which made the operations very inefficient.
- Many of the File methods did not scale. Requesting a large directory listing on a server could result in a hang.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Further, the overall I/O was not written to be extended. Developers had requested the ability to develop their own file system implementations. For example, by keeping a pseudofile system in memory, or by formatting files as zip files.

File Systems, Paths, Files

In NIO.2, both files and directories are represented by a path, which is the relative or absolute location of the file or directory.



File Systems

Prior to the NIO.2 implementation in JDK 7, files were represented by the `java.io.File` class.

In NIO.2, instances of `java.nio.file.Path` objects are used to represent the relative or absolute location of a file or directory.

File systems are hierarchical (tree) structures. File systems can have one or more root directories. For example, typical Windows machines have at least two disk root nodes: C:\ and D:\.

Note that file systems may also have different characteristics for path separators, as shown in the slide.

Relative Path Versus Absolute Path

- A path is either *relative* or *absolute*.
- An absolute path always contains the root element and the complete directory list required to locate the file.
- Example:

```
...
/home/peter/statusReport
...
```

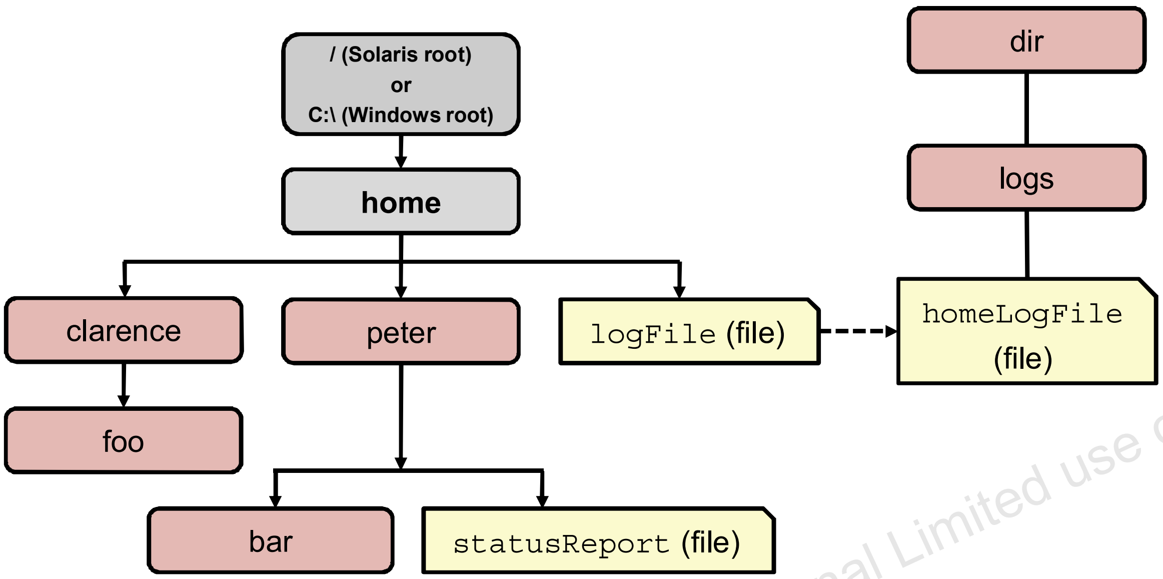
- A relative path must be combined with another path in order to access a file.
- Example:

```
...
clarence/foo
...
```

A path can either be relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, `/home/peter/statusReport` is an absolute path. All the information needed to locate the file is contained in the path string.

A relative path must be combined with another path in order to access a file. For example, `clarence/foo` is a relative path. Without more information, a program cannot reliably locate the `clarence/foo` directory in the file system.

Symbolic Links



File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a "symlink" or a "soft link."

A symbolic link is a special file that serves as a reference to another file. A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

In the slide's diagram, `logFile` appears to the user to be a regular file, but it is actually a symbolic link to `dir/logs/homeLogFile`. `homeLogFile` is the target of the link.

Java NIO.2 Concepts

Prior to JDK 7, the `java.io.File` class was the entry point for all file and directory operations. With NIO.2, there is a new package and classes:

- `java.nio.file.Path`: Locates a file or a directory by using a system-dependent path
- `java.nio.file.Files`: Using a `Path`, performs operations on files and directories
- `java.nio.file.FileSystem`: Provides an interface to a file system and a factory for creating a `Path` and other objects that access a file system
- All the methods that access the file system throw `IOException` or a subclass.

Java NIO.2

A significant difference between NIO.2 and `java.io.File` is the architecture of access to the file system. With the `java.io.File` class, the methods used to manipulate path information are in the same class with methods used to read and write files and directories.

In NIO.2, the two concerns are separated. Paths are created and manipulated using the `Path` interface, while operations on files and directories is the responsibility of the `Files` class, which operates only on `Path` objects.

Finally, unlike `java.io.File`, `Files` class methods that operate directly on the file system, throw an `IOException` (or a subclass). Subclasses provide details on what the cause of the exception was.

`Path` Interface

The `java.nio.file.Path` interface provides the entry point for the NIO.2 file and directory manipulation.

- To obtain a `Path` object, obtain an instance of the default file system, and then invoke the `getPath` method:

```
FileSystem fs = FileSystems.getDefault();
Path p1 = fs.getPath ("D:\\labs\\resources\\myFile.txt");
```

Escaped backward slash

- The `java.nio.file` package also provides a static final helper class called `Paths` to perform `getDefault`:

```
Path p1 = Paths.get ("D:\\labs\\resources\\myFile.txt");
Path p2 = Paths.get ("D:", "labs", "resources", "myFile.txt");
Path p3 = Paths.get ("/temp/foo");
Path p4 = Paths.get (URI.create ("file:///~/somefile");
```

The entry point for the NIO.2 file and directory manipulation is an instance of the `Path` interface. The provider (in this case, the default provider) creates an object that implements this class and handles all the operations to be performed on a file or a directory in a file system.

`Path` objects are immutable—once they are created, they cannot be changed.

Note that if you plan to use the default file system—that is, the file system that the JVM is running on for the `Path` operations—the `Paths` utility is the shorter method. However, if you wanted to perform `Path` operations on a file system other than the default, you would get an instance of the file system that you wanted and use the first approach to build `Path` objects.

Note: The windows file system uses a backward slash by default. However, Windows can accept both backward and forward slashes in applications (except the command shell). Further, backward slashes in Java must be escaped. In order to represent a backward slash in a string, you must type the backward slash twice. Because this looks ugly, and Windows uses both forward and backward slashes, the examples shown in this course will use the forward slash in strings.

`Path` Interface Features

The `Path` interface defines the methods used to locate a file or a directory in a file system. These methods include:

- To access the components of a path:
  - `getFileName`, `getParent`, `getRoot`, `getNameCount`
- To operate on a path:
  - `normalize`, `toUri`, `toAbsolutePath`, `subpath`, `resolve`, `relativize`
- To compare paths:
  - `startsWith`, `endsWith`, `equals`

`Path` Objects Are Like `String` Objects

It is best to think of `Path` objects in the same way you think of `String` objects. `Path` objects can be created from a single text string, or a set of components:

- A *root component*, that identifies the file system hierarchy
- A *name element*, farthest from the root element, that defines the file or directory the path points to
- Additional elements may be present as well, separated by a special character or delimiter that identify directory names that are part of the hierarchy

`Path` objects are immutable. Once created, operations on `Path` objects return new `Path` objects.

`Path`: Example

```
public class PathTest
    public static void main(String[] args) {
        Path p1 = Paths.get(args[0]);
        System.out.format("getFileName: %s%n", p1.getFileName());
        System.out.format("getParent: %s%n", p1.getParent());
        System.out.format("getNameCount: %d%n", p1.getNameCount());
        System.out.format("getRoot: %s%n", p1.getRoot());
        System.out.format("isAbsolute: %b%n", p1.isAbsolute());
        System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
        System.out.format("toURI: %s%n", p1.toUri());
    }
}
```

```
java PathTest D:/Temp/Foo/file1.txt
getFileName: file1.txt
getParent: D:\Temp\Foo
getNameCount: 3
getRoot: D:\
isAbsolute: true
toAbsolutePath: D:\Temp\Foo\file1.txt
toURI: file:///D:/Temp/Foo/file1.txt
```

Run on a Windows machine. Note that except in a `cmd` shell, forward and backward slashes are legal.

Unlike the `java.io.File` class, files and directories are represented by instances of `Path` objects in a *system-dependent* way.

The `Path` interface provides several methods for reporting information about the path:

- `Path getFileName`: The end point of this `Path`, returned as a `Path` object
- `Path getParent`: The parent path or null. Everything in Path up to the file name (file or directory)
- `int getNameCount`: The number of name elements that make up this path
- `Path getRoot`: The root component of this `Path`
- `boolean isAbsolute`: `true` if this path contains a system-dependent root element. **Note:** Because this example is being run on a Windows machine, the *system-dependent* root element contains a drive letter and colon. On a UNIX-based OS, `isAbsolute` returns true for any path that begins with a slash.
- `Path toAbsolutePath`: Returns a path representing the absolute path of this path
- `java.net.URI toUri`: returns an absolute URI.

Note: A `Path` object can be created for any path. The actual file or directory need not exist.

Removing Redundancies from a `Path`

- Many file systems use “`.`” notation to denote the current directory and “`..`” to denote the parent directory.
- The following examples both include redundancies:

```
/home/./clarence/foo
/home/peter/../clarence/foo
```

- The `normalize` method removes any redundant elements, which includes any “`.`” or “`directory/..`” occurrences.
- Example:

```
Path p = Paths.get("/home/peter/../clarence/foo");
Path normalizedPath = p.normalize();
```

```
/home/clarence/foo
```

Many file systems use “`.`” notation to denote the current directory and “`..`” to denote the parent directory. You might have a situation where a `Path` contains redundant directory information. Perhaps a server is configured to save its log files in the “`/dir/logs/.`” directory, and you want to delete the trailing “`/.`” notation from the path.

The `normalize` method removes any redundant elements, which includes any “`.`” or “`directory/..`” occurrences. The slide examples would be normalized to `/home/clarence/foo`.

It is important to note that `normalize` does not check the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if `peter` were a symbolic link, removing `peter/..` might result in a path that no longer locates the intended file.

Creating a Subpath

- A portion of a path can be obtained by creating a subpath using the `subpath` method:

```
Path subpath(int beginIndex, int endIndex);
```

- The element returned by `endIndex` is one less that the `endIndex` value.
- Example:

Temp = 0
foo = 1
bar = 2

```
Path p1 = Paths.get ("D:/Temp/foo/bar");
Path p2 = p1.subpath (1, 3);
```

```
foo\bar
```

Include the element at index 2.

The element name closest to the root has index 0.

The element farthest from the root has index `count-1`.

Note: The returned `Path` object has the name elements that begin at `beginIndex` and extend to the element at index `endIndex-1`.

Joining Two Paths

- The `resolve` method is used to combine two paths.
- Example:

```
Path p1 = Paths.get("/home/clarence/foo");
p1.resolve("bar");     // Returns /home/clarence/foo/bar
```

- Passing an absolute path to the `resolve` method returns the passed-in path.

```
Paths.get("foo").resolve("/home/clarence"); // Returns /home/clarence
```

The `resolve` method is used to combine paths. It accepts a partial path, which is a path that does not include a root element, and that partial path is appended to the original path.

Creating a Path Between Two Paths

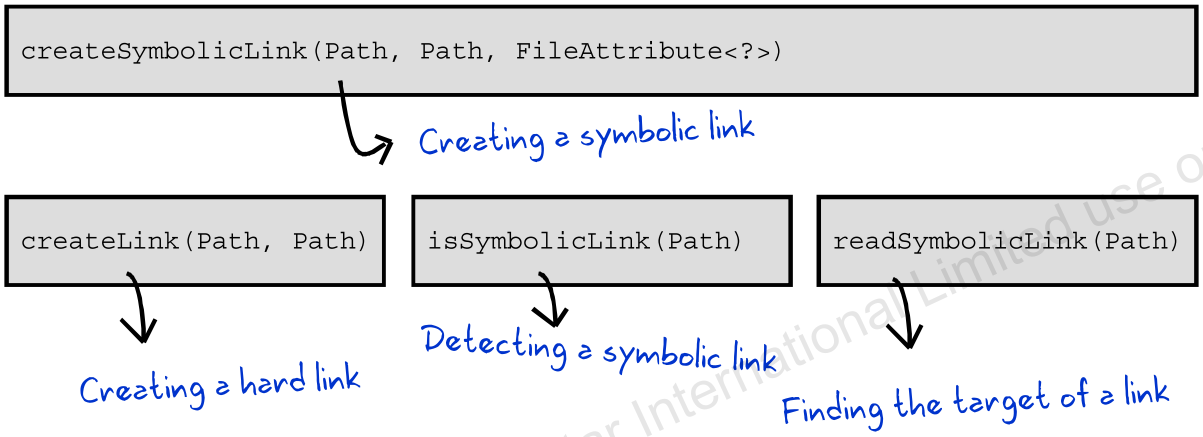
- The `relativize` method enables you to construct a path from one location in the file system to another location.
- The method constructs a path originating from the original path and ending at the location specified by the passed-in path.
- The new path is relative to the original path.
- Example:

```
    Path p1 = Paths.get("peter");
    Path p2 = Paths.get("clarence");

    Path p1Top2 = p1.relativize(p2);   // Result is ../clarence
    Path p2Top1 = p2.relativize(p1);   // Result is ../peter
```

A common requirement when you are writing file I/O code is the capability to construct a path from one location in the file system to another location. You can accomplish this by using the `relativize` method. This method constructs a path originating from the original path and ending at the location specified by the passed-in path. The new path is relative to the original path.

Working with Links

- `Path` interface is "link aware."
- Every `Path` method either:
  - Detects what to do when a symbolic link is encountered, or
  - Provides an option enabling you to configure the behavior when a symbolic link is encountered



The `java.nio.file` package and the `Path` interface in particular are "link aware." Every `Path` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

Some file systems also support hard links. Hard links are more restrictive than symbolic links, as follows:

- The target of the link must exist.
- Hard links are generally not allowed on directories.
- Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
- A hard link looks, and behaves, like a regular file, so they can be hard to find.
- A hard link is, for all intents and purposes, the same entity as the original file. They have the same file permissions, time stamps, and so on. All attributes are identical.

Because of these restrictions, hard links are not used as often as symbolic links, but the `Path` methods work seamlessly with hard links.

Quiz

Given a `Path` object with the following path:

`/export/home/heimer/../williams/./documents`

What `Path` method would remove the redundant elements?

a. `normalize`

b. `relativize`

c. `resolve`

d. `toAbsolutePath`

Answer: a

Quiz

Given the following path:

```
Path p = Paths.get
("/home/export/tom/documents/coursefiles/JDK7");
```

and the statement:

```
Path sub = p.subPath (x, y);
```

What values for x and y will produce a `Path` that contains `documents/coursefiles`?

a. x = 3, y = 4
b. x = 3, y = 5
c. x = 4, y = 5
d. x = 4, y = 6

Answer: b

Quiz

Given this code fragment:

```
Path p1 = Paths.get("D:/temp/foo/");
Path p2 = Paths.get("../bar/documents");
Path p3 = p1.resolve(p2).normalize();
System.out.println(p3);
```

What is the result?

a. Compiler error
b. IOException
c. `D:\temp\foo\documents`
d. `D:\temp\bar\documents`
e. `D:\temp\foo\..\bar\documents`

Answer: d

`File` Operations

- Checking a File or Directory
- Deleting a File or Directory
- Copying a File or Directory
- Moving a File or Directory
- Managing Metadata
- Reading, Writing, and Creating Files
- Random Access Files
- Creating and Reading Directories

The `java.nio.file.Files` class is the primary entry point for operations on `Path` objects.

Static methods in this class read, write, and manipulate files and directories represented by `Path` objects.

The `Files` class is also link aware—methods detect symbolic links in `Path` objects and automatically manage links or provide options for dealing with links.

Checking a File or Directory

A `Path` object represents the concept of a file or a directory location. Before you can access a file or directory, you should first access the file system to determine whether it exists using the following `Files` methods:

- `exists(Path p, LinkOption... option)`
  Tests to see whether a file exists. By default, symbolic links are followed.
- `notExists(Path p, LinkOption... option)`
  Tests to see whether a file does not exist. By default, symbolic links are followed.
- Example:

```
Path p = Paths.get(args[0]);
System.out.format("Path %s exists: %b%n", p,
                  Files.exists(p, LinkOption.NOFOLLOW_LINKS));
```

Optional argument

Recall that `Path` objects may point to files or directories that do not exist. The `exists()` and `notExists()` methods are used to determine whether the `Path` points to a legitimate file or directory, and the particulars of that file or directory.

When testing for the existence of a file, there are three outcomes possible:

- The file is verified to exist.
- The file is verified to not exist.
- The file's status is unknown. This result can occur when the program does not have access to the file.

Note: `!Files.exists(path)` is not equivalent to `Files.notExists(path)`. If both `exists` and `notExists` return false, the existence of the file or directory cannot be determined. For example, in Windows, it is possible to achieve this by requesting the status of an off-line drive, such as a CD-ROM drive.

Checking a File or Directory

To verify that a file can be accessed, the `Files` class provides the following `boolean` methods.

- `isReadable(Path)`
- `isWritable(Path)`
- `isExecutable(Path)`

Note that these tests are not atomic with respect to other file system operations. Therefore, the results of these tests may not be reliable once the methods complete.

- The `isSameFile (Path, Path)` method tests to see whether two paths point to the same file. This is particularly useful in file systems that support symbolic links.

The result of any of these tests is immediately outdated once the operation completes. According to the documentation: "Note that result of this method is immediately outdated. There is no guarantee that a subsequent attempt to open the file for writing will succeed (or even that it will access the same file). Care should be taken when using this method in security-sensitive applications."

Creating Files and Directories

Files and directories can be created using one of the following methods:

```
Files.createFile (Path dir);
Files.createDirectory (Path dir);
```

- The `createDirectories` method can be used to create directories that do not exist, from top to bottom:

```
Files.createDirectories(Paths.get("D:/Temp/foo/bar/example"));
```

The `Files` class also has methods to create temporary files and directories, hard links, and symbolic links.

Deleting a File or Directory

You can delete files, directories, or links. The `Files` class provides two methods:

- `delete(Path)`
- `deleteIfExists(Path)`

```
//...
Files.delete(path);
//...
```

Throws a NoSuchFileException, DirectoryNotEmptyException, or IOException

```
//...
Files.deleteIfExists(Path)
//...
```

No exception thrown

ORACLE

The `delete(Path)` method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist, a `NoSuchFileException` is thrown.

The `deleteIfExists(Path)` method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you do not want to throw an exception just because one thread did so first.

Copying a File or Directory

- You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method.
- When directories are copied, the files inside the directory are not copied.

StandardCopyOption parameters

```
//...
copy(Path, Path, CopyOption...)
//...
```

```
REPLACE_EXISTING
COPY_ATTRIBUTES
NOFOLLOW_LINKS
```

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.copy(source, target, REPLACE_EXISTING, NOFOLLOW_LINKS);
```

You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method. The copy fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the `NOFOLLOW_LINKS` or `REPLACE_EXISTING` option.

The following `StandardCopyOption` and `LinkOption` enums are supported:

- **`REPLACE_EXISTING`:** Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the `FileAlreadyExistsException` exception.
- **`COPY_ATTRIBUTES`:** Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system– and platform-dependent, but last-modified-time is supported across platforms and is copied to the target file.
- **`NOFOLLOW_LINKS`:** Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).

Copying Between a Stream and Path

You may also want to be able to copy (or write) from a `Stream` to file or from a file to a `Stream`. The `Files` class provides two methods to make this easy:

```
copy(InputStream source, Path target, CopyOption... options)
copy(Path source, OutputStream out)
```

- An interesting use of the first method is copying from a web page and saving to a file:

```
 Path path = Paths.get("D:/Temp/oracle.html");
URI u = URI.create("http://www.oracle.com/");
try (InputStream in = u.toURL().openStream()) {
     Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
} catch (final MalformedURLException | IOException e) {
     System.out.println("Exception: " + e);
}
```

The alternative to the stream to path copy is a path to stream method. This method may be used to write a file to a socket, or some other type of stream.

Moving a File or Directory

- You can move a file or directory by using the `move(Path, Path, CopyOption...)` method.
- Moving a directory will not move the contents of the directory.

StandardCopyOption parameters

```
//...
move(Path, Path, CopyOption...)
//...
```

```
REPLACE_EXISTING
ATOMIC_MOVE
```

- Example:

```
import static java.nio.file.StandardCopyOption.*;
//...
Files.move(source, target, REPLACE_EXISTING);
```

Guidelines for moves:

- If the target path is a directory and that directory is empty, the move succeeds if `REPLACE_EXISTING` is set.
- If the target directory does not exist, the move succeeds. Essentially, this is a rename of the directory.
- If the target directory exists and is not empty, a `DirectoryNotEmptyException` is thrown.
- If the source is a file and the target is a directory that exists, and `REPLACE_EXISTING` is set, the move will rename the file to the intended directory name.

To move a directory containing files to another directory, essentially you need to recursively copy the contents of the directory, and then delete the old directory.

You can also perform the move as an atomic file operation using `ATOMIC_MOVE`.

- If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

Listing a Directory's Contents

The `DirectoryStream` class provides a mechanism to iterate over all the entries in a directory.

```
Path dir = Paths.get("D:/Temp");
// DirectoryStream is a stream, so use try-with-resources
// or explicitly close it when finished
try (DirectoryStream<Path> stream =
                     Files.newDirectoryStream(dir, "*.zip")) {
    for (Path file : stream) {
        System.out.println(file.getFileName());
    }
} catch (PatternSyntaxException | DirectoryIteratorException |
         IOException x) {
    System.err.println(x);
}
```

- `DirectoryStream` scales to support very large directories.

The `Files` class provides a method to return a `DirectoryStream`, which can be used to iterate over all the files and directories from any `Path` (root) directory.

`DirectoryIteratorException` is thrown if there is an I/O error while iterating over the entries in the specified directory.

`PatternSyntaxException` is thrown when the pattern provided (second argument of the method) is invalid.

Reading/Writing All Bytes or Lines from a File

- The `readAllBytes` or `readAllLines` method reads entire contents of the file in one pass.
- Example:

```
Path source = ...;
List<String> lines;
Charset cs = Charset.defaultCharset();
lines = Files.readAllLines(file, cs);
```

- Use `write` method(s) to write bytes, or lines, to a file.

```
Path target = ...;
Files.write(target, lines, cs, CREATE, TRUNCATE_EXISTING, WRITE);
```

`StandardOpenOption` enums.

If you have a small file and you would like to read its entire contents in one pass, you can use the `readAllBytes(Path)` or `readAllLines(Path, Charset)` method. These methods take care of most of the work, such as opening and closing the stream, but because they bring the entire file into memory at once, they are not intended for handling large files.

You can use one of the write methods to write bytes, or lines, to a file.

- `write(Path, byte[], OpenOption...)`
- `write(Path, Iterable<? extends CharSequence>, Charset, OpenOption...)`

Channels and `ByteBuffers`

- Stream I/O reads a character at a time, while channel I/O reads a buffer at a time.
- The `ByteChannel` interface provides basic read and write functionality.
- A `SeekableByteChannel` is a `ByteChannel` that has the capability to maintain a position in the channel and to change that position.
- The two methods for reading and writing channel I/O are:

```
newByteChannel(Path, OpenOption...)
newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)
```

- The capability to move to different points in the file and then read from or write to that location makes random access of a file possible.

NIO.2 supports channel and buffered stream I/O.

While stream I/O reads a character at a time, channel I/O reads a buffer at a time. The `ByteChannel` interface provides basic read and write functionality. A `SeekableByteChannel` is a `ByteChannel` that has the capability to maintain a position in the channel and query the file for its size.

The capability to move to different points in the file and then read from or write to that location makes random access of a file possible.

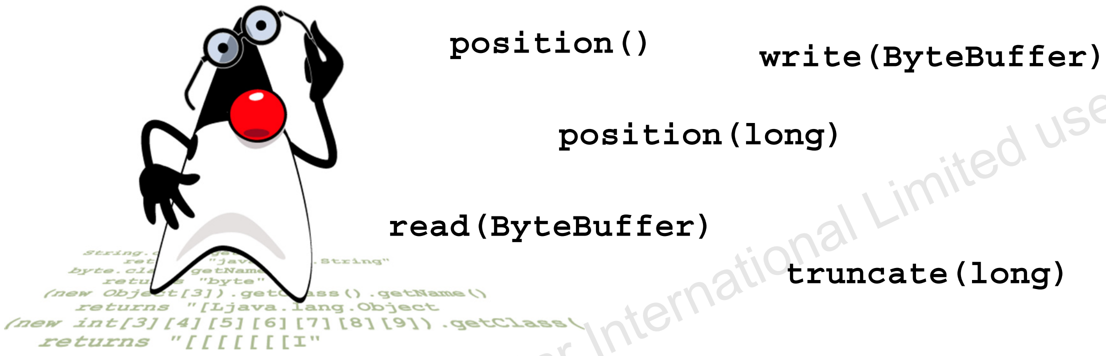
There are two methods for reading and writing channel I/O:

- `newByteChannel(Path, OpenOption...)`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

Note: The `newByteChannel` methods return an instance of a `SeekableByteChannel`. With a default file system, you can cast this seekable byte channel to a `FileChannel` providing access to more advanced features such as mapping a region of the file directly into memory for faster access, locking a region of the file so other processes cannot access it, or reading and writing bytes from an absolute position without affecting the channel's current position. Refer to the "I/O Fundamentals" lesson for an example of using `FileChannel`.

Random Access Files

- Random access files permit non-sequential, or random, access to a file's contents.
- To access a file randomly, open the file, seek a particular location, and read from or write to that file.
- Random access functionality is enabled by the `SeekableByteChannel` interface.



Random access files permit non-sequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

This functionality is possible with the `SeekableByteChannel` interface. The `SeekableByteChannel` interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

- **`position()`:** Returns the channel's current position
- **`position(long)`:** Sets the channel's position
- **`read(ByteBuffer)`:** Reads bytes into the buffer from the channel
- **`write(ByteBuffer)`:** Writes bytes from the buffer to the channel
- **`truncate(long)`:** Truncates the file (or other entity) connected to the channel

Buffered I/O Methods for Text Files

- The `newBufferedReader` method opens a file for reading.

```
//...
BufferedReader reader = Files.newBufferedReader(file, charset);
line = reader.readLine();
```

- The `newBufferedWriter` method writes to a file using a `BufferedWriter`.

```
//...
BufferedWriter writer = Files.newBufferedWriter(file, charset);
writer.write(s, 0, s.length());
```

Reading a File by Using Buffered Stream I/O

The `newBufferedReader(Path, Charset)` method opens a file for reading, returning a `BufferedReader` that can be used to read text from a file in an efficient manner.

Byte Streams

- NIO.2 also supports methods to open byte streams.

```
InputStream in = Files.newInputStream(file);
BufferedReader reader = new BufferedReader(new InputStreamReader(in));
line = reader.readLine();
```

- To create a file, append to a file, or write to a file, use the `newOutputStream` method.

```
import static java.nio.file.StandardOpenOption.*;
//...
Path logfile = ...;
String s = ...;
byte data[] = s.getBytes();
OutputStream out =
       new BufferedOutputStream(file.newOutputStream(CREATE, APPEND);
out.write(data, 0, data.length);
```

Managing Metadata

Method	Explanation
`size`	Returns the size of the specified file in bytes
`isDirectory`	Returns true if the specified `Path` locates a file that is a directory
`isRegularFile`	Returns true if the specified `Path` locates a file that is a regular file
`isSymbolicLink`	Returns true if the specified `Path` locates a file that is a symbolic link
`isHidden`	Returns true if the specified `Path` locates a file that is considered hidden by the file system
`getLastModifiedTime`	Returns or sets the specified file's last modified time
`setLastModifiedTime`	
`getAttribute`	Returns or sets the value of a file attribute
`setAttribute`	

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the `Files` class provides two `readAttributes` methods to fetch a file's attributes in one bulk operation.

- `readAttributes(Path, String, LinkOption...)`
- `readAttributes(Path, Class<A>, LinkOption...)`

File Attributes (DOS)

- File attributes can be read from a file or directory in a single call:

```
DosFileAttributes attrs =
   Files.readAttributes (path, DosFileAttributes.class);
```

- DOS file systems can modify attributes after file creation:

```
Files.createFile (file);
Files.setAttribute (file, "dos:hidden", true);
```

The `setAttribute` types (for DOS) extend the `BasicFileAttributeView` and view the standard four bits on file systems that support DOS attributes:

- `dos:hidden`
- `dos:readonly`
- `dos:system`
- `dos:archive`

Other supported attribute views include:

- `BasicFileAttributeView`: Provides a set of basic attributes supported by all file system implementations
- `PosixFileAttributeView`: Extends the `BasicFileAttributeView` with attributes that support the POSIX family of standards, such as UNIX
- `FileOwnerAttributeView`: Is supported by any file system implementation that supports the concept of a file owner
- `AclFileAttributeView`: Supports reading or updating a file's Access Control List (ACL). The NFSv4 ACL model is supported.
- `UserDefinedFileAttributeView`: Enables support of user-defined metadata

DOS File Attributes: Example

```
DosFileAttributes attrs = null;
Path file = ...;
try { attrs =
          Files.readAttributes(file, DosFileAttributes.class);
} catch (IOException e) { ///... }
FileTime creation = attrs.creationTime();
FileTime modified = attrs.lastModifiedTime();
FileTime lastAccess = attrs.lastAccessTime();
if (!attrs.isDirectory()) {
   long size = attrs.size();
}
// DosFileAttributes adds these to BasicFileAttributes
boolean archive = attrs.isArchive();
boolean hidden = attrs.isHidden();
boolean readOnly = attrs.isReadOnly();
boolean systemFile = attrs.isSystem();
```

The code fragment in the slide illustrates the use of the `DosFileAttributes` class. In the one call to the `readAttributes` method, the attributes of the file (or directory) are returned.

POSIX Permissions

With NIO.2, you can create files and directories on POSIX file systems with their initial permissions set.

```
Path p = Paths.get(args[0]);
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attrs =
    PosixFilePermissions.asFileAttribute(perms);
try {
    Files.createFile(p, attrs);
} catch (FileAlreadyExistsException f) {
    System.out.println("FileAlreadyExists" + f);
} catch (IOException i) {
    System.out.println("IOException:" + i);
}
```

Create a file in the `Path p` with optional attributes.

File systems that implement the Portable Operating System Interface (POSIX) standard can create files and directories with their initial permissions set. This solves a common problem in I/O programming where a file is created. Permissions on that file may be changed before the next execution to set permissions.

Permissions can be set only for POSIX-compliant file systems, such as MacOS, Linux, and Solaris. Windows (DOS-based) is not POSIX compliant. DOS-based files and directories do not have permissions, but rather file attributes.

Note: You can determine whether or not a file system supports POSIX programmatically by looking for what file attribute views are supported. For example:

```
boolean unixFS = false;
Set<String> views =
    FileSystems.getDefault().supportedFileAttributeViews();
for (String s : views) {
    if (s.equals("posix")) unixFS = true;
}
```

Quiz

Given the following fragment:

```
Path p1 = Paths.get("/export/home/peter");
Path p2 = Paths.get("/export/home/peter2");
Files.move(p1, p2, StandardCopyOption.REPLACE_EXISTING);
```

If the `peter2` directory does not exist, and the `peter` directory is populated with subfolders and files, what is the result?

a. `DirectoryNotEmptyException`
b. `NotDirectoryException`
a. Directory `peter2` is created.
b. Directory `peter` is copied to `peter2`.
c. Directory `peter2` is created and populated with files and directories from `peter`.

Answer: e

Quiz

Given this fragment:

```
Path source = Paths.get(args[0]);
Path target = Paths.get(args[1]);
Files.copy(source, target);
```

Assuming `source` and `target` are not directories, how can you prevent this copy operation from generating `FileAlreadyExistsException`?

a. Delete the `target` file before the copy.
b. Use the `move` method instead.
c. Use the `copyExisting` method instead.
d. Add the `REPLACE_EXISTING` option to the method.

Answer: a, d

Quiz

Given this fragment:

```
Path source = Paths.get("/export/home/mcginn/HelloWorld.java");
Path newdir = Paths.get("/export/home/heimer");
Files.copy(source, newdir.resolve(source.getFileName());
```

Assuming there are no exceptions, what is the result?

a. The contents of `mcginn` are copied to `heimer`.
b. `HelloWorld.java` is copied to `/export/home`.
c. `HelloWorld.java` is coped to `/export/home/heimer`.
d. The contents of `heimer` are copied to `mcginn`.

Answer: c

Practice 11-1 Overview: Writing a File Merge Application

In this practice, use the `Path` interface and `Files` class to open a letter template form, and substitute the name in the template with a name from a file containing a list of names.

- Use the `Path` interface to create a new file name for the custom letter.
- Use the `Files` class to read all of the strings from both files into `List` objects.
- Use the `Matcher` and `Pattern` classes to search for the token to replace in the template.

Recursive Operations

The `Files` class provides a method to walk the file tree for recursive operations, such as copies and deletes.

- `walkFileTree (Path start, FileVisitor<T>)`
- Example:

```
public class PrintTree implements FileVisitor<Path> {
    public FileVisitResult preVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult postVisitDirectory(Path, BasicFileAttributes){}
    public FileVisitResult visitFile(Path, BasicFileAttributes){}
    public FileVisitResult visitFileFailed(Path, BasicFileAttributes){}
}
```

```
public class WalkFileTreeExample {
    public printFileTree(Path p) {
        Files.walkFileTree(p, new PrintTree());
    }
}
```

The file tree is recursively explored. Methods defined by `PrintTree` are invoked as directories and files are reached in the tree. Each method is passed the current path as the first argument of the method.

The `FileVisitor` interface includes methods that are invoked as each node in a file tree is visited:

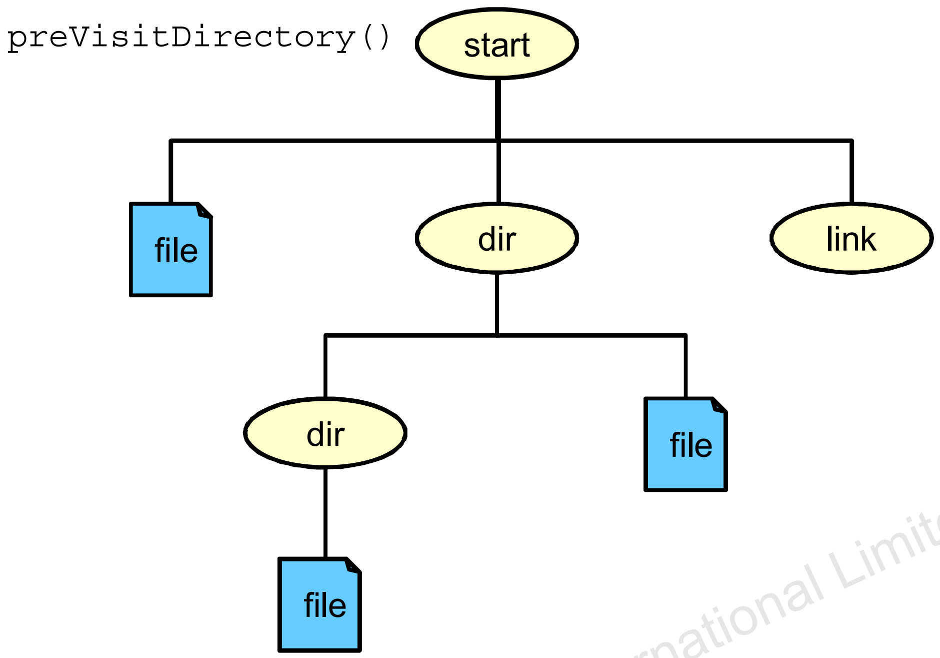
- `preVisitDirectory:` Invoked on a directory before the entries in the directory are visited
- `visitFile`: Invoked for a file in a directory
- `postVisitDirectory`: Invoked after all the entries in a directory and their descendants have been visited
- `visitFileFailed`: Invoked for a file that could not be visited

The return result from each of the called methods determines actions taken after a node is reached (pre or post). These are enumerated in the `FileVisitResult` class:

- `CONTINUE`: Continue to the next node
- `SKIP_SIBLINGS`: Continue without visiting the siblings of this file or directory
- `SKIP_SUBTREE`: Continue without visiting the entries in this directory
- `TERMINATE`

Note: There is also a class, `SimpleFileVisitor`, that implements each method in `FileVisitor` with a return type of `FileVisitResult.CONTINUE` or rethrows any `IOException`. If you plan on using only some of the methods in the `FileVisitor` interface, this class is easier to extend and override just the methods that you need.

`FileVisitor` Method Order



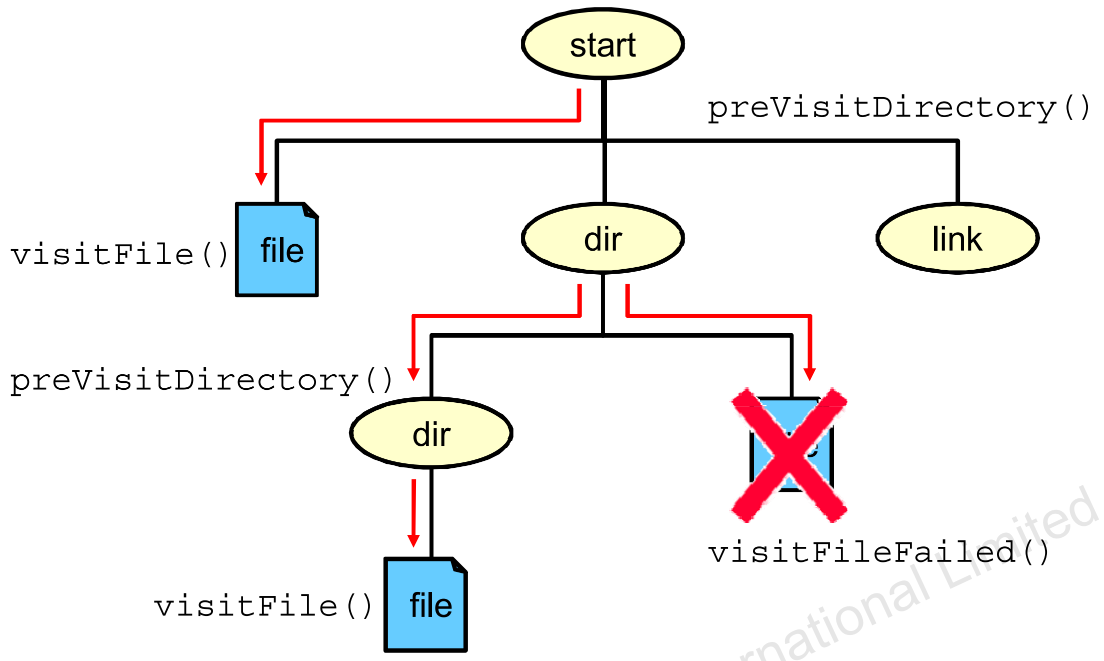
`FileVisitor`

Starting at the first directory node, and at every subdirectory encountered, the `preVisitDirectory(Path, BasicFileAttributes)` method is invoked on the class passed to the `walkFileTree` method.

Assuming that the return type from the invocation of `preVisitDirectory()` returns `FileVisitResult.CONTINUE`, the next node is explored.

Note: The file tree traversal is depth-first with the given `FileVisitor` invoked for each file encountered. File tree traversal completes when all accessible files in the tree have been visited, or a visit method returns a result of `TERMINATE`. Where a visit method terminates due an `IOException`, an uncaught error, or a runtime exception, the traversal is terminated and the error or exception is propagated to the caller of this method.

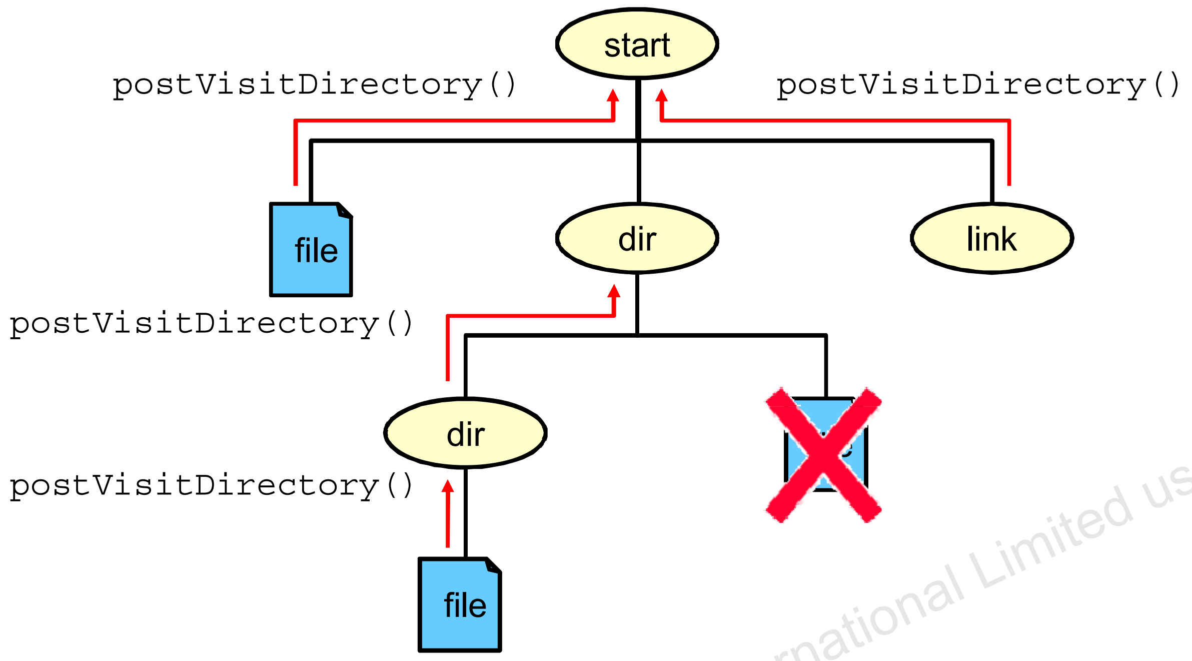
`FileVisitor` Method Order



When a file is encountered in the tree the `walkFileTree` method attempts to read its `BasicFileAttributes`. If the file is not a directory, the `visitFile` method is invoked with the file attributes. If the file attributes cannot be read, due to an I/O exception, the `visitFileFailed` method is invoked with the I/O exception.

.

`FileVisitor` Method Order

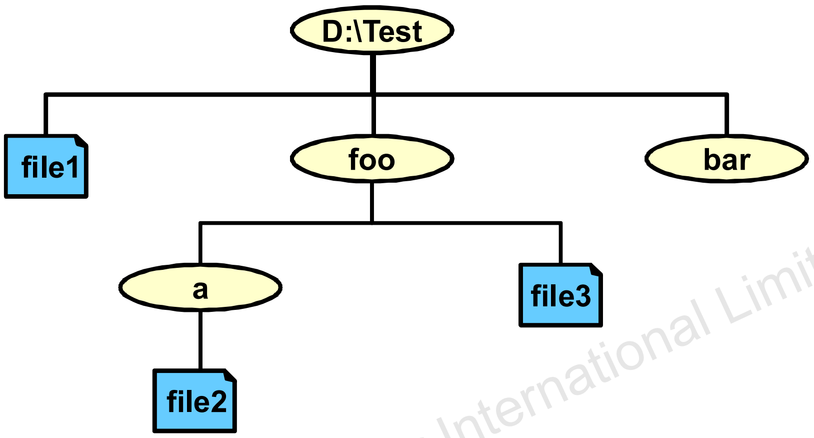


After reaching all the children in a node, the `postVisitDirectory` method is invoked on each of the directories.

Note: The progression illustrated here assumes that `FileVisitResult` return type is `CONTINUE` for each of the `FileVisitor` methods.

Example: WalkFileTreeExample

```
Path path = Paths.get("D:/Test");
try {
    Files.walkFileTree(path, new PrintTree());
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```



In this example, the `PrintTree` class implements each of the methods in `FileVisitor` and prints out the type, name, and size of the directory and file at each node. Using the diagram shown in the slide, the resulting output (on Windows) is shown below:

```
preVisitDirectory: Directory: D:\Test (0 bytes)
preVisitDirectory: Directory: D:\Test\bar (0 bytes)
postVisitDirectory: Directory: D:\Test\bar
visitFile: Regular file: D:\Test\file1 (328 bytes)
preVisitDirectory: Directory: D:\Test\foo (0 bytes)
preVisitDirectory: Directory: D:\Test\foo\a (0 bytes)
visitFile: Regular file: D:\Test\foo\a\file2 (22 bytes)
postVisitDirectory: Directory: D:\Test\foo\a
visitFile: Regular file: D:\Test\foo\file3 (12 bytes)
postVisitDirectory: Directory: D:\Test\foo
postVisitDirectory: Directory: D:\Test
```

The complete code for this example is in the `examples/WalkFileTreeExample` project.

Finding Files

To find a file, typically, you would search a directory. You could use a search tool, or a command, such as:

```
dir /s *.java
```

- This command will recursively search the directory tree, starting from where you are for all files that contain the `.java` extension.

The `java.nio.file.PathMatcher` interface includes a match method to determine whether a `Path` object matches a specified search string.

- Each file system implementation provides a `PathMatcher` that can be retrieved by using the `FileSystems` factory:

```
PathMatcher matcher = FileSystems.getDefault().getPathMatcher
(String syntaxAndPattern);
```

PathMatcher Syntax and Pattern

- The `syntaxAndPattern` string is of the form:

  `syntax:pattern`

  Where *`syntax`* can be “glob” and “regex”.
- The glob syntax is similar to regular expressions, but simpler:

Pattern Example	Matches
`*.java`	A path that represents a file name ending in `.java`
`*.*`	Matches file names containing a dot
`*.{java,class}`	Matches file names ending with `.java` or `.class`
`foo.?`	Matches file names starting with `foo.` and a single character extension
`C:\\*`	Matches `C:\foo` and `C:\bar` on the Windows platform (Note that the backslash is escaped. As a string literal in the Java Language, the pattern would be C:\\\\*.)

The following rules are used to interpret glob patterns:

- The `*` character matches zero or more characters of a name component without crossing directory boundaries.
- The `**` characters matches zero or more characters crossing directory boundaries.
- The `?` character matches exactly one character of a name component.
- The backslash character (`\`) is used to escape characters that would otherwise be interpreted as special characters. The expression `\\` matches a single backslash and `\{` matches a left brace for example.
- The `[ ]` characters are a bracket expression that matches a single character of a name component out of a set of characters. For example, `[abc]` matches `a`, `b`, or `c`. The hyphen (-) may be used to specify a range, so `[a-z]` specifies a range that matches from `a` through `z` (inclusive). These forms can be mixed so `[abce-g]` matches `a`, `b`, `c`, `e`, `f`, or `g`. If the character after the `[` is a `!`, then it is used for negation, so `[!a-c]` matches any character except `a`, `b`, or `c`.

- Within a bracket expression, the *, ? and \ characters match themselves. The (-) character matches itself if it is the first character within the brackets, or the first character after the ! if negating.
- The { } characters are a group of subpatterns, where the group matches if any subpattern in the group matches. The "," character is used to separate the subpatterns. Groups cannot be nested.
- Leading period/dot characters in a file name are treated as regular characters in match operations. For example, the “*” glob pattern matches file name `.login`. The `Files.isHidden(java.nio.file.Path)` method may be used to test whether a file is considered hidden.
- All other characters match themselves in an implementation-dependent manner. This includes characters representing any name separators.
- The matching of root components is highly implementation-dependent and is not specified.

When the syntax is “regex,” the pattern component is a regular expression as defined by the Pattern class.

For both the glob and regex syntaxes, the matching details, such as whether the matching is case-sensitive, are implementation-dependent and, therefore, not specified.

PathMatcher: Example

```
public static void main(String[] args) {
    // ... check for two arguments
    Path root = Paths.get(args[0]);
    // ... check that the first argument is a directory
    PathMatcher matcher =
        FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
    // Finder is class that implements FileVisitor
    Finder finder = new Finder(root, matcher);
    try {
         Files.walkFileTree(root, finder);
     } catch (IOException e) {
         System.out.println("Exception: " + e);
     }
     finder.done();
}
```

In this code fragment in the slide (the complete example is in the examples directory), two arguments are passed to the main.

The first argument is tested to see whether it is a directory. The second argument is used to create a `PathMatcher` instance with a regular expression using the `FileSystems` factory.

`Finder` is a class that implements the `FileVisitor` interface, so that it can be passed to a `walkFileTree` method. This class is used to call the match method on each of the files visited in the tree.

Finder Class

```
public class Finder extends SimpleFileVisitor<Path> {
    private Path file;
    private PathMatcher matcher;
    private int numMatches;
    // ... constructor stores Path and PathMatcher objects
    private void find(Path file) {
        Path name = file.getFileName();
        if (name != null && matcher.matches(name)) {
            numMatches++;
             System.out.println(file);
         }
     }
     @Override
     public FileVisitResult visitFile(Path file,
                                      BasicFileAttributes attrs) {
         find(file);
         return CONTINUE;
     }
     //...
 }
```

The slide shows a portion of the `Finder` class. This class is used to walk the tree and look for matches between file and the file reached by the `visitFile` method.

Other Useful NIO.2 Classes

- The `FileStore` class is useful for providing usage information about a file system, such as the total, usable, and allocated disk space.

```
Filesystem                    kbytes         used        avail
System (C:)                209748988     72247420    137501568
Data  (D:)                  81847292       429488     81417804
```

- An instance of the `WatchService` interface can be used to report changes to registered `Path` objects. `WatchService` can be used to identify when files are added, deleted, or modified in a directory.

```
ENTRY_CREATE: D:\test\New Text Document.txt
ENTRY_CREATE: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_MODIFY: D:\test\Foo.txt
ENTRY_DELETE: D:\test\Foo.txt
```

The slide shows examples output from the `DiskUsageExample` project and `WatchDirExample` project.

Moving to NIO.2

A method was added to the `java.io.File` class for JDK 7 to provide forward compatibility with NIO.2.

```
Path path = file.toPath();
```

- This enables you to take advantage of NIO.2 without having to rewrite a lot of code.
- Further, you could replace your existing code to improve future maintenance—for example, replace `file.delete();` with:

```
Path path = file.toPath();
Files.delete (path);
```

- Conversely, the `Path` interface provides a method to construct a `java.io.File` object:

```
File file = path.toFile();
```

Legacy java.io.File code

One of the benefits of the NIO.2 package is that you can enable legacy code to take advantage of the new API.

Summary

In this lesson, you should have learned how to:

- Use the `Path` interface to operate on file and directory paths
- Use the `Files` class to check, delete, copy, or move a file or directory
- Use `Files` class methods to read and write files using channel I/O and stream I/O
- Read and change file and directory attributes
- Recursively access a directory tree
- Find a file by using the `PathMatcher` class

Quiz

To copy, move, or open a file or directory using NIO.2, you must first create an instance of:

a. `Path`

b. `Files`

c. `FileSystem`

d. `Channel`

Answer: a

Quiz

Given any starting directory path, which `FileVisitor` method(s) would you use to delete a file tree?

a. `preVisitDirectory()`
b. `postVisitDirectory()`
c. `visitFile()`
d. `visitDirectory()`

Answer: b, c

You should use `visitFile` to delete a file discovered in the directory. `postVisitDirectory` can then delete the empty directory.

Quiz

Given an application where you want to count the depth of a file tree (how many levels of directories), which FileVisitor method should you use?

a. `preVisitDirectory()`
b. `postVisitDirectory()`
c. `visitFile()`
d. `visitDirectory()`

Answer: a

`preVisitDirectory` is called only once per directory, prior to visiting that node.

Practice 11-2 Overview: Recursive Copy

This practice covers creating a class by implementing `FileVisitor` to recursively copy one directory tree to another location.

- Allow the user of your application to decide to overwrite an existing directory or not.

(Optional) Practice 11-3 Overview: Using `PathMatcher` to Recursively Delete

This practice covers the following topics:

- Creating a class by implementing `FileVisitor` to delete a file by using a wildcard (That is, delete all the text files by using `*.txt`.)
- (Optional) Running the `WatchDirExample` in the examples directory while deleting files from a directory (or using the recursive copy application) to watch for changes

12

Threading

Objectives

After completing this lesson, you should be able to:

- Describe operating system task scheduling
- Define a thread
- Create threads
- Manage threads
- Synchronize threads accessing shared data
- Identify potential threading problems

Task Scheduling

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

- Processes: A process is an area of memory that contains both code and data. A process has a thread of execution that is scheduled to receive CPU time slices.
- Thread: A thread is a scheduled execution of a process. Concurrent threads are possible. All threads for a process share the same data memory but may be following different paths through a code section.

Preemptive Multitasking

Modern computers often have more tasks to execute than CPUs. Each task is given an amount of time (called a time slice) during which it can execute on a CPU. A time slice is usually measured in milliseconds. When the time slice has elapsed, the task is forcefully removed from the CPU and another task is given a chance to execute.

Why Threading Matters

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

- Resource Contention: Two or more tasks waiting for exclusive use of a resource
- Blocking I/O operations: Doing nothing while waiting for disk or network data transfers
- Underutilization of CPUs: A single-threaded application uses only a single CPU

Multithreaded Servers

Even if you do not write code to create new threads of execution, your code might be run in a multithreaded environment. You must be aware of how threads work and how to write thread-safe code. When creating code to run inside of another piece of software (such as a middleware or application server), you must read the products documentation to discover whether threads will be created automatically. For instance, in a Java EE application server, there is a component called a Servlet that is used to handle HTTP requests. Servlets must always be thread-safe because the server starts a new thread for each HTTP request.

The `Thread` Class

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

- Extends the `Thread` class
  - Simpler code
- Implements the `Runnable` interface
  - More flexible
  - `extends` is still free.

Extending `Thread`

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

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

The `run` Method

The code to be executed in a new thread of execution should be placed in a `run` method. You should avoid calling the `run` method directly. Calling the `run` method does not start a new thread and the effect would be no different than calling any other method.

Starting a `Thread`

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

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

Schedules the run method to be called

The `start` Method

The `start` method is used to begin executing a thread. The Java Virtual Machine will call the `Thread`'s `run` method. Exactly when the `run` method begins executing is beyond your control. A `Thread` can be started only once.

Implementing `Runnable`

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

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

The `run` Method

Just as when extending `Thread`, calling the `run` method does not start a new thread. The benefit of implementing `Runnable` is that you may still extend a class of your choosing.

Executing `Runnable` Instances

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

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

The `start` Method

The `Thread`'s `start` method is used to begin executing a thread. After the thread is started, the Java Virtual Machine will invoke the `run` method in the `Thread`'s associated `Runnable`.

A `Runnable` with Shared Data

Static and instance fields are potentially shared by threads.

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

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

Potentially shared variable

One Runnable: Multiple Threads

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

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

A single `Runnable` instance

Multiple Threads with One `Runnable`

It is possible to pass a single `Runnable` instance to multiple `Thread` instances. There are only as many `Runnable` instances as you create. Multiple `Thread` instances share the `Runnable` instance's fields.

Static fields can also be concurrently accessed by multiple threads.

Quiz

Creating a new thread requires the use of:

a. `java.lang.Runnable`

b. `java.lang.Thread`

c. `java.util.concurrent.Callable`

Answer: b

Problems with Shared Data

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

- Are created in an area of memory known as heap space
- Can potentially be shared by any thread
- Might be changed concurrently by multiple threads
  - There are no compiler or IDE warnings.
  - “Safely” accessing shared fields is your responsibility.

The preceding slides might produce the following:

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

Zero produced twice

Out of sequence

Debugging Threads

Debugging threads can be difficult because the frequency and duration of time each thread is allocated can vary for many reasons including:

- Thread scheduling is handled by an operating system and operating systems may use different scheduling algorithms
- Machines have different counts and speeds of CPUs
- Other applications may be placing load on the system

This is one of those cases where an application may seem to function perfectly while in development, but strange problems might manifest after it is in production because of scheduling variations. It is your responsibility to safeguard access to shared variables.

Nonshared Data

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

- Local variables
- Method parameters
- Exception handler parameters

Shared Thread-Safe Data

Any shared data that is immutable, such as `String` objects or final fields, are thread-safe because they can only be read and not written.

Quiz

Variables are thread-safe if they are:

a. `local`

b. `static`

c. `final`

d. `private`

Answer: a, c

Atomic Operations

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

- `i++;`
  - Creates a temporary copy of the value in `i`
  - Increments the temporary copy
  - Writes the new value back to `i`
- `l = 0xffff_ffff_ffff_ffff;`
  - 64-bit variables might be accessed using two separate 32-bit operations.

What inconsistencies might two threads incrementing the same field encounter?

What if that field is long?

Inconsistent Behavior

One possible problem with two threads incrementing the same field is that a lost update might occur. Imagine if both threads read a value of 41 from a field, increment the value by one, and then write their results back to the field. Both threads will have done an increment but the resulting value is only 42. Depending on how the Java Virtual Machine is implemented and the type of physical CPU being used, you may never or rarely see this behavior. However, you must always assume that it could happen.

If you have a long value of `0x0000_0000_ffff_ffff` and increment it by `1`, the result should be `0x0000_0001_0000_0000`. However, because it is legal for a 64-bit field to be accessed using two separate 32-bit writes, there could temporarily be a value of `0x0000_0001_ffff_ffff` or even `0x0000_0000_0000_0000` depending on which bits are modified first. If a second thread was allowed to read a 64-bit field while it was being modified by another thread, an incorrect value could be retrieved.

Out-of-Order Execution

- Operations performed in one thread may not appear to execute in order if you observe the results from another thread.
  - Code optimization may result in out-of-order operation.
  - Threads operate on cached copies of shared variables.
- To ensure consistent behavior in your threads, you must synchronize their actions.
  - You need a way to state that an action happens before another.
  - You need a way to flush changes to shared variables back to main memory.

Synchronizing Actions

Every thread has a *working memory* in which it keeps its own *working copy* of variables that it must use or assign. As the thread executes a program, it operates on these working copies. There are several actions that will synchronize a thread's *working memory* with main memory:

- A volatile read or write of a variable (the `volatile` keyword)
- Locking or unlocking a monitor (the `synchronized` keyword)
- The first and last action of a thread
- Actions that start a thread or detect that a thread has terminated

Quiz

Which of the following cause a thread to synchronize variables?

a. Reading a volatile field

b. Calling `isAlive()` on a thread

c. Starting a new thread

d. Completing a synchronized code block

Answer: a, b, c, d

The `volatile` Keyword

A field may have the `volatile` modifier applied to it:

```
public volatile int i;
```

- Reading or writing a `volatile` field will cause a thread to synchronize its working memory with main memory.
- `volatile` does not mean atomic.
  - If `i` is `volatile`, `i++` is still not a thread-safe operation.

Because the manipulation of `volatile` fields may not be atomic, it is not sufficient to make anything other than reads and writes of single variables thread-safe. A good example of using `volatile` is shown in the examples in the following slides about how to stop a thread.

Stopping a Thread

A thread stops by completing its `run` method.

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

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

Shared volatile variable

Stopping a Thread

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

The Main Thread

The `main` method in a Java SE application is executed in a thread, sometimes called the main thread, which is automatically created by the JVM. Just with any thread, when the main thread writes to the `timeToQuit` field, it is important that the write will be seen by the t1 thread. If the `timeToQuit` field was not `volatile`, there is no guarantee that the write would be seen immediately. Note that if you forget to declare a similar field as `volatile`, an application might function perfectly for you but occasionally fail to quit for someone else.

The `synchronized` Keyword

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

- Causes a thread to write all of its changes to main memory when the end of the block is reached
  - Similar to `volatile`
- Is used to group blocks of code for exclusive execution
  - Threads block until they can get exclusive access
  - Solves the atomic problem

`synchronized` Methods

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

Synchronized Method Behavior

In the example in the slide, you can call only one method at a time in a `ShoppingCart` object because all its methods are `synchronized`. In this example, the synchronization is per `ShoppingCart`. Two `ShoppingCart` instances could be used concurrently.

If the methods were not `synchronized`, calling `removeItem` while `printCart` is iterating through the `Item` collection might result in unpredictable behavior. An iterator may support fail-fast behavior. A fail-fast iterator will throw a `java.util.ConcurrentModificationException`, a subclass of `RuntimeException`, if the iterator's collection is modified while being used.

`synchronized` Blocks

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

Synchronization Bottlenecks

Synchronization in multithreaded applications ensures reliable behavior. Because `synchronized` blocks and methods are used to restrict a section of code to a single thread, you are potentially creating performance bottlenecks. `synchronized` blocks can be used in place of `synchronized` methods to reduce the number of lines that are exclusive to a single thread.

Use synchronization as little as possible for performance, but as much as needed to guarantee reliability.

Object Monitor Locking

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

- `synchronized` methods use the monitor for the `this` object.
- `static synchronized` methods use the classes' monitor.
- `synchronized` blocks must specify which object's monitor to lock or unlock.

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

- `synchronized` blocks can be nested.

Nested `synchronized` Blocks

A thread can lock multiple monitors simultaneously by using nested `synchronized` blocks.

Detecting Interruption

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

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

static `Thread` method

Interruption Does Not Mean Stop

When a thread is interrupted, it is up to you to decide what action to take. That action could be to return from the `run` method or to continue executing code.

Interrupting a Thread

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

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

Interrupt a thread

Benefits of Interruption

Using the interruption features of `Thread` is a convenient way to stop a thread. In addition to eliminating the need for you to write your own thread-stopping logic, it also can interrupt a thread that is blocked. For more information, see http://download.java.net/jdk7/docs/api/java/lang/Thread.html#interrupt().

Thread.sleep()

A `Thread` may pause execution for a duration of time.

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

`interrupt()` called while sleeping

How Long Will a Thread Sleep?

A request of `Thread.sleep(4000)` means that a thread wants to stop executing for 4 seconds. After that 4 seconds elapse, the thread is scheduled for execution again. This does not mean that the thread start up exactly 4 seconds after the call to `sleep()`, instead it means the thread will begin executing 4 seconds or longer after it began to sleep. The exact sleep duration is affected by machine hardware, operating system, and system load.

Sleep Interrupted

If you call `interrupt()` on a thread that is sleeping, the call to `sleep()` will throw an `InterruptedException` which must be handled. How you should handle the exception depends on how your application is designed. If calling `interrupt()` is just meant to interrupt the `sleep()` call and not the execution of a thread, then you might swallow the exception. Other cases might require you to rethrow the exception or return from a `run()` method.

Quiz

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

a. True

b. False

Answer: b

Additional `Thread` Methods

- There are many more `Thread` and threading-related methods:
  - `setName(String)`, `getName()`, and `getId()`
  - `isAlive()`: Has a thread finished?
  - `isDaemon()` and `setDaemon(boolean)`: The JVM can quit while daemon threads are running.
  - `join()`: A current thread waits for another thread to finish.
  - `Thread.currentThread()`: `Runnable` instances can retrieve the `Thread` instance currently executing.
- The `Object` class also has methods related to threading:
  - `wait()`, `notify()`, and `notifyAll()`: Threads may go to sleep for an undetermined amount of time, waking only when the `Object` they waited on receives a wakeup notification.

Learning More

Daemon threads are background threads that are less important than normal threads. Because the main thread is not a daemon thread, all threads that you create will also be nondaemon threads. Any nondaemon thread that is still running (alive) will keep the JVM from quitting even if your main method has returned. If a thread should not prevent the JVM from quitting, then it should be set as a daemon thread. There are more multithreading concepts and methods to learn about. For additional reading material, see http://download.oracle.com/javase/tutorial/essential/concurrency/further.html.

Methods to Avoid

Some `Thread` methods should be avoided:

- `setPriority(int)` and `getPriority()`
  - Might not have any impact or may cause problems
- The following methods are deprecated and should never be used:
  - `destroy()`
  - `resume()`
  - `suspend()`
  - `stop()`

Deprecation

Classes, interfaces, methods, variables, and other components of any Java library may be marked as deprecated. Deprecated components might cause unpredictable behavior or simply might have not followed proper naming conventions. You should avoid using any deprecated APIs in your applications. Deprecated APIs are still included in libraries to ensure backwards compatibility, but could potentially be removed in future versions of Java.

For more information about why the methods mentioned above are deprecated, refer to docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html, which is available online at http://download.oracle.com/javase/7/ or as part of the downloadable JDK documentation.

Deadlock

Deadlock results when two or more threads are blocked forever, waiting for each other.

```
synchronized(obj1) {
  synchronized(obj2) {
  }
}
```

Thread 1 pauses after locking obj1's monitor.

```
synchronized(obj2) {
  synchronized(obj1) {
  }
}
```

Thread 2 pauses after locking obj2's monitor.

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked; they are simply too busy responding to each other to resume work.

Summary

In this lesson, you should have learned how to:

- Describe operating system task scheduling
- Define a thread
- Create threads
- Manage threads
- Synchronize threads accessing shared data
- Identify potential threading problems

Practice 12-1 Overview: Synchronizing Access to Shared Data

This practice covers the following topics:

- Printing thread IDs
- Using `Thread.sleep()`
- Synchronizing a block of code

In this practice, you write a class that is added to an existing multithreaded application. You must make your class thread-safe.

Practice 12-2 Overview: Implementing a Multithreaded Program

This practice covers the following topics:

- Implementing `Runnable`
- Starting a `Thread`
- Checking the status of a `Thread`
- Interrupting a `Thread`

In this practice, you create, start, and interrupt basic threads by using the `Runnable` interface.

13 Concurrency

Objectives

After completing this lesson, you should be able to:

- Use atomic variables
- Use a `ReentrantReadWriteLock`
- Use the `java.util.concurrent` collections
- Describe the synchronizer classes
- Use an `ExecutorService` to concurrently execute tasks
- Apply the Fork-Join framework

The `java.util.concurrent` Package

Java 5 introduced the `java.util.concurrent` package, which contains classes that are useful in concurrent programming. Features include:

- Concurrent collections
- Synchronization and locking alternatives
- Thread pools
  - Fixed and dynamic thread count pools available
  - Parallel divide and conquer (Fork-Join) new in Java 7

The `java.util.concurrent.atomic` Package

The `java.util.concurrent.atomic` package contains classes that support lock-free thread-safe programming on single variables

```
AtomicInteger ai = new AtomicInteger(5);
if(ai.compareAndSet(5, 42)) {
    System.out.println("Replaced 5 with 42");
}
```

An atomic operation ensures that the current value is 5 and then sets it to 42.

Non-Blocking Operation

On CPU architectures that support a native compare and set operation there will be no need for locking when executing the example shown. Other architectures may require some form of internal locking.

The `java.util.concurrent.locks` Package

The `java.util.concurrent.locks` package is a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.

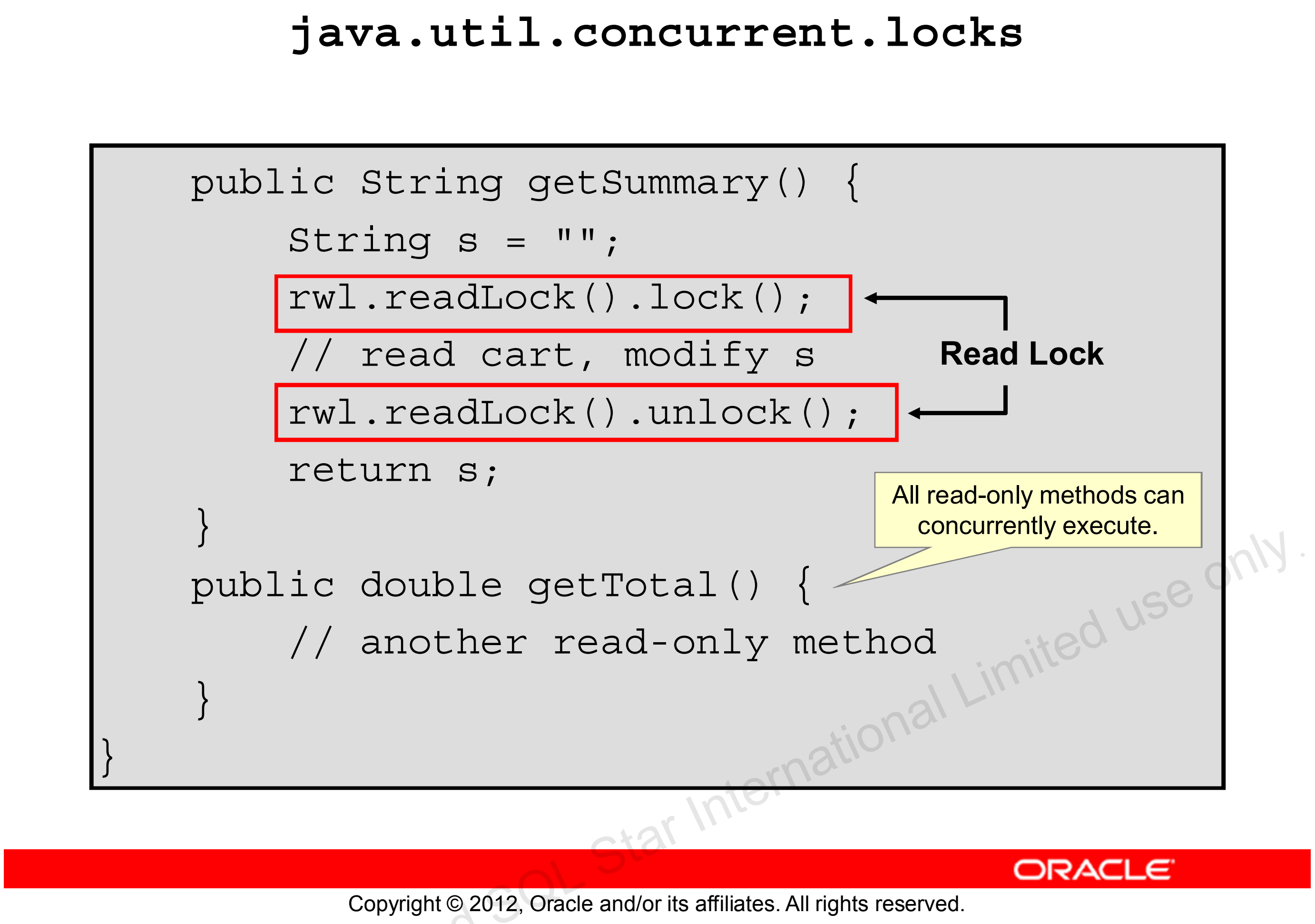
```
public class ShoppingCart {
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();

    public void addItem(Object o) {
        rwl.writeLock().lock();
        // modify shopping cart
        rwl.writeLock().unlock();
    }
```

A single writer, multi-reader lock

Write Lock

Multi-Reader, Single Writer Lock

One of the features of the `java.util.concurrent.locks` package is an implementation of a multi-reader, single writer lock. A thread may not have or obtain a read lock while a write lock is in use. Multiple threads can concurrently acquire the read lock but only one thread may acquire the write lock. The lock is reentrant; a thread that has already acquired the write lock may call additional methods that also obtain the write lock without a fear of blocking.



java.util.concurrent.locks

```
    public String getSummary() {
        String s = "";
        rwl.readLock().lock();
        // read cart, modify s
        rwl.readLock().unlock();
        return s;
    }
    public double getTotal() {
        // another read-only method
    }
}
```

Read Lock
All read-only methods can concurrently execute.

Multiple Concurrent Reads

In the example, all methods that are determined to be read-only can add the necessary code to lock and unlock a read lock. A `ReentrantReadWriteLock` allows concurrent execution of both a single read-only method and of multiple read-only methods.

Thread-Safe Collections

The `java.util` collections are not thread-safe. To use collections in a thread-safe fashion:

- Use synchronized code blocks for all access to a collection if writes are performed
- Create a synchronized wrapper using library methods, such as `java.util.Collections.synchronizedList(List<T>)`
- Use the `java.util.concurrent` collections

Note: Just because a `Collection` is made thread-safe, this does not make its elements thread-safe.

ORACLE

Concurrent Collections

The `ConcurrentLinkedQueue` class supplies an efficient scalable thread-safe nonblocking FIFO queue. Five implementations in `java.util.concurrent` support the extended `BlockingQueue` interface, which defines blocking versions of put and take: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`.

Besides queues, this package supplies `Collection` implementations designed for use in multithreaded contexts: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. When many threads are expected to access a given collection, a `ConcurrentHashMap` is normally preferable to a synchronized `HashMap`, and a `ConcurrentSkipListMap` is normally preferable to a synchronized `TreeMap`. A `CopyOnWriteArrayList` is preferable to a synchronized `ArrayList` when the expected number of reads and traversals greatly outnumber the number of updates to a list.

Quiz

A `CopyOnWriteArrayList` ensures the thread-safety of any object added to the `List`.

a. True

b. False

Answer: b

Synchronizers

The `java.util.concurrent` package provides five classes that aid common special-purpose synchronization idioms.

Class	Description
Semaphore	Semaphore is a classic concurrency tool.
CountDownLatch	A very simple yet very common utility for blocking until a given number of signals, events, or conditions hold
CyclicBarrier	A resettable multiway synchronization point useful in some styles of parallel programming
Phaser	Provides a more flexible form of barrier that may be used to control phased computation among multiple threads
Exchanger	Allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs

The synchronizer classes allow threads to block until a certain state or action is reached.

`Semaphore`: A `Semaphore` maintains a set of permits. Threads try to acquire permits and may block until other threads release permits.

`CountDownLatch`: A `CountDownLatch` allows one or more threads to await (block) until completion of a countdown. After the countdown is complete all awaiting threads continue. A `CountDownLatch` cannot be reused.

`CyclicBarrier`: Created with a party count. After the number of parties (threads) have called `await` on the `CyclicBarrier` they will be released (unblock). A `CyclicBarrier` can be reused.

`Phaser`: A more versatile version of a `CyclicBarrier` new to Java 7. Parties can register and deregister over time causing the number of threads required before advancement to change.

`Exchanger`: Allows two threads to swap a pair of objects, blocking until an exchange takes place. It is a bidirectional, memory efficient, alternative to a `SynchronousQueue`.

java.util.concurrent.CyclicBarrier

The `CyclicBarrier` is an example of the synchronizer category of classes provided by `java.util.concurrent`.

```
final CyclicBarrier barrier = new CyclicBarrier(2);

new Thread() {
    public void run() {
        try {
            System.out.println("before await - thread 1");
            barrier.await();
            System.out.println("after await - thread 1");
        } catch (BrokenBarrierException|InterruptedException ex) {

        }
    }
}.start();
```

Two threads must await before they can unblock.

May not be reached

`CyclicBarrier` Behavior

In this example, if only one thread calls `await()` on the barrier, that thread may block forever. After a second thread calls `await()`, any additional call to `await()` will again block until the required number of threads is reached. A `CyclicBarrier` contains a method, `await(long timeout, TimeUnit unit)`, which will block for a specified duration and throw a `TimeoutException` if that duration is reached.

High-Level Threading Alternatives

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

- `java.util.concurrent.ExecutorService`, a higher level mechanism used to execute tasks
  - It may create and reuse `Thread` objects for you.
  - It allows you to submit work and check on the results in the future.
- The Fork-Join framework, a specialized work-stealing `ExecutorService` new in Java 7

Synchronization Alternatives

Synchronized code blocks are used to ensure that data that is not thread-safe will not be accessed concurrently by multiple threads. However the use of synchronized code blocks can result in performance bottlenecks. Several components of the `java.util.concurrent` package provide alternatives to using synchronized code blocks. In addition to leveraging the concurrent collections, queues, and synchronizers, there is another way to ensure that data will not be incorrectly access by multiple threads: Simply do not allow multiple threads to process the same data. In some scenarios, it may be possible to create multiple copies of your data in RAM and allow each thread to process a unique copy.

`java.util.concurrent.ExecutorService`

An `ExecutorService` is used to execute tasks.

- It eliminates the need to manually create and manage threads.
- Tasks **might** be executed in parallel depending on the `ExecutorService` implementation.
- Tasks can be:
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
- Implementing instances can be obtained with `Executors`.

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

The Behavior of an `ExecutorService`

A cached thread pool `ExecutorService`:

- Creates new threads as needed
- Reuses its threads (Its threads do not die after finishing their task.)
- Terminates threads that have been idle for 60 seconds

Other types of `ExecutorService` implementations are available:

```
int cpuCount = Runtime.getRuntime().availableProcessors();
ExecutorService es = Executors.newFixedThreadPool(cpuCount);
```

A fixed thread pool `ExecutorService`:

- Contains a fixed number of threads
- Reuses its threads (Its threads do not die after finishing their task.)
- Queues up work until a thread is available
- Could be used to avoid over working a system with CPU-intensive tasks

`java.util.concurrent.Callable`

The `Callable` interface:

- Defines a task submitted to an `ExecutorService`
- Is similar in nature to `Runnable`, but can:
  - Return a result using generics
  - Throw a checked exception

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

java.util.concurrent.Future

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

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

}
```

`ExecutorService` controls when the work is done.

Gets the result of the `Callable`'s `call` method (blocks if needed).

If the `Callable` threw an `Exception`

Waiting on a Future

Because the call to `Future.get()` will block, you must do one of the following:

- Submit all your work to the `ExecutorService` before calling any `Future.get()` methods.
- Be prepared to wait for that `Future` to obtain the result.
- Use a non-blocking method such as `Future.isDone()` before calling `Future.get()` or use `Future.get(long timeout, TimeUnit unit)`, which will throw a `TimeoutException` if the result is not available within a given duration.

Shutting Down an `ExecutorService`

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

```
es.shutdown();

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

Stop accepting new `Callables`.

If you want to wait for the `Callables` to finish

Quiz

An `ExecutorService` will always attempt to use all of the available CPUs in a system.

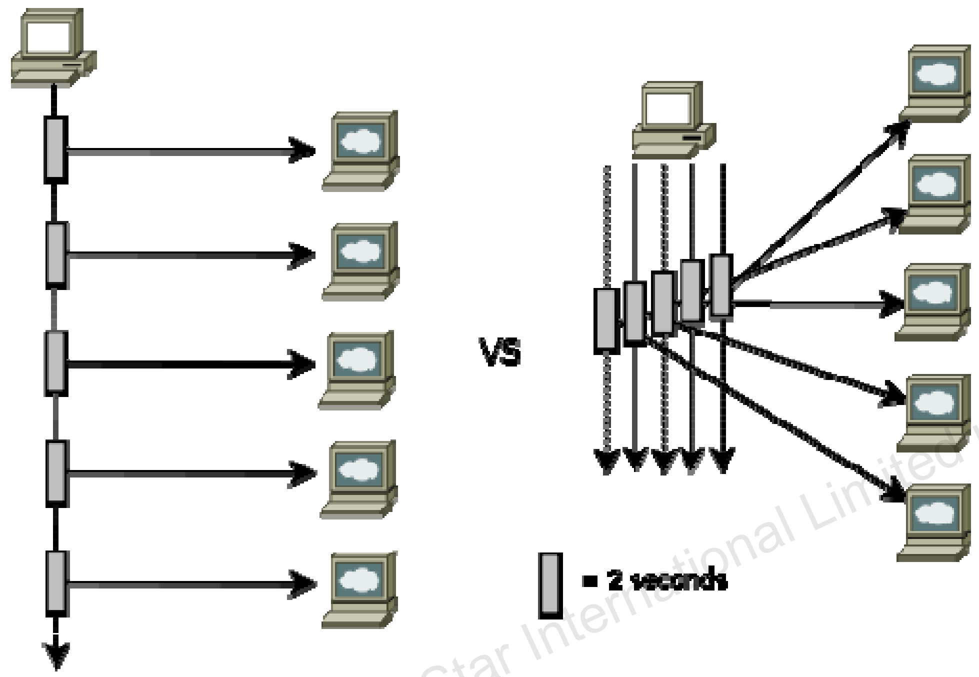
a. True

b. False

Answer: b

Concurrent I/O

Sequential blocking calls execute over a longer duration of time than concurrent blocking calls.



Wall Clock

There are different ways to measure time. In the graphic a sequence of five sequential calls to network servers will take approximately 10 seconds if each call takes 2 seconds. On the right side of the graphic, five concurrent calls to network servers may only take a little over 2 seconds if each call takes 2 seconds. Both examples use approximately the same amount of CPU time, the amount of CPU cycles consumed, but have different overall durations or wall clock time.

A Single-Threaded Network Client

```
public class SingleThreadClientMain {
    public static void main(String[] args) {
        String host = "localhost";
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup =
             new RequestResponse(host, port);
            try (Socket sock = new Socket(lookup.host, lookup.port);
                 Scanner scanner = new Scanner(sock.getInputStream());){
                lookup.response = scanner.next();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (NoSuchElementException|IOException ex) {
                System.out.println("Error talking to " + host + ":" +
                    port);
            }
        }
    }
}
```

Synchronous Invocation

In the example in this slide, we are trying to discover which vendor offers the lowest price for an item. The client will communicate with ten different network servers; each server will take approximately two seconds to look up the requested data and return it. There may be additional delays introduced by network latency.

This single-threaded client must wait for each server to respond before moving on to another server. About 20 seconds is required to retrieve all the data.

A Multithreaded Network Client (Part 1)

```
public class MultiThreadedClientMain {
    public static void main(String[] args) {
        //ThreadPool used to execute Callables
        ExecutorService es = Executors.newCachedThreadPool();
        //A Map used to connect the request data with the result
        Map<RequestResponse,Future<RequestResponse>> callables =
            new HashMap<>();

        String host = "localhost";
        //loop to create and submit a bunch of Callable instances
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup = new RequestResponse(host, port);
            NetworkClientCallable callable =
                new NetworkClientCallable(lookup);
            Future<RequestResponse> future = es.submit(callable);
            callables.put(lookup, future);
        }
```

Asynchronous Invocation

In the example in this slide, we are trying to discover which vendor offers the lowest price for an item. The client will communicate with ten different network servers, each server will take approximately two seconds to look up the requested data and return it. There may be additional delays introduced by network latency.

This multithreaded client does *not* wait for each server to respond before attempting to communicate with another server. About 2 seconds instead of 20 is required to retrieve all the data.

A Multithreaded Network Client (Part 2)

```
//Stop accepting new Callables
es.shutdown();

try {
    //Block until all Callables have a chance to finish
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

A Multithreaded Network Client (Part 3)

```
        for(RequestResponse lookup : callables.keySet()) {
            Future<RequestResponse> future = callables.get(lookup);
            try {
                lookup = future.get();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (ExecutionException|InterruptedException ex) {
                //This is why the callables Map exists
                //future.get() fails if the task failed
                System.out.println("Error talking to " + lookup.host +
                    ":" + lookup.port);
            }
        }
    }
}
```

A Multithreaded Network Client (Part 4)

```
public class RequestResponse {
    public String host; //request
    public int port; //request
    public String response; //response

    public RequestResponse(String host, int port) {
        this.host = host;
        this.port = port;
    }

    // equals and hashCode

}
```

A Multithreaded Network Client (Part 5)

```
public class NetworkClientCallable implements Callable<RequestResponse> {
    private RequestResponse lookup;

    public NetworkClientCallable(RequestResponse lookup) {
        this.lookup = lookup;
    }

    @Override
    public RequestResponse call() throws IOException {
        try (Socket sock = new Socket(lookup.host, lookup.port);
             Scanner scanner = new Scanner(sock.getInputStream());) {
            lookup.response = scanner.next();
            return lookup;
        }
    }
}
```

Parallelism

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

- Divide and conquer: A task should be divided into subtasks. You should attempt to identify those subtasks that can be executed in parallel.
- Some problems can be difficult to execute as parallel tasks.
- Some problems are easier. Servers that support multiple clients can use a separate task to handle each client.
- Be aware of your hardware. Scheduling too many parallel tasks can negatively impact performance.

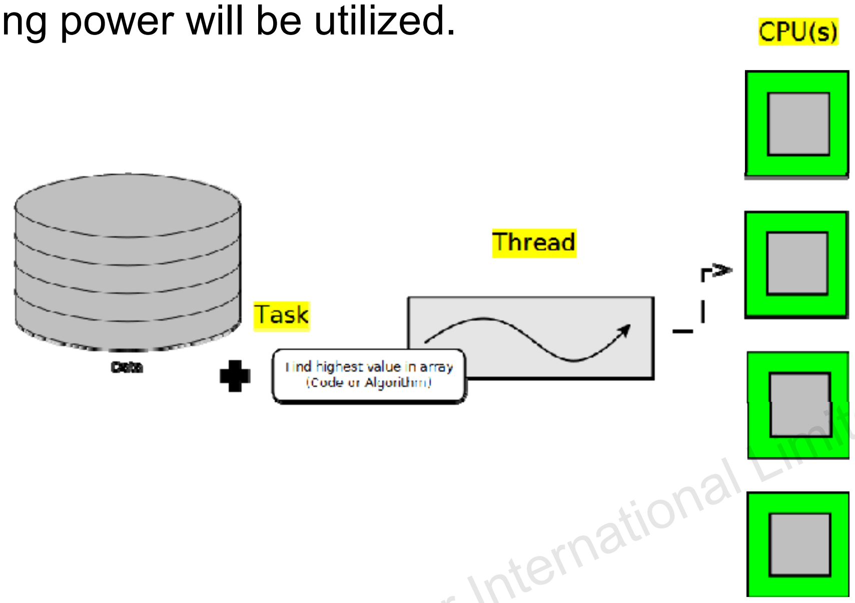
CPU Count

If your tasks are compute-intensive as opposed to I/O intensive, the number of parallel tasks should not greatly outnumber the number of processors in your system. You can detect the number of processors easily in Java:

```
int count = Runtime.getRuntime().availableProcessors();
```

Without Parallelism

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

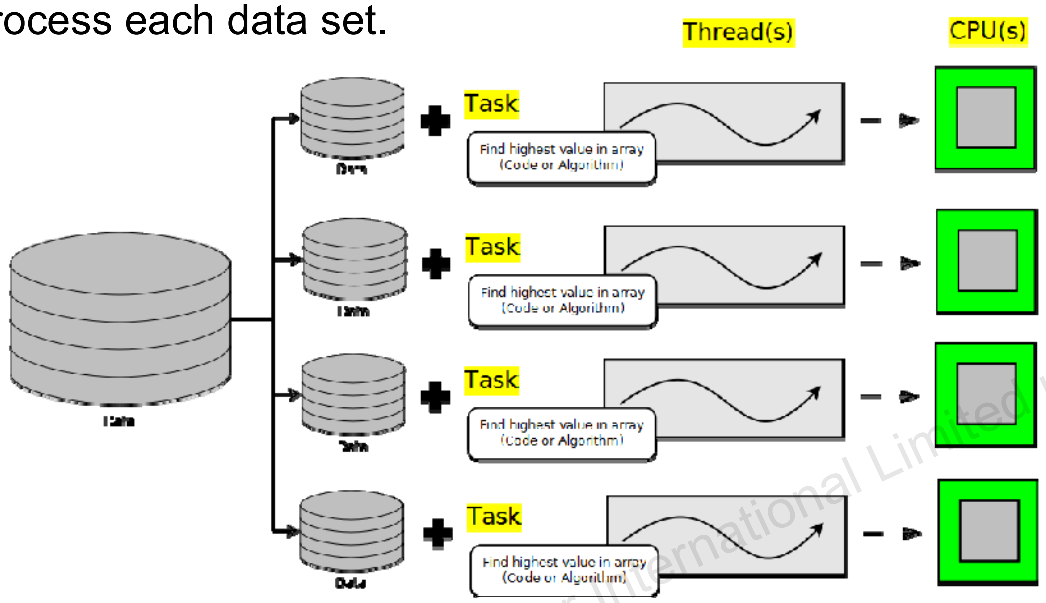


Setting the Stage

If you have a large amount of data to process but only one thread to process that data, a single CPU will be used. In the slide's graphic, a large set of data (an array, possibly) is be processed. The array processing could be a simple task such as finding the highest value in the array. In a four CPU system, there would be three CPUs sitting idle while the array was being processed.

Naive Parallelism

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


Thread(s)
CPU(s)
Task
Find highest value in array (Code or Algorithm)
Task
Find highest value in array (Code or Algorithm)
Task
Find highest value in array (Code or Algorithm)
Task
Find highest value in array (Code or Algorithm)
Data

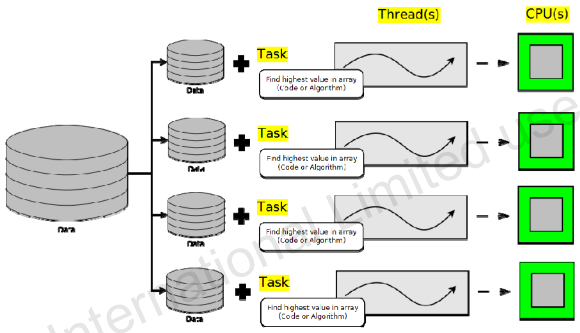
Splitting the Data

In the slide's graphic, a large set of data (an array, possibly) is split into four subsets of data, one subset for each CPU. A thread per CPU is created to process the data. After processing, the subsets of data the results will have to combined in a meaningful way. There are several ways to subdivide the large dataset to be processed. It would be overly memory-intensive to create a new array per thread that contains a copy of a portion of the original array. Each array can share a reference to the single large array but access only a subset in a non-blocking thread-safe way.

The Need for the Fork-Join Framework

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

- CPUs may run a different speeds
- Non-Java tasks require CPU time and may reduce the time available for a Java thread to spend executing on a CPU
- The data being analyzed may require varying amounts of time to process

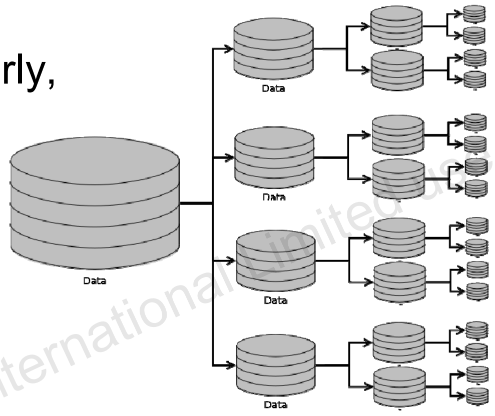


Work-Stealing

To keep multiple threads busy:

- Divide the data to be processed into a large number of subsets
- Assign the data subsets to a thread's processing queue
- Each thread will have many subsets queued

If a thread finishes all its subsets early, it can "steal" subsets from another thread.



Work Granularity

By subdividing the data to be processed until there are more subsets than threads, we are facilitating "work-stealing." In work-stealing, a thread that has run out of work can steal work (a data subset) from the processing queue of another thread. You must determine the optimal size of the work to add to each thread's processing queue. Overly subdividing the data to be processed can cause unnecessary overhead, while insufficiently subdividing the data can result in underutilization of CPUs.

A Single-Threaded Example

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

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

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

A very large dataset

Fill up the array with values.

Sequentially search the array for the largest value.

Parallel Potential

In this example there are two separate tasks that could be executed in parallel. Initializing the array with random values and searching the array for the largest possible value could both be done in parallel.

java.util.concurrent.ForkJoinTask<V>

A `ForkJoinTask` object represents a task to be executed.

- A task contains the code and data to be processed. Similar to a `Runnable` or `Callable`.
- A huge number of tasks are created and processed by a small number of threads in a Fork-Join pool.
  - A `ForkJoinTask` typically creates more `ForkJoinTask` instances until the data to processed has been subdivided adequately.
- Developers typically use the following subclasses:
  - `RecursiveAction`: When a task does not need to return a result
  - `RecursiveTask`: When a task does need to return a result

`RecursiveTask` Example

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

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

Result type of the task

The data to process

Where the work is done.
Notice the generic return type.

ORACLE

`compute` Structure

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

Asynchronously execute

Process in current thread

Block until done

`compute` Example (Below Threshold)

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

You decide the threshold.

The range within the array

`compute` Example (Above Threshold)

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

Task for left half of data

Task for right half of data

Memory Management

Notice that the same array is passed to every task but with different start and end values. If the subset of values to be processed were copied into a new array each time a task was created, memory usage would quickly skyrocket.

`ForkJoinPool` Example

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

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

The task's `compute` method is automatically called .

Fork-Join Framework Recommendations

- Avoid I/O or blocking operations.
  - Only one thread per CPU is created by default. Blocking operations would keep you from utilizing all CPU resources.
- Know your hardware.
  - A Fork-Join solution will perform slower on a one-CPU system than a standard sequential solution.
  - Some CPUs increase in speed when only using a single core, potentially offsetting any performance gain provided by Fork-Join.
- Know your problem.
  - Many problems have additional overhead if executed in parallel (parallel sorting, for example).

Parallel Sorting

When using Fork-Join to sort an array in parallel, you end up sorting many small arrays and then having to combine the small sorted arrays into larger sorted arrays. For an example see the sample application provided with the JDK in `C:\Program Files\Java\jdk1.7.0\sample\forkjoin\mergesort`.

Quiz

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

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Use atomic variables
- Use a `ReentrantReadWriteLock`
- Use the `java.util.concurrent` collections
- Describe the synchronizer classes
- Use an `ExecutorService` to concurrently execute tasks
- Apply the Fork-Join framework

(Optional) Practice 13-1 Overview: Using the `java.util.concurrent` Package

This practice covers the following topics:

- Using a cached thread pool (`ExecutorService`)
- Implementing `Callable`
- Receiving `Callable` results with a `Future`

In this practice, you create a multithread network client.

(Optional) Practice 13-2 Overview: Using the Fork-Join Framework

This practice covers the following topics:

- Extending `RecursiveAction`
- Creating and using a `ForkJoinPool`

In this practice, you create a multithread network client.

14

Building Database Applications with JDBC

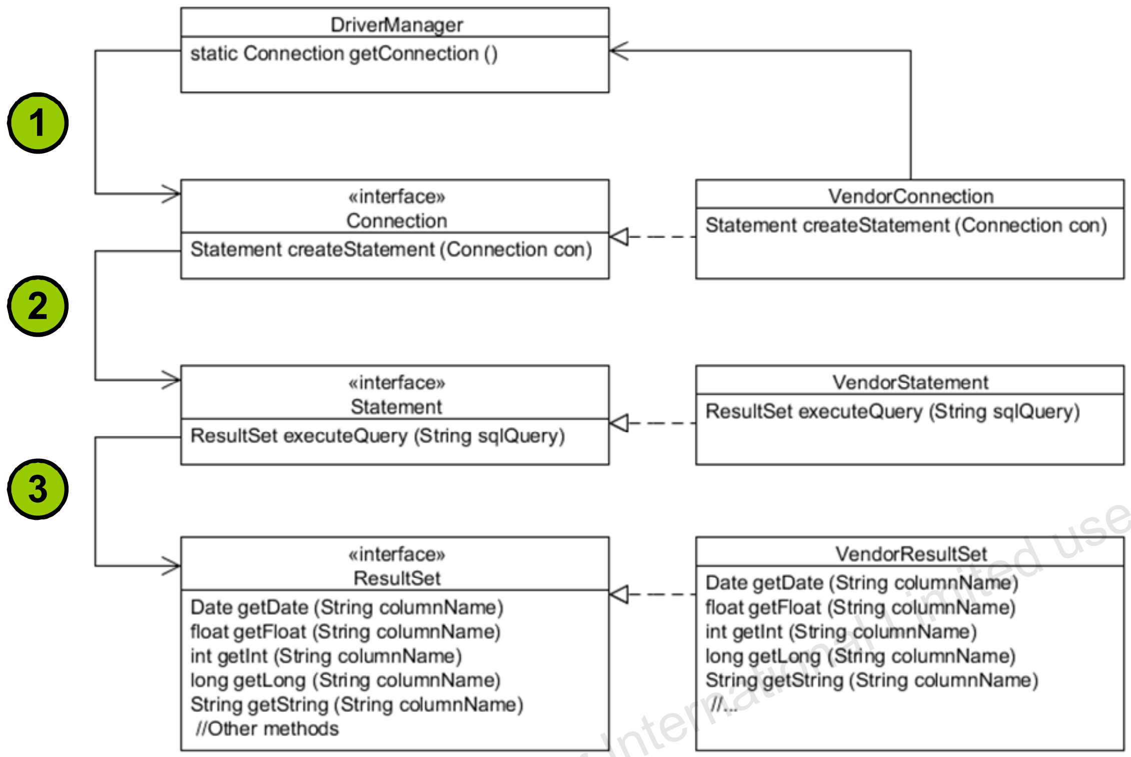
ORACLE

Objectives

After completing this lesson, you should be able to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Use transactions with JDBC
- Use the JDBC 4.1 `RowSetProvider` and `RowSetFactory`
- Use a Data Access Object Pattern to decouple data and business methods

Using the JDBC API



DriverManager
static Connection getConnection ()
1
«interface»
Connection
Statement createStatement (Connection con)
VendorConnection
Statement createStatement (Connection con)
2
«interface»
Statement
ResultSet executeQuery (String sqlQuery)
VendorStatement
ResultSet executeQuery (String sqlQuery)
3
«interface»
ResultSet
Date getDate (String columnName)
float getFloat (String columnName)
int getInt (String columnName)
long getLong (String columnName)
String getString (String columnName)
//Other methods
VendorResultSet
Date getDate (String columnName)
float getFloat (String columnName)
int getInt (String columnName)
long getLong (String columnName)
String getString (String columnName)
//...

The JDBC API is made up of some concrete classes, such as `Date`, `Time`, and `SQLException`, and a set of interfaces that are implemented in a driver class that is provided by the database vendor.

Because the implementation is a valid instance of the interface method signature, once the database vendor's Driver classes are loaded, you can access them by following the sequence shown in the slide:

1. Use the class `DriverManager` to obtain a reference to a `Connection` object using the `getConnection` method . The typical signature of this method is `getConnection (url, name, password)`, where `url` is the JDBC URL, and `name` and `password` are strings that the database will accept for a connection.
2. Use the `Connection` object (implemented by some class that the vendor provided) to obtain a reference to a `Statement` object through the `createStatement` method. The typical signature for this method is `createStatement ()` with no arguments.
3. Use the `Statement` object to obtain an instance of a `ResultSet` through an `executeQuery (query)` method. This method typically accepts a string (`query`) where `query` is a static string SQL.

Using a Vendor’s Driver Class

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

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

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

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

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

Example:

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

DriverManager

Any JDBC 4.0 drivers that are found in the class path are automatically loaded. The `DriverManager.getConnection` method will attempt to load the driver class by looking at the file `META_INF/services/java.sql.Driver`. This file contains the name of the JDBC driver’s implementation of `java.sql.Driver`. For example, the contents of `META-INF/services/java.sql.driver` file in the `derbyclient.jar` contains `org.apache.derby.jdbc.ClientDriver`.

Drivers prior to JDBC 4.0 must be loaded manually by using:

```
try {
    java.lang.Class.forName("<fully qualified path of the driver>");
} catch (ClassNotfoundException c) {
}
```

Driver classes can also be passed to the interpreter on the command line:

```
java -djdbc.drivers=<fully qualified path to the driver> <class to
run>
```

Key JDBC API Components

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

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

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

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

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

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

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

Connections, Statements, and ResultSets

The real beauty of the JDBC API lies in the way it provides a flexible and portable way to communicate with a database.

The JDBC driver that is provided by a database vendor implements each of these Java interfaces. Your Java code can use the interface knowing that the database vendor provided the implementation of each of the methods in the interface.

`Connection` is an interface that provides a session with the database. While the connection object is open, you can access the database, create statements, get results, and manipulate the database. When you close a connection, the access to the database is terminated and the open connection closed.

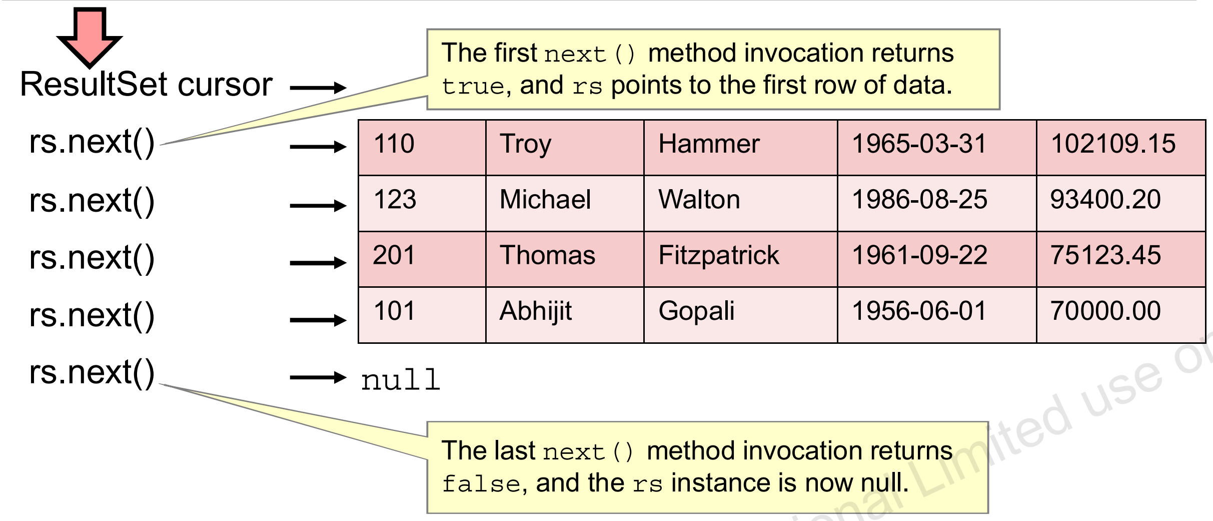
`Statement` is an interface that provides a class for executing SQL statements and returning the results. The `Statement` interface is for static SQL queries. There are two other subinterfaces: `PreparedStatement`, which extends `Statement` and `CallableStatement`, which extends `PreparedStatement`.

`ResultSet` is an interface that manages the resulting data returned from a `Statement`.

Note: SQL commands and keywords are case-insensitive—that is, you can use `SELECT`, or `Select`. SQL table and column names (identifiers) may be case-insensitive or case-sensitive depending upon the database. SQL identifiers are case-insensitive in the Derby database (unless delimited).

Using a `ResultSet` Object

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



The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

110	Troy	Hammer	1965-03-31	102109.15
123	Michael	Walton	1986-08-25	93400.20
201	Thomas	Fitzpatrick	1961-09-22	75123.45
101	Abhijit	Gopali	1956-06-01	70000.00

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

`ResultSet` Objects

- `ResultSet` maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The `ResultSet.next()` method is called to position the cursor in the next row.
- The default `ResultSet` is not updatable and has a cursor that points only forward.
- It is possible to produce `ResultSet` objects that are scrollable and/or updatable. The following code fragment, in which `con` is a valid `Connection` object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable:

```
Statement stmt
      = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

Note: Not all databases support scrollable result sets.

`ResultSet` has accessor methods to read the contents of each column returned in a row. `ResultSet` has a `getter` method for each type.

Putting It All Together

```
1 package com.example.text;
2
3 import java.sql.DriverManager;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.Date;
7
8 public class SimpleJDBCTest {
9
10    public static void main(String[] args) {
11        String url = "jdbc:derby://localhost:1527/EmployeeDB";
12        String username = "public";
13        String password = "tiger";
14        String query = "SELECT * FROM Employee";
15        try (Connection con =
16             DriverManager.getConnection (url, username, password);
17             Statement stmt = con.createStatement ();
18             ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password is just for this simple example.

In this slide and in the following slide, you see a complete example of a JDBC application, a simple one that reads all the rows from an Employee database and returns the results as strings to the console.

- **Line 15–16:** Use a `try`-with-resources statement to get an instance of an object that implements the `Connection` interface.
- **Line 17:** Use that object to get an instance of an object that implements the `Statement` interface from the `Connection` object.
- **Line 18:** Create a `ResultSet` by executing the string query using the `Statement` object.

Note: Hard coding the JDBC URL, username, and password makes an application less portable. Instead, consider using `java.io.Console` to read the username and password and/or some type of authentication service.

Putting It All Together

Loop through all of the rows in the `ResultSet`.

```
19          while (rs.next()) {
20              int empID = rs.getInt("ID");
21              String first = rs.getString("FirstName");
22              String last = rs.getString("LastName");
23              Date birthDate = rs.getDate("BirthDate");
24              float salary = rs.getFloat("Salary");
25              System.out.println("Employee ID:    " + empID + "\n"
26              + "Employee Name: " + first + " " + last + "\n"
27              + "Birth Date:    " + birthDate + "\n"
28              + "Salary:        " + salary);
29          } // end of while
30      } catch (SQLException e) {
31          System.out.println("SQL Exception: " + e);
32      } // end of try-with-resources
33   }
34 }
```

- **Lines 20–24:** Get the results of each of the data fields in each row read from the `Employee` table.
- **Lines 25–28:** Print the resulting data fields to the system console.
- **Line 30:** `SQLException`: This class extends `Exception` thrown by the `DriverManager`, `Statement`, and `ResultSet` methods. (More about this exception class in the next slide.)
- **Line 32:** This is the closing brace for the `try`-with-resources statement on line 15.

This example is from the `SimpleJDBCExample` project.

Output:

```
run:
Employee ID:    110
Employee Name: Troy Hammer
Birth Date:     1965-03-31
Salary:         102109.15
```

etc.

Writing Portable JDBC Code

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

- Most databases support a standard set of SQL syntax and semantics described by the American National Standards Institute (ANSI) SQL-92 Entry-level specification.
- You can programmatically check for support for this specification from your driver:

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

In general, you will probably write an application that leverages the capabilities and features of the database you are working with. However, if you want to write a portable application, you need to consider what support each database will provide for SQL types and functionality. Fortunately, you can query the database driver programmatically to determine what level of support the driver provides. The `DatabaseMetaData` interface has a set of methods that the driver developer uses to indicate what the driver supports, including support for the entry, intermediate, or full support for SQL-92.

The `DatabaseMetaData` interface also includes other methods that determine what type of support the database provides for queries, types, transactions, and more.

The `SQLException` Class

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

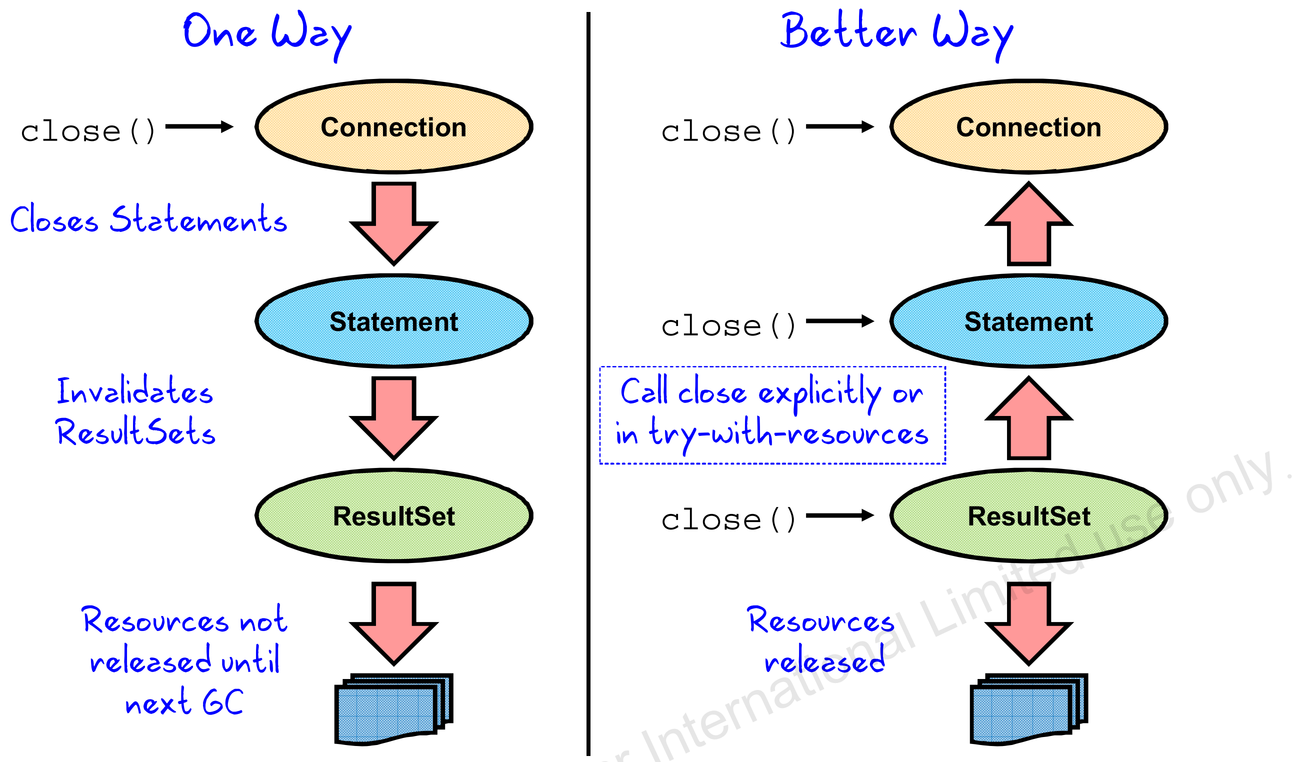
```
catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("Error Code:" + ex.getErrorCode());
        System.out.println("Message:   " + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null)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

Vendor-dependent state codes, error codes and messages

- A `SQLException` is thrown from errors that occur in one of the following types of actions: driver methods, methods that access the database, or attempts to get a connection to the database.
- The `SQLException` class also implements `Iterable`. Exceptions can be chained together and returned as a single object.
- `SQLException` is thrown if the database connection cannot be made due to incorrect username or password information or simply the database is offline.
- `SQLException` can also result by attempting to access a column name that is not part of the SQL query.
- `SQLException` is also subclassed, providing granularity of the actual exception thrown.

Note: `SQLState` and `SQLErrorCode` values are database dependent. For Derby, the `SQLState` values are defined here: http://download.oracle.com/javadb/10.8.1.2/ref/rrefexcept71493.html.

Closing JDBC Objects



- Closing a `Connection` object will automatically close any `Statement` objects created with this `Connection`.
- Closing a `Statement` object will close and invalidate any instances of `ResultSet` created by the `Statement` object.
- Resources held by the `ResultSet`, may not be released until garbage is collected, so it is a good practice to explicitly close `ResultSet` objects when they are no longer needed.
- When the `close()` method on `ResultSet` is executed, external resources are released.
- `ResultSet` objects are also implicitly closed when an associated Statement object is re-executed.

In summary, it is a good practice to explicitly close JDBC `Connection`, `Statement` and `ResultSet` objects when you no longer need them.

Note: A connection with the database can be an expensive operation. It is a good practice to either maintain `Connection` objects for as long as possible, or use a connection pool.

The `try`-with-resources Construct

Given the following `try`-with-resources statement:

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

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The close method is automatically called at the end of the try block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the try block, separated by semicolons.

One of the JDK 7 features is the `try`-with-resources statement. This is an enhancement that will automatically close open resources.

With JDBC 4.1, the JDBC API classes including `ResultSet`, `Connection`, and `Statement`, implement `java.lang.AutoCloseable`. The `close()` method of the `ResultSet`, `Statement`, and `Connection` objects will be called in order in this example.

`try`-with-resources: Bad Practice

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

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

- However, only the close method of `ResultSet` is called, which is not a good practice.
- Always keep in mind which resources you need to close when using `try`-with-resources.

Avoid This `try`-with-resources Pitfall

It may appear to be a time-saving way to write these three statements, but the net effect is that the `Connection` returned by the `DriverManager` is never explicitly closed after the end of the try block, which is not a good practice.

Writing Queries and Getting Results

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

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

- Use the Statement instance to execute a SQL query:

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

- Note that there are three Statement execute methods:

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

A SQL statement is executed against a database using an instance of a Statement object. The Statement object is a wrapper object for a query. A Statement object is obtained through a Connection object—the database connection. So it makes sense that from a Connection, you get an object that you can use to write statements to the database.

The Statement interface provides three methods for creating SQL queries and returning a result. Which one you use depends upon the type of SQL statement you want to use:

- `executeQuery(sqlString)`: For a `SELECT` statement, returns a `ResultSet` object
- `executeUpdate(sqlString)`: For `INSERT`, `UPDATE`, and `DELETE` statements, returns an `int` (number of rows affected), or 0 when the statement is a Data Definition Language (DDL) statement, such as `CREATE TABLE`.
- `execute(sqlString)`: For any SQL statement, returns a `boolean` indicating if a `ResultSet` was returned. Multiple SQL statements can be executed with `execute`.

Practice 14-1 Overview: Working with the Derby Database and JDBC

This practice covers the following topics:

- Starting the JavaDB (Derby) database from within NetBeans IDE
- Populating the database with data (the Employee table)
- Running SQL queries to look at the data
- Compiling and running the sample JDBC application

In this practice, you will start the database from within NetBeans, populate the database with data, run some SQL queries, and compile and run a simple application that returns the rows of the Employee database table.

ResultSetMetaData

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

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

Note that these methods are indexed from 1, not 0.

The `ResultSetMetaData` class is obtained from a `ResultSet`.

The `getColumnCount` returns the number of columns returned in the query that produced the `ResultSet`.

The `getColumnName` and `getColumnTypeName` methods return strings. These could be used to perform a dynamic retrieval of the column data.

Note: These methods use 1 to indicate the first column, not 0.

Given a query of "SELECT * FROM Employee" and the Employee data table from the practices, this fragment produces this result:

```
Number of columns returned: 5
Column names/types returned:
ID : INTEGER
FIRSTNAME : VARCHAR
LASTNAME : VARCHAR
BIRTHDATE : DATE
SALARY : REAL
```

Getting a Row Count

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

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

Move the cursor to the last row, this method returns `false` if the `ResultSet` is empty.

Returning the row cursor to its original position before the call is a good practice.

- To use this technique, the `ResultSet` must be scrollable.

Note: Recall that to create a `ResultSet` that is scrollable, you must define the `ResultSet` type in the `createStatement` method:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

There is another technique for non-scrollable ResultSets. Using the SQL COUNT function, one query determines the number of rows and a second reads the results. Be aware that this technique requires locking control over the tables to ensure the count is not changed during the operation:

```
ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM EMPLOYEE");
rs.next();
int count = rs.getInt(1);
rs.stmt.excuteQuery ("SELECT * FROM EMPLOYEE");
// process results
```

Controlling `ResultSet` Fetch Size

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

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

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

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

Note: Normally the most efficient fetch size is already the default for the driver.

Using `PreparedStatement`

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

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

Parameter for substitution.

Substitutes `value` for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- `PreparedStatement` is useful when you have a SQL statements that you are going to execute multiple times.

`PreparedStatement`

The SQL statement in the example in the slide is precompiled and stored in the `PreparedStatement` object. This statement can be used efficiently to execute this statement multiple times. This example could be in a loop, looking at different values.

Prepared statements can also be used to prevent SQL injection attacks. For example, where a user is allowed to enter a string, that when executed as a part of a SQL statement, enables the user to alter the database in unintended ways (like granting themselves permissions).

Note: `PreparedStatement` setXXXX methods index parameters from 1, not 0. The first parameter in a prepared statement is 1, the second parameter is 2, and so on.

Using `CallableStatement`

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

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

The `IN` parameter is passed in to the stored procedure.

The `OUT` parameter is returned from the stored procedure.

- Stored procedures are executed on the database.

Derby Stored Procedures

The Derby database uses the Java programming language for its stored procedures.

In the example shown in the slide, the stored procedure is declared using the following syntax:

```
CREATE PROCEDURE EmplAgeCount (IN age INTEGER, OUT num INTEGER)
DYNAMIC RESULT SETS 0
LANGUAGE JAVA
EXTERNAL NAME 'DerbyStoredProcedure.countAge'
PARAMETER STYLE JAVA
READS SQL DATA;
```

A Java class is loaded into the Derby database using the following syntax:

```
CALL SQLJ.install_jar ('D:\temp\DerbyStoredProcedure.jar',
'PUBLIC.DerbyStoredProcedure', 0);
```

```
CALL
syscs_util.syscs_set_database_property('derby.database.classpath',
'PUBLIC.DerbyStoredProcedure');
```

The Java class stored in the Derby database that performs the stored procedure calculates a date that is age years in the past based on today's date. The SQL query counts the number of unique employees that are older (or equal to) the number of years passed in and returns that count as the second parameter of the stored procedure. The code in this example looks like this:

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Calendar;

public class DerbyStoredProcedure {
    public static void countAge (int age, int[] count) throws SQLException {
        String url = "jdbc:default:connection";
        Connection con = DriverManager.getConnection(url);
        String query = "SELECT COUNT(DISTINCT ID) " +
                       "AS count FROM Employee " +
                       "WHERE Birthdate <= ?";
        PreparedStatement ps = con.prepareStatement(query);
        Calendar now = Calendar.getInstance();
        now.add(Calendar.YEAR, (age*-1));
        Date past = new Date (now.getTimeInMillis());
        ps.setDate(1, past);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            count[0] = rs.getInt(1);
        } else {
            count[0] = 0;
        }
        con.close();
    }
}
```

Consult the Derby Reference Manual and Derby Tools and Utilities Guide for more information on creating stored procedures.

What Is a Transaction?

- A transaction is a mechanism to handle groups of operations as though they were one.
- Either all operations in a transaction occur or none occur at all.
- The operations involved in a transaction might rely on one or more databases.

A classic example of when a transaction would be used is as follows. Suppose that a client application needs to make a service request that might involve multiple read and write operations to a database. If any one invocation is unsuccessful, any state that is written (either in memory or, more typically, to a database) must be rolled back.

Consider an interbank fund transfer application in which money is transferred from one bank to another.

The transfer operation requires the server to make the following invocations:

1. Invoking the debit method on one account at the first bank
2. Invoking the credit method on another account at the second bank

If the credit invocation on the second bank fails, the banking application must roll back the previous debit invocation on the first bank.

Note: When a transaction spans multiple databases, more complicated transaction services may be required.

ACID Properties of a Transaction

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

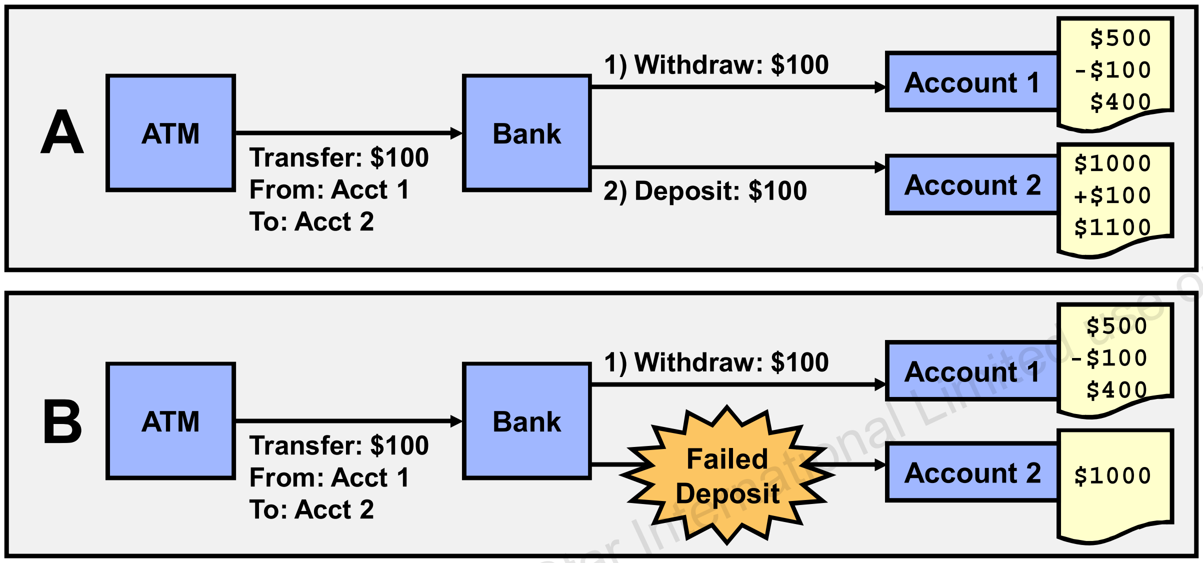
- **Atomicity:** A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **Consistency:** A transaction transforms a system from one consistent state to another consistent state.
- **Isolation:** Each transaction occurs independently of other transactions that occur at the same time.
- **Durability:** Completed transactions remain permanent, even during system failure.

Transactions should have the following ACID properties:

- **Atomicity:** All or nothing; all operations involved in the transaction are implemented or none are.
- **Consistency:** The database must be modified from one consistent state to another. In the event the system or database fails during the transaction, the original state is restored (rolled back).
- **Isolation:** An executing transaction is isolated from other executing transactions in terms of the database records it is accessing.
- **Durability:** After a transaction is committed, it can be restored to this state in the event of a system or database failure.

Transferring Without Transactions

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



Transactions are appropriate in the following scenarios. Each situation describes a transaction model that is supported by the resource local transaction model implementation in the EntityManager instance.

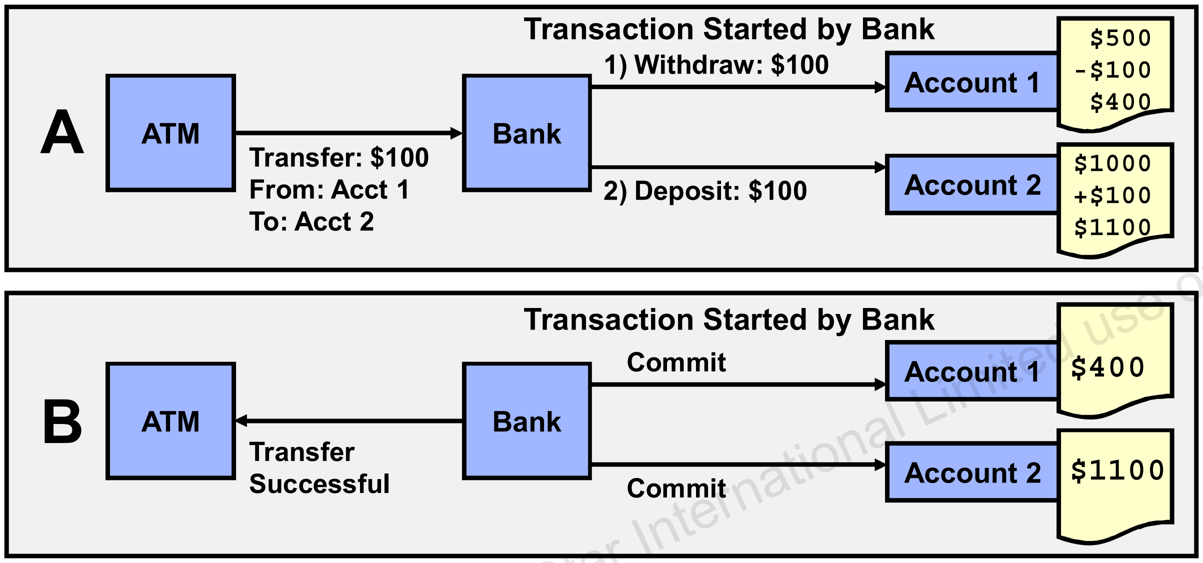
A client application must converse with an object that is managed, and it must make multiple invocations on a specific object instance. The conversation can be characterized by one or more of the following:

A. Data is cached in memory or written to a database during or after each successive invocation.
B. Data is written to a database at the end of the conversation.
C. The client application requires that the object maintains an in-memory context between each invocation; each successive invocation uses the data that is maintained in memory.
D. At the end of the conversation, the client application requires the capability to cancel all the database write operations that may have occurred during or at the end of the conversation.

Consider a shopping cart application. Users of the client application browse an online catalog and make multiple purchase selections. They proceed to check out and enter credit card information to make the purchase. If the credit card verification fails, the shopping application must cancel all the pending purchase selections in the shopping cart or roll back the purchase transactions made during the conversation.

Successful Transfer with Transactions

- Changes within a transaction are buffered. (A)
- If a transfer is successful, changes are committed (made permanent). (B)



If the transaction is successful, the buffered changes are committed, that is, made permanent.

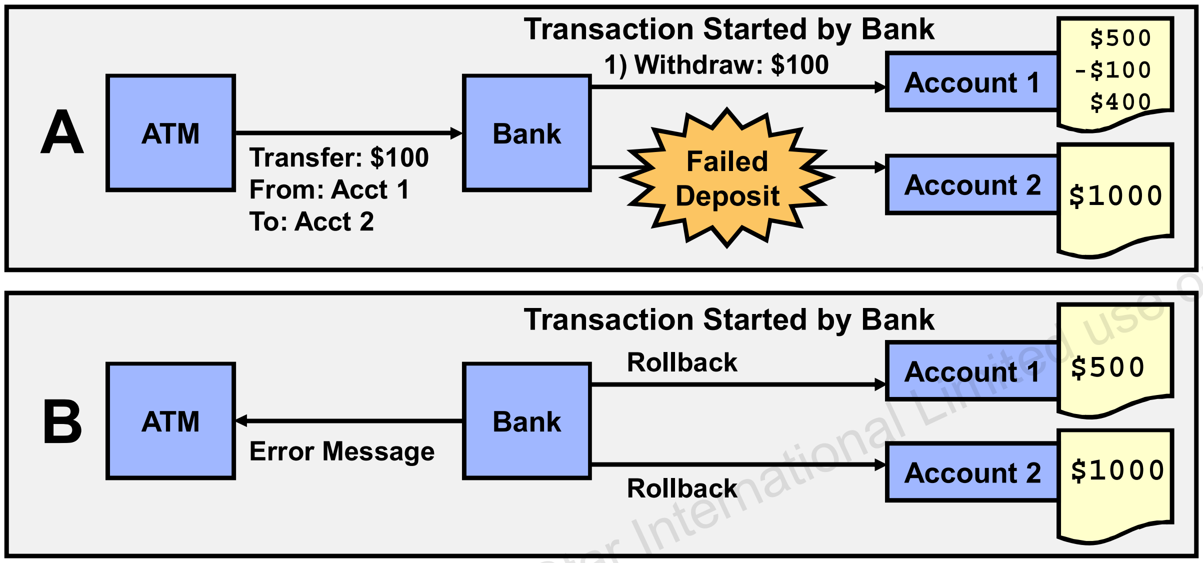
Within the scope of one client invocation on an object, the object performs multiple changes to the data in a database. If one change fails, the object must roll back all the changes. Consider a banking application. The client invokes the transfer operation on a teller object. The operation requires the teller object to make the following invocations on the bank database:

1. Invoking the debit method on one account
2. Invoking the credit method on another account

If the credit invocation on the bank database fails, the banking application must roll back the previous debit invocation.

Unsuccessful Transfer with Transactions

- Changes within a transaction are buffered. (A)
- If a problem occurs, the transaction is rolled back to the previous consistent state. (B)



If the transaction is unsuccessful, the buffered changes are thrown out and the database is rolled back to its previous consistent state.

JDBC Transactions

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

- Each individual SQL statement is treated as a transaction and automatically committed after it is executed.
- To group two or more statements together, you must disable auto-commit mode.

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

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

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

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

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

By default, JDBC auto-commits all SQL statements. However, when you want to create an atomic operation that involves multiple SQL statements, you must disable auto-commit.

After auto-commit is disabled, no SQL statements are committed to the database until you explicitly call the commit method.

The other advantage of managing your own transactions is the ability to rollback a set of SQL statements in the event of a failure using the rollback method.

Note: JDBC does not have an API to explicitly begin a transaction. The JDBC JSR (221) provides the following guidelines:

- When auto-commit is disabled for a `Connection` object, all subsequent `Statements` are in a transaction context until either the `Connection` commit or rollback method is executed.
- If the value of auto-commit is changed in the middle of a transaction, the current transaction is committed.

In addition, the Derby driver documentation adds the following:

- A transaction context is associated with a single `Connection` object (and database). A transaction cannot span multiple `Connections` (or databases).

Note: A sample application using transactions is in the project file `JDBCTransactionsExample` in the examples directory.

RowSet 1.1: `RowSetProvider` and `RowSetFactory`

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

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

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

- The default `RowSetFactory` implementation is:

```
com.sun.rowset.RowSetFactoryImpl
```

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

RowSet 1.1

New for JDK7 are the `javax.sql.rowset.RowSetProvider` and `javax.sql.rowset.RowSetFactory` classes. These two classes are used to construct instances of RowSets as discussed in the next slide.

Using RowSet 1.1 `RowSetFactory`

`RowSetFactory` is used to create instances of `RowSet` implementations:

RowSet type	Provides
`CachedRowSet`	A container for rows of data that caches its rows in memory
`FilteredRowSet`	A RowSet object that provides methods for filtering support
`JdbcRowSet`	A wrapper around ResultSet to treat a result set as a JavaBeans component
`JoinRowSet`	A RowSet object that provides mechanisms for combining related data from different RowSet objects
`WebRowSet`	A RowSet object that supports the standard XML document format required when describing a RowSet object in XML

- **`CachedRowSet`**: A `CachedRowSet` object is a container for rows of data that caches its rows in memory. This makes it possible to operate without always being connected to its data source. Further, it is a JavaBeans component and is scrollable, updatable, and serializable. A `CachedRowSet` object typically contains rows from a result set, but it can also contain rows from any file with a tabular format, such as a spreadsheet. The reference implementation supports getting data only from a `ResultSet` object, but developers can extend the `SyncProvider` implementations to provide access to other tabular data sources.
- **`FilteredRowSet`**: A JDBC `FilteredRowSet` standard implementation implements the `RowSet` interfaces and extends the `CachedRowSet` class. The `CachedRowSet` class provides a set of protected cursor manipulation methods, which a `FilteredRowSet` implementation can override to supply filtering support.

- **`JdbcRowSet`**: A **`JdbcRowSet`** is a wrapper around a `ResultSet` object that makes it possible to use the result set as a JavaBeans component. Thus, a `JdbcRowSet` object can be one of the Beans that a tool makes available for composing an application. Because a `JdbcRowSet` is a connected rowset, that is, it continually maintains its connection to a database using a JDBC technology–enabled driver, it also effectively makes the driver a JavaBeans component.
- **`JoinRowSet`**: The `JoinRowSet` interface provides a mechanism for combining related data from different `RowSet` objects into one `JoinRowSet` object, which represents a SQL JOIN. In other words, a `JoinRowSet` object acts as a container for the data from the `RowSet` objects that form a SQL JOIN relationship.
- **`WebRowSet`**: The `WebRowSet` interface describes the standard XML document format required when describing a `RowSet` object in XML, and must be used by all standard implementations of the `WebRowSet` interface to ensure interoperability. In addition, the `WebRowSet` schema uses specific SQL/XML Schema annotations, thus ensuring greater cross-platform interoperability. This is an effort currently under way at the ISO organization.

Example: Using `JdbcRowSet`

```
try (JdbcRowSet jdbcRs =
     RowSetProvider.newFactory().createJdbcRowSet()) {
    jdbcRs.setUrl(url);
    jdbcRs.setUsername(username);
    jdbcRs.setPassword(password);
    jdbcRs.setCommand("SELECT * FROM Employee");
    jdbcRs.execute();
    // Now just treat JDBC Row Set like a ResultSet object
    while (jdbcRs.next()) {
        int empID = jdbcRs.getInt("ID");
        String first = jdbcRs.getString("FirstName");
        String last = jdbcRs.getString("LastName");
        Date birthDate = jdbcRs.getDate("BirthDate");
         float salary = jdbcRs.getFloat("Salary");
    }
  //... other methods
}
```

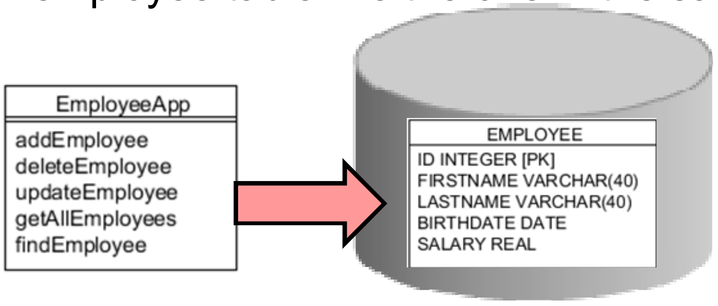
In the code fragment in the slide, you create a `JdbcRowSet` instance from `RowSetProviderFactory`.

You then treat the object like a `RowSet` JavaBean. You can use `setter` methods to set the `url`, `username`, and `password`, and then execute a SQL command and obtain a `ResultSet` to retrieve the column values.

This example is from the `SimpleJDBCRowSetExample` project.

Data Access Objects

Consider an employee table like the one in the sample JDBC code.



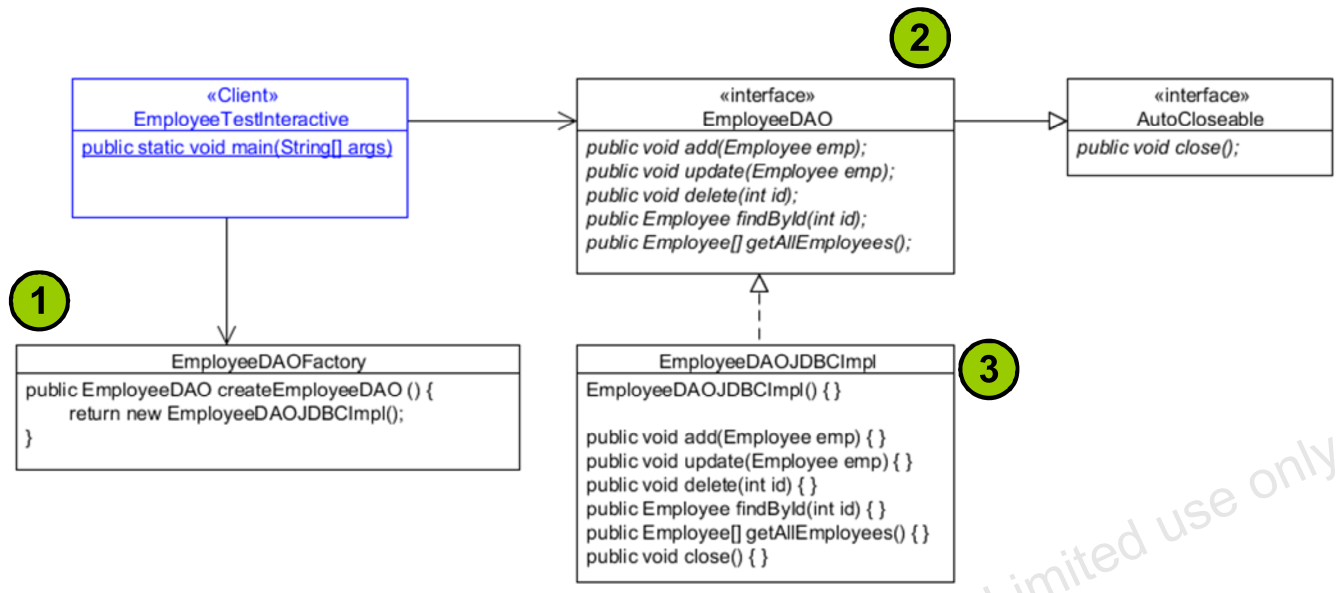
- By combining the code that accesses the database with the “business” logic, the data access methods and the Employee table are tightly coupled.
- Any changes to the table (such as adding a field) will require a complete change to the application.
- Employee data is not encapsulated within the example application.

The Employee Table

In the `SimpleJDBCExample` application shown in the previous slide, there is tight coupling between the operations used to access the data and the Employee table itself. Granted that the example is simple, but if you imagine this type of access in a larger application, perhaps with multiple tables with inter-table relationships, you can see how directly accessing the database in the same class as the business methods could create problems later if the Employee table were to change.

Further, because you are accessing the data directly, you do not have any way of passing the notion of an Employee around. You need to treat an Employee as an object.

The Data Access Object Pattern


2
«Client»
EmployeeTestInteractive
public static void main(String[] args)
«interface»
EmployeeDAO
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
«interface»
AutoCloseable
public void close();
1
EmployeeDAOFactory
public EmployeeDAO createEmployeeDAO () {
return new EmployeeDAOJDBCImpl();
}
EmployeeDAOJDBCImpl
EmployeeDAOJDBCImpl() { }
public void add(Employee emp) { }
public void update(Employee emp) { }
public void delete(int id) { }
public Employee findById(int id) { }
public Employee[] getAllEmployees() { }
public void close() { }
3

The Data Access Object and Factory Pattern

The purpose of a Data Access Object (DAO) is to separate database-related activities from the business model. In this design pattern, there are two techniques to insure future design flexibility.

1. A factory is used to generate instances (references) to an implementation of the `EmployeeDAO` interface. A factory makes it possible to insulate the developer using the DAO from the details about how a DAO implementation is instantiated. As you have seen, this same pattern was used to create an implementation where the data was stored in memory.
2. An `EmployeeDAO` interface is designed to model the behavior that you want to allow on the Employee data. Note that this technique of separating behavior from data demonstrates a *separation of concerns*. The `EmployeeDAO` interface encourages additional separation between the implementation of the methods required to support the DAO and references to `EmployeeDAO` objects.
3. The `EmployeeDAOJDBCImpl` implements the `EmployeeDAO` interface. The implementation class can be replaced with a different implementation without impacting the client application.

Summary

In this lesson, you should have learned how to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Use transactions with JDBC
- Use the JDBC 4.1 `RowSetProvider` and `RowSetFactory`
- Use a Data Access Object Pattern to decouple data and business methods

Quiz

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

a. `stmt.execute(query);`
b. `stmt.executeUpdate(query);`
c. `stmt.executeQuery(query);`
d. `stmt.query(query);`

ORACLE

Answer: b

a: `execute` returns a `boolean` (`true` if there is a ResultSet. This can be used with any SQL statement).
c: `executeQuery` returns a ResultSet (used with a `SELECT` statement).
d: This is not a valid `Statement` method.

Quiz

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

a. True

b. False

Answer: b (False)

A ResultSet's pointer is always pointing to just before the first row, regardless of whether it is one row or multiple rows.

Quiz

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

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

a. True

b. False

Answer: a (True)

This illustrates a good practice: explicitly closing the ResultSet in `try`-with-resources.

Quiz

Given:

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

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

a. Each `itemCount` value for customer Bob Smith
b. Compiler error
c. A run time error
d. A `SQLException`

ORACLE

Answer: c

Notice on line15, the `PreparedStatement` `setObject` method is called using the array index, 0, instead of 1 for the first parameter. To fix this code, you should replace line 15 with:
`pStmt.setObject(i+1, params[i]);`

Practice 14-2 Overview: Using the Data Access Object Pattern

This practice covers the following topics:

- Refactoring the memory-based DAO application to use JDBC.
- Using the interactive Employee client application, test your code.

In this practice, you will refactor the existing memory-based DAO from Exceptions and Assertions to use JDBC instead. An interactive client is provided so you can experiment with creating, reading, updating, and deleting records by using JDBC.

15

Localization

Objectives

After completing this lesson, you should be able to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle
- Format text for localization by using `NumberFormat` and `DateFormat`

Why Localize?

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

- Region- and language-aware software
- Dates, numbers, and currencies formatted for specific countries
- Ability to plug in country-specific data without changing code

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

In addition to language changes, culturally dependent elements, such as dates, numbers, currencies, and so on must be translated.

The goal is to design for localization so that no coding changes are required.

A Sample Application

Localize a sample application:

- Text-based user interface
- Localize menus
- Display currency and date localizations

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

In the remainder of this lesson, this simple text-based user interface will be localized for French, Simplified Chinese, and Russian. Enter the number indicated by the menu and that menu option will be applied to the application. Enter `q` to exit the application.

`Locale`

A `Locale` specifies a particular language and country:

- Language
  - An alpha-2 or alpha-3 ISO 639 code
  - “en” for English, “es” for Spanish
  - Always uses lowercase
- Country
  - Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code
  - "US" for United States, "ES" for Spain
  - Always uses uppercase
- See The Java Tutorials for details of all standards used

In Java, a locale is specified by using two values: language and country. See the Java Tutorial for standards used:
`http://download.oracle.com/javase/tutorial/i18n/locale/create.html`

Language Samples

- de: German
- en: English
- fr: French
- zh: Chinese

Country Samples

- DE: Germany
- US: United States
- FR: France
- CN: China

Resource Bundle

- The `ResourceBundle` class isolates locale-specific data:
  - Returns key/value pairs stored separately
  - Can be a class or a `.properties` file
- Steps to use:
  - Create bundle files for each locale.
  - Call a specific locale from your application.

Design for localization begins by designing the application so that all the text, sounds, and images can be replaced at run time with the appropriate elements for the region and culture desired. Resource bundles contain key/value pairs that can be hard-coded within a class or located in a `.properties` file.

Resource Bundle File

- Properties file contains a set of key/value pairs.
  - Each key identifies a specific application component.
  - Special file names use language and country codes.
- Default for sample application:
  - Menu converted into resource bundle

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

The slide shows a sample resource bundle file for this application. Each menu option has been converted into a name/value pair. This is the default file for the application. For alternative languages, a special naming convention is used:

```
MessageBundle_xx_YY.properties
```

where `xx` is the language code and `YY` is the country code.

Sample Resource Bundle Files

Samples for French and Chinese

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

```
MessagesBundle_zh_CN.properties
menu1 = 设置为英语
menu2 = 设置为法语
menu3 = 设置为中文
menu4 = 设置到俄罗斯
menu5 = 显示日期
menu6 = 显示我的钱!
menuq = 输入q退出
```

The slide shows the resource bundle files for French and Chinese. Note that the file names include both language and country. The English menu item text has been replaced with French and Chinese alternatives.

Quiz

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

a. `MessagesBundle_ES_US.properties`
b. `MessagesBundle_es_es.properties`
c. `MessagesBundle_es_US.properties`
d. `MessagesBundle_ES_us.properties`

Answer: c

Initializing the Sample Application

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

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

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

    // more init code here

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

With the resource bundles created, you simply need to load the bundles into the application. The source code in the slide shows the steps. First, create a `Locale` object that specifies the language and country. Then load the resource bundle by specifying the base file name for the bundle and the current `Locale`.

Note that there are a couple of ways to define a `Locale`. The `Locale` class includes default constants for some countries. If a constant is not available, you can use the language code with the country code to define the location. Finally, you can use the `getDefault()` method to get the default location.

Sample Application: Main Loop

```
public void run(){
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```

For this application, a run method contains the main loop. The loop runs until the letter "q" is typed in as input. A string switch is used to perform an operation based on the number entered. A simple call is made to each method to make locale changes and display formatted output.

The `printMenu` Method

Instead of text, resource bundle is used.

- `messages` is a resource bundle.
- A key is used to retrieve each menu item.
- Language is selected based on the `Locale` setting.

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

Instead of printing text, the resource bundle (`messages`) is called and the current `Locale` determines what language is presented to the user.

Changing the `Locale`

To change the `Locale`:

- Set `currentLocale` to the desired language.
- Reload the bundle by using the current locale.

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

After the menu bundle is updated with the correct locale, the interface text is output by using the currently selected language.

Sample Interface with French

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

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

The updated user interface is shown in the slide. The first and last lines of the application could be localized as well.

Format Date and Currency

- Numbers can be localized and displayed in their local format.
- Special format classes include:
  - `DateFormat`
  - `NumberFormat`
- Create objects using `Locale`.

Changing text is not the only available localization tool. Dates and numbers can also be formatted based on local standards.

Initialize Date and Currency

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

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

Date today = new Date();
DateFormat df;
```

Before any formatting can take place, date and number objects must be set up. Both today's date and a Double object are used in this application.

Displaying a Date

- Format a date:
  - Get a `DateFormat` object based on the `Locale`.
  - Call the `format` method passing the date to format.

```
public void showDate(){

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

- Sample dates:

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

Create a date format object by using the locale and the date is formatted for the selected locale.

Customizing a Date

- `DateFormat` constants include:
  - SHORT: Is completely numeric, such as 12.13.52 or 3:30pm
  - MEDIUM: Is longer, such as Jan 12, 1952
  - LONG: Is longer, such as January 12, 1952 or 3:30:32pm
  - FULL: Is completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST
- `SimpleDateFormat`:
  - A subclass of a `DateFormat` class

Letter	Date or Time	Presentation	Examples
G	Era	Text	AD
y	Year	Year	1996; 96
M	Month in Year	Month	July; Jul; 07

The `DateFormat` object includes a number of constants you can use to format the date output.

The `SimpleDateFormat` class is a subclass of `DateFormat` and allows you a great deal of control over the date output. See documentation for all the available options.

In some cases, the number of letters can determine the output. For example, with month:

```
MM 07
MMM Jul
MMMM July
```

Displaying Currency

- Format currency:
  - Get a currency instance from `NumberFormat`.
  - Pass the `Double` to the `format` method.

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

- Sample currency output:

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

Create a `NumberFormat` object by using the selected locale and get formatted output.

Quiz

Which date format constant provides the most detailed information?

```
a. LONG
b. FULL
c. MAX
d. COMPLETE
```

Answer: b

Summary

In this lesson, you should have learned how to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle
- Format text for localization by using `NumberFormat` and `DateFormat`

Practice 15-1 Overview: Creating a Localized Date Application

This practice covers creating a localized application that displays dates in a variety of formats.

(Optional) Practice 15-2 Overview: Localizing a JDBC Application

This practice covers creating a localized version of the JDBC application from the previous lesson.

A

SQL Primer

Objectives

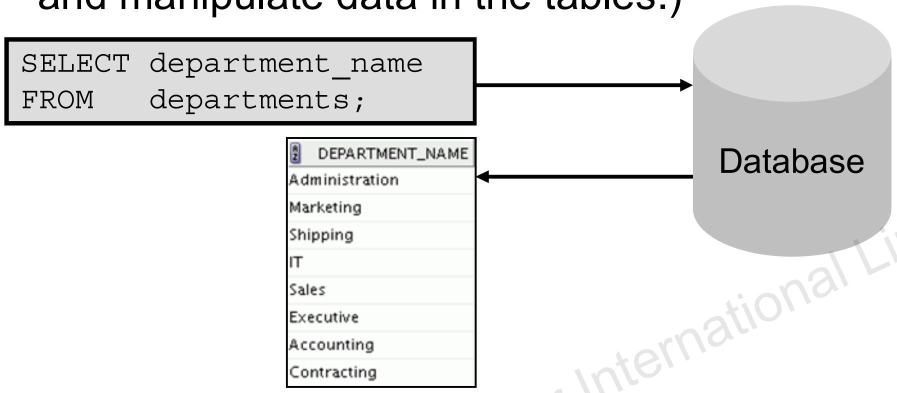
After completing this lesson, you should be able to:

- Describe the syntax of basic SQL-92/1999 commands, including:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - CREATE TABLE
  - DROP TABLE
- Define basic SQL-92/1999 data types

Using SQL to Query Your Database

Structured query language (SQL) is:

- The ANSI standard language for operating relational databases
- Efficient, easy to learn, and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)



```
SELECT  department_name
FROM    departments;
```

DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

In a relational database, you do not specify the access route to the tables, and you do not need to know how the data is arranged physically.

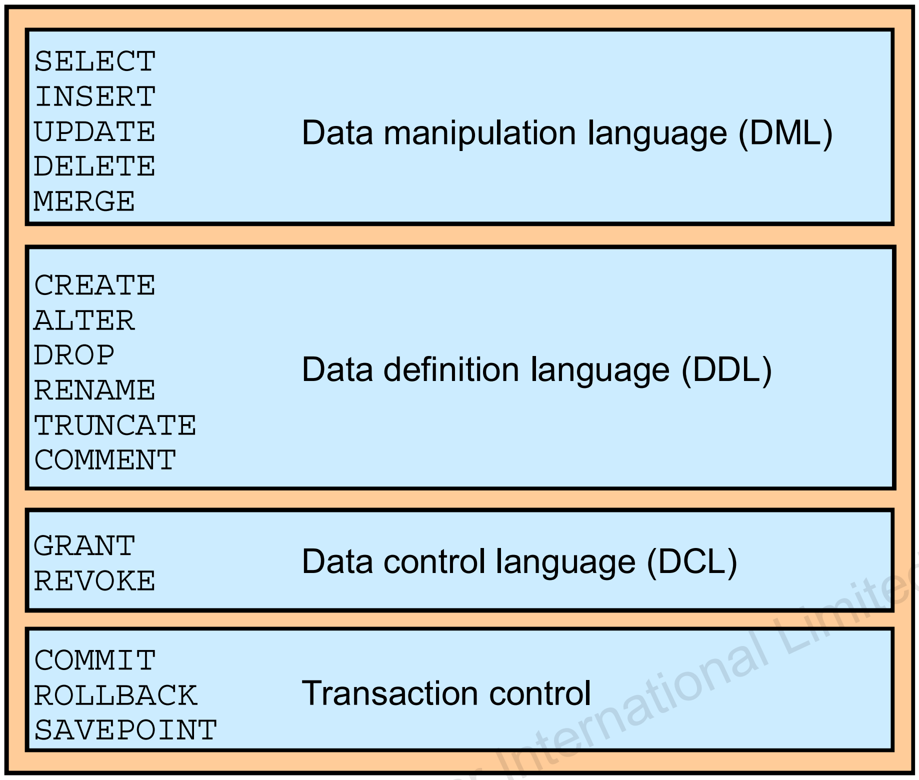
To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. SQL is a set of statements with which all programs and users access data in an Oracle Database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications, in turn, must use SQL when executing the user's request.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language and enables you to work with data at a logical level.

SQL Statements



`SELECT` `INSERT` `UPDATE` `DELETE` `MERGE`	Data manipulation language (DML)
`CREATE` `ALTER` `DROP` `RENAME` `TRUNCATE` `COMMENT`	Data definition language (DDL)
`GRANT` `REVOKE`	Data control language (DCL)
`COMMIT` `ROLLBACK` `SAVEPOINT`	Transaction control

SQL statements supported by Oracle comply with industry standards. Oracle Corporation ensures future compliance with evolving standards by actively involving key personnel in SQL standards committees. The industry-accepted committees are ANSI and International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

Statement	**Description**
`SELECT` `INSERT` `UPDATE` `DELETE` `MERGE`	Retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML)
`CREATE` `ALTER` `DROP` `RENAME` `TRUNCATE` `COMMENT`	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL)
`GRANT` `REVOKE`	Provides or removes access rights to both the Oracle Database and the structures within it
`COMMIT` `ROLLBACK` `SAVEPOINT`	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions

Basic SELECT Statement

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM    table;
```

- SELECT identifies the columns to be displayed.
- FROM identifies the table containing those columns.

In its simplest form, a SELECT statement must include the following:

- A SELECT clause, which specifies the columns to be displayed
- A FROM clause, which identifies the table containing the columns that are listed in the SELECT clause

In the syntax:

`SELECT`	Is a list of one or more columns
`*`	Selects all columns
`DISTINCT`	Suppresses duplicates
column|expression	Selects the named column or the expression
alias	Gives the selected columns different headings
`FROM` *table*	Specifies the table containing the columns

.

Note: Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A *keyword* refers to an individual SQL element—for example, `SELECT` and `FROM` are keywords.
- A *clause* is a part of a SQL statement—for example, `SELECT employee_id, last_name`, and so on.
- A *statement* is a combination of two or more clauses—for example, `SELECT * FROM employees`

Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the WHERE clause:

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- The WHERE clause follows the FROM clause.

You can restrict the rows that are returned from the query by using the WHERE clause. A WHERE clause contains a condition that must be met and it directly follows the FROM clause. If the condition is true, the row meeting the condition is returned.

In the syntax:

WHERE	Restricts the query to rows that meet a condition
condition	Is composed of column names, expressions, constants, and a comparison operator. A condition specifies a combination of one or more expressions and logical (Boolean) operators, and returns a value of TRUE, FALSE, or UNKNOWN.

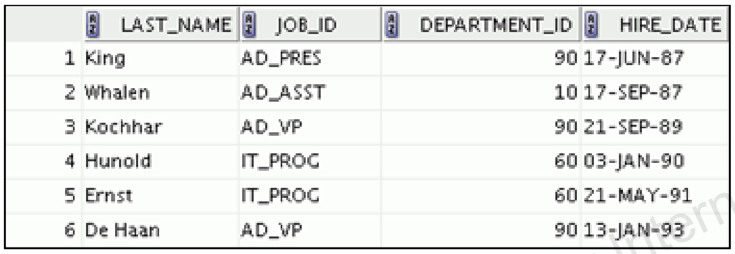
The WHERE clause can compare values in columns, literal, arithmetic expressions, or functions. It consists of three elements:

- Column name
- Comparison condition
- Column name, constant, or list of values

Using the ORDER BY Clause

- Sort the retrieved rows with the ORDER BY clause:
  - ASC: Ascending order, default
  - DESC: Descending order
- The ORDER BY clause comes last in the SELECT statement:

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```



	LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1	King	AD_PRES	90	17-JUN-87
2	Whalen	AD_ASST	10	17-SEP-87
3	Kochhar	AD_VP	90	21-SEP-89
4	Hunold	IT_PROG	60	03-JAN-90
5	Ernst	IT_PROG	60	21-MAY-91
6	De Haan	AD_VP	90	13-JAN-93

...

The order of rows that are returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. However, if you use the ORDER BY clause, it must be the last clause of the SQL statement. Further, you can specify an expression, an alias, or a column position as the sort condition.

Syntax

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY   {column, expr, numeric_position} [ASC|DESC]];
```

In the syntax:

ORDER BY	specifies the order in which the retrieved rows are displayed
ASC	orders the rows in ascending order (This is the default order.)
DESC	orders the rows in descending order

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.

Note: Use the keywords NULLS FIRST or NULLS LAST to specify whether returned rows containing null values should appear first or last in the ordering sequence.

INSERT Statement Syntax

- Add new rows to a table by using the `INSERT` statement:

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

You can add new rows to a table by issuing the `INSERT` statement.

In the syntax:

table	Is the name of the table
column	Is the name of the column in the table to populate
value	Is the corresponding value for the column

Note: This statement with the `VALUES` clause adds only one row at a time to a table.

`UPDATE` Statement Syntax

- Modify existing values in a table with the `UPDATE` statement:

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Update more than one row at a time (if required).

You can modify the existing values in a table by using the `UPDATE` statement.

In the syntax:

table	Is the name of the table
column	Is the name of the column in the table to populate
value	Is the corresponding value or subquery for the column
condition	Identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

Note: In general, use the primary key column in the `WHERE` clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the `EMPLOYEES` table by name is dangerous, because more than one employee may have the same name.

`DELETE` Statement

You can remove existing rows from a table by using the `DELETE` statement:

```
DELETE [FROM]   table
[WHERE          condition];
```

`DELETE` Statement Syntax

You can remove existing rows from a table by using the `DELETE` statement.

In the syntax:

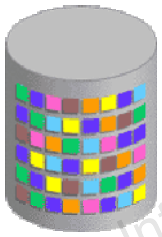
table	Is the name of the table
condition	Identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

`CREATE TABLE` Statement

- You must have:
  - The `CREATE TABLE` privilege
  - A storage area

```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

- You specify:
  - The table name
  - The column name, column data type, and column size



You create tables to store data by executing the SQL `CREATE TABLE` statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle Database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.

To create a table, a user must have the `CREATE TABLE` privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

In the syntax:

schema	Is the same as the owner's name
table	Is the name of the table
DEFAULT *expr*	Specifies a default value if a value is omitted in the `INSERT` statement
column	Is the name of the column
datatype	Is the column's data type and length

Defining Constraints

- Syntax:

```
CREATE TABLE [schema.]table
        (column datatype [DEFAULT expr]
        [column_constraint],
        ...
        [table_constraint][,...]);
```

- Column-level constraint syntax:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table-level constraint syntax:

```
column,...
   [CONSTRAINT constraint_name] constraint_type
   (column, ...),
```

The slide gives the syntax for defining constraints when creating a table. You can create constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.

`NOT NULL` constraints must be defined at the column level.

Constraints that apply to more than one column must be defined at the table level.

In the syntax:

`schema`	Is the same as the owner's name
`table`	Is the name of the table
`DEFAULT expr`	Specifies a default value to be used if a value is omitted in the `INSERT` statement
`column`	Is the name of the column
`datatype`	Is the column's data type and length
`column_constraint`	Is an integrity constraint as part of the column definition
`table_constraint`	Is an integrity constraint as part of the table definition

Defining Constraints

- Example of a column-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6)
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,
  first_name   VARCHAR2(20),
  ...);
```

1

- Example of a table-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6),
  first_name   VARCHAR2(20),
  ...
  job_id       VARCHAR2(10) NOT NULL,
  CONSTRAINT emp_emp_id_pk
    PRIMARY KEY (EMPLOYEE_ID));
```

2

ORACLE

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also be temporarily disabled.

Both examples in the slide create a primary key constraint on the `EMPLOYEE_ID` column of the `EMPLOYEES` table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

More details about the primary key constraint are provided later in this lesson.

Including Constraints

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - `NOT NULL`
  - `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`

Constraints

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

Data Integrity Constraints

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table.
CHECK	Specifies a condition that must be true

Data Types

Data Type	Description
`VARCHAR2(size)`	Variable-length character data
`CHAR(size)`	Fixed-length character data
`NUMBER(p,s)`	Variable-length numeric data
`DATE`	Date and time values
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Character data (up to 4 GB)
`RAW and LONG RAW`	Raw binary data
`BLOB`	Binary data (up to 4 GB)
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	A base-64 number system representing the unique address of a row in its table

Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	**Description**
`VARCHAR2(size)`	Variable-length character data (A maximum *size* must be specified: minimum *size* is 1; maximum *size* is 4,000.)
`CHAR [(size)]`	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.)
`NUMBER [(p,s)]`	Number having precision *p* and scale *s* (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 to 38, and scale can range from –84 to 127.)
`DATE`	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Character data (up to 4 GB)

Data Type	**Description**
`RAW(size)`	Raw binary data of length *size* (A maximum *size* must be specified: maximum *size* is 2,000.)
`LONG RAW`	Raw binary data of variable length (up to 2 GB)
`BLOB`	Binary data (up to 4 GB)
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	A base-64 number system representing the unique address of a row in its table

Guidelines

- A `LONG` column is not copied when a table is created using a subquery.
- A `LONG` column cannot be included in a `GROUP BY` or an `ORDER BY` clause.
- Only one `LONG` column can be used per table.
- No constraints can be defined on a `LONG` column.
- You might want to use a `CLOB` column rather than a `LONG` column.

Dropping a Table

- Moves a table to the recycle bin
- Removes the table and all its data entirely if the `PURGE` clause is specified
- Invalidates dependent objects and removes object privileges on the table

```
DROP TABLE dept80;
```

```
DROP TABLE dept80 succeeded.
```

ORACLE

The `DROP TABLE` statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the `PURGE` clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count toward the user's space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated
with it.

Syntax

`DROP TABLE` *`table`* `[PURGE]`

In the syntax, *`table`* is the name of the table.

Guidelines

- All the data is deleted from the table.
- Any views and synonyms remain, but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

Summary

In this lesson, you should have learned how to:

- Describe the syntax of basic SQL-92/1999 commands, including:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - CREATE TABLE
  - DROP TABLE
- Define basic SQL-92/1999 data types