Java SE 7 Programming

Student Guide

D67238GC20
Edition 2.0
June 2012
D74998

ORACLE

Chapter 1

Introduction

Course Goals

- ❑ This course covers the core APIs that you use to design object-oriented applications with Java. This course also covers writing database programs with JDBC.
- ❑ Use this course to further develop your skills with the Java language and prepare for the Oracle Certified Professional, Java SE 7 Programmer Exam.

Course Objectives

After completing this course, you should be able to do the following:

- ❑ Create Java technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- ❑ Execute a Java application from the command line
- ❑ Create applications that use the Collections framework
- ❑ Implement error-handling techniques using exception handling
- ❑ Implement input/output (I/O) functionality to read from and write to data and text files and understand advanced I/O streams
- ❑ Manipulate files, directories, and file systems using the JDK7 NIO.2 specification
- ❑ Perform multiple operations on database tables, including creating, reading, updating, and deleting, using the JDBC API
- ❑ Process strings using a variety of regular expressions
- ❑ Create high-performing multi-threaded applications that avoid deadlock
- ❑ Localize Java applications

Audience

The target audience includes those who have:

- ❑ Completed the *Java SE 7 Fundamentals* course or have experience with the Java language, and can create, compile, and execute programs
- ❑ Experience with at least one programming language
- ❑ An understanding of object-oriented principles
- ❑ Experience with basic database concepts and a basic knowledge of SQL

Prerequisites

To successfully complete this course, you must know how to:

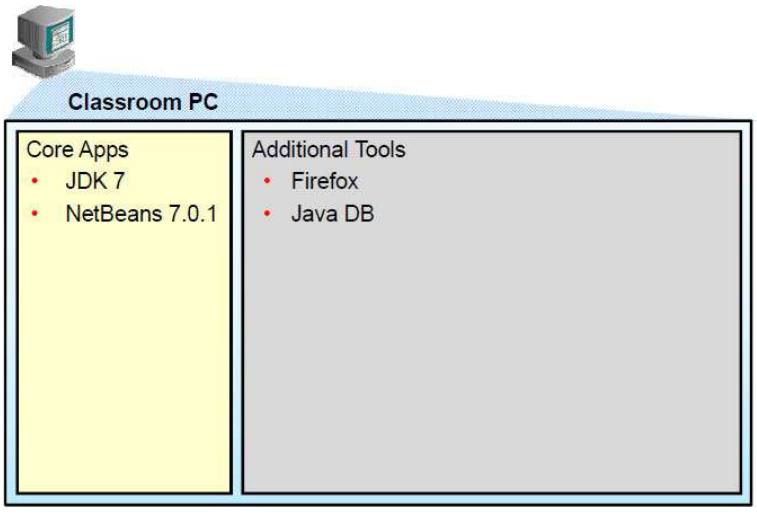
- ❑ Compile and run Java applications
- ❑ Create Java classes
- ❑ Create object instances using the `new` keyword
- ❑ Declare Java primitive and reference variables
- ❑ Declare Java methods using return values and parameters
- ❑ Use conditional constructs such as `if` and `switch` statements
- ❑ Use looping constructs such as `for`, `while`, and `do` loops
- ❑ Declare and instantiate Java arrays
- ❑ Use the Java Platform, Standard Edition API Specification (Javadocs)

Class Introductions

Briefly introduce yourself:

- ❑ Name
- ❑ Title or position
- ❑ Company
- ❑ Experience with Java programming and Java applications
- ❑ Reasons for attending

Course Environment

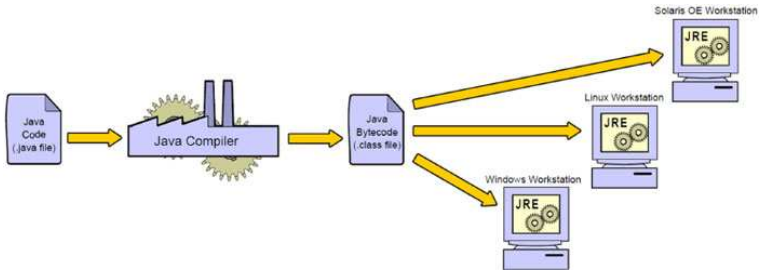


In this course, the following products are preinstalled for

the lesson practices:

- **JDK 7**: The Java SE Development Kit includes the command-line Java compiler (`javac`) and the Java Runtime Environment (JRE), which supplies the `java` command needed to execute Java applications.
- **Firefox**: A web browser is used to view the HTML documentation (Javadoc) for the Java SE Platform libraries.
- **NetBeans 7.0.1**: The NetBeans IDE is a free and open-source software development tool for professionals who create enterprise, web, desktop, and mobile applications. NetBeans 7.0.1 fully supports the Java SE 7 Platform. Support is provided by Oracle's Development Tools Support offering.
- **Java DB**: Java DB is Oracle's supported distribution of the open-source Apache Derby 100% Java technology database. It is fully transactional, secure, easy-to-use, standards-based SQL, JDBC API, and Java EE yet small, only 2.5 MB.

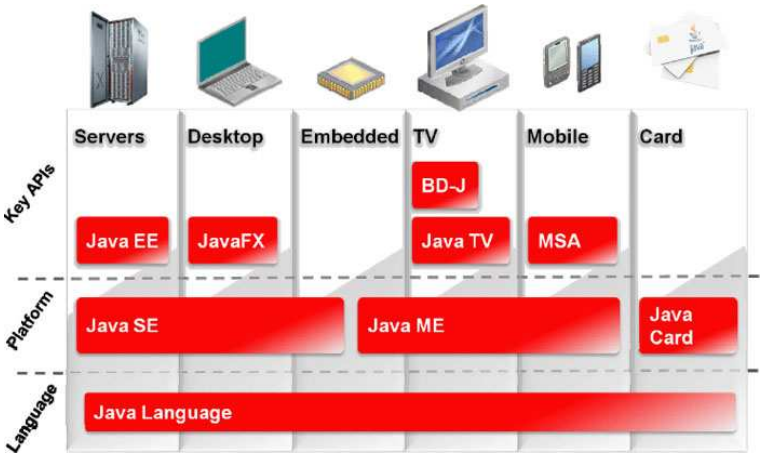
Java Programs are Platform-Independent



Platform-Independent Programs

Java technology applications are written in the Java programming language and compiled to Java bytecode. Bytecode is executed on the Java platform. The software that provides you with a runnable Java platform is called a Java Runtime Environment (JRE). A compiler, included in the Java SE Development Kit (JDK), is used to convert Java source code to Java bytecode.

Java Technology Product Groups



Identifying Java Technology Groups

Oracle provides a complete line of Java technology products ranging from kits that create Java technology programs to emulation (testing) environments for consumer devices, such as cellular phones. As indicated in the graphic, all Java technology products share the foundation of the Java language. Java technologies, such as the Java Virtual Machine, are included (in different forms) in three different groups of products, each designed to fulfill the needs of a particular target market. The figure illustrates the three Java technology product groups and their target device types. Among other Java technologies, each edition includes a Software Development kit (SDK) that allows programmers to create, compile, and execute Java technology programs on a particular platform:

- **Java Platform, Standard Edition (Java SE)**: Develops applets and applications that run within Web browsers and on desktop computers, respectively. For example, you can use the Java SE Software Development Kit (SDK) to create a word processing program for a personal computer. You can also use the Java SE to create an application that runs in a browser.

Note: Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, whereas applications are launched within an operating system.

Java SE Platform Versions

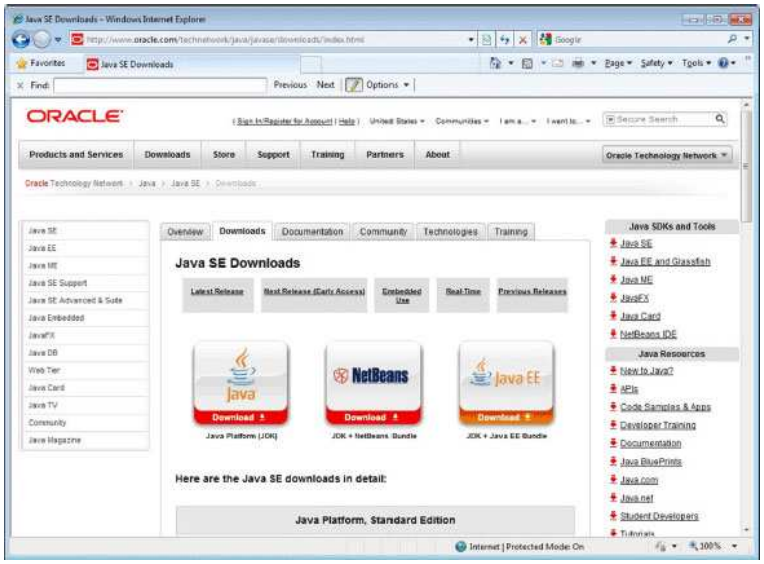
Year	Developer Version (JDK)	Platform
1996	1.0	1
1997	1.1	1
1998	1.2	2
2000	1.3	2
2002	1.4	2

2004	1.5	5
2006	1.6	6
2011	1.7	7

How to Detect Your Version

If Java SE is installed on your system, you can detect the version number by running `java -version`. Note that the `java` command is included with the Java Runtime Environment (JRE). As a developer, you also need a Java compiler, typically `javac`. The `javac` command is included in the Java SE Development Kit (JDK). Your operation system's `PATH` may need to be updated to include the location of `javac`.

Downloading and Installing the JDK



1. Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Choose the Java Platform, Standard Edition (Java SE) link.
3. Download the version that is appropriate for your operation system.
4. Follow the installation instructions.
5. Set your `PATH`.

Java in Server Environments

Java is common in enterprise environments:

- ❑ Oracle Fusion Middleware
  - Java application servers
    - GlassFish
    - WebLogic
- ❑ Database servers
  - MySQL
  - Oracle Database

Enterprise Environments

In this course, you develop Java SE applications. There are standard patterns you need to follow when implementing Java SE applications, such as always creating a `main` method that may be different when implementing enterprise applications. Java SE is only the starting point in your path to becoming a Java developer. Depending on the needs of your organization, you may be required to develop applications that run inside Java EE application servers or other types of Java middleware.

Often, you will also need to manipulate information stored inside relational databases such as MySQL or Oracle Database. This course introduces you to the fundamentals of database programming.

The Java Community

What Is the Java Community?

At a very high level, *Java Community* is the term used to refer to the many individuals and organizations that develop, innovate, and use Java technology. This community includes developers as individuals, organizations, businesses, and open-source projects. It is very common for you to download and use Java libraries from non-Oracle sources within the Java community. For instance, in this course, you use an Apache-developed JDBC library to access a relational database.

The Java Community Process (JCP)

The JCP is used to develop new Java standards:

- http://jcp.org
- Free download of all Java Specification Requests (JSRs)
- Early access to specifications
- Public review and feedback opportunities
- Open membership

JCP.next

The JCP produces the JSRs that outline the standards of the Java platform. The behavior of the JCP itself is also defined and improved through the JSR process. The JCP is evolving and its improvements are defined in JSR-348. JSR-348 introduces changes in the areas of transparency, participation, agility, and governance.

- **Transparency**: In the past, some aspects of the development of a JSR may have occurred behind closed doors. Transparent development is now the recommended practice.
- **Participation**: Individuals and Java User Groups are encouraged to become active in the JCP.
- **Agility**: Slow-moving JSRs are now actively discouraged.
- **Governance**: The SE and ME expert groups are merging into a single body.

OpenJDK

OpenJDK is the open-source implementation of Java:

- http://openjdk.java.net/
- GPL licensed open-source project
- JDK reference implementation
- Where new features are developed
- Open to community contributions
- Basis for Oracle JDK

Why OpenJDK Is Important

Because it is open source, OpenJDK enables users to port Java to operating systems and hardware platforms of their choosing. Ports are underway for many platforms (besides those already supported) including FreeBSD, OpenBSD, NetBSD, and MacOS X.

Oracle Java SE Support

Java is available free of charge. However, Oracle does provide pay-for Java solutions:

- The Java SE Support Program provides updates for end-of-life Java versions.
- Oracle Java SE Advanced and Oracle Java SE Suite:
  - JRockit Mission Control
  - Memory Leak Detection
  - Low Latency GC (Suite)
  - JRockit Virtual Edition (Suite)

Still Free

Java (Oracle JDK) is freely available at no cost. Oracle offers advanced commercial solutions at cost. The previously offered "Java for Business" program has been replaced by Oracle Java SE Support, which provides access to Oracle Premier Support and the Oracle Java SE Advanced and Oracle Java SE Suite binaries. For more information, visit http://www.oracle.com/us/technologies/java/java-se-suite-394230.html.

Additional Resources

Topic	Website
Education and Training	http://education.oracle.com
Product Documentation	http://www.oracle.com/technology/documentation
Product Downloads	http://www.oracle.com/technology/software
Product Articles	http://www.oracle.com/technology/pub/articles

Product Support	http://www.oracle.com/support
Product Forums	http://forums.oracle.com
Product Tutorials	http://www.oracle.com/technetwork/tutorials/index.html
Sample Code	https://www.samplecode.oracle.com
Oracle Technology Network for Java Developers	http://www.oracle.com/technetwork/java/index.html
Oracle Learning Library	http://www.oracle.com/goto/oll

The table in the slide lists various web resources that are available for you to learn more about Java SE programming.

Summary

In this lesson, you should have learned about:

- The course objectives
- Software used in this course
- Java platforms (ME, SE, and EE)
- Java SE version numbers
- Obtaining a JDK
- The open nature of Java and its community
- Commercial support options for Java SE

Chapter 2

Java Syntax and Class Review

Objectives

After completing this lesson, you should be able to do the following:

- Create simple Java classes
  - Create primitive variables
  - Manipulate Strings
  - Use `if-else` and `switch` branching statements
  - Iterate with loops
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements

Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Basic `if-else` and `switch` branching constructs
- Iteration with `for` and `while` loops

Class Structure

```
package <package_name>;

import <other_packages>;

public class ClassName {
   <variables(also known as fields)>;

   <constructor method(s)>;

   <other methods>;
}
```

A Java class is described in a text file with a `.java` extension. In the example shown, the Java keywords are highlighted in bold.

- The `package` keyword defines where this class lives relative to other classes, and provides a level of access control. You use access modifiers (such as `public` and `private`) later in this lesson.
- The `import` keyword defines other classes or groups of classes that you are using in your class. The `import` statement helps to narrow what the compiler needs to look for when resolving class names used in this class.
- The `class` keyword precedes the name of this class. The name of the class and the file name must match when the class is declared `public` (which is a good practice). However, the keyword `public` in front of the `class` keyword is a modifier and is not required.
- Variables, or the data associated with programs (such as integers, strings, arrays, and references to other objects), are called *instance fields* (often shortened to *fields*).
- Constructors are functions called during the creation (instantiation) of an object (a representation in memory of a Java class.)
- Methods are the functions that can be performed on an object. They are also referred to as *instance methods*.

A Simple Class

A simple Java class with a main method:

```
public class Simple {

    public static void main(String args[]){

    }
}
```

To run a Java program, you must define a `main` method as shown in the slide. The `main` method is automatically called when the class is called from the command line. Command-line arguments are passed to the program through the `args[]` array.

Note: A method that is modified with the keyword `static` is invoked without a reference to a particular object. The class name is used instead. These methods are referred to as *class methods*. The `main` method is a special method that is invoked when this class is run using the Java runtime.

Code Blocks

- Every class declaration is enclosed in a code

block.

- Method declarations are enclosed in code blocks.
- Java fields and methods have block (or class) scope.
- Code blocks are defined in braces:

```
{ }
```

- Example:

```
public class SayHello {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Java fields (variables) and methods have a class scope defined by the opening left curly brace and ending at the closing right curly brace.

Class scope allows any method in the class to call or invoke any other method in the class. Class scope also allows any method to access any field in the class.

Code blocks are always defined using braces `{}`. A block is executed by executing each of the statements defined within the block in order from first to last (left to right).

The Java compiler ignores white space that precedes or follows the elements that make up a line of code. Line indentation is not required but makes code much more readable. In this course, the line indentation is four spaces, which is the default line indentation used by the NetBeans IDE.

Primitive Data Types

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 07 0xff	3.0F .3337F 4.022E23	'a' '\u0061' '\n'	true false
0	0.0	'\u0000'	false

Append uppercase or lowercase "`L`" or "`F`" to the number to specify a long or a float number.

Integer

Java provides four different integer types to accommodate different size numbers. All the numeric types are signed, which means that they can hold positive or negative numbers. The integer types have the following ranges:

- `byte` range is –128 to +127. Number of bits = 8.
- `short` range is –32,768 to +32,767. Number of bits = 16.
- `int` range is –2,147,483,648 to +2,147,483,647. The most common integer type is `int`. Number of bits = 32.
- `long` range is –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. Number of bits = 64.

Floating Point

The floating-point types hold numbers with a fractional part and conform to the IEEE 754 standard. There are two types of floating points: `float` and `double`.

`double` is so called because it provides double the precision of `float`. A `float` uses 32 bits to store data, whereas a `double` uses 64 bits.

Character

The `char` type is used for individual characters, as opposed to a string of characters (which is implemented as a `String` object). Java supports Unicode, an international standard for representing a character in any written language in the world in a single 16-bit value. The first 256 characters coincide with the ISO Latin 1 character set, part of which is ASCII.

Boolean

The `boolean` type can hold either `true` or `false`.

Note: `true` and `false` may appear to be keywords, but they are technically boolean literals.

Default Values

If a value is not specified, a default value is used. The values in red in the slide are the defaults used. The default `char` value is `null` (represented as `'\u0000'`), and the default value for `boolean` is `false`.

Note: Local variables (that is, variables declared within methods) do not have a default value. An attempt to use a local variable that has not been assigned a value will cause a compiler error. It is a good practice always to supply a default value to any variable.

Java SE 7 Numeric Literals

In Java SE 7 (and later versions), any number of underscore characters (_) can appear between digits in a numeric field. This can improve the readability of your code.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Rules for Literals

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an `F` or `L` suffix
- In positions where a string of digits is expected

Note: The Java language is case-sensitive. In Java, the variable `creditCardNumber` is different from `CREDITCARDNUMBER`. Convention indicates that Java variables and method names use "lower camel case"—lowercase for the first letter of the first element of a variable name and uppercase for the first letter of subsequent elements.

Java SE 7 Binary Literals

In Java SE 7 (and later versions), binary literals can also be expressed using the binary system by adding the prefixes `0b` or `0B` to the number:

```
// An 8-bit 'byte' value:
byte aByte = 0b0010_0001;

// A 16-bit 'short' value:
short aShort = (short)0b1010_0001_0100_0101;

// Some 32-bit 'int' values:
int anInt1 = 0b1010_0001_0100_0101_1010_0001_0100_0101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.
```

Binary literals are Java `int` values. A cast is required when the integer value of the literal exceeds the greatest non-negative value that the type can hold. For example:

```
byte aByte = 0b0111_1111; // aByte is 127
byte aByte = 0b1000_0000; // compiler error - a cast is required // (value is -128)
```

Operators

- Simple assignment operator
  = Simple assignment operator
- Arithmetic operators
  + Additive operator (also used for String concatenation)
  – Subtraction operator
  * Multiplication operator
  / Division operator
  % Remainder operator
- Unary operators
  + Unary plus operator; indicates positive
  – Unary minus operator; negates an expression
  ++ Increment operator; increments a value by 1
  – – Decrement operator; decrements a value by 1
  ! Logical complement operator; inverts the value of a Boolean

Because numbers have been introduced, the slide shows a list of common operators. Most are common to any programming language, and a description of each is provided in the slide.

The binary and bitwise operators have been omitted for brevity. For details about those operators, refer to the Java Tutorial:
http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Note: Operators have definitive precedence. For the complete list, see the Java Tutorial link mentioned above. Precedence can be overridden using parentheses.

Strings

```
1 public class Strings {
2
3     public static void main(String args[]){
4
5         char letter = 'a';
6
7         String string1 = "Hello";
8         String string2 = "World";
9         String string3 = "";
10        String dontDoThis = new String ("Bad Practice");
11
12        string3 = string1 + string2; // Concatenate strings
13
14        System.out.println("Output: " + string3 + " " + letter);
15
16    }
17 }
```

String literals are also String objects.

The code in the slide demonstrates how text characters are represented in Java. Single characters can be represented with the `char` type. However, Java also includes a `String` type for representing multiple characters. Strings can be defined as shown in the slide and combined using the "+" sign as a concatenation operator.

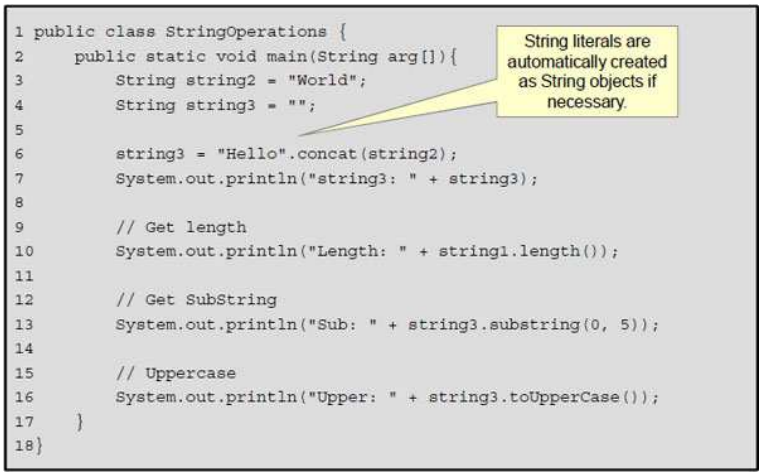
The output from the code in the slide is:

```
Output: HelloWorld a
```

Caution: Strings should always be initialized using the assignment operator "=" and text in quotation marks, as

shown in the examples. The use of `new` to initialize a `String` is strongly discouraged. The reason is that "`Bad Practice`" in line 10 is a `String` literal of type `String`, Using the `new` keyword simply creates another instance functionally identical to the literal. If this statement appeared inside of a loop that was frequently invoked, there could be a lot of needless `String` instances created.

String Operations




```
1 public class StringOperations {
2     public static void main(String arg[]){
3         String string2 = "World";
4         String string3 = "";
5
6         string3 = "Hello".concat(string2);
7         System.out.println("string3: " + string3);
8
9         // Get length
10        System.out.println("Length: " + string1.length());
11
12        // Get SubString
13        System.out.println("Sub: " + string3.substring(0, 5));
14
15        // Uppercase
16        System.out.println("Upper: " + string3.toUpperCase());
17    }
18}
```

This slide demonstrates some common string methods, including:

- `concat()`
- `length()`
- `substring()`
- `toUpperCase()`

To see what other methods can be used on a `String`, see the API documentation. The output from the program is:

```
string3: HelloWorld
Length: 5
Sub: Hello
Upper: HELLOWORLD
```

Note: `String` is a class, not a primitive type. Instances of the class `String` represent sequences of Unicode characters. `String` literals are stored as `String` objects and "interned", meaning that for strings with matching characters, they all point to the same `String` object.

if else

```
1 public class IfElse {
2
3     public static void main(String args[]){
4         long a = 1;
5         long b = 2;
6
7         if (a == b){
8             System.out.println("True");
9         } else {
10            System.out.println("False");
11        }
12
13    }
14 }
```

The example in the slide demonstrates the syntax for an `if-else` statement in Java. The output from the code in the slide is as follows:

```
False
```

Logical Operators

- Equality and relational operators
  == Equal to
  != Not equal to
  > Greater than
  >= Greater than or equal to
  < Less than
  <= Less than or equal to
- Conditional operators
  && Conditional-AND
  || Conditional-OR
  ?: Ternary (shorthand for `if-then-else` statement)
- Type comparison operator
  `instanceof` Compares an object to a specified type

The slide shows a summary of the logic and conditional operators in Java.

Arrays and `for-each` Loop

```
1 public class ArrayOperations {
2     public static void main(String args[]){
3
4         String[] names = new String[3];
5
6         names[0] = "Blue Shirt";
7         names[1] = "Red Shirt";
8         names[2] = "Black Shirt";
9
10         int[] numbers = {100, 200, 300};
11
12         for (String name:names){
13             System.out.println("Name: " + name);
14         }
15
16         for (int number:numbers){
17             System.out.println("Number: " + number);
18         }
19     }
20 }
```

Arrays are objects. Array objects have a final field length.

This class demonstrates how to define arrays in Java. The first example creates a `String` array and initializes each element separately. The second `int` array is defined in a single statement.

Each array is iterated through using the Java `for-each` construct. The loop defines an element which will represent each element of the array and the array to loop through. The output of the class is shown here:

```
Name: Blue Shirt
Name: Red Shirt
Name: Black Shirt
Number: 100
Number: 200
Number: 300
```

Note: Arrays are also objects by default. All arrays support the methods of the class Object. You can always obtain the size of an array using its `length` field.

`for` Loop

```
1 public class ForLoop {
2
3     public static void main(String args[]){
4
5         for (int i = 0; i < 9;  i++ ){
6             System.out.println("i: " + i);
7         }
8
9     }
10 }
```

The classic `for` loop is shown in the slide. A counter is initialized and incremented for each step of the loop. When the condition statement evaluates to false (when `i` is no longer less than 9), the loop exits. Here is the sample output for this program.

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
```

`while` Loop

```
1 public class WhileLoop {
2
3     public static void main(String args[]){
4
5         int i = 0;
6         int[] numbers = {100, 200, 300};
7
8         while (i < numbers.length ){
9             System.out.println("Number: " + numbers[i]);
10             i++;
11         }
12     }
13 }
```

The `while` loop performs a test and continues if the expression evaluates to `true`. The `while` loop, shown here, iterates through an array using a counter. Here is the output from the code in the slide:

```
Number: 100
Number: 200
Number: 300
```

Note: There is also a `do-while` loop, where the test after the expression has run at least once:

```
class DoWhileDemo {
  public static void main(String[]
args){
     int count = 1;
     do {
       System.out.println("Count is: " +
count);
       count++;
     } while (count < = 11);
   }
}
```

String `switch` Statement

```
1 public class SwitchStringStatement {
2    public static void main(String args[]){
3
4        String color = "Blue";
5        String shirt = " Shirt";
6
7        switch (color){
8            case "Blue":
9                shirt = "Blue" + shirt;
10               break;
11           case "Red":
12               shirt = "Red" + shirt;
13               break;
14           default:
15               shirt = "White" + shirt;
16       }
17
18       System.out.println("Shirt type: " + shirt);
19   }
20 }
```

This example shows a `switch` statement in Java using a `String`. Prior to version 7 of Java, only enums and `byte`, `short`, `char`, and `int` primitive data types could be used in a `switch` statement. You will see enums in the lesson titled "Advanced Class Design."

Java Naming Conventions

```
1 public class CreditCard {
2     public final int VISA_ = 5001;
3     public String accountName;
4     public String cardNumber;
5     public Date expDate;
6
7     public double getCharges(){
8         // ...
9     }
10
11    public void disputeCharge(String chargeId, float amount){
12        // ...
13    }
14}
```

Class names are nouns in upper camel case.

Constants should be declared in all uppercase. letters

Variable names are short but meaningful in lower camel case.

Methods should be verbs, in lower camel case.

- Class names should be nouns in mixed case, with the first letter uppercase and the first letter of each internal word capitalized. This is approach is termed "upper camel case."
- Methods should be verbs in mixed case, with the first letter lowercase and the first letter of each internal word capitalized. This is termed "lower camel case."
- Variable names should be short but meaningful. The choice of a variable name should be mnemonic: designed to indicate to the casual observer the intent of its use.
- One-character variable names should be avoided except as temporary "throwaway" variables.
- Constants should be declared using all uppercase letters.

Note: The keyword `final` is used to declare a variable whose value may only be assigned once. Once a `final` variable has been assigned, it always contains the same value. You will learn more about the keyword `final` in the lesson "Advanced Class Design."

For the complete *Code Conventions for the Java Programming Language* document, go to http://www.oracle.com/technetwork/java/codeconv-138413.html.

A Simple Java Class: `Employee`

A Java class is often used to represent a concept.

```
1 package com.example.domain;
2 public class Employee {  class declaration
3     public int empId;
4     public String name;
5     public String ssn;
6     public double salary;
7
8     public Employee () {   a constructor
9     }
10
11     public int getEmpId () {    a method
12       return empId;
13    }
14 }
```

fields

A Java class is often used to store or represent data for the construct that the class represents. For example, you could create a model (a programmatic representation) of an Employee. An `Employee` object defined using this model will contain values for `empId`, `name`, Social Security Number (`ssn`), and `salary`.

The constructor in this class creates an instance of an object called `Employee`.

A constructor is unique in Java. A constructor is used to create an instance of a class. Unlike methods, constructors do not declare a return type, and are declared with the same name as their class. Constructors can take arguments and you can declare more than one constructor, as you will see in the lesson titled "Java Class Design."

Methods

When a class has data fields, a common practice is to provide methods for storing data (setter methods) and retrieving data (getter methods) from the fields.

```
1 package com.example.domain;
2 public class Employee {
3     public int empId;
4     // other fields...
5     public void setEmpId(int empId) {
6         this.empId = empId;
7     }
8     public int getEmpId() {
9         return empId;
10    }
11    // getter/setter methods for other fields...
12 }
```

Often a pair of methods to set and get the current field value.

Adding Instance Methods to the `Employee` Class

A common practice is to create a set of methods that manipulate field data: methods that set the value of each field, and methods that get the value of each field. These methods are called *accessors* (getters) and *mutators* (setters).

The convention is to use `set` and `get` plus the name of the field with the first letter of the field name capitalized (lower camel case). Most modern integrated development environments (IDEs) provide an easy way to automatically generate the accessor (getter) and mutator (setter) methods for you.

Notice that the set methods use the keyword `this`. The `this` keyword allows the compiler to distinguish between the field name of the class (`this`) and the parameter name being passed in as an argument. Without the keyword `this`, the net effect is you are assigning a value to itself. (In fact, NetBeans provides a warning: "Assignment to self.")

In this simple example, you could use the `setName` method to change the employee name and the `setSalary` method to change the employee `salary`.

Note: The methods declared on this slide are called *instance* methods. They are invoked using an instance of this class (described on the next slide.)

Creating an Instance of an Object

To construct or create an instance (object) of the `Employee` class, use the `new` keyword.

```
/* In some other class, or a main method */
Employee emp = new Employee();
emp.empId = 101;    // legal if the field is public,
                    // but not good OO practice
emp.setEmpId(101); // use a method instead
emp.setName("John Smith");
emp.setSsn("011-22-3467");
emp.setSalary(120345.27);
```

Invoking an instance method.

- In this fragment of Java code, you construct an instance of the `Employee` class and assign the reference to the new object to a variable called `emp`.
- Then you assign values to the `Employee` object.

Creating an Instance of the Employee Class

In order to use the `Employee` class to hold the information of an employee, you need to allocate memory for the `Employee` object and call a constructor method in the class. An instance of an object is created when you use the `new` keyword with a constructor. All of the fields declared in the class are provided memory space and initialized to their default values. If the memory allocation and constructor are successful, a reference to the object is returned as a result. In the example in the slide, the reference is assigned to a variable called `emp`.

To store values (data) into the `Employee` object instance, you could just assign values to each field by accessing the fields directly. However, this is not a good practice and negates the principle of encapsulation. Instead, you should invoke instance methods and pass a value to the method to set the value of each data field. Later in this lesson you will look at restricting access to the fields to promote encapsulation.

Once all the data fields are set with values, you have an instance of an `Employee` with an `empId` with a value of 101, `name` with the string John Smith, Social Security number string (`ssn`) set to 011-22-3467, and `salary` with the value of 120,345.27.

Constructors

```
public class Employee {
    public Employee() {
    }
}
```

A simple no-argument (no-arg) constructor.

```
Employee emp = new Employee();
```

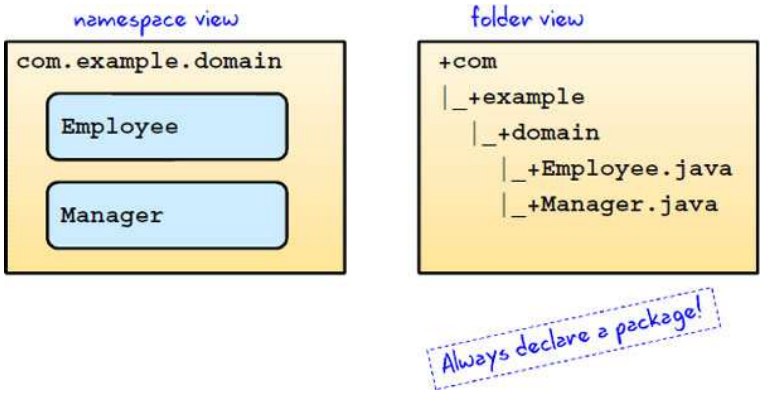
- A constructor is used to create an instance of a class.
- Constructors can take parameters.
- A constructor that takes no arguments is called a *no-arg* constructor.

A constructor is used to create an object. In the Java programming language, constructors are declared with the same name as their class used to create an instance of an object. Constructors are invoked using the `new` keyword.

Constructors are covered in more detail in the lesson titled "Encapsulation and Subclassing."

`package` Statement

The `package` keyword is used in Java to group classes together. A package is implemented as a folder and, like a folder, provides a *namespace* to a class.



Packages

In Java, a package is a group of (class) types. There can be only one `package` declaration for a file.

Packages are more than just a convenience. Packages create a namespace, a logical collection of things, like a directory hierarchy.

It is a good practice to always use a `package` declaration. The `package` declaration is always at the top of the file.

`import` Statements

The `import` keyword is used to identify classes you want to reference in your class.

- The `import` statement provides a convenient way to identify classes that you want to reference in your class.

```
import java.util.Date;
```

- You can import a single class or an entire package:

```
import java.util.*;
```

- You can include multiple `import` statements:

```
import java.util.Date;
import java.util.Calendar;
```

- It is good practice to use the full package and class name rather than the wildcard `*` to avoid class name conflicts.

Imports

You could refer to a class using its fully qualified namespace in your applications, as in the following example:

```
java.util.Date date = new java.util.Date();
```

But that would quickly lead to a lot of typing! Instead, Java provides the import statement to allow you to declare that you want to reference a class in another package.

Note: It is a good practice to use the specific, fully qualified package and class name to avoid confusion when there are two classes with the same name, as in the following example: `java.sql.Date` and `java.util.Date`. The first is a `Date` class used to store a `Date` type in a database, and `java.util.Date` is a general purpose `Date` class. As it turns out, `java.sql.Date` is a subclass of `java.util.Date`. This is covered in more detail later in the course.

Note: Modern IDEs, like NetBeans and Eclipse, automatically search for and add import statements for you. In NetBeans, for example, use the Ctrl + Shift + I key sequence to fix imports in your code.

More on `import`

- Import statements follow the package declaration and precede the class declaration.
- An import statement is not required.
- By default, your class always imports `java.lang.*`
- You do not need to import classes that are in the same package:

```
package com.example.domain;
import com.example.domain.Manager; // unused import
```

Details about the `java.lang` package and its classes are covered later in the course.

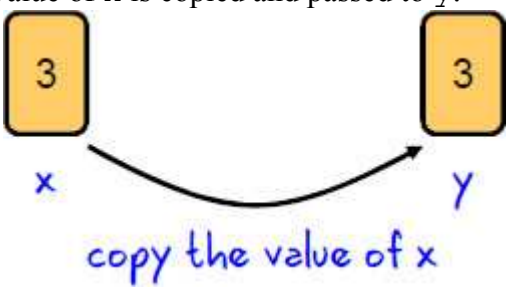
Java is Pass-By-Value

The Java language (unlike C++) uses pass-by-value for all assignment operations.

- To visualize this with primitives, consider the following:

```
int x = 3;
int y = x;
```

- The value of `x` is copied and passed to `y`:



- If `x` is later modified (for example, `x = 5;`), the value of `y` remains unchanged.

The Java language uses pass-by-value for all assignment operations. This means that the argument on the right side of the equal sign is evaluated, and the value of the argument is assigned to the left side of the equal sign.

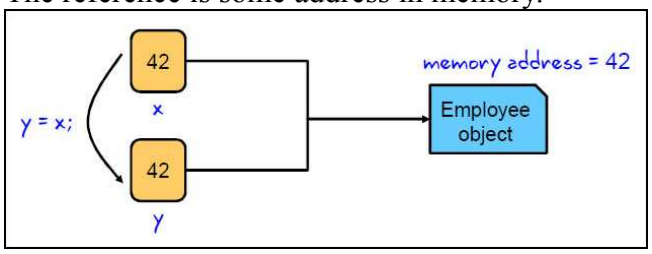
For Java primitives, this is straightforward. Java does not pass a reference to a primitive (such as an integer), but rather a copy of the value.

Pass-By-Value for Object References

For Java objects, the *value* of the right side of an assignment is a reference to memory that stores a Java object.

```
Employee x = new Employee();
Employee y = x;
```

- The reference is some address in memory.



- After the assignment, the value of `y` is the same as the value of `x`: a reference to the same `Employee` object.

For Java objects, the value of an object reference is the memory pointer to the instance of the `Employee` object created.

When you assign the value of `x` to `y`, you are not creating a new `Employee` object, but rather a copy of the value of the reference.

Note: An object is a class instance or an array. The reference values (references) are pointers to these objects, and a special `null` reference, which refers to no object.

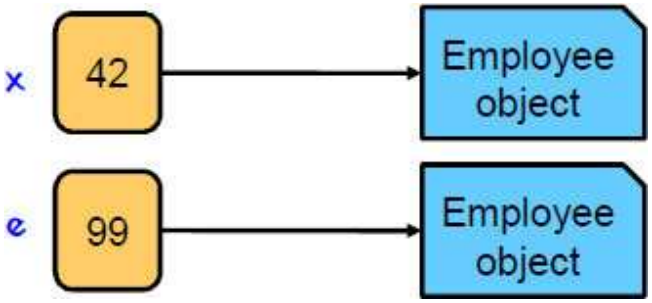
Objects Passed as Parameters

- Whenever a new object is created, a new reference is created. Consider the following code fragments:

```
Employee x = new Employee();
foo(x);
```

```
public void foo(Employee e) {
    e = new Employee();
    e.setSalary (1_000_000.00);  // What happens to x here?
}
```

- The value of `x` is unchanged as a result of the method call `foo`:



In the first line of code, a new object (Employee) is created and the reference to that object is assigned to the variable `x`.

In the second line of code, the value of that reference is passed to a method called `foo`.

When the `foo` method is called, (`Employee e`) holds a reference to the Employee object, `x`. In the next line, the value of `e` is now a new Employee object, by virtue of the call to the constructor.

The reference to the `x` object is replaced by a reference to a new object. The `x` object remains unchanged.

Note: The object `e`, created inside of the method `foo`, can no longer be referenced when the method finishes. As a result, it will be eligible for garbage collection at some future point.

If the code in the `foo` method was written differently, like this:

```
public void foo(Employee e) {
  e.setSalary(1_000_000.00):
}
```

Then referenced object that the `setSalary` method is being called on is the object referenced by `x`, and after the `foo` method returns, the object `x` is modified.

Note: The memory locations 42 and 99 are simply for illustrative purposes!

How to Compile and Run

Java class files must be compiled before running them.

To compile a Java source file, use the Java compiler (`javac`).

```
javac -cp <path to other classes> -d <complier output
path> <path to source>.java
```

- You can use the `CLASSPATH` environment variable to the directory above the location of the package hierarchy.
- After compiling the source `.java` file, a `.class` file is generated.
- To run the Java application, run it using the Java interpreter (`java`):

```
java -cp <path to other classes> <package
name>.<classname>
```

CLASSPATH

The default value of the `classpath` is the current working directory (.), however, specifying the `CLASSPATH` variable or the `-cp` command line switch overrides this value.

The `CLASSPATH` variable is used by both the Java compiler and the Java interpreter (runtime).

The `classpath` can include:

- A list of directory names (separated by semicolons in Windows and colons in UNIX)
  - The classes are in a package tree relative to any of the directories on the list.
- A `.zip` or `.jar` file name that is fully qualified with its path name
  - The classes in these files must be zipped with the path names that are derived from the directories formed by their package names.

Note: The directory containing the root name of the package tree must be added to the `classpath`. Consider putting `classpath` information in the command window or even in the Java command, rather than hard-coding it in the environment.

Compiling and Running: Example

- Assume that the class shown in the notes is in the directory `D:\test\com\example`:

```
javac -d D:\test D:\test\com\example\HelloWorld.java
```

- To run the application, you use the interpreter and the fully qualified class name:

```
java -cp D:\test com.example.HelloWorld
Hello World!

java -cp D:\test com.example.HelloWorld Tom
Hello Tom!
```

- The advantage of an IDE like NetBeans is that management of the class path, compilation, and running the Java application are handled through the tool.

Example

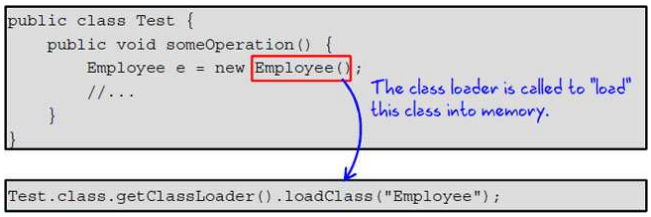
Consider the following simple class in a file named `HelloWorld.java` in the `D:\test\com\example` directory:

```
package com.example;
  public class HelloWorld {
    public static void main (String []
args) {
      if (args.length < 1) {
        System.out.println("Hello
World!");
      } else {
        System.out.println("Hello " +
args[0] + "!");
      }
    }
}
```

Java Class Loader

During execution of a Java program, the Java Virtual Machine loads the compiled Java class files using a Java class of its own called the "class loader" (`java.lang.ClassLoader`).

- The class loader is called when a class member is used for the first time:

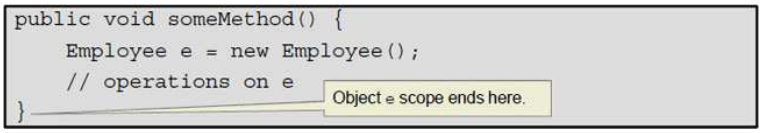


Typically, the use of the class loader is completely invisible to you. You can see the results of the class loader by using the `-verbose` flag when you run your application. For example:

```
java -verbose -classpath D:\test
com.example.HelloWorld
[Loaded java.lang.Object from shared
objects file]
[Loaded java.io.Serializable from
shared objects file]
[Loaded java.lang.Comparable from
shared objects file]
[Loaded java.lang.CharSequence from
shared objects file]
[Loaded java.lang.String from shared
objects file]
[Loaded
java.lang.reflect.GenericDeclaration
from shared objects
file]
[Loaded java.lang.reflect.Type from
shared objects file]
[Loaded
java.lang.reflect.AnnotatedElement from
shared objects file]
[Loaded java.lang.Class from shared
objects file]
[Loaded java.lang.Cloneable from shared
objects file]
[Loaded java.lang.ClassLoader from
shared objects file]
... and many more
```

Garbage Collection

When an object is instantiated using the `new` keyword, memory is allocated for the object. The scope of an object reference depends on where the object is instantiated:



- When `someMethod` completes, the memory referenced by `e` is no longer accessible.
- Java's garbage collector recognizes when an instance is no longer accessible and eligible for collection.

Note: When an object's memory is freed depends upon a number of factors.

Java's garbage collection scheme can be tuned depending on the type of application you are creating. For more information, consider taking the Oracle University course *Java Performance Tuning and Optimization* (D69518GC10).

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
  - Create primitive variables
  - Manipulate Strings
  - Use `if-else` and `switch` branching statements
  - Iterate with loops
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements

Quiz

1. In the following fragment, what three issues can you identify?

```
package com.oracle.test;
public class BrokenClass {
    public boolean valid = "false";
    public String s = new String ("A new string");
    public int i = 40_000.00;
    public BrokenClass() { }
}
```

   a. An `import` statement is missing.
   b. The `boolean valid` is assigned a String.
   c. `String s` is created using `new`.
   d. `BrokenClass` method is missing a `return` statement.
   e. Need to create a new `BrokenClass` object.
   f. The integer value `i` is assigned a double.

   Answer: b and f will cause compilation errors. c will compile but is not a good practice.

   a. An `import` statement is not required, unless the class uses classes outside of `java.lang`.

   `d. BrokenClass()` is a constructor.

   e. Construction of a `BrokenClass` instance would typically happen in another class.

2. Using the `Employee` class defined in this lesson, determine what will be printed in the following fragment:

```
public Employee changeName (Employee e, String name) {
    e.name = name;
    return (e);
}
//... in another method in the same class
Employee e = new Employee();
e.name = "Fred";
e = changeName(e, "Bob");
System.out.println (e.getName());
```

   a. Fred
   b. Bob
   c. `null`
   d. an empty String

   Answer: b

3. In the following fragment, what is the printed result?

```
public float average (int[] values) {
    float result = 0;
    for (int i = 1; i < values.length; i++)
        result += values[i];
    return (result/values.length);
}
// ... in another method in the same class
int[] nums = {100, 200, 300};
System.out.println (average(nums));
```

   a. 100.00
   b. 150.00
   c. 166.66667
   d. 200.00

   Answer: c

   Arrays begin with an index of 0. This average method is only averaging the 2nd through Nth values. Therefore, the result is the average of 200 +300/3 = 166.66667. Change the for loop to int = 0; to properly calculate the average.

Practice 2-1 Overview: Creating Java Classes

This practice covers the following topics:

- Creating a Java class using the NetBeans IDE
- Creating a Java class with a `main` method
- Writing code in the body of the `main` method to create an instance of the `Employee` object and print values from the class to the console
- Compiling and testing the application using the NetBeans IDE

Chapter 3

Encapsulation and Subclassing

Objectives

After completing this lesson, you should be able to do the following:

- ❑ Use encapsulation in Java class design
- ❑ Model business problems using Java classes
- ❑ Make classes immutable
- ❑ Create and use Java subclasses
- ❑ Overload methods
- ❑ Use variable argument methods

Encapsulation

The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it. In object-oriented programming, encapsulation covers, or wraps, the internal workings of a Java object.

- ❑ Data variables, or fields, are hidden from the user of the object.
- ❑ Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
- ❑ As long as the services do not change, the implementation can be modified without impacting the user.

The term *encapsulation* is defined by the Java Technology Reference Glossary as follows:

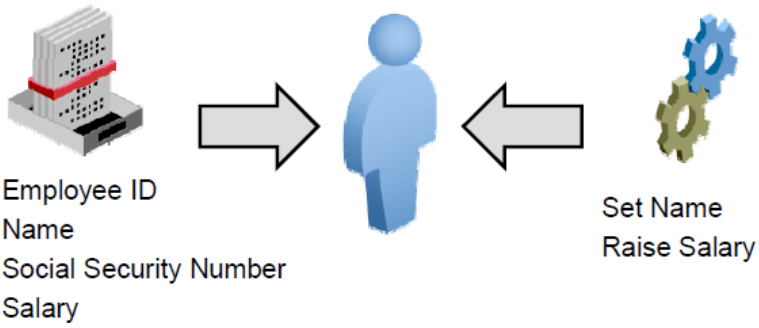
"The localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but if the services provided by the object remain the same, the code that uses the object can continue to use it without being rewritten."

An analogy for encapsulation is the steering wheel of a car. When you drive a car, whether it is your car, a friend's car, or a rental car, you probably never worry about how the steering wheel implements a right-turn or left-turn function. The steering wheel could be connected to the front wheels in a number of ways: ball and socket, rack and pinion, or some exotic set of servo mechanisms.

As long as the car steers properly when you turn the wheel, the steering wheel encapsulates the functions you need—you do not have to think about the implementation.

Encapsulation: Example

What data and operations would you encapsulate in an object that represents an employee?



A Simple Model

Suppose that you are asked to create a model of a typical employee. What data might you want to represent in an object that describes an employee?

- ❑ **Employee ID**: You can use this as a unique identifier for the employee.
- ❑ **Name**: Humanizing an employee is always a good idea!
- ❑ **Social Security Number**: For United States employees only. You may want some other identification for non-U.S. employees.
- ❑ **Salary**: How much the employee makes is always good to record.

What operations might you allow on the employee object?

- ❑ **Change Name**: If the employee gets married or divorced, there could be a name change.
- ❑ **Raise Salary**: Increases based on merit.

After an employee object is created, you probably do not want to allow changes to the Employee ID or Social Security fields. Therefore, you need a way to create an employee without alterations except through the allowed methods.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields `private`.

```
1 public class CheckingAccount {
2     private int custID;
3     private String name;
4     private double amount;
5     public CheckingAccount {
6     }
7     public void setAmount (double amount) {
8         this.amount = amount;
9     }
10     public double getAmount () {
11         return amount;
12     }
13     //... other public accessor and mutator methods
14 }
```

Declaring fields `private` prevents direct access to this data from a class instance.

In this example, the fields `custID`, `name`, and `amount` are now marked `private`, making them invisible outside of the methods in the class itself.

For example:

```
CheckingAccount ca = new CheckingAccount():
ca.amount = 1_000_000_000.00; // illegal!
```

Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
CheckingAccount chk = new CheckingAccount ();
chk.amount = 200; // Compiler error - amount is a private field
chk.setAmount (200); // OK
```

- The `private` keyword can also be applied to a method to hide an implementation detail.

```
// Called when a withdrawal exceeds the available funds
private void applyOverdraftFee () {
    amount += fee;
}
```

Revisiting Employee

The `Employee` class currently uses `public` access for all of its fields. To encapsulate the data, make the fields `private`.

```
package come.example.model;
public class Employee {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    //... constructor and methods
}
```

Encapsulation step 1: Hide the data (fields).

Method Naming: Best Practices

Although the fields are now hidden using `private` access, there are some issues with the current `Employee` class.

- The setter methods (currently `public` access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original `Employee` class design.
- Two best practices for methods:
  - Hide as many of the implementation details as possible.
  - Name the method in a way that clearly identifies its use or functionality.
- The original model for the `Employee` class had a Change Name and Increase Salary operation.

Choosing Well-Intentioned Methods

Just as fields should clearly define the type of data that they store, methods should clearly identify the operations that they perform. One of the easiest ways to improve the readability of your code (Java code or any other) is to write method names that clearly identify what they do.

`Employee` Class Refined

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     public Employee () {
5     }
6     // Remove all of the other setters
7     public void setName(String newName) {
8         if (newName != null) {
9             this.name = newName;
10         }
11     }
12
13     public void raiseSalary(double increase) {
14         this.salary += increase;
15     }
16 }
```

Encapsulation step 2: These method names make sense in the context of an Employee.

The current setter methods in the class allow any class that uses an instance of `Employee` to alter the object's ID, salary, and SSN fields. From a business standpoint, these are not operations you would want on an employee. Once the employee is created, these fields should be immutable (no changes allowed).

The `Employee` model as defined in the slide titled "Encapsulation: Example" had only two operations: one for changing an employee name (as a result of a marriage or divorce) and one for increasing an employee's salary.

To refine the `Employee` class, the first step is to remove the setter methods and create methods that clearly identify their purpose. Here there are two methods, one to change

an employee name (`setName`) and the other to increase an employee salary (`raiseSalary`).

Note that the implementation of the `setName` method tests the string parameter passed in to make sure that the string is not a null. The method can do further checking as necessary.

Make Classes as Immutable as Possible

```
1 package com.example.domain;
2 public class Employee {
3     // private fields ...
4     // Create an employee object
5     public Employee (int empId, String name,
6                      String ssn, double salary) {
7         this.empId = empId;
8         this.name = name;
9         this.ssn = ssn;
10        this.salary = salary;
11    }
12
13    public void setName(String newName) { ... }
14
15    public void raiseSalary(double increase) { ... }
16 }
```

Encapsulation step 3: Replace the no-arg constructor with a constructor to set the value of all fields.

Good Practice: Immutability

Finally, because the class no longer has setter methods, you need a way to set the initial value of the fields. The answer is to pass each field value in the construction of the object. By creating a constructor that takes all of the fields as arguments, you can guarantee that an `Employee` instance is fully populated with data *before* it is a valid employee object. This constructor *replaces* the no-arg constructor.

Granted, the user of your class could pass null values, and you need to determine if you want to check for those in your constructor. Strategies for handling those types of situations are discussed in later lessons.

Removing the setter methods and replacing the no-arg constructor also guarantees that an instance of `Employee` has immutable Employee ID and Social Security Number (SSN) fields.

Creating Subclasses

You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

```
1  package com.example.domain;
2  public class Manager {
3      private int empId;
4      private String name;
5      private String ssn;
6      private double salary;
7      private String deptName;
8      public Manager () {  }
9      // access and mutator methods...
10 }
```

wait a minute... this code looks very familiar....

Specialization Using Java Subclassing

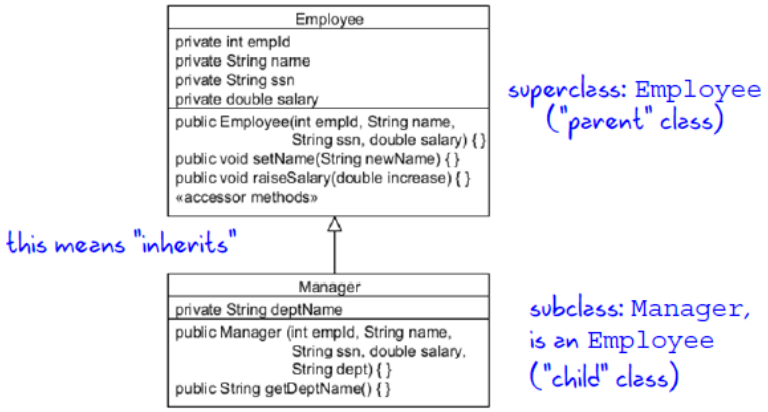
The `Manager` class shown here closely resembles the `Employee` class, but with some specialization. A `Manager` also has a department, with a department name. As a result, there are likely to be additional operations as well.

What this demonstrates is that a `Manager` is an `Employee`—but an `Employee` with additional features.

However, if we were to define Java classes this way, there would be a lot of redundant coding!

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



A Simple Java Program

When an existing class is subclassed, the new class created is said to inherit the characteristics of the other class. This new class is called the *subclass* and is a specialization of the superclass. All of the non-private fields and methods from the superclass are part of the subclass.

So in this diagram, a `Manager` class gets `empId`, `name`, `SSN`, `salary`, and all of the public methods from `Employee`.

It is important to grasp that although `Manager` specializes `Employee`, a `Manager` is still an `Employee`.

Note: The term *subclass* is a bit of a misnomer. Most people think of the prefix "sub-" as meaning "less."

However, a Java subclass is the sum of itself and its parent. When you create an instance of a subclass, the resulting in-memory structure contains all codes from the parent class, grandparent class, and so on all the way up the class hierarchy until you reach the class `Object`.

`Manager` Subclass

```
1 package com.example.domain;
2 public class Manager extends Employee {
3     private String deptName;
4     public Manager (int empId, String name,
5                     String ssn, double salary, String dept) {
6         super (empId, name, ssn, salary);
7         this.deptName = dept;
8     }
9
10     public String getDeptName () {
11         return deptName;
12     }
13     // Manager also gets all of Employee's public methods!
14 }
```

The `super` keyword is used to call the constructor of the parent class. It must be the first statement in the constructor.

Java Syntax for Subclassing

The keyword `extends` is used to create a subclass.

The `Manager` class, by extending the `Employee` class, inherits all of the non-private data fields and methods from `Employee`. After all, if a manager is also an employee, then it follows that `Manager` has all of the same attributes and operations of `Employee`.

Note that the `Manager` class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide. The constructor that `Manager` declares in line 4 calls the constructor of its parent class, `Employee`, using the `super` keyword. This sets the value of all of the `Employee` fields: `id`, `name`, `ssn`, and `salary`. `Manager` is a specialization of `Employee`, so constructing a `Manager` requires a department name, which is assigned to the `deptName` field in line 7.

What other methods might you want in a model of Manager? Perhaps you want a method that adds an `Employee` to this `Manager`. You can use an array or a special class called a *collection* to keep track of the employees for whom this manager is responsible. For details about collections, see the lesson titled "Generics and Collections."

Constructors are not Inherited

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- ❑ Write your own constructor.
- ❑ Use the default constructor.
  - If you do not declare a constructor, a default no-argument constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.

Constructors in Subclasses

Every subclass inherits the non-private fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

"Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding."

Using `super` in Constructors

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- ❑ In its constructor, `Manager` calls the constructor of `Employee`.

```
super (empId, name, ssn, salary);
```

- ❑ The `super` keyword is used to call a parent's constructor.
- ❑ It must be the first statement of the constructor.
- ❑ If it is not provided, a default call to `super()` is inserted for you.
- ❑ The `super` keyword may also be used to invoke a parent's method or to access a parent's (non-private) field.

The `Manager` class declares its own constructor and calls the constructor of the parent class using the `super` keyword.

Note: The `super` call of the parent's constructor must appear first in the constructor.

The `super` keyword can also be used to explicitly call the methods of the parent class or access fields.

Constructing a `Manager` Object

Creating a `Manager` object is the same as creating an `Employee` object:

```
Manager mgr = new Manager (102, "Barbara Jones",
                           "107-99-9078", 109345.67, "Marketing");
```

- ❑ All of the `Employee` methods are available to `Manager`:

```
mgr.raiseSalary (10000.00);
```

- The `Manager` class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```

Even though the `Manager.java` file does not contain all of the methods from the `Employee.java` class (explicitly), they are included in the definition of the object. Thus, after you create an instance of a `Manager` object, you can use the methods declared in `Employee`.
You can also call methods that are specific to the `Manager` class as well.

What is Polymorphism?

The word *polymorphism*, strictly defined, means "many forms."

```
Employee emp = new Manager();
```

- This assignment is perfectly legal. An employee can be a manager.
- However, the following does not compile:

```
emp.setDeptName ("Marketing"); // compiler error!
```

- The Java compiler recognizes the `emp` variable only as an `Employee` object. Because the `Employee` class does not have a `setDeptName` method, it shows an error.

In object-oriented programming languages such as Java, *polymorphism* is the ability to refer to an object using either its actual form or a parent form.
This is particularly useful when creating a general-purpose business method. For example, you can raise the salary of any `Employee` object (parent or child) by simply passing the object reference to a general-purpose business method that accepts an `Employee` object as an argument.

Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
  - Argument lists *must* differ.
  - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)
public String print (int i)
```

You might want to design methods with the same intent (method name), like `print`, to print out several different types. You could design a method for each type:

```
printInt(int i)
printFloat(float f)
printString(String s)
```

But this would be tedious and not very object-oriented. Instead, you can create a reusable method name and just change the argument list. This process is called *overloading*.
With overloading methods, the argument lists must be different–in order, number, or type. And the return types can be different. However, two methods with the same argument list that differ only in return type are not allowed.

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

Methods with a Variable Number of the Same Type

One case of overloading is when you need to provide a set of overloaded methods that differ in the number of the same type of arguments. For example, suppose you want to have methods to calculate an average. You may want to calculate averages for 2, 3, or 4 (or more) integers.
Each of these methods performs a similar type of computation–the average of the arguments passed in, as in this example:

```
public class Statistics {
public float average(int x1, int x2)
{ return (x1 + x2) / 2; }
public float average(int x1, int x2,
int x3) {
  return (x1 + x2 + x3) / 3;
}
public float average(int x1, int x2,
int x3, int x4) {
  return (x1 + x2 + x3 + x4) / 4;
```

```
}
}
```

Java provides a convenient syntax for collapsing these three methods into just one and providing for any number of arguments.

- Java provides a feature called *varargs* or *variable arguments*.

```
1 public class Statistics {
2     public float average(int... nums) {
3         int sum = 0;
4         for (int x : nums) {  // iterate int array nums
5             sum += x;
6         }
7         return ((float) sum / nums.length);
8     }
9 }
```

The varargs notation treats the `nums` parameter as an array.

- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.

Using Variable Arguments

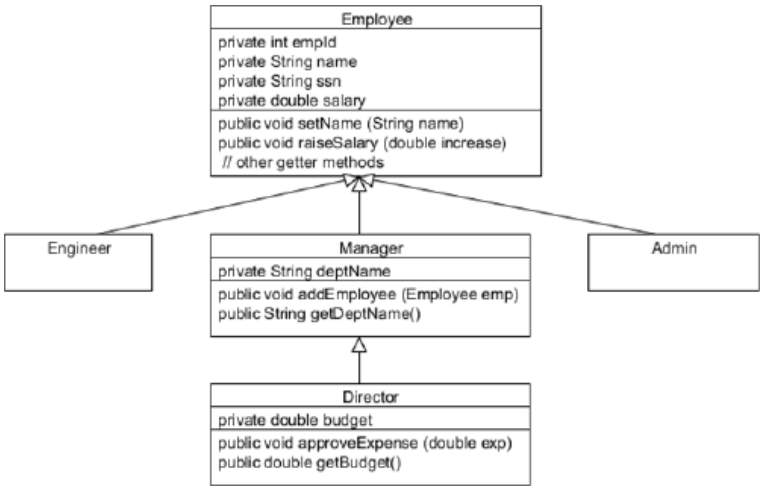
The `average` method shown in the slide takes any number of integer arguments. The notation `(int... nums)` converts the list of arguments passed to the `average` method into an array object of type `int`.

Note: Methods that use varargs can also take no parameters?an invocation of `average()` is legal. You will see varargs as optional parameters in use in the NIO.2 API in the lesson titled "Java File I/O." To account for this, you could rewrite the `average` method in the slide as follows:

```
public float average(int... nums) {
int sum = 0; float result = 0;
  if (nums.length > 0) {
   for (int x : nums) // iterate int
array nums
    sum += x;
   result = (float) sum / nums.length;
   }
 return (result);
}
}
```

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Although Java does not permit more than one class to a subclass, the language does provide features that enable multiple classes to implement the features of other classes. You will see this in the lesson on inheritance.

Single inheritance does not prevent continued refinement and specialization of classes as shown above.

In the diagram shown in the slide, a manager can have employees, and a director has a budget and can approve expenses.

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
- Use encapsulation in Java class design
- Model business problems using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods
- Use variable argument methods

Quiz

1. Given the diagram in the slide titled "Single Inheritance" and the following Java statements, which statements do *not* compile?

```
Employee e = new Director();
Manager m = new Director();
Admin a = new Admin();
```

   a. `e.addEmployee (a);`
   b. `m.addEmployee(a);`
   c. `m.approveExpense (100000.00);`
   d. All of them fail to compile.

Answer: a, c

   a. A compiler error because the `Employee` class does not have an `addEmployee` method. This is a part of the `Manager` class.
   b. Compiles properly because, although the

constructor is creating a `Director`, it is the `Manager` class that the compiler is looking at to determine if there is an `addEmployee` method

c. A compiler error because the `Manager` class does not contain an `approveExpense` method

2. Consider the following classes that do not compile:

```
public class Account {
    private double balance;
    public Account(double balance) { this.balance = balance; }
    //... getter and setter for balance
}
public class Savings extends Account {
    private double interestRate;
    public Savings(double rate) { interestRate = rate; }
}
```

What fix allows these classes to compile?

a. Add a no-arg constructor to `Savings`.
b. Call the `setBalance` method of `Account` from `Savings`.
c. Change the access of `interestRate` to `public`.
d. Replace the constructor in `Savings` with one that calls the constructor of `Account` using `super`.

Answer: d

`Savings` must call the constructor of its parent class (`Account`). To do that, you must replace the current `Savings` constructor with one that includes an initial balance, and calls the Account constructor using super, as in this example:

```
public Savings (double balance,
double rate) {
super(balance);
interestRate = rate;
}
```

3. Which of the following declarations demonstrates the application of good Java naming conventions?

1. `public class repeat { }`
2. `public void Screencoord (int x, int y){}`
3. `private int XCOORD;`
4. `public int calcOffset (int x1, int y1, int x2, int y2) { }`

Answer: d

1. Uses a lowercase first letter and a verb for a class name. Class names should be nouns with an initial uppercase letter.
2. Is a method name with its first letter uppercase, rather than lower camel case (with the first letter lowercase and the first letter of each name element in uppercase). In addition, `Screencoord` sounds like a noun rather than a verb.
3. Is questionable because it appears to be a constant. It is in uppercase, however, it is not declared final and there is no assigned value.
4. Follows the Java naming convention. It clearly identifies its intent and will calculate the offset between the two coordinate pairs passed as arguments.

4. What changes would you perform to make this class immutable? (Choose all that apply.)

```
public class Stock {
    public String symbol;
    public double price;
    public int shares;
    public double getStockValue() { }
    public void setSymbol(String symbol) { }
    public void setPrice(double price) { }
    public void setShares(int number) { }
}
```

a. Make the fields `symbol`, `shares`, and `price private`.
b. Remove `setSymbol`, `setPrice`, and `setShares`.
c. Make the `getStockValue` method `private`.
d. Add a constructor that takes `symbol`, `shares`, and `price` as arguments.

Answer: a, b, d

"Immutable" simply means that the object cannot be changed after it is created. Making the fields `private` prevents access from outside the class. Removing the setter methods prevents changes. Adding the constructor allows the object to be built for the first time with values. The `getStockValue` method does not change any of the fields of the object, so it does not need to be removed.

Practice 3-1 Overview: Creating Subclasses

This practice covers the following topics:

- ❑ Applying encapsulation principles to the `Employee` class that you created in the previous practice
- ❑ Creating subclasses of `Employee`, including `Manager`, `Engineer`, and Administrative assistant (`Admin`)
- ❑ Creating a subclass of `Manager` called `Director`
- ❑ Creating a test class with a `main` method to test your new Classes

(Optional) Practice 3-2 Overview: Adding a Staff to a Manager

This practice covers the following topics:

- Creating an array of `Employees` called `staff`
- Creating a method to add an employee to the manager's `staff`
- Creating a method to remove an employee from the manager's `staff`

Chapter 4

Java Class Design

Objectives

After completing this lesson, you should be able to do the following:

- Use access levels: `private`, `protected`, default, and `public`
- Override methods
- Overload constructors and other methods appropriately
- Use the `instance of` operator to compare object types
- Use virtual method invocation
- Use upward and downward casts
- Override methods from the `Object` class to improve the functionality of your class

Using Access Control

You have seen the keywords `public` and `private`. There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or method marked with the access modifier in the left column.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes *	
`public`	Yes	Yes	Yes	Yes

Classes can be default (no modifier) or `public`.

The access modifier keywords shown in this table are `private`, `protected`, and `public`. When a keyword is absent, the *default* access modifier is applied.

The `private` keyword provides the greatest control over access to fields and methods. With `private`, a data field or method can be accessed only within the same Java class.

The `public` keyword provides the greatest access to fields and methods, making them accessible anywhere: in the class, package, subclasses, and any other class.

The `protected` keyword is applied to keep access within the package and subclass. Fields and methods that use `protected` are said to be "subclass-friendly."

***Note**: `protected` access is extended to subclasses that reside in a package different from the class that owns the `protected` feature. As a result, `protected` fields or methods are actually more accessible than those marked with default access control. You should use `protected` access when it is appropriate for a class's subclass, but not unrelated classes.

Protected Access Control: Example

```
1 package demo;
2 public class Foo {
3     protected int result = 20;
4     int other = 25;
5 }
```

subclass-friendly declaration

```
1 package test;
2 import demo.Foo;
3 public class Bar extends Foo {
4      private int sum = 10;
5      public void reportSum () {
6          sum += result;
7          sum += other;
8      }
9 }
```

compiler error

In this example, there are two classes in two packages. Class `Foo` is in the package `demo`, and declares a data field called `result` with a `protected` access modifier. In the class `Bar`, which extends `Foo`, there is a method, `reportSum`, that adds the value of `result` to `sum`. The method then attempts to add the value of `other` to sum. The field `other` is declared using the default modifier, and this generates a compiler error. Why?

Answer: The field `result`, declared as a `protected` field, is available to all subclasses-even those in a different package. The field `other` is declared as using default access and is only available to classes and subclasses declared in the same package.

This example is from the `JavaAccessExample` project.

Field Shadowing: Example

```
1 package demo;
2 public class Foo2 {
3     protected int result = 20;
4 }
```

```
1 package test;
2 import demo.Foo2;
3 public class Bar2 extends Foo2 {
4     private int sum = 10;
5     private int result = 30;
6     public void reportSum() {
7         sum += result;
8     }
9 }
```

result field shadows the parent's field.

In this example, the class `Foo2` declares the field `result`. However, the class `Bar2` declares its own field `result`. The consequence is that the field `result` from class `Foo2` is shadowed by the field `result` in class `Bar2`. What is `sum` in this example? `sum` is now 40 (10 + 30). Modern IDEs (such as Net Beans) detect shadowing and produce a warning.

Methods with the same name are not shadowed but are overridden. You learn about overriding later in this lesson.

Access Control: Good Practice

A good practice when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
1 package demo;
2 public class Foo3 {
3    private int result = 20;
4    protected int getResult() { return result; }
5 }
```

```
1 package test;
2 import demo.Foo3;
3 public class Bar3 extends Foo3 {
4     private int sum = 10;
5     public void reportSum() {
6         sum += getResult();
7     }
8 }
```

A slightly modified version of the example using the `protected` keyword is shown in the slide. If the idea is to limit access of the field result to classes within the package and the subclasses (package-protected), you should make the access explicit by defining a method purposefully written for package and subclass-level access.

Overriding Methods

Consider a requirement to provide a String that represents some details about the `Employee` class fields.

```
1 public class Employee {
2     private int empId;
3     private String name;
4     // ... other fields and methods
5     public String getDetails () {
6         return "Employee id:  " + empId +
7                " Employee name:" + name;
8     }
9 }
```

Although the `Employee` class has getters to return values for a print statement, it might be nice to have a utility method to get specific details about the employee. Consider a method added to the `Employee` class to print details about the `Employee` object.

In addition to adding fields or methods to a subclass, you can also modify or change the existing behavior of a method of the parent (superclass).

You may want to specialize this method to describe a `Manager` object.

In the `Manager` class, by creating a method with the same signature as the method in the `Employee` class, you are *overriding* the `getDetails` method:

```
1 public class Manager extends Employee {
2     private String deptName;
3     // ... other fields and methods
4     public String getDetails () {
5         return super.getDetails () +
6                " Department: " + deptName;
7     }
8 }
```

A subclass can invoke a parent method by using the `super` keyword.

When a method is overridden, it replaces the method in the superclass (parent) class. This method is called for any `Manager` instance.

A call of the form `super.getDetails()` invokes the `getDetails` method of the parent class.

Note: If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method.

Invoking an Overridden Method

- Using the previous examples of `Employee` and `Manager`:

```
Employee e = new Employee (101, "Jim Smith", "011-12-2345",
100_000.00);
Manager m = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
System.out.println (m.getDetails());
```

- The correct `getDetails` method of each class is called:

```
Employee id: 101 Employee name: Jim Smith
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

During run time, the Java Virtual Machine invokes the `getDetails` method of the appropriate class. If you comment out the `getDetails` method in the `Manager` class shown in the previous slide, what happens when `m.getDetails()` is invoked?

Answer: Recall that methods are inherited from the parent class. So, at run time, the `getDetails` method of the parent class(`Employee`) is executed.

Virtual Method Invocation

- What happens if you have the following?

```
Employee e = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
```

- During execution, the object's runtime type is determined to be a `Manager` object:

```
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

- The compiler is satisfied because the `Employee` class has a `getDetails` method, and at run time the method that is executed is referenced from a `Manager` object.
- This is an aspect of polymorphism called *virtual method invocation*.

Compiler Versus Runtime Behavior

The important thing to remember is the difference between the compiler (which checks that each method and field is accessible based on the strict definition of the class) and the behavior associated with an object determined at run time.

This distinction is an important and powerful aspect of polymorphism: The behavior of an object is determined by its runtime reference.

Because the object you created was a `Manager` object, at run time, when the `getDetails` method was invoked, the run time reference is to the `getDetails` method of a `Manager` class, even though the variable e is of the type `Employee`.

This behavior is referred to as *virtual method invocation*.

Note: If you are a C++ programmer, you get this behavior in C++ only if you mark the method using the C++ keyword `virtual`.

Accessibility of Overridden Methods

An overriding method must provide at least as much access as the overridden method in the parent class.

```
public class Employee {
    //... other fields and methods
    public String getDetails() { ... }
}
```

```
public class Manager extends Employee {
    //... other fields and methods
    private String getDetails() { //... }  // Compile time error
}
```

To override a method, the name and the order of arguments must be identical.

When a method is a subclass overrides a method in the parent class, it must provide the same or greater access than the parent class. For example, if the parent method get Details() on the slide was protected, then the overriding method getDetails() in the subclass must be protected or public.

Applying Polymorphism

Suppose that you are asked to create a new class that calculates a stock grant for employees based on their salary and their role (manager, engineer, or admin):

```
1 public class EmployeeStockPlan {
2    public int grantStock (Manager m) {
3       // perform a calculation for a Manager
4    }
5    public int grantStock (Engineer e) {
6       // perform a calculation for an Engineer
7    }
8    public int grantStock (Admin a) {
9       // perform a calculation for an Admin
10   }
11   //... one method per employee type
12}
```

not very object-oriented!

Design Problem

What is the problem in the example in the slide? Each method performs the calculation based on the type of employee passed in, and returns the number of shares.

Consider what happens if you add two or three more employee types. You would need to add three additional methods, and possibly replicate code depending upon the business logic required to compute shares.

Clearly, this is not a good way to treat this problem. Although the code will work, this is not easy to read and is likely to create much duplicate code.

A good practice is to pass parameters and write methods that use the most generic form of your object as possible.

```
public class EmployeeStockPlan {
   public int grantStock (Employee e) {
      // perform a calculation based on Employee data
   }
}
```

```
// In the application class
EmployeeStockPlan esp = new EmployeeStockPlan ():
Manager m = new Manager ();
int stocksGranted = grantStock (m);
...
```

Use the Most Generic Form

A good practice is to design and write methods that take the most generic form of your object as possible.
In this case, `Employee` is a good base class to start from.
Adding a method to `Employee` allows `EmployeeStockPlan` to use polymorphism to calculate stock.

```
public class Employee {
    protected int calculateStock() { return 10; }
}
```

```
public class Manager extends Employee {
    public int calculateStock() { return 20; }
}
```

```
public class EmployeeStockPlan {
    private float stockMultiplier;   // Calculated elsewhere
    public int grantStock (Employee e) {
        return (int)(stockMultipier * e.calculateStock());
    }
}
```

In this modified `EmployeeStockPlan`, the `grantStock` method calls a method defined by the base class, `Employee`, that returns a base stock count. The `EmployeeStockPlan` class uses a multiplier to determine how many stocks to grant. A class may override the `calculateStock` method (as the `Manager` class does here) based on business policy.
In this approach, regardless of the number of different Employee types (all extending the `Employee` object) the `EmployeeStockPlan` class will always function.

Using the `instanceof` Keyword

The Java language provides the ins tanceof keyword to determine an object's class type at run time.

```
1 public class EmployeeRequisition {
2    public boolean canHireEmployee(Employee e) {
3        if (e instanceof Manager) {
4            return true;
5        } else {
6            return false;
7        }
8    }
9 }
```

In this example, a class, `EmployeeRequisition` has a method that uses the `instanceof` keyword to determine if the object can open an requisition for an employee. Per the business policy, only Managers and above can open a requisition for a new employee.

Casting Object References

In order to access a method in a subclass passed as an generic argument, you must cast the reference to the class that will satisfy the compiler:

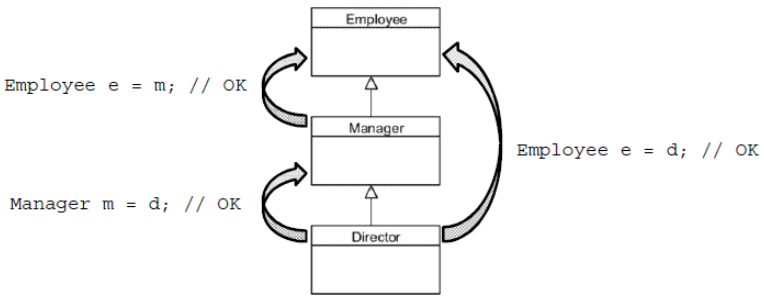
```
1 public void modifyDeptForManager (Employee e, String dept) {
2    if (e instanceof Manager) {
3       Manager m = (Manager) e;
4       m.setDeptName (dept);
5    }
6 }
```

Without the cast to `Manager`, the `setDeptName` method would not compile.
Although a generic superclass reference is useful for passing objects around, you may need to use a method from the subclass.
In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.
However, there are rules for casting references. You see these in the next slide.
Note: The `instanceof` operator shown on the slide is not required by the compiler before the cast, however, without checking the object's type, if a non-`Manager` object type is passed to the `modifyDeptForManager` method, an exception (`ClassCastException`) will be thrown at runtime.

Casting Rules

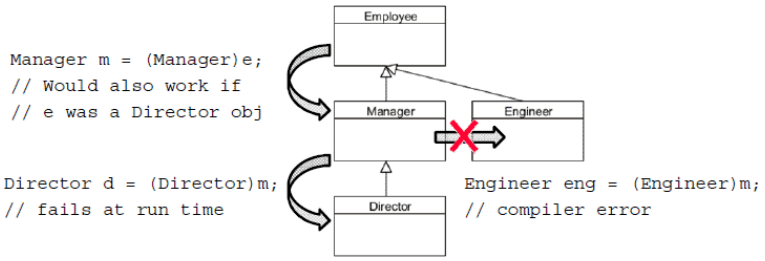
Upward casts are always permitted and do not require a cast operator.

```
Director d = new Director();
Manager m = new Manager();
```



For downward casts, the compiler must be satisfied that the cast is at least possible.

```
Employee e = new Manager();
Manager m = new Manager();
```



Downward Casts

With a downward cast, the compiler simply determines if the cast is possible; if the cast down is to a subclass, then it is possible that the cast will succeed.

Note that at run time the cast results in a `java.lang.ClassCastException` if the object reference is of a superclass and not of the class type or a subclass.

The cast of the variable `e` to a Manager reference `m` satisfies the compiler, because `Manager` and `Employee` are in the same class hierarchy, so the cast will possibly succeed. This cast also works at run time, because it turns out that the variable `e` is actually a `Manager` object. This cast would also work at run time if `e` pointed to an instance of a `Director` object.

The cast of the variable `m` to a `Director` instance satisfies the compiler, but because m is actually a `Manager` instance, this cast fails at run time with a `ClassCastException`.

Finally, any cast will fail that is outside the class hierarchy, such as the cast from a `Manager` instance to an `Engineer`. A `Manager` and an `Engineer` are both employees, but a `Manager` is not an `Engineer`.

Overriding Object Methods

One of the advantages of single inheritance is that every class has a parent object by default. The root class of every Java class is `java.lang.Object`.

- You do not have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```

is equivalent to:

```
public class Employee extends Object { //... }
```

- The root class contains several non-final methods, but there are three that are important to consider overriding:
  - `toString`, `equals`, and `hashCode`

Best Practice: Overload Object Methods

The `java.lang.Object` class is the root class of all classes in the Java programming language. All classes will subclass `Object` by default.

`Object` defines several non-final methods that are designed to be overridden by your class. These are `equals`, `hashCode`, `toString`, `clone`, and `finalize`. Of these, there are three methods that you should consider overriding.

Object toString Method

The `toString` method is called to return the string value of an object. For example, using the method `println`:

```
Employee e = new Employee (101, "Jim Kern", ...)
System.out.println (e);
```

- String concatenation operations also invoke `toString`:

```
String s = "Employee: " + e;
```

- You can use toString to provide instance information:

```
public String toString () {
    return "Employee id:  " + empId + "\n"
           "Employee name:" + name;
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.

The println method is overloaded with a number of parameter types. When you invoke `System.out.println(e);` the method that takes an `Object` parameter is matched and invoked. This method in turn invokes the `toString()` method on the object instance.

Note: Sometimes you may want to be able to print out the name of the class that is executing a method. The `getClass()` method is an `Object` method used to return the `Class` object instance, and the `getName()` method provides the fully qualified name of the runtime class.

```
getClass().getName();  // returns the name of this class instance
```

These methods are in the `Object` class.

Object equals Method

The `Object equals` method compares only object references.

- If there are two objects x and y in any class, x is equal to y if and only if x and y refer to the same object.
- Example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Because what we really want is to test the contents of the Employee object, we need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```

The `equals` method of `Object` determines (by default) only if the values of two object references point to the same object. Basically, the test in the `Object` class is simply as follows:

If x == y, return true.

For an object (like the `Employee` object) that contains values, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.

Overriding `equals` in `Employee`

An example of overriding the `equals` method in the `Employee` class compares every field for equality:

```
1 public boolean equals (Object o) {
2     boolean result = false;
3     if ((o != null) && (o instanceof Employee)) {
4         Employee e = (Employee)o;
5         if ((e.empId == this.empId) &&
6             (e.name.equals(this.name)) &&
7             (e.ssn.equals(this.ssn)) &&
8             (e.salary == this.salary)) {
9                result = true;
10         }
11     }
12     return result;
13 }
```

This simple `equals` test first tests to make sure that the object passed in is not null, and then tests to make sure that it is an instance of an `Employee` class (all subclasses are also employees, so this works). Then the `Object` is cast to `Employee`, and each field in `Employee` is checked for equality.

Note: For `String` types, you should use the `equals` method to test the strings character by character for equality.

Overriding `Object hashCode`

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer hashcode returned for the two objects should also be equal.

```
1 // Generated by NetBeans
2 public int hashCode() {
3     int hash = 7;
4     hash = 83 * hash + this.empId;
5     hash = 83 * hash + Objects.hashCode(this.name);
6     hash = 83 * hash + Objects.hashCode(this.ssn);
7     hash = 83 * hash +
8             (int)(Double.doubleToLongBits(this.salary) ^
9             (Double.doubleToLongBits(this.salary) >>> 32));
10     return hash;
11 }
```

Overriding `hashCode`

The Java documentation for the `Object` class states:
"... It is generally necessary to override the `hashCode` method whenever this method [`equals`] is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."
The `hashCode` method is used in conjunction with the `equals` method in hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`.
This method is easy to get wrong, so you need to be careful. The good news is that IDEs such as NetBeans can generate `hashCode` for you.
To create your own hash function, the following will help approximate a reasonable hash value for equal and unequal instances:

1. Start with a non-zero integer constant. Prime numbers result in fewer hashcode collisions.
2. For each field used in the `equals` method, compute an `int` hash code for the field. Notice that for the `Strings`, you can use the `hashCode` of the `String`.
3. Add the computed hash codes together.
4. Return the result.

Summary

In this lesson, you should have learned how to:

- Use access levels: `private`, `protected`, default, and `public`
- Override methods
- Overload constructors and other methods appropriately
- Use the `instanceof` operator to compare object types
- Use virtual method invocation
- Use upward and downward casts
- Override methods from the `Object` class to improve the functionality of your class

Quiz

1. Suppose that you have an `Account` class with a `withdraw ()` method, and a `Checking` class that extends `Account` that declares its own `withdraw ()` method. What is the result of the following code fragment?

```
1  Account acct = new Checking();
2  acct.withdraw(100);
```

   a. The compiler complains about line 1.
   b. The compiler complains about line 2.
   c. Runtime error: incompatible assignment (line 1)
   d. The `Account.withdraw ()` method executes.
   e. The `Checking.withdraw ()` method executes.

   Answer: e

   Because the `Checking` class extends `Account`, the `withdraw` method declared in `Checking` overrides the `withdraw` method in `Account`. At run time, the method for the object (`Checking`) is executed.

2. Suppose that you have an `Account` class and a `Checking` class that extends `Account`. The body of the `if` statement in line 2 will execute.

```
1  Account acct = new Checking();
2  if (acct instanceof Checking) { // will this block run? }
```

   a. True
   b. False

   Answer: a

   Actually, acct is also an `instanceof` the `Account` class.

3. Suppose that you have an `Account` class and a `Checking` class that extends `Account`. You also have a `Savings` class that extends `Account`. What is the result of the following code?

```
1  Account acct1 = new Checking();
2  Account acct2 = new Savings();
3  Savings acct3 = (Savings)acct1;
```

   a. `acct 3` contains the reference to `acct 1`.
   b. A runtime `ClassCastException` occurs.
   c. The compiler complains about line 2.
   d. The compiler complains about the cast in line 3.

   Answer: b

4. The compiler will assume that it is possible to cast an `Account` type object to another `Account`. Because `Savings` extends from `Account`, this looks like a typical downward cast. However, at run time, the true type of the object is determined, and you cannot cast between children.

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- ❑ Modifying the `Employee`, `Manager`, and `Director` classes; overriding the `tostring ()` method
- ❑ Creating an `EmployeestockPlan` class with a grant stock method that uses the `instanceof` keyword

Chapter 5

Advanced Class Design

Objectives

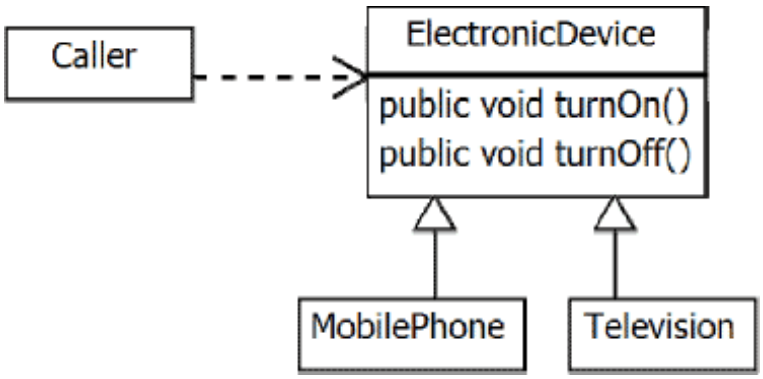
After completing this lesson, you should be able to do the following:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Model business problems by using the `static` and `final` keywords
- Implement the singleton design pattern
- Distinguish between top-level and nested classes

Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- Method inheritance: Subclasses avoid code duplication by inheriting method implementations.
- Generalization: Code that is designed to rely on the most generic type possible is easier to maintain.



Class Inheritance Diagram

Class Inheritance

When designing an object-oriented solution, you should attempt to avoid code duplication.
One technique to avoid duplication is to create library methods and classes. Libraries function as a central point to contain often reused code. Another technique to avoid code duplication is to use class inheritance. When there is a shared base type identified between two classes, any shared code may be placed in a parent class.
When possible, use object references of the most generic base type possible. In Java, generalization and specialization enable reuse through method inheritance and virtual method invocation (VMI). VMI, sometimes called "late-binding," enables a caller to dynamically call a method as long as the method has been declared in a generic base type.

Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.

Coding for Generalization

Always use the most generic reference type possible. Java IDEs may contain refactoring tools that assist in changing existing references to a more generic base type.

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized.

```
public class Television extends ElectronicDevice {

    public void turnOn() {
        changeChannel(1);
        initializeScreen();
    }
    public void turnOff() {}

    public void changeChannel(int channel) {}
    public void initializeScreen() {}

}
```

Method Implementations

When sibling classes have a common method, it is typically placed in a parent class. Under some circumstances, however, the parent class's implementation will always need to be overridden with a specialized implementation.
In these cases, inclusion of the method in a parent class has both advantages and disadvantages. It allows the use of generic reference types, but developers can easily forget

to supply the specialized implementation in the subclasses.

Defining Abstract Classes

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class cannot be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```

Declaring a class as abstract prevents any instances of that class from being created. It is a compile-time error to instantiate an abstract class. An abstract class will typically be extended by a child class and may be used as a reference type.

Defining Abstract Methods

A method can be declared as abstract by using the `abstract` method-level modifier.

```
public abstract class ElectronicDevice {

   public abstract void turnOn();
   public abstract void turnOff();
}
```

No braces

An abstract method:

- Cannot have a method body
- Must be declared in an abstract class
- Is overridden in subclasses

Inheriting Abstract Methods

When a child class inherits an abstract method, it is inheriting a method signature but no implementation. For this reason, no braces are allowed when defining an abstract method. An abstract method is a way to guarantee that any child class will contain a method with a matching signature.

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and non-abstract methods.
- When inheriting from an abstract class, you must do either of the following:
  - Declare the child class as abstract.
  - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override
abstract method turnOn() in ElectronicDevice
```

Making Use of Abstract Classes

While it is possible to avoid implementing an abstract method by declaring child classes as abstract, this only serves to delay the inevitable. Applications require non-abstract methods to create objects. Use abstract methods to outline functionality required in child classes.

Quiz

To compile successfully, an abstract method must not have:

a. A return value
b. A method implementation
c. Method parameters
d. `private` access

Answer: b, d

`static` Keyword

The `static` modifier is used to declare fields and methods as class-level resources. Static class members:

- Can be used without object instances
- Are used when a problem is best solved without objects
- Are used when objects of the same type need to share fields
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason

Java: Object-oriented by Design

The Java programming language was designed as an object-oriented language, unlike languages like Objective-C and C++, which inherited the procedural design of C. When developing in Java, you should always attempt to design an object-oriented solution.

Static Methods

Static methods are methods that can be called even if the class they are declared in has not been instantiated. Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented.

- `java.lang.Math` contains many static methods

- ❑ Are commonly used in place of constructors to perform tasks related to object initialization
- ❑ Cannot access non-static members within the same class
- ❑ Can be hidden in subclasses but not overridden
  - No virtual method invocation

Factory Methods

In place of directly invoking constructors, you will often use static methods to retrieve object references. Unless something unexpected happens, a new object is created whenever a constructor is called. A static factory method could maintain a cache of objects for reuse or create new instances if the cache was depleted. A factory method may also produce an object that subclasses the method's return type.

Example:

```
NumberFormat nf =
NumberFormat.getInstance();
```

Implementing Static Methods

```
public class StaticErrorClass {
    private int x;

    public static void staticMethod() {
        x = 1; // compile error
        instanceMethod(); // compile error
    }

    public void instanceMethod() {
        x = 2;
    }
}
```

Static Method Limitations

Static methods can be used before any instances of their enclosing class have been created. Chronologically speaking, this means that in a running Java Virtual Machine, there may not be any occurrences of the containing classes instance variables. Static methods can never access their enclosing classe's instance variables or call their non-static methods.

Calling Static Methods

```
double d = Math.random();
StaticUtilityClass.printMessage();
StaticUtilityClass uc = new StaticUtilityClass();
uc.printMessage(); // works but misleading
sameClassMethod();
```

When calling static methods, you should:

- ❑ Qualify the location of the method with a class name if the method is located in a different class than the caller
  - Not required for methods within the same class
- ❑ Avoid using an object reference to call a static method

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated. Static variables are:

- ❑ Called class variables
- ❑ Limited to a single copy per JVM
- ❑ Useful for containing shared data
  - Static methods store data in static variables.
  - All object instances share a single copy of any static variables.
- ❑ Initialized when the containing class is first loaded

Class Loading

Application developer-supplied classes are typically loaded on demand (first use). Static variables are initialized when their enclosing class is loaded. An attempt to access a static class member can trigger the loading of a class.

Defining Static Variables

```
public class StaticCounter {
    private static int counter = 0;

    public StaticCounter() {
        counter++;
    }

    public static int getCount() {
        return counter;
    }
}
```

Only one copy in memory

Persisting Static Variables

Many technologies that are used to persist application state in Java only save instance variables. Maintaining a single object that keeps track of "shared" state may be used as an alternative to static variables.

Using Static Variables

```
double p = Math.PI;
```

```
new StaticCounter();
new StaticCounter();
System.out.println("count: " + StaticCounter.getCount());
```

When accessing static variables, you should:

- ❑ Qualify the location of the variable with a class name if the variable is located in a different class than the caller
  - Not required for variables within the same class
- ❑ Avoid using an object reference to access a static variable

Object References to Static Members

Just as using object references to static methods should be avoided, you should also avoid using object references to access static variables. If all the members of a class are static, consider using a private constructor to prevent object instantiation.

Static Imports

A static import statement makes the static members of a class available under their simple name.

- ❑ Given either of the following lines:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- ❑ Calling the `Math.random()` method can be written as:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```

Overusing static import can negatively affect the readability of your code. Avoid adding multiple static imports to a class.

Quiz

The number of instances of a static variable is related to the number of objects that have been created.

a. True
b. False

Answer: b

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```

Performance Myths

There is little to no performance benefit when you declare a method as final. Methods should be declared as final only to disable method overriding.

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

Final Variables

The `final` modifier can be applied to variables. Final variables may not change their values after they are initialized. Final variables can be:

- ❑ Class fields
  - Final fields with compile-time constant expressions are constant variables.
  - Static can be combined with final to create an alwaysavailable, never-changing variable.
- ❑ Method parameters
- ❑ Local variables

Note: Final references must always reference the same object, but the contents of that object may be modified.

Benefits and Drawbacks of Final Variables

Bug Prevention

Final variables can never have their values modified after they are initialized. This behavior functions as a bug-prevention mechanism.

Thread Safety

The immutable nature of final variables eliminates any of

the concerns that come with concurrent access by multiple threads.

Final Reference to Objects

A `final` object reference only serves to prevent a reference from pointing to another object. If you are designing immutable objects, you must prevent the object's fields from being modified. Final references also prevent you from assigning a value of `null` to the reference. Maintaining an object's references prevents that object from being available for garbage collection.

Declaring Final Variables

```
public class VariableExampleClass {
    private final int field;
    private final int forgottenField;
    private final Date date = new Date();
    public static final int JAVA_CONSTANT = 10;

    public VariableExampleClass() {
        field = 100;
        // compile-time error - forgottenField
        // not initialized
    }

    public void changeValues(final int param) {
        param = 1; // compile-time error
        date.setTime(0); // allowed
        date = new Date(); // compile-time error
        final int localVar;
        localVar = 42;
        localVar = 43; // compile-time error
    }
}
```

Final Fields

Initializing

Final fields (instance variables) must be either of the following:

- ❑ Assigned a value when declared
- ❑ Assigned a value in every constructor

Static and Final Combined

A field that is both static and final is considered a constant. By convention, constant fields use identifiers consisting of only uppercase letters and underscores.

Quiz

A final field (instance variable) can be assigned a value either when declared or in all constructors.

1. True
2. False

Answer: a

When to Avoid Constants

`public static final` variables can be very useful, but there is a particular usage pattern you should avoid. Constants may provide a false sense of input validation or value range checking.

- ❑ Consider a method that should receive only one of three possible values:

```
Computer comp = new Computer();
comp.setState(Computer.POWER_SUSPEND);
```

This is an `int` constant that equals 2.

- ❑ The following lines of code still compile:

```
Computer comp = new Computer();
comp.setState(42);
```

Runtime Range Checking

In the example in the slide, you must perform a runtime range check when using an `int` to represent state. Within the `setState` method, an `if` statement can be used to validate that only the values `0`, `1`, or `2` are accepted. This type of check is performed every time the `setState` method is called, resulting in additional overhead.

Typesafe Enumerations

Java 5 added a typesafe enum to the language. Enums:

- ❑ Are created using a variation of a Java class
- ❑ Provide a compile-time range check

```
public enum PowerState {
    OFF,
    ON,
    SUSPEND;
}
```

These are references to the only three `PowerState` objects that can exist.

An enum can be used in the following way:

```
Computer comp = new Computer();
comp.setState(PowerState.SUSPEND);
```

This method takes a `PowerState` reference.

Compile-Time Range Checking

In the example in the slide, the compiler performs a compile-time check to ensure that only valid `PowerState` instances are passed to the `setState` method. No range checking overhead is incurred at run time.

Enum Usage

Enum references can be statically imported.

```
import static com.example.PowerState.*;

public class Computer extends ElectronicDevice {
    private PowerState powerState = OFF;
    //...
}
```

`PowerState.OFF`

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {
    switch(state) {
        case OFF:
        //...
    }
}
```

Note: When an `enum` is used in a `switch` statement, only the `enum` constants can be used as labels for the case statements.

Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {
    OFF("The power is off"),
    ON("The usage power is high"),
    SUSPEND("The power usage is low");

    private String description;
    private PowerState(String d) {
        description = d;
    }
    public String getDescription() {
        return description;
    }
}
```

Call a `PowerState` constructor to initialize the `public static final OFF` reference.

The constructor may not be `public` or `protected`.

Enum Constructors

You may not instantiate an enum instance with `new`.

Quiz

An enum can have a constructor with the following access levels:

a. public
b. protected
c. Default (no declared access level)
d. private

Answer: c, d

Design Patterns

Design patterns are:

- ❑ Reusable solutions to common software development problems
- ❑ Documented in pattern catalogs
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the "Gang of Four")
- ❑ A vocabulary used to discuss design

Design Pattern Catalogs

Pattern catalogs are available for many programming languages. Most of the traditional design patterns apply to any object-oriented programming language. One of the most popular books, *Design Patterns: Elements of Reusable Object-Oriented Software*, uses a combination of C++, Smalltalk, and diagrams to show possible pattern implementations. Many Java developers still reference this book because the concepts translate to any object-oriented language.
You learn more about design patterns and other Java best practices in the *Java Design Patterns* course.

Singleton Pattern

The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {
1   private static final SingletonClass instance =
            new SingletonClass();

2   private SingletonClass() {}

    public static SingletonClass getInstance() {
3       return instance;
    }
}
```

Implementing the Singleton Pattern

The singleton design pattern is one of the creational design patterns that are categorized in *Design Patterns: Elements of Reusable Object-Oriented Software*.
To implement the singleton design pattern:

1. Use a static reference to point to the single instance. Making the reference final ensures that it will never reference a different instance.
2. Add a single private constructor to the singleton class. The private modifier allows only "same class" access, which prohibits any attempts to instantiate the singleton class except for the attempt in step 1.
3. A public factory method returns a copy of the singleton reference. This method is declared static to access the static field declared in step 1. Step 1 could use a public variable, eliminating the need for the factory method. Factory methods provide greater flexibility (for example, implementing a per-thread singleton solution) and are typically used in most singleton implementations.

To obtain a singleton reference, call the `getInstance` method:

```
SingletonClass ref =
SingletonClass.getInstance();
```

Nested Classes

A nested class is a class declared within the body of another class. Nested classes:

- Have multiple categories
  - Inner classes
    - Member classes
    - Local classes
    - Anonymous classes
  - Static nested classes
- Are commonly used in applications with GUI elements
- Can limit utilization of a “helper class” to the enclosing toplevel Class

Reasons to Use Nested Classes

The following information is taken from http://download.oracle.com/javase/tutorial/java/javaOO/nested.html.

Logical Grouping of Classes

If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such “helper classes" makes their package more streamlined.

Increased Encapsulation

Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A’s members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More Readable, Maintainable Code

Nesting small classes within top-level classes places the code closer to where it is used.

Inner Class: Example

```
public class Car {
    private boolean running = false;
    private Engine engine = new Engine();

    private class Engine {
        public void start() {
            running = true;
        }
    }

    public void start() {
        engine.start();
    }
}
```

Inner Classes Versus Static Nested Classes

An inner class is considered part of the outer class and inherits access to all the private members of the outer class. The example in the slide shows an inner class, which is a member class. Inner classes can also be declared inside a method block (local classes).
A static nested class is not an inner class, but its declaration appears similar with an additional `static` modifier on the nested class. Static nested classes can be instantiated before the enclosing outer class and, therefore, are denied access to all non-static members of the enclosing class.

Anonymous Inner Classes

An anonymous class is used to define a class with no name.

```
public class AnonymousExampleClass {
    public Object o = new Object() {
        @Override
        public String toString() {
            return "In an anonymous class method";
        }
    };
}
```

A Class with No Name

In the example in the slide, the `java.lang.Object` class is being subclassed, and it is that subclass that is being instantiated. When you compile an application with anonymous classes, a separate class file following a naming convention of `Outer$1.class` will be generated, where `1` is the index number of anonymous classes in an enclosing class and `Outer` is the name of the enclosing class.
Anonymous inner classes can also be local classes.

Quiz

Which of the following nested class types are inner classes?

a. anonymous
b. local
c. static
d. member

Answer: a, b, d

Summary

In this lesson, you should have learned how to:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Model business problems by using the `static` and `final` keywords
- Implement the singleton design pattern
- Distinguish between top-level and nested classes

Practice 5-1 Overview: Applying the Abstract Keyword

This practice covers the following topics:

- Identifying potential problems that can be solved using abstract classes
- Refactoring an existing Java application to use abstractclasses and methods

Practice 5-2 Overview: Applying the Singleton Design Pattern

This practice covers using the static and final keywords and refactoring an existing application to implement the singleton design pattern.

Practice 5-3 Overview: (Optional) Using Java Enumerations

This practice covers taking an existing application and refactoring the code to use an enum.

(Optional) Practice 5-4 Overview: Recognizing Nested Classes

This practice covers analyzing an existing Java application and identifying the number and types of classes present.

Chapter 6

Inheritance with Java Interfaces

Objectives

After completing this lesson, you should be able to do the following:

- ❑ Model business problems by using interfaces
- ❑ Define a Java interface
- ❑ Choose between interface inheritance and class inheritance
- ❑ Extend an interface
- ❑ Refactor code to implement the DAO pattern

Implementation Substitution

The ability to outline abstract types is a powerful feature of Java. Abstraction enables:

- ❑ Ease of maintenance
  - Classes with logic errors can be substituted with new and improved classes.
- ❑ Implementation substitution
  - The `java.sql` package outlines the methods used by developers to communicate with databases, but the implementation is vendor-specific.
- ❑ Division of labor
  - Outlining the business API needed by an application's UI allows the UI and the business logic to be developed in tandem.

Abstraction

You just learned how to define abstract types by using classes. There are two ways to define type abstraction in Java: abstract classes and interfaces. By writing code to reference abstract types, you no longer depend on specific implementing classes. Defining these abstract types may seem like extra work in the beginning but can reduce refactoring at a later time if used appropriately.

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- ❑ Are similar to abstract classes containing only public abstract methods
- ❑ Outline methods that must be implemented by a class
  - Methods must not have an implementation {braces}.
- ❑ Can contain constant fields
- ❑ Can be used as a reference type
- ❑ Are an essential component of many design patterns

In Java, an interface outlines a contract for a class. The contract outlined by an interface mandates the methods that must be implemented in a class. Classes implementing the contract must fulfill the entire contract or be declared `abstract`.

Developing Java Interfaces

Public, top-level interfaces are declared in their own `.java` file. You implement interfaces instead of extending them.

```
public interface ElectronicDevice {
    public void turnOn();
    public void turnOff();
}
```

```
public class Television implements ElectronicDevice {
    public void turnOn() { }
    public void turnOff() { }
    public void changeChannel(int channel) {}
    private void initializeScreen() {}
}
```

Rules for Interfaces

Access Modifiers

All methods in an interface are `public`, even if you forget to declare them as `public`. You may not declare methods as `private` or `protected` in an interface. The contract that an interface outlines is a public API that must be provided by a class.

Abstract Modifier

Because all methods are implicitly `abstract`, it is redundant (but allowed) to declare a method `abstract`. Because all interface methods are abstract, you may not provide any method implementation, not even an empty set of braces.

Implements and Extends

A class can extend one parent class and then implement a comma-separated list of interfaces.

Constant Fields

Interfaces can have constant fields.

```
public interface ElectronicDevice {
    public static final String WARNING =
        "Do not open, shock hazard";
    public void turnOn();
    public void turnOff();
}
```

Only constant fields are permitted in an interface. When you declare a field in an interface, it is implicitly `public`, `static`, and `final`. You may redundantly specify these modifiers. Avoid grouping all the constant values for an application in a single interface; good design distributes the constant values of an application across many classes and interfaces. Creating monolithic classes or interfaces that contain large groupings of unrelated code does not follow best practices for object-oriented design.

Interface References

You can use an interface as a reference type. When using an interface reference type, you must use only the methods outlined in the interface.

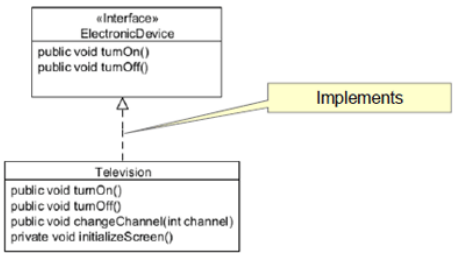
```
ElectronicDevice ed = new Television();
ed.turnOn();
ed.turnOff();
ed.changeChannel(2); // fails to compile
String s = ed.toString();
```

An interface-typed reference can be used only to reference an object that implements that interface. If an object has all the methods outlined in the interface but does not implement the interface, you may not use the interface as a reference type for that object. Interfaces implicitly include all the methods from `java.lang.Object`.

`instanceof` Operator

You can use `instanceof` with interfaces.

```
Television t = new Television();
if (t instanceof ElectronicDevice) { }
```



Television is an instance of an ElectronicDevice.

Previously, you used `instanceof` on class types. Any type that can be used as a reference can be used as an operand for the `instanceof` operator. In the slide, a `Television` implements `ElectronicDevice`. Therefore, a `Television` is an instance of a `Television`, an `ElectronicDevice`, and a `java.lang.Object`.

Marker Interfaces

- Marker interfaces define a type but do not outline any methods that must be implemented by a class.

```
public class Person implements java.io.Serializable { }
```

- The only reason these type of interfaces exist is type checking.

```
Person p = new Person();
if (p instanceof Serializable) {

}
```

`java.io.Serializable` is a marker interface used by Java's I/O library to determine if an object can have its state serialized. When implementing `Serializable`, you are not required to provide method implementations. Testing (in the form of the `instanceof` operator) for the serializability of an object is built into the standard I/O libraries. You use this interface in the lesson titled "Java I/O Fundamentals."

Casting to Interface Types

You can cast to an interface type.

```
public static void turnObjectOn(Object o) {
    if (o instanceof ElectronicDevice) {
        ElectronicDevice e = (ElectronicDevice)o;
        e.turnOn();
    }
}
```

Casting Guidelines

Just as you do when casting to class types, if you cast to a type that is invalid for that object, your application generates an exception and is even likely to crash. To verify that a cast will succeed, you should use an `instanceof` test.

The example in the slide shows poor design because the `turnObjectOn()` method operates only on `ElectronicDevices`. Using `instanceof` and casting adds overhead at run time. When possible, use a compile-time test by rewriting the method as:

```
public static void turnObjectOn
(ElectronicDevice e) {
  e.turnOn();
}
```

Using Generic Reference Types

- Use the most generic type of reference wherever possible:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

`EmployeeDAOMemoryImpl` implements `EmployeeDAO`

- By using an interface reference type, you can use a different implementing class without running the risk of breaking subsequent lines of code:

```
EmployeeDAOMemoryImpl dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

It is possible that you could be using `EmployeeDAOMemoryImpl` only methods here.

When creating references, you should use the most generic type possible. This means that, for the object you are instantiating, you should declare the reference to be of an interface type or of a parent class type. By doing this, all usage of the reference is not tied to a particular implementing class and, if need be, you could use a different implementing class. By using an interface that several classes implement as the reference type, you have the freedom to change the implementation without affecting your code. An `EmployeeDAOMemoryImpl` typed reference could be used to invoke a method that appears only in the `EmployeeDAOMemoryImpl` class. References typed to a specific class cause your code to be more tightly coupled to that class and potentially cause greater refactoring of your code when changing implementations.

Implementing and Extending

- Classes can extend a parent class and implement an interface:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat { }
```

- You can also implement multiple interfaces:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Use a comma to separate your list of interfaces.

Extends First

If you use both `extends` and `implements`, `extends` must come first.

Extending Interfaces

- Interfaces can extend interfaces:

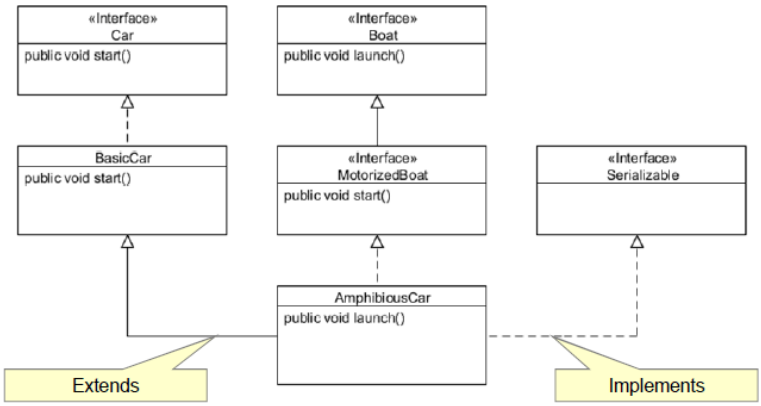
```
public interface Boat { }
```

```
public interface MotorizedBoat extends Boat { }
```

- By implementing `MotorizedBoat`, the `AmphibiousCar` class must fulfill the contract outlined by both `MotorizedBoat` and `Boat`:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Interfaces in Inheritance Hierarchies



Interface Inheritances

Interfaces are used for a form of inheritance that is referred to as *interface inheritance*. Java allows multiple interface inheritance but only single class inheritance.

Extending an Implementing Class

If you write a class that extends a class that implements an interface, the class you authored also implements the interface. For example, `AmphibiousCar` extends `BasicCar`. `BasicCar` implements `Car`; therefore, `AmphibiousCar` also implements `Car`.

Interfaces Extending Interfaces

An interface can extend another interface. For example, the interface `MotorizedBoat` can extend the `Boat` interface. If the `AmphibiousCar` class implements `MotorizedBoat`, then it must implement all methods from `Boat` and `MotorizedBoat`.

Duplicate Methods

When you have a class that implements multiple interfaces, directly or indirectly, the same method signature may appear in different implemented interfaces. If the signatures are the same, there is no conflict and only one implementation is required.

Quiz

A class can implement multiple interfaces.

a. True
b. False

Answer: a

Design Patterns and Interfaces

- One of the principles of object-oriented design is to: *"Program to an interface, not an implementation."*
- This is a common theme in many design patterns. This principle plays a role in:
  - The DAO design pattern
  - The Factory design pattern

Object-Oriented Design Principles

"Program to an interface, not an implementation" is a practice that was popularized in the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

You can learn more about object-oriented design principles and design patterns in the *Java Design Patterns* course.

DAO Pattern

The Data Access Object (DAO) pattern is used when creating an application that must persist information. The DAO pattern:

- Separates the problem domain from the persistence mechanism
- Uses an interface to define the methods used for persistence. An interface allows the persistence implementation to be replaced with:
  - Memory-based DAOs as a temporary solution
  - File-based DAOs for an initial release
  - JDBC-based DAOs to support database persistence
  - Java Persistence API (JPA)-based DAOs to support database persistence

Why Separate Business and Persistence Code?

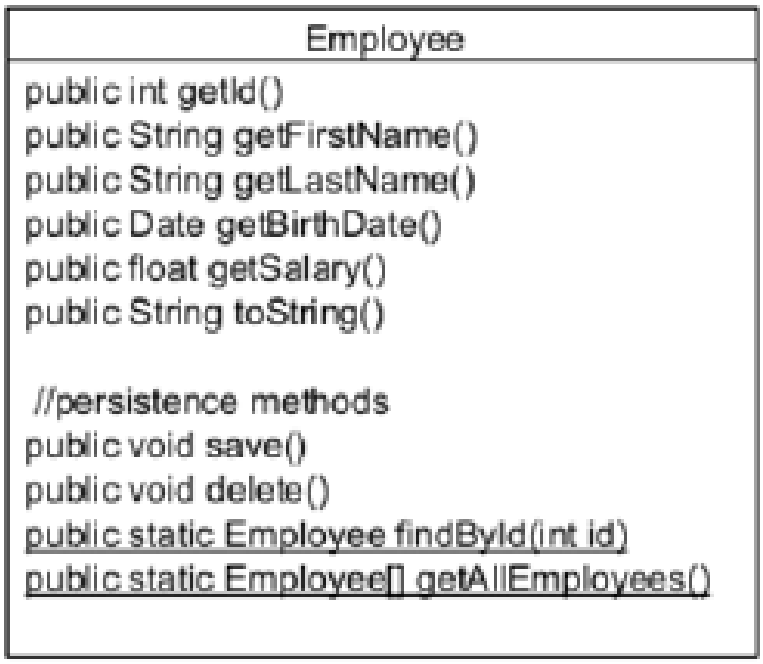
Just as the required functionality of an application will influence the design of your classes, so will other concerns. A desire for ease of maintenance and for the ability to enhance an application also influences its design. Modular code that is separated by functionality is easier to update and maintain.

By separating business and persistence logic, applications become easier to implement and maintain at the expense of additional classes and interfaces. Often these two types of logic have different maintenance cycles. For example, the persistence logic might need to be modified if the database used by the application was migrated from MySQL to Oracle 11*g*.

If you create interfaces for the classes containing the persistence logic, it becomes easier for you to replace your persistence implementation.

Before the DAO Pattern

Notice the persistence methods mixed in with the business methods.



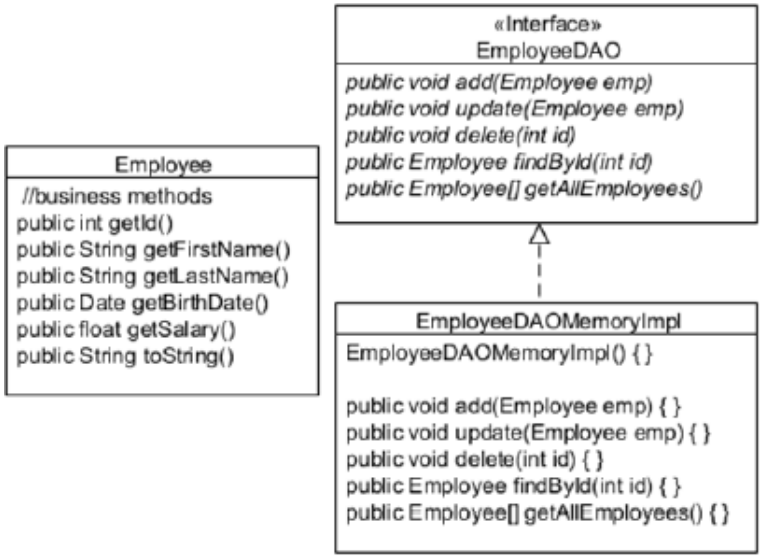
Before the DAO Pattern

The Single-Responsibility Principle

The `Employee` class shown in the slide has methods that focus on two different principles or concerns. One set of methods focuses on manipulating the representation of a person, and the other deals with persisting `Employee` objects. Because these two sets of responsibilities may be modified at different points in the lifetime of your applications, it makes sense to split them up into different classes.

After the DAO Pattern

The DAO pattern moves the persistence logic out of the domain classes and into separate classes.



After Refactoring to the DAO Pattern

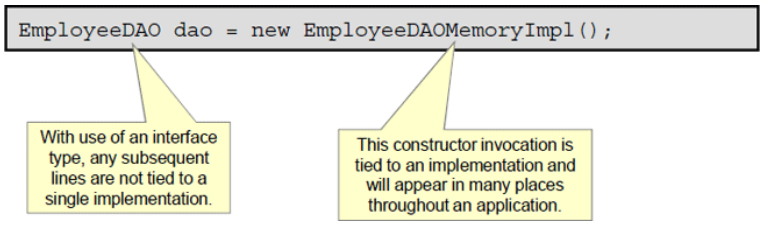
DAO Implementations

If you think that you might need to change your DAO implementation at a later time to use a different persistence mechanism, it is best to use an interface to define the contract that all DAO implementations must meet.

Your DAO interfaces outline methods to create, read, update, and delete data, although the method names can vary. When you first implement the DAO pattern, you will not see the benefit immediately. The payoff comes later, when you start modifying or replacing code. In the lesson titled "Building Database Applications with JDBC," we discuss replacing the memory-based DAO with file- and database-enabled DAOs.

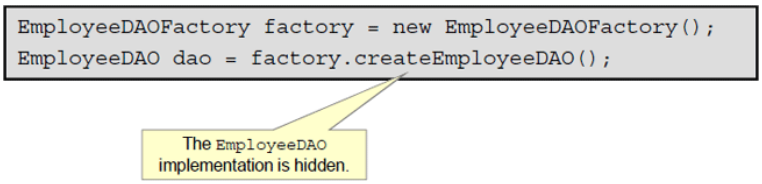
The Need for the Factory Pattern

The DAO pattern depends on using interfaces to define an abstraction. Using a DAO implementation's constructor ties you to a specific implementation.



Using the Factory Pattern

Using a factory prevents your application from being tightly coupled to a specific DAO implementation.

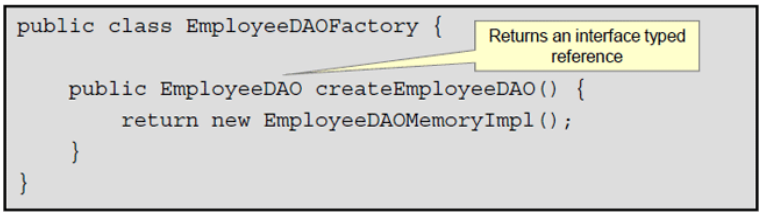


This pattern eliminates direct constructor calls in favor of invoking a method. A factory is often used when implementing the DAO pattern.

In the example in the slide, you have no idea what type of persistence mechanism is used by `EmployeeDAO` because it is just an interface. The factory could return a DAO implementation that uses files or a database to store and retrieve data. As a developer, you want to know what type of persistence is being used because it factors into the performance and reliability of your application. But you do not want the majority of the code you write to be tightly coupled with the type of persistence.

The Factory

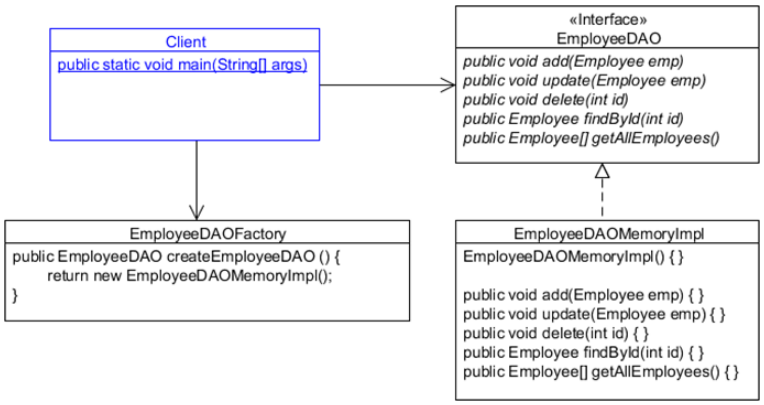
The implementation of the factory is the only point in the application that should depend on concrete DAO classes.



For simplicity, this factory hardcodes the name of a concrete class to instantiate. You could enhance this factory by putting the class name in an external source such as a text file and use the `java.lang.Class` class to instantiate the concrete subclass. A basic example of using the `java.lang.Class` is the following:

```
String name =
"com.example.dao.EmployeeDAOMemoryImpl"
;
Class clazz = Class.forName(name);
EmployeDAO dao = (EmployeeDAO)
clazz.newInstance();
```

The DAO and Factory Together



Clients Depending only on Abstract DAOs

Quiz

A typical singleton implementation contains a factory method.

a. True
b. False

Answer: a

Code Reuse

Code duplication (copy and paste) can lead to maintenance problems. You do not want to fix the same bug multiple times.

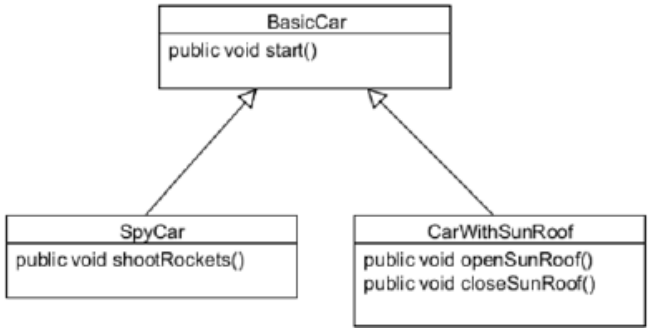
- ❑ "Don't repeat yourself!" (DRY principle)
- ❑ Reuse code in a good way:
  - Refactor commonly used routines into libraries.
  - Move the behavior shared by sibling classes into their parent class.
  - Create new combinations of behaviors by combining multiple types of objects together (composition).

Copying and pasting code is *not* something that must always be avoided. If duplicated code serves as a starting point and is heavily modified, that may be an acceptable situation for you to copy and paste lines of code. You should be aware of how much copying and pasting is occurring in a project. Besides performing manual code audits, there are tools you can use to detect duplicated code. For one such example, refer to http://pmd.sourceforge.net/cpd.html.

Design Difficulties

Class inheritance allows for code reuse but is not very modular.

- ❑ How do you create a SpyCarWithSunRoof?



Method Implementations Located Across Different Classes

Limitations with Inheritance

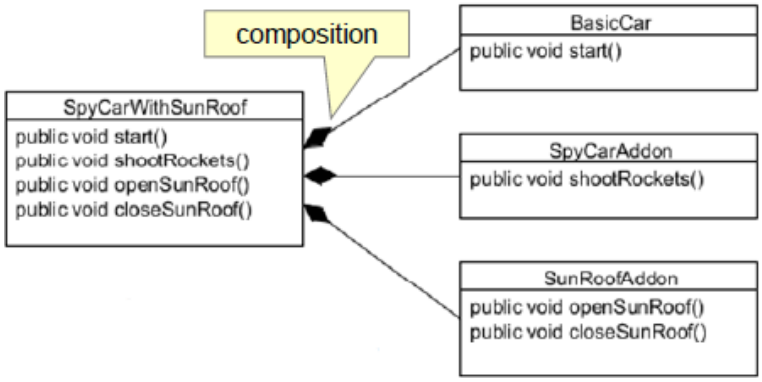
Java supports only single class inheritance, which eliminates the possibility of inheriting different implementations of a method with the same signature. Multiple interface inheritance does not pose the same problem as class inheritance because there can be no conflicting method implementations in interfaces.

Composition

Object composition allows you to create more complex objects.

To implement composition, you:

1. Create a class with references to other classes.
2. Add same signature methods that forward to the referenced objects.



Combining Multiple Classes' Methods Through Composition and Forwarding

Delegation

Method delegation and method forwarding are two terms that are often used interchangeably. Method forwarding is when you write a method that does nothing except pass execution over to another method. In some cases, delegation may imply more than simple forwarding. For more on the difference between the two, refer to page 20 of the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

Composition Implementation

```
public class SpyCarWithSunRoof {
    private BasicCar car = new BasicCar();
    private SpyCarAddon spyAddon = new SpyCarAddon();
    private SunRoofAddon roofAddon = new SunRoofAddon();

    public void start() {
        car.start();
    }

    // other forwarded methods
}
```

Method forwarding

IDE Wizards Make Implementing Composition Easy

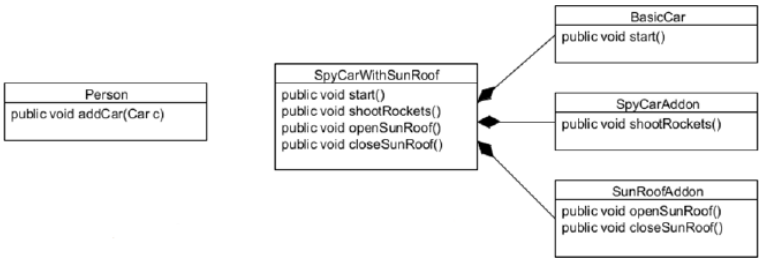
To implement composition with the NetBeans IDE, use the Insert Code tool as follows:

1. Right-click within the braces of the complex class and choose "Insert Code."
2. Select "Delegate Method." The Generate Delegate Methods dialog box appears.
3. Select the method calls that you want to forward. The methods are inserted for you.

Repeat these steps for each delegate class.

Polymorphism and Composition

Polymorphism should enable us to pass any type of Car to the `addCar` method. Composition does not enable polymorphism unless.



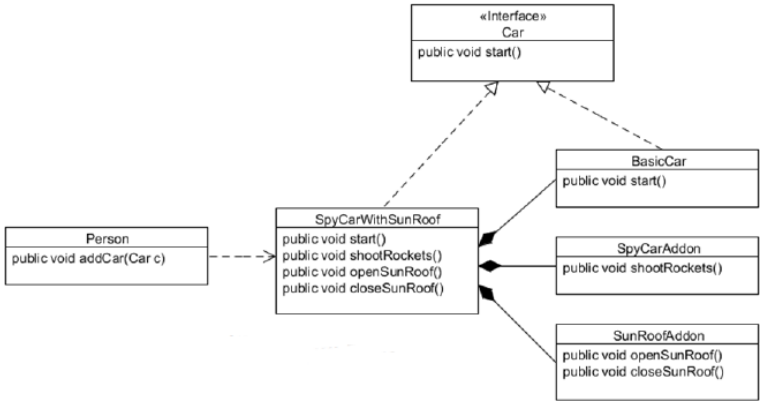
A Complex Car Object that Cannot be Passed to a Method Expecting a Simple Car

Code Reuse

The ability to use the `addCar` method for any type of Car, no matter how complex, is another form of code reuse. We cannot currently say the following:

```
addCar(new SpyCarWithSunRoof());
```

Use interfaces for all delegate classes to support polymorphism.



Composition with Interfaces to Support Polymorphism

Each delegate class that you use in a composition should have an interface defined. When creating the composing class, you declare that it implements all of the delegate interface types. By doing this, you create an object that is a composition of other objects and has many types.

Now we can say:

```
addCar(new SpyCarWithSunRoof());
```

Quiz

Method delegation is required to create complex objects using:

a. Polymorphism
b. Composition

Answer: b

Summary

In this lesson, you should have learned how to:

- ❑ Model business problems by using interfaces
- ❑ Define a Java interface
- ❑ Choose between interface inheritance and class inheritance
- ❑ Extend an interface
- ❑ Refactor code to implement the DAO pattern

Practice 6-1 Overview: Implementing an Interface

This practice covers the following topics:

- ❑ Writing an interface
- ❑ Implementing an interface
- ❑ Creating references of an interface type
- ❑ Casting to interface types

Practice 6-2 Overview: Applying the DAO Pattern

This practice covers the following topics:

- ❑ Rewriting an existing domain object with a memory-based persistence implementation using the DAO pattern
- ❑ Leveraging an abstract factory to avoid depending on concrete implementations

(Optional) Practice 6-3 Overview: Implementing Composition

This practice covers the following topics:

- ❑ Rewriting an existing application to better support code reuse through composition
- ❑ Using interfaces to enable polymorphism

Chapter 7

Generics and Collections

Objectives

After completing this lesson, you should be able to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a stack by using a deque
- Use enumerated types

Generics

- Provide flexible type safety to your code
- Move many common errors from runtime to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Cache Class Without Generics

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

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

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

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```

The two examples in the slide show very simple caching classes. Even though each class is very simple, a separate class is required for any object type.

Generic Cache Class

```
1 public class CacheAny <T>{
2
3    private T t;
4
5    public void add(T t){
6        this.t = t;
7    }
8
9    public T get(){
10       return this.t;
11    }
12 }
```

To create a generic version of the `CacheAny` class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for "type" and can represent any type. As the example shows, the code has changed to use `t` instead of a specific type information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. A number of letters are commonly used with generics.

Note: You can use any identifier you want. The following values are merely strongly suggested.

Here are the conventions:

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S`, `U`: Used if there are second types, third types, or more

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
1 public static void main(String args[]){
2    CacheString myMessage = new CacheString(); // Type
3    CacheShirt myShirt = new CacheShirt();     // Type
4
5    //Generics
6    CacheAny<String> myGenericMessage = new CacheAny<String>();
7    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();
8
9    myMessage.add("Save this for me"); // Type
10    myGenericMessage.add("Save this for me"); // Generic
11
12 }
```

Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration is changed to generic, no changes need to be made to the remaining code.

The example code can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Type Inference Diamond

- Syntax
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

The type inference diamond is a new feature in JDK 7. In the generic code, notice how the right-side type definition is always equivalent to the left-side type definition. In JDK 7, you can use the diamond to indicate that the right type definition is equivalent to the left. This helps to avoid typing redundant information over and over again.

Example: `TestCacheAnyDiamond.java`

Note: In a way, it works in an opposite way from a "normal" Java type assignment. For example, `Employee emp = new Manager();` makes `emp` object an instance of `Manager`.

But in the case of generics:

`ArrayList<Manager> managementTeam = new ArrayList<>();`

It is the left side of the expression (rather than the right side) that determines the type.

Quiz

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

a. T: Table
b. E: Element
c. K: Key
d. V: Value

Answer: a

Collections

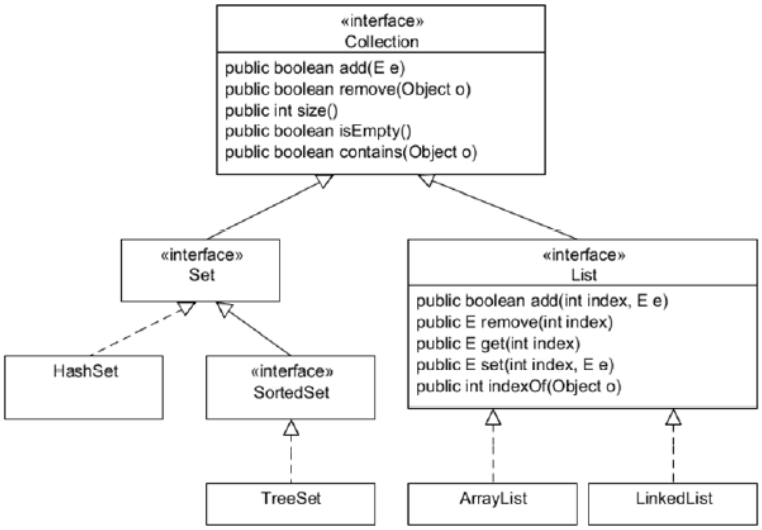
- A collection is a single object designed to manage a group of objects:
  - Objects in a collection are called *elements.*
  - *Primitives are not allowed in a collection.*
- Various collection types implement many common data structures:
  - Stack, queue, dynamic array, hash
- The Collections API relies heavily on generics for its implementation.

A collection is a single object that manages a group of objects. Objects in the collection are called *elements*. Various collection types implement standard data structures including stacks, queues, dynamic arrays, and hashes. All the collection objects have been optimized for use in Java applications.

Note: The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

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

Collection Types



The diagram in the slide shows all the collection types that descend from `Collection`. Some sample methods are provided for both `Collection` and `List`. Note the use of generics.

Characteristics of Implementation Classes

- `HashSet`: A collection of elements that contains no duplicate elements
- `TreeSet`: A sorted collection of elements that contains no duplicate elements
- `ArrayList`: A dynamic array implementation
- `Deque`: A collection that can be used to implement a stack or a queue

Note: The `Map` interface is a separate inheritance tree and is discussed later in the lesson.

`List` Interface

- `List` is an interface that defines generic list behavior.

- An ordered collection of elements

`List` behaviors include:

- Adding elements at a specific index
- Adding elements to the end of the list
- Getting an element based on an index
- Removing an element based on an index
- Overwriting an element based on an index
- Getting the size of the list

- Use `List` as a reference type to hide implementation details.

The `List` interface is the basis for all Collections classes that exhibit list behavior.

`ArrayList` Implementation Class

- Is a dynamically growable array
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

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

An `ArrayList` implements a `List` collection. The implementation exhibits characteristics of a dynamically growing array. A "to-do list" application is a good example of an application that can benefit from an `ArrayList`.

`ArrayList` Without Generics

```
1  public class OldStyleArrayList {
2    public static void main(String args[]){
3      List partList = new ArrayList(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Oops a string!");
9
10     Iterator elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = (Integer)(elements.next()); // error?
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

Java example using syntax prior to Java 1.5.

Runtime error: ClassCastException

In the example in the slide, a part number list is created using an `ArrayList`. Using syntax prior to Java version 1.5, there is no type definition. So any type can be added to the list as shown on line 8. So it is up to the programmer to know what objects are in the list and in what order. If an assumption were made that the list was only for `Integer` objects, a runtime error would occur on line 12.

On lines 10-16, with a non-generic collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless "syntactic sugar" (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111
Part number: 2222
Part number: 3333
```

Generic `ArrayList`

```
1  public class GenericArrayList {
2    public static void main(String args[]) {
3      List<Integer> partList = new ArrayList<>(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Bad Data");  // compile error now
9
10     Iterator<Integer> elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = elements.next();
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

Java example using SE 7 syntax.

No cast required.

With generics, things are much simpler. When the `ArrayList` is initialized on line 3, any attempt to add an invalid value (line 8) results in a compile-time error.

Note: On line 3, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

Generic `ArrayList`: Iteration and Boxing

```
for (Integer partNumberObj:partList){
      int partNumber = partNumberObj; // Demos auto unboxing
      System.out.println("Part number: " + partNumber);
}
```

- The enhanced `for` loop, or `for-each` loop, provides cleaner code.
- No casting is done because of autoboxing and unboxing.

Using the `for-each` loop is much easier and provides

much cleaner code. No casts are done because of the autounboxing feature of Java.

Autoboxing and Unboxing

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1 public class AutoBox {
2    public static void main(String[] args){
3        Integer intObject = new Integer(1);
4        int intPrimitive = 2;
5
6        Integer tempInteger;
7        int tempPrimitive;
8
9        tempInteger = new Integer(intPrimitive);
10        tempPrimitive = intObject.intValue();
11
12        tempInteger = intPrimitive; // Auto box
13        tempPrimitive = intObject;  // Auto unbox
```

Lines 9 and 10 show a traditional method for moving between objects and primitives. Lines 12 and 13 show boxing and unboxing.

Autoboxing and Unboxing

Autoboxing and unboxing are Java language features that enable you to make sensible assignments without formal casting syntax. Java provides the casts for you at compile time.

Note: Be careful when using autoboxing in a loop. There is a performance cost to using this feature.

Quiz

Assuming a valid Employee class, and given this fragment:

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

What change is required to allow this code to compile?

a. Line 1: The (3) needs to be (4)
b. Line 8: Need to cast `elements.next()` to Employee before invoking `getName()`
c. Line 1: `Object` needs to be `Employee`
d. Line 5: The 3 needs to be 4

Answer: c

`Set` Interface

- A set is a list that contains only unique elements.
- A set has no index.
- Duplicate elements are not allowed in a set.
- You can iterate through elements to access them.
- `TreeSet` provides sorted implementation.

As an example, a set can be used to track a list of unique part numbers.

`Set` Interface: Example

A set is a collection of unique elements.

```
1 public class SetExample {
2    public static void main(String[] args){
3        Set<String> set = new TreeSet<>();
4
5        set.add("one");
6        set.add("two");
7        set.add("three");
8        set.add("three"); // not added, only unique
9
10        for (String item:set){
11            System.out.println("Item: " + item);
12        }
13    }
14 }
```

A set is a collection of unique elements. This example uses a `TreeSet`, which sorts the items in the set. If the program is run, the output is as follows:

```
Item: one
Item: three
Item: two
```

`Map` Interface

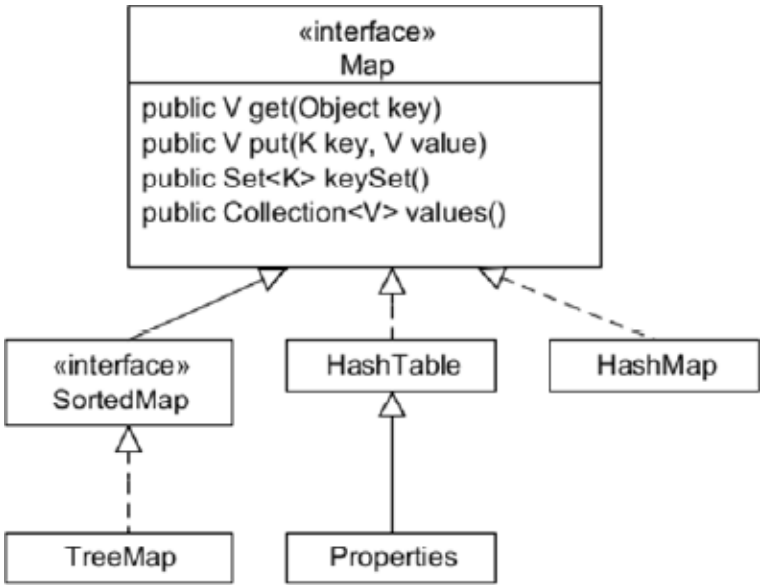
- A collection that stores multiple key-value pairs
  - Key: Unique identifier for each element in a collection
  - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key Value

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

A `Map` is good for tracking things like part lists and their descriptions (as shown in the slide).

`Map` Types



The `Map` interface does not extend the `Collection` interface because it represents mappings and not a collection of objects. Some of the key implementation classes include:

- `TreeMap`: A map where the keys are automatically sorted.
- `HashTable`: A classic associative array implementation with keys and values. `HashTable` is synchronized.
- `HashMap`: An implementation just like `HashTable` except that it accepts null keys and values. Also, it is not synchronized.

`Map` Interface: Example

```
1 public class MapExample {
2    public static void main(String[] args){
3        Map <String, String> partList = new TreeMap<>();
4        partList.put("S001", "Blue Polo Shirt");
5        partList.put("S002", "Black Polo Shirt");
6        partList.put("H001", "Duke Hat");
7
8        partList.put("S002", "Black T-Shirt"); // Overwrite value
9        Set<String> keys = partList.keySet();
10
11        System.out.println("=== Part List ===");
12        for (String key:keys){
13            System.out.println("Part#: " + key + " " +
14                               partList.get(key));
15        }
16    }
17 }
```

The example shows how to create a `Map` and perform standard operations on it. The output from the program is:

```
=== Part List ===
Part#: 111111 Blue Polo Shirt
Part#: 222222 Black T-Shirt
Part#: 333333 Duke Hat
```

`Deque` Interface

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

- Means "double-ended queue" (and is pronounced "deck")
- A queue provides FIFO (first in, first out) operations
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations
  - `push(e)` and `pop()` methods

`Deque` is a child interface of `Collection` (just like `Set` and `List`).

A queue is often used to track asynchronous message requests so they can be processed in order. A stack can be very useful for traversing a directory tree or similar structures.

Stack with Deque: Example

```
1 public class TestStack {
2    public static void main(String[] args){
3        Deque<String> stack = new ArrayDeque<>();
4        stack.push("one");
5        stack.push("two");
6        stack.push("three");
7
8        int size = stack.size() - 1;
9        while (size >= 0 ) {
10            System.out.println(stack.pop());
11            size--;
12        }
13    }
14 }
```

A deque (pronounced "deck") is a "doubled-ended queue." Essentially this means that a deque can be used as a queue (first in, first out [FIFO] operations) or as a stack (last in, first out [LIFO] operations).

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
  - Both are implemented using generics.
- Using the `Comparable` interface:
  - Overrides the `compareTo` method
  - Provides only one sort option
- Using the `Comparator` interface:
  - Is implemented by using the `compare` method
  - Enables you to create multiple `Comparator` classes
  - Enables you to create and use numerous

sorting options

The Collections API provides two interfaces for ordering elements: `Comparable` and `Comparator`.

The `Comparable` interface is implemented in a class and provides a single sorting option for the class.

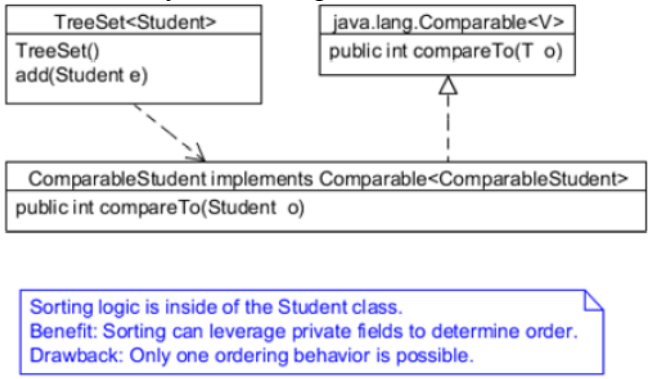
The `Comparator` interface enables you to create multiple sorting options. You plug in the designed option whenever you want.

Both interfaces can be used with sorted collections, such as `TreeSet` and `TreeMap`.

`Comparable` Interface

Using the `Comparable` interface:

- Overrides the `compareTo` method
- Provides only one sort option



The slide shows how the `ComparableStudent` class relates to the `Comparable` interface and `TreeSet`.

`Comparable`: Example

```
1 public class ComparableStudent implements Comparable<ComparableStudent>{
2    private String name; private long id = 0; private double gpa = 0.0;
3
4    public ComparableStudent(String name, long id, double gpa){
5       // Additional code here
6    }
7    public String getName(){ return this.name; }
8       // Additional code here
9
10    public int compareTo(ComparableStudent s){
11       int result = this.name.compareTo(s.getName());
12       if (result > 0) { return 1; }
13       else if (result < 0){ return -1; }
14       else { return 0; }
15    }
16 }
```

This example in the slide implements the `Comparable` interface and its `compareTo` method. Notice that because the interface is designed using generics, the angle brackets define the class type that is passed into the `compareTo` method. The `if` statements are included to demonstrate the comparisons that take place. You can also merely return a result.

The returned numbers have the following meaning.

- **Negative number**: `s` comes before the current element.
- **Positive number**: `s` comes after the current element.
- **Zero**: `s` is equal to the current element.

In cases where the collection contains equivalent values, replace the code that returns zero with additional code that returns a negative or positive number.

`Comparable` Test: Example

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

    studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));
    studentList.add(new ComparableStudent("John Adams", 2222, 3.9));
    studentList.add(new ComparableStudent("George Washington", 3333, 3.4));

    for(ComparableStudent student:studentList){
      System.out.println(student);
    }
  }
}
```

In the example in the slide, an `ArrayList` of `ComparableStudent` elements is created. After the list is initialized, it is sorted using the `Comparable` interface. The output of the program is as follows:

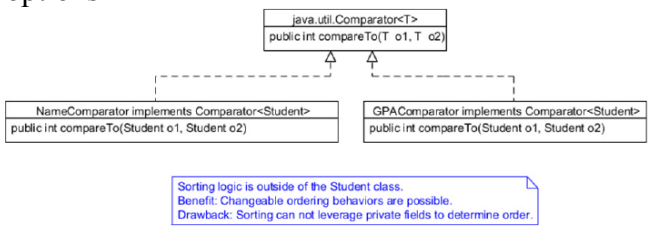
```
Name: George Washington ID: 3333 GPA:3.4
Name: John Adams ID: 2222 GPA:3.9
Name: Thomas Jefferson ID: 1111 GPA:3.8
```

Note: The `ComparableStudent` class has overridden the `toString()` method.

`Comparator` Interface

Using the `Comparator` interface:

- Is implemented by using the `compare` method
- Enables you to create multiple `Comparator` classes
- Enables you to create and use numerous sorting options



The slide shows two `Comparator` classes that can be used with the `Student` class. The example in the next slide shows how to use `Comparator` with an unsorted interface like `ArrayList` by using the `Collections` utility class.

Comparator: Example

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

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

Here the compare logic is reversed and results in descending order.

The example in the slide shows the `Comparator` classes that are created to sort based on Name and GPA. For the name comparison, the `if` statements have been simplified.

Comparator Test: Example

```
1  public class TestComparator {
2    public static void main(String[] args){
3        List<Student> studentList = new ArrayList<>(3);
4        Comparator<Student> sortName = new StudentSortName();
5        Comparator<Student> sortGpa = new StudentSortGpa();
6
7        // Initialize list here
8
9        Collections.sort(studentList, sortName);
10        for(Student student:studentList){
11            System.out.println(student);
12        }
13
14        Collections.sort(studentList, sortGpa);
15        for(Student student:studentList){
16            System.out.println(student);
17        }
18    }
19 }
```

The example in the slide shows how the two `Comparator` objects are used with a collection.

Note: Some code has been commented out to save space.

Notice how the `Comparator` objects are initialized on lines 4 and 5. After the `sortName` and `sortGpa` variables are created, they can be passed to the `sort()` method by name. Running the program produces the following output.

```
Name: George Washington ID: 3333 GPA:3.4
Name: John Adams ID: 2222 GPA:3.9
Name: Thomas Jefferson ID: 1111 GPA:3.8
Name: John Adams ID: 2222 GPA:3.9
Name: Thomas Jefferson ID: 1111 GPA:3.8
Name: George Washington ID: 3333 GPA:3.4
```

Notes

- ❑ The `Collections` utility class provides a number of useful methods for various collections. Methods include `min()`, `max()`, `copy()`, and `sort()`.
- ❑ The `Student` class has overridden the `toString()` method.

Quiz

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

a. `Comparable`
b. `Comparison`
c. `Comparator`
d. `Comparinator`

Answer: c

Summary

In this lesson, you should have learned how to:

- ❑ Create a custom generic class
- ❑ Use the type inference diamond to create an object
- ❑ Create a collection without using generics
- ❑ Create a collection by using generics
- ❑ Implement an `ArrayList`
- ❑ Implement a `Set`
- ❑ Implement a `HashMap`
- ❑ Implement a stack by using a deque
- ❑ Use enumerated types

Practice 7-1 Overview: Counting Part Numbers by Using a HashMap

This practice covers the following topics:

- ❑ Creating a map to store a part number and count
- ❑ Creating a map to store a part number and description
- ❑ Processing the list of parts and producing a report

Practice 7-2 Overview: Matching Parentheses by Using a Deque

This practice covers processing programming statements to ensure that the number of parentheses matches.

Practice 7-3 Overview: Counting Inventory and Sorting with

Comparators

This practice covers processing inventory transactions that generate two reports sorted differently using `Comparators`.

Chapter 8

String Processing

Objectives

After completing this lesson, you should be able to:

- Read data from the command line
- Search strings
- Parse strings
- Create strings by using a `StringBuilder`
- Search strings by using regular expressions
- Parse strings by using regular expressions
- Replace strings by using regular expressions

Command-Line Arguments

- Any Java technology application can use command-line arguments.
- These string arguments are placed on the command line to launch the Java interpreter after the class name:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line argument is placed in the args array that is passed to the static main method:

```
public static void main(String[] args)
```

When a Java program is launched from a terminal window, you can provide the program with zero or more command-line arguments.

These command-line arguments enable the user to specify the configuration information for the application. These arguments are strings: either stand-alone tokens (such as arg1) or quoted strings (such as "another arg").

```
public class TestArgs {
   public static void main(String[] args) {
     for ( int i = 0; i < args.length; i++ ) {
        System.out.println("args[" + i + "] is '" +
                           args[i] + "'");
     }
   }
}
```

Example execution:

```
java TestArgs "Ted Baxter" 45 100.25
args[0] is 'Ted Baxter'
args[1] is '45'
args[2] is '100.25'
```

Command-line arguments are always passed to the main method as strings, regardless of their intended type. If an application requires command-line arguments other than type `String` (for example, numeric values), the application should convert the string arguments to their respective types using the wrapper classes, such as the `Integer.parseInt` method, which can be used to convert the string argument that represents the numeric integer to type `int`.

Properties

- The `java.util.Properties` class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file:

```
hostName = www.example.com
userName = user
password = pass
```

- File name ends in `.properties`.
- File can be anywhere that compiler can find it.

The benefit of a properties file is the ability to set values for your application externally. The properties file is typically read at the start of the application and is used for default values. But the properties file can also be an integral part of a localization scheme, where you store the values of menu labels and text for various languages that your application may support.

The convention for a properties file is `<filename>.properties`, but the file can have any extension you want. The file can be located anywhere that the application can find it.

Loading and Using a Properties File

```
1   public static void main(String[] args) {
2     Properties myProps = new Properties();
3     try {
4       FileInputStream fis = new FileInputStream("ServerInfo.properties");
5       myProps.load(fis);
6     } catch (IOException e) {
7       System.out.println("Error: " + e.getMessage());
8     }
9
10     // Print Values
11     System.out.println("Server: " + myProps.getProperty("hostName"));
12     System.out.println("User: " + myProps.getProperty("userName"));
13     System.out.println("Password: " + myProps.getProperty("password"));
14  }
```

In the code fragment, you create a `Properties` object. Then, using a `try` statement, you open a file relative to the source files in your NetBeans project. When it is loaded, the name-value pairs are available for use in your application.

Properties files enable you to easily inject configuration information or other application data into the application.

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the -D option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values

```
java -Dusername=user myApp
```

- Get the `Properties` data from the `System` object:

```
String userName = System.getProperty("username");
```

Property information can also be passed on the command line. The advantage to passing properties from the command line is simplicity. You do not have to open a file and read from it. However, if you have more than a few parameters, a properties file is preferable.

PrintWriter and the Console

The `PrintWriter` class writes characters instead of bytes. The class implements all of the print methods found in `PrintStream`.

```
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is some output.");

    }
}
```

The `PrintStream` class converts characters into bytes using the platform's default character encoding.
Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character is included in the output.
The example in the slide shows how to create the object using the `autoFlush` option. The `true` option is required to force `PrintWriter` to flush each line printed to the console.

`printf` format

Java provides many ways to format strings:

- `printf` and `String.format`

```
public class PrintfExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        double price = 24.99; int quantity = 2; String color = "Blue";
        System.out.printf("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        System.out.format("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        String out = String.format("We have %03d %s Polo shirts that cost
    $%3.2f.", quantity, color, price);
        System.out.println(out);
        pw.printf("We have %03d %s Polo shirts that cost $%3.2f.\n",
    quantity, color, price);
    }
}
```

You can perform the `printf` format using both the String class and any output stream. The slide shows several different string formatting examples. See the Java API documentation for details about all the options.

- **%s**: String
- **%d**: Decimal
- **%f**: Float

The program output is the following:

```
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
```

Quiz

Which two of the following are valid formatted print statements?

a. `System.out.printf("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`
b. `System.out.format("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`
c. `System.out.println("Red Polo shirts cost $35.00.\n");`
d. `System.out.print("Red Polo shirts cost $35.00.\n");`

Answer: a, b

String Processing

- `StringBuilder` for constructing string
- Built-in string methods
  - Searching
  - Parsing
  - Extracting substring
- Parsing with `StringTokenizer`

The first part of this section covers string functions that are not regular expressions. When you perform simple string manipulations, there are a number of very useful built-in methods.

StringBuilder and StringBuffer

- `StringBuilder` and `StringBuffer` are the preferred tools when string concatenation is nontrivial.
  - More efficient than "+"
- Concurrency
  - `StringBuilder` (not thread-safe)
  - `StringBuffer` (thread-safe)
- Set capacity to the size you actually need.
  - Constant buffer resizing can also lead to performance problems.

The `StringBuilder` and `StringBuffer` classes are the preferred way to concatenate strings.

StringBuilder: Example

```
public class StringBuilding {
    public static void main(String[] args){
        StringBuilder sb = new StringBuilder(500);

        sb.append(", the lightning flashed and the thunder
    rumbled.\n");
        sb.insert(0, "It was a dark and stormy night");

        sb.append("The lightning struck...\n").append("[  ");
        for(int i = 1; i < 11; i++){
            sb.append(i).append("  ");
        }
        sb.append("] times");

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

The example in the slide shows some common `StringBuilder` methods. You can use `StringBuilder` to insert text in position. Chaining append calls together is a best practice for building strings. The output from the program is as follows:

```
It was a dark and stormy night, the
lightning flashed and the thunder
rumbled.
The lightning struck...
[ 1 2 3 4 5 6 7 8 9 10 ] times
```

Sample String Methods

```
1   public class StringMethodsExample {
2       public static void main(String[] args){
3           PrintWriter pw = new PrintWriter(System.out, true);
4           String tc01 = "It was the best of times";
5           String tc02 = "It was the worst of times";
6
7           if (tc01.equals(tc02)){
8               pw.println("Strings match..."); }
9           if (tc01.contains("It was")){
10              pw.println("It was found"); }
11          String temp = tc02.replace("w", "zw");
12          pw.println(temp);
13          pw.println(tc02.substring(5, 12));
14      }
15  }
```

The code in the slide demonstrates some of the more useful string methods of the String class.

- **`equals()`**: Tests the equality of the contents of two strings. This is preferable to ==, which tests whether two objects point to the same reference.
- **`contains()`**: Searches a string to see if it contains the string provided.
- **`replace()`**: Searches for the string provided and replaces all instances with the target string provided. There is a `replaceFirst()` method for replacing only the first instance.
- **`substring()`**: Returns a string based on its position in the string.

Running the programs in the slide returns the following output:

```
It was found
It zwas the zworst of times
s the w
```

Using the `split()` Method

```
1   public class StringSplit {
2       public static void main(String[] args){
3           String shirts = "Blue Shirt, Red Shirt, Black
    Shirt, Maroon Shirt";
4
5           String[] results = shirts.split(", ");
6           for(String shirtStr:results){
7               System.out.println(shirtStr);
8           }
9       }
10  }
```

The simplest way to parse a string is using the `split()` method. Call the method with the character (or characters) that will split the string apart. The result is captured in an array.

Note: The delimiter can be defined using regular expressions.

The output of the program in the slide is as follows:

```
Blue Shirt
Red Shirt
```

```
Black Shirt
Maroon Shirt
```

Parsing with `StringTokenizer`

```
1   public class StringTokenizerExample {
2       public static void main(String[] args){
3           String shirts = "Blue Shirt, Red Shirt, Black Shirt, Maroon
    Shirt";
4
5           StringTokenizer st = new StringTokenizer(shirts, ", ");
6
7           while(st.hasMoreTokens()){
8               System.out.println(st.nextToken());
9           }
10      }
11  }
```

The `StringTokenizer` class does the same thing as `split()` but takes a different approach. You must iterate the tokens to get access to them. Also note that the delimiter "`,  `" in this case means use both commas and spaces as delimiters. Thus, the result from parsing is the following:

```
Blue
Shirt
Red
Shirt
Black
Shirt
Maroon
Shirt
```

Scanner

A `Scanner` can tokenize a string or a stream.

```
1       public static void main(String[] args) {
2           Scanner s = null;
3           StringBuilder sb = new StringBuilder(64);
4           String line01 = "1.1, 2.2, 3.3";
5           float fsum = 0.0f;
6
7           s = new Scanner(line01).useDelimiter(", ");
8           try {
9               while (s.hasNextFloat()) {
10                  float f = s.nextFloat();
11                  fsum += f;
12                  sb.append(f).append(" ");
13              }
14              System.out.println("Values found: " + sb.toString());
15              System.out.println("FSum: " + fsum);
16          } catch (Exception e) {
17              System.out.println(e.getMessage());
18          }
```

A `Scanner` can be used to tokenize an input stream or a string. In addition, a `Scanner` can be used to tokenize numbers and convert them into any primitive number type. Note how the `Scanner` is defined on line 7. The resulting object can be iterated over based on a specific type. In this case, a `float` is used.

The output from this code segment is as follows:

```
Values found: 1.1 2.2 3.3
FSum: 6.6000004
```

Regular Expressions

- A language for matching strings of text
  - Very detailed vocabulary
  - Search, extract, or search and replace
- With Java, the backslash (\) is not fun.
- Java objects
  - `Pattern`
  - `Matcher`
  - `PatternSyntaxException`
  - `java.util.regex`

`Pattern` and `Matcher`

- `Pattern`: Defines a regular expression
- `Matcher`: Specifies a string to search

```
1   import java.util.regex.Matcher;
2   import java.util.regex.Pattern;
3
4   public class PatternExample {
5       public static void main(String[] args){
6           String t = "It was the best of times";
7
8           Pattern pattern = Pattern.compile("the");
9           Matcher matcher = pattern.matcher(t);
10
11          if (matcher.find()) { System.out.println("Found match!"); }
12      }
13  }
```

The `Pattern` and `Matcher` objects work together to provide a complete solution.

The `Pattern` object defines the regular expression that will be used for the search. As shown in the example, a regular expression can be as simple as a word or phrase. The `Matcher` object is then used to select the target string to be searched. A number of methods are available for matcher. They are covered in the following slides.

When run, the example produces the following output:

```
Found match!
```

Character Classes

Character	Description
.	Matches any single character (letter, digit, or special character), except

	end-of-line markers
[abc]	Would match any “a,” “b,” or “c” in that position
[^abc]	Would match any character that is not “a,” “b,” or “c” in that position
[a-c]	A range of characters (in this case, “a,” “b,” and “c”)
|	Alternation; essentially an “or” indicator

Character classes enable you to match one character in a number of ways.

Character Class: Examples

Target String	**It was the best of times**

Pattern	**Description**	**Text Matched**
w.s	Any sequence that starts with a “w” followed by any character followed by “s”.	It was the best of times
w[abc]s	Any sequence that starts with a “w” followed by “a”, “b”, or “c” and then “s”.	It was the best of times
t[^aeo]mes	Any sequence that starts with a “t” followed any character that is not “a”, “e”, or “o” followed by “mes”.	It was the best of times

The code for this example can be found in the `StringExamples` project in the `CustomCharClassExamples.java` file.

Character Class Code: Examples

```
1   public class CustomCharClassExamples {
2       public static void main(String[] args) {
3           String t = "It was the best of times";
4
5           Pattern p1 = Pattern.compile("w.s");
6           Matcher m1 = p1.matcher(t);
7           if (m1.find()) { System.out.println("Found: " + m1.group());
8           }
9
10          Pattern p2 = Pattern.compile("w[abc]s");
11          Matcher m2 = p2.matcher(t);
12          if (m2.find()) { System.out.println("Found: " + m2.group());
13          }
14
15          Pattern p3 = Pattern.compile("t[^eou]mes");
16          Matcher m3 = p3.matcher(t);
17          if (m3.find()) { System.out.println("Found: " + m3.group());
18          }
```

The example in the slide shows two ways to find “`was`” and a way to find “`times`”.
To make this happen in Java:

1. Create a `Pattern` object to store the regular expression that you want to search with.
2. Create a `Matcher` object by passing the text to be searched to your `Pattern` object and returning a `Matcher`.
3. Call `Matcher.find()` to search the text with the `Pattern` you defined.
4. Call `Matcher.group()` to display the characters that match your pattern.

Predefined Character Classes

Predefined Character	Character Class	Negated Character	Negated Class
\d (digit)	[0-9]	\D	[^0-9]
\w (word char)	[a-zA-Z0-9_]	\W	[^a-zA-Z0-9_]
\s (white space)	[\r\t\n\f\0XB]	\S	[^ \r\t\n\f\0XB]

A number of character classes are used repeatedly. These classes are turned into predefined character classes. Classes exist to identify digits, word characters, and white space.

White-Space Characters

\t: Tab character
\n: New-line character
\r: Carriage return
\f: Form feed
\x0B: Vertical tab

Predefined Character Class: Examples

Target String	**Jo told me 20 ways to San**

	Jose in 15 minutes.

Pattern	Description	Text Matched
\\d\\d	Find any two digits.**	Jo told me 20 ways to San Jose in 15 minutes.
\\sin\\s	Find “in” surrounded by two spaces and then the next three characters.	Jo told me 20 ways to San Jose in 15 minutes.
\\Sin\\S	Find “in” surrounded by two non-space characters and then the next three characters.	Jo told me 20 ways to San Jose in 15 minutes.

** If there are additional matches in the current line, additional calls to `find()` will return the next match on that line.

Example:

```
Pattern p1 = Pattern.compile("\\d\\d");
  Matcher m1 = p1.matcher(t);
  while (m1.find()){
    System.out.println("Found:    "    +
m1.group());
  }
```

Produces:

```
Found: 20
Found: 15
```

The code for this example can be found in the `StringExamples` project in the `PredefinedCharClassExample.java` file.

Quantifiers

Quantifier	Description
*	The previous character is repeated zero or more times.
+	The previous character is repeated one or more times.
?	The previous character must appear once or not at all.
{n}	The previous character appears exactly n times.
{m,n}	The previous character appears from m to n times.
{m,}	The previous character appears m or more times.
(xx){n}	This group of characters repeats n times.

Quantifiers enable you to easily select a range of characters in your queries.

Quantifier: Examples

Target String	Longlonglong ago, in a galaxy far far away

Pattern	Description	Text Matched
ago.*	Find “ago” and then 0 or all the characters remaining on the line.	Longlonglong ago, in a galaxy far far away
gal.{3}	Match “gal” plus the next three characters. This replaces “...” as used in a previous example.	Longlonglong ago, in a galaxy far far away
(long){2}	Find “long” repeated twice.	Longlonglong ago, in a galaxy far far away

The code for this example can be found in the `StringExamples` project in the `QuantifierExample.java` file.

Greediness

- A regular expression always tries to grab as many characters as possible.
- Use the `?` operator to limit the search to the shortest possible match.

Target String	Longlonglong ago, in a galaxy far far away.

Pattern	Description	Text Matched
ago.*far	A regular expression always	Longlonglong ago, in a galaxy far far

	grabs the most characters possible.	away.
ago.*?far	The "?" character essentially turns off greediness.	Longlonglong ago, in a galaxy far far away.

A regular expression always tries to match the characters that return the most characters. This is known as the "greediness principle." Use the `?` operator to limit the result to the fewest characters needed to match the pattern. The code for this example can be found in the `StringExamples` project in the `GreedinessExample.java` file.

Quiz

Which symbol means that the character is repeated one or more times?

a. `*`
b. `+`
c. `.`
d. `?`

Answer: b

Boundary Matchers

Anchor	Description
^	Matches the beginning of a line
$	Matches the end of a line
\b	Matches the start or the end of a word
\B	Does not match the start or end of a word

Boundary characters can be used to match different parts of a line.

Boundary: Examples

Target String	it was the best of times or it was the worst of times

Pattern	Description	Text Matched
^it.*?times	The sequence that starts a line with "it" followed by some characters and "times", with greediness off.	it was the best of times or it was the worst of times
\\sit.*times$	The sequence that starts with "it" followed by some characters and ends the line with "times".	it was the best of times or it was the worst of times
\\bor\\b.{3}	Find "or" surrounded by word boundaries, plus the next three characters.	it was the best of times or it was the worst of times

The code for this example can be found in the `StringExamples` project in the `BoundaryCharExample.java` file.

Quiz

Which symbol matches the end of a line?

a. `*`
b. `+`
c. `$`
d. `^`

Answer: c

Matching and Groups

Target String	george.washington@example.com

Match 3 Parts	(george).(washington)@(example.com)

Group Numbers	(1).(2)@(3)

Pattern	(\\S+?)\\.(\\S+?)@(\\S+)

With regular expressions, you can use parentheses to identify parts of a string to match. This example matches the component parts of an email address. Notice how each pair of parentheses is numbered. In a regular expression, `group(0)` or `group()` matches all the text matched when groups are used. Here is the source code for the example:

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

```
        String email =
"george.washington@example.com";
        Pattern p1 = Pattern.compile
("(\\S+?)\\.(\\S+?)@(\\S+)");
        Matcher m1 = p1.matcher(email);
        if (m1.find()){
            System.out.println("First:
" + m1.group(1));
            System.out.println("Last: "
+ m1.group(2));
            System.out.println("Domain:
" + m1.group(3));
            System.out.println
("Everything Matched: " + m1.group(0));
        }
    }
}
```

Using the `replaceAll` Method

Using the `replaceAll` method, you can search and replace.

```
public class ReplacingExample {
    public static void main(String[] args){
        String header = "<h1>This is an H1</h1>";

        Pattern p1 = Pattern.compile("h1");
        Matcher m1 = p1.matcher(header);
        if (m1.find()){
            header = m1.replaceAll("p");
            System.out.println(header);
        }
    }
}
```

You can do a search-and-replace by using the `replaceAll` method after performing a find. The output from the program is as follows:

```
<p>This is an H1</p>
```

Summary

In this lesson, you should have learned how to:

- Read data from the command line
- Search strings
- Parse strings
- Create strings by using a `StringBuilder`
- Search strings by using regular expressions
- Parse strings by using regular expressions
- Replace strings by using regular expressions

Practice 8-1 Overview: Parsing Text with `split()`

This practice covers using the `String.split()` method to parse text.

Practice 8-2 Overview: Creating a Regular Expression Search Program

This practice covers creating a program that searches through a text file using a regular expression.

Practice 8-3 Overview: Transforming HTML by Using Regular Expressions

This practice covers transforming the HTML of a file by using several regular expressions.

Chapter 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

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.

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();
}
```

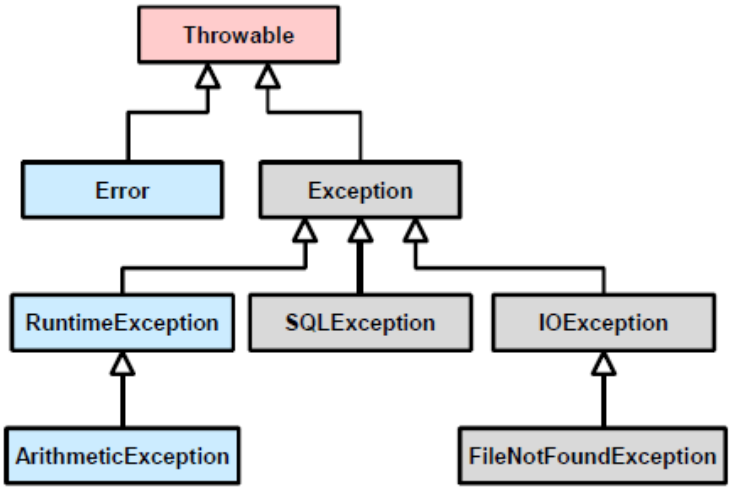
Logging Exceptions

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 `RuntimeExceptions` 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

1. A `NullPointerException` must be caught by using a `try-catch` statement.
    a. True
    b. False

    Answer: b
2. 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.

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());
    }
}
```

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/catchmultiple. 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);
    }
}
```

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:

```
1   if (x > 0) {
2     // do this
3   } else {
4     // do that
5   }
```

- The solution is:

```
1   if (x > 0) {
2     // do this
3   } else {
4     assert ( x == 0 );
5     // do that, unless x is negative
6   }
```

Control Flow Invariants

Example:

```
1  switch (suit) {
2      case Suit.CLUBS: // ...
3        break;
4      case Suit.DIAMONDS: // ...
5        break;
6      case Suit.HEARTS: // ...
7        break;
8      case Suit.SPADES: // ...
9        break;
10     default: assert false : "Unknown playing card suit";
11        break;
12  }
```

Postconditions and Class Invariants

Example:

```
1  public Object pop() {
2    int size = this.getElementCount();
3    if (size == 0) {
4      throw new RuntimeException("Attempt to pop from empty stack");
5    }
6
7    Object result = /* code to retrieve the popped element */ ;
8
9    // test the postcondition
10     assert (this.getElementCount() == size - 1);
11
12   return result;
13   }
```

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

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.

Chapter 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

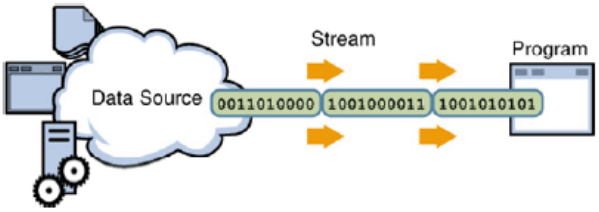
The Java programming language provides a comprehensive set of libraries to perform input/output (I/O) functions.

- 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.

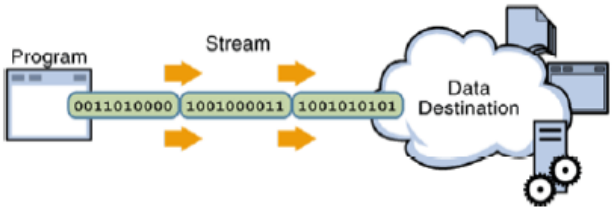
Some streams simply pass on data; others manipulate and transform the data in useful ways.

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 presentthe same simple model to programs that use them: A stream is a sequence of data.

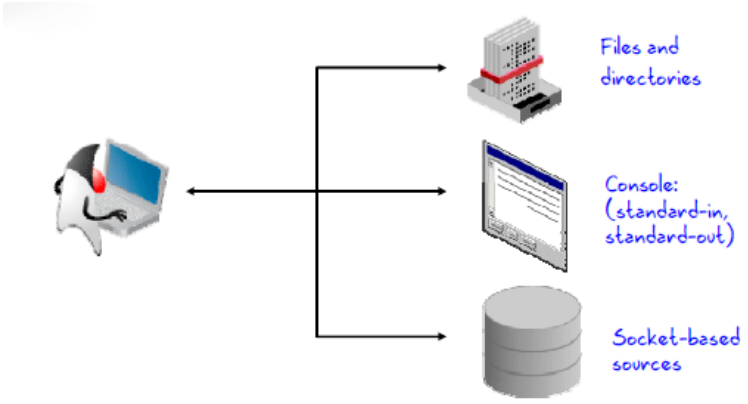
A stream is a flow of data. A stream can come from a source or can be generated to a sink.

- 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

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.io.FileNotFoundException; import java.io.IOException;
3
4 public class ByteStreamCopyTest {
5     public static void main(String[] args) {
6         byte[] b = new byte[128]; int bLen = b.length;
7         // Example use of InputStream methods
8         try (FileInputStream fis = new FileInputStream (args[0]);
9              FileOutputStream fos = new FileOutputStream (args[1])) {
10             System.out.println ("Bytes available: " + fis.available());
11            int count = 0; int read = 0;
12            while ((read = fis.read(b)) != -1) {
13                if (read < bLen) fos.write(b, 0, read);
14                else fos.write(b);
15                count += read;
16            }
17            System.out.println ("Wrote: " + count);
18        } catch (FileNotFoundException f) {
19            System.out.println ("File not found: " + f);
20        } catch (IOException e) {
21            System.out.println ("IOException: " + e);
22        }
23    }
24 }
```

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.

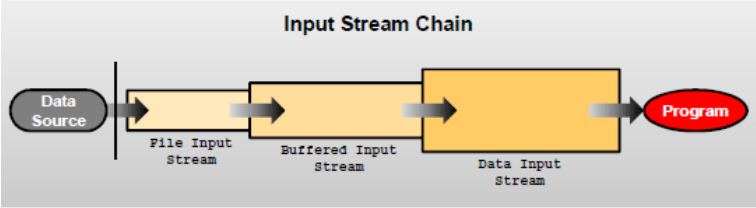
Character Stream Example

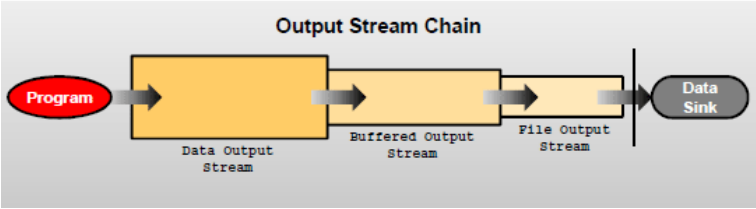
```
1 import java.io.FileReader; import java.io.FileWriter;
2 import java.io.IOException; import java.io.FileNotFoundException;
3
4 public class CharStreamCopyTest {
5     public static void main(String[] args) {
6         char[] c = new char[128]; int cLen = c.length;
7         // Example use of InputStream methods
8         try (FileReader fr = new FileReader(args[0]);
9              FileWriter fw = new FileWriter(args[1])) {
10             int count = 0;
11             int read = 0;
12             while ((read = fr.read(c)) != -1) {
13                 if (read < cLen) fw.write(c, 0, read);
14                 else fw.write(c);
15                 count += read;
16             }
17             System.out.println("Wrote: " + count + " characters.");
18         } catch (FileNotFoundException f) {
19             System.out.println("File " + args[0] + " not found.");
20         } catch (IOException e) {
21             System.out.println("IOException: " + e);
22         }
23     }
24 }
```

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

```
1 import java.io.BufferedReader; import java.io.BufferedWriter;
2 import java.io.FileReader; import java.io.FileWriter;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class BufferedStreamCopyTest {
6    public static void main(String[] args) {
7        try (BufferedReader bufInput
8                = new BufferedReader(new FileReader(args[0]));
9            BufferedWriter bufOutput
10               = new BufferedWriter(new FileWriter(args[1]))) {
11           String line = "";
12           while ((line = bufInput.readLine()) != null) {
13               bufOutput.write(line);
14               bufOutput.newLine();
15           }
16       } catch (FileNotFoundException f) {
17           System.out.println("File not found: " + f);
18       } catch (IOException e) {
19           System.out.println("Exception: " + e);
20       }
21   }
22}
```

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:

```
10 Console cons = System.console();
11 if (cons != null) {
12     String userTyped; String pwdTyped;
13     do {
14         userTyped = cons.readLine("%s", "User name: ");
15         pwdTyped = new String(cons.readPassword("%s", "Password: "));
16         if (userTyped.equals("oracle") && pwdTyped.equals("tiger")) {
17            userValid = true;
18         } else {
19           System.out.println("Wrong user name/password. Try again.\n");
20         }
21     } while (!userValid);
22 }
```

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.

```
1 import java.io.FileInputStream; import java.io.FileOutputStream;
2 import java.nio.channels.FileChannel; import java.nio.ByteBuffer;
3 import java.io.FileNotFoundException; import java.io.IOException;
4
5 public class ByteChannelCopyTest {
6     public static void main(String[] args) {
7         try (FileChannel fcIn = new FileInputStream(args[0]).getChannel();
8              FileChannel fcOut = new FileOutputStream(args[1]).getChannel()) {
9             ByteBuffer buff = ByteBuffer.allocate((int) fcIn.size());
10             fcIn.read(buff);
11            buff.position(0);
12            fcOut.write(buff);
13        } catch (FileNotFoundException f) {
14            System.out.println("File not found: " + f);
15        } catch (IOException e) {
16            System.out.println("IOException: " + e);
17        }
18    }
19 }
```

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

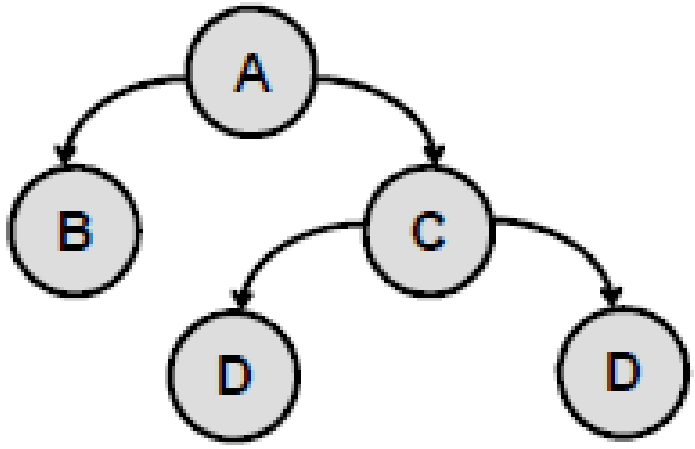
Saving data to some type of permanent storage is called persistence. An object that is persistent-capable, can be stored on disk (or any other storage device), or sent to another machine to be stored there.

- 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.

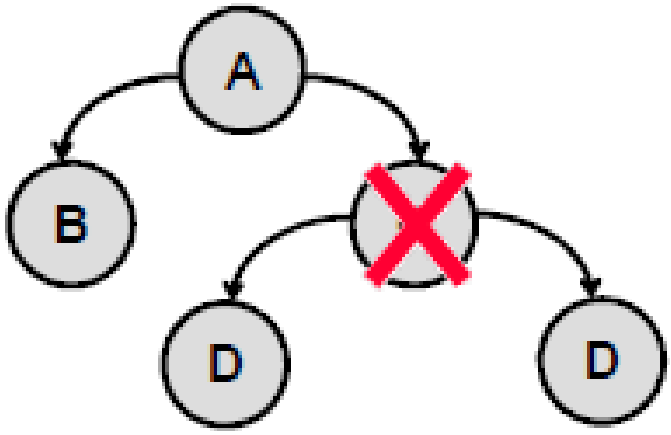


Object Graphs

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, seriaiVersionUID, is used to associate the serialized output with the class used in the serialization process.
- Upon deserialization, the seriaiVersionUID 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 seriaiVersionUID, deserialization will result in an `InvalidClassException`.
- A serializable class can declare its own seriaiVersionUID by explicitly declaring a field named `serialversionUID` as a static final and of type long:

```
private static long
serialVersionUID = 42Li
```

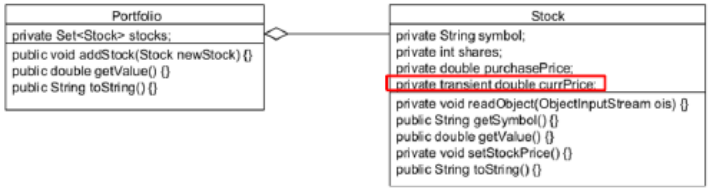
Note: The documentation for `java.io.Serializable` states the following:

If a serializable class does not explicitly declare a seriaiVersionUIO, then the serialization run time will calculate a default seriaiVersionUIO 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 seriaiVersionUIO values, since the default seriaiVersionUIO computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected lnvalidCiassExceptions during deserialization. Therefore, to guarantee a consistent seriaiVersionUIO value across different java compiler implementations, a serializable class must declare an explicit seriaiVersionUIO value. It is also strongly advised that explicit seriaiVersionUIO declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--seriaiVersionUIO fields are not useful as inherited members. Array classes cannot declare an explicit seriaiVersionUIO, so they always have the default computed value, but the requirement for matching seriaiVersionUIO 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.



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

readObject Example

```
1 public class Stock implements Serializable {
2     private static final long serialVersionUID = 100L;
3     private String symbol;
4     private int shares;
5     private double purchasePrice;
6     private transient double currPrice;
7
8     public Stock(String symbol, int shares, double purchasePrice) {
9         this.symbol = symbol;
10        this.shares = shares;
11        this.purchasePrice = purchasePrice;
12        setStockPrice();
13    }
14
15    // This method is called post-serialization
16    private void readObject(ObjectInputStream ois)
17                 throws IOException, ClassNotFoundException {
18        ois.defaultReadObject();
19        // perform other initialization
20        setStockPrice();
21    }
22 }
```

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

1. 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).

2. 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.
    c. `final` has no effect on the serialization of the field and also changes the meaning of the field, making it immutable.

3. 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`.

4. 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)

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.

Chapter 11

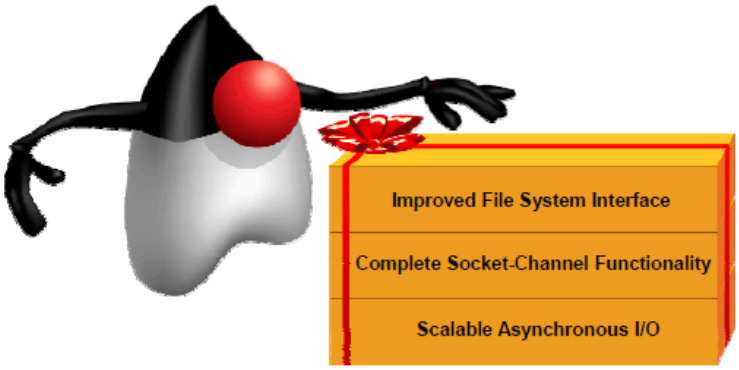
Java File I/O (NIO.2)

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 (NI0.2)



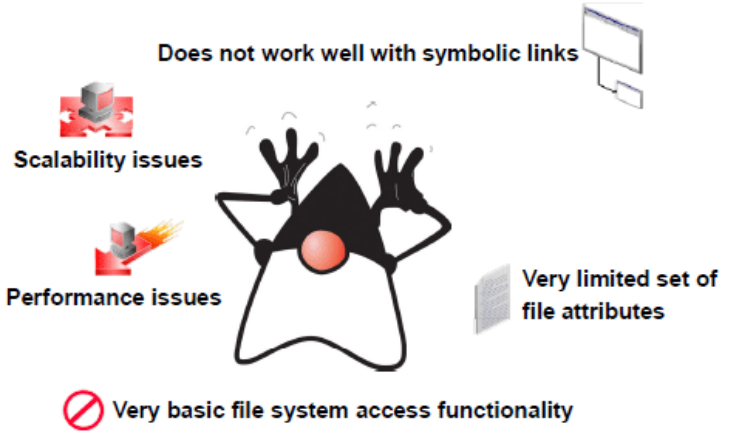
NIO API in JSR 51 established the basis for NIO in Java, focusing on buffers, channels, and charsets. JSR 51 delivered the first piece of the scalable socket I/Os into the platform, providing a non-blocking, multiplexed I/O API, thus allowing the development of highly scalable servers vvithout having 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`



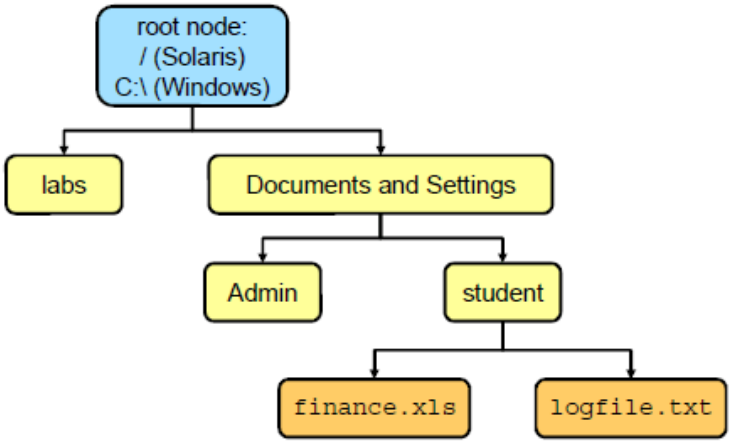
The Java I/O File API (`java. io. File`) presented challenges for developers.

- 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 NI0.2, 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 NI0.2 implementation in JDK 7, files were represented by the `java.io.File` class.

In NI0.2, 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
...
```

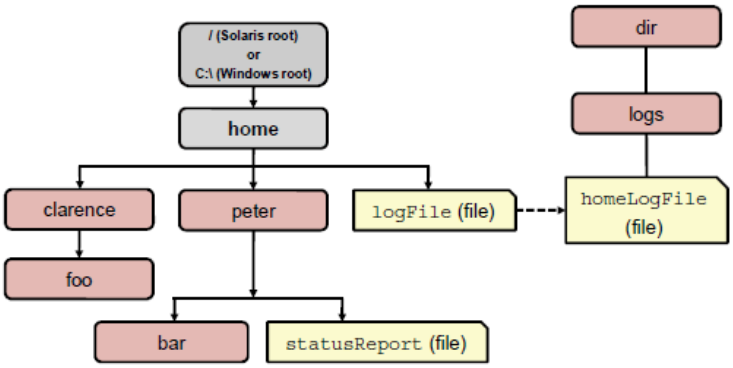
- A relative path must be combined with another path in order to access a file.
- Example:

```
...
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 NI0.2 Concepts

Prior to JDK 7, the `java.io.File` class was the entry point for all file and directory operations. With NI0.2, 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 NI0.2

A significant difference between NI0.2 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 NI0.2, 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 NI0.2 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 NI0.2 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

```
1 public class PathTest
2     public static void main(String[] args) {
3         Path p1 = Paths.get(args[0]);
4         System.out.format("getFileName: %s%n", p1.getFileName());
5         System.out.format("getParent: %s%n", p1.getParent());
6         System.out.format("getNameCount: %d%n", p1.getNameCount());
7         System.out.format("getRoot: %s%n", p1.getRoot());
8         System.out.format("isAbsolute: %b%n", p1.isAbsolute());
9         System.out.format("toAbsolutePath: %s%n", p1.toAbsolutePath());
10        System.out.format("toURI: %s%n", p1.toUri());
11    }
12 }
```

```
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-dependentway.

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 "`directoryI ..`" 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`.

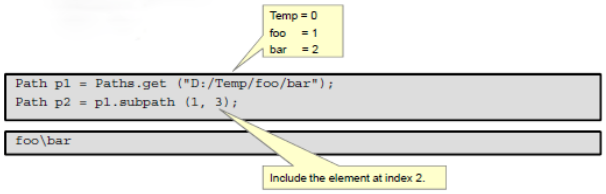
It is important to note that `normalize` does not check the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if `peter` were a symbolic link, removing `peter/..` might result in a path that no longer locates the intended file.

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:



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

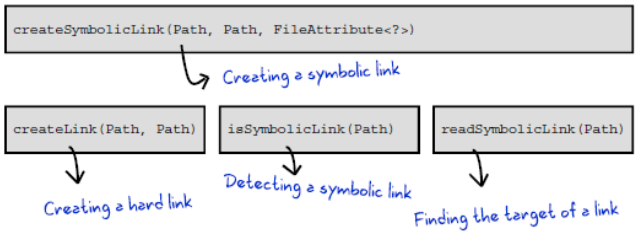
- The `relativize` method enables you to construct a path from one location in the file system to another location.
- The method constructs a path originating from the original path and ending at the location specified by the passed-in path.
- The new path is relative to the original path.
- Example:

```
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

1. 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
2. Given the following path:
   ```
   Path p = Paths.get
   ("/home/export/tom/documents/coursefiles/JDK711");
   ```
   and the statement:
   ```
   Path sub = p.subPath (X,y);
   ```
   What values for x andy 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
3. Given this code fragment:
   ```
   Path pl = Paths.get("D:/temp/foo/");
   Path p2 = Paths.get("../bar/documents");
   Path p3 = pl.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.

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

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)
```

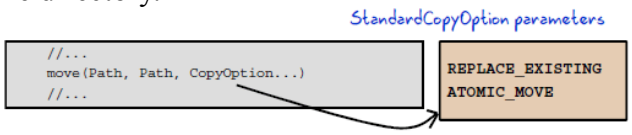
- An interesting use of the first method is copying from a web page and saving to a file:

```
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.



- 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.

```
1 Path dir = Paths.get("D:/Temp");
2 // DirectoryStream is a stream, so use try-with-resources
3 // or explicitly close it when finished
4 try (DirectoryStream<Path> stream =
5                     Files.newDirectoryStream(dir, "*.zip")) {
6     for (Path file : stream) {
7         System.out.println(file.getFileName());
8     }
9 } catch (PatternSyntaxException | DirectoryIteratorException |
10         IOException x) {
11     System.err.println(x);
12 }
```

- `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 ifthere is an I/O errorwhile 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 `read.AllLines (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.

NI0.2 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:

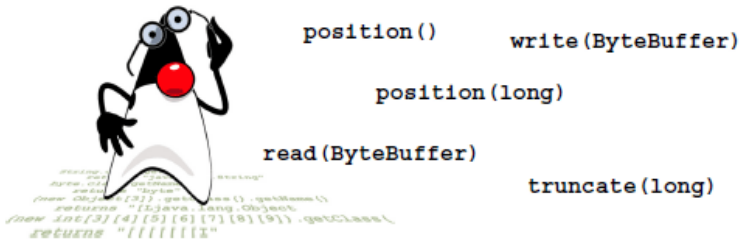
- `newByteChannel(Path, OpenOption... )`
- `newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?> ... )`

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

- NI0.2 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 `read.Attributes` methods to fetch a file's attributes in one bulk operation.

- `read.Attributes(Path, String, LinkOption...)`
- `read.Attributes(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 `read.Attributes` method, the attributes of the file (or directory) are returned.

POSIX Permissions

With NI0.2, you can create files and directories on POSIX file systems with their initial permissions set.

```
1  Path p = Paths.get(args[0]);
2  Set<PosixFilePermission> perms =
3      PosixFilePermissions.fromString("rwxr-x---");
4  FileAttribute<Set<PosixFilePermission>> attrs =
5      PosixFilePermissions.asFileAttribute(perms);
6  try {
7      Files.createFile(p, attrs);
8  } catch (FileAlreadyExistsException f) {
9      System.out.println("FileAlreadyExists" + f);
10 } catch (IOException i) {
11     System.out.println("IOException:" + i);
12 }
```

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

1. 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`
   c. Directory `peter2` is created.
   d. Directory `peter` is copied to `peter2`.
   e. Directory `peter2` is created and populated with files and directories from `peter`.

Answer: e

2. Given the 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

3. 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:

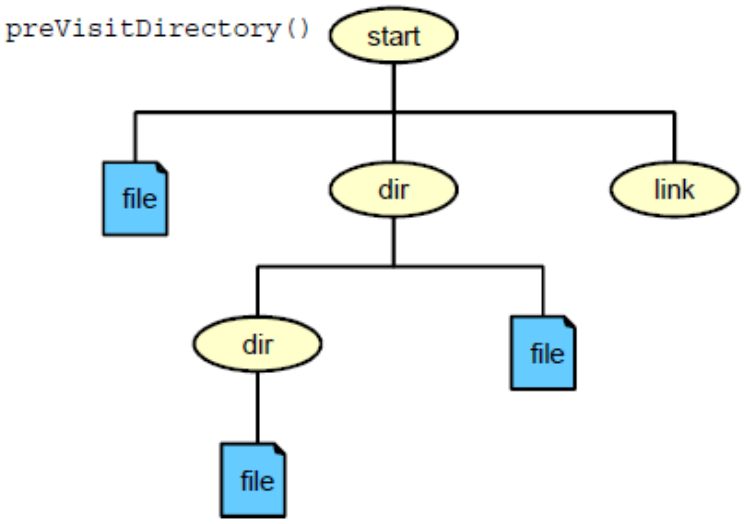
- `preVisitDirectory`: Invoked on a directory before the entries in the directory are visited
- `visitFile`: Invoked for a file in a directory
- `postVisitDirectory`: Invoked after all the entries in a directory and their descendants have been visited
- `visitFileFailed`: Invoked for a file that could 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
- `TERIVliNATE`

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

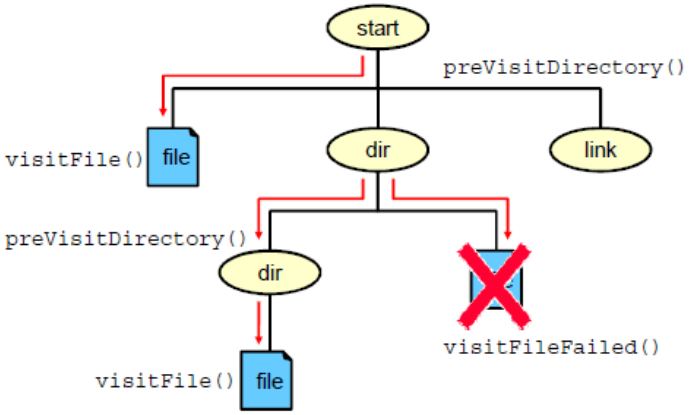


FileVisitor

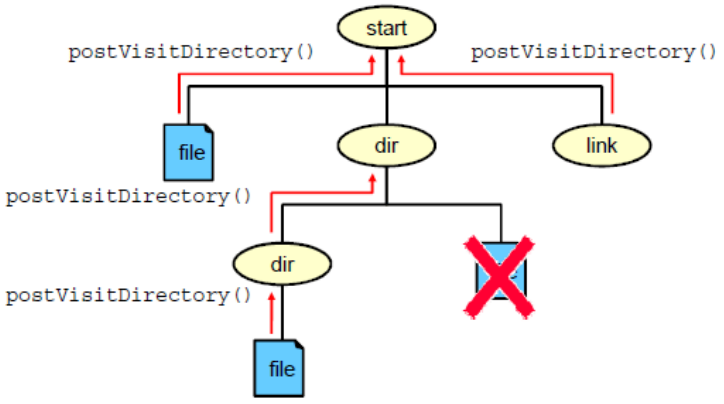
Starting at the first directory node, and at every subdirectory encountered, the `preVisitDirectory (Path/BasicFileAttributes)` method is invoked on the class passed to the `walkFileTree` method.

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.



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.

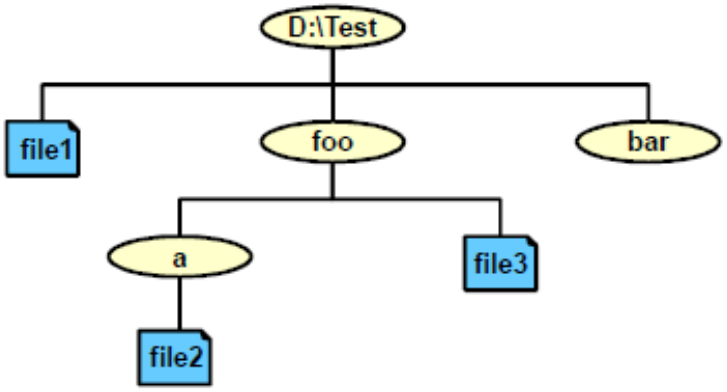


After reaching all the children in a node, the `postVisitDirectory` method is invoked on each of the directories.

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

```
1 public static void main(String[] args) {
2     // ... check for two arguments
3     Path root = Paths.get(args[0]);
4     // ... check that the first argument is a directory
5     PathMatcher matcher =
6         FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
7     // Finder is class that implements FileVisitor
8     Finder finder = new Finder(root, matcher);
9     try {
10         Files.walkFileTree(root, finder);
11     } catch (IOException e) {
12         System.out.println("Exception: " + e);
13     }
14     finder.done();
15 }
```

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

```
1 public class Finder extends SimpleFileVisitor<Path> {
2     private Path file;
3     private PathMatcher matcher;
4     private int numMatches;
5     // ... constructor stores Path and PathMatcher objects
6     private void find(Path file) {
7         Path name = file.getFileName();
8         if (name != null && matcher.matches(name)) {
9             numMatches++;
10             System.out.println(file);
11         }
12     }
13     @Override
14     public FileVisitResult visitFile(Path file,
15                                      BasicFileAttributes attrs) {
16         find(file);
17         return CONTINUE;
18     }
19     //...
20 }
```

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 NI0.2

A method was added to the `java.io.File` class for JDK 7 to provide forward compatibility with NI0.2.

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

- This enables you to take advantage of NI0.2 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 NI0.2 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

1. To copy, move, or open a file or directory using NI0.2, you must first create an instance of:
    a. `Path`
    b. `Files`

c. `FileSystem`
d. `Channel`

Answer: a

2. 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.
3. 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.

Chapter 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

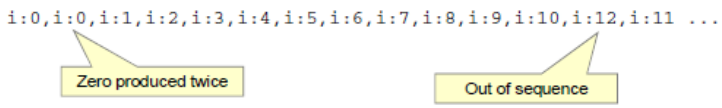
Problems with Shared Data

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:



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

```
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:

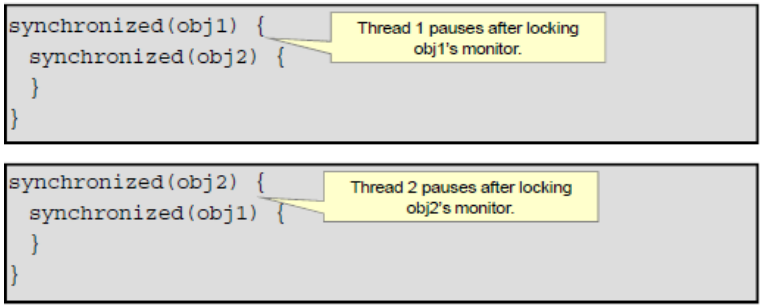
- `destroy()`
- `resume()`
- `suspend()`
- `stop()`

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.



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.

Chapter 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.

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.

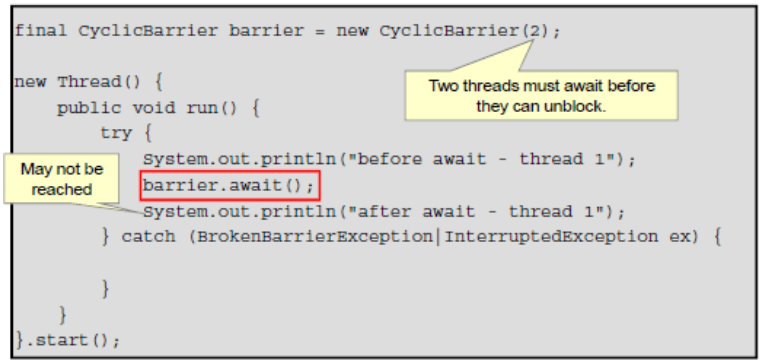
`CyclicBarrier`: Created with a party count. After the number of parties (threads) have called `await` on the `CyclicBarrier` they will be released (unblock). A `CyclicBarrier` can be reused.

`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`.



`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

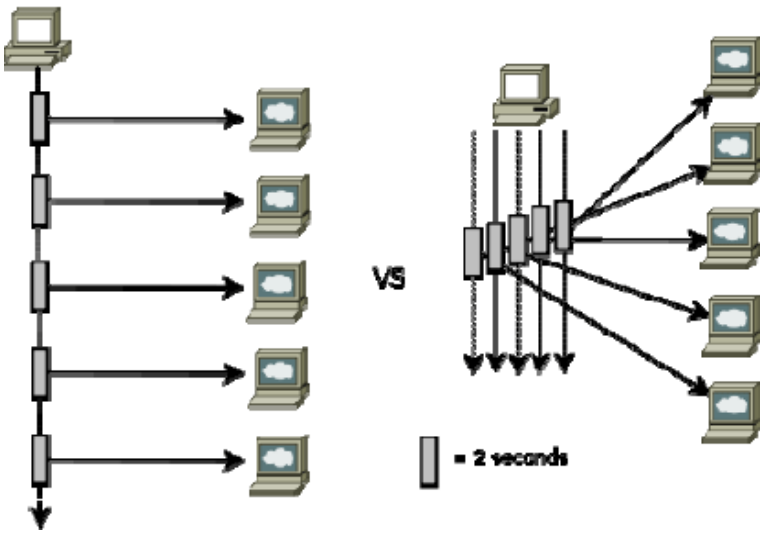
An `ExecutorService` will always attempt to use all of the available CPUs in a system.

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)

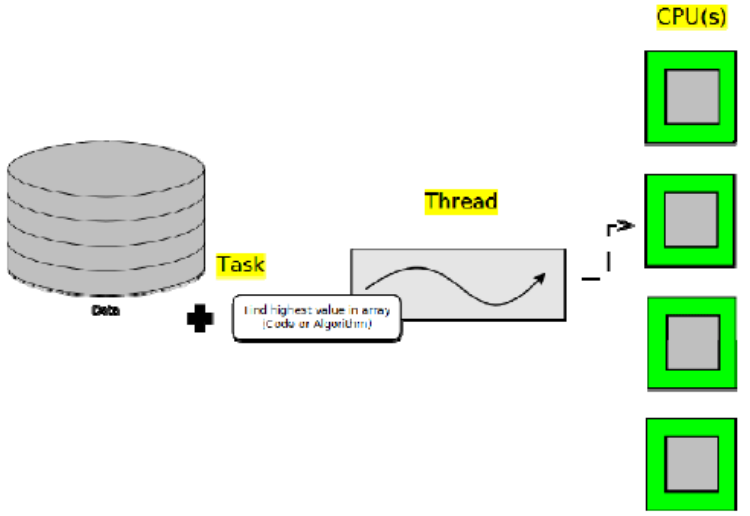
```
        for(RequestResponse lookup : callables.keySet()) {
            Future<RequestResponse> future = callables.get(lookup);
            try {
                lookup = future.get();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } 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

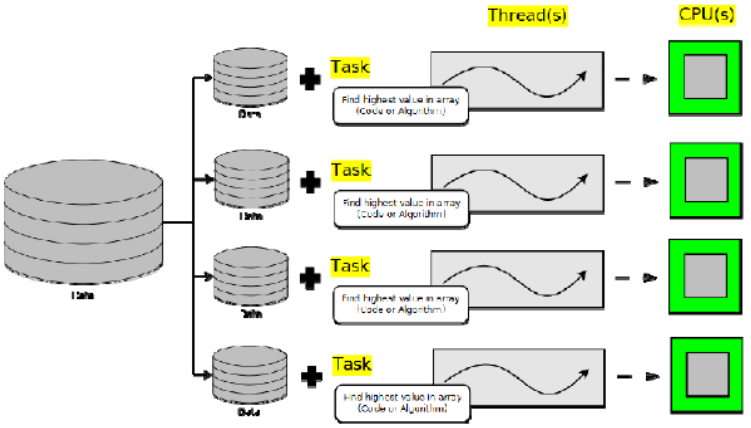
Modern systems contain multiple CPUs. If you do not leverage threads in some way, only a portion of your system's processing power will be utilized.

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.



Splitting the Data

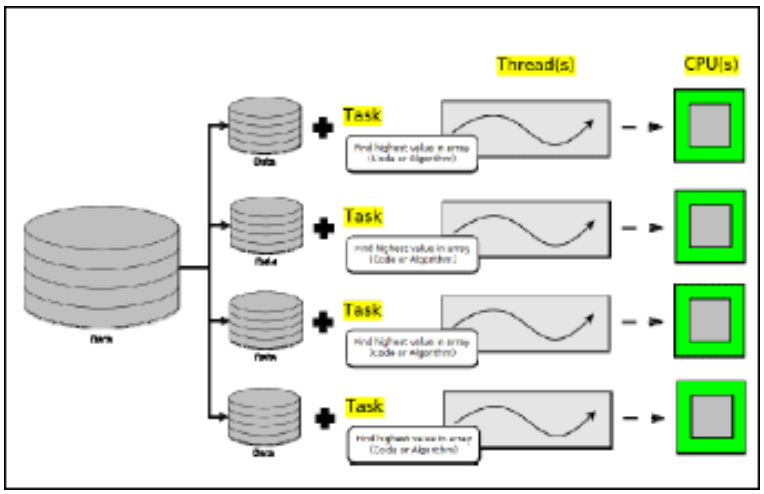
In the slide's graphic, a large set of data (an array, possibly) is split into four subsets of data, one subset for each CPU. A thread per CPU is created to process the data. After processing, the subsets of data the results will have to combined in a meaningful way. There are several ways to subdivide 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 nonblocking 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

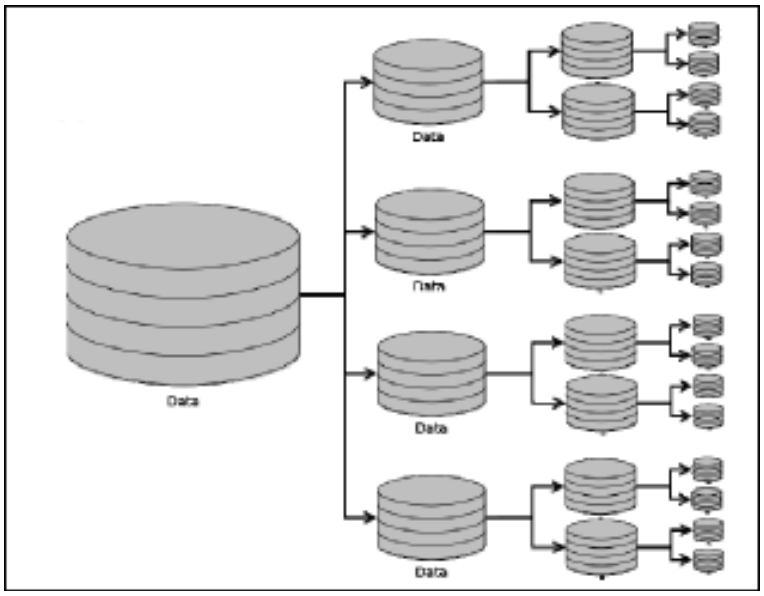


Work-Stealing

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

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.

`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.

Chapter 14

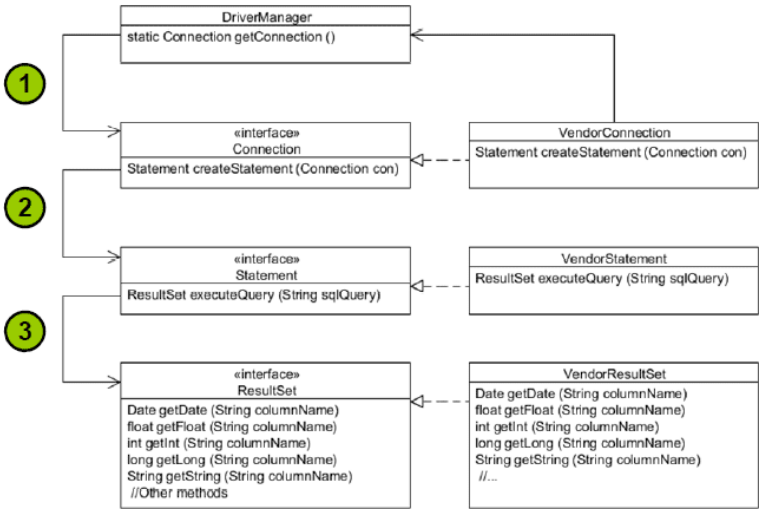
Building Database Applications with JDBC

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



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 `METAINF/ 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.

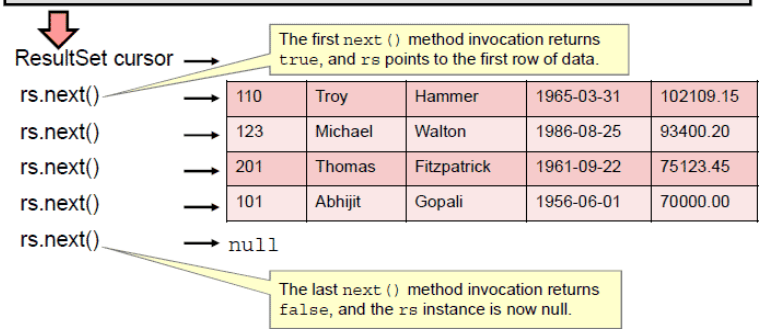
`Statement` is an interface that provides a class for executing SQL statements and returning the results. The `Statement` interface is for static SQL queries. There are two other subinterfaces: `PreparedStatement`, which extends `Statement` and `CallableStatement`, which extends `PreparedStatement`.

`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);
```



`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_INSENSITIV
E, 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.

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
```

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 `SQLExceptions` thrown:

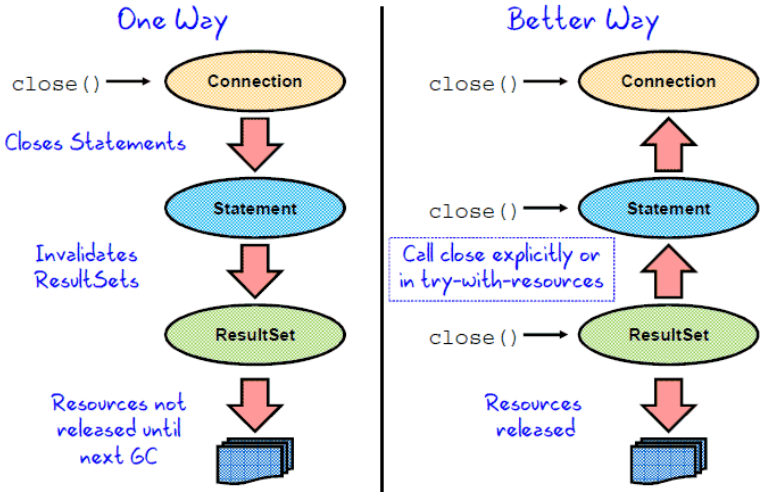
```
1 catch(SQLException ex) {
2     while(ex != null) {
3         System.out.println("SQLState:   " + ex.getSQLState());
4         System.out.println("Error Code:" + ex.getErrorCode());
5         System.out.println("Message:    " + ex.getMessage());
6         Throwable t = ex.getCause();
7         while(t != null) {
8             System.out.println("Cause:" + t);
9             t = t.getCause();
10         }
11         ex = ex.getNextException();
12     }
13 }
```

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.

```
1 int numCols = rs.getMetaData().getColumnCount();
2 String [] colNames = new String[numCols];
3 String [] colTypes = new String[numCols];
4 for (int i= 0; i < numCols; i++) {
5     colNames[i] = rs.getMetaData().getColumnName(i+1);
6     colTypes[i] = rs.getMetaData().getColumnTypeName(i+1);
7 }
8 System.out.println ("Number of columns returned: " + numCols);
9 System.out.println ("Column names/types returned: ");
10 for (int i = 0; i < numCols; i++) {
11     System.out.println (colNames[i] + " : " + colTypes[i]);
12 }
```

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?"

```
1 public int rowCount(ResultSet rs) throws SQLException{
2     int rowCount = 0;
3     int currRow = rs.getRow();
4     // Valid ResultSet?
5     if (!rs.last()) return -1;
6     rowCount = rs.getRow();
7     // Return the cursor to the current position
8     if (currRow == 0) rs.beforeFirst();
9     else rs.absolute(currRow);
10    return rowCount;
11 }
```

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`

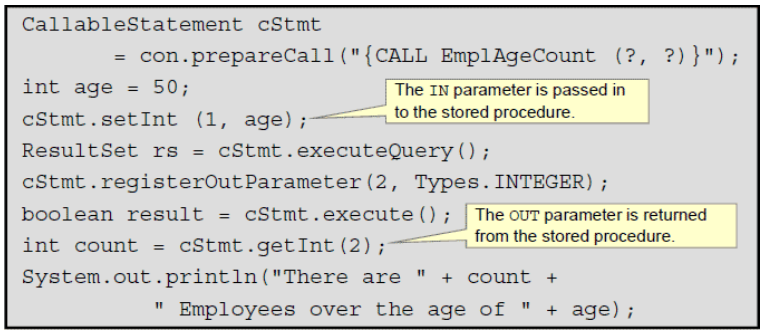
The SQL statement in the example in the slide is precompiled and stored in the `PreparedStatement` object. This statement can be used efficiently to execute this statement multiple times. This example could be in a loop, looking at different values.

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.



- 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.

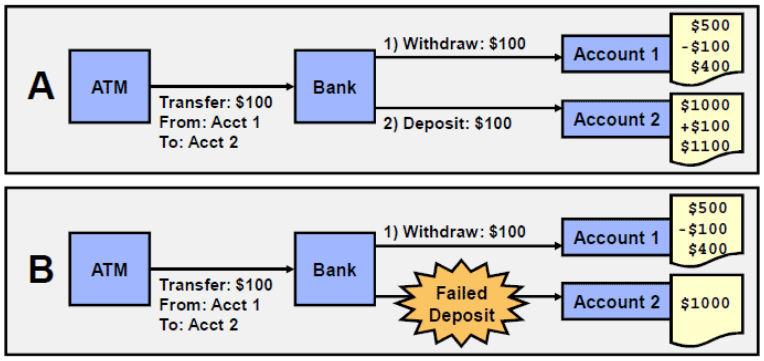
- **Atomicity**: A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **Consistency**: A transaction transforms a system from one consistent state to another consistent state.
- **Isolation**: 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.

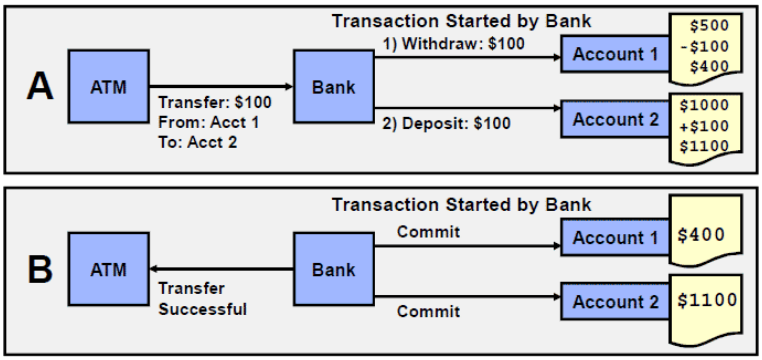
C.The client application requires that the object maintains an in-memory context between each invocation; each successive invocation uses the data that is maintained in memory.

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.

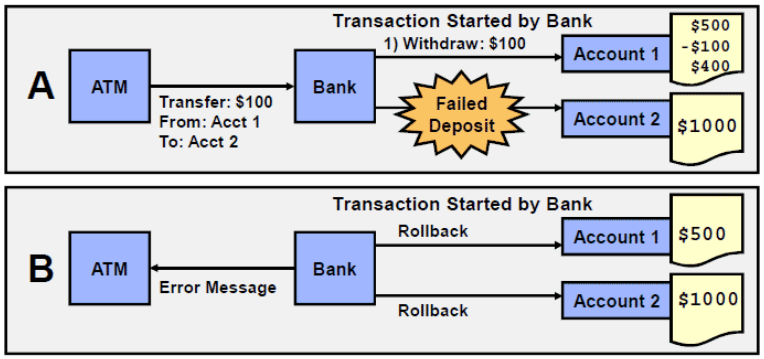
Within the scope of one client invocation on an object, the object performs multiple changes to the data in a database. If one change fails, the object must roll back all the changes. Consider a banking application. The client invokes the transfer operation on a teller object. The operation requires 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.

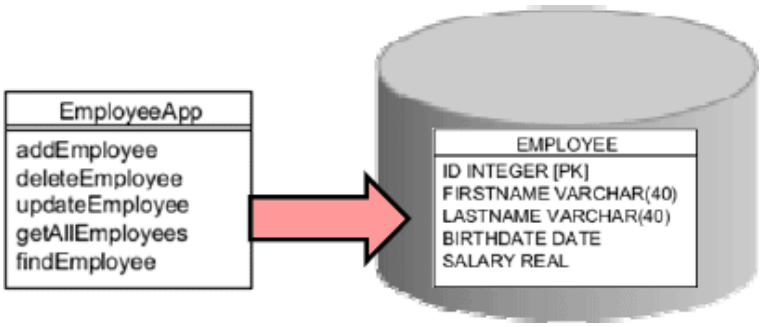
Example: Using JdbcRowSet

```
10 try (JdbcRowSet jdbcRs =
11       RowSetProvider.newFactory().createJdbcRowSet()) {
12     jdbcRs.setUrl(url);
13     jdbcRs.setUsername(username);
14     jdbcRs.setPassword(password);
15     jdbcRs.setCommand("SELECT * FROM Employee");
16     jdbcRs.execute();
17     // Now just treat JDBC Row Set like a ResultSet object
18     while (jdbcRs.next()) {
19         int empID = jdbcRs.getInt("ID");
20         String first = jdbcRs.getString("FirstName");
21         String last = jdbcRs.getString("LastName");
22         Date birthDate = jdbcRs.getDate("BirthDate");
23          float salary = jdbcRs.getFloat("Salary");
24     }
25   //... other methods
26 }
```

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.

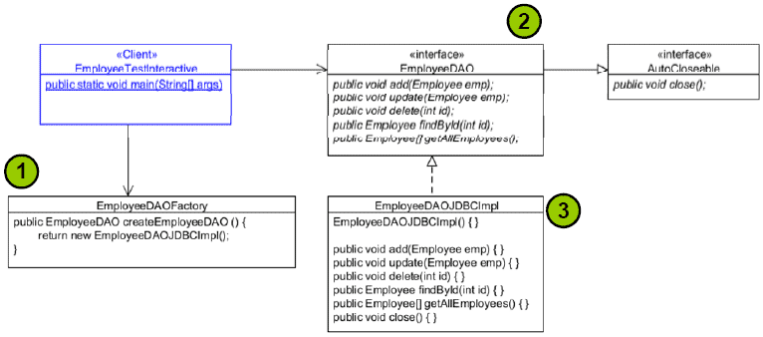
- ❑ Any changes to the table (such as adding a field) will require a complete change to the application.
- ❑ Employee data is not encapsulated within the example application.

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



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

1. 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);`

   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.
2. 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.
3. 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.

4. 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`

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.

Chapter 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 ??б. 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.

Appendix A

SQL Primer

Objectives

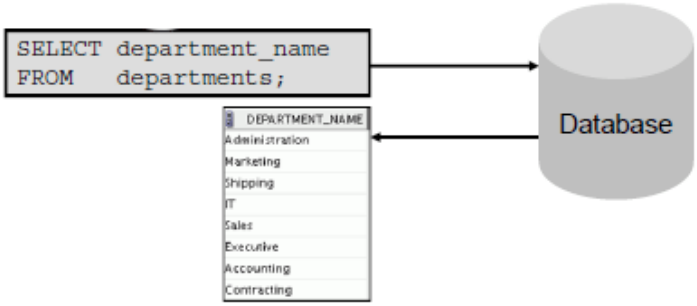
After completing this lesson, you should be able to:

- 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.)



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.

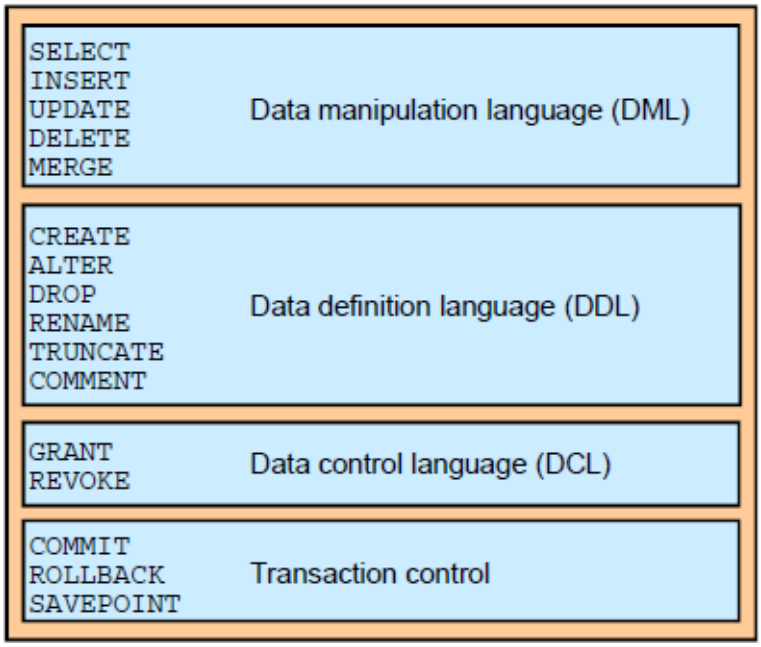
To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. SQL is a set of statements with which all programs and users access data in an Oracle Database. Application 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



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	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language (DDL)*

COMMENT	
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 WHEREclause:

```
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

- 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)

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

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

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)
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.
```

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