Java Introduction to Server-Side Programming

Version 2.2.0 March 2021



Copyright © Descriptor Systems, 2012
All Rights Reserved.

This document may not be copied or reproduced in any form without prior written consent of Joel Barnum of Descriptor Systems.

All trademarks belong to their respective companies.

You can contact Descriptor Systems at:

http://www.descriptor.com
info@descriptor.com
319-362-3906
P.O. Box 461
Marion, IA 52302

Java Introduction to Server-Side Programming

Chapter 1: Introduction to the Course

Java Introduction to Server-Side Programming....1-1
Legal Information....1-2
Java Introduction to Server-Side Programming....1-3
Introductions....1-4
Course Description....1-5
Course Objectives....1-6
Sample Agenda....1-7
Sample Agenda, cont'd....1-8
Sample Agenda, cont'd....1-9

Chapter 2: Introduction to Java

Introduction to Java....2-1
What is Java?....2-2
Java History....2-3
Java Language Family Tree....2-4
Java Programming Platforms....2-5
Java Standard Edition Version History....2-6
Java Standard Edition Version History, cont'd....2-7
Portability....2-8
Java Development Process....2-9
The Java SDK versus the JRE....2-10
The Java Class Library....2-11
Introduction to JEE....2-12
Where Does Java Run?....2-13
Hello, World....2-14
Chapter Summary....2-15

Chapter 3: Data Types and Assignment

Data Types and Assignment....3-1
Java Program Structure....3-2

Java Comments....3-3
Java Statements....3-4
Java is Strongly Typed....3-5
Variables....3-6
Rules for Identifiers....3-7
Reserved Words....3-8
Java Data Types....3-9
Integer Types....3-10
Floating Point Types....3-11
Character Type....3-12
Character Escape Codes....3-13
Boolean Type....3-14
The String Type....3-15
Assignment....3-16
Cast Operator....3-17
Scope of Variables....3-18
Scope of Variables, cont'd....3-19
The Stack and the Heap....3-20
Quick Practice....3-21
Chapter Summary....3-22

Chapter 4: Operators

Operators....4-1
Operators....4-2
Math Operators....4-3
Math Operators, cont'd....4-4
Other Math Operations....4-5
Quick Practice....4-6
Compound Assignment Operators....4-7
Increment and Decrement Operators....4-8
Equality Operators....4-9
Relational Operators....4-10
Integer Bitwise Operators....4-11
Boolean Logical Operators....4-12

Conditional Operators 4-13
Shift Operators 4-14
Conditional Operator 4-15
The Boolean ! Operator 4-16
String Concatenation 4-17
Other Operators 4-18
Operator Precedence 4-19
Chapter Summary 4-20

Chapter 5: Flow Control

Flow Control 5-1
Defining Blocks 5-2
Conditional Statements 5-3
The if Statement 5-4
The if else Statement 5-5
The if else if Statement 5-6
The if else if Statement, cont'd 5-7
Quick Practice 5-8
The while Loop 5-9
The while Loop, cont'd 5-10
The for Loop 5-11
The for Loop, cont'd 5-12
The for Loop, cont'd 5-13
Breaking Out of a Loop 5-14
The switch-case Statement 5-15
The switch-case Statement, cont'd 5-16
The switch-case Statement, cont'd 5-17
Strings in Switch Statement 5-18
Quick Practice 5-19
Chapter Summary 5-20

Chapter 6: Introduction to Object Orientation

Introduction to Object Orientation 6-1
What is Object-Oriented Programming? 6-2
Object-Oriented Languages 6-3

Goals of Object Orientation 6-4
The Three Pillars 6-5
What is an Object? 6-6
Why are Objects Useful? 6-7
Objects Respond to Messages 6-8
Objects and Classes 6-9
A Class is Like a Cookie Cutter 6-10
Writing a Class 6-11
Sample Java Class 6-12
Instantiating Objects 6-13
Methods Operate on Objects 6-14
Methods Must Have a Receiver 6-15
Implicit Receivers 6-16
The Stack and the Heap 6-17
Comparing References 6-18
Comparing References, cont'd 6-19
Garbage Collection 6-20
Introduction to Inheritance 6-21
Introduction to Polymorphism 6-22
Synonym Alert! 6-23
Introduction to Packages 6-24
Encapsulation 6-25
Using Encapsulation 6-26
Java Encapsulation Summary 6-27
Get/Set Methods 6-28
Get/Set Methods, cont'd 6-29
Introduction to Interfaces 6-30
Quick Practice 6-31
Chapter Summary 6-32

Chapter 7: Methods

Methods 7-1
What is a Method? 7-2
Calling a Method 7-3

Method Receivers....7-4
How 'this' Works....7-5
Class Structure....7-6
Method Syntax....7-7
Instance Method Examples....7-8
Calling Instance Methods....7-9
Calling Methods, cont'd....7-10
Quick Practice....7-11
Quick Practice 2....7-12
Overloading Methods....7-13
Calling Overloading Methods....7-14
Constructor Methods....7-15
The new Operator....7-16
Overloaded Constructors....7-17
Calling Constructors....7-18
Quick Practice....7-19
The No-Argument Constructor....7-20
Calling One Constructor From Another....7-21
Field Initializers....7-22
Method Modifiers....7-23
Static Methods....7-24
Static Methods, cont'd....7-25
Implicit Receiver for Static Method....7-26
Static Fields....7-27
Static Fields, cont'd....7-28
Quick Practice....7-29
The final Keyword....7-30
Chapter Summary....7-31

Chapter 8: Exception Handling

Exception Handling....8-1
Traditional Error Handling....8-2
Traditional Error Handling Issues....8-3
Java Exception Handling....8-4

Advantages of Exception Handling....8-5
Uncaught Exceptions....8-6
Exception Categories....8-7
Exceptions Are Objects....8-8
Multiple Catch Blocks....8-9
Handle the Most Specific Exception First....8-10
Passing Exceptions Back....8-11
The finally Block....8-12
Throwing Exceptions....8-13
Writing an Exception Class....8-14
Writing an Exception Class, cont'd....8-15
Enhanced Catch Block Syntax....8-16
Try With Resources....8-17
Quick Practice....8-18
Chapter Summary....8-19

Chapter 9: Arrays and Collections

Arrays and Collections....9-1
What is an Array?....9-2
Using Arrays....9-3
Array Bounds Checking....9-4
Arrays of Object References....9-5
Initializing Arrays....9-6
Quick Practice....9-7
The Enhanced for Loop....9-8
Introduction to Collections....9-9
History of Java Collections....9-10
Collections Can Automatically Grow....9-11
Basic Collection Types....9-12
Concrete Collection Classes....9-13
Data Structures....9-14
Using an ArrayList....9-15
Using an ArrayList, Pre Java 5....9-16
Using a LinkedList....9-17

List Performance 9-18
Perfomance Quiz 9-19
Using a HashSet 9-20
Using a HashMap 9-21
Quick Practice 9-22
Chapter Summary 9-23

Chapter 10: JDBC Programming

JDBC Programming 10-1
Database Fundamentals 10-2
Introduction to ODBC 10-3
Introduction to JDBC 10-4
What Can You Do with JDBC? 10-5
JDBC History 10-6
Connection Pools 10-7
JDBC URLs 10-8
SQL Exceptions 10-9
The JDBC API for Data Sources 10-10
Opening a Connection, JDBC 1.x 10-11
Configuring a Datasource, JDBC 2.x 10-12
Locating a Datasource, JDBC 2.x 10-13
Opening a Connection, JDBC 2.x 10-14
The JDBC API for Connections 10-15
The JDBC API for Statements 10-16
Executing a Simple Query 10-17
Processing a ResultSet 10-18
SQL Types and Java Types 10-19
Processing a ResultSet, cont'd 10-20
Closing a Connection 10-21
Inserting a Row 10-22
Updating a Row 10-23
Removing a Row 10-24
Using Prepared Statements 10-25
Using Prepared Statements, cont'd 10-26

Chapter Summary....10-27

Chapter 11: Introduction to Servlets

Introduction to Servlets....11-1
What is a Servlet?....11-2
Why Use Servlets?....11-3
Servlet Containers....11-4
The Servlet API....11-5
Writing a Servlet....11-6
A Sample Servlet....11-7
Servlet Lifecycle....11-8
Initialization....11-9
Termination....11-10
Processing Requests....11-11
The HTTPServletRequest Interface....11-12
Get versus Post....11-13
Accessing Client Arguments....11-14
Accessing Client Arguments, cont'd....11-15
The HTTPServletResponse Class....11-16
Setting the Content Type....11-17
Obtaining an Output Stream....11-18
Generating Output....11-19
Using HTML Forms....11-20
Servlet Example: The HTML Form....11-21
Servlet Example: The Desired Output....11-22
Servlet Example: The Servlet....11-23
Servlet Example: The Servlet, cont'd....11-24
Chapter Summary....11-25

Chapter 12: Introduction to JavaServer Pages

Introduction to JavaServer Pages....12-1
What are JavaServer Pages?....12-2
JSPs Are Actually Servlets....12-3
Why Use JSPs?....12-4
Deploying JSPs....12-5

JSP Version History 12-6
Intro to the JSP Expression Language 12-7
Model 1 vs Model 2 12-8
Separating Content From Presentation 12-9
What is a JavaBean? 12-10
What is a JavaBean Property? 12-11
Using a JavaBean from a JSP 12-12
JavaBean Scopes 12-13
Using Expressions to Read Properties 12-14
Intro to the JSP Standard Tag Library 12-15
JSTL Core Tags 12-16
c:forEach Example 12-17
Chapter Summary 12-18

Chapter 13: The Model-View-Controller Architecture

The Model-View-Controller Architecture 13-1
Servlets versus JSPs 13-2
Combining Servlets and JSPs 13-3
What is the MVC Architecture? 13-4
MVC and Web Applications 13-5
MVC Tradeoffs 13-6
Transfering Control 13-7
Forwarding Requests 13-8
Forwarding Requests, cont'd 13-9
Redirecting Requests 13-10
Redirecting Requests, cont'd 13-11
MVC Example: Student Information System 13-12
MVC Example User Interface 13-13
MVC Example User Interface, cont'd 13-14
The Find Student HTML 13-15
The List All Students HTML 13-16
The Controller Servlet 13-17
The Controller Servlet, cont'd 13-18
The Display Student JSP 13-19

The Display Student JSP, Alternate..... 13-20
The List All Students JSP.....13-21
Chapter Summary.....13-22

Java Introduction to Server-Side Programming

Version 2.2.0 March 2021



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Legal Information

Copyright (C) Descriptor Systems 2001, 2021

All Rights Reserved

All trademarks are owned by their respective companies

You can contact Descriptor Systems at:
http://www.descriptor.com
jbarnum@descriptor.com

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Introduction to Server-Side Programming

- Introductions
- Objectives

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introductions

- Please introduce yourself:
  - Name:
  - Company:
  - Java background (if any):
  - HTML and Web programming background
  - Other programming languages you know:
  - Your goal for the course:

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Course Description

Description: This course introduces the student to writing object-oriented programs in Java that run on a Web server

Prerequisites: Prior study in object-orientation and UML is helpful.

Audience: Business analysts, developers, managers and other people interested in learning how to write Java server-side programs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Course Objectives

After taking this course, you will be able to:

- Write Java classes using object-oriented techniques such as encapsulation, inheritance and polymorphism
- Write Java programs using basic syntax elements for looping and flow of control
- Write Java programs that define and manipulate standard Java data types
- Write Java programs that create and manipulate Java arrays and collections
- Write Java classes with constructors, overloaded methods and static members
- Write Java programs that catch and throw exceptions
- Write Java programs that use JDBC to access relational databases
- Write servlets and JSPs in a JEE Web application

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda

Day 1
_____

Introduction to the Course
Introduction to Java
Data Types and Assignment

Day 2
______

Operators
Flow Control
Object-Oriented Programming Fundamentals

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda, cont'd

Day 3

Methods
Exception Handling

Day 4

Arrays and Collections
Introduction to JDBC
Introduction to Servlets

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Agenda, cont'd

Day 5

Introduction to JavaServer Pages (JSPs)
Writing MVC-based JEE Web applications

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Java

- What is Java?
- Hello, World

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Java?

- Java is an object-oriented programming language created by Sun Microsystems, now owned by Oracle
- But Java is more than just a language -- it defines a portable platform for developing software that can execute in many environments



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java History

- Java was created at Sun in the early 1990s by James Gosling, who was initially attempting to create a processor-independent language for TV set-top boxes
- Java's original name was "Oak" after the tree outside of Gosling's office window
- First working application was a Web browser named WebRunner, later renamed to HotJava

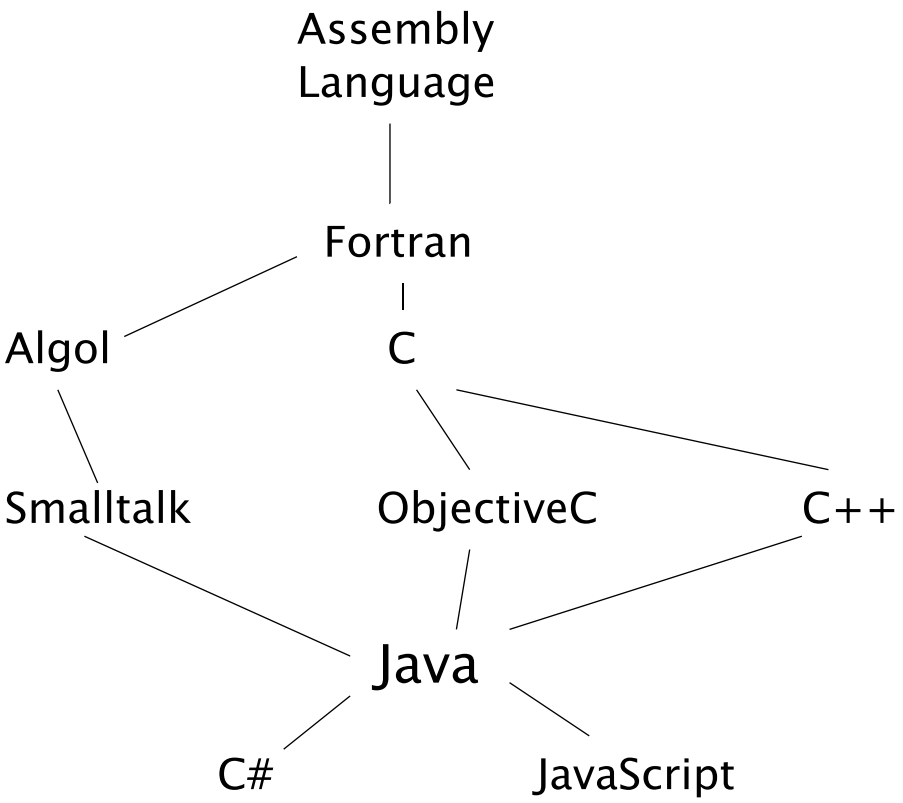
---

You can read more about Java's history here:

http://en.wikibooks.org/wiki/Java_Programming/History

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Language Family Tree



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Programming Platforms

- *Java Standard Edition (JSE)* provides the core Java functionality for client and server-side programing
- *Java Enterprise Edition (JEE)* extends JSE so developers can create server-side applications (e.g. Web applications)
- *Java Micro Edition (JME)* a.k.a *Java ME Embedded* is a specialized Java for small devices (e.g. Cellphones and personal digital assistants)

Note that there's also a Java SE Embedded product which provides a larger set of functionality than JME, but requires more memory and storage. For details, see:

`http://www.oracle.com/technetwork/java/embedded/embedded-se/overview/index.html`

Also note that previously, these platforms had slightly different names that included the digit "2", for example J2EE. The "2" was dropped with the introduction of Java 5.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Standard Edition Version History

Version 1.0
Basic class library, AWT GUI toolkit

Version 1.1
Added new I/O libraries, event-model enhanced internationalization support and inner classes

Version 1.2
New security model, Java Foundation Classes including the Swing GUI toolkit

Version 1.3
Mostly performance enhancements and fixes

Version 1.4
XML support, logging

Java Standard Edition 5
Generics, annotations, formatted I/O

Java Standard Edition 6
Client-side Web services, built-in database

Actually, the term "Standard Edition" didn't come about until Java version 1.2, which Sun's marketing department dubbed "Java 2".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Standard Edition Version History, cont'd

Version 7.0
try-with-resources, strings in switch-case

Version 8.0
Functional programming, parallel programming

Version 9.0
Modular JVM, HTTP v2 support

Versions 10-11
Internal improvements, the "var" keyword

Versions 12-14
Internal improvements, "switch" statement enhanced

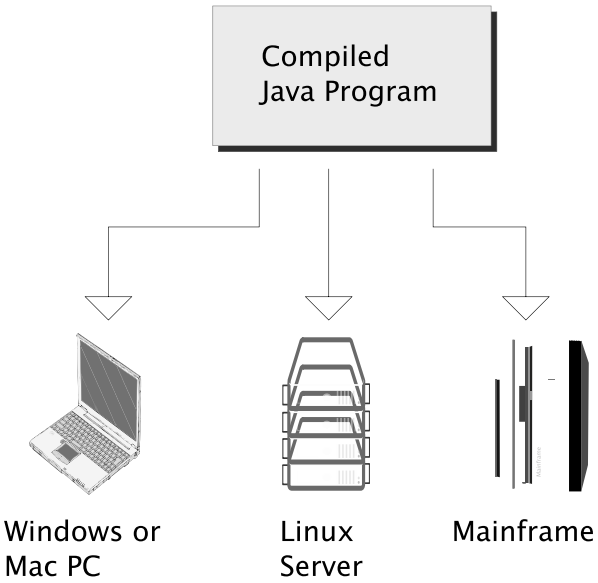
Starting with Java 10, Oracle adopted a faster, 6-month release cadence, with occasional versions referred to as "Long Term Support" (LTS) versions. Java 8 and Java 11 are the current LTS versions (Java 8 support ends in December 2020).

Also, Oracle has open-sourced Java Standard Edition to the OpenJDK project -- Oracle provides support for LTS versions and requires a license for commercial use. OpenJDK versions are completely free.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Portability

- Java's motto is *Write Once, Run Anywhere* -- compiled Java programs should run unchanged on different operating systems and computing platforms



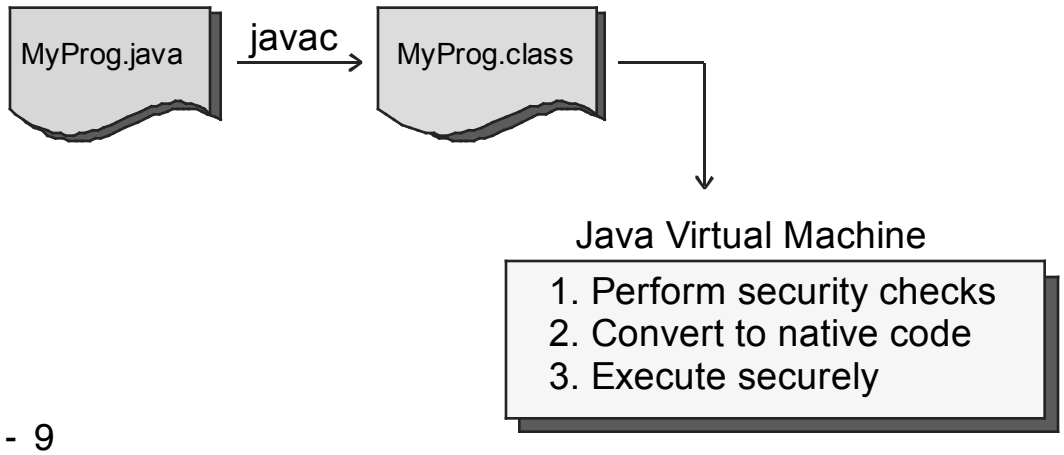
Portability is one of the primary advantages of Java over other languages -- note that we are talking about executable portability, not source-code portability.

Of course, this picture depends on the Java platform itself being ported to each of the environments.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Development Process

- To develop a Java program, you create a source file, then use the Java compiler to create a file containing *byte codes*
- You then pass the byte codes to a *Java Virtual Machine* (JVM) that converts the byte codes to native code and executes



Java programs are compiled, but not into native code. Instead, the compiler produces an intermediary form known as "byte codes". One of the JVM's jobs is to translate the byte codes into native codes on a particular operating system and CPU. So for Java to run on any operating system or CPU, someone must port the JVM to it.

Sun/Oracle has published the specification for the JVM -- anyone can implement it. There currently are several JVMs available from Oracle, IBM and other vendors.

Most developers now use an Integrated Development Environment (IDE) such as Eclipse, NetBeans or IntelliJ that handles the details of compiling and running so the developer doesn't need to use the command line.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Java SDK versus the JRE

- To run Java programs, you need the Java Runtime Environment, which includes the JVM
- To develop Java programs, you need a Java Software Development Kit
- The Oracle Java SDK includes a JRE

The JRE also includes accessory files such as fonts.

Note that some integrated development environments (IDE) such as Eclipse include a Java compiler and only requires a JRE.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Java Class Library

- Java ships with a portable set of pre-written code known as the Java class library

  User-interfaces (AWT and Swing)
  Network I/O (e.g. sockets, HTTP)
  File I/O
  Threads
  Security
  Internationalization
  Collection data structures
  XML processing
  . . .

This is another strength of Java -- unlike languages like C++, which have rudimentary (at best) library support, the JRE contains a very large (and sometimes intimidating) set of functions that you can use in your Java programs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to JEE

- Java Enterprise Edition is a set of specifications for enterprise computing
- Most JEE technologies focus on server-side development

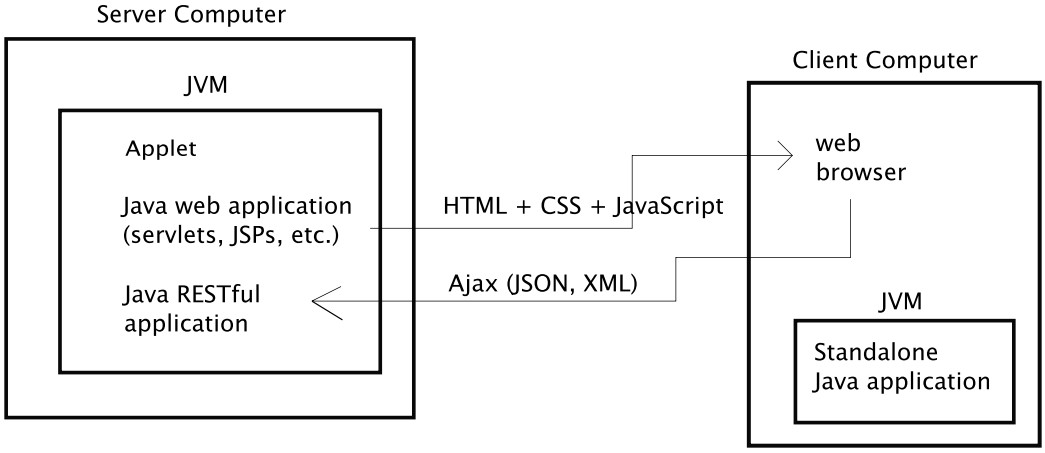
Servlets
JavaServer Pages
Enterprise JavaBeans
Java Naming and Directory Interface
JDBC
Java Message Service
. . .

JEE previously was known as Java 2 Enterprise Edition (J2EE).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Where Does Java Run?

- When Java was first introduced, Sun envisioned that developers would use it to write **standalone** client-side applications and **applets** downloaded from a web server
- However, currently, most Java development is for **server-side** applications that run in response to web requests



Applets are still supported in Java, but have fallen out of favor due to performance and security concerns.

REST is an architecture in which server-side programs provide data, typically formatted using XML or JSON. In that model, JavaScript running in the browser makes asynchronous requests (Ajax) for data as the user interacts with a web page.

Java is also a popular language for developing "batch style" programs that run unattended with no real-time user input.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Hello, World

Test.java

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5         System.out.println ("Hello, world");
6      }
7    }
```

```
c:\> javac Test.java
c:\> dir
03/01/2005  08:16 AM               414 Test.class
03/01/2005  08:16 AM               115 Test.java
c:\> java Test
Hello, world
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- What Java is all about

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Data Types and Assignment

- Defining Variables
- Primitive Data Types
- Assignment

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Program Structure

- The **class** is the basic building block of a Java program
- Classes define state (variables) and behaviors (methods)

```
1    public class MyClass
2    {
3        private int x; // class-scoped variable (field)
4
5        public static void main(String[] args)
6        {
7           int z = 12;        // local variable
8
9           System.out.println("z: " + z);
10       }
11   }
```

In later chapters, you will learn more about classes, but we show a sample class here so you can see the overall structure of a Java program.

Classes define variables to hold data and define methods (a.k.a. functions or procedures) in which you write Java code.

The "main()" method shown here is special since it acts as the entry point of the class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Comments

- Java supports three kinds of comments:
  - Single line comments
  - Block comments
  - JavaDoc comments

```
1    /**
2      @author Paul Westerberg
3      @version 1.0
4    */
5    public class Test
6    {
7      // a single-line comment
8      /* a block comment
9         of multiple lines
10     */
11   }
```

Single-line comments comment-out until the end of the line.

Block comments comment-out until the end of the comment block.

JavaDoc comments are special comments used to create automatically generated documentation for a Java class. The Java SDK includes a javadoc utility that scans source files for JavaDoc comments and generates HTML documentation. Note that the source JavaDoc comments can contain special tags that have meaning to the javadoc processor. For more information on JavaDoc, see:

http://www.oracle.com/technetwork/articles/java/index-137868.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Statements

- In general, Java programs consist of **statements**, each of which is terminated with a semicolon
- Statements must reside within a **block** defined with curly braces

```
 1    public class MyClass
 2    {
 3        private int x;   // class-scoped variable (field)
 4
 5        public static void main(String[] args)
 6        {
 7           int z = 12;        // local variable
 8
 9           System.out.println("z: " + z);
10        }
11    }
```

3 - 4

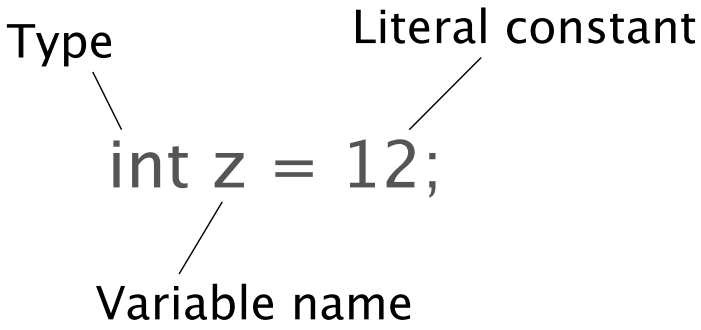
Lines 3, 7 and 9 are examples of statements. Note that statements can span multiple lines -- it's the semicolon that terminates the statement.

This program defines two blocks: one that goes from line 2 until line 11, the other is a nested block from lines 6 to 10. You can nest blocks indefinitely as required.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java is Strongly Typed

- The Java compiler strictly enforces type:
  - You must specify a type for each variable
  - The compiler checks type when you use variables
  - The type of a variable never changes
  - Even literal constant values have a type



Some languages, e.g. JavaScript, have a looser typing system where you don't need to specify a type for variables. The advantage of a strongly typed language is that the compiler can detect more potentially bad uses of variables rather than those errors occuring at runtime.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Variables

- **Variables** are locations in memory that hold a value
- In Java, you define a variable with a statement that specifies both a type and a name (a.k.a an identifier)

Basic syntax:

Type Name [=initial value];

```
1    int count = 12;
2    float interestRate;
3    String sql="SELECT * FROM employee";
```

Any programming language needs variables to hold data for processing. In Java, the basic syntax to define a variable requires you to specify the variable's type, followed by its name, followed by an optional initialization value.

The three examples shown here demonstrate defining variables that contain an initialized integer, an un-initialized floating-point number and a String of characters.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Rules for Identifiers

- Any length combination of letters, digits and special characters: ('_ ', '$') but must begin with an letter or special character
- Java is case sensitive!

Valid	Invalid
count	1account
interestRate	order-number
purchase_Number1	delivery date

The rules for identifiers are straightforward. You can (and should) use names that are descriptive and you can use underscores or uppercase characters to separate words within identifiers.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Reserved Words

abstract	final	outer	try
boolean	finally	package	var
break	float	private	void
byte	for	protected	volatile
byvalue	future	public	while
case	generic	rest	
cast	goto	return	
catch	if	short	
class	implements	static	
clear	import	strictfp	
const	inner	super	
continue	instanceof	switch	
default	int	synchronized	
do	interface	this	
double	long	throw	
else	native	throws	
extends	new	transient	
false	null	true	

This page lists the reserved identifiers that you must not use when you assign names to your variables and other identifiers. Some of these keywords are reserved, but not implemented (e.g. goto).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Data Types

- Java is a strongly typed language -- when you perform an assignment, the compiler flags an error if the types are not compatible
- There are two basic kinds of types in Java: **primitive** types and **reference** types

```
Primitive types:            Reference types:
----------------            ----------------

byte, short, int, long      java.lang.String
float, double               java.util.Date
char                        mypackage.BankAccount
boolean                     ...
```

Primitive types are basic types such as characters, integers and floating-point numbers. Reference types are more complex data, such as Dates, Strings, Connections and so forth. As we will see later, variables of primitive types hold values, while variables of reference types hold a reference to the actual storage for the data.

All reference types are defined in a "package", e.g. java.util. Packages let you organize the types into categories. You can omit the package name for types defined in the "java.lang" package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Integer Types

- Integers are primitive types that contain whole numbers
- Numeric literal constants with no decimal point take on the *int* type

Type	Size (bits)	Size (bytes)	Values	Example
byte	8 bits	1 byte	-128 to 127	byte b = 12
short	16 bits	2 bytes	-32768 to 32767	short sh = 1000;
int	32 bits	4 bytes	-2**31 to 2**31-1	int i2 = 45900;
long	64 bits	8 bytes	-2**63 to 2**63-1	long l = 1000000;

Integers are the workhorse type for most Java programs. Note that there are four sizes of integers that let you trade off storage space and capacity.

Unlike in languages like C, all Java types, including the integer types have a fixed size and range on all platforms. That makes Java more portable.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Floating Point Types

- Floating point values are primitives that contain both whole and fractional parts
- By default, numeric literal constants with a decimal point or scientific notation are of type *double*

Type	Size (bits)	Size (bytes)	Values	Example
float	32 bits	4 bytes	1.4e-45 to 1.4e+38	float f = 12.23F;
double	64 bits	8 bytes	4.9e-324 to 1.7e+308	double d =26.77e3;

Java floating point types adhere to the IEEE standard for floating point numbers.

Floating-point literal constants with no specifiers (e.g. 12.34) take on the double type.

The java.lang.Math package contains useful methods and fields to work with floating-point values (e.g. Math.PI).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Character Type

- The char type holds a single 16-bit Unicode character
- To form a character literal constant, use single quotes

```
1    char ch1 = 'A';
2    char ch2 = "B";  // won't compile
```


Unicode is an encoding scheme that maps each character from most (if not all) national languages into a unique number. Unicode uses at least 16 bits to perform this mapping and thus can map far more characters than the old ASCII and EBCDIC encoding schemes. Since Java uses Unicode internally, Java is well positioned to let you write internationalized (I18N) programs.

For more information on Unicode, see www.unicode.org.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Character Escape Codes

- Java defines special escape codes for common non-printable characters
- You can initialize a character to any Unicode value using an integer constant

```
\b      Backspace
\t      Horizontal tab
\n      Linefeed
\f      Form feed
\r      Carriage return
\"      Double quote
\'      Single quote
\\      Backslash
\uxxxx  Hexadecimal Unicode code point
\xxx    Octal Unicode code point
```


You can specify any Unicode character using the character's code point and the '\uxxxx' notation, but common characters have shortcuts (e.g. \n). Examples:

char ch1 = '\n'; char ch2 = '\u00DC'; // German U umlaut

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Boolean Type

- The boolean type allows only the literal values true and false
- The most common use of booleans is in flow control statements (e.g. the "if" statement)

```
boolean finished = false;
boolean ready = true;
```


Java programmers often use boolean variables in loops or conditional statements (e.g. "if" statements). The keywords "true" and "false" are considered literal boolean constants.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The String Type

- Strings are NOT primitives - they are reference types that hold sequences of characters
- To define a literal String, use double quotes
- You can concatenate Strings using the + operator

```
1    String s1 = "If Only You Were Lonely";
2    int index = s1.indexOf ( "Only" );
3    boolean same = s1.equals ( "The Ledge" );
4
5    String s2 = s1 + " Not";
```


Unlike the previously covered types, Strings are object types, not primitives. That means that Strings are "smarter" than primitives -- for example, Strings support searching, tests for equality, extraction of sub-strings and so forth.

As we will cover later in the course, String is actually an object-oriented class that's part of the standard Java class library.

When you initialize a variable for an object type, the variable holds a reference to the object, not the object's value.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Assignment

- The = operator assigns the right-side value to the left-side
- The right-side data type must be compatible with the left side, or else the compile flags an error

```
1    int i1 = 123456;
2    short s1 = 12;
3    float f1 = 12.34F;   // OK
4    float f2 = 12.34;    // compile error
5    double d1 = 1.4e87;
6    String str1 = "ABCD";
7
8    int i2 = s1;    // OK
9    short s2 = i1;  // compile error
10
11   double d2 = f1; // OK
12   float f3 = d1;  // error
13
14   i2 = str1;      // compile error
```


For an assignment to be avoid compile errors, the left-side type must be at least as "large" as the right-side type.

Note that it's generally not OK to assign a reference type (e.g. String) to a primitive variable or vice versa. The exception to the rule is that the Java class library contains so-called "wrapper" classses such as lava.lang.Integer and java.lang.Character. In Java v5 and later, the compiler lets you assign to/from the wrappers and primitives - a process referred to as auto boxing/unboxing.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Cast Operator

- To force an assignment, you can use the cast operator on the right side of an expression
- You should avoid casting if you can!

```
1    int i1 = 123456;
2    short s1 = 12;
3    float f1 = 12.34F;
4    double d1 = 1.4e87;
5
6    short s2 = (short)i1;   // compiles OK, truncates
7
8    float f2 = (float)d1;  // compiles OK,
9                           // result is Infinity
10                          // (overflow)
```


The cast operator converts from one type to another. To form a cast, you enclose the target type within parenthesis.

Casting does help in some situations to avoid compile errors, but may result in odd runtime behavior.

It is possible to cast from one reference type to another, but not to/from a reference type and a primitive.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Scope of Variables

- Variables must be defined within a *block* defined by curly braces { }
- Variables defined within a block are local to that block

```
1    {
2       int i = 12;
3       {
4          int j = 13;
5          int k = i;   // OK
6       }
7       int m = j;  // compile error
8    }
```


You can define a block by using a pair of curly braces -- any variables defined in a block are local to that block (and any sub-block). Code within a sub-block can access variables defined in any outer block.

The code shown here is a contrived example to show the syntax, but it's quite common to nest blocks within other blocks, especially using looping or conditional statements.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Scope of Variables, cont'd

- Variables defined at *class scope* are global to all methods in the class

```
1    public class Test
2    {
3        private int i1 = 15;  // field
4
5        public void method1()
6        {
7           int i2 = 14;  // local variable
8           i2 = i1 + 5;  // OK
9        }
10       public void method2()
11       {
12          int i3 = i2;  // compile error
13          int i4 = i1 + 5;  // OK
14       }
15   }
```

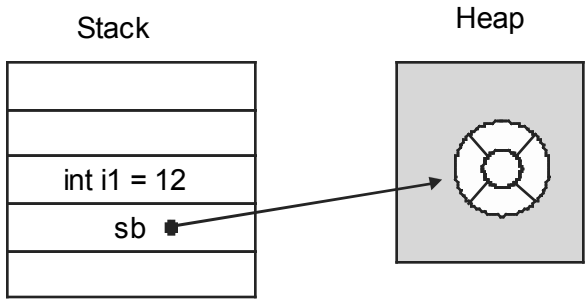

The code on this page shows a Java class definition -- we will cover the syntax in more detail later, but please note that a class defines a block using curly braces. In lines 5 to 9, we nest a method (a.k.a. function) definition which also defines a block. A class can contain many methods, each with its own block -- in lines 10-14 we define another method.

In line 3, we define a variable at "class scope" -- such variables are global to all methods within the class and are referred to as "fields". In contrast, line 7 defines a local variable, "i2", which is only available to the method in which it's defined -- that's why line 12 generates a compile error.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Stack and the Heap

- The Java Virtual Machine defines two basic areas for variables: the *stack* and the *heap*
- Local variables are allocated on the stack, while dynamically allocated storage comes from the heap




```
1    public class Test
2    {
3      public void method1()
4      {
5         int i1 = 12;
6         StringBuffer sb = new StringBuffer();
7      }
8    }
```

3 - 20

The "method1" method defines two local variables: i1 and sb. When the method is called, these two variables will be allocated on the JVM's stack and will be de-allocated when the method finishes.

StringBuffer, like the String type covered earlier, is a reference type that's part of the Java standard class library. Lines 6 and 7 allocate a StringBuffer object using Java's "new" operator, which allocates storage for the StringBuffer object from the JVM's "heap", which is the free storage in the JVM.

Note that the sb variable, like all reference variables, does not contain the actual StringBuffer object, but instead references, or points to the actual object on the heap.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Define an integer variable and initialize it to your favorite whole number:
- Define a floating point number variable and initialize it to your favorite floating-point number:
- Define a String variable and initialize it to your name:

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The rules for defining and initializing variables
- The Java primitive data types
- About the stack and the heap

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Operators

- Operator Types
- Assignment

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Operators

- Operators let you perform math, comparisons and so forth
- Java provides more than 30 operators, which we can divide into categories:
    - Math
    - Equality and Relational
    - Assignment
    - Boolean Logical
    - Integer Bitwise
    - Shift

Note that Java 8 provides a few more operators.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Math Operators

- Java defines math operators that work on numeric types (int, float and so forth)
- The compiler will promote operands to the larger type if necessary
- Integer division by zero causes an ArithmeticException to be thrown

```
+    Addition
-    Subtraction
*    Multiplication
/    Division
%    Remainder
```

These operators should be familiar to anyone who has taken an algebra course! The only one that might not be familiar is the remainder operator, which returns the remainder of integer division. For example, 7 % 3 = 1.

You should also note that just like in algebra, you can use parenthesis to force an math expression to evaluate in the order you desire.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Math Operators, cont'd

```
1    int firstInt = 12;
2    double firstDoub = 12.34;
3
4    System.out.println (firstInt + firstDoub);
5
6    int secondInt = 0;
7    try
8    {
9        System.out.println (firstInt / secondInt);
10   }
11   catch (ArithmeticException exc)
12   {
13       System.out.println ("Math error! " +exc);
14   }
```

This code fragment demonstrates promotion of an integer to a double and also how to handle an ArithmeticException. We will cover exception handling in more detail later in the course.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Other Math Operations

- The java.lang.Math class defines additional math operations beyond the standard math operators (e. g. trig functions)
- These **methods** are not intrinsic Java syntax, but are part of the Java class library

```
1    double d = Math.sin ( Math.PI / 2 );
2    System.out.println ( d );
3
4    double radius = 5.3;
5    double area = Math.pow ( radius, 2 ) * Math.PI;
6    System.out.println ( area );
```

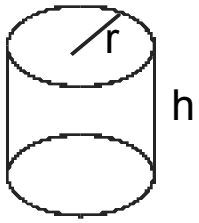
Just for completeness, we show here some of mathematical functions from the Math class. These are not actually part of the Java syntax -- instead, they are defined as methods in that class. Please read the Math documentation for a complete listing of the methods in this class.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- The equation for the surface area of a cylinder is:

$$2\pi r^2 + 2\pi rh$$



- Complete this program fragment to calculate a cylinder's surface area:

```
1    double r = 12.0;
2    double h = 3.0;
3
4    double surfaceArea =
5
6    System.out.println(surfaceArea);
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Compound Assignment Operators

- Java provides several compound assignment operators

```
*= /= %= += -= <<= >>= >>>= &= ^= |=
```

```
1    int myInt = 12;
2
3    myInt += 45; // same as myInt = myInt + 45;
4    System.out.println (myInt);
```


Compound assignment operators are shorthand syntax for performing an operation and an assignment. The example demonstrates using a compound assignment to perform an addition and assignment. With the exception of += and Strings, all of the compound assignment operators work only on primitive types.

We will cover the other operators later in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Increment and Decrement Operators

- These shortcut operators come in two forms: **prefix** and **postfix**
- Postfix operators work after the expression is evaluated, prefix before

```
          ++ --
```

```
1    int firstInt = 12;
2    firstInt++; // same as firstInt = firstInt + 1;
3    System.out.println (firstInt);
4
5    int secondInt = --firstInt * 3;
6    System.out.println (secondInt);
```

These operators work only on primitives and change the value of a variable. The interesting thing about them is that you can use them in expressions -- they are especially useful in looping constructs.

The example shows using the postfix ++ and prefix --.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Equality Operators

- Equality operators return a boolean result and are most often used in "if" statements or loops
- Equality tests of primitives test values, while equality tests of reference types tests the references

```
== !=
```

```
1    int firstInt = 12;
2    int secondInt = 14;
3
4    boolean b = firstInt == secondInt;
5    System.out.println (b);
6
7    if (firstInt != secondInt)
8        System.out.println ("Different!");
```

These operators work on: primitive numeric types, booleans, reference types or the literal value "null". The operators always return a boolean type.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Relational Operators

- The numeric comparison operators work on primitive numeric types and return a boolean result
- The **instanceof** operator works on reference types and returns true only if the right-side is of the same type or subtype

```
< <= > >= instanceof
```

```
1    int firstInt = 12;
2    int secondInt = 14;
3
4    boolean b = firstInt >= secondInt;
5    System.out.println (b);
6
7    if (firstInt < secondInt)
8        System.out.println ("Smaller!");
```

The relational operators let you compare numeric values or reference types and always return a boolean result. The numeric relational operators are especially handy for "if" statements and looping constructs.

The "instanceof" operator returns true if a reference variable is compatible with a specified type. It's mostly useful when using inheritance, which we will cover later in the class, but here's a trivial syntax example:

```
String s = "Hello";
if(s instanceof String)
  System.out.println("Yep, it's a String");
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Integer Bitwise Operators

- You can use integer bitwise operators to test or manipulate bit patterns within integer values (byte, short, int, long)

```
& ^ |
```

```
1    int firstInt = 18;
2    int secondInt = firstInt & 8;   // 10010 and 01000
3    int thirdInt =  firstInt | 8;   // 10010 or  01000
4    int fourthInt = firstInt ^ 16;  // 10010 xor 10000
5    System.out.println(secondInt);  // 0
6    System.out.println(thirdInt);  // 26
7    System.out.println(fourthInt);  // 2
```

When you use these operators on integral numeric primitive types, they perform bit-level manipulation. They are not valid for characters, floating-point types or reference types. They do work on booleans as discussed on the next page.

For an "and" to result in a "1", both inputs have to be "1".

For an "or" to result in a "1", at least one input needs to be "1".

For an "exclusive or" to result in a "1", the inputs must be different.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Boolean Logical Operators

- You can use logical operators on booleans to build more complex testing expressions

```
          & ^ |
```

```
1    boolean atHome = true;
2    boolean eatingDinner = false;
3
4    boolean result = atHome & eatingDinner;
5    System.out.println ( result );
6
7    result = atHome | eatingDinner;
8    System.out.println ( result );
9
10   result = atHome ^ eatingDinner;
11   System.out.println ( result );
```

These operators are most often used in "if" statements or looping constructs. For an "and" expression to be true, both operands must be true. For "or" to be true, either operand can be true. For exclusive or (^) to evaluate to true, the operands must be different.

Note that there are similar logical operators for and (&&) and or (||) shown on the next page.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Conditional Operators

- The **conditional operators** work similarly to the boolean logical operators, but they can "short-circuit"

&& ||

```
1    int first = 12;
2    int second = 13;
3
4    boolean result = (first < second) || (++first == second );
5    System.out.println (result);
6    System.out.println (first);
7
8    result = (first > second) && (++first == second );
9    System.out.println (result);
10   System.out.println (first);
```

We call these operators short-circuit operators because Java will not bother evaluating the second part of the expression if it doesn't need to. In the example, in line 4, the second expression containing the ++ operator will not execute, since an "or" expression is true if either operand is true and the first expression is true. In other words, Java can stop evaluating an "or" expression as soon as it finds a true value. Same for line 8, since Java can stop evaluating "and" as soon as it finds a false value.

While this might seem tricky, it can be useful in expressions like:

boolean result = (myString != null) && (myString.length() > 5);

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Shift Operators

- You can use the shift operators to move bits within integer values (byte, short, int, long)
- You can use shift operators to multiply and divide by powers of two

<< >> >>>

```
1    int first = 18;  // 10010
2    System.out.println (first << 1); // 18 * 2 = 36
3    System.out.println (first >> 2); // 18 / 4 = 4
4    int second = -18; // 11111111111111111111111111101110
5    System.out.println (second <<  1); // -18 * 2 = -36
6    System.out.println (second >>  2); // -18 / 4 = -5
7    System.out.println (second >>> 2); // 1073741819
```

These operators perform a low-level operation on numeric integral types. In some cases performing multiplication and division in this fashion may be faster than using the multiply and divide operators.

In memory, Java stores integers using so-called "two's complement" format, in which the most-significant bit acts as a "sign bit" - a zero in that position indicates a positive number, while a one designates negative. For more details, see:

https://en.wikipedia.org/wiki/Two%27s_complement

Note that you can easily print an integer's binary representation using code like:

```
System.out.println(Integer.toBinaryString(second));
```

The >>> operator performs a right-shift without preserving the integer's sign, while >> keeps the sign bit at its original value, i.e. positive numbers stay positive and negative stay negative.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Conditional Operator

- The conditional operator is a shortcut for an if-else statement

```
? :
```

```
1    int first = 12;
2    int second = 14;
3
4    int min = first < second ? first : second;
5    System.out.println (min);
```

This ternary (three operand) operator is a syntax that Java inherited from the C language. Java evaluates the first operand -- if it evaluates to true, the entire expression evaluates to the second operand. If false, the entire expression evaluates to the third operand.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Boolean ! Operator

- The **not** operator flips the value of a boolean and is often used in "if" statements or loops

!

```
1    boolean atHome = true;
2    boolean eatingDinner = false;
3
4    boolean result = !(atHome && eatingDinner);
5    System.out.println ( result );
```


This unary (single-operand) operator has a high precedence, and thus is often used with parenthesis.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

String Concatenation

- You can use the + and += operators to concatenate Strings

```
1    String s1 = "Hello";
2    String s2 = s1 + " world";
3
4    s1 += " Dolly";
```


The String class is the only non-primitive type that supports this operator. Under the covers, when you concatenate Strings, Java actually creates a new String with the additional characters and discards the original String (makes it available for garbage collection).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Other Operators

.	Object member access
[]	Array element access
()	Method call
new	Creates a new object or array from the heap
()	Cast operator
-	"Flips" the sign of numeric values

For completeness, here we show the operators not covered explicitly in this chapter. We will see most of them later in the course.

Here's a quick example of the negation operator that "flips" the sign of a numeric value:

```
int a = 12;
int b = -a;
System.out.println(b);  // -12

int c = -456;
int d = -c;
System.out.println(d);  // 456
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Operator Precedence



postfix	[] ,., (params), expr++, expr–
Unary	++expr, –expr, +expr, –expr !
Multiplicative	* / %
Additive	+ -
Relational	<, >, <=, >=
Equality	==, !=
Logical AND	&&
Logical OR	||
Ternary	?:
Assignment	=, +=, -=, *=, /=, %=

To force an order of precedence, use paranthesis.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About all of the Java operators
- How to create expressions that you can use in "if" statements and loops

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Flow Control

- Conditional Statements
- Looping

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Defining Blocks

- In Java, all executable code must be within a block defined by a pair of curly braces
- You can nest blocks within other blocks

```
1    public class Test
2    {
3      public void myMethod(int x)
4      {
5         if ( x == 12 )
6         {
7            // do something interesting
8         }
9      }
10   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Conditional Statements

- Like all complete programming languages, Java provides syntax for making decisions:
  - if
  - Conditional (ternary) operator

Conditional statements let programs make decisions, generally by evaluating an expression that evaluates to a boolean. The basic conditional statement in Java is the "if" statement.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The if Statement

- The *if* statement examines an arbitrarily complex boolean expression

```
if ( <expr> )
{
  // statements
}


if ( <expr> )
  // one statement
```


The "if" statement evaluates the provided boolean expression and executes the block or single statement that follows. The parenthesis are required.

If you only need a single statement, you can use the form that does not define a block. However, it never hurts to define the block and can save time later.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The if else Statement

- The *if else* statement examines an arbitrarily complex boolean expression

```
if ( <expr> )
{
  // statements
}
else
{
  // statements
}
```


The "if - else" statement lets you conditionally execute one of two blocks. And as shown on the previous page, if you have only one statement in the block, you can omit the curly braces.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The if else if Statement

- The *if else if* statement examines an arbitrarily complex boolean expression

```
if ( <expr> )
{
  // statements
}
else if ( <expr> )
{
  // statements
}
else
{
  // statements
}
```


The "if - else if - else" lets you create complex decisions. Note that the final "else" is optional.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The if else if Statement, cont'd

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4      {
5        java.util.Random rand = new java.util.Random();
6        int a = rand.nextInt();
7        int b = rand.nextInt();
8        int c = rand.nextInt();
9        if ( (a + b) > c )
10         System.out.println ( "Sum is greater" );
11       else if ( (a+b) == c )
12         System.out.println ( "The same" );
13       else
14         System.out.println ( "Sum NOT greater" );
15     }
16   }
```

This simple Java program retrieves three random integers, then determines if the sum of the first two is greater than, less than, or equal to the third.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- The *System.in.read()* method accepts a keystroke from the user as a *char*. Write a program fragment that accepts a keystroke and calls the *System.exit(0)* method if the keystroke is an upper or lower case Q. For any other keystroke, display the keystroke character to the console.

```
char key = (char)System.in.read();
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The while Loop

- The *while* loop evaluates a boolean expression and stays in the loop as long as the express is *true*

```
while ( <expr> )
{
  // statements
}

do
{
  // statements
}
while ( <expr> );
```

The difference between "while" and "do while" is that "do while" always does at least one pass through the loop.

Also note that if you have only one statement, you can omit the curly braces.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The while Loop, cont'd

```
 1    public class FlowControl
 2    {
 3      public static void main (String[] args)
 4      {
 5        int i = 0;
 6        while ( i < 10 )
 7        {
 8          System.out.println ( "in while: " + i );
 9          i++;
10        }
11        do
12        {
13          System.out.println ( "in do-while: " + i );
14          i++;
15        } while ( i < 10 );
16      }
17    }
```

This Java program has two loops. The first, from lines 6 to 10, runs 10 times with "i" ranging from 0 to 9.

The second loop, from lines 11 to 15, runs once.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The for Loop

- The *for* loop is a convenient and concise form of the *while* loop

```
for ( <init statement(s)>;<expr>;<exec statement(s)> )
{
  // statements
}
```


The "for" loop is "syntactic sugar" that lets you concisely write common looping constructs. The syntax requires a parenthesis with three fields, separated with semicolons. The first field contains statements that are executed before the loop begins -- if you have multiple statements, separate them with commas. The second field (between the semicolons), must be an expression that evaluates to a boolean -- the loop continues as long as the expression is true. The third field is any statements that are executed at the end of the loop -- again, if you have more than one statement, use commas to separate them.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The for Loop, cont'd

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4      {
5        for ( int i = 0; i < 10; i++ )
6        {
7          System.out.println ( "for loop: " + i );
8        }
9        for ( int i = 10,j = 0; i > 0; i--, j++ )
10       {
11         System.out.println ( "(" + i + "," +
12                                    j + ")" );
13       }
14     }
15   }
```

Here we show two examples of the "for" loop. The first, is a simple loop that counts from 0 to 9. The second illustrates using multiple "initial statements" and "executing statements". Its output looks like: (10,0)
 (9,1)
 (8,2)
 (7,3)
 (6,4)
 (5,5)
 (4,6)
 (3,7)
 (2,8)
 (1,9)

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The for Loop, cont'd

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4      {
5        int i = 0;
6        for ( ; i < 10; )
7        {
8          System.out.println ( "for loop: " + i );
9          i++;
10       }
11     }
12   }
```


This "for" loop works the same as a "while" loop shown a few pages ago. It illustrates that you can write any "for" loop as a "while" loop and also shows that you can leave portions of the "for" loop empty, with semicolons as placeholders.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Breaking Out of a Loop

- Sometimes it's useful to exit a loop prematurely using the *break* statement

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4      {
5        java.util.Random rand = new java.util.Random();
6        while ( true )
7        {
8          int i = rand.nextInt();
9          if ( i < 0 )
10           break;
11         System.out.println ( i );
12       }
13     }
14   }
```


Note that there's also a labeled "break" that is mostly useful for exiting nested loops and also terminating an outer loop.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The switch-case Statement

- The *switch-case* statement lets you choose between values of an integer or character type

```
switch ( <int or char type> )
{
  case <int or char const>:
    // statements
    break;
  case <int or char const>:
    // statements
    break;
  default:
    // statements
    break;
}
```


The "switch-case" statement has some rather odd characteristics, but is sometimes useful. The syntax requires parenthesis around an integer or character variable and then zero or more "cases" and an optional "default". At the end of each case, you can use the "break" statement to terminate the case -- if you omit it, then execution falls through to the next case.

The allowable types for the <int or char type> are "byte", "char", "short" or "int". In Java 7 and later, you can also use java.lang.String.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The switch-case Statement, cont'd

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4      {
5        java.util.Random rand = new java.util.Random();
6        int a = rand.nextInt();
7        switch ( a )
8        {
9          case 1:
10           System.out.println ( "Value is One!" );
11           break;
12
13         case 42:
14           System.out.println ( "Value is 42!" );
15           break;
16
17         default:
18           System.out.println ( "Value is something else" );
19       }
20     }
21   }
```

This examples uses the "switch-case" statement to determine if a random integer has a value of 1, 42 or something else.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The switch-case Statement, cont'd

```
1    public class FlowControl
2    {
3      public static void main (String[] args)
4        throws Exception
5      {
6        while ( true )
7        {
8          char c = (char)System.in.read();
9          switch ( c )
10         {
11           case 'd':
12           case 'D':
13             System.out.println(new java.util.Date());
14             break;
15           case 'q':
16           case 'Q':
17             System.exit ( 0 );
18         }
19       }
20     }
21   }
```


Here's another "switch-case" example that illustrates a reasonable use of the case "fall-through".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Strings in Switch Statement

- Prior to Java 7, the **switch** statement only worked with **byte**, **short**, **int** and **char**
- Java 7 and later also support the java.lang.String type

```
1    String input = ????
2
3    switch(input)
4    {
5      case "sam":
6        System.out.println("Hi sam!");
7        break;
8
9      case "sue":
10       System.out.println("Hi there sue!");
11       break;
12
13     default:
14       System.out.println("Who are you?");
15   }
```


Note that the string comparison is case sensitive.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Write a loop that counts from 10 to 100 by fives and displays each value:

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The basics of Java conditionals
- About looping contructs

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Object Orientation

- What is an Object?
- Three Pillars of OO

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Object-Oriented Programming?

- Object oriented programming (OO) is a programming philosophy
- OO is also a set of techniques to achieve that philosophy
- Some programming languages, like Java, provide support for OO techniques
- OO will not solve all of the worlds programming ills

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Object-Oriented Languages

- To best apply OO techniques, you should use a language written expressly for OO
- OO languages include:
  - Smalltalk
  - Java
  - C#
  - C++
  - JavaScript

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Goals of Object Orientation

- Reuse (the Holy Grail)
- Easier to understand code
- Malleability
- Easier to maintain and debug
- Manage complexity for large projects
- Model the real world

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Three Pillars

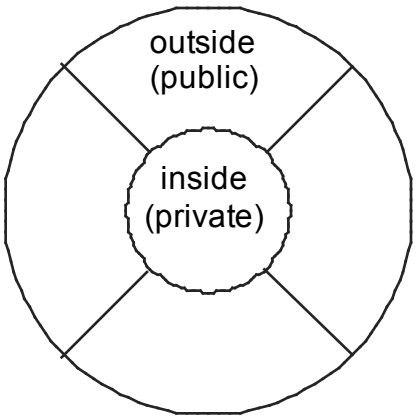
- To be considered a "true" OO language, the language should support:
  - Encapsulation
  - Inheritance
  - Polymorphism

Peter Wegner of Brown University devised a set of requirements for languages, known as the Wegner Definitions. In the Wegner Definitions, to be considered an object-oriented language, the language must support objects, inheritance and polymorphism.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is an Object?

- An object is a self-contained chunk of state (data) and behavior (functions)
- Objects have private and public sections
- You can think of objects as being "alive", e.g. we might say that an object does this or that (executes its behavior)



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

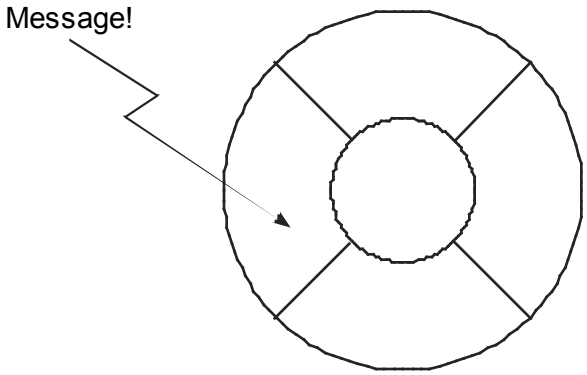
Why are Objects Useful?

- People naturally think of objects in the "real world"
- We intuitively understand how real-world objects have state (e.g. a dog's size) and behavior (e.g. eat)
- Often, we can model real-world objects with objects in a program (e.g. a BankAccount object)

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

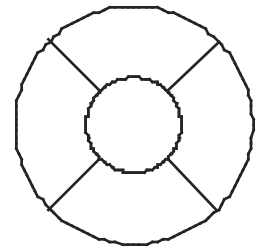
Objects Respond to Messages

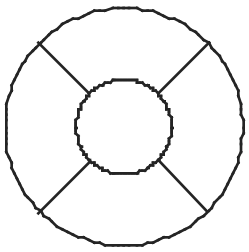
- We refer to executing an object's behavior as "sending the object a message". In Java, we "send messages" by calling methods in the object
- The methods then typically operate on the state (data)

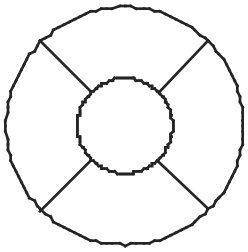


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Objects and Classes



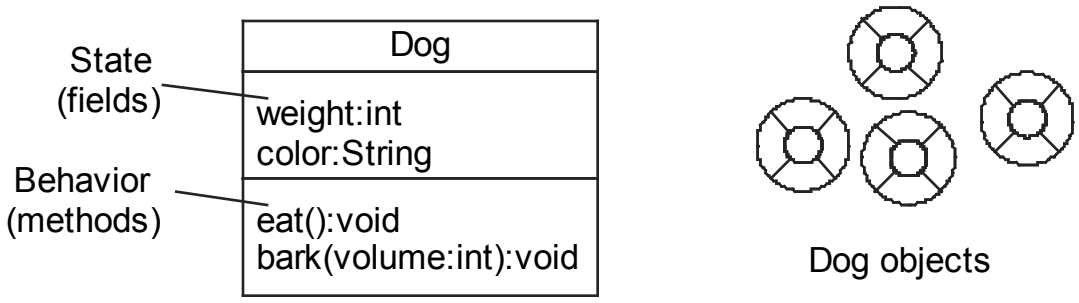




Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Class is Like a Cookie Cutter

- In most O-O languages, you must define the class before you can create objects
- You can then use the class to "stamp out" objects of that class



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing a Class

- A class is an abstraction that defines a collection of data (state) and the functions (behavior) that operate on the data

```
public class Dog
{
  // define fields (state)

  // define methods (behavior)
}
```


In Java, by convention, class names begin with an upper-case character.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Java Class

You will write the class for Dog during lecture.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Instantiating Objects

- A class is like a cookie-cutter for objects
- The act of creating objects is referred to as *instantiation*

```
1    public class TestDog
2    {
3      public static void main(String[] args)
4      {
5         Dog lassie = new Dog();
6         Dog fido = new Dog();
7         lassie.eat();
8         fido.bark(8);
9      }
10   }
```

The Java "new" operator instantiates objects.

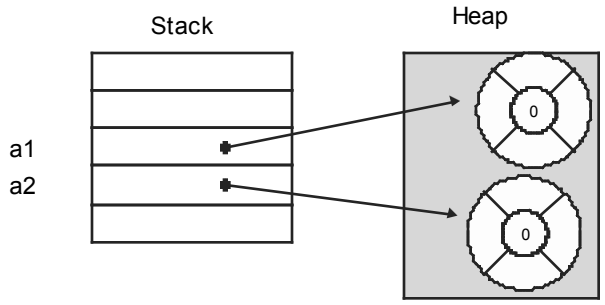
Since Java supports runtime garbage collection, there is no "delete" operator -- the Java Virtual Machine automatically gets rid of unreferenced objects.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Methods Operate on Objects

- Methods typically manipulate the object's state
- Methods invoked on one object don't affect other objects

```
1    {
2      Account a1 = new Account();
3      Account a2 = new Account();
4
5      a1.withdraw (100);
6      a2.deposit (300);
7    }
```



This is a very important notion in regards to object-orientation. You should think of methods as being like messages sent to an object -- messages processed by one object have no effect on other objects of the same type.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Methods Must Have a Receiver

- To invoke a method, you must explicitly or implicitly specify the **receiver** of the method
- This makes Java different than languages like C or C++ where you can have global methods
- The receiver is normally an object, but is a class for **static** methods (covered later)

```
Account a2 = new Account();

a2.deposit(300);
```

receiver method argument(s)

This is why referring to calling methods as "sending a message" is useful -- it emphasizes that the method must run in the context of an object (or a class, in the case of static methods).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implicit Receivers

- If you call a method from within the same class, you can either explicitly specify **this** as the receiver, or use it implicitly

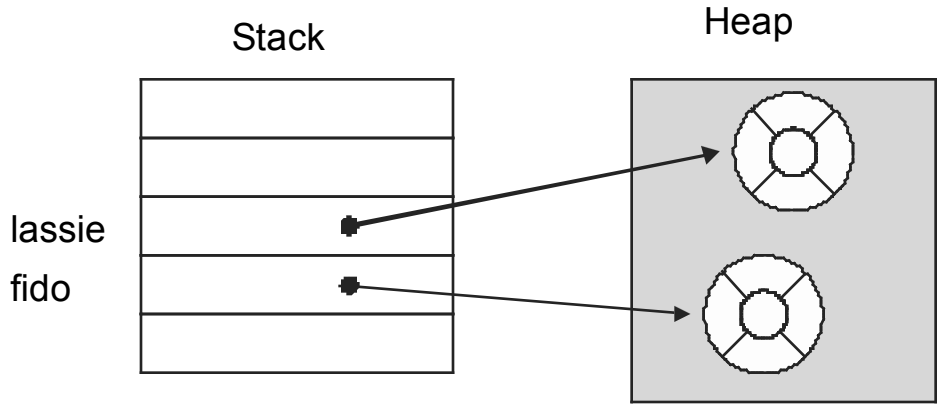
```
1    public class Test
2    {
3       public void method1()
4       {
5       }
6       public void method2()
7       {
8           this.method1();
9           method1();
10      }
11   }
```

The keyword "this" means "the current object". In SmallTalk, the same concept uses the keyword "self", which might be more intuitive than Java's "this".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Stack and the Heap

- The *new* operator carves out storage for the object from a memory area referred to as the heap



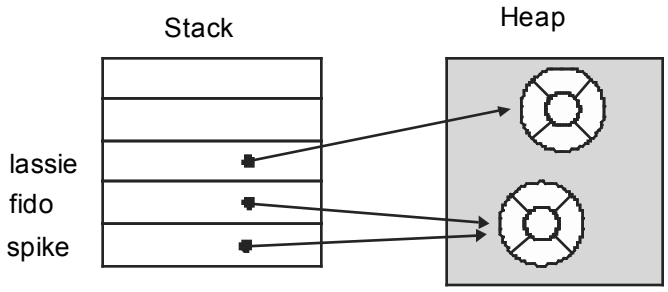
The heap is the free storage remaining in the Java Virtual Machine.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Comparing References

- You can use the *equality* operator to compare references

```
1    {
2      Dog lassie = new Dog();
3      Dog fido = new Dog();
4      Dog spike = fido;
5
6      if (fido == spike)
7        System.out.println ("Same!");
8      if (lassie == fido)
9        System.out.println ("Same!");
10   }
```



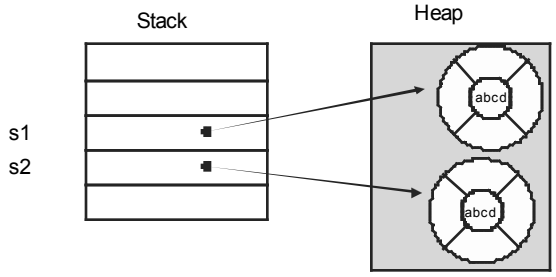
When you use the equality operator on references, it compares the reference values, not the objects themselves.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Comparing References, cont'd

- When applied to references, the *equality* operator compares the references themselves, not the objects to which the references refer

```
1    {
2      String s1 = new String("abcd");
3      String s2 = new String("abcd");
4
5      if (s1 == s2)
6        System.out.println("Same!");
7      if (s1.equals(s2))
8        System.out.println("Equals!");
9    }
```



Even though the two String objects have the exact same characters, the equality operator applied to the objects' references returns false. That's because there are two distinct objects on the heap, each with a different address.

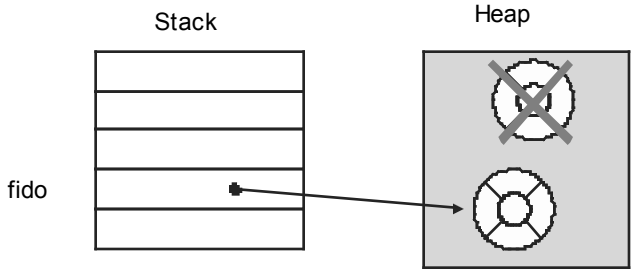
To compare the contents of the objects, the String class provides the equals() method, which returns true only if the two Strings have the same characters (it is a case-sensitive comparison).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Garbage Collection

- The garbage collector runs as a *background thread* that periodically scans the stack and the heap looking for unreferenced objects
- The garbage collector automatically deletes all unreferenced objects

```
1    {
2      Dog lassie = new Dog();
3      Dog fido = new Dog();
4      . . .
5      lassie = null;
6      . . .
7    }
```



Java defines the keyword "null" which you can assign to a reference to indicate that the reference is no longer in use.

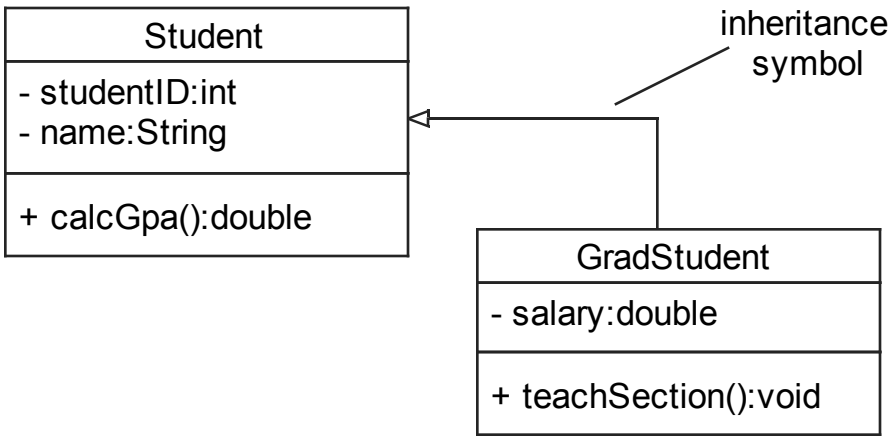
An object on the heap becomes unreferenced either when you explicitly set all of its references to null, or if the local variables that contain the object's references go out of scope.

You can force the garbage collector to run by calling the gc() method on the System class, which is part of the Java class library included in the JRE.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Inheritance

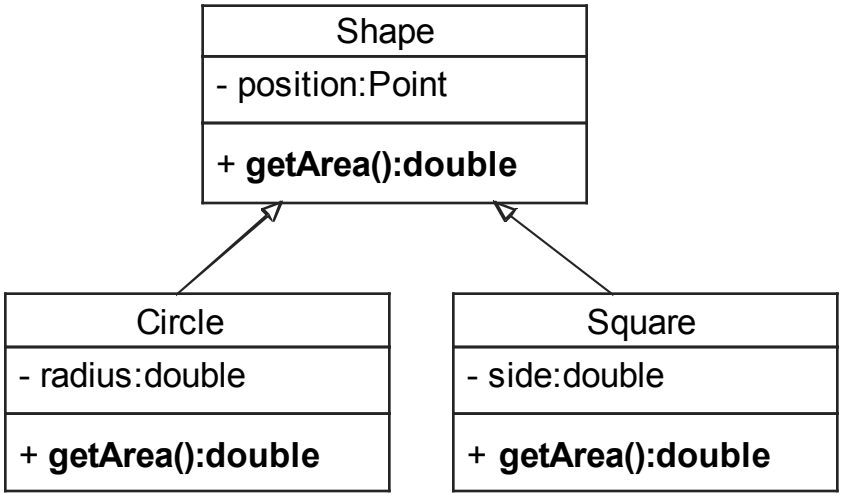
- A subclass inherits state and behavior from its superclass
- Inheritance makes it easier to extend and reuse code
- **Important:** A subclass object is a superset of all of its superclasses



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Polymorphism

- Polymorphism means "many forms"
- Polymorphism helps you design extensible code where subclasses override behavior from superclasses



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Synonym Alert!

Generic OO Term	C++ Term	Java Term
state	member data	field
behavior	member function	method
inherit from (specialize)	subclass	extend
superclass	base class	superclass
subclass	derived class	subclass

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Packages

- Java lets you organize your classes into *packages*
- If a class needs to use a class from another package, it must either fully-qualify the reference or use the *import* statement

```
1    package mypackage;
2    import java.util.Date;
3    import java.net.*;
4
5    public class Test
6    {
7      public static void main(String[] args)
8      {
9         Date d = new Date();
10        Socket s = new Socket();
11        java.io.File myFile =
12          new java.io.File ( "test.txt" );
13     }
14   }
```


Packages define namespaces so you can potentially re-use class names (i.e. you can use the same class name as long as the classes are in different packages). You can also nest packages within other packages.

If present, the "package" statement must be the first non-comment line in the file.

We refer to classes that have no package statement as being in the "unnamed" or "default" package.

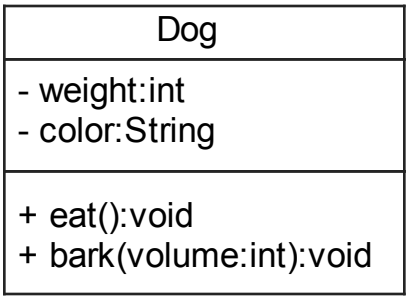
The compiler automatically imports everything from the java.lang package for you.

Note that the import statement has two syntaxes: one that imports a single type and another that imports all of the types from a specified package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Encapsulation

- OO languages enforce encapsulation by restricting access to class functions and data
- Encapsulation makes it easier to maintain and reuse code (black box approach): you can safely modify hidden parts



- private
+ public

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Encapsulation

```
1    public class Dog
2    {
3       private int weight;
4       private String color;
5
6       public void eat()
7       {
8       }
9
10      public void bark (int volume)
11      {
12      }
13   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Encapsulation Summary

```
1    public class Test
2    {
3      public int a;
4      private int b;
5      protected int c;
6      int d;
7    }
```

Access Specifier	Within same class	Within a subclass	Within same package	Outside world
private	●			
protected	●	●	●	
default	●		●	
public	●	●	●	●

Java provides four ways to encapsulate fields and methods: public, private, protected and default, which is known as "package-level".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Get/Set Methods

- Since fields are normally private, you can provide controlled access with get/set methods

```
public class Account
{
  private double balance = 100.00;
  . . .
  public double getBalance()
  {
    return this.balance;
  }
  public void setBalance(double balance)
  {
    this.balance = balance;
  }
  . . .

}
```


Get/Set methods, also known as accessor/mutator methods, give the outside world controlled access to private fields. In other words, the only way code outside of the class can read or write private fields is via the get/set methods, in which we can write code to do whatever we want (e.g. write a "set" method that only allows changes within a range of values). If you were to make fields public, then you would have no control over how the fields are read and written.

This code also shows using the Java "this" keyword, which refers to the current object. If you omit this keyword on a field access, the compile assumes it. Note that for the "set" method, we need to use it to differentiate between the name of the field and the name of the parameter to the "set" method.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Get/Set Methods, cont'd

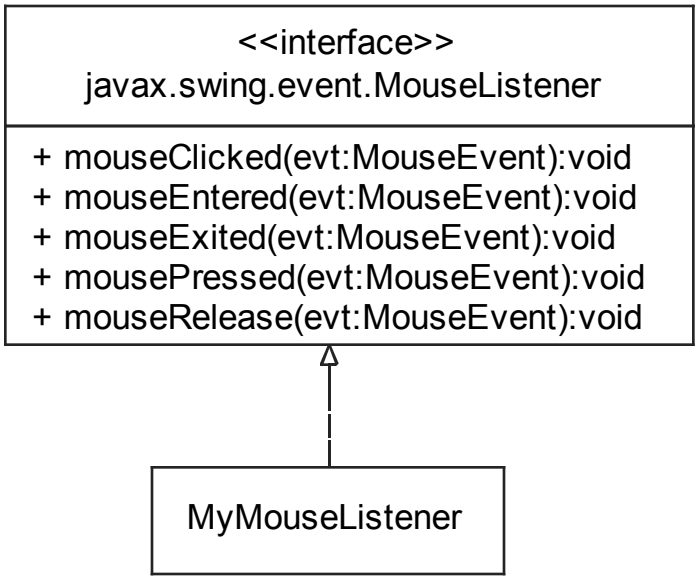
```
1    public class TestAccount
2    {
3      public static void main(String[] args)
4      {
5        Account a1 = new Account();
6        Account a2 = new Account();
7
8        a1.balance = 100;  // ERROR!
9
10       a1.setBalance(100);
11       a2.setBalance(200);
12
13       double d1 = a1.getBalance();
14       double d2 = a2.getBalance();
15     }
16   }
```


This code demonstrates creating a couple of Account objects and then exercising the get/set methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Interfaces

- An **interface** is a group of methods with no implementation
- Classes **implement** interfaces by providing the methods defined in the interface



The Java Runtime Environment defines many interfaces that define a set of behavior that a class can provide. Here we show the MouseListener interface, which specifies the methods that a Java GUI component calls when the mouse interacts with the component.

You can also create your own interfaces to define sets of behavior.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Write a Java class named Policy that represents an insurance policy. The class should maintain state for the policy holder's name, the policy amount and whether the holder is a smoker. The class should define a behavior named **rate** that returns true if the holder is a non-smoker and the amount is less than 100000 and false otherwise.

- Write another class named TestPolicy with a *main* method that creates an instance of the above class and prints the object's "rating" to the console.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The basics of object orientation
- About the three pillars: encapsulation, inheritance and polymorphism

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Methods

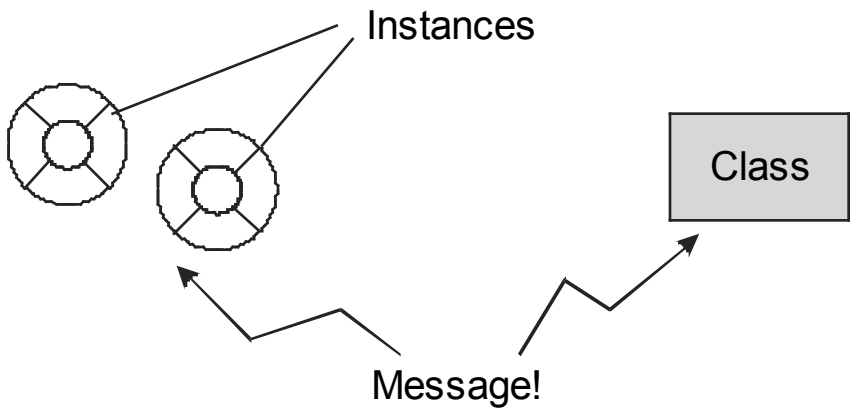
- Instance Methods
- Constructors
- Static Methods

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a Method?

- Methods represent behaviors in an object or a class
- **Instance** methods run as part of an individual object, while **static** methods run as part of the class
- Calling a method is sometimes referred to as "sending a message"



Recall that an object consists of state and behavior -- in Java, we write methods to provide behaviors. In this chapter, we will start by looking at instance methods and later cover static methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling a Method

- To call an **instance** method, you must first create a object, then use the object reference to invoke the method
- To call a **static** method, you invoke the method on the class itself

General Form

```
type returnVal = receiver.method(arg1, arg2)
```

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5         java.util.Random rand = new Random();
6         int i = rand.nextInt();   // instance method
7
8         double result =
9           Math.sqrt(8.33);  // static method
10     }
11   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Method Receivers

- Method invocations must **always** be directed to a **receiver**, which is either an object (for instance methods) or a class (for static methods)
- In some cases, the receiver can be implicit

```
1    public class MyClass
2    {
3      public void myMethod()
4      {
5         anotherMethod();
6         this.anotherMethod();
7      }
8      public void anotherMethod()
9      {
10     }
11   }
```

If a method in a class needs to call another instance method in the same class, you can use the "this" keyword to explicitly specify the receiver.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

How 'this' Works

- Whenever you call an instance method, the JVM defines a special local variable named **this**
- The method can use **this** to access fields and call methods in the same object (i.e. "self message")

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5         MyClass m1 = new MyClass();
6         m1.myMethod();    // within myMethod, this = m1
7
8         MyClass m2 = new MyClass();
9         m2.myMethod();    // within myMethod, this = m2
10     }
11   }
```

During the execution of an instance method, "this" is an alias to the object that the caller created.

In most cases, you don't have to explicitly specify "this" since the compiler implicitly provides it. However, if an instance method has a parameter (or local variable) whose name matches a field, then you must use "this" to access the field:

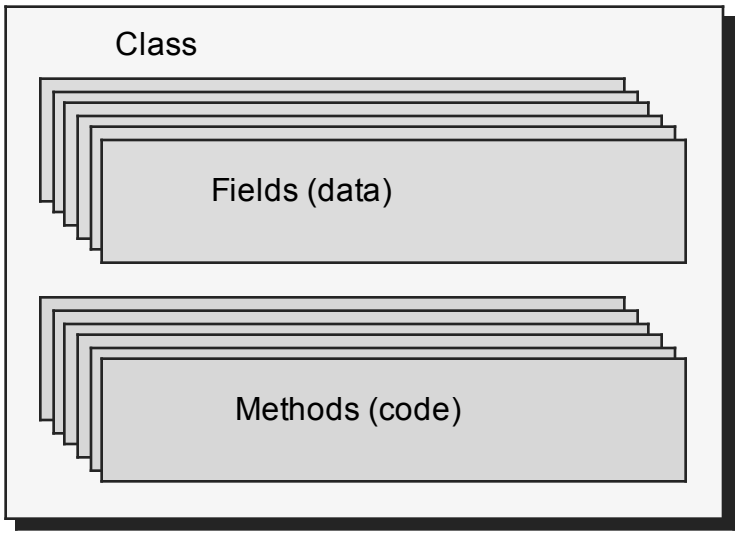
```
public class SomeClass {
  private int x;

  public void setX(int x) {
     this.x = x;
  }
}
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Class Structure

- In Java, all code must be contained within a method (a.k.a function or subroutine)
- In turn, methods must reside within a class



The "class" is the basic building block of Java -- each class typically defines methods (behaviors) and fields (state). In this chapter, we'll focus on the methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Method Syntax

- To define a method, you must specify the method's return type, name and arguments (if any)
- If the method has no return value, use *void*
- If the method has no arguments, you still must use parenthesis

```
[options] <return-type>
   methodName ( [arg-type argName]* )
```


By convention, Java method names begin with a lower-case first character.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Instance Method Examples

```
1    public class Methods
2    {
3      public void method1()
4      {
5        // statements
6      }
7      public int method2 ( int x )
8      {
9        return x * x * x;
10     }
11     public double method3(int x, int y)
12     {
13       return Math.PI * Math.PI + x + y;
14     }
15   }
```


Here we show a class with three methods. The first method has no arguments and no return value. The second accepts a single integer and returns an integer. The third accepts two integers and returns a double.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling Instance Methods

- Unless a method is defined as *static*, you must create an object before calling the method
- These non-static, or *instance* methods run in the context of the object in which they reside

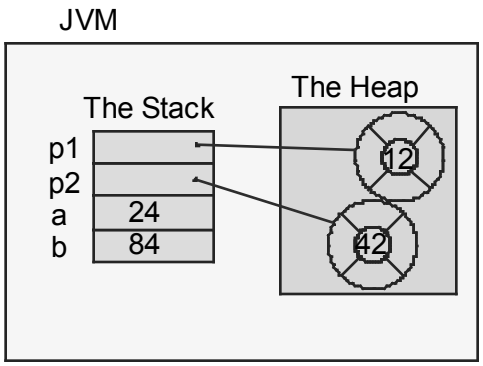
```
1    public class Person
2    {
3      private int age = 14;
4
5      public void setAge(int a)
6      {
7        age = a;
8      }
9      public int method1 (int z)
10     {
11       return age * z;
12     }
13   }
```

If you omit the "static" option when you define a method, the method is considered to be an "instance" method, which means it runs as part of an individual object. Instance methods typically operate on the object's state.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling Methods, cont'd

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        Person p1 = new Person();
6        Person p2 = new Person();
7        p1.setAge(12);
8        p2.setAge(42);
9        int a = p1.method1(2);
10       int b = p2.method1(2);
11   }
```



Here we show a program that uses the class defined on the last page. It creates two objects and then calls instance methods on each. Note how the methods operate on the each individual object's data.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Given the *Methods* class shown on page 8, write a program fragment that creates an instance of the *Methods* class and calls each of its methods

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice 2

- Look up the *java.util.Random* class in the JavaDocs and write a program fragment that creates an instance of the *Random* class and calls its *setSeed* and *nextLong* methods.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overloading Methods

- You can define multiple methods with the same name as long as each has a different *signature*
- This technique is referred to as *overloading*

```
1    public class Account
2    {
3      public void withdraw ( int amount )
4      {
5        // statements
6      }
7      public void withdraw ( double amount )
8      {
9        // statements
10     }
11   }
```

Overloaded methods with a class must have different signatures. Note that when determining a method signature, Java doesn't use the return value -- in other words, the number and/or types of the parameters must be different.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling Overloading Methods

- When a program calls an overloaded method, the compiler looks for a matching signature

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        Account a = new Account();
6
7        a.withdraw ( 20 );
8        a.withdraw ( 55.55 );
9      }
10   }
```


If the compiler doesn't find a method with the correct name and signature, you get a compile error!

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Constructor Methods

- Constructor methods are special methods that participate in the construction of objects
- Constructor methods must have the same name as the class and have no return value

```
1    public class Account
2    {
3       private double balance;
4
5       public Account()
6       {
7         // statements
8       }
9    }
```


The job of a constructor is to initialize the object so it's ready for use. The Java virtual machine calls constructors during the execution of the "new" operator.

Normally, constructors are public.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The new Operator

- The *new* operator does the following:
  - Allocates storage from the heap
  - Sweeps the storage to value zero
  - Runs field initializers
  - Calls a constructor
  - Returns the reference



Constructors run as part of the "new" operator.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overloaded Constructors

- A class can define multiple constructors, as long as each has a different signature

```
1    public class Account
2    {
3      private int accountID;
4      private double balance;
5
6      public Account ( int id )
7      {
8         accountID = id;
9         balance = 0;
10     }
11     public Account ( int id, double bal )
12     {
13        accountID = id;
14        balance = bal;
15     }
16   }
```


Overloaded constructor methods have the same signature uniqueness rules as non-constructor methods.

Often, we use overloaded constructors to provide different ways to initialize new objects.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling Constructors

- Calling programs never directly invoke a constructor
- Instead, the Java virtual machine calls a constructor during the *new* operator

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        Account a1 = new Account ( 12 );
6        Account a2 = new Account ( 14, 45.67 );
7      }
8    }
```


When a calling program uses the "new" operator, Java looks for a constructor with a matching signature. If one does not exist, you get a compile error.

Note that by providing multiple constructors, the class author gives the class user flexibility in how they create objects.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Write a class named **Claim** that represents a claim on a life insurance policy. The class should have state for the date of the claim, the claim number and claim amount. Write two constructors in the class: one that accepts claim number and amount, and another that accepts data corresponding to all of the fields.

- Write another class named **TestClaim** with a *main* method that creates two objects, using both constructors.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The No-Argument Constructor

- If a class defines no constructors, Java provides one that accepts no parameters
- But if the class defines any constructors, Java no longer provides the no-argument constructor

Why will this code not compile?

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        Account a1 = new Account ( 12 );
6        Account a2 = new Account ( 14, 45.67 );
7        Account a3 = new Account();
8      }
9    }
```

Assuming we are using the Account class defined a couple of pages ago, this code will get a compile error in line 7. That's because the Account class doesn't define a no-argument constructor.

The no-argument constructor is sometimes referred to as the "default" constructor since Java provides it if the class has no other constructors.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Calling One Constructor From Another

- To avoid duplicated code, a constructor can use the *this* operator to call another constructor

```
1    public class Account
2    {
3      private int accountID;
4      private double balance;
5
6      public Account()
7      {
8        // complex calculation here
9      }
10     public Account ( int id )
11     {
12       this();
13       accountID = id;
14     }
15   }
```


The "this" keyword refers to the current object. Here we are using it to invoke a constructor method.

Note that any call to another constructor must be the first executable line in the calling constructor.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Field Initializers

- If you wish to initialize fields to values known at compile time, you don't need to write a constructor

```
1    public class Account
2    {
3      private int accountID = 14;
4      private double balance = 100.00;
5      private String uniqueHashCode;
6
7      public Account()
8      {
9        // calculate unique hash code
10     }
11   }
```

If you don't initialize fields, the "new" operator sweeps them to "natural zero", which means zero for numeric types like integers and sets object references to null.

Note that you may still need to write a constructors to initialize fields to values not known at compile time (e.g. data from a database) or if the initialization takes more than a single assignment (e.g. you need to calculate a value).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Method Modifiers

- When you define a method, you can specify optional modifiers that configure the method

`public`
`private`
`protected`
	Access to the method

`final`	Method cannot be overriden by a subclass
`abstract`	Method has no body (must be provided by a subclass)
`static`	Method is part of the class, not part of any individual object

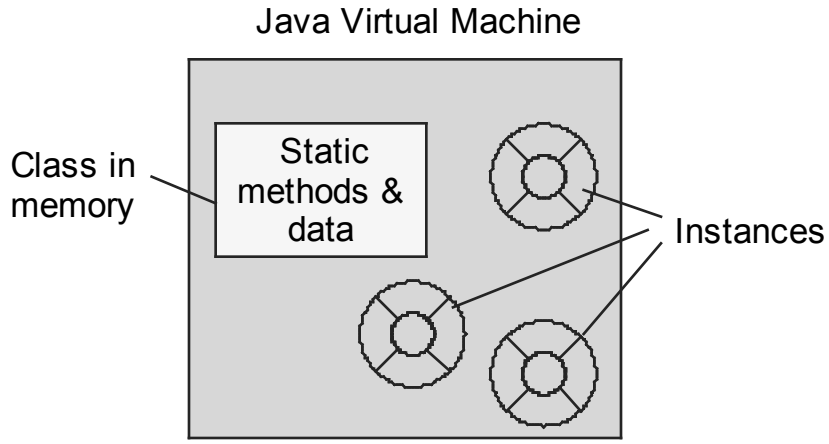
The "public" access modifier means that any code can call the method. The "private" modifier means that only other methods in the same class can call it. The "protected" modifier means that the method is also available to be called from subclasses.

If you don't specify any access modifier, the method has "package" protection, which means it can be called from other classes that are in the same package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Static Methods

- **Static methods** are part of the class in which they are defined, not part of any individual object
- You can call static methods without creating a class instance
- Static methods cannot access non-static class members (fields, methods)



Remember that instances are created by the "new" operator.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Static Methods, cont'd

```
1    public class MyClass
2    {
3      private static int x;
4      private int y;
5      public static void myMethod()
6      {
7         System.out.println (x); //OK
8         System.out.println (y); //ERROR
9      }
10   }
```

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        MyClass.myMethod();
6      }
7    }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implicit Receiver for Static Method

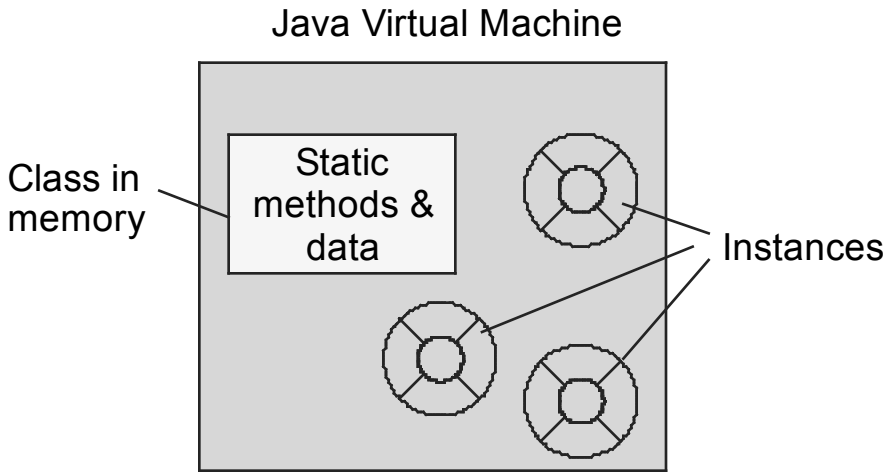
- If a method needs to call a static method in the same class, you can omit the class name as the receiver

```
1    public class TestQuadratic
2    {
3      public static double solveQuadracticPlus(double a,
4         double b, double c)
5      {
6          return (-b + Math.sqrt(b*b-4*a*c))/2*a;
7      }
8      public static void main(String[] args)
9      {
10        double result = solveQuadraticPlus(1,3,-4);
11        result =
12          TestQuadratic.solveQuadraticPlus(1,3,-4);
13     }
14   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Static Fields

- Static **fields** are part of the class in which they are defined, not part of any individual object
- You can access static fields without creating a class instance
- Static fields can be accessed by either static or non-static methods



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Static Fields, cont'd

```
1    public class MyClass
2    {
3      public static int x;
4    }
```

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        System.out.println (MyClass.x);
6      }
7    }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Use the JavaDoc to look up the *java.io.File* class and its *createTempFile* method that accepts two Strings. Write a program fragment that creates a temporary file with prefix="myfile" and a *null* suffix. Then display the temporary file's *path* to the console.

```
1    import java.io.File;
2
3
4    File f =
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The final Keyword

- Final **methods** cannot be overriden in a subclass
- Final **fields** are read-only after construction

```
1    public class MyClass
2    {
3      public static final int count=12;
4
5       public final void myMethod()
6       {
7       }
8    }
```

```
1    public class Test
2    {
3      public static void main(String[] args)
4      {
5        System.out.println (MyClass.count);
6      }
7    }
```


Using static and final together lets you define constants. Note that when you define a final field, you must initialize it as part of the definition or in a constructor.

By convention, constants are written in all uppercase characters.

Also be aware that J2SE 1.5 defines enumerations, which are a more powerful way to create constants.

NOTE: You can also define classes as final -- such classes cannot be subclassed.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- How to define methods
- About constructors and static methods

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Exception Handling

- Traditional Error Handling
- Java Exceptions

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Traditional Error Handling

- Traditionally, programmers have handled potential errors by checking return codes from functions

```
1    public void myMethod()
2    {
3      int errcode=someAPI();
4      if ( errcode != 0 )
5        processError(errcode);
6      else
7      {
8        // do other stuff
9      }
10   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Traditional Error Handling Issues

- Error checking clutters the code
- In a fully debugged program, error checking adds unneeded overhead
- It's difficult to handle errors globally, especially given nested function calls

Is there a better way?

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java Exception Handling

- Instead of explicitly checking for errors, use a *try-catch* block

```
1    public void myMethod()
2    {
3      try
4      {
5        someAPI();
6        // do other stuff
7      }
8      catch ( SomeException exc )
9      {
10       processError(exc);
11     }
12   }
```

If no exception occurs within the "try" block, it executes normally and then skips over the "catch" block. If the "SomeException" occurs, then Java jumps out of the "try" block into the "catch" block -- this is where you can respond to the exceptional condition.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Advantages of Exception Handling

- Code that incurs no exceptions doesn't need overhead of *if* statements checking for errors
- You can write multiple *try-catch* blocks within a single method
- Compiler will force you to handle exceptions if you call methods that are marked as throwing an exception
- One downside is that the syntax is rather verbose

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Uncaught Exceptions

- If the Java virtual machine detects an uncaught exception, it terminates the program and displays a stack trace

```
1    public class Uncaught
2    {
3      public static void main(String[] args)
4      {
5        myMethod();
6      }
7      public static void myMethod()
8      {
9        String s = null;
10       int len = s.length();
11     }
12   }
```

```
C>java Uncaught
java.lang.NullPointerException
  at Uncaught.myMethod(Uncaught.java:10)
  at Uncaught.main(Uncaught.java:5)
Exception in thread "main"
```


Uncaught exceptions are passed back through any nested method calls -- if nobody catches it, the exception finally makes it to the virtual machine.

Here we show a program that attempts to use a null reference, which causes the NullPointerException.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Exception Categories

- Java supports three categories of exceptional conditions:
    - **Errors** are conditions that programs often do not catch since they represent severe problems
    - **Unchecked** exceptions are conditions that programs can handle but the compiler doesn't force them to
    - **Checked** exceptions are those that the compiler will force programs to handle

Errors are conditions that you typically don't handle in your program. Examples include VirtualMachineError, which isn't worth catching, since it means there's a severe problem with the JVM itself.

The compiler forces you to handle checked exceptions - if you don't your code will not compile, let alone run.

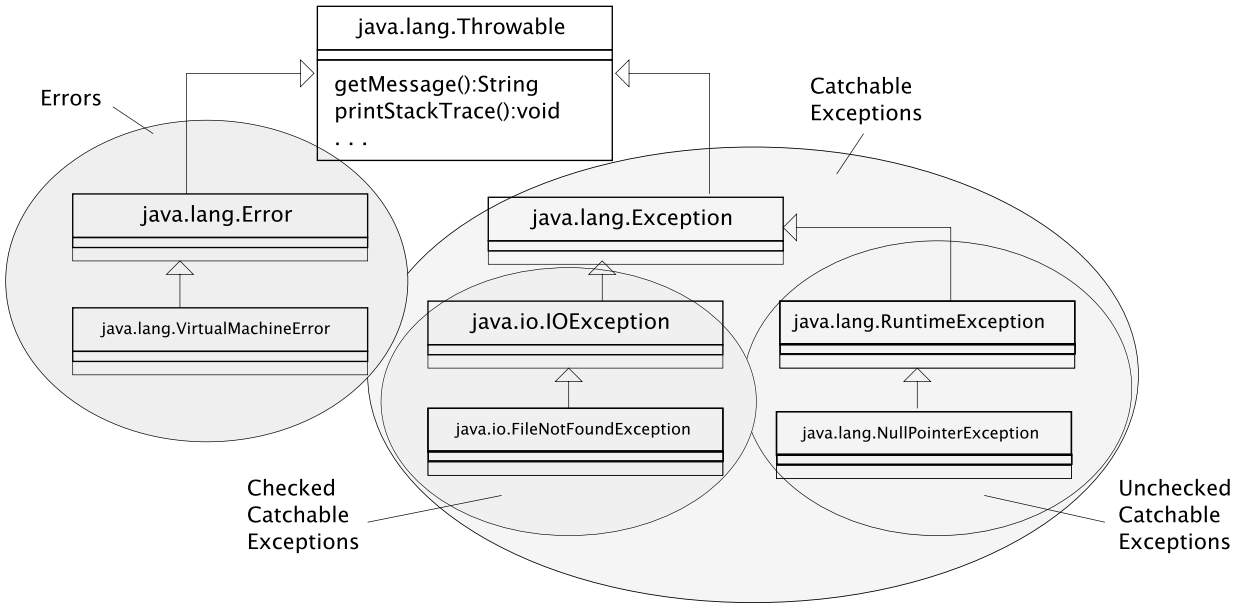
While the compiler doesn't force you handle unchecked exceptions, you can if you wish. NullPointerExcetion is an example that we saw on an earlier page.

In general, checked represent "recoverable" situations, while unchecked are for unrecoverable programming errors, but this is not a hard-and-fast rule.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Exceptions Are Objects

- **Errors** are subclasses of java.lang.Error, while **Unchecked exceptions** subclass java.lang. RuntimeException



Like all objects, exceptions are members of a class. In addition, exception classes are releated by inheritance. java.lang.Throwable is the root of all exception classes.

Here we show some of the exception classes in the Java class library -- there are many, many more and you can create your own.

When you are reading the JavaDoc for an exception, you can whether it's checked or unchecked by looking for java.lang.RuntimeException in its inheritance hiearchy. If it's there, then the exception is unchecked.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Multiple Catch Blocks

```
1    try
2    {
3      FileReader fr =
4         new FileReader ( "/temp/test.txt" );
5      int ch = fr.read();
6      while ( ch != -1 )
7      {
8        System.out.print ( (char)ch );
9        ch = fr.read();
10     }
11     fr.close();
12   }
13   catch (FileNotFoundException e)
14   {
15     System.out.println ( "Unable to open file" );
16   }
17   catch (IOException e)
18   {
19     System.out.println ( "Error reading file" );
20   }
```

This code demonstrates using code in a "try" block that throws more than type of exception. The statement in line 3 throws the FileNotFoundException, while the code in lines 4, 8 and 10 can throw the generic IOException.

Note that in Java 7 and later, you can also write "multi" catch blocks, using the | (operator):

```
catch (IOException | SQLException ex)
{
  // handle the exception
}
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Handle the Most Specific Exception First

- When an exception occurs, the JVM uses the first matching *catch* block
- Since exception classes are related by inheritance, you must code any catch block before another catch block of its superclass

```
1    try {
2    . . .
3    }
4    catch (FileNotFoundException e) {
5    }
6    catch (IOException e) {
7    }
8    catch (Exception e) {
9    }
```

The code show here demonstrates how to catch specific exceptions before the more generic ones. In this case, the "Exception" catch block acts like a catch-all for exceptions that we will not handle specifically.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Passing Exceptions Back

- Instead of writing a *try-catch* block, a method can pass exceptions back to its own caller
- This lets you decide whether to handle exceptions locally or at a more global level

```
1    public void readFile()
2      throws IOException,FileNotFoundException
3    {
4      FileReader fr =
5        new FileReader ( "/temp/test.txt" );
6      int ch = fr.read();
7      while ( ch != -1 )
8      {
9        System.out.print ( (char)ch );
10       ch = fr.read();
11     }
12     fr.close();
13   }
```

A method that calls other methods that are marked as throwing exceptions has two choices on how to handle the exceptions:

1.The calling method can use a try-catch block as shown previously

2.The calling method can pass the exceptions back to its own caller

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The finally Block

```
1    public static void myMethod()
2    {
3      try
4      {
5        // statements
6        return;
7      }
8      catch ( Exception exc )
9      {
10       System.out.println ( exc.getMessage() );
11       return;
12     }
13     finally
14     {
15       // always executes!
16     }
17   }
```

The "finally" block is an optional part of the "try" syntax. If present, it must be after any "catch" blocks.

Even if the code in a "try" or "catch" block returns or throws an exception, the virtual machine guarantees it will execute the statements in the finally block.

This is especially useful in situations where you want to be sure to close a resource (e.g. a database connection).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Throwing Exceptions

- A method can raise, or **throw** one or more exceptions to indicate an exceptional condition
- Since exceptions are objects, we use the **new** keyword, specifying one of the exception class's constructors

```
1    // java.lang.IllegalArgumentException is unchecked
2    public static void myMethod2 (int x)
3    {
4      if ( x < 0 )
5        throw new IllegalArgumentException (
6          "I don't like negatives!" );
7
8      // statements
9    }
```

IllegalArgumentException is an unchecked exception since it's a subclass of java.lang.RuntimeException.

We create a new exception object so that we can throw it and that we pass a message string to the exception class's constructor. That "human-readable" string is stored by the java.lang.Throwable superclass.

Note that when a method throws an exception, that terminates the method's execution and returns control back to the caller.

If a method throws any CHECKED exceptions (i.e. not a subclass of RuntimeException, then it must list those in the "throws" clause of the method definition:

```
public static void myMethod3()
  throws java.io.FileNotFoundException
{
. . .
   throw new FileNotFoundException("Uh oh");
}
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing an Exception Class

- Java applications can define their own exception classes to represent domain-specific exceptions
- Your choice of superclass determines whether the new exception class is checked or unchecked

```
1    public class InsufficientFundsException
2      extends Exception
3    {
4      public InsufficientFundsException (
5         String msg )
6      {
7        super ( msg );
8      }
9    }
```


Here we've defined an exception for a banking application that an application can throw if a withdrawal would result in a negative balance. Since we choose java.lang.Exception as the superclass, this is a "checked" exception.

The "super" keyword calls a constructor in the InsufficientFundsException's superclass (i.e. Exception). Note that writing a constructor in the exception class is optional, but lets us provide a human-readable message for the exception. In addition to the human-readable message, it's a good idea to override the toString() method and provide a "detail" message that gives more information about why the exception occurred.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing an Exception Class, cont'd

```
1    public class CheckingAccount
2    {
3      private double curBalance;
4
5      public void writeCheck ( double amt )
6        throws InsufficientFundsException
7      {
8        if ( curBalance - amt < 0 )
9          throw new InsufficientFundsException (
10               "Out of money" );
11
12       curBalance -= amt;
13     }
14   }
```


Here we show using the exception class defined on the last page. Since the InsufficientFundsException is a "checked" exception, we list it in the method's definition.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Enhanced Catch Block Syntax

- Prior to Java 7, writing a **try** with multiple **catch** blocks that did the same thing was tedious and error prone
- Java 7 lets you use the | (or) operator in **catch** blocks

```
1    try {
2      FileReader fr = new FileReader("test.txt");
3      Socket s = new Socket("oracle.com", 80);
4      . . .
5      s.close();
6      fr.close();
7    }
8    catch (FileNotFoundException | UnknownHostException e) {
9      e.printStackTrace();
10   }
11   catch (IOException e) {
12     e.printStackTrace();
13   }
```


Note that in the "multiple" catch block, the compiler will flag an error if you list exceptions related by inheritance. Thus the separate catch block for IOException, which is the superclass of both FileNotFoundException and UnknownHostException.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Try With Resources

- Prior to Java 7, you could use **finally** blocks to ensure that resources were closed properly
- Java 7 provides the **try-with-resources** syntax to make this cleaner

```
1    try
2     (
3        FileReader fr = new FileReader("test.txt");
4        Socket s = new Socket("oracle.com", 80)
5     )
6    {
7    . . .
8    }
9    catch (Exception e)
10   {
11     e.printStackTrace();
12   }
```

For this to work, the resource class must implement the java.lang.AutoCloseable interface - you will get a compilation error if it doesn't. Many classes that require closing in the standard Java 7 class library do implement this interface.

Note the rather odd syntax - the try clause uses parentheses, and the one or more statements (except for the last) are terminated with the normal semicolon. You can put a semicolon on the last statement, but it's not required.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Use the JavaDocs to look up the *java.net.Socket* class. Examine the constructor that accepts a hostname String and a "port" integer. Then write a program fragment that creates a Socket to connect to *google.com* on port 80. Use try/catch blocks as necessary.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- Exception handling fundamentals
- How to handle exceptions
- Writing your own exception classes

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Arrays and Collections

- Using Arrays
- Using Collections

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is an Array?

- Arrays are simple data structures that store homogeneous data
- You access arrays using an index, which provides fast random access to array elements
- Array indexes start at zero

```
1    int[] intArray = new int[10];
2    double[] doubArray = new double[5];
3
4    intArray[0] = 3;
5    intArray[1] = 77;
6    . . .
7    int x = intArray[1];
8
9    doubArray[0] = 77.77;
10   doubArray[1] = 6444.33;
11   . . .
```

Like objects, arrays must be allocated with the "new" operator before you can use them.

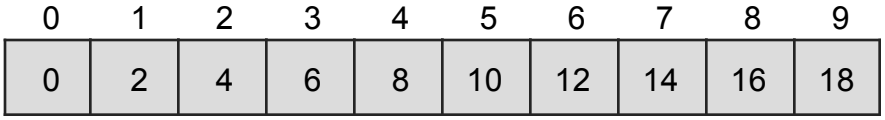
The "new" operator sets all of the array elements to "logical zero" for the type of the array. For example, integers are set to the value zero and arrays of object references are set to "null".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Arrays

- It's common to use a **for** loop to access array elements
- Remember to start the loop at zero!

```
1    public static void main (String[] args)
2    {
3      int[] intArray = new int[10];
4      for ( int i = 0; i < intArray.length; i++ )
5        intArray[i] = i * 2;
6    }
```



Here we demonstrate using a "for" loop to initialize array elements. Note that the loop starts at zero and the ending index is "n - 1", where "n" is the number of elements allocated for the array.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Array Bounds Checking

- The virtual machine checks each array access to ensure that the index is within bounds
- Violations generate the ArrayIndexOutOfBoundsException

```
for ( int i = 1; i <= intArray.length; i++ )
  System.out.print (intArray[i]);
```

```
C>java Arrays
246810121416 18java.lang.ArrayIndexOutOfBoundsException: 10
at introjava.UsingArrays.main(UsingArrays.java:12)
Exception in thread "main"
```

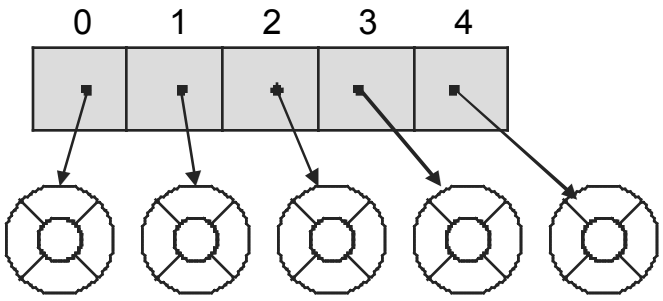

Here we show a common mistake with arrays -- indexes should start at zero and range to n-1 where n is the array's length.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Arrays of Object References

- When you allocate an Array that holds a reference type, the *new* operator sets the array's contents to *null*
- Before use, you must first set the references so they refer to actual objects

```
1    Student[] stArray = new Student[5];
2
3    for ( int i = 0; i < stArray.length; i++ )
4      stArray[i] = new Student();
```



Arrays of object references are quite useful, but you have to be careful to remember that allocating the array itself does not allocate the objects!

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Initializing Arrays

- For types that have literal initializers, you can allocate and initialize an array in one step

```
1    int[] a1 = {45, 55, 77};
2
3    String[] a2 = {"ab", "def", "x" };
```


Defining and initializing an array in one step is a useful shortcut.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Write a program fragment that defines an array to hold 5 doubles. Then write a loop to initialize the doubles to multiples of Math.PI.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Enhanced for Loop

- In Java 5 or later, you can use the **enhanced for** loop to iterate an array

```
1    public static void main (String[] args)
2    {
3      int[] intArray = new int[10];
4      for (int i = 0; i < intArray.length; i++)
5        intArray[i] = i * 2;
6
7      for(int i : intArray)
8      {
9         System.out.println(i);
10     }
11   }
```

You can also use the "enhanced for loop" on so-called generic collections.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Collections

- Arrays are useful, but are difficult to work with if you don't know in advance how many elements the array will hold
- It's also difficult to re-size an array or to insert a new element in the middle of an array
- Java provides *collection classes* that address these issues
- **Note:** Collections store only object references, not primitive values

Most of the collection stuff is in the "java.util" package.

Note that you can store primitive values in a collection, but you need to wrap them up in an object:

```
myCollection.add(new Integer(24));
```

In Java 5 and later, the JVM provides "autoboxing" that creates the wrapper objects for you.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

History of Java Collections

- Java 1.0 shipped with collections such as **java.lang.Vector** and **java.util.Hashtable** - these are known as **legacy** collections are still supported
- Java 1.2 add the Collections Framework including **java.util.ArrayList** and the other collections covered in this chapter
- Java 5 introduced **generic** collections and the **enhanced for loop** for iteration as well as a few additional collection types such as **java.util.concurrent.BlockingQueue**
- Java 7 added a new **diamond syntax** to simplify instantiating collections
- Java 8 added **streaming** collections that support parallel processing and **lambda expressions**

We will not cover the Java 8 streams in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Collections Can Automatically Grow

- As you add elements to a collection, it will automatically resize itself
- Of course, re-sizing takes time!

```
1    java.util.ArrayList<String> myList =
2      new java.util.ArrayList<String> ( 10 );
3
4    for ( int i = 0; i < 1000; i++ )
5      myList.add ( "adding element " + i );
```


Here we show adding String objects to a collection.

You could conceivably write code to do this with arrays, but collections resize automatically!

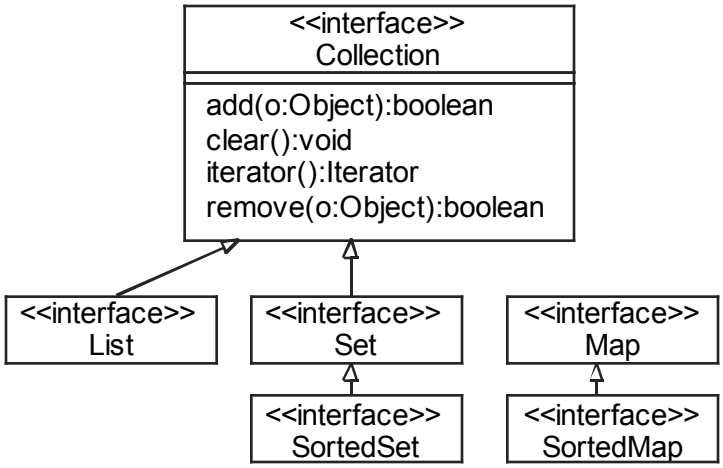
Even though collections do only store object references, it is possible to store primitive values - you can use the so-called "wrapper" classes:

```
ArrayList<Integer> myList = new ArrayList<Integer>();
myList.add (123);
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic Collection Types

- The *Collection* interface is the base type for the collections in Java
- **Lists** are sequential collections that allow duplicate elements
- **Sets** reject duplicates and are optionally sorted
- **Maps** are key/value collections where the key must be unique



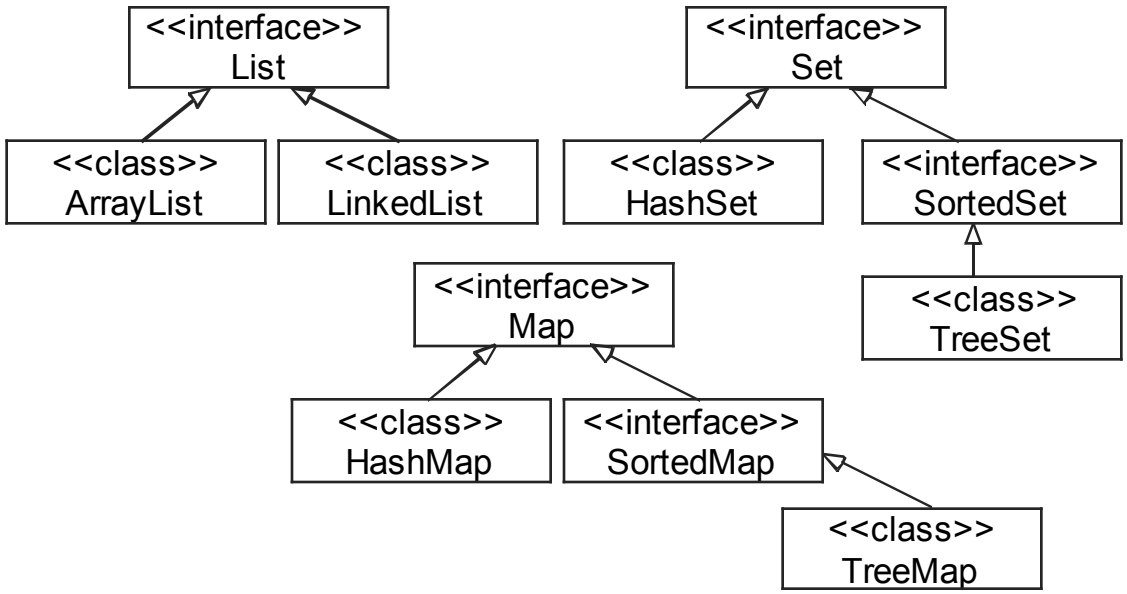
These are all "interfaces", which are like classes but cannot be instantiated. Interfaces define a set of methods that actual classes can implement -- in other words, these are not actual collections, just "prototypes" for the actual collections we will see in a moment.

The Map interface is not a subtype of Collection since maps work somewhat differently than other collections.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Concrete Collection Classes

- Behind the scenes, collections use well-known data structures to store elements

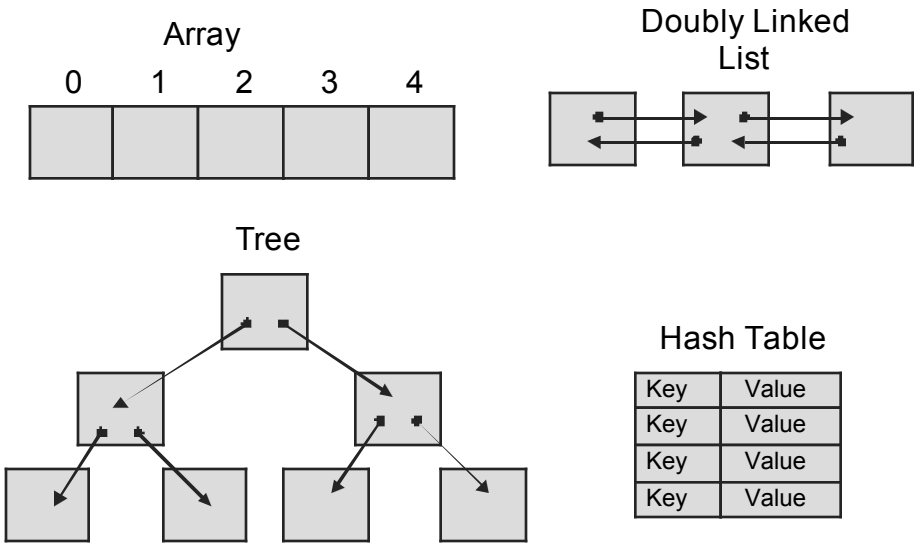


This figure shows a few of the "concrete" collection classes that Java provides; there are others. Note that each concrete class uses a different type of underlying data structure to implement the interface -- for example, an ArrayList uses an array, while a LinkedList uses a doubly-linked list.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Data Structures

- Each type of data structure has different performance, sorting and retrieval characteristics



9 - 14

Array-based collections have fast random access, but it's relatively slow to insert items in the middle, since the collection must move any existing elements back.

Linked lists have fast inserts, but relatively slow random access, since the collection must, on average, navigate through half of the collection to get to a given element (Java collections use doubly-linked lists).

Trees provide automatic sorting and relatively fast inserts and random access.

Hash tables use a key and a value -- you can look up values using the key relatively quickly.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using an ArrayList

- **ArrayLists** provide fast random access of sequentially ordered elements

```
1    ArrayList<String> myList = new ArrayList<>();
2
3    for ( int i = 0; i < 10; i++ )
4      myList.add ( "adding element " + i );
5
6    for(String s : mylist)
7    {
8        System.out.println(s);
9    }
```


In Java 5 and later, we recommend that you specify the type within angle brackets < > as shown here. This is referred to as typing a "generic" collection.

Here we show storin g some String objects in an ArrayList and the so-called "enahanced for loop" to retrieve them.

Note: The Vector class works similarly to ArrayList -- the difference is that Vector is automatically thread-safe, while ArrayList is not.

Java 7 added the so-called "diamond syntax" that we use here to simplify creating a collection object, omitting the type on the right side of the assignment. In older Java, you needed to repeat yourself:

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

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using an ArrayList, Pre Java 5

- Prior to Java 5, programs used **iterators** to visit a collection's elements

```
1    ArrayList myList = new ArrayList();
2
3    for ( int i = 0; i < 10; i++ )
4      myList.add ( "adding element " + i );
5
6    Iterator iter = myList.iterator();
7    while ( iter.hasNext() )
8    {
9      String s = (String)iter.next();
10     System.out.println ( s );
11   }
```


Prior to Java 5, Java didn't support generic collections and the enhanced "for" loop.

An iterator object acts like a cursor and keeps track of the current element as you loop.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a LinkedList

- **LinkedLists** provide relatively fast inserts anywhere in the collection of sequentially ordered elements

```
1    LinkedList<String> linkList = new LinkedList<>();
2
3    for ( int i = 0; i < 10; i++ )
4      linkList.add ( "adding element " + i );
5
6    for(String s : linkList)
7    {
8       System.out.println(s);
9    }
```


This code is almost the same as on the last page, but the performance is a little different. LinkedLists are faster than ArrayLists for inserts in the middle of the collection, but slower for random access.

So if your application needs fast random access and only adds elements at the end, ArrayList is a better choice than LinkedList. But if your program constantly needs to find a particular element, and then add a new element after it, LinkedList will be faster than ArrayList.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

List Performance

- All lists maintain order and allow duplicates, but they differ in performance for different operations:

```
1    ArrayList<Student> arrList = new ArrayList<>();
2    LinkedList<Student> lnkList = new LinkedList<>();
3
4    // add a large number of Student object refs to each
5    // (code not shown)
6
7    Student s;
8
9    s = arrList.get(0);      // very fast
10   s = arrList.get(10000);  // just as fast
11   s = lnkList.get(0);      // very fast
12   s = lnkList.get(10000);  // much slower
13   arrList.add(0, s);       // pretty slow
14   lnkList.add(0, s);       // very fast
```


Array-based collections provide constant-time random access to elements, regardless of where the element is in the collection.

Linked-list-based collections have to start at the beginning (or end) to get to elements in the middle. So for linked-list collections, the average access time for random elements is linear (based on the number of elements in the collection).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Perfomance Quiz

- Which block of code executes faster?

```
1    LinkedList<Student> lnkList = new LinkedList<>();
2
3    // add a large number of Student obj refs
4    // code not shown
5
6    // Technique #1: Using enhanced "for" loop
7    for(Student s : lnkList) {
8       System.out.println(s);
9    }
10
11   // Technique #2: Using "get()" method
12   for(int i = 0; i < lnkList.size(); i++) {
13      System.out.println(lnkList.get(i));
14   }
```

Behind the scenes, the enhanced for loop uses an "iterator" retrieved from the collection.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a HashSet

- **HashSets**, like all Sets, rejects duplicates

```
1    String[] beatles
2      = {"Paul","George","Ringo","John","Paul" };
3
4    HashSet<String> mySet = new HashSet<>();
5    for ( int i = 0; i < beatles.length; i++ )
6    {
7      boolean added = mySet.add ( beatles[i] );
8      System.out.println ( "Added? " + added );
9    }
10
11   for(String s : mySet)
12   {
13     System.out.println(s);
14   }
```


Here we show attempting to add five strings to a HashSet, but since the last string is the same as the first, the last add will fail.

Note that to retrieve items from the HashSet, we can use the by-now familiar iterator.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a HashMap

- **HashMaps** provide keyed storage where the key values are unique

```
1    HashMap<String, String> myMap = new HashMap<>();
2
3    myMap.put ( "John", "555-1111" );
4    myMap.put ( "Paul", "555-2222" );
5
6    String paulsNumber = myMap.get ("Paul");
7    System.out.println (paulsNumber);
8
9    Set<String> keys = myMap.keySet();
10
11   for(String key : keys)
12   {
13       System.out.println(myMap.get(key));
14   }
```

All Map-type collections store pairs of objects -- a key and a associated value. The utility of this is that you can thus associate a key with a value and quickly look up the value, given the key.

Like Sets, Maps reject duplicates; but with Maps, it's the key that must be unique.

Note that we can use a for loop to traverse through a Map, but we first must retrieve the collection of all of the keys.

Also note in Java 7, the "diamond" syntax simplifies creating HashMaps:

```
HashMap<String, String> myMap = new HashMap<>();
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quick Practice

- Look in the JavaDocs for the *java.net.URL* class and note that it defines a constructor that takes a String url (e.g. "http://www.google.com"). Write a program fragment that creates an ArrayList and adds a few URL objects that reference your favorite Web sites. Then use a *for* loop that retrieves each URL object and prints it to the console.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- Array and collection fundamentals
- How to code arrays
- The differences between various types of collections

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JDBC Programming

- JDBC Fundamentals
- Connection Pooling
- Querying a ResultSet
- Prepared Statements

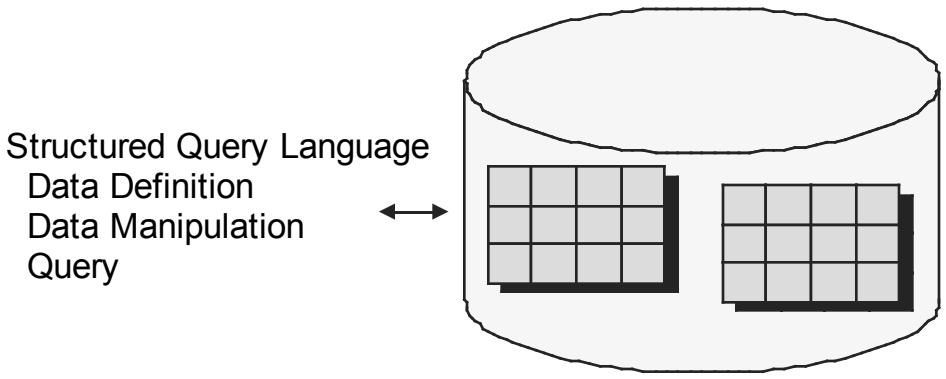
In this chapter, you will learn how servlets can use JDBC to access data stored in relational databases.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Database Fundamentals

- This chapter focuses on relational databases, which are composed of tables that can be joined together
- The standard "language" for relational database manipulation is SQL



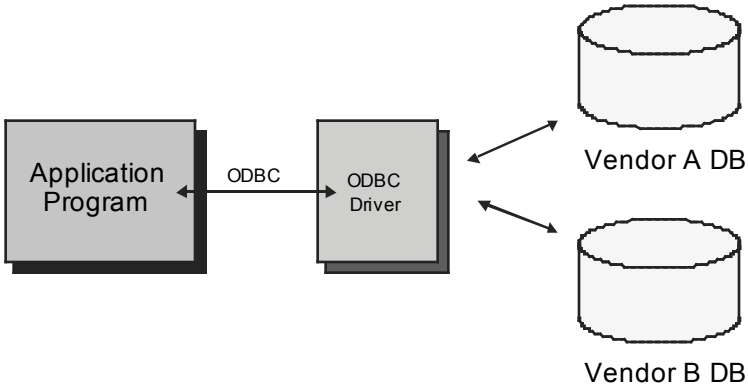
A relational database consists tables of information that you can join or relate together to query information. The concept of relational databases was invented by Edgar Codd at IBM San Jose in the early 1970s and popularized by Oracle.

Starting in the early 1990s, the ANSI and ISO committees jointly worked to define SQL as a standard language for manipulating relational data. Most modern relational databases adhere to the SQL standard.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to ODBC

- Most database vendors define a proprietary API for accessing the database
- Microsoft created the ODBC API to insulate application programs from vendor-specific API
- ODBC is a C-based API and uses pointers heavily (not suitable for Java)



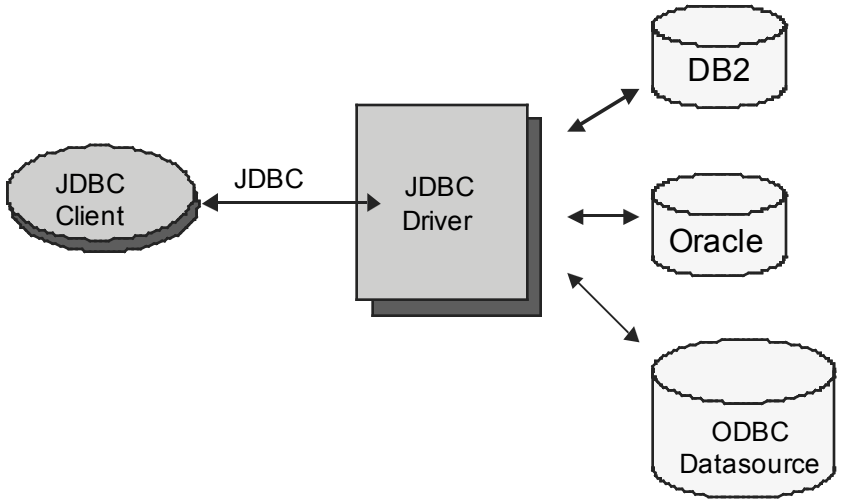
ODBC was developed by Microsoft in the early 1990s to provide a database-neutral programming interface for Windows applications. ODBC insulates applications from database specifics using ODBC drivers, which are generally written by the database vendor. In other words, the program talks to the driver, not the database -- thus the program should be portable across databases.

The problem with ODBC for Java is that ODBC was originally written for the C programming language and uses C language features that don't map well to Java.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to JDBC

- Like ODBC, JDBC lets you connect to relational databases using a vendor-independent API
- Each database vendor must provide a JDBC driver



JDBC is similar in concept to ODBC -- it provides a database-neutral API so that Java programs can access relation data. Like ODBC, JDBC has the notion of drivers that insulate the application from the database. In other words, your JDBC program, referred to as the JDBC client), uses the API provided by the driver.

Note that even though the program using JDBC is referred to as a "client", it doesn't necessarily have to run on a client computer -- a JDBC client is any program that uses the JDBC API. In fact, in this course, our JDBC clients will be servlets.

Note also that if you need to access a database that has an ODBC driver but no native JDBC driver, you can use the JDBC-ODBC bridge driver supplied by Sun. You should avoid the bridge if possible, since it's not known for fast performance.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What Can You Do with JDBC?

- JDBC supports the normal activities you'd expect from a database program, for example:
  - Creating records, retrieving records (queries), updating records, deleting records. These are sometimes referred to as CRUD
  - Accessing database metadata (e.g. table names, table column names)
  - Calling stored procedures
- JDBC is not really object-oriented, however

CRUD stands for Create Retrieve Update and Delete.

JDBC lets you access the raw data stored in a database, but it does not let you treat database rows directly as objects. For example, if you have a Student database table, JDBC lets you read and write the Student information from the table, but it's your job, if you desire, to define and create Student objects</i> from the rows in the table.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

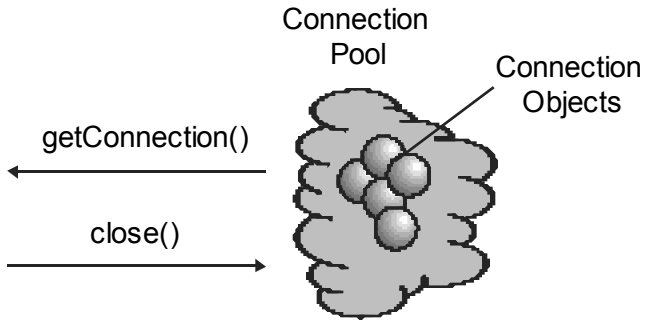
JDBC History

- The JDBC 1.0 API was originally introduced in JDK 1.1 but is now a core technology of Java2 Enterprise Edition
- JDBC 2.0 API adds support for *connection pools* (in the so-called Optional Package), performance enhancements and additional APIs
- JDBC 3.0 adds support for non-relational data sources (e.g. text files)

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Connection Pools

- To avoid expensive creating and closing connections, some JDBC drivers implement the JDBC 2.0 connection pooling feature
- This is especially helpful in Web applications since there's a lot of short-lived access to databases



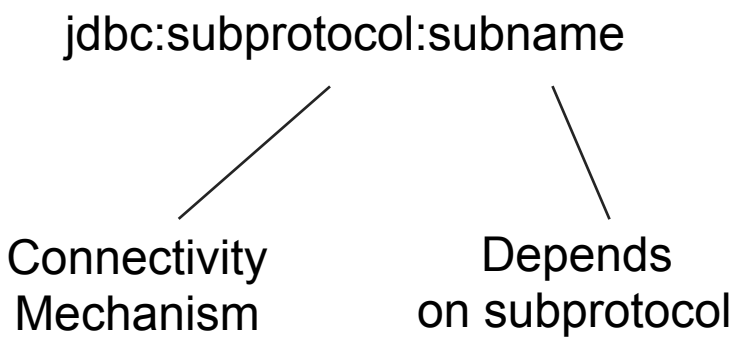
Connection pooling is almost a requirement for Web applications. To support pooling from a servlet, you need both a compliant JDBC driver and a container that supports pooling.

The way pooling works is that the container pre-creates a number of connections at startup. When a servlet needs a connection, the container retrieves a connection from a pool. When the servlet is finished using the connection, the close method does not destroy the connection, but simply returns it to the pool.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JDBC URLs

- JDBC uses URLs to represent data sources
- Much of the URL syntax is vendor-specific
- Examples:
  - jdbc:db2:MyDb
  - jdbc:hsqldb:hsql://localhost



To reference a data source, JDBC programs specify a URL, similar to the URL for a web page. The URL contains three parts: the protocol, which is "jdbc:", the sub-protocol and the sub-name. The last two parts depend on which JDBC driver you are using. The sub-protocol essentially identifies the driver, while the sub-name locates the data source itself.

This page shows two URLs for different databases. The first is for IBM's DB2 -- the sub-name "MyDb" is the name of a DB2 database. The second example is for the free HSQLDB database. Sub-names for that database reference a URL for the database server.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

SQL Exceptions

- Most JDBC operations throw the SQLException if a database error occurs
- In JDBC 2.x, the JDNI API may also throw the NamingException

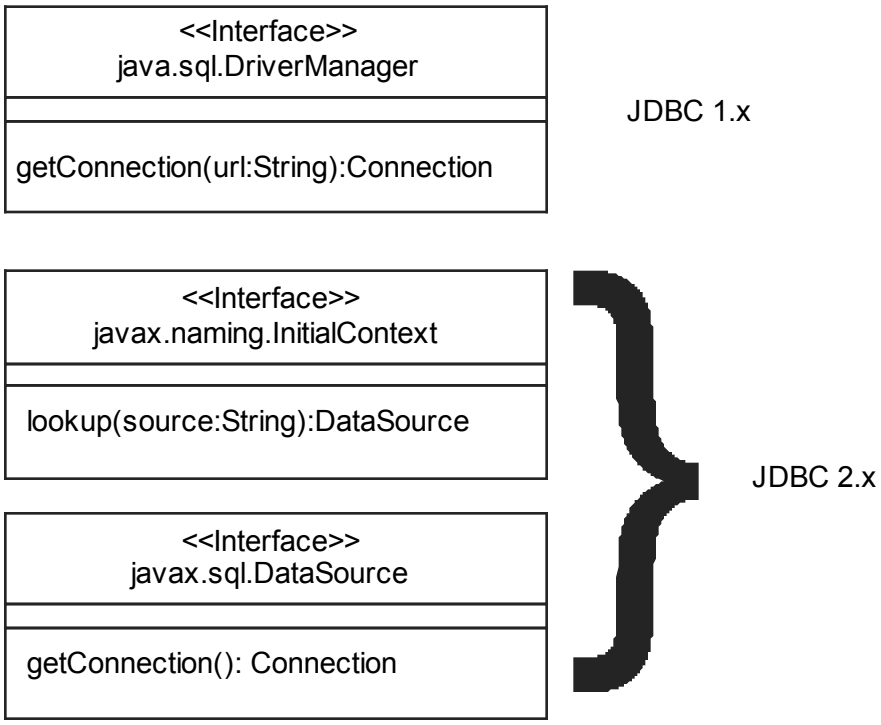
```
try
{
  // some JDBC call(s)
}
catch ( java.sql.SQLException exc )
{

}
```


Whenever you work with JDBC, your program needs to be prepared for exceptions. This page shows some sample code that executes some JDBC statements and catches the SQLException.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Data Sources



There are two ways to connect to a data source in JDBC depending on whether your driver (and container) support the JDBC 2.0 Optional Package.

The JDBC 1.0 uses the DriverManager.getConnection static method to retrieve a connection. This method accepts the JDBC URL and returns an object of type Connection which you can use to access the data source.

Before using the JDBC 2.0 Optional Package API, an administrator must create a Data Source definition using tools provided by the application server vendor. The Data Source definition includes the URL for the data source. The administrator also assigns a name to the Data Source and registers the name in the Java Naming and Directory Interface (JNDI) namespace. Then, a JDBC client can look up the Data Source using JNDI and retrieve a Connection object for the data source.

Note that regardless of which technique you use, you end up with a Connection object with which you can access the relational data.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Opening a Connection, JDBC 1.x

```
1    Connection con = null;
2    try
3    {
4      Class c =
5         Class.forName ("org.hsql.jdbcDriver");
6      con =
7         DriverManager.getConnection (
8             "jdbc:HypersonicSQL:c:\\test" );
9    }
10   catch ( ClassNotFoundException exc )
11   {
12     System.out.println (
13         "Unable to load driver: "  + exc );
14   }
15   catch (SQLException exc)
16   {
17     System.out.println (
18        "Unable to connect: "  + exc );
19   }
```

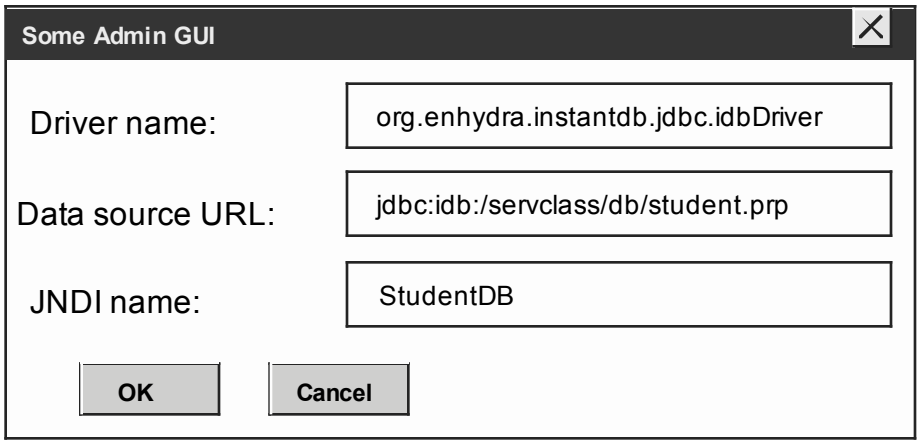

Once you are assured that the JDBC driver's available, then you can call the static "getConnection" method on the DriverManager type, which is part of the JDBC API. You pass this method a URL which names the data source.

If something goes wrong (e.g. the URL syntax is incorrect), the JDBC driver will throw an exception.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Configuring a Datasource, JDBC 2.x

- In JDBC 2.x, an "administrator" must configure each datasource using tools provided by the application server vendor
- The configuration specifies the datasource's Java Naming and Directory Interface (JNDI) name and JDBC URL



Now let's switch gears and talk about using JDBC 2.0 Optional Package. This figure shows a stylized user interface for configuring a Data Source and registering it into the JNDI namespace. Note that we specify the JDBC driver name, here for InstantDB, specify the JDBC URL and assign the JNDI name. The JDBC client will use "StudentDB" to retrieve the Data Source.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Locating a Datasource, JDBC 2.x

- Before you can open a connection in JDBC 2.x, you must first look up the datasource using the JNDI name

```
1    DataSource ds = null;
2    try
3    {
4        InitialContext ctx = new InitialContext();
5        ds = (DataSource)ctx.lookup (
6                "java:comp/env/jdbc/StudentDB" );
7    }
8    catch ( NamingException exc )
9    {
10       System.out.println ( "Bad lookup" +
11                            exc );
12   }
```


This code shows how to use JNDI to retrieve the DataSource defined on the last page. In line 4, we instantiated an InitialContext object, which acts as a gateway to JNDI. In lines 5 and 6, we then use the lookup method to retrieve the DataSource, using the name assigned by the administrator.

Note that the name string contains more than just the text "StudentDB" -- it is prefixed by the "java:comp/env/jdbc/" context string. This prefix string is somewhat dependent on your application server, so consult its documentation for details. The prefix string shown here works for the JRun application server.

The InitialContext type is defined in the javax.naming package.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Opening a Connection, JDBC 2.x

- Once you have retrieved a datasource reference, you can open a connection
- If the datasource implements connection pooling, this may not actually create a new connection

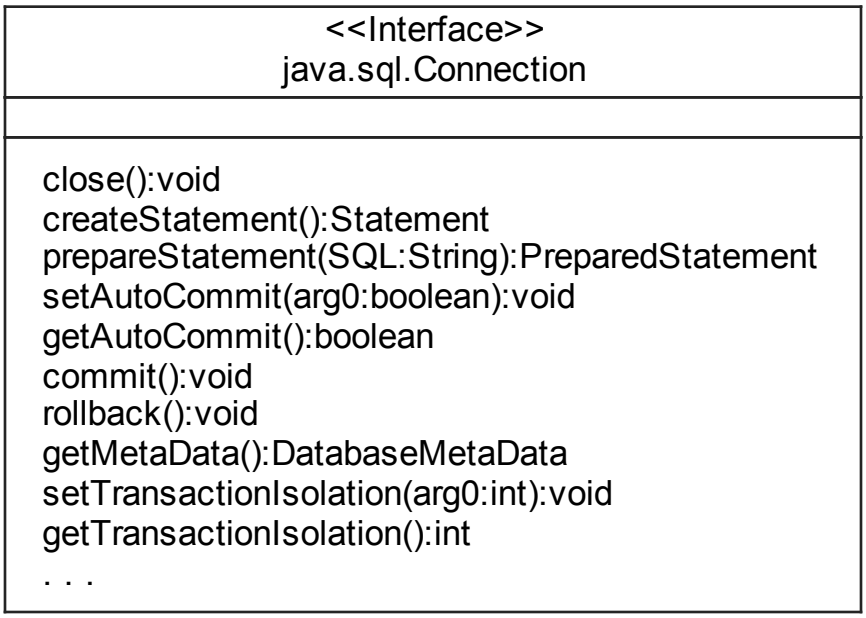
```
1    Connection con = null;
2    try
3    {
4        con = ds.getConnection();
5    }
6    catch ( SQLException exc )
7    {
8        System.out.println ( "Unable to connect"  +
9                             exc );
10   }
```

After retrieving a DataSource, a JDBC 2.0 client can retrieve a Connection object using the DataSource getConnection method as shown in line 4.

Note that there are other overloaded versions of getConnection with additional parameters, such as for database username and password.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Connections



Whether you are using JDBC 1.0 or JDBC 2.0's Optional Package, you end up with a Connection object. This diagram shows some of the more useful methods from the Connection interface.

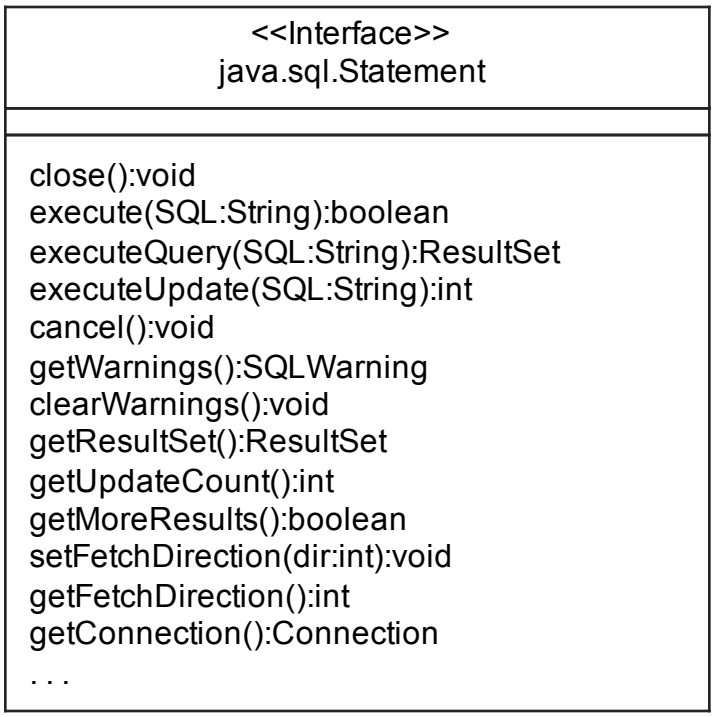
You should be sure to close a Connection when you are finished with it to prevent memory leaks. Recall that if you are using connection pooling, the close method returns the object to the pool instead of destroying it.

One particularly useful method is createStatement, which returns an object of type Statement. As you will see on the next few pages, you can use the Statement object to perform database operations, such as queries and updates. The PreparedStatement type is similar to Statement, but may have performance benefits. We will cover it later.

Also note that Conneciton type defines several methods that work with transactions. We will cover transactions in more detail later.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JDBC API for Statements



The Statement type provides several methods for manipulating a database. For example, executeQuery accepts an SQL query string and returns a ResultSet object that represents the rows matching the query. We will see more about it in a moment.

Another useful method is executeUpdate, which accepts an SQL string specifying a command to modify the database, for example to insert a new row. The executeUpdate method returns an integer that contains the count of rows affected.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Executing a Simple Query

- To execute a query, you can create a Statement and issue the executeQuery method, passing the SQL "select" statement

Student

studentID	name	GPA
12	Bill	3.45
45	Sue	3.22

```
1    Statement stmt = null;
2    ResultSet rs = null;
3
4    try
5    {
6        stmt = con.createStatement();
7        rs = stmt.executeQuery ( "SELECT * FROM student" );
8    }
9    catch ( SQLException exc )
10   {
11       System.out.println ( "Query error: "
12                           + exc );
13   }
```

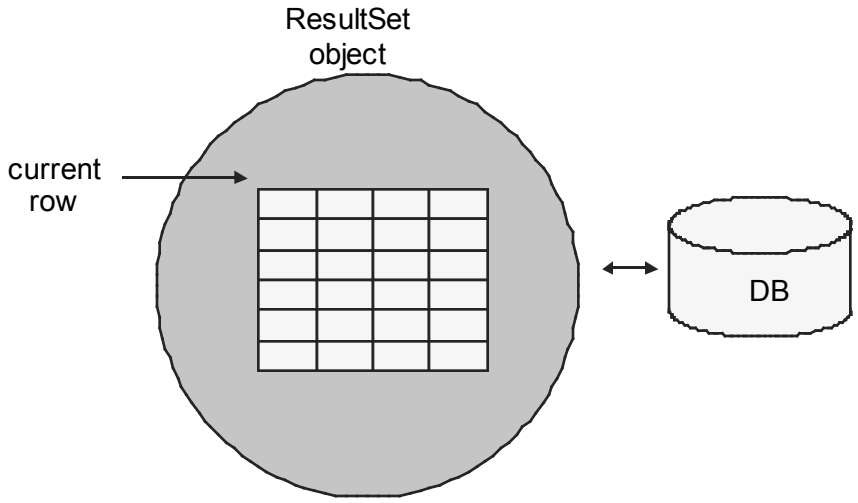

Now let's see some JDBC code to process a simple query. Assume that we've opened a data source containing a table named Student as shown in the figure. The Student table contains three columns of various types.

To query all rows, all columns from the Student table, in line 6, the program creates a Statement from the Connection, and then in line 7 calls executeQuery, passing the SQL string to query all rows, all columns. This method returns a ResultSet object -- we will see how to process it on the next few pages

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing a ResultSet

- The ResultSet acts as a collection of the matching rows -- you can iterate through the results one row at a time

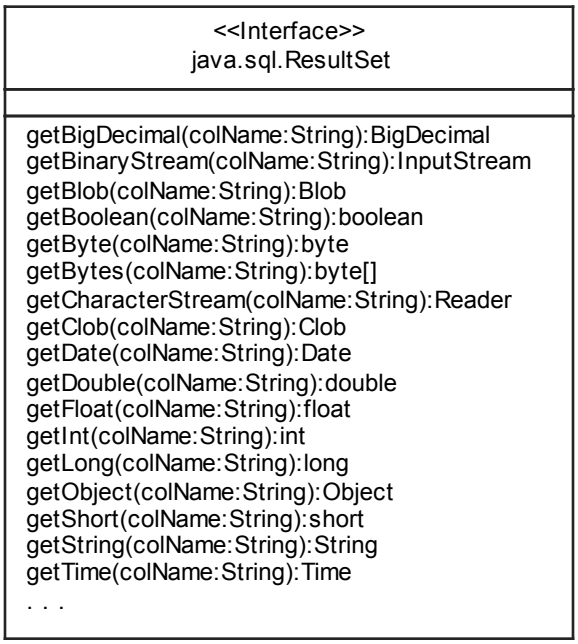


The ResultSet object represents the returned rows from a query. It contains the notion of a "current row" -- when the ResultSet is first created, the current-row pointer points to "above" the first row. To retrieve data, you loop and each time through the loop, you move the current-row pointer to point to the next row. To process a row, you can use APIs to retrieve the data for a given column on the current row.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

SQL Types and Java Types

- Since Java types and SQL types are not identical, JDBC must convert
- You can call getXXX() methods on a ResultSet row and column to force a conversion



Unfortunately, the data types used by Java and by databases are not identical. Therefore, to use data from a database, you will need to convert the data into a Java type. To help, the ResultSet type provides a number of "get" methods that accept a column name or number and convert the data to a particular type. For example, getString ("studentID") would return the studentID column from the current row as a Java String. This would work even though the studentID was stored as some numeric type in the database, for example INTEGER. If the conversion is inappropriate, the "get" method will throw an exception.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing a ResultSet, cont'd

- You can retrieve a column's data either by using the column's name or column index (starting at zero)

```
1    try
2    {
3        while ( rs.next() )
4        {
5          int studentID = rs.getInt ( "studentID" );
6          String studentName = rs.getString ( "name" );
7          float gpa = rs.getFloat ( "GPA" );
8        . . .
9        }
10   }
11   catch ( SQLException exc )
12   {
13       System.out.println ( "Error: " + exc );
14   }
```


Now let's see some code that processes the ResultSet for a query from the Student table shown earlier. In line 3, we start a loop that iterates the ResultSet. We stay in the loop until the next method hits the end of the ResultSet, at which point it returns false.

In lines 5 to 7, we retrieve columnar data from the current row, converting to the Java types of int, String and float. The ellipsis in line 8 represents doing something with the data -- if this code was in a servlet, perhaps we could generate an HTML table row here.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Closing a Connection

- When you are finished with a connection, you should close it to free resources
- If you are using connection pools, calling "close" may not actually close the connection

```
1    Connection con = null;
2    try
3    {
4       con = . . .;
5       . .
6    }
7    catch (SQLException exc)
8    {
9       // report error
10   }
11   finally
12   {
13      if (con != null) con.close();
14   }
```


To avoid tying up the database, you should always close the Statement and Connection when you are finished with them. Note that you don't explicitly need to close a ResultSet -- closing the Statement does that automatically. Also note that if you do another execute method on a Statement, that automatically closes the ResultSet, too.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Inserting a Row

- To insert a new row, create a Statement and issue executeUpdate with the SQL "insert" command as the argument
- The executeUpdate method returns the number of rows inserted

```
1    try
2    {
3        String cmd =
4            "INSERT INTO student VALUES (" +
5            "12, 'Paul', 3.54)";
6
7        Statement stmt = con.createStatement();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```


JDBC is not limited to reading a table. Here we demonstrate using the executeUpdate method to insert a new row into the Student table. Note that constructing the SQL command string is tedious and error-prone due to the fact that you must "escape" quotes for SQL character data. We recommend that you construct the string in a separate String as we did here rather than embedding it directly in the executeUpdate method -- that way, if you're having problems, you can easily write the String to a log so you can debug it.

The executeUpdate command returns the count of rows updated. In this example, it should return the number one to indicate that a single row was inserted.

For simplicity, here we hard-code the values (e.g. 3.54) but in a real program they would probably be in variables. In addition, if this was from a servlet, the data would likely come from an HTML input form.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Updating a Row

- To modify existing row(s), create a Statement and issue executeUpdate with the SQL "update" command as the argument
- The executeUpdate method returns the number of rows updated

```
1    try
2    {
3        String cmd =
4            "UPDATE student SET gpa = 3.87" +
5            " WHERE studentID=12";
6
7        Statement stmt = con.createStatement ();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```


If you have an existing row in a table that you wish to change, you can use JDBC to issue an Update command. In lines 3 to 5, this code defines a String that modifies the "gpa" column of a particular row in the Student table. The "WHERE" clause selects the Student in question.

As with the Insert command shown on the last page, the executeUpdate method returns the number of rows updated, which should be one assuming that "studentID" is the unique primary key of the Student table.

And as on the last page, here we hard-code the values, but a real program would use variables.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Removing a Row

- To get rid of row(s), create a Statement and issue executeUpdate with the SQL "delete" command as the argument
- The executeUpdate method returns the number of rows deleted

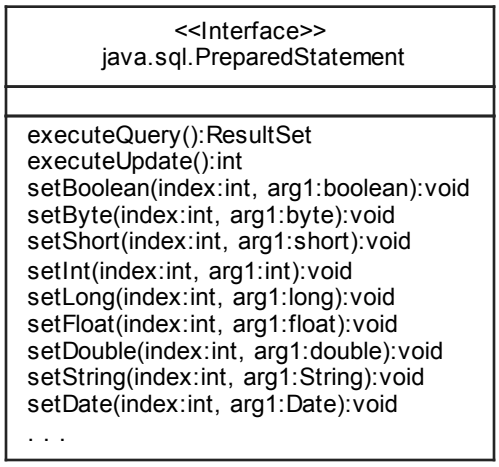
```
1    try
2    {
3        String cmd =
4            "DELETE FROM student" +
5            " WHERE studentID=12";
6
7        Statement stmt = con.createStatement ();
8        int rows = stmt.executeUpdate ( cmd );
9    }
10   . . .
```


This code shows hot to remove a row from a table. Again we hard-code the "WHERE" clause to select the Student in question.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Prepared Statements

- Using prepared statements can increase performance, since they are precompiled
- You typically precompile the statement and then use it multiple times, specifying different parameters each time
- Prepared statements are also less susceptible to **SQL injection** attacks



PreparedStatements are especially useful if you plan to use the same sort of SQL command multiple times. With a PreparedStatement, you write the basic SQL command with placeholders for arguments that you can substitute before you execute the command.

Like most things in JDBC, the performance benefit of PreparedStatements is quite dependent on the database in question and its JDBC driver. But potentially, your database operations can run more efficiently if you use PreparedStatements instead of simple Statements. You should benchmark with your own access patterns to choose which approach provides the best performance for your application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Prepared Statements, cont'd

```
1    int ID = 12; double gpa = 3.87; String name = "Lynn";
2
3    PreparedStatement ps = null;
4    try
5    {
6        String cmd =
7            "UPDATE student SET GPA=?, " +
8            "NAME=? WHERE studentID = ?";
9
10       ps = con.prepareStatement ( cmd );
11       ps.setDouble ( 1, gpa );
12       ps.setString ( 2, name );
13       ps.setInt ( 3, ID );
14
15       int rows = ps.executeUpdate();
16   }
17   . . .
```


This code demonstrates a PreparedStatement. In line 3, we define a reference to a PreparedStatement -- we initialize the reference in line 10, using the String from lines 6 to 8. In that String, we write an SQL "update" command, using the question mark character to indicate a replaceable parameter. Before we actually execute the command, in lines 11 to 12, we call methods on the PreparedStatement to provide the values for the replacement parameters. Note that parameter indexes start at one, not zero!

In a real program, we would not hard-code the parameter values as we did here. If this code was in a servlet, perhaps we would read the parameter values from an HTML form.

Note one other benefit of PreparedStatements as compared to simple Statements -- we don't have to worry about "escaping" quoted character data -- see line 12, which substitutes a String. That makes the code a bit easier to read and maintain.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the architecture of JDBC
- How to use the JDBC API to access relational data sources

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Servlets

- Why Use Servlets?
- The Servlet Container
- The Servlet API
- The Servlet Lifecycle

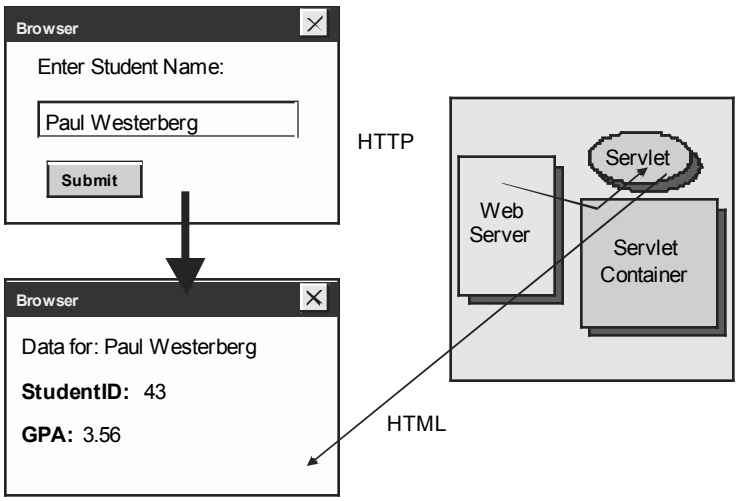
In this chapter, you will learn the basics of the servlet architecture and will examine a simple but complete servlet.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a Servlet?

- A servlet is a Java program that's invoked by a Web server -- servlets typically generate HTML
- Servlets can take the place of traditional CGI programs



11 - 2

This figure shows a typical use of a servlet: responding to an HTML form. When the user submits the form, the Web server passes control to the servlet, which generates an HTML response back to the client.

The picture is much the same as in the last chapter on CGI -- the difference is that we now use a servlet instead of a Perl program. Note also that we refer to the servlet- support runtime as the servlet container</i>.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use Servlets?

- Servlets scale better than CGI programs where the Web server spawns a separate process for each request. In contrast, servlets can stay in memory once they are loaded
- Since servlets are written in Java, they are portable across platforms
- Unlike other proprietary schemes, servlets are widely supported by application servers and thus are portable across vendors

Recall from the last chapter, that some CGI implementations execute a separate process for each request, causing problems when a large number of users access the site at the same time. Servlet applications don't have this problem since the servlet container can keep a servlet in memory for subsequent requests. Thus servlet applications scale well as more users connect to the application.

The key advantage of all Java server-side technologies is that they are portable, both across platforms, and across vendors. For example, say you write a servlet that works with a particular vendor's servlet runtime. Since Java is portable, you should be able run that servlet on any platform that the vendor supports, for example Windows or Linux.

But even more important, you should also be able to change vendors with minimum impact on your servlet code. For example, suppose you initially deploy your servlet under IBM WebSphere, but then decide to use BEA WebLogic instead. Since the servlet specification is vendor independent, you should be able to do that without changing your code.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Containers

- Servlet support is not included with web servers like Apache. To support servlets, you must install a a container for the servlet objects
- There are several popular servlet containers including:
  - Apache Tomcat
  - IBM WebSphere
  - Oracle WebLogic

This page lists some of the popular servlet containers available today. Note that just installing a Web server is not sufficient: if you want to deploy servlets, you also need a servlet container.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Servlet API

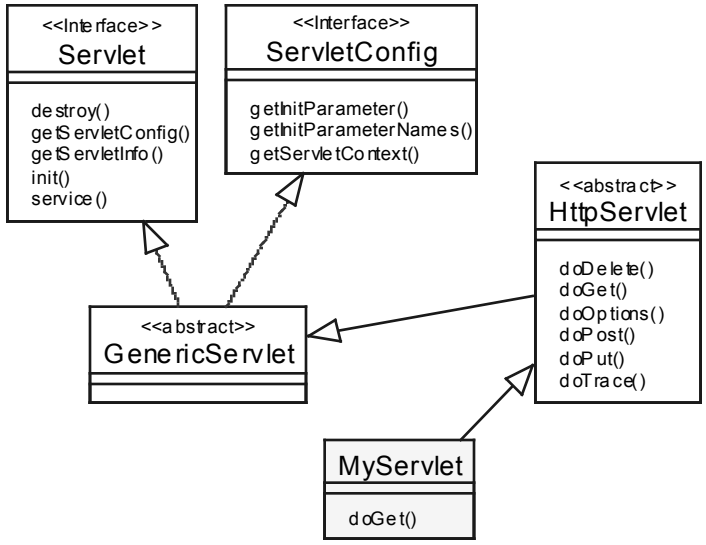
- Sun has defined a set of standard interfaces that you must implement when you write a servlet. The container calls the methods in the interfaces at well defined times
- The container also implements a standard set of interfaces that the servlet can call if required

In the rest of this course, you will learn much of the servlet API. Note that the API falls into two categories: callbacks from the container into the servlet and calls from the servlet to the container. We will examine both categories in this course.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing a Servlet

- You can extend the HttpServlet abstract class to implement your servlet
- We will examine these interfaces in more detail later in the course



This figure is a class diagram that uses Unified Modeling Language (UML) notation to describe the inheritance hierarchy for a typical servlet. This servlet extends HttpServlet and overrides the doGet method so it can respond to a "get" request from a browser.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Sample Servlet

```
1    import java.io.*;
2    import javax.servlet.*;
3    import javax.servlet.http.*;
4    import javax.servlet.annotation.*;
5
6    @WebServlet("/MyServlet")
7    public class MyServlet extends HttpServlet
8    {
9        public void doGet ( HttpServletRequest request
10                           , HttpServletResponse response)
11           throws ServletException, IOException
12       {
13           response.setContentType ( "text/html" );
14           PrintWriter out = response.getWriter();
15           out.println ("<html><body>" );
16           for ( int i=0; i < 5; i++ )
17               out.println ("<p>Hello, <b>world</b></p>" );
18           out.println ("</body></html>" );
19       }
20   }
```

Here is the code for the sample servlet modeled on the last page. In lines 1-4, it imports the java.io package and the standard servlet packages.

In line 6, we use an annotation introduced in Java Enterprise Edition (JEE) Version 6 to assign a URL to the servlet.

In line 7, the servlet extends HttpServlet as shown in the UML diagram on the last page. In line 9, we override doGet, accepting request and response arguments. In line 13, we use the response argument to set the content type. Then, in line 13, we retrieve an output writer to which we can send the HTML response, which we generate in lines 15 to 18.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Lifecycle

- When a client request first references a servlet, the container creates the servlet object and routes the request to it
- The servlet object stays in memory for subsequent requests at the whim of the container
- The container can terminate the servlet at any time - the container's JVM will eventually garbage collect the servlet object

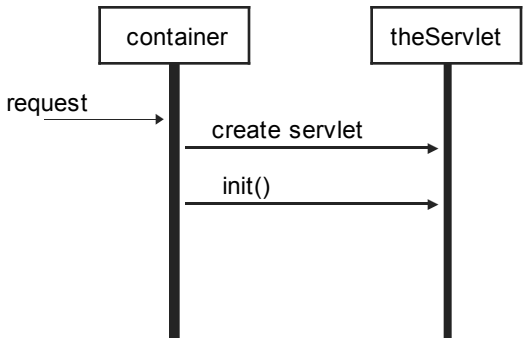
```
1    public class MyServlet extends HttpServlet {
2    . . .
3       public void init() {
4          // your initialization code here
5       }
6       public void destroy() {
7          // your cleanup code here
8       }
9    }
```

At some point, the container will determine that it's time to remove the servlet. Perhaps the container itself is shutting down or perhaps the container's JVM is running low on memory. In any case, eventually, the container will garbage collect the servlet instance.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Initialization

- When a container loads a servlet, it calls the *init* method so that the servlet can perform initialization tasks (e.g. open JDBC connections) or access initialization parameters
- Note that the init method will again be called if the container unloads the servlet and then reloads it

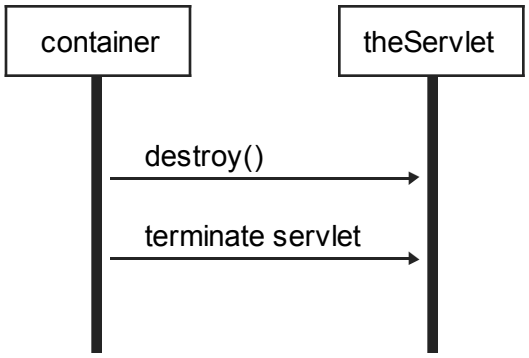


Your guarantee from the container is that the container will always call the init method before dispatching any requests to the servlet. That way, your servlet can safely accomplish its initialization. If your servlet does not need any initialization, you don't need to override this method at all.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Termination

- When the container decides to unload a servlet, it calls the destroy method so that the servlet can clean up (e.g. close JDBC connections)
- The container decides to unload servlets using implementation-specific rules



Like the last page, the guarantee here is that the container will always call the destroy method before disposing of the servlet. That way, your servlet will always be able to clean up after itself.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Processing Requests

- To process client requests, an HTTP servlet usually:
  - Sets the content type
  - Retrieves client arguments (e.g. from HTML forms)
  - Retrieves an output stream
  - Generates output (i.e. the HTML result)

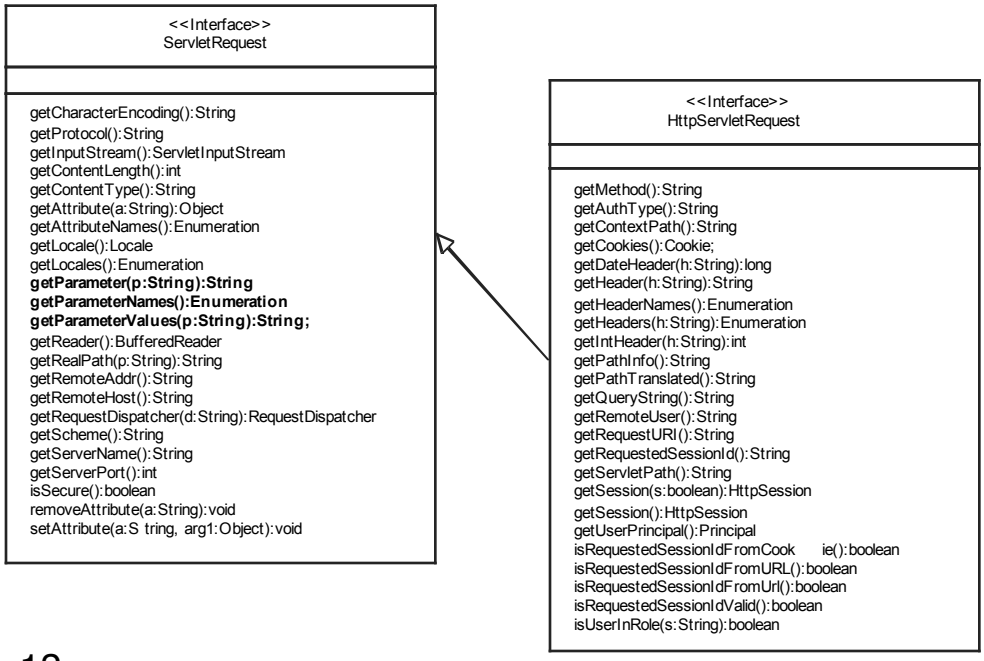
```
public class MyServlet extends HttpServlet
{
  public void doGet ( HttpServletRequest req
                    , HttpServletResponse res )
  {
        . . .
  }
}
```


In this code, the servlet has overridden the doGet method so it will work with "GET" requests from the browser. This page also enumerates the typical action of such a servlet: it sets the content type, retrieves arguments from the request, retrieves a reference to an output stream, and then sends HTML to the output stream. We will look at these steps in more detail now.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The HTTPServletRequest Interface

- The HTTPServletRequest interface represents the request from the client



The doXXX methods, for example doGet, accept two arguments: one corresponding to the request, the other for the response. Here we show the class diagram for the request object (we will see the response object in a moment). Note that the servlet can retrieve a plethora of information about the request. The most commonly used methods include the getParameter calls from ServletRequest and the getHeader calls from HttpServletRequest. We will examine those in more detail soon.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Get versus Post

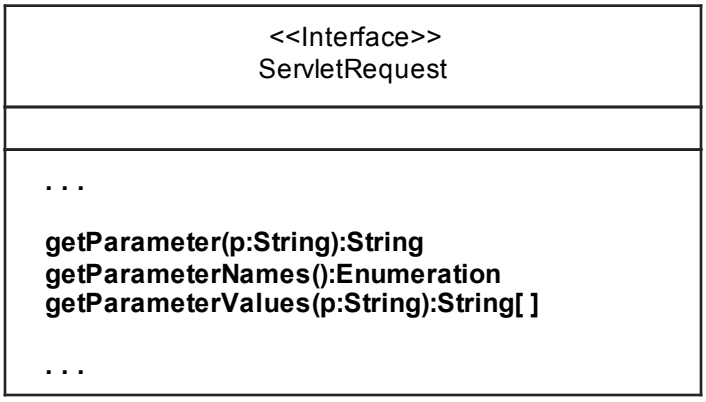
- The CGI standard supports two methods of requests: get and post
- Rule of thumb: use get for requests that read from the server; post for requests that modify data on the server
- The servlet infrastructure makes the choice transparent except that you must override the appropriate doGet or doPost method

In CGI, the choice of "get" versus "post" has a large effect on your program, since CGI passes arguments differently for get and post. The good news is that for servlets, you don't need to worry too much since the servlet infrastructure makes the parameters available to the servlet in the same way, regardless if get or post was used.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Accessing Client Arguments

- You can retrieve named client arguments (e.g. from HTML forms) using the getParameter, getParameterNames and getParameterValues methods



This diagram shows a subset of the UML diagram we saw a couple of pages ago, highlighting the methods that let your servlet retrieve parameters from the request. The getParameter call accepts the name of a parameter and returns the parameter as a String, or null if there is no parameter with that name. getParameterNames returns an Enumeration object that contains a list of all of the parameter names. You could use a loop to iterate through the Enumeration and retrieve all of the parameters. Finally, getParameterValues returns an array -- the array contains the values of a multi-valued parameter name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Accessing Client Arguments, cont'd

From HTML Form:

```
<form action = ...>
...
<input type="text" name="SerialNumber">
...
</form>
```

Servlet code:

```
String serialStr =
  req.getParameter ( "SerialNumber" );
```

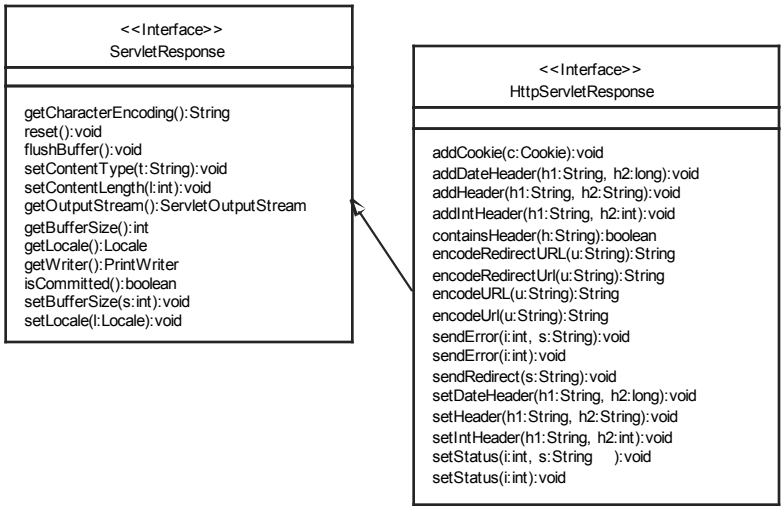
Note: Parameter names are case sensitive!

This page shows an example of how a servlet can retrieve arguments from an HTML form. This form defines an input element which we've named SerialNumber. Then, in the doGet or doPost method, the servlet can retrieve whatever the user entered into the text field by calling getParameter, passing the name of the input element.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The HTTPServletResponse Class

- The HTTPServletResponse class represents the response that a servlet sends back to the client
- You can use this class to set the response's content type, set response HTTP headers or redirect the request



During doGet or doPost, the container passes the servlet an argument of type HttpServlet. On the next few pages, we will see some of the things we can do with this object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Setting the Content Type

- You must set the content type to the MIME type of the response
- Generally, the content type will be HTML
- You must set the content type before retrieving the output stream object

```
1    public void doGet ( HttpServletRequest req
2                      , HttpServletResponse res )
3    {
4        res.setContentType ( "text/html" );
5
6        . . .
7    }
```


Most, but not all servlets generate HTML, so they can use the code in line 4. However, if your servlet generates something else, for example XML or Wireless Markup Language, you should set the content-type appropriately.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Obtaining an Output Stream

- You can use the getWriter method to obtain a PrintWriter for generating text-based content (e.g. HTML)
- You can instead use the getOutputStream method to retrieve a ServletOutputStream object for generating binary content (e.g. a GIF file)

```
1    public void doGet ( HttpServletRequest req
2                      , HttpServletResponse res )
3    {
4        res.setContentType ( "text/html" );
5        PrintWriter out = res.getWriter();
6        . . .
7    }
```


The PrintWriter object retrieved in line 6 is designed for textual output, like HTML or XML. Objects of type ServletOutputStream work properly for binary output.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Generating Output

- The servlet can use the familiar *print* and *println* methods to send content to the output stream
- The container will transmit the content back to the client to satisfy the original request

```
1    public void doGet ( HttpServletRequest req
2                      , HttpServletResponse res )
3    {
4        res.setContentType ( "text/html" );
5        PrintWriter out = res.getWriter();
6
7        out.println ( "<html>Hi!</html>" );
8        . . .
9    }
```


This is the normal activity of a servlet -- generating HTML in response to a request. Here we show outputting a small chunk of HTML content.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using HTML Forms

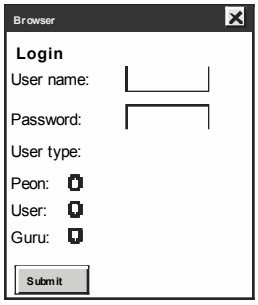
Input Type	Return from getParameterValue
Text	Text entered by user or default as specified by optional VALUE attribute
Password	Text entered by user or default as specified by optional VALUE attribute
Checkbox	If not checked, null. Otherwise the VALUE attribute or "on" if no VALUE specified
Radio	VALUE attribute of selected radio button or null if no radio button selected

This chart summarizes some of the common "widgets" used in HTML forms. Each type of input element has a required Name attribute and an optional Value attribute. The servlet can query the Value by calling getParameterValue on the request object, specifying the parameter Name..

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Example: The HTML Form

```
1    <html>
2    <body>
3
4    <h1>Login</h1>
5    <form action="Login">
6    User name: <input type="text" name="username"></input><br>
7    Password: <input type="password" name="password"></input><br>
8    User type:<br>
9    Peon<input type="radio" name="usertype" value="peon"
10                  checked="true"></input><br>
11   User<input type="radio" name="usertype" value="user"></input><br>
12   Guru<input type="radio" name="usertype" value="guru"></input><br>
13   <input type="submit" value="Submit"></input>
14   </form>
15
16   </body>
17   </html>
```



This is a simple form that defines two text input elements and a group of radio buttons. Note the names of the input fields: "username" and "password" -- the servlet uses those names to retrieve the values that the user entered.

Since the radio buttons are in a group, at most one can be selected at any time. We organize the group by assigning each radio button the same name attribute: "usertype". When the servlet calls getParameter with "usertype" as the argument, that method will return the value of the currently selected button, for example "guru". Also note in line 10, we use the checked attribute for one of the radio buttons to indicate it should be initially selected.

Finally, we define the submit button that will invoke the servlet specified in the action attribute back in line 5.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Example: The Desired Output

```
1    <html><body>
2    <h2>Login successful</h2>
3    <p>You are logged in as:
4    Paul Westerberg, user</p>
5    </body></html>
```

or...

```
1    <html><body>
2    <h2>Login failed</h2>
3    </body></html>
```


Before we examine the servlet, let's look at what the resulting HTML should look like. If the login is successful, the HTML shows user name and type. If the login fails, the resulting HTML reflects that event.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Example: The Servlet

```
1    import javax.servlet.*;
2    import javax.servlet.http.*;
3    import java.io.*;
4    import java.util.*;
5    import javax.servlet.annotation.*;
6
7    @WebServlet("/Login")
8    public class Login extends HttpServlet
9    {
10       public void doGet ( HttpServletRequest req
11                         , HttpServletResponse res )
12           throws ServletException, IOException
13       {
14           res.setContentType ( "text/html" );
15           PrintWriter out = res.getWriter();
16           out.println ( "<html><body>" );
17
18           String userName = req.getParameter ( "username" );
19           String password = req.getParameter ( "password" );
20           String userType = req.getParameter ( "usertype" );
```


As usual, the Login servlet extends HttpServlet and overrides the doGet method. After setting the content type and retrieving a PrintWriter, lines 18 to 20 retrieve the values that the user entered into the HTML form. Note that the Strings passed to getParameter must exactly match the names defined in the HTML form.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlet Example: The Servlet, cont'd

```
21          if  ( ( userName.equals ( "Paul Westerberg" ) &&
22                ( password.equals ( "Tim" ) ) ) )
23          {
24              out.println ( "<h2>Login successful</h2>" );
25              out.println ( "<p>You are logged in as: " );
26              out.println ( userName + ", " +
27                      userType + "</p>" );
28          }
29          else
30          {
31              out.println ( "<h2>Login failed</h2>" );
32          }
33          out.println ( "</body></html>" );
34      }
35  }
```


After retrieving the parameters, this simple servlet compares the username and password parameters against some hard-coded values. The servlet then generates an HTML response based on the comparison. While this logic is hardly appropriate for a real-world login, it demonstrates how a servlet can access parameters from a form and generate different HTML depending on the parameters.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The advantages of using servlets
- About servlet containers
- How to write a simple servlet

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to JavaServer Pages

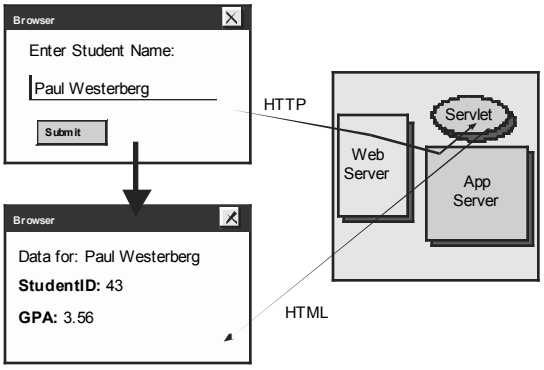
- What are JavaServer Pages?
- Separating Content From Presentation

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What are JavaServer Pages?

- JSPs are a JEE technology that combine static HTML with embedded Java code
- In order to use JSPs, you need a JSP container as part of your application server
- Popular JSP containers include IBM WebSphere, JBoss, Apache Tomcat and BEA WebLogic



JSPs are similar to other server-side scripting technologies such as Microsoft's Active Server Pages (ASP). Like ASPs, well-written JSPs are mostly HTML with small snippets of Java code (not JavaScript).

And just as with servlets, to use JSPs, you need a container to host the JSPs. In fact, many vendors supply both JSP and servlet containers in the same application server product.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSPs Are Actually Servlets

- You write a JSP by coding HTML with snippets of Java code referred to as *scriptlets*
- When the JSP container loads the JSP, it translates it into a servlet and then compiles it on the fly

```
1    <jsp:useBean id="now" class="java.util.Date"/>
2    <html>
3      <body>
4        <h3>My First Page</h3>
5        <p>
6          The <b>current</b> time is: ${now}
7        </p>
8      </body>
9    </html>
```


This page shows a simple JSP that outputs HTML for "Hello, world" five times. Note that the JSP consists of standard HTML boilerplate in lines 1 and 8, and a chunk of Java code bracketed by <% and %>. The code within this bracket is regular, old Java code. Note that it references an implicit variable</i> named "out" -- this variable is defined for the JSP and is similar to the PrintWriter object used by servlets to write content back to the client.

To run a JSP, a client enters the JSP's URL into a browser.

It's important to realize the JSPs end up being servlets -- the first time a reference is made to the JSP's URL, the container translates the JSP into a servlet and then compiles it and deploys it into a servlet container. The servlet container then calls the normal methods (for example "init") on the generated servlet.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use JSPs?

- JSPs are easier than servlets for non-programmers to create - you can use standard HTML authoring tools
- Since JSPs are compiled into servlets, they perform well
- JSPs can act as the View portion of a Model-View-Controller architected web application

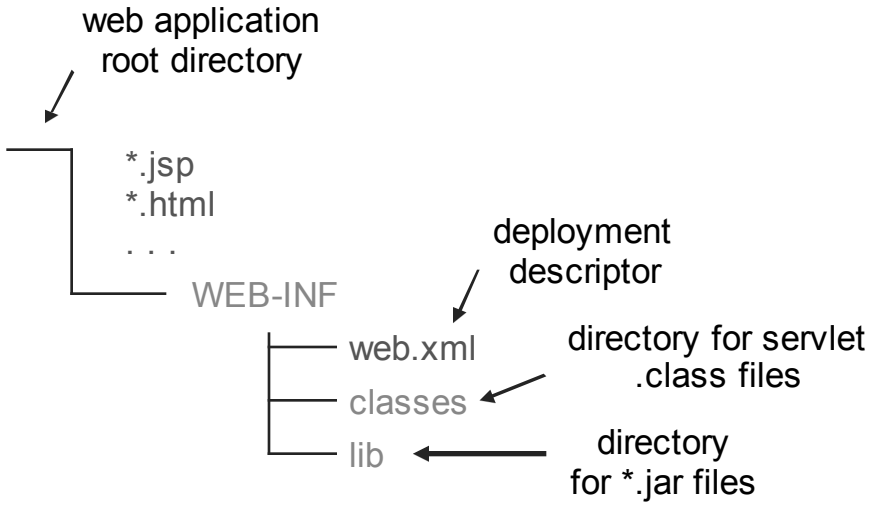
Since JSPs end up being servlets, you may wonder why you would JSPs instead of servlets. The main reason is that JSPs are mainly HTML and thus can be written by web-page designers using standard HTML authoring tools. And since they are really servlets, they have all of the advantages of servlets including good performance, platform independence and so forth.

Finally, JSPs work well together with servlets using an architecture known as Model-View-Controller. We will cover MVC in the next chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Deploying JSPs

- You normally deploy JSP files in the same directory hierarchy as static HTML files



Even though JSPs end up being servlets, you deploy them into the root directory of the web application's directory hierarchy. The container will then translate the JSP into a servlet, compile it and deploy it in a container-specific fashion.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSP Version History

- Version 1.0 introduced JSPs and defined basic syntax of **directives** and **Java scripting elements**
- Version 1.1 added **action elements** and **tag libraries**
- Version 2.0 added an **expression language** and a simpler way to write tag libraries

```
<!-- Directive -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!-- Action -->
<jsp:useBean id="now" class="java.util.Date" />

<!-- Scripting element (rarely used in modern JSPs) -->
<%
   out.println("Hi");
%>
```


A "directive" is a command to the page translator, while a "scripting element" lets you embed executable Java statements within a JSP.

"Action elements" are like directives -- they extend the set of commands that the JSP translator recognizes. A "tag library" is a collection of user-written custom action elements.

An "expression language" is a set of syntax that lets you perform common operations like adding or subtracting numbers, accessing well-known variables and so forth.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Intro to the JSP Expression Language

- The JSP **expression language** is a language, based on Java, that makes it easy for JSP authors to perform calculations and access data defined in *JavaBeans*
- Prior to JSP 2.0, you could only use the expression language in conjunction with the *JSP Standard Tag Library* (JSTL)

```
<p>
  The total is: ${item.quantity * item.price}
</p>
```

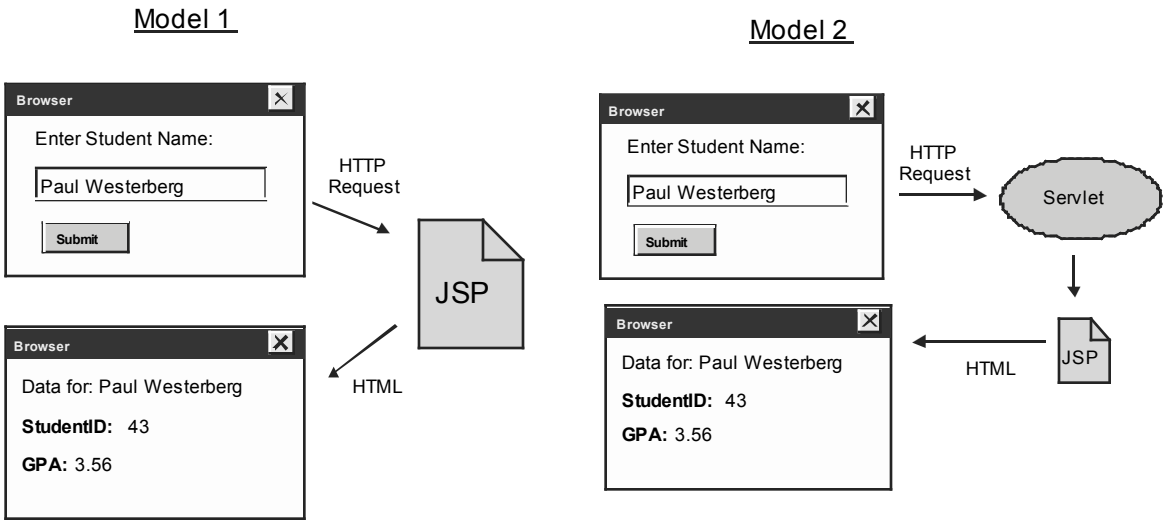

Expressions are delimited by '${' at the start and '}' at the end. The expression shown on this page multiples the value of a JavaBean property named "quantity" times another property named "price".

Before the advent of the expression language, the only way to add programmability to JSPs was to write scripting elements, which consist of Java code. The downside to scripting elements is that the page author needs to be familiar with Java syntax and notions such as garbage collection, primitives versus references and so forth. The expression language is much easier for non-programmers to learn than full-blown Java.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Model 1 vs Model 2

- In a **Model 1** architected application, JSPs respond directly to HTML forms
- In a **Model 2** (aka MVC) application, servlets process forms and then transfer control to JSPs



There are two possible ways to architect JEE Web applications -- Model 1-based applications use JSPs exclusively, while Model 2 apps use a mixture of servlets and JSPs. Model 2-based designs are considered superior for large applications, since Model 2 provides easier maintenance. On the other hand, Model 1 is simpler for smaller applications.

Note there is a third possibility -- using only servlets. There's no catchy "Model x" moniker for that choice however.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Separating Content From Presentation

- If you are not careful, you can end up with JSPs that are "cluttered" with too much code in scriptlets, obscuring the static HTML
- That makes pages hard to understand and maintain -- in addition, if the business logic is in the JSP itself, you need to update the JSP if the logic changes
- A better design is to separate the bulk of the logic into external JavaBeans or EJBs and concentrate on the presentation in the JSP

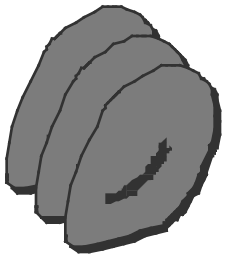
JSPs with too much code obviate the advantages of JSPs: they cannot easily be written or maintained by non-programmers. And since the JSP is a single "source" file, too much code quickly becomes unwieldy and hard to edit. In fact, the best JSPs have no code -- all the programmability is provided by action elements.

So if you find yourself writing a lot of code in your JSPs, you should consider re-architecting your system so that the JSPs call out to external Java code. We will cover that technique next.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a JavaBean?

- A JavaBean is a Java class that obeys a few simple conventions:
  - Use get/set methods to expose properties
  - Use public methods to expose behavior
  - Support firing events
- Typically, we don't use JavaBean events with JSPs



One recommended technique of minimizing JSP "code bloat" is to have the JSP reference JavaBeans that reside in the same Java Virtual Machine as the generated servlet. Sun wrote the JavaBean specification to allow just this sort of thing.

This page shows some of the conventions that you should follow so that a Java class can be considered a JavaBean. The most useful characteristic of JavaBean for JSPs is the notion of properties that we will cover on the next

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a JavaBean Property?

- A property is defined by the existence of "getter" and "setter" methods
- A readonly property defines only a "getter" method

```
1    public String getAddress()
2    {
3       ...
4    }
5    public void setAddress ( String s )
6    {
7      ...
8    }
9    public int getSerialNumber ()
10   {
11     ...
12   }
```

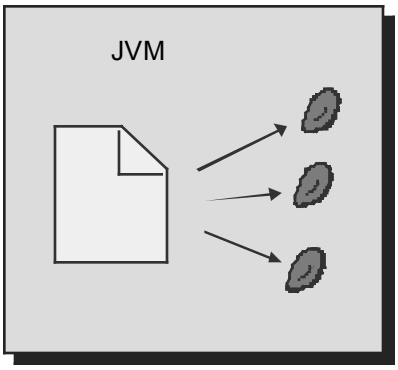

JavaBeans can define properties by providing methods that follow a naming convention. A normal property is made up of a "get" method and a "set" method -- the name of the property is what follows after the "get" or "set". On this page, lines 1 to 8 define a property named "address". Note that the property name starts with a lower-case character. Every property also has a type -- the "address" property's type is String.

This page also shows a read-only property of type "int" named "serialNumber".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a JavaBean from a JSP

- There are two ways to access a JavaBean from a JSP:
  - Using built-in **JSP actions** (we will not cover)
  - Using the **JSP expression language**



Now that you know how to write a Bean, let's turn our attention on how to write a JSP that uses the Bean.

Those of you taking the class that are Java programmers know that to access any Java object, you must first call the "new" operator to instantiate the object and then call the object's methods. To understand that process, you first need to learn quite a few fundamental Java concepts such as references, garbage collection and the like. Fortunately, non-programmers writing JSPs can avoid that learning curve by using the special syntax that we will cover next.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JavaBean Scopes

- JEE web applications can share JavaBeans and other objects by storing or accessing them using **scopes**:
    - **Page** -- local to a single JSP (no sharing)
    - **Request** -- shared among servlets and JSPs running on a single HTTP request
    - **Session** -- shared among servlets and JSPs running within a session (series of requests)
    - **Application** -- singletons shared among all JSPs and servlets

A session is a series of HTTP requests from the same user, running the same browser. Sessions provide a way for JEE web applications to "remember" data across requests, for example, login credentials.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Expressions to Read Properties

- You can use JSP expressions to access bean properties using the dot operator
- The expression evaluator searches for the bean at each of the four scopes: **page**, **request**, **session** and **application**

```
1    <!- assume a bean named "student"
2         is stored at session scope -->
3
4    <p>The student's GPA: ${student.gpa}</p>
5    <p>
6      <!-- this syntax rarely used -->
7      The student's name:
8         ${sessionScope.student.name}
9    </p>
```


Since the expression evaluator automatically searches the four scopes, you can reference a bean with only its name. However, as the second example shows, you can explicitly specify the scope if desired.

Behind the scenes, the expression evaluator calls the "get" methods on a bean to retrieve its properties.

Note that to use the expressions, you don't need to use the jsp:useBean action (assuming that another JSP or servlet stores the bean at some scope).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Intro to the JSP Standard Tag Library

- A **tag library** is a set of **custom actions** that extend the JSP syntax
- The JSTL is a collection of custom actions that aims to make it easier for non-programmers to write useful JSPs without writing scripting elements
- By providing a standard set of actions, avoids "re-inventing the wheel"
- The JSTL, along with the EL, supplies a Perl or PHP-like environment where the page author can treat everything as strings

The JSTL library was developed under the Java Community Process and draws inspiration from tag libraries from the Apache Jakarta project and other authors.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

JSTL Core Tags

- The core tags provide basic functions that are useful in many kinds of JSPs

```
<%@ taglib prefix="c"
   uri="http://java.sun.com/jsp/jstl/core" %>
```

Tag	Description
c:out	Write to output stream
c:if	Simple conditional
c:choose,c:when,c:otherwise	Like Java switch-case
c:forEach	Iterating (looping)
c:set	Define or modify an EL variable
c:get	Retrieve an EL variable
c:exception	Handle exception locally
c:import	Similar to jsp:include
c:url	Encodes, rewrites URLs
c:redirect	Redirects to a URL

JSTL provides several categories of tags -- here we examine the "core" tags that have wide applicability.

Before you can use any tag library like the JSTL, the JSP must write the approriate "taglib" directive as shown here.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

c:forEach Example

```
1    <c:forEach var="i"
2               begin="0"
3               end="5">
4        <p>${i}. My List </p>
5    </c:forEach>
```



This example shows iterating with a fixed count. This is similar to a "for" loop in Java, but is not quite as flexible: the "step" must be > zero and the "begin" must be >= zero.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The basics of writing JSPs
- How to access JavaBeans from a JSP

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Model-View-Controller Architecture

- The Model-View-Controller Architecture
- A Sample MVC Web Application

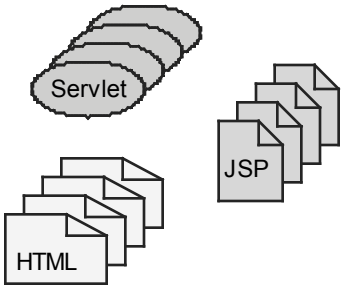
Previous chapters have built the foundation for applications that use both servlets and JSPs. This chapter discusses techniques for effectively combining them.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Servlets versus JSPs

- Servlets are Java classes and thus can be code-intensive and implement complex logic
- JSPs are (or should be) mostly HTML with small snippets of Java code. JSPs are ideal for presentation, not complex logic.
- Many web applications, especially smaller ones, could be successful using either technology exclusively

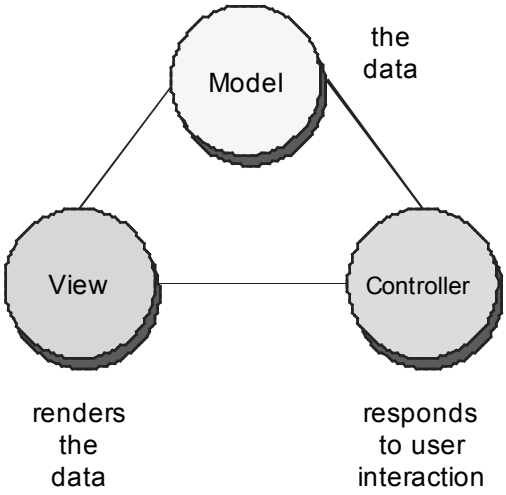


This page describes the strengths and weaknesses of servlets and JSPs. Servlets work well if you have skilled Java programmers and don't expect to modify the HTML very often. JSPs are ideal if you have complex and changing HTML but don't need much programmability.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Combining Servlets and JSPs

- Large projects can leverage the strengths of both technologies by combining them
- One useful way to combine servlets and JSPs is the Model-View-Controller architecture



The Model-View-Controller architecture (MVC) splits an application into three objects. The model component stores the application's data -- the model may connect to a database. The view component reads the data from the model and presents it to the user. There may be multiple views on the same model. The controller responds to requests from the user and updates the model as required. If the model has changed, the view needs to be regenerated.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is the MVC Architecture?

- MVC originated in the 1980s in Smalltalk as a technique to help build graphical user interfaces (GUIs) - Java's Swing GUI toolkit uses a version of MVC
- MVC is effective since it decomposes applications into components that you can develop, debug and modify separately

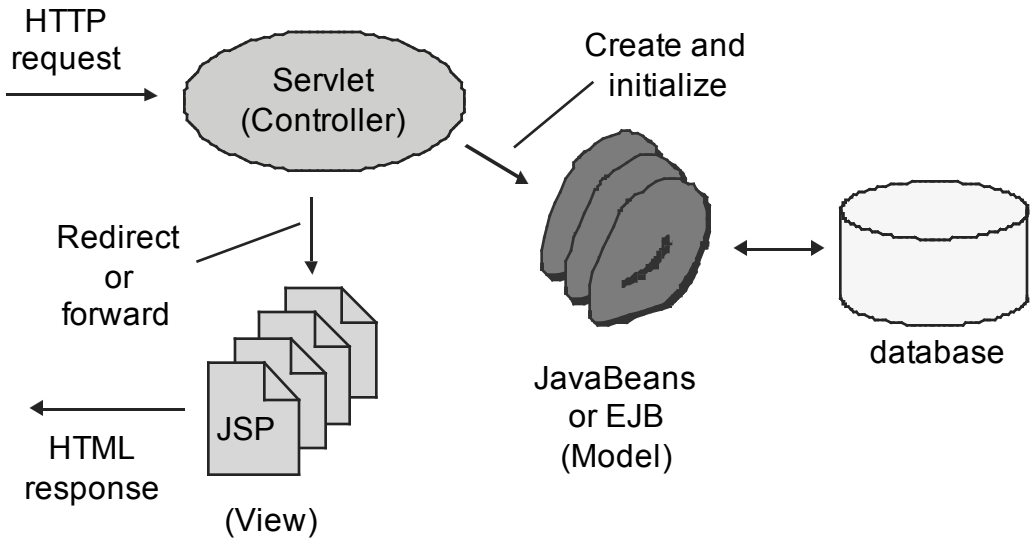
MVC is not a new architecture -- it has been used effectively in many GUI applications.

Let's know see how we can apply MVC to web applications.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

MVC and Web Applications

- The servlet can create Bean instances and store their references in a session before passing control to the JSPs, which generate the presentation



To use MVC in a web application, a servlet or servlets play the role of controller. The servlet or servlets respond to HTTP requests and then typically instantiate or update JavaBeans, which act as the model. The servlet then transfers control to a JSP, which reads the model (JavaBeans) and formats an HTML response.

This arrangement takes advantage of the strengths of servlets and JSPs -- the controller can contain complex Java code that works with JavaBeans that contain the data and business logic of the application. The controller does not generate HTML -- that's the job of the JSPs. So if you need to update just the presentation, e.g. change a font, you only need to modify the JSPs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

MVC Tradeoffs

Plus

Good separation of concerns
Easier to maintain
Flexible design for heterogeneous clients

Minus

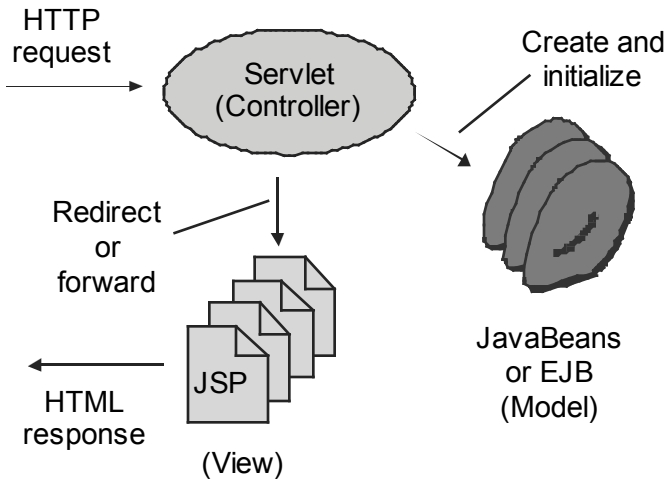
More complexity
More up-front design needed
Controller can get overly "busy"

This page lists some of the good and bad characteristics of MVC designs. Even though MVC may seem like overkill for a small project, remember that often, small projects turn into large ones.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Transfering Control

- In MVC applications, controller servlets need to transfer control to another servlet or JSP
- There are two basic ways to transfer control:
  - Forwarding the request
  - Redirecting the request

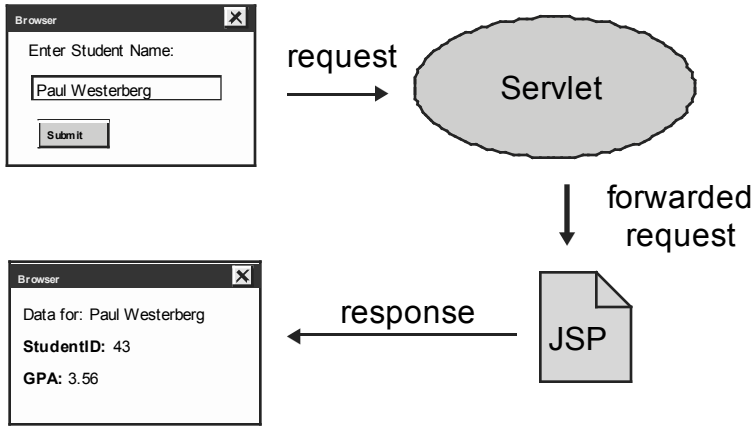


As an MVC architect/programmer, you will need to decide whether to use redirection or forwarding to transfer control. The next few pages will give you some information to help you make that choice.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Forwarding Requests

- Forwarding a request transfers control to another URL using the same request -- objects stored at request scope are available to the target
- Forwarding is a relatively fast operation since the client is not involved



Forwarding a request is almost like performing an internal function call on the server. When you forward, the container discards any buffered output and then transfers control to target servlet or JSP. The target normally receives the same request and response objects as the source, and thus can access client arguments or objects stored at request scope.

The downside of using forward is that even after the response, the URL in the browser remains the URL for the source page, not the forwarded page. That can cause undesirable side effects if the user reloads the page in the browser, which sends another request to the source page. For example, suppose the source page adds a record to a database -- if the user hits the refresh button, you can end up with an extra row in the database. In that situation, you should use redirection instead of forwarding, though redirection is slower.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Forwarding Requests, cont'd

- A servlet can use the RequestDispatcher to forward a request

```
1    public void doGet (
2                  HttpServletRequest request
3                , HttpServletResponse response )
4              throws ServletException, IOException
5        {
6           . . .
7           ServletContext ctx = getServletContext();
8           RequestDispatcher disp =
9                   ctx.getRequestDispatcher(
10                        "/mytarget.jsp");
11          disp.forward(request, response);
12          . . .
```

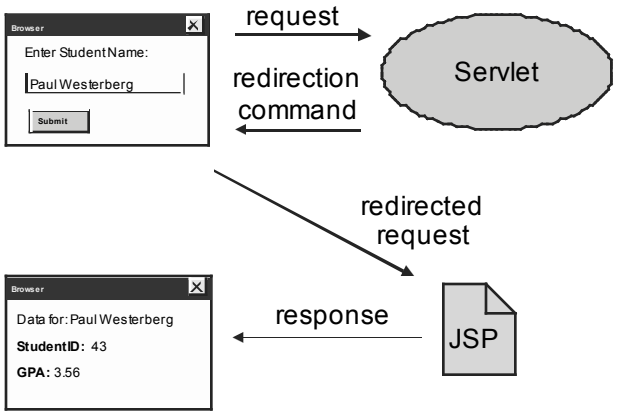

Here we show code in a servlet that forwards a request to the "mytarget.jsp" JSP. In line 7, we retrieve a reference to the servlet context, from which we can create a RequestDispatcher object for the target URL (lines 8 to 10). Then in line 11, we forward the request, passing the servlet's own request and response parameters.

The container interprets the target URL relative to the current web application -- essentially that means that you can only forward to pages that are within the current web application.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Redirecting Requests

- Redirecting a request transfers control to another URL by telling the client to make a new request -- objects stored at request scope are lost, but objects at session scope are OK
- Redirecting is slower than forwarding, but works properly if the user reloads the target page in the browser



When you redirect a request, the container sends an HTTP response code back to the browser which instructs the browser to send a new request to the target URL. Since redirection involves another round trip between the client and server, it's slower than using forward.

The main advantage of redirection over forwarding is that afterwards, the browser "knows" the URL of the redirected page, not the source page. That way, if the user reloads the page, the source page doesn't execute again. The disadvantage is slower performance, plus the fact that session-scope objects persist longer than request-scope objects -- using session-scope objects can impact scalability due to memory consumption.

You can use either relative or absolute URLs when you redirect, but if you use a relative URL, the container converts it to an absolute URL before returning the redirection response to the client.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Redirecting Requests, cont'd

- A servlet can use the ServletRequest to force a redirect

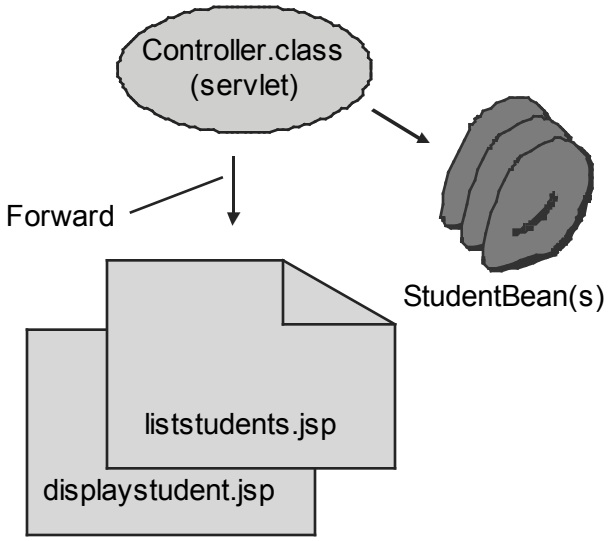
```
1    public void doGet (
2                  HttpServletRequest request
3                , HttpServletResponse response )
4              throws ServletException, IOException
5        {
6              . . .
7              response.sendRedirect (
8                "mytarget.jsp" );
9              . . .
```


Here we show a servlet redirecting a response to a "mytarget.jsp" JSP using a relative URL. As part of redirecting the request, the container discards any buffered output the servlet has performed before calling "sendRedirect".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

MVC Example: Student Information System

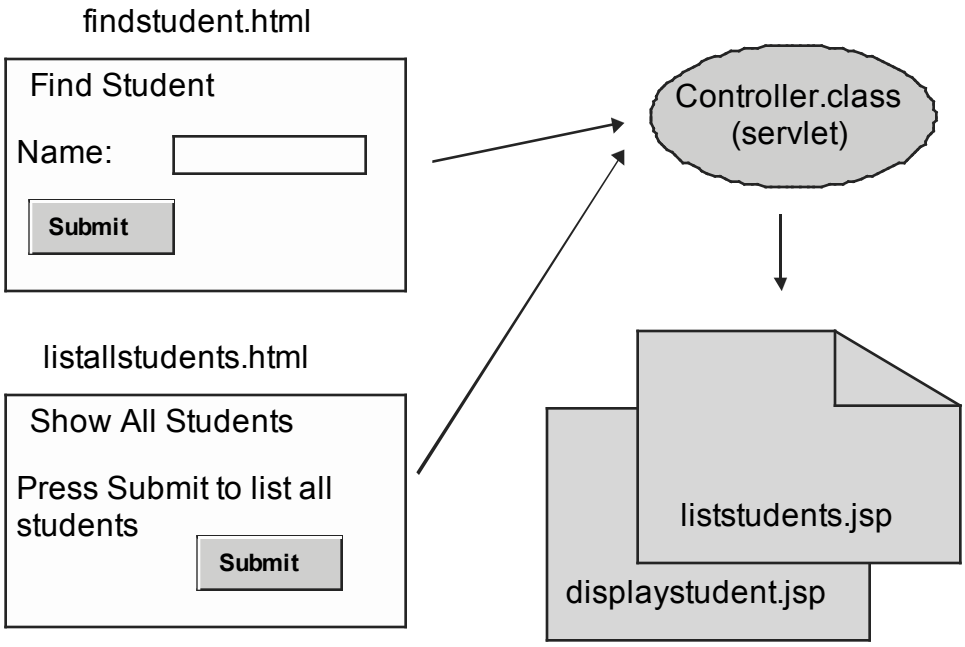
- This simple MVC web application uses a single servlet as Controller, StudentBeans as the Model and two JSPs for the View



For the next few pages, we will examine a small but functional MVC application. This application has a single servlet that instantiates a JavaBean and then transfers control to one of two JSPs.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

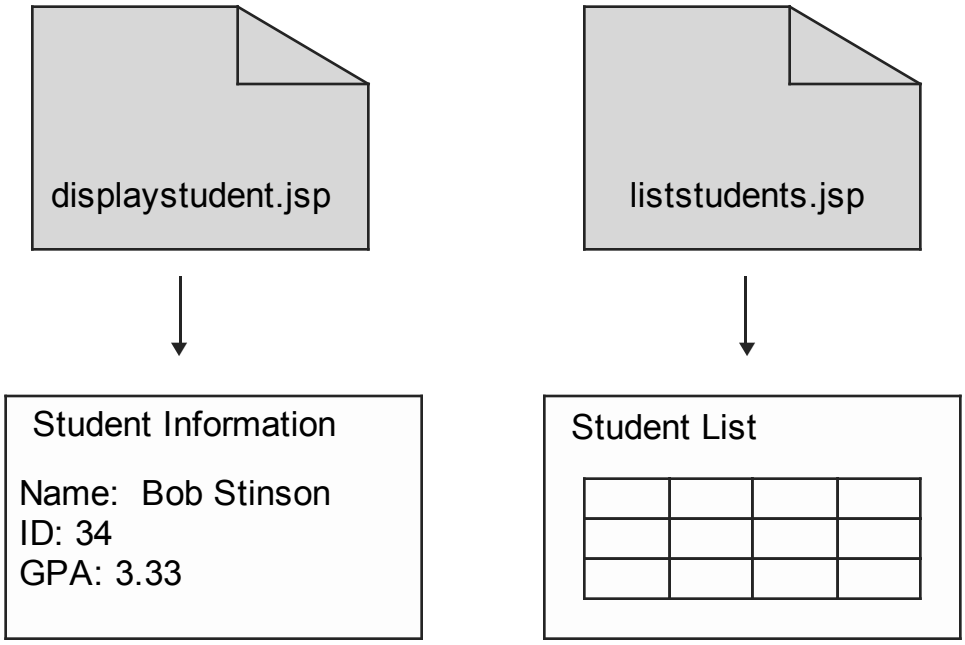
MVC Example User Interface



The initial part of the user interface consists of two static HTML pages that contain forms that reference the Controller servlet. This application lets the user query for a single student or obtain a list of all students.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

MVC Example User Interface, cont'd



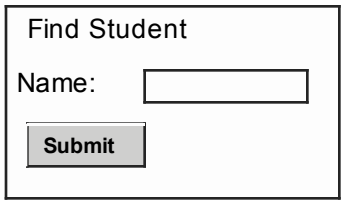
The next part of the user interface is the pages generated by the JSPs. The display-student JSP displays a simple HTML page that shows information for a single student. The list-students JSP displays an HTML table with a row for each student.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Find Student HTML

- We use a hidden form field to pass an argument to the Controller servlet that indicates which action the servlet should take

```
. . .
<h1>Find Student</h1>
<form
  action="Controller">
Name: <input type="text" name="StudentName"><p>
<input type="Hidden" name="Action" value="FindStudent">
<input type="submit" value="Submit">
</form>
. . .
```



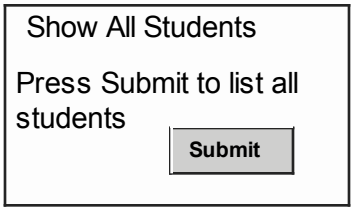
Since this application uses a single servlet to respond to all user requests, we need some way for the servlet to differentiate the requests. The technique used here is to include a hidden field in each form named "Action" that identifies the request. Here we show a fragment of the static HTML page that lets the user query information about a single student.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The List All Students HTML

- We use a hidden form field to pass an argument to the Controller servlet that indicates which action the servlet should take

```
. . .
<h1>Show All Students</h1>
Press Submit to list all students
<form
  action="Controller">
<input type="hidden" name="Action" value="ListAll">
<input type="submit" value="Submit">
</form>
. . .
```



This page shows a fragment of the static HTML form that lets the user ask for a list of all students. Note that we use the same hidden form-field name, "Action", with a different value.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Controller Servlet

```
1    public void doGet ( HttpServletRequest request
2                      , HttpServletResponse response )
3        throws ServletException, IOException
4    {
5        String action = request.getParameter ( "Action" );
6        if  ( action.equals ( "FindStudent" ) )
7        {
8            studentinfo.StudentBean sb =
9               new studentinfo.StudentBean ();
10
11           // initialize Bean from DB
12
13           request.setAttribute ( "student", sb );
14           ServletContext ctx = getServletContext();
15           RequestDispatcher disp =
16                  ctx.getRequestDispatcher(
17                        "/displaystudent.jsp");
18           disp.forward(request, response);
19       }
```


In line 5, the controller servlet first determines which HTML form generated the request by examining the Action parameter, which corresponds to the hidden form field. The code in lines 7 through 17 handles requests from the "FindStudent" form. In line 8, we instantiate a JavaBean that represents a Student and initialize its properties from a database (code removed for brevity). Then in lines 13 and 14, we store a reference to the bean in the request using the attribute name "student". Finally, we pass control to the appropriate JSP in lines 14 to 18. Note that since this is a query, we use a foward and request scope for best performance.

The code for the other use-case, list-all students, is similar, but not shown here.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Controller Servlet, cont'd

```
20      else if  (action.equals("ListAll"))
21      {
22          Vector studentList = new Vector();
23
24          // retrieve all students from DB
25          // populate studentList with intialized beans
26
27          request.setAttribute ("studentList",
28               studentList );
29          ServletContext ctx = getServletContext();
30          RequestDispatcher disp =
31                 ctx.getRequestDispatcher(
32                       "/liststudents.jsp");
33          disp.forward(request, response);
34      }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Display Student JSP

- The **useBean** action attempts to find a Bean with the specified *id* in the request object
- The **getProperty** action retrieves a Bean property given the property's name

```
1    . . .
2    <h1>Student Information</h1>
3
4    <jsp:useBean id="student" class="studentinfo.StudentBean"
5                 scope="request"/>
6
7    <p>Name: <jsp:getProperty name="student" property="name"/>
8    <p>ID:   <jsp:getProperty name="student" property="studentID"/>
9    <p>GPA:  <jsp:getProperty name="student" property="gpa"/>
10   . . .
```

This page shows a fragment of code from the display-student JSP. In line 4, we first retrieve the Student bean at request scope, using the same name, "student" under which the servlet stored the reference into the request. That ensures that the JSP retrieves a reference to the same Bean.

Then in lines 7 to 9, we retrieve properties from the Student bean and emit their values as part of the generated HTML.

The key thing to understand is how the servlet and JSP communicate using a JavaBean stored at request scope.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Display Student JSP, Alternate

- Starting in JSP 2.x, you can use the **JSP Expression Language** to retrieve and display properties

```
1    . . .
2    <h1>Student Information</h1>
3
4    <p>Name: ${student.name}
5    <p>ID:   ${student.studentID}
6    <p>GPA:  ${student.gpa}
7    . . .
```


If you have a JSP 2.x container or later, it's simple to display the properties of beans stored at any of the four scopes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The List All Students JSP

```
1    <%@ taglib prefix="c"
2          uri="http://java.sun.com/jsp/jstl/core" %>
3
4    <h1>List All Students</li>
5
6    <table>
7      <c:forEach items="${studentList}" var="aStudent">
8        <tr>
9          <td>${aStudent.studentID}</td>
10         <td>${aStudent.name}</td>
11         <td>${aStudent.gpa}</td>
12       </tr>
13     </c:forEach>
14   </table>
```


Here we show the list-all students JSP. Here we assume a JSP 2.x container, since we are using the JSP EL to display bean properties.

We are also using the JSP Standard Tag Library (JSTL), which, among other things, lets us iterate over collections, in this case, the Vector stored in the request by the controller.

We will cover JSTL in more detail in a later chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the MVC architecture
- How to create a web application that combines servlets, JSPs, JavaBeans and static HTML to implement the MVC architecture

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum