Object Oriented Programming with Java

Version 2.2.0 February 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

Object Oriented Programming with Java

Chapter 1: Introduction to the Course

Object-Oriented Programming with Java....1-1
Legal Information....1-2
Object-Oriented Programming with Java....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: Inheritance and Polymorphism

Inheritance and Polymorphism....10-1
What is Inheritance?....10-2
Why Use Inheritance?....10-3
Inheritance Vs Composed-Of Relationships....10-4
Java and Inheritance....10-5
Subclass Objects are a Superset....10-6
Accessing Superclass State and Behavior....10-7
Building Class Hierarchies....10-8
Quiz: What Kind of Relationship?....10-9
Quiz: What Kind of Relationship?....10-10
Quiz: What Kind of Relationship?....10-11
Overriding Behaviors....10-12
The Object Class....10-13
Calling Methods....10-14
Constructors and Superclasses....10-15
Constructors and Superclasses, cont'd....10-16
Polymorphism....10-17
A Typical App....10-18
Polymorphic Reference Assignment....10-19
Writing Polymorphic Algorithims....10-20
Using Abstract Classes....10-21
The final Keyword....10-22
Quick Practice....10-23
Review Questions....10-24
Chapter Summary....10-25

Chapter 11: The Java Class Library

The Java Class Library....11-1

The Class Library 11-2
Java Strings 11-3
String versus StringBuffer/StringBuilder 11-4
Strings are Immutable 11-5
Searching a String: Finding Substrings 11-6
Converting Case 11-7
Parsing Strings into Primitives 11-8
StringBuffer/StringBuilder are Mutable 11-9
The java.util.Date Class 11-10
Parsing Date Strings 11-11
Java 8 Date/Time API 11-12
The BigDecimal Class 11-13
The BigDecimal Class, cont'd 11-14
Chapter Summary 11-15

Chapter 12: Introduction to JUnit

Introduction to JUnit 12-1
Types of Testing 12-2
What is Unit Testing? 12-3
Benefits of Unit Testing 12-4
Terminology 12-5
Introduction to JUnit 12-6
History of JUnit 12-7
IDE Integration 12-8
Steps for Using JUnit 12-9
Sample Class Under Test 12-10
Writing a Test 12-11
JUnit Assertions 12-12
Sample Testing Class 12-13
Test Runners 12-14
Setup and Tear Down Methods 12-15
What is a Test Fixture? 12-16
Testing Exceptions 12-17
Organizing and Packaging Tests 12-18

Chapter Summary....12-19

Chapter 13: Java 8 to Java 11

Java 8 to Java 11....13-1
Java 8 Feature Overview....13-2
Java 8 Date/Time API....13-3
Default Methods....13-4
Introduction to Functional Programming....13-5
Passing Functions Around....13-6
Passing Functions Around: The Old Way....13-7
What is a Lambda Expression?....13-8
Lambda Expression Syntax....13-9
Functional Interfaces....13-10
Using Lambda Expressions Example....13-11
The java.util.function Package....13-12
Predicate Example....13-13
Introduction to Java 8 Streams....13-14
Stream Example....13-15
Java 8 and JavaScript....13-16
Java 9 Feature Overview....13-17
Java 10 and 11 Feature Overview....13-18
Chapter Summary....13-19

Object-Oriented Programming with Java

Version 2.2.0 February 2021



1 - 1

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

Object-Oriented Programming with Java

- 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):
  - 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.

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

Audience: Business analysts, developers, managers and other people interested in learning how to program Java.

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 define and implement interfaces
- Write unit tests for 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

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
Inheritance and Polymorphism
Interfaces

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

Java Class Library
Java Versions 8-11
Introduction to JUnit

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

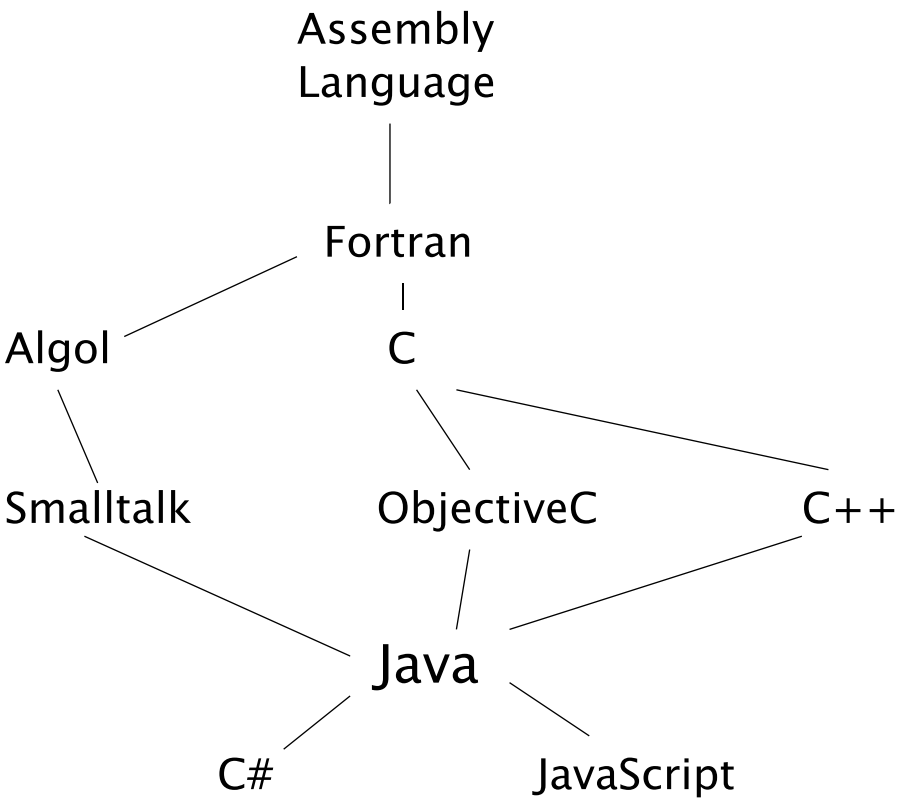
---

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

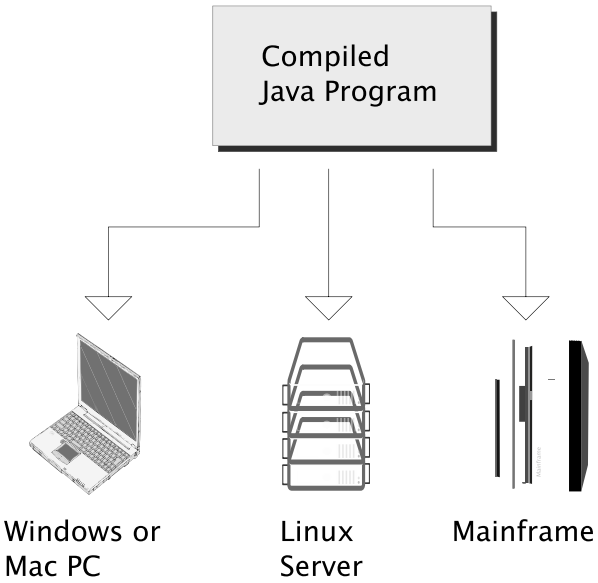
Starting with Java 10, Oracle adopted a faster, 6-month release cadence, with occasional versions referred to as "Long Term Support" (LTS) versions. Java 8 and Java 11 are the current LTS versions (Java 8 support ends in December 2020).

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



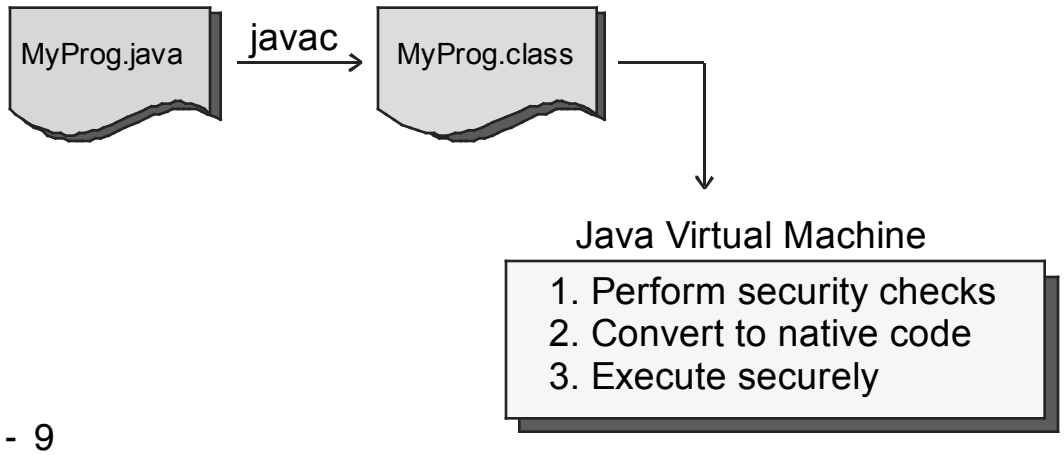
Portability is one of the primary advantages of Java over other languages -- note that we are talking about executable portability, not source-code portability.

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

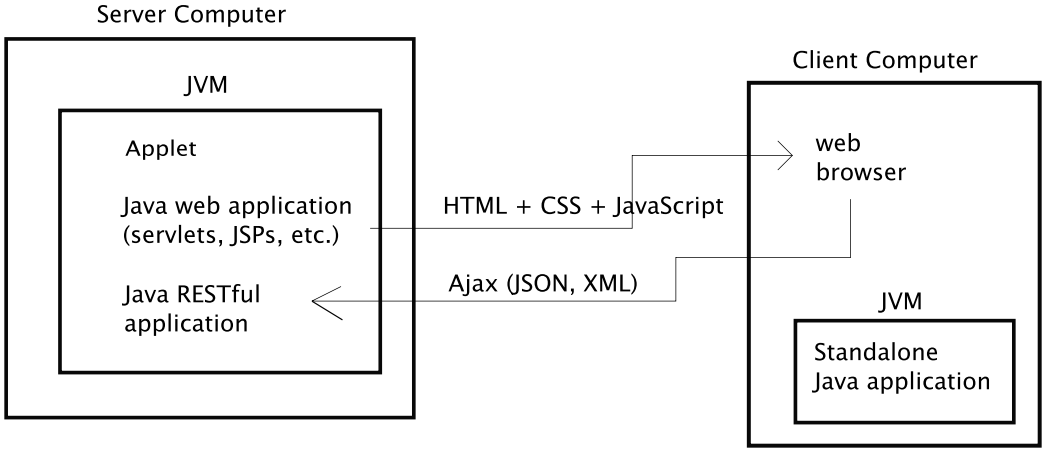
Servlets
JavaServer Pages
Enterprise JavaBeans
Java Naming and Directory Interface
JDBC
Java Message Service
. . .

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

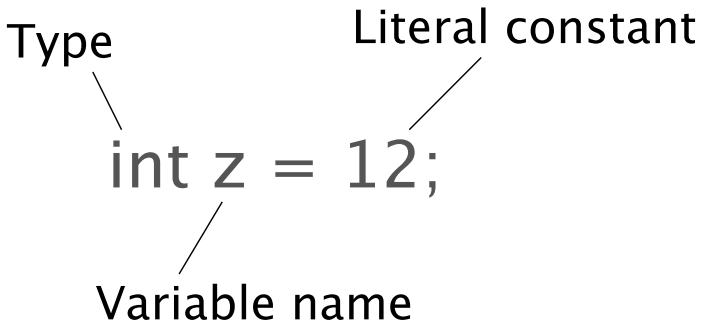
Lines 3, 7 and 9 are examples of statements. Note that statements can span multiple lines -- it's the semicolon that terminates the statement.

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   }
```

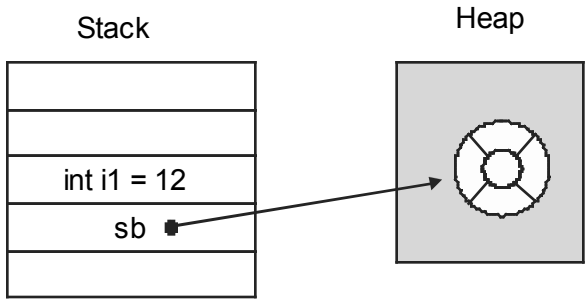

The code on this page shows a Java class definition -- we will cover the syntax in more detail later, but please note that a class defines a block using curly braces. In lines 5 to 9, we nest a method (a.k.a. function) definition which also defines a block. A class can contain many methods, each with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

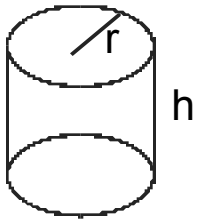
Just for completeness, we show here some of mathematical functions from the Math class. These are not actually part of the Java syntax -- instead, they are defined as methods in that class. Please read the Math documentation for a complete listing of the methods in this class.

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

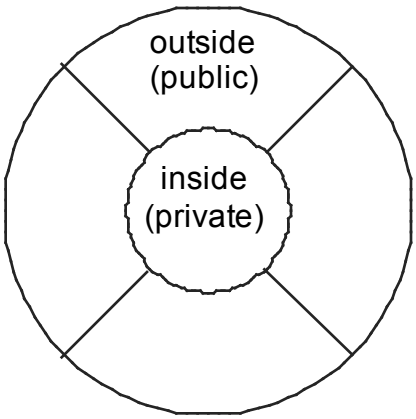
- To be considered a "true" OO language, the language should support:
  - Encapsulation
  - Inheritance
  - Polymorphism

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

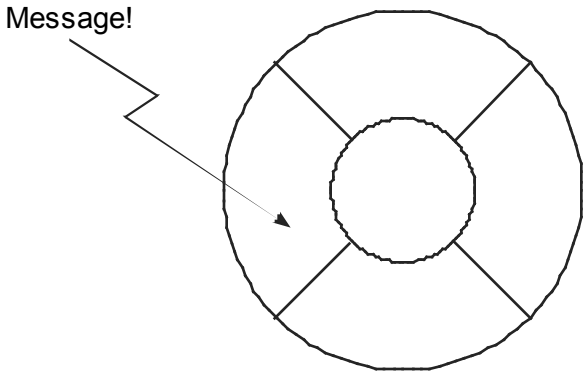
Why are Objects Useful?

- 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

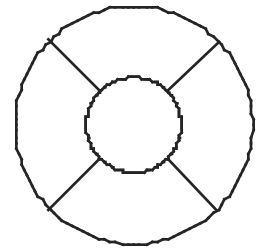
Objects Respond to Messages

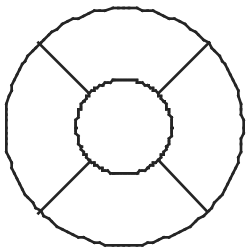
- We refer to executing an object's behavior as "sending the object a message". In Java, we "send messages" by calling methods in the object
- The methods then typically operate on the state (data)

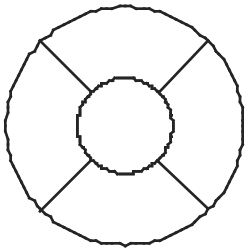


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Objects and Classes



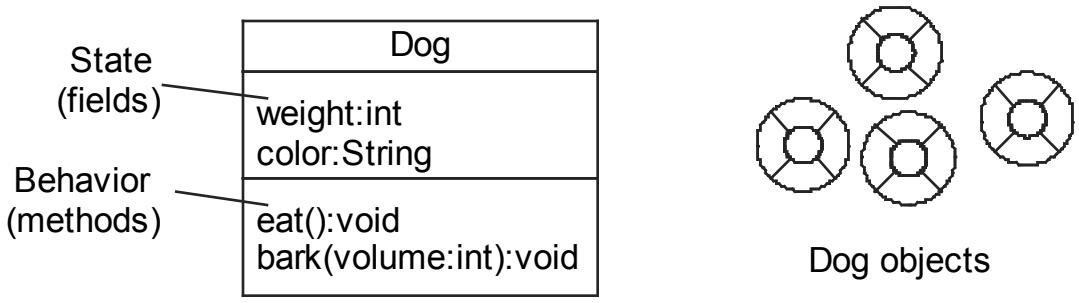




Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

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.

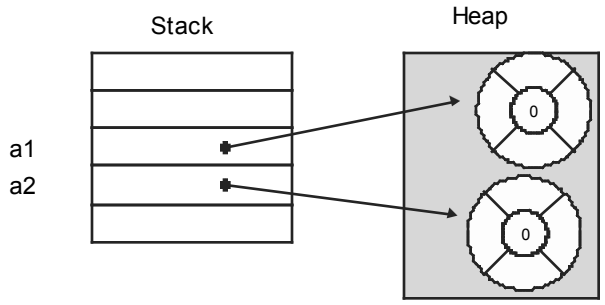
Since Java supports runtime garbage collection, there is no "delete" operator -- the Java Virtual Machine automatically gets rid of unreferenced objects.

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

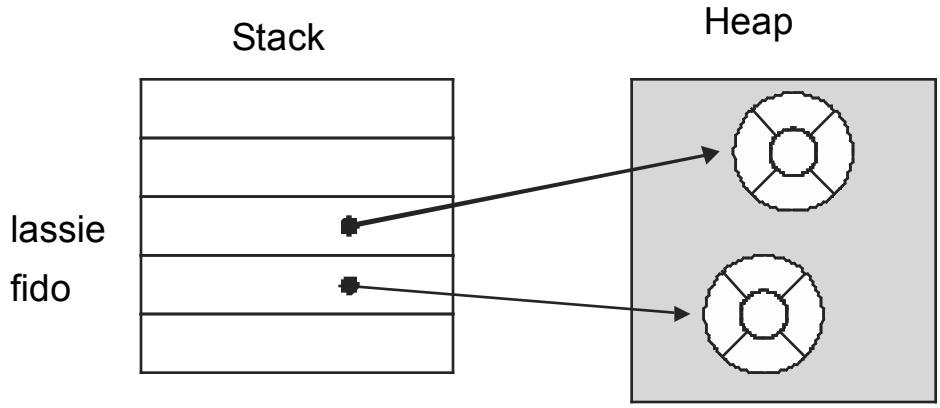
```
1    public class Test
2    {
3       public void method1()
4       {
5       }
6       public void method2()
7       {
8           this.method1();
9           method1();
10      }
11   }
```

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



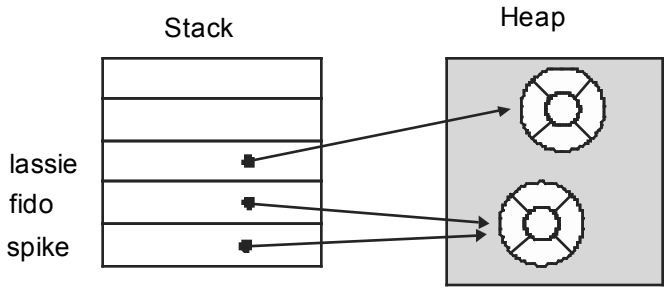
The heap is the free storage remaining in the Java Virtual Machine.

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   }
```



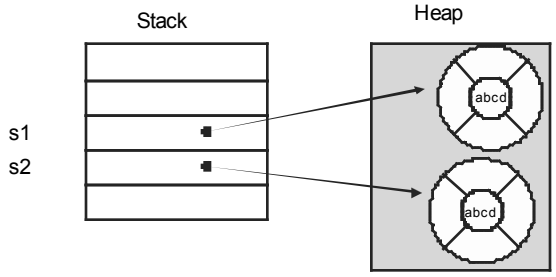
When you use the equality operator on references, it compares the reference values, not the objects themselves.

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.

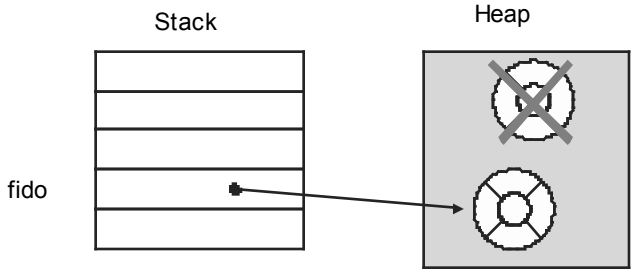
To compare the contents of the objects, the String class provides the equals() method, which returns true only if the two Strings have the same characters (it is a case-sensitive comparison).

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.

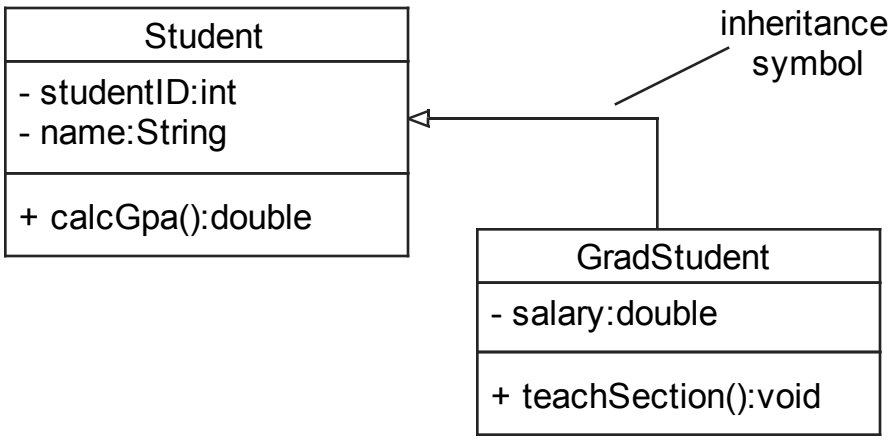
An object on the heap becomes unreferenced either when you explicitly set all of its references to null, or if the local variables that contain the object's references go out of scope.

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

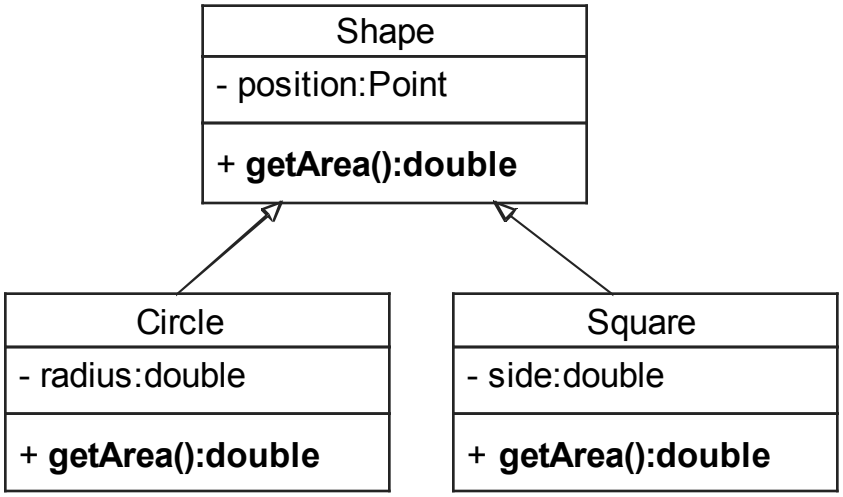
- A subclass inherits state and behavior from its superclass
- Inheritance makes it easier to extend and reuse code
- **Important:** A subclass object is a superset of all of its superclasses



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.

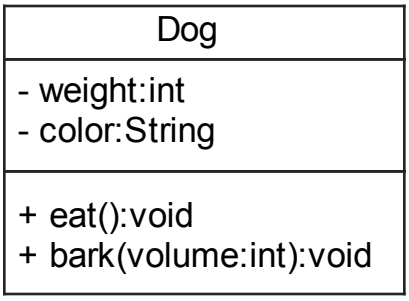
The compiler automatically imports everything from the java.lang package for you.

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

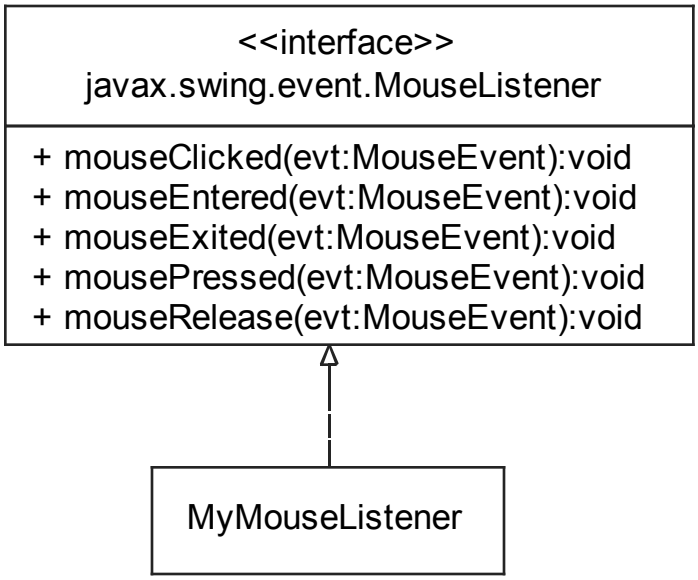
```
1    public class TestAccount
2    {
3      public static void main(String[] args)
4      {
5        Account a1 = new Account();
6        Account a2 = new Account();
7
8        a1.balance = 100;  // ERROR!
9
10       a1.setBalance(100);
11       a2.setBalance(200);
12
13       double d1 = a1.getBalance();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

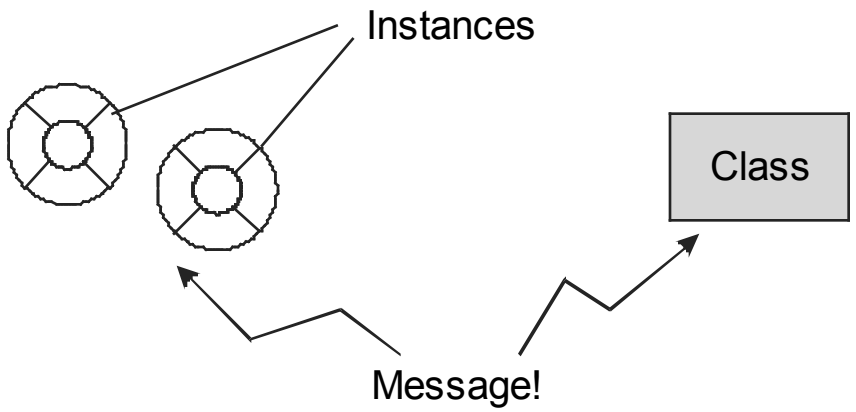
- Instance Methods
- Constructors
- Static Methods

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:

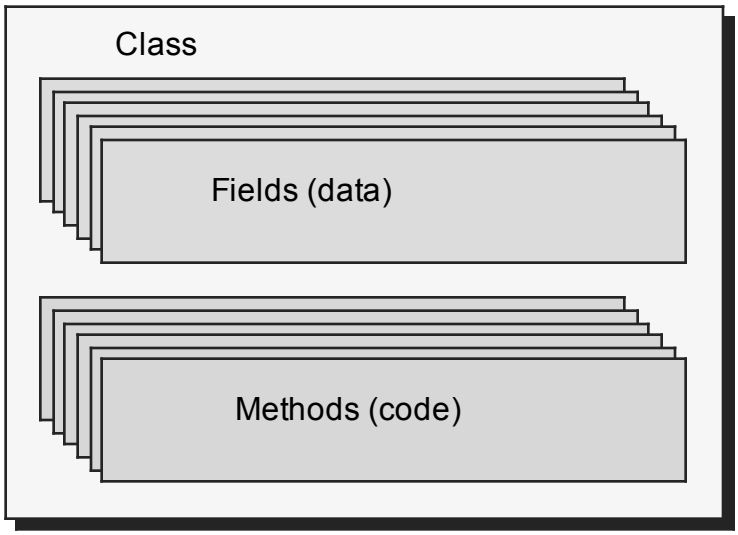
```
public class SomeClass {
  private int x;

  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

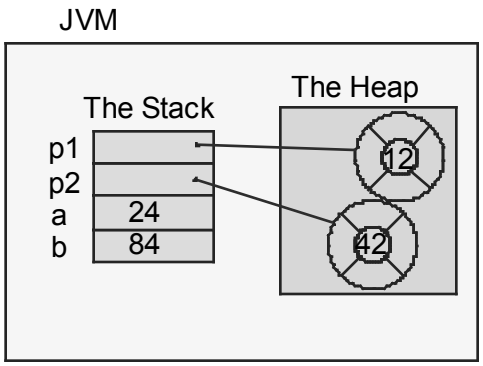
```
1    public class Person
2    {
3      private int age = 14;
4
5      public void setAge(int a)
6      {
7        age = a;
8      }
9      public int method1 (int z)
10     {
11       return age * z;
12     }
13   }
```

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

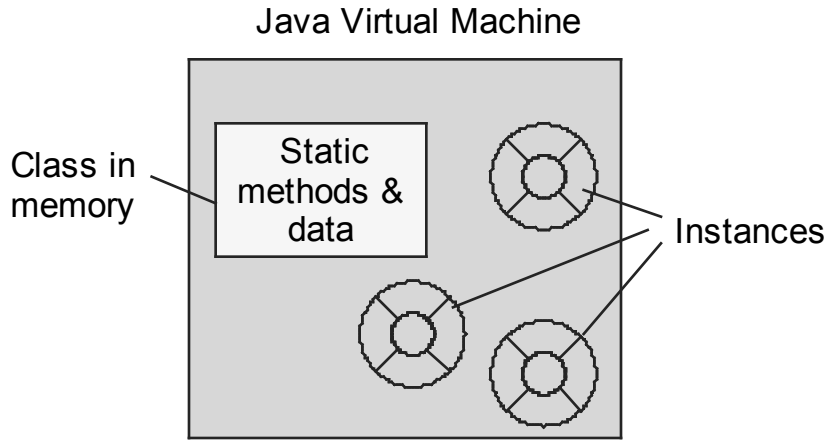
The "public" access modifier means that any code can call the method. The "private" modifier means that only other methods in the same class can call it. The "protected" modifier means that the method is also available to be called from subclasses.

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

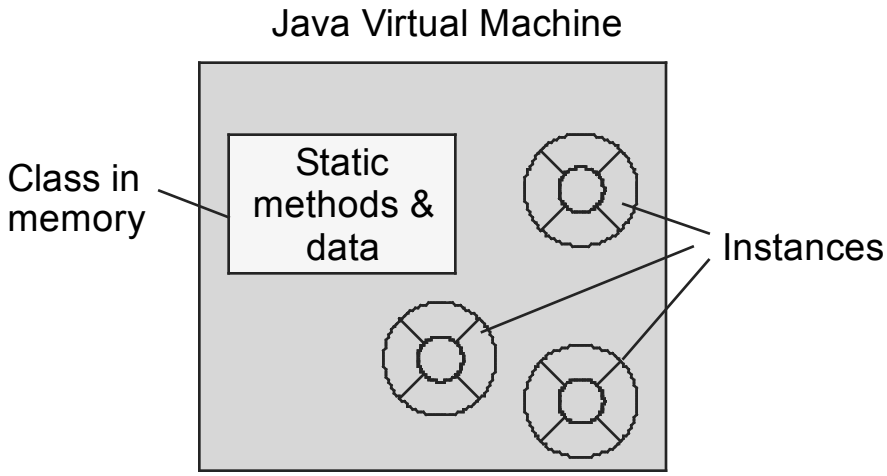
- If a method needs to call a static method in the same class, you can omit the class name as the receiver

```
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.

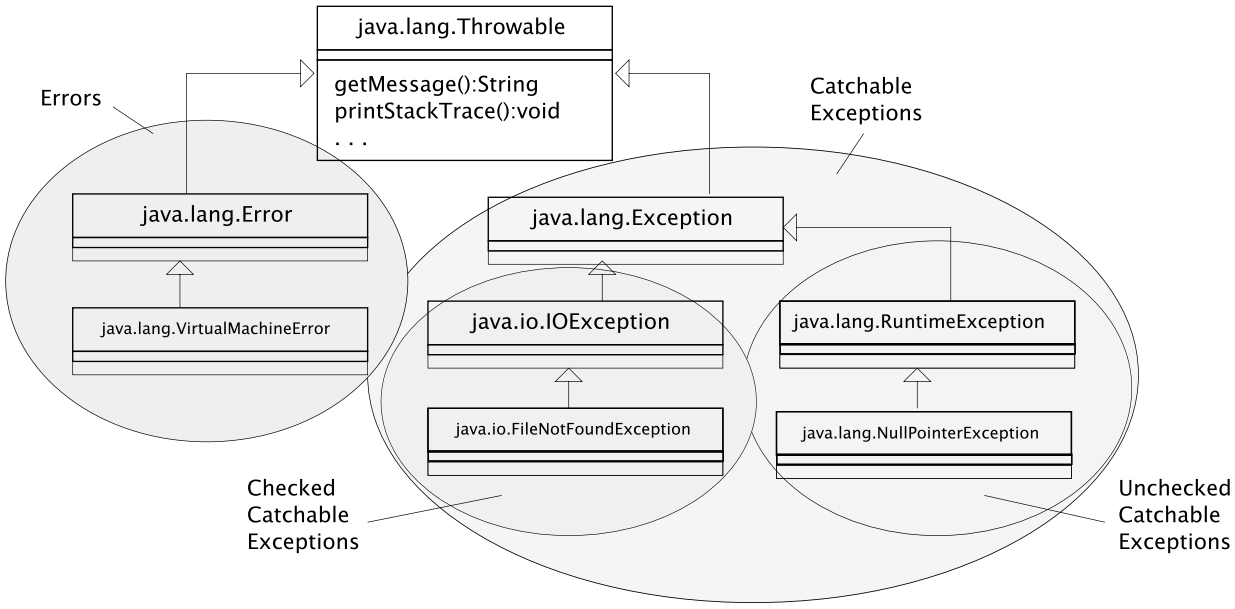
While the compiler doesn't force you handle unchecked exceptions, you can if you wish. NullPointerExcetion is an example that we saw on an earlier page.

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.

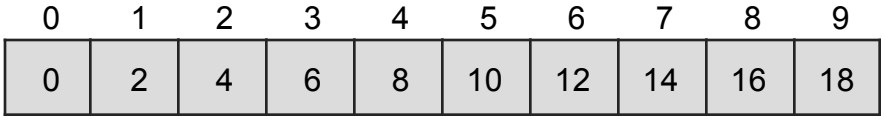
The "new" operator sets all of the array elements to "logical zero" for the type of the array. For example, integers are set to the value zero and arrays of object references are set to "null".

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"
```

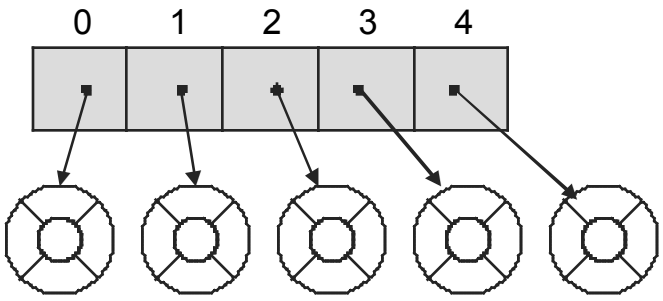

Here we show a common mistake with arrays -- indexes should start at zero and range to n-1 where n is the array's length.

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!

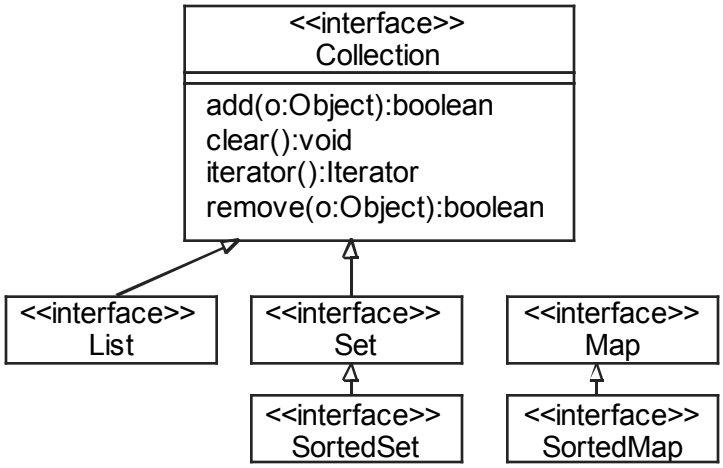
Even though collections do only store object references, it is possible to store primitive values - you can use the so-called "wrapper" classes:

```
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



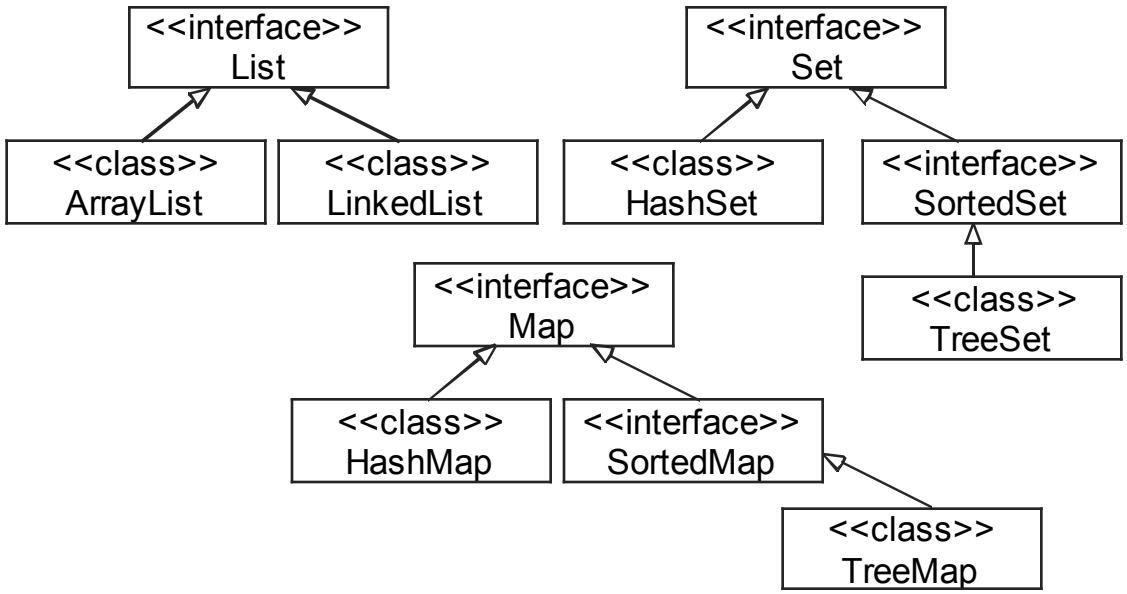
These are all "interfaces", which are like classes but cannot be instantiated. Interfaces define a set of methods that actual classes can implement -- in other words, these are not actual collections, just "prototypes" for the actual collections we will see in a moment.

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

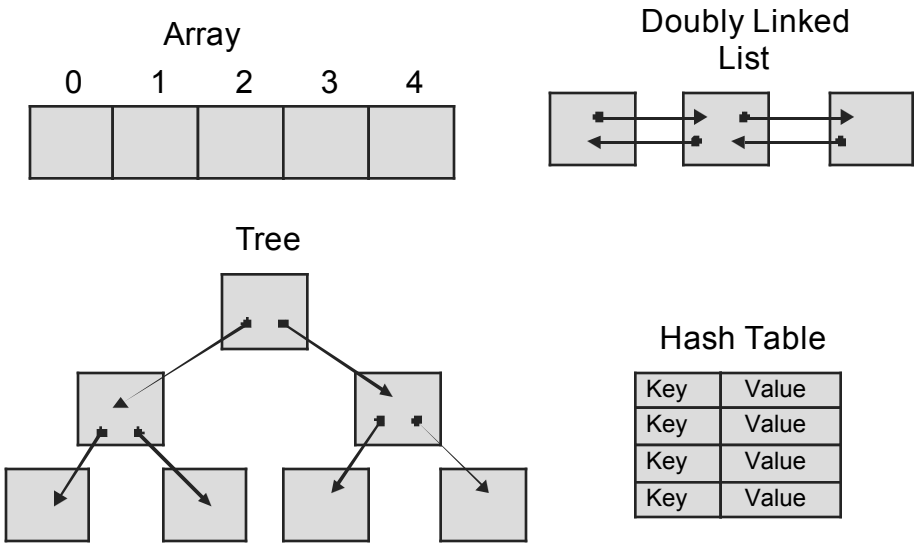


This figure shows a few of the "concrete" collection classes that Java provides; there are others. Note that each concrete class uses a different type of underlying data structure to implement the interface -- for example, an ArrayList uses an array, while a LinkedList uses a doubly-linked list.

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

Inheritance and Polymorphism

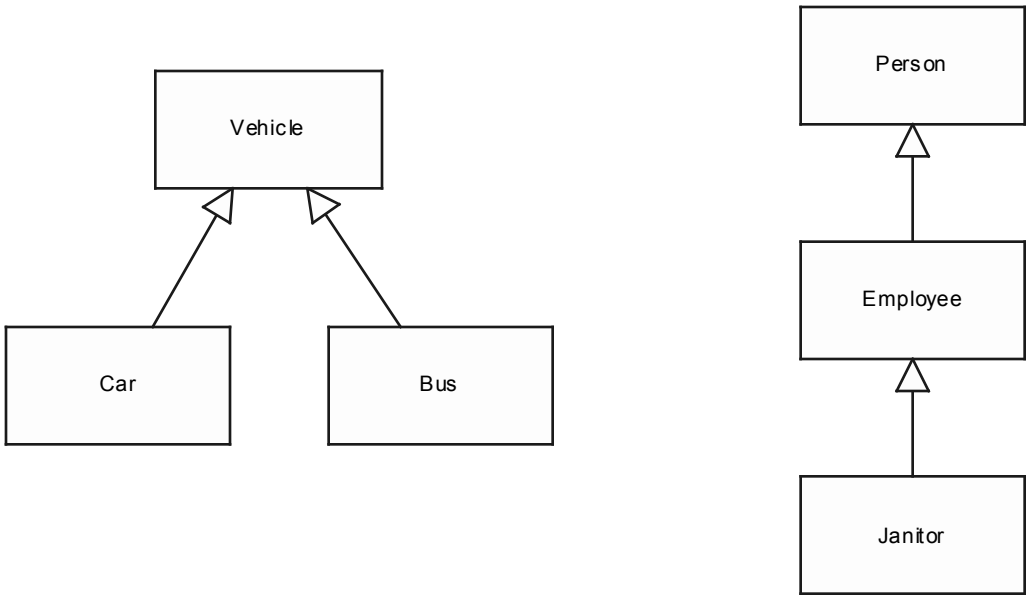
- What is Inheritance?
- Inheritance versus Composition
- Polymorphism

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

What is Inheritance?

- In the real world, and in software systems, many things are "a kind of" another, more general thing



The notion of entities being a more specific thing than others of a similar type is common both in the real world and in software systems.

In a software system, we can model the "is-a" or "is-a-kind-of" relationship using inheritance. Inheritance is really just another kind of relationship in addition to the associations covered earlier.

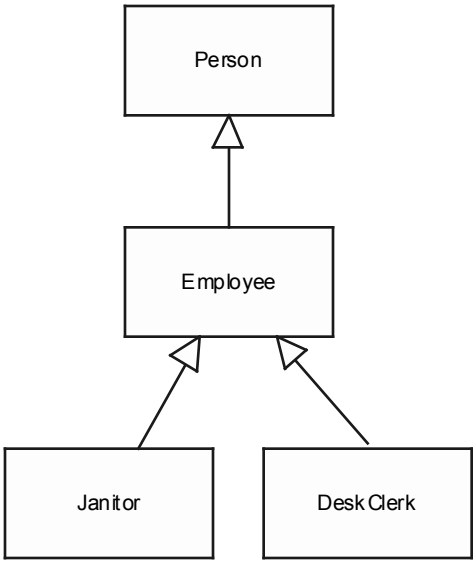
The UML symbol for inheritance is a line with a broad-headed arrowhead. The "pointed-to" class is referred to as the superclass, while the other class is called the subclass.

Also remember that we model classes in UML, but what really counts in the relationships between objects of the classes.

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 Inheritance?

- If you can discern "is a kind of" relationships in your analysis, you can use inheritance to make your objects more reusable and extensible



There a couple of key benefits to using inheritance:

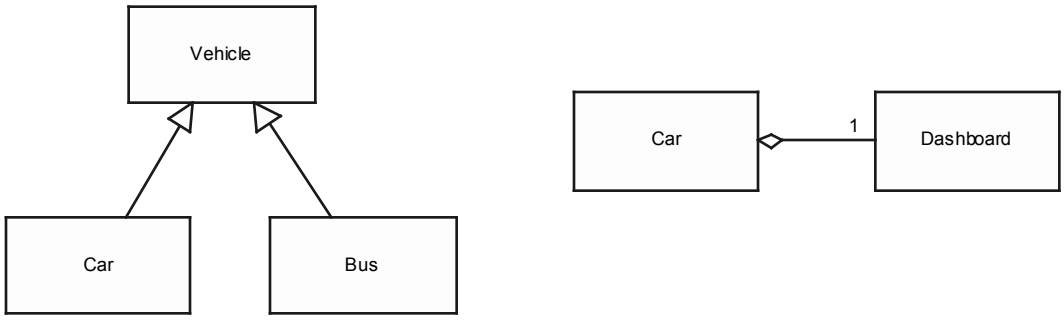
1. Inheritance lets you create models that closely approximate the real-world entity upon which the model is based. Lowering the so-called "representational gap" makes your software easier to understand and develop.

2. When combined with polymorphism (covered later), inheritance lets you create systems that are easy to extend without risking breakage to existing code.

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

Inheritance Vs Composed-Of Relationships

- We have seen classes that have aggregation or composition relationships with other classes
- You can generally determine this kind of relationship using the phrases "is a" (inheritance) versus "has a" (aggregation)

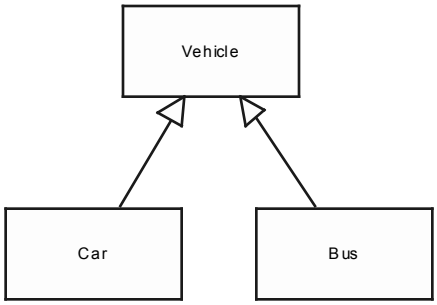


Sometimes it's difficult to determine whether to use inheritance or aggregation to implement a relationship, but you can use the "is-a" and "has-a" phrases as a quick test.

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

Java and Inheritance

```
1    public class Vehicle
2    {
3    }
4
5    public class Car extends Vehicle
6    {
7    }
8
9    public class Bus extends Vehicle
10   {
11   }
```

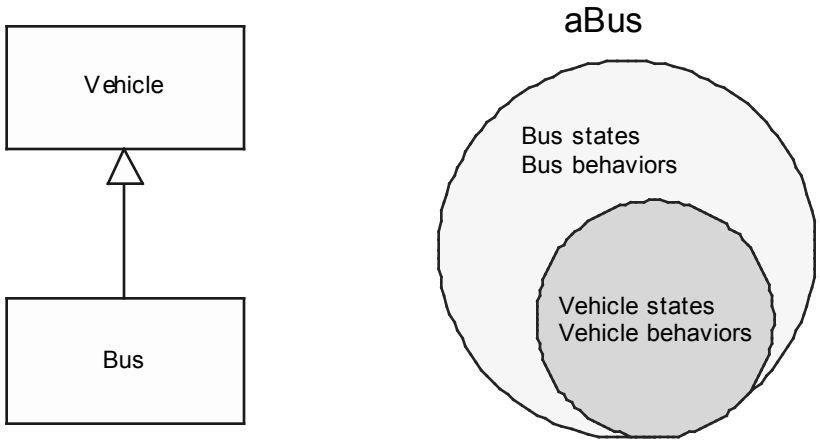


Java uses the keyword "extends" to implement an inheritance relationship.

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

Subclass Objects are a Superset

- Once instantiated, an object of a subclass type has all of the behaviors and states of its superclass in addition to any additional features defined by the subclass



10 - 6

This is the very basis of inheritance -- it allows you to create successively more refined and concrete classes that augment or replace states and behaviors from the superclass.

In this example, the additional state of a Bus might include the number of seats, whether the Bus has overhead storage, 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

Accessing Superclass State and Behavior

- In most object-oriented programming languages, features marked as private are not accessible to subclasses
- That improves encapsulation and reduces the risk that changes in the superclass break subclasses

```
1    public class Vehicle
2    {
3      private int weight;
4    }
5
6    public class Bus extends Vehicle
7    {
8      public void myMethod()
9      {
10        System.out.println(weight); // ERROR!
11     }
12   }
```


While it might seem weird that a subclass method cannot directly access states and behaviors defined in a superclass, the restriction actually makes for less brittle software. If a subclass can directly access features in a superclass, then the subclass might need to be changed if we change the feature in the superclass. That's unacceptable, especially considering that it's common for a different programmer to code the superclass and the subclasses.

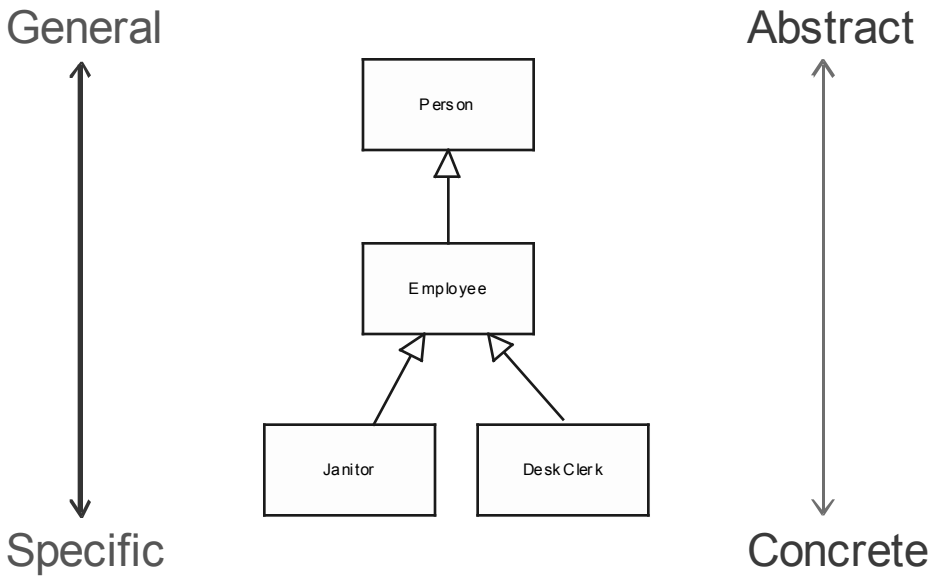
So how would the Bus class access the Vehicle weight? The common practice is for the superclass to define public "get/set" methods that give controlled access to private members.

The downside of using "get/set" methods is that they are slower than direct access. So many object-oriented languages define an access level named "protected", that allows subclasses direct access. Protected access thus falls between private and public and provides weaker encapsulation, thus decreasing the malleability of the system, but perhaps increasing performance.

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

Building Class Hierarchies

- In a typical hierarchy, the superclasses are more general and less concrete than the subclasses

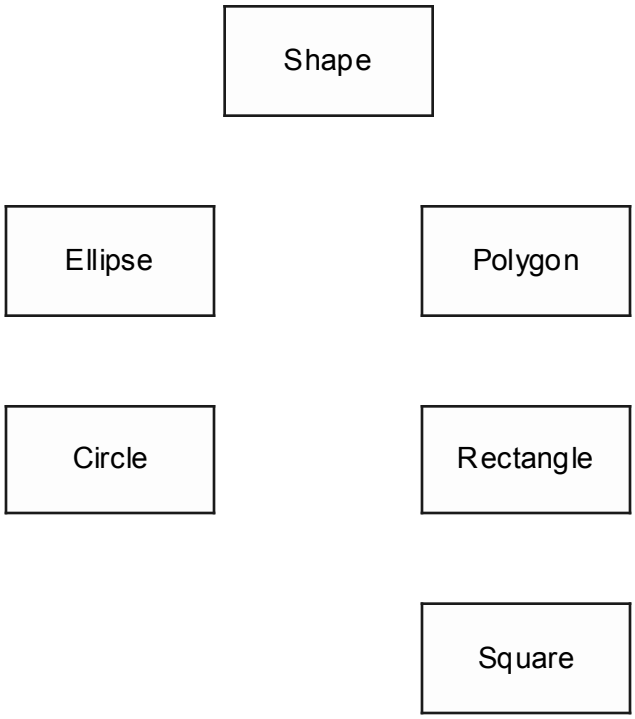


As you define subclasses, you add states and behaviors so that classes at the "bottom" of the hierarchy are more "real" than classes at the top.

As you design a hierarchy, you often "factor out" common states and behaviors and define them in a common superclass. This "factoring out" leads to designs that are easier to maintain, since if you need to change something, you only need to change it one place.

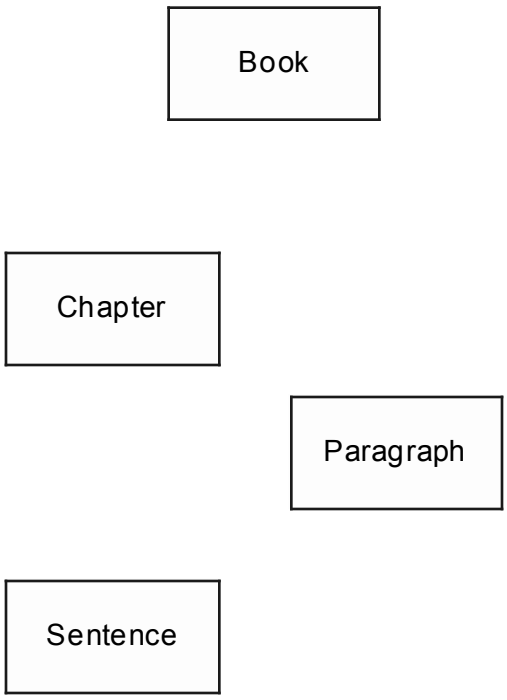
Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quiz: What Kind of Relationship?



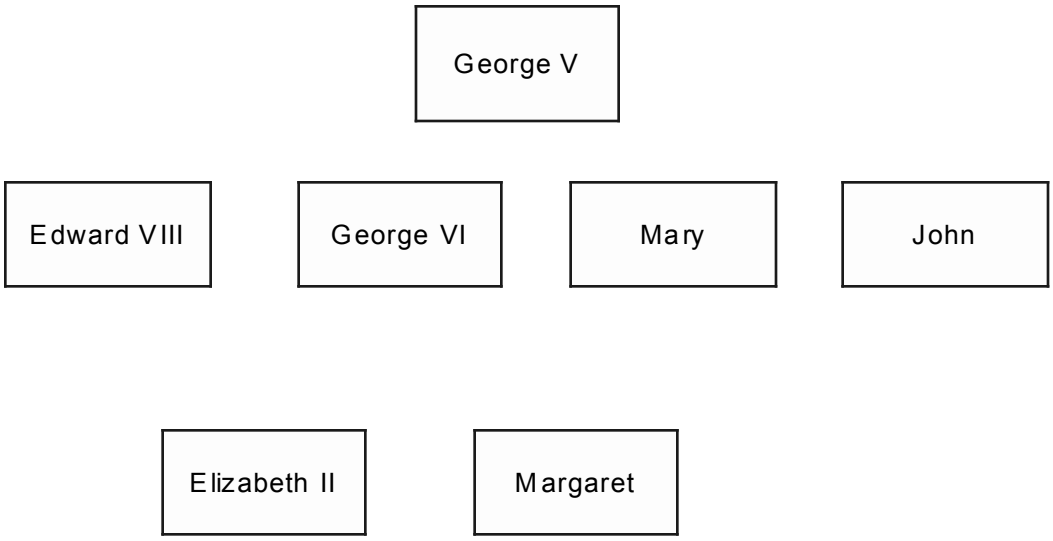
Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quiz: What Kind of Relationship?



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

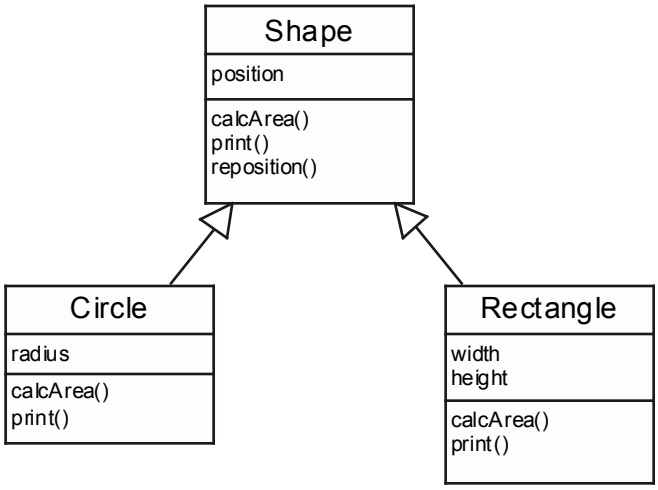
Quiz: What Kind of Relationship?



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

Overriding Behaviors

- If a subclass provides a method with the same name and arguments as in the superclass, the subclass's method overrides the superclass method
- Overriding lets subclasses modify behaviors provided by superclasses



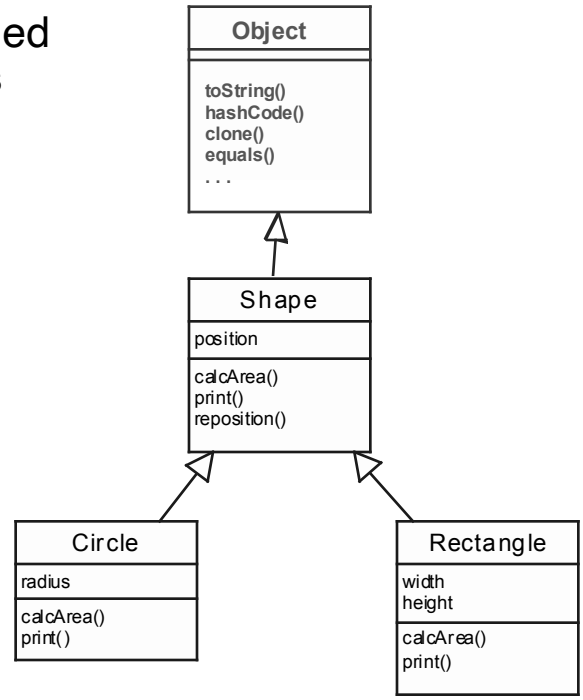
As we've already seen, a subclass can provide additional states and behaviors. A subclass can also CHANGE inherited behaviors by providing a method with the same name and arguments (method signature). When you create an instance of the subclass and call the method, the system invokes the method implementation defined in the subclass.

There are basically three things you can do in an overriden method: 1. Completely replace the implementation provided by the superclass(es). 2. Augment behavior provided by the superclass(es). In other words, do additional work, and also call back to the superclass implementation. 3. Completely remove the behavior by writing an empty method in the subclass.

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

The Object Class

- In Java, if you don't specify a superclass, Java uses the predefined **java.lang.Object** class
- All Java classes ultimately derive from Object



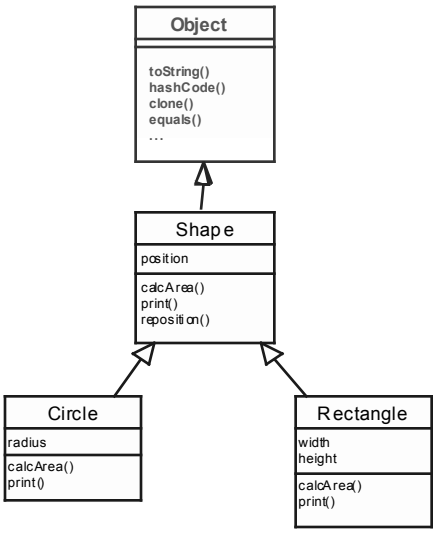
Object represents a generic Java object and provides the basic behavior required by all objects in Java.

The Object class is the only class that does not have superclass.

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

```
1    int area = 0;
2    Point pos = new Point(2,3);
3
4    Shape s = new Shape();
5    Circle c = new Circle();
6    Rectangle r = new Rectangle();
7
8    s.reposition(pos);    // OK?
9    c.reposition(pos);    // OK?
10   r.reposition(pos);    // OK?
11
12   area = s.calcArea();  // OK?
13   area = c.calcArea();  // OK?
14   area = r.calcArea();  // OK?
15
16   String str = c.toString(); // OK?
```



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

Constructors and Superclasses

- Every constructor in a subclass must call a superclass constructor as its first line
- If you omit the call to the superclass, Java will insert it for you

```
1    public class Circle extends Shape
2    {
3       public Circle()
4       {
5          super();
6       }
7       public Circle(double radius)
8       {
9          System.out.println("1-arg ctor");
10      }
11      public Circle(Point position)
12      {
13         super(position);
14      }
15   }
```


Since constructors often have the job of initializing a class's fields, it's important that superclasses have the opportunity to initialize fields too. Therefore, Java requires that all constructors in subclasses, as their first executable line, invoke a constructor in the superclass using the super() keyword.

The subclass constructor can invoke ANY superclass constructor, not just a superclass constructor that has a signature matching the subclass constructor.

If you omit the call to super(), then Java automatically inserts a call to the superclass's no-argument constructor before any other executable statements in the subclass constructor.

If you explicitly call super(), the super() statement must be the first executable line or else the compiler will flag an error.

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

Constructors and Superclasses, cont'd

- Where's the compile error in this code?

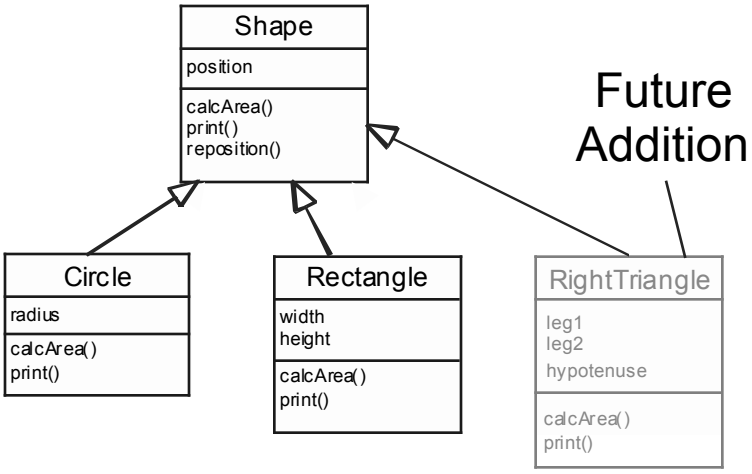
```
1    public class Shape
2    {
3       public Shape(Point position)
4       {
5       }
6    }
7
8    public class Rectangle extends Shape
9    {
10      public Rectangle()
11      {
12        System.out.println ("no-arg ctor");
13      }
14   }
```

This code will not compile since the Rectangle subclass's constructor doesn't have an explicit super() -- therefore the compiler assumes the no-argument constructor. Since the superclass (Shape) doesn't have a no-argument constructor, the compiler flags an error.

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

Polymorphism

- Polymorphism lets you design extensible, malleable systems to which you can add new classes without breaking existing code
- If you find yourself writing programs that use "if-else if-else" statements to determine an object's type, you probably are not using polymorphism correctly



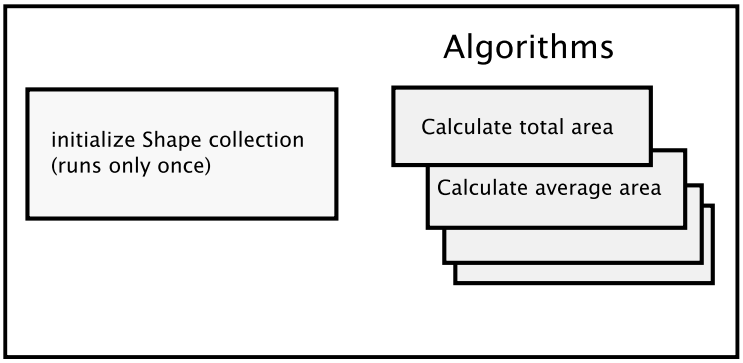
The idea behind polymorphism is that we often need to extend existing systems by adding new functionality and classes. Ideally, if we add a new class, we don't want the addition to require us to change any existing code.

The issue is that if our existing code uses some sort of "if" or "switch-case" statement to perform operations based on an object's type, then that code will need to be updated if we add a new class to the system. That's what polymorphism helps us avoid.

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

A Typical App

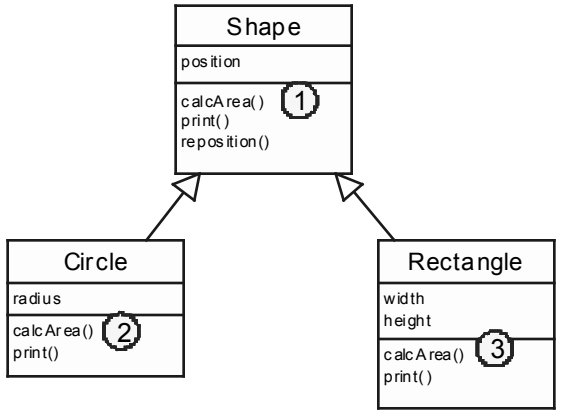
- Applications often initialize collections of objects, perhaps from a flat file, database or via user input
- The app might have numerous **algorithms** that process the collection, performing different calculations or operations (e.g. print)
- Polymorphism will make it easier to maintain such programs, eliminating the need to modify the algorithms if we add a new class to the hierarchy



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

Polymorphic Reference Assignment

```
1    Shape p1 = new Shape();           // OK?
2    Circle p2 = new Circle();         // OK?
3    Rectangle p3 = new Rectangle();   // OK?
4
5    Shape p4 = new Circle();     // OK?
6    Shape p5 = new Rectangle(); // OK?
7    Circle p6 = new Shape();     // OK?
8
9    // which implementation runs?
10   p1.calcArea();
11   p2.calcArea();
12   p3.calcArea();
13   p4.calcArea();
14   p5.calcArea();
15   p4 = p5;
16   p4.calcArea();
```



Java allows you to assign references to subclass objects to variables typed as a superclass reference.

When you invoke a method on a reference, Java runs the subclass implementation. That means that at runtime, the JVM must examine the object's type, not the type of the reference. This is referred to as "late binding", since the chosen implementation is determined at runtime rather than at compile time.

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

Writing Polymorphic Algorithims

- Using polymorphism, you can write algorithms that work without modification even if you add new subclasses
- If you find yourself writing programs that use "if" statements to determine an object's type, you probably are not using polymorphism correctly

```
1    // initialization
2    ArrayList<Shape> shapes =
3       new ArrayList<Shape>();
4    shapes.add(new Circle());
5    shapes.add(new Rectangle());
6
7    // an algorithm
8    double totalArea=0;
9    for (Shape s : shapes)
10   {
11     totalArea += s.calcArea();
12   }
```


The code shown here creates a Java ArrayList collection and then stores a few Circle and Square objects in the collection. Note that Circle and Square are subclasses of Shape.

Then to calculate the total area, we iterate (loop) through the collection, calling the calcArea() method on each Shape.

The key is that if we defined a new Shape, say a RightTriangle, the algorithm would still work the same! In other words, since all Shapes provide an overriden calcArea() method, our code doesn't break if we add a new kind of Shape. That's what polymorphism 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

Using Abstract Classes

- An abstract class cannot be instantiated, but can be subclassed
- Any class that contains any abstract methods must itself be marked as abstract

```
1    public abstract class Shape
2    {
3       private Point position;
4
5       public abstract double calcArea();
6       public void print()
7       {
8          System.out.println("Pos: " + position);
9       }
10   }
```



An abstract method consists of a method definition with no body (i.e. no open and close curly braces). Any concrete subclass must provide an implementation of the method, or else the compiler will flag an error.

An abstract class can define data and can have non-abstract methods.

Classes with abstract methods let you define partially implemented classes high in a class hierarchy and force subclasses to provide required behaviors.

The chief differences between an abstract class and an interface is that interfaces can contain no code at all and that any given class can implement multiple interfaces, but can extend only one class.

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

- A field marked as **final** is a constant
- A method marked as **final** in a superclass cannot be overriden in a subclass
- A class marked as **final** cannot be subclassed

```
1    public class A
2    {
3       public final static double avogadro = 6.0221409e+23;
4
5       public final void myMethod()
6       {
7       }
8    }
9
10   public final class B
11   {
12   }
```

The author of a superclass can use the "final" keyword to prevent subclassers from overriding methods, or from even subclassing at all.

Note that instead of using final constants to define related values, you might consider using an "enum" instead.

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 **Policy** that represents an insurance policy. The class should have state for the policy number and amount. Then write a subclass named **AutoPolicy** that has additional state for the covered car model (String).

- If you have time, write constructors in each of the classes. The subclass constructor should invoke the superclass constructor.

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

Review Questions

- Why is good that subclass methods cannot access private state and behaviors from superclasses?
- What is the major benefit of polymorphism?

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 benefits of inheritance and class hierarchies
- The basics of using polymorphism

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

- Manipulating Strings
- Working with Dates and Times
- BigDecimal for financial calculations

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 Class Library

- The Java Virtual Machine (JVM) includes thousands of prewritten classes referred to as the **Java Class Library**
- This chapter examines a few of the commonly used classes in the library

The class library is organized into packages, such as java.io, java.net 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

Java Strings

- The **java.lang.String** class represents strings of **Unicode** characters
- The String class defines numerous useful methods for manipulating strings

```
1    String s1 = "The quick brown fox";
2
3    String s6 = new String();
4    String s7 = new String(s1);
5    String s8 = new String (new StringBuffer());
6    char[] ca = {'a', 'b', 'c'};
7    String s9 = new String (ca);
```


Unicode is a specification that assigns a unique number (code point) to most of the world's characters. Associated with Unicode, there are several encoding schemes, or "charsets", that define how to map Unicode code points onto binary values. Examples include US-ASCII, UTF-8, UTF-16, and windows-1252 (Latin I), which is the default charset on US versions of Microsoft Windows.

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

String versus StringBuffer/StringBuilder

- String, StringBuffer and StringBuilder are classes defined in the **java.lang** package
- StringBuilder and StringBuffer have the same API, but StringBuilder isn't thread-safe
- All of these classes let you work with sets of Unicode characters

	String	StringBuffer
Mutable	No	Yes
Accepted by most library methods	Yes	No
Recognized by the JVM	Yes	No

Many, if not most, Java programs will need to work with character data. For maximum flexibility, Java provides two classes that let you manipulate character strings.

Note that Java strings use the Unicode encoding scheme, which can represent most of the world's characters. For more information on Unicode, go to:

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

Strings are Immutable

- Once created, a String cannot be modified
- The "+" operator applied to Strings actually creates a new String

```
1    String s3 = "abcd";
2    String s4 = s3;
3    s3 += "efgh";
4    System.out.println(s3 == s4); // false
5
6    // how many objects?
7    String s5 = "";
8    for (int i=0; i<100; i++)
9       s5 += i;
```


By design, Java String objects cannot be modified after creation. That makes them thread safe and guarantees that there's no issues with cloning objects that contain Strings. You can verify that Strings are immutable by looking at the JavaDoc -- there are no methods that actually modify the String itself. For example the toUpperCase() method doesn't convert the String -- it returns a new String with all upper-case characters.

The String concatenation operator is convenient, but can sap performance as shown in the second chunk of code, which creates a new String object each time through the loop. Since creating new objects is fairly performance intensive, the loop is very inefficient. Using a StringBuffer would be a better choice.

Note that modern JVMs may actually optimize the the concatenation operator to use a StringBuilder in the generated byte code, which is more efficient as we will see 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

Searching a String: Finding Substrings

```
1    String s10 = "A countryman between two lawyers" +
2       " is like a fish between two cats.";
3
4    int i6 = s10.indexOf("two");
5    int i7 = s10.lastIndexOf("two");
6    int i8 = s10.indexOf("two", 22);
7
8    System.out.println(i6 + " " + i7 +
9           " " + i8); // 21 56 56
```

```
012345678901234567890123456789012345678901234567890123456789012345678901234
A countryman between two lawyers is like a fish between two cats.
```


The String class provides methods to search for substrings, starting the search either from the beginning or end of the String. You can also specify a starting index in either case.

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

Converting Case

- Though Strings are immutable, you can create a copy of a String with all upper or lower-case characters

```
1    String s1 = "The quick brown fox";
2
3    String s13 = s1.toLowerCase();
4    System.out.println(s13);
5    String s14 = s1.toUpperCase();
6    System.out.println(s14);
7
8    s1 = s1.toUpperCase();
```


The last line shows that you can re-use a String reference -- that line actually creates a new String and makes the original String un-referenced.

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

Parsing Strings into Primitives

- To parse Strings into primitive values, you can use the so-called *wrapper* classes

```
1    String s19 = "3.141592653589793";
2    double d1 = Double.parseDouble(s19);
3
4    double d2 = d1 * 2;
5    System.out.println(d2); // 6.283185307179586
6
7    String s20 = "abc";
8    int i9 =
9      Integer.parseInt(s20);  // NumberFormatException!
```


Strings are useful for text manipulation, but you cannot use math operators on Strings. So if your program has a String, perhaps via user input, that you believe is formatted numerically, you can use static methods on the "wrapper" classes to convert.

There is a separate "wrapper" class for each primitive type. And since they are classes, their names start with an upper-case first character as is conventional.

Note that parsing can throw a NumberFormatException if the input String is not formatted correctly for the attempted conversion. You can of course handle this with a try-catch block.

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

StringBuffer/StringBuilder are Mutable

- Unlike Strings, you can modify StringBuffers and StringBuilders as you see fit
- It's easy to convert a StringBuffer or StringBuilder to/from a String

```
1    String s1 = "The quick brown fox";
2
3    StringBuffer sb1 = new StringBuffer(s1);
4    sb1.append('y');
5    String s2 = sb1.toString();
6    System.out.println(s2); // The quick brown foxy
7
8    // how many objects?
9    StringBuffer sb2 = new StringBuffer();
10   for (int i=0; i<100; i++)
11       sb2.append(i);
12   String s3 = sb2.toString();
```


StringBuffers cannot be used as widely as Strings, but you can modify the characters in a StringBuffer. And as shown here, you can use StringBuffer constructors and the toString() method to convert between String and StringBuffer.

The second chunk of code is far more efficient than the equivalent code shown earlier that used the String concatenation operator. It runs much faster since it doesn't create a new object each time through the 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 java.util.Date Class

- The **java.util.Date** class has been in Java since the beginning and represent a date/time

```
1    // current time/date
2    java.util.Date date = new java.util.Date();
```


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

Parsing Date Strings

- The **java.text** package contains several classes that allow you to convert Date objects to/from String

```
1    DateFormat df = new SimpleDateFormat ( "yyyy-MM-dd" );
2    String s21 = "2005-03-01";
3    String s22 = "2005-03-03";
4    try
5    {
6        Date da1 = df.parse(s21);
7        Date da2 = df.parse(s22);
8        boolean b4 = da1.before(da2);
9        System.out.println(b4);
10   }
11   catch (ParseException e)
12   {
13       e.printStackTrace();
14   }
```

Working with date strings is challenging because there are several common formats for date strings and because the formats vary across cultures. Java has good support for internationalization (I18N), however. Please look in the java.text package for more details on date, time and currency formatting strings.

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

Java 8 Date/Time API

- The Java date/time/calendar API has long been tedious and difficult to use
- Java 8 introduces the **java.time** package that includes classes such as **LocalDate** and **LocalTime**

```
1    LocalDate dob = LocalDate.of(1994, 8, 27);
2    System.out.println(dob);
3
4    LocalTime appointment = LocalTime.of(13, 30);
5    System.out.println(appointment);
```

The Java 8 Date/Time API was inspired by the popular Joda Time library.

This new API was introduced as JSR-310 before being incorporated into Java 8.

Unlike the old java.util.Date and Calendar, objects of these classes are immutable, and thus are thread-safe.

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

The BigDecimal Class

- Java provides primitives for floating point: **float** and **double**, but arithmetic with primitives can lead to unexpected round-off errors, especially in large financial applications
- Java provides the **java.math.BigDecimal** class that gives you complete control over rounding behavior

The real issue with floating point primitives is that it's impossible for them to represent irrational real numbers - for example, consider the value PI - its double value is an approximation of the actual value. And if you perform a large number of calculations involving such numbers, rounding errors can accumulate to the point at which they are significant, especially for financial calcuations.

The classic movie "Office Space" has a plot point in which the protaganists use round-off errors to embezzle money from their employer.

See the JavaDocs for complete details:

https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

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

The BigDecimal Class, cont'd

```
1    BigDecimal quantity = new BigDecimal("3");
2    BigDecimal unitPrice = new BigDecimal("12.84");
3    BigDecimal discountRate = new BigDecimal("0.12");
4    BigDecimal taxRate = new BigDecimal("0.06875");
5
6    BigDecimal amount = quantity.multiply(unitPrice);
7    BigDecimal discount = amount.multiply(discountRate);
8    BigDecimal discountedAmount = amount.subtract(discount);
9    BigDecimal tax = discountedAmount.multiply(taxRate);
10   BigDecimal total = discountedAmount.add(tax);
11   System.out.println(total); // 36.228060000
12
13   BigDecimal roundedTotal = total.setScale(
14      2, RoundingMode.HALF_EVEN);
15   System.out.println(roundedTotal);  // 36.23
```

This example adapted from excellent article at:

https://www.baeldung.com/java-bigdecimal-biginteger

Note how we use a BigDecimal constructor that accepts a String - this provides the best possible precision when creating the BigDecimal. If you need to convert a double or float to BigDecimal, you should use the valueOf() static method which internally first converts the double to a string:

```
BigDecimal pi = BigDecimal.valueOf(Math.PI);
```

The HALF_EVEN rounding rule rounds in the manner required by the IRS for tax returns:

Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor. Behaves as for ROUND_HALF_UP if the digit to the left of the discarded fraction is odd; behaves as for ROUND_HALF_DOWN if it's even. Note that this is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations.

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 a few of the classes in the standard Java class library

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 JUnit

- What is JUnit?
- JUnit Fundamentals

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

Types of Testing

- Unit Testing
- Integration Testing
- Acceptance Testing

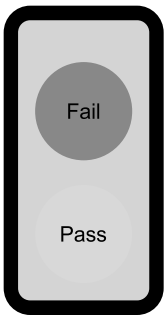
There are many kinds of testing to ensure program correctness. Unit testing checks the correctness of an individual part of a program, generally a class in OO-based programs. Integration testing tests how the pieces of the program work together as a whole. Regression testing lets you determine if changes to the code break it and acceptance testing lets you verify that the end-user approves the program.

This chapter focuses on unit testing.

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 Unit Testing?

- Unit tests are generally written by the developer writing the code under test
- Unit tests should be easy to write, fully automated and generate a boolean result (pass or fail)



Unit testing is a low-level test of a small part of a program's correctness. Since they are typically written by the developer, it's important that they be easy to write and run or else developers will tend not to write them, especially when time is short.

In addition, it's nice to have a unit-testing framework in place before beginning development so that you can write the unit tests as you go.

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

Benefits of Unit Testing

- Higher quality code
- Higher confidence in the code's correctness
- Higher confidence that changes or additions do not break existing code
- Unit testing is an integral part of **test-driven development** (TDD)

Writing unit tests incrementally saves a big job at the end of a project. This is why many "extreme programming" methodologies advocate unit testing as part of the process.

Refactoring is the act of reorganizing your code (e.g. factoring common code segments into a method). There is a growing body of work on refactoring - see the "Refactoring: Improving the Design of Existing Code" book by Martin Fowler, et. al.

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

Terminology

- **Object Under Test or System Under Test** - the code being tested
- **Test Case** - the code performing the test
- **Test Suite** - a collection of test cases

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 JUnit

- Overview statement from www.junit.org:

  JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java. JUnit is Open Source Software, release under IBM's Common Public License and is hosted on SourceForge



The notion of unit testing goes beyond Java - there are other xUnit frameworks available for other languages and technologies (e.g. HttpUnit).

Erich Gamma is well known for his work on Design Patterns (he is one of the "Gang of Four").

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 JUnit

- You can read about how JUnit came about at http://www.martinfowler.com/bliki/Xunit.html
- As of this writing, version 4.12 is the current version, with version 5 on the horizon
- JUnit versions 4.0 and newer require a Java 5 virtual machine and support **annotations**

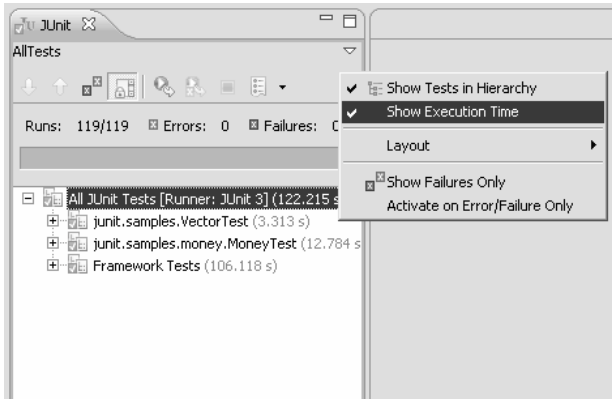
Annotations are a Java 5 technique to provide metadata within Java code. For example, there's a standard annotation, @SuppressWarning to tell the compiler not to issue a particular warning message. In additions, applications like JUnit can define their own custom annotations and then process them at runtime.

In this chapter, we will mostly cover JUnit 4 with a few backward looks at JUnit 3.

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

IDE Integration

- Most modern integrated development environments provide built-in support for JUnit



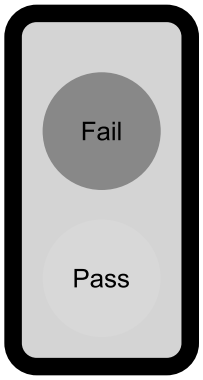
The screenshot shows the Eclipse IDE's JUnit GUI. Note the "green bar" that indicates that all of the tests passed.

Eclipse also ships with the JUnit JAR files and provides "wizards" to create tests and run them.

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

Steps for Using JUnit

1. Start writing the class under test
2. Write a test-case class
3. In test-case class, write methods to test methods in the class under test
4. Run the tests
5. Repeat



In JUnit 3, your testing class must be a subclass of the JUnit TestCase superclass.

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

Sample Class Under Test

```
1    public class CreditCard
2    {
3      private String cardNo;
4
5      public void setCardNumber(String s)
6      {
7        cardNo = s;
8      }
9
10     // implement the Luhn formula
11     public boolean isValid()
12     {
13         . . .
14     }
15   }
```


Here we show a fragment of a class under test (CUT).

The class validates credit-card numbers according to the Luhn algorithm. See http://en.wikipedia.org/wiki/Luhn_algorithm for details.

Note that it defines a method named isValid that actually checks the card number stored in the cardNo field.

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 Test

- Write a class that will act as the unit-test class - many developers name the class based on the class under test's name
- Write one or more methods annotated with JUnit annotations such as **@Test**

JUnit v3 requires that the unit-test class subclass a JUnit superclass and testing methods follow a naming convention, but JUnit v4 relaxes those rules.

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

JUnit Assertions

- In a testing method, you typically **assert** that some condition is met
- JUnit provides various assertXxxx methods for this purpose
- We consider a test run to be successful only if all of the assertions pass

junit.framework.Assert
assertEquals(msg:String, expected:Object, actual:Object):void assertEquals(msg:String, expected:String, actual:String):void assertEquals(msg:String, expected:boolean, actual:boolean):void . . . assertEquals(expected:boolean, actual:boolean):void assertEquals(expected:byte, actual:byte):void . . . assertFalse(msg:String, condition:boolean):void assertFalse(condition:boolean):void assertNotNull(obj:Object):void assertNotNull(msg:String, obj:Object):void assertNotSame(expected:Object, actual:Object):void assertNotSame(msg:String, expected:Object, actual:Object):void assertNull(obj:Object):void assertNull(msg:String, obj:Object):void assertSame(expected:Object, actual:Object):void assertSame(msg:String, expected:Object, actual:Object):void assertTrue(msg:String, condition:boolean):void assertTrue(condition:boolean):void fail():void fail(msg:String):void . . .

This is the fundamental working part of JUnit: "assert" methods that check some condition in the class under test.

A given testing class will typically perform many assertions. Only if ALL of the assertions pass is the test considered successful.

Note that the "assert" methods are overloaded to allow different types of expected and actual values, as well as to provide an optional message string that is displayed if the assertion fails.

The difference between "assertSame()" and "assertEquals()" is that the former checks for equality using the == operator, while the latter calls the "equals()" method.

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

Sample Testing Class

```
 1    package test;
 2
 3    import static org.junit.Assert.*;
 4    import org.junit.Test;
 5
 6    public class CreditCardTest
 7    {
 8      @Test
 9      public void testIsValid()
10      {
11        CreditCard cc = new CreditCard();
12        cc.setCardNumber("49927398716");
13        assertTrue(cc.isValid());
14        cc.setCardNumber("49927398717");
15        assertFalse(cc.isValid());
16      }
17   }
12 - 13
```

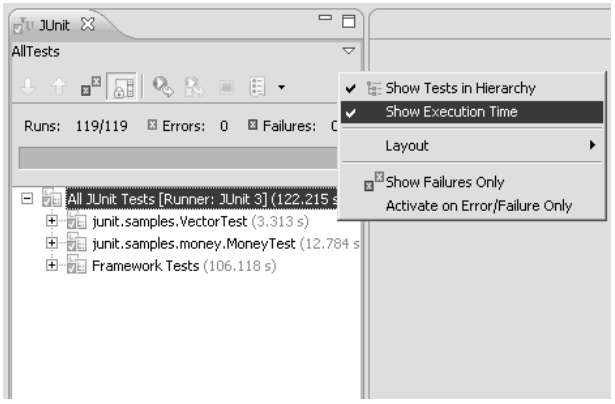
Here we show a JUnit 4 testing class for the CUT shown on the last page. Note the use of the @Test annotation on the "testIsValid()" method -- that marks that method so that the JUnit 4 runtime invokes the method when we run the test.

Note also how the "testIsValid()" method asserts the return from the "isValid()" method. Both of these assertions should pass, since 49927398716 is valid according to the Luhn formula, but 49927398717 is not (note the incorrect last digit).

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

Test Runners

- Many IDEs such as Eclipse provide JUnit integration



When JUnit 4 was written, they decided not to provide graphical test runners since most developers use the runner provided by their IDE.

If you really want to use a graphical test runner, you can put the JUnit 3 JAR in your CLASSPATH and hook up either the JUnit 3 Swing or AWT test runner. See

http://www.cavdar.net/2008/07/21/junit-4-in-60-seconds/

for details.

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

Setup and Tear Down Methods

- Often tests have common initialization and clean up requirements
- To avoid duplication, you can put the common code in "set up" and "tear down" methods

```
1    public class CreditCardTest
2    {
3      private CreditCard cc;
4
5      @Before
6      public void setUp()
7      {
8        cc = new CreditCard();
9      }
10
11     @Test
12     public void testIsValid()
13     {
14       cc.setCardNumber("49927398716");
15   . . .
```


You can use this technique to refactor your testing code itself and reduce duplication.

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 Test Fixture?

- Often tests depend on a system in some state, e.g. initialized objects and data structures
- We refer to this state as a **test fixture**
- You typically build the fixture in a "set up" method and perform any necessary clean up in a "tear down" method




```
public class CreditCardTest
{
  private CreditCard cc;

  @Before
  public void setUp()
  {
    cc = new CreditCard();
  }

  @Test
  public void testIsValid()
  {
    cc.setCardNumber("49927398716");
. . .
```


This example demonstrates a very simple test fixture that has only one object. Test fixtures can be arbitrarily complex.

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

Testing Exceptions

- If a method can throw an exception, you can test that behavior
- JUnit 4 provides the **expected** option on the @Test annotation for that purpose

Class Under Test

```
public boolean isValid()
  throws NumberFormatException
  {
    for (int i = 0;
         i < cardNo.length(); i++)
    {
      char ch = cardNo.charAt(i);
      if (!Character.isDigit(ch))
      {
        throw new
         NumberFormatException();
      }
    }
. . .
```

Testing Class

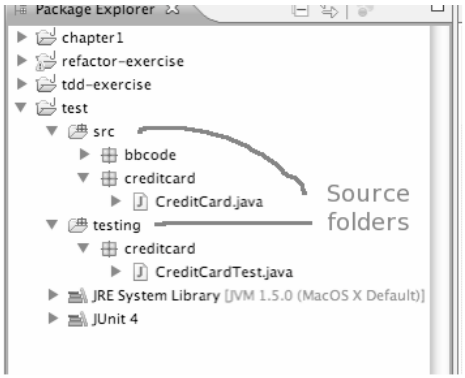
```
@Test(
  expected=
   java.lang.NumberFormatException.class)
public void testBadNumber()
{
  cc.setCardNumber("abc");
  cc.isValid();
}
```


Testing for expected exceptions is an important part of verifying correctness of Java code.

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

Organizing and Packaging Tests

- You typically do not want to ship the test case code with your deployed application
- To package the deployable code, you can use a scripting tool such as Apache Ant
- There's an advantage to having the test code in the same Java package as the code under test: can test *protected and package-scope* methods
- Many IDEs provide the notion of *source folders* that make it easier to segregate the test code from the deployable code



If you don't put your testing code in the same package as the code under test, your testing code will need to use reflection to access protected and package-scoped methods.

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 fundamentals of unit testing with JUnit
- How to use assertions
- How to organize your testing code

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

Java 8 to Java 11

- Java 8 Features
- Functional Programming
- Java 9 to 11 Overviews

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 8 Feature Overview

- Java 8 is a significant update to earlier versions of Java and provides several major additions, including:
  - **New Date and Time API** - Classes to make working with dates and times easier and safer
  - **Default interfaces methods** - Syntax to add concrete methods to interfaces
  - **Functional programming** - Syntax to make methods **first class citizens**
  - **Parallel processing** - Classes to make writing parallel algorithms easier
  - **JavaScript integration** - A new, more powerful JavaScript engine in the JVM so you can execute JavaScripts from a Java program

For a complete listing of new Java 8 features, see:
http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.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 8 Date/Time API

- The Java date/time/calendar API has long been tedious and difficult to use
- Java 8 introduces the **java.time** package that includes classes such as **LocalDate** and **LocalTime**

```
1    LocalDate dob = LocalDate.of(1994, 8, 27);
2    System.out.println(dob);
3
4    LocalTime appointment = LocalTime.of(13, 30);
5    System.out.println(appointment);
```


The Java 8 Date/Time API was inspired by the popular Joda Time library.

This new API was introduced as JSR-310 before being incorporated into Java 8.

Unlike the old java.util.Date and Calendar, objects of these classes are immutable, and thus are thread-safe.

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

Default Methods

- During the life of a project, sometimes it's useful to add new functionality to base interfaces
- Java 8 provides **default methods** also known as **extension methods** that let you add concrete methods to existing interfaces without breaking existing classes that implement the interfaces

MyInterface.java

MyClass.java

TestDefaultMethods.java

Java 8 also lets you define static methods in interfaces. As usual, static methods in interfaces are considered part of the interface itself in memory, not part of the instances of classes that implement the interface.

The Java 8 class library uses default and static methods to add new functionality to interfaces such as java.util.Comparator to support other new Java 8 functionality such as streams.

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

```
1    package defaultmethods;
2
3    public interface MyInterface
4    {
5        void method1();
6
7        default void method2()
8        {
9            System.out.println("Method 2");
10       }
11
12       static void method3()
13       {
14           System.out.println("Method 3");
15       }
16   }
```

13 - 4A : MyInterface.java

```
1    package defaultmethods;
2
3    public class MyClass implements MyInterface
4    {
5        @Override
6        public void method1()
7        {
8            System.out.println("Method 1");
9        }
10
11   }
```

13 - 4B : MyClass.java

```
1    package defaultmethods;
2
3    public class TestDefaultMethods
4    {
5        public static void main(String[] args)
6        {
7            MyClass m = new MyClass();
8
9            m.method1();
10           m.method2();
11           MyInterface.method3();
12       }
13   }
```

Introduction to Functional Programming

- **Functional programming** is a discipline in which you write programs that consist of functions that don't modify any external state, and thus have no **side effects**
- Functional programming also requires the ability to pass functions (methods) around in the same way we pass values
- Languages that provide functional programming features include **Lisp**, **Scheme**, **Haskell**, **Scala** and **Clojure**
- A complete discussion of functional programming is beyond the scope of this course, but we will cover **lambda expressions** and other Java 8 functional-programming support

Many other "imperative languages" like Java include support for functional techniques. For example, JavaScript provides several features that enable functional programming including anonymous functions and closures.

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

Passing Functions Around

- Java 8 provides the **method reference** syntax so you can pass existing methods as arguments to other other methods
- To reference a static method, use ClassName: :methodName
- To reference an instance method, use objectRefName: :methodName

```
1    File f = new File(".");
2
3    File[] files = f.listFiles(File::isHidden);
```


The ability to pass function (method) references around is sometimes referred to as making functions "first-class citizens" of the language. Prior to Java 8, there was no Java syntax to do this.

Note that there's also syntax to reference a constructors.

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

Passing Functions Around: The Old Way

- Prior to Java 8, developers often used **anonymous inner classes** with a single method, but that syntax is verbose and perhaps confusing

```
1    JButton jb = new JButton("press me");
2
3    // pass anonymous inner-class object as parameter
4    jb.addActionListener(new ActionListener()
5    {
6      public void actionPerformed(ActionEvent e)
7      {
8        System.out.println("I've been clicked!");
9      }
10   });
```


A JButton can call a method when the user clicks it, but prior to Java 8, there was no easy way to provide the method's reference. One approach that worked, but was clumsy, was to:

1. Use an interface with a single method (ActionListener)
2. Write a class that implements the interface (actionPerformed())
3. Create an object of that class
4. Pass the object's reference to the button

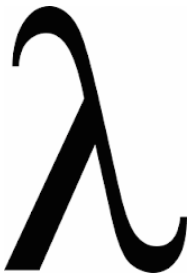
The code shown here implements the interface using a technique called an anonymous inner class. The syntax might seem rather obscure if you're not used to it.

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 Lambda Expression?

- **Lambda expressions** let you define anonymous functions (methods) that you can pass as parameters to other methods

```
1    JButton jb = new JButton("press me");
2
3    // use a lambda to implement the
4    // ActionListener interface
5    ActionListener al = (ae) -> System.out
6      .println("I've been clicked!");
7
8    jb.addActionListener(al);
```

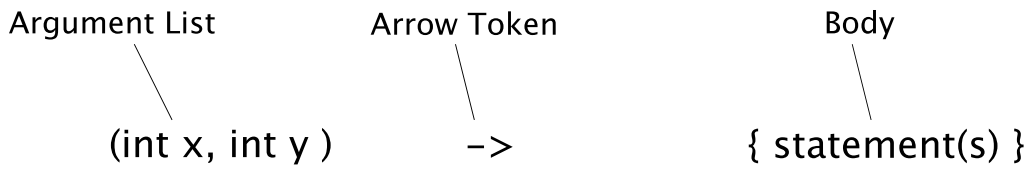


The term "lambda expression" originally came from mathematics, where the Greek letter lambda is used to define a mathematical function arguments.

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

Lambda Expression Syntax

- Lambda expressions consist of an **argument list**, an **arrow token** and a **body**




```
1    n -> n % 2 != 0
2    (char c) -> c == 'y'
3    (x, y) -> x + y
4    (int x, int y) -> x * x + y * y
5    () -> 42;
6    () -> { return 3.14; }
7    (String s) -> { System.out.println(s); }
8    s -> System.out.println(s);
9    () -> { System.out.println("Hello World!"); }
```


Note that the syntax shown here is just for the lambda - we don't show using the generated anonymous function. The complete example on the next page does, however.

In the argument list, you can omit the parameter types, and if there's exactly one parameter, you can omit the parenethis.

If there's only one statement in the body, you can omit the curly braces.

You can also omit the return statement.

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

Functional Interfaces

- A **functional interface** is an interface with a single method
- You can use functional interfaces as types of variables and parameters and then provide a lambda expression

LambdaSyntax.java

The examples in the following listing show the lambda expressions from the previous page, assigning them to variables whose type are functional interfaces.

Note that the java.awt.event.ActionListener we used a couple of pages ago is a functional interface with a single method named actionPerformed().

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

```
1    package lambdas;
2
3    public class LambdaSyntax
4    {
5        interface Functional1
6        {
7            boolean method1(int x);
8        }
9
10       interface Functional2
11       {
12           boolean method1(char c);
13       }
14
15       interface Functional3<T>
16       {
17           T method1(T a, T b);
18       }
19
20       interface Functional4
21       {
22           int method1(int a, int b);
23       }
24
25       interface Functional5
26       {
27           int method1();
28       }
29
30       interface Functional6
31       {
32           double method1();
33       }
34
35       interface Functional7
36       {
37           void method1(String s);
38       }
39
40       interface Functional8
41       {
42           void method1();
43       }
44
45       public static void main(String[] args)
```

```
46      {
47          Functional1 a = n -> n % 2 != 0;
48          Functional2 b = (char c) -> c == 'y';
49          Functional3<Integer> c = (x, y) -> x + y;
50          Functional4 d = (int x, int y) -> x * x + y * y;
51          Functional5 e = () -> 42;
52          Functional6 f = () -> { return 3.14; };
53          Functional7 g = (String s) -> { System.out.println(s); };
54          Functional7 h = s -> System.out.println(s);
55          Functional8 i = () -> { System.out.println("Hello World!"); };
56      }
57  }
```

Using Lambda Expressions Example

- In Java 8, **java.util.Comparator** can act as a functional interface so you can use lambdas to sort collections

Student.java

TestComparator.java

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

```java
package university;

import java.time.LocalDate;

public class Student implements Comparable<Student>
{
    private int studentID;
    private String name;
    private double gpa;
    private LocalDate dateOfBirth;

    public Student(int studentID, String name, double gpa, LocalDate dateOfBirth)
    {
        this.studentID = studentID;
        this.name = name;
        this.gpa = gpa;
        this.dateOfBirth = dateOfBirth;
    }

    @Override
    public int compareTo(Student o)
    {
        return this.name.compareTo(o.name);
    }

    public int getStudentID()
    {
        return studentID;
    }

    public void setStudentID(int studentID)
    {
        this.studentID = studentID;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

```

```
46      public double getGpa()
47      {
48          return gpa;
49      }
50
51      public void setGpa(double gpa)
52      {
53          this.gpa = gpa;
54      }
55
56      public LocalDate getDateOfBirth()
57      {
58          return dateOfBirth;
59      }
60
61      public void setDateOfBirth(LocalDate dateOfBirth)
62      {
63          this.dateOfBirth = dateOfBirth;
64      }
65
66      @Override
67      public String toString()
68      {
69          return "Student [studentID=" + studentID + ", name=" + name + ", gpa="
70                  + gpa + ", dateOfBirth=" + dateOfBirth + "]";
71      }
72
73      @Override
74      public int hashCode()
75      {
76          final int prime = 31;
77          int result = 1;
78          result = prime * result + studentID;
79          return result;
80      }
81
82      @Override
83      public boolean equals(Object obj)
84      {
85          if (this == obj)
86              return true;
87          if (obj == null)
88              return false;
89          if (getClass() != obj.getClass())
90              return false;
```

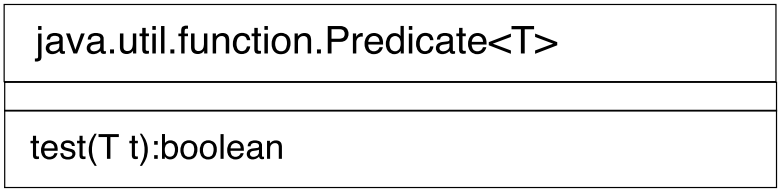
```
91            Student other = (Student) obj;
92            if (studentID != other.studentID)
93                return false;
94            return true;
95        }
96    }
```

```
1    package lambdas;
2
3    import java.time.LocalDate;
4    import java.util.Comparator;
5    import java.util.SortedSet;
6    import java.util.TreeSet;
7
8    import university.Student;
9
10   public class TestComparator
11   {
12       private static SortedSet<Student> students = new TreeSet<Student>();
13
14       private static void init()
15       {
16           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
17                   12)));
18           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
19                   .of(1992, 6, 22)));
20           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
21                   18)));
22           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
23                   1994, 8, 27)));
24       }
25
26       public static void main(String[] args)
27       {
28           init();
29
30           // print using natural sort order
31           System.out.println("Natural order");
32           System.out.println("-------------");
33           for (Student s : students)
34               System.out.println(s);
35           System.out.println();
36
37           // print in descending GPA using a lambda
38           Comparator<Student> sorter = (Student s1, Student s2) -> {
39               int retVal = Double.compare(s1.getGpa(), s2.getGpa());
40               return retVal;
41           };
42           SortedSet<Student> sorted = new TreeSet<Student>(sorter);
43           sorted.addAll(students);
44           System.out.println("Sorted by GPA");
45           System.out.println("-------------");
```

```
46            for (Student s : sorted)
47                System.out.println(s);
48            System.out.println();
49        }
50    }
```

The java.util.function Package

- Java 8 introduces the **java.util.function** package that contains functional interfaces that are useful with lambda expressions
- One such interface is **Predicate**, which contains a single method, **test** that accepts an object and returns a boolean
- Predicates typically act as a filter for objects in a collection



The java.util.function.Predicate interface actually has a few more methods (e.g and(), or() and negate()), but they are default or static methods and thus don't violate the requirement that functional interfaces have a single method.

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

Predicate Example

- This example uses Predicates to process various subsets of a collection

TestPredicate.java

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

```
1    package lambdas;
2
3    import java.time.LocalDate;
4    import java.util.SortedSet;
5    import java.util.TreeSet;
6    import java.util.function.Predicate;
7
8    import university.Student;
9
10   public class TestPredicate
11   {
12       private static SortedSet<Student> students = new TreeSet<Student>();
13
14       private static void init()
15       {
16           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
17                   12)));
18           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
19                   .of(1992, 6, 22)));
20           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
21                   18)));
22           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
23                   1994, 8, 27)));
24       }
25
26       public static void print(Predicate<Student> p)
27       {
28           for (Student s : students)
29           {
30               if (p.test(s))
31                   System.out.println(s);
32           }
33       }
34
35       public static void main(String[] args)
36       {
37           init();
38
39           Predicate<Student> highGpa = aStudent -> aStudent.getGpa() > 3.4;
40           Predicate<Student> youngster = aStudent -> aStudent.getDateOfBirth()
41                   .getYear() >= 1994;
42
43           // print all high GPA students
44           System.out.println("High GPA students");
45           System.out.println("-------------");
```

```
46          print(highGpa);
47          System.out.println();
48
49          // print all young students
50          System.out.println("Young students");
51          System.out.println("-------------");
52          print(youngster);
53
54      }
55
56  }
```

Introduction to Java 8 Streams

- **Streams** are sequences of elements that support operations such as filtering and sorting in a declarative fashion akin to SQL
- Streams support both sequential and parallel operations
- You can retrieve a stream from any collection that implements java.util.Collection

Parallel operations make efficient use of multiple processors, transparently running operations in multiple threads.

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

Stream Example

- Many Stream methods return a stream so you can chain method calls

TestStream.java

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

```
1    package streams;
2
3    import static java.util.Comparator.comparing;
4
5    import java.time.LocalDate;
6    import java.util.List;
7    import java.util.SortedSet;
8    import java.util.TreeSet;
9    import java.util.stream.Collectors;
10
11   import university.Student;
12
13   public class TestStream
14   {
15       private static SortedSet<Student> students = new TreeSet<Student>();
16
17       private static void init()
18       {
19           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
20                   12)));
21           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
22                   .of(1992, 6, 22)));
23           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
24                   18)));
25           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
26                   1994, 8, 27)));
27       }
28
29       public static void main(String[] args)
30       {
31           init();
32
33           // Comparator<Student> sorter = (Student s1, Student s2) -> {
34           // int retVal = Double.compare(s1.getGpa(), s2.getGpa());
35           // return retVal;
36           // };
37
38           // Stream<Student> stream = students.stream();
39           // Stream<Student> honorRoll = stream.filter(s -> s.getGpa() >= 3.0);
40           // Stream<Student> honorRollSorted = honorRoll.sorted(sorter);
41           // Stream<Student> honorRollSorted =
42           // honorRoll.sorted(java.util.Comparator.comparing(Student::getGpa));
43           // List<Student> honorRollSortedList =
44           // honorRollSorted.collect(Collectors.toList());
45
```

```
46          List<Student> honorRollSortedList = students.stream()
47                  .filter(s -> s.getGpa() >= 3.0)
48                  .sorted(comparing(Student::getGpa))
49                  .collect(Collectors.toList());
50
51          for (Student s : honorRollSortedList)
52              System.out.println(s);
53      }
54
55  }
```

Java 8 and JavaScript

- Java 8 provides a fast, ECMAScript v5.1-compliant JavaScript engine known as **Nashorn**
- Java programs can thus evaluate JavaScripts in the JVM

test.js

TestNashorn.java

Java 7 provided a JavaScript engine based on Mozilla Rhino - Rhino hasn't been updated recently and had some performance issues.

The example on the next pages shows a Java program calling a JavaScript function - it's also possible for JavaScript code to call Java methods.

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

```
1    function calc()
2    {
3        var array2 = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18];
4
5        var total = 0;
6
7        for(var i = 0; i < array2.length; i++)
8        {
9          total += array2[i];
10       }
11
12       return total;
13   }
```

```
1    package javascript_Nashorn;
2
3    import java.io.FileReader;
4
5    import javax.script.Invocable;
6    import javax.script.ScriptEngine;
7    import javax.script.ScriptEngineManager;
8
9    public class TestNashorn
10   {
11       public static void main(String[] args) throws Exception
12       {
13           ScriptEngine engine = new ScriptEngineManager()
14                   .getEngineByName("nashorn");
15
16           engine.eval(new FileReader("test.js"));
17
18           Invocable invocable = (Invocable) engine;
19
20           Object result = invocable.invokeFunction("calc");
21           System.out.println(result);
22       }
23
24   }
```

13 - 16B : TestNashorn.java

Java 9 Feature Overview

- Java 9 was released in September 2017
- Major features:
  - **Java Platform Module System** that lets you create **modules** of Java packages to help avoid "JAR Hell" and so you can ship only required parts of the Java SE platform
  - **Reactive Streams** support so you can write non-blocking stream-processing programs
  - **jshell**, which is an interactive Read-Evaluate-Print-Loop tool for learning and prototyping

---

For more information on modules, see:

https://www.oracle.com/corporate/features/understanding-java-9-modules.html

For an introduction to Reactive Streams, see:

https://www.baeldung.com/java-9-reactive-streams

For an introduction to jshell, see:

https://docs.oracle.com/javase/9/jshell/introduction-jshell.htm

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

Java 10 and 11 Feature Overview

- Most new Java 10 and 11 features improve the JVM itself with new garbage-collection algorithms, JIT compilers and improved security
- One notable programming feature added in Java 10 is **local type inference** (the *var* keyword)

```
jshell> var s = "Hello, world"
s ==> "Hello, world"

jshell> var x = Math.PI
x ==> 3.141592653589793

jshell> var myList = new ArrayList<URL>()
myList ==> []

jshell> var count = myList.size()
count ==> 0
```


Java 10 was released in March 2018 and Java 11's release was September 2018.

For more on type inference, see:

https://developer.oracle.com/java/jdk-10-local-variable-type-inference

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 Java 8 features:

- Date/Time Classes
- Functional Programming
- Streams
- JavaScript Integration
- Java 9 to 11 Overviews

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