

Java Programming



Will Provost

Version 17

Java. Java Programming

Version 17

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

© 1999-2022 Will Provost.

Used internally by Amica Mutual Insurance with permission of the author.

All other rights reserved by Capstone Courseware, LLC.

Published in the United States.

This book is printed on 100% recycled paper.

Course Overview

Chapter 1	The Java Environment
Chapter 2	Language Fundamentals
Chapter 3	Data Types
Chapter 4	Flow Control
Chapter 5	Object-Oriented Software
Chapter 6	Classes and Objects
Chapter 7	Inheritance and Polymorphism
Chapter 8	Using Classes Effectively
Chapter 9	Interfaces and Abstract Classes
Chapter 10	Generics and Collections
Chapter 11	Exception Handling and Logging
Chapter 12	Nested Classes
Chapter 13	Functional Programming
Chapter 14	Stream Processing

Prerequisites

- This course is intended for experienced programmers.
- No prior experience with Java itself is necessary.
- It is assumed that the student has written code in some language other than Java, and is familiar with basic programming techniques and problems:
 - Writing procedural code
 - Working with variables and expressions
 - Conditionals such as IF/ELSE and SWITCH/CASE
 - Looping constructs such as FOR, WHILE, DO/WHILE and REPEAT/UNTIL
- No knowledge of object-oriented concepts is assumed, although that will be helpful.
 - We begin with purely procedural problems for the first few chapters.
 - We introduce structured programming soon after that.
 - Then a chapter is dedicated to introducing OOAD prior to using any Java OO constructs.

Labs

- The course relies on hands-on experience in various topics and techniques.
- All lab code is written to build and run according to the standards of the Java Platform, Standard Edition 17.
- Lab exercises are deployed as standard Java projects, all under a common root folder.
 - You may have gotten these files as a ZIP from your instructor.
 - Or you may have downloaded or cloned a Git repository.
- Throughout the coursebook we'll refer to projects and other paths relative to that root – wherever you've located it on your local system – for example **Ellipsoid/Step1**.
 - You can import these projects into your IDE of choice.

Table of Contents

Chapter 1. The Java Environment.....	1
The Java Language.....	3
Structured Programming	4
Object-Oriented Programming.....	5
Functional Programming.....	6
Strong Typing and Strict Checking.....	7
Compilation and Interpretation.....	8
Forms for Java Software.....	10
Three Java Platforms.....	11
The Big Picture	12
The Java Virtual Machine.....	13
The Core API	14
The Java Runtime Environment.....	15
Development Cycle	16
Who Owns Java?.....	17
Tools.....	18
Using a Command Shell.....	19
Using an IDE.....	20
The Java Class Path	21
Searching the Class Path.....	22
Example: Hello, Java!	23
How Does It Work?	26
Performance of Java Bytecode	27
A Java Source File.....	28
Classes.....	29
Defining a Class	30
Java Applications.....	31
The Java Compiler (javac).....	32
The Java Launcher (java).....	33
Built-In I/O Streams	34
Application Input and Output.....	35
Command-Line Development Process.....	36
Integrated Development Process.....	37
Lab 1: Getting Ambitious	38

Chapter 2. Language Fundamentals43

Zooming In (Ignorance is Bliss)	45
An Expression-Based Language	46
Identifiers	47
Examples of Identifiers	48
Literals	49
Operators.....	50
Declaring and Invoking Methods	51
Variable Parameter Lists.....	52
Expressions.....	53
Evaluating Expressions	54
Declarations	55
Grammar for Method Code.....	56
Example: Expressions	57
Comments	59
Annotations	60
Lambda Expressions and Method References	61
Converting Strings to Numbers	62
Lab 2: Basic Arithmetic.....	63
Reference – Java Operators and Precedence.....	64
Reference – Java Separators	65
Reference – Java Reserved Words.....	66

Chapter 3. Data Types.....71

Strong Type Model.....	73
Primitive Types.....	74
Type Conversion	75
Value Spaces.....	76
Integral Types	77
Floating-Point Types	78
Characters.....	80
Booleans.....	81
Enumerated Types	82
Example: Data Types	83
Example: Type Conversions	86
Formatted Output	88
Formatting Strings	89
Formatting Examples.....	90
Lab 3A: More Math.....	91
Core API Documentation	92
Object References.....	93

Comparing References.....	94
Assigning References	95
Strings	96
Strings are Immutable.....	97
Comparing and Assigning Strings	98
Lab 3B: Asset Management.....	100
Arrays.....	101
Allocating Arrays.....	102
Processing Arrays.....	104
Lab 3C: A Mileage Table	105

Chapter 4. Flow Control..... 115

Getting Started.....	117
Method Declarations	118
Calling Methods	119
Structured Programming	120
Structured Programming in Java.....	121
Example: Flow Control.....	122
Testing Conditions.....	123
Handling Multiple Cases	124
Example: switch-case Examples	125
Short-Circuit Evaluation	128
Loops.....	129
The while Loop	130
The do-while Loop	131
The for Loop	132
Processing Arrays.....	133
Looping and Enumerated Types	134
Processing Variable Argument Lists.....	135
Example: Collecting Statistics	136
The Ternary Flow-Control Operator.....	139
Lab 4A: Test Scores	140
Lab 4B: DNA	141
Overriding Loop Flow	142
Issues with break and continue	143
Controlling Nested Loops	144
Lab 4C: Statistics.....	145
Lab 4D: Looking up Driving Distances	146
Recursion.....	147
Example: “Long” Division.....	148
Lab 4E: Sorting Algorithms	149

Chapter 5. Object-Oriented Software..... 167

Getting Our Bearings.....	169
Complex Systems	170
Case Study – A Car Dealership.....	171
Analysis.....	172
Advantages of Decomposition.....	173
What is a Car?	174
Classes	175
Objects	176
Advantages of Encapsulation.....	177
What is a Date?	178
A Date Class	179
Unified Modeling Language	180
UML Relationships	181
Iterative Development	183
Embracing Change.....	184
Car Dealership Design – First Pass	185

Chapter 6. Classes and Objects 187

The Java Class as an Encapsulation.....	189
Constructors	191
Garbage Collection.....	192
Example: An Ellipsoid Class	193
Naming Conventions and JavaBeans.....	195
Example: Ellipsoid as a JavaBean	196
Example: The Car Class.....	197
Example: Building the Inventory	199
new Objects vs. new Arrays.....	201
Implementing Relationships.....	202
Car Dealership Design – Second Pass.....	203
Using this.....	205
Visibility	206
Lab 6A: Listing and Finding Cars.....	207
Packages.....	208
Package and Class Names	209
Organizing Class Files by Package	210
Package Naming and Design	211
Imports	212
Best Practices with Imports.....	214
Demo: Packaging Hello, Java.....	215
Car Dealership Packages	218

Package Visibility	219
Overloaded Methods.....	220
Overloading Constructors.....	221
Rules for Overloading.....	222
Lab 6B: Finding More Cars, and Driving Them.....	223
Lab 6C: Selling Cars	224
The Java ARchive	225
Working with JARs.....	226
Lab 6D: Java in a JAR	227

Chapter 7. Inheritance and Polymorphism239

UML Specialization.....	241
Advantages of Specialization	242
Example: Used Cars	243
Extending a Class	244
Example: Implementing UsedCar	245
Using Derived Classes.....	249
Type Conversion	250
Type Identification.....	251
Example: Type Conversions	252
Type Identification and Casting.....	254
Car Dealership Design – Third Pass	255
Lab 7A: Listing and Selling Used Cars.....	256
Protected Visibility.....	257
Polymorphism	259
Overriding Methods.....	260
Rules for Overriding	261
The @Override Annotation	262
Referencing the Superclass.....	263
Example: Bank Accounts.....	264
Lab 7B: The All-Important	266

Chapter 8. Using Classes Effectively273

Class Loading.....	275
Static Fields	276
Static Methods	277
Crossing the Boundary	278
Example: Fixed Discounts.....	279
Static_INITIALIZER Blocks	280
Static Imports.....	281
Example: Calendar Constants.....	282
The Object Class.....	283

Prohibiting Inheritance	284
Costs of Object Creation	285
The Secret Lives of Strings	286
Performance of Strings	287
Example: DNA.....	288
StringBuffer and StringBuilder	291
Using StringBuilders	292
Example: Tuning Up DNA	293
Working with Strings.....	294
Controlling Object Creation	296
Understanding Enumerated Types	297
Example: Disassembling Color.....	298
Stateful and Behavioral Enumerations	300
Example: Car Handling	301
Lab 8: Condition as a Behavioral Enumeration.....	303

Chapter 9. Interfaces and Abstract Classes307

Interface vs. Implementation	309
UML «Interface» and Realization	310
The Java Interface.....	312
Implementing Interfaces	313
Single Implementation Inheritance	314
Example: Interfaces in the Core API.....	315
Car Dealership Design – Fourth Pass	316
Demo: Simple Seller	317
System Properties	321
Lab 9A: The Art of the Deal	322
Abstract Classes	323
Designing with Abstract Classes	324
Example: Message Cracking.....	325
Car Dealership Design – Fifth Pass.....	326
Lab 9B: Polymorphic Inventory	327

Chapter 10. Generics and Collections.....337

Limitations of Arrays	339
Dynamic Collections.....	340
Generic Types	341
Declaring Generic Classes	342
Using Generics.....	343
Implicit Type Arguments.....	344
Example: A Generic Pair of Objects.....	345
The Collections API	347

Collection Types.....	348
The Collection<E> Interface.....	349
The List<E> Interface	350
The ArrayList<E> Class.....	351
The LinkedList<E> Class.....	352
Building Collections.....	353
Reading Elements.....	354
Looping Over Collections	355
Collecting Primitive Values	356
Auto-Boxing.....	357
xxxAll Methods.....	358
Convertibility of Generics	359
Wildcards	360
Example: Car Dealership.....	361
Lab 10A: Better Management of Test Scores	364
The Iterator<E> Interface	365
Maps.....	366
The Map<K,V> Interface	367
Example: Post Offices	368
Sorted Sets and Maps	369
Example: 50 States.....	370
Lab 10B: Gathering Statistics in a Map.....	371
The Collections Class Utility.....	372
Example: Algorithms	373
Example: Sorting Scores	375
Conversion Utilities	376

Chapter 11. Exception Handling and Logging385

Reporting and Trapping Errors.....	387
Exception Handling	388
Exceptions	389
Throwing Exceptions.....	390
Unchecked Exceptions	391
Declaring Exceptions	392
Trying and Catching	393
Catching Exceptions	394
Demo: Bad Geometry	395
Multiple catch Blocks.....	401
Catching Multiple Exception Types.....	402
Catch-and-Release.....	403
Chaining Exceptions	404
Lab 11A: Exceptions in the Car Dealership	405
try-with-resources	406
Example: Auto-Closing	407
Logging	409
Logging APIs.....	410
The Java Logging API	411
Using a Logger	412
Logging Levels	413
Log Hierarchy	414
Example: Logging and Configuration.....	415
Lab 11B: Logging in the Car Dealership.....	420

Chapter 12. Nested Classes433

Classes Everywhere!	435
Nested and Static Classes.....	436
Using Static Classes	437
Example: A Private Record Type.....	438
Inner Classes	439
Instantiating Inner Classes.....	440
Visibility to the Outer Class	441
Using Inner Classes.....	442
Example: Monitoring Scores.....	443
Local Classes	447
Anonymous Classes	448
Using Anonymous Classes.....	449
Example: Named vs. Anonymous	450

Chapter 13. Functional Programming	453
Utilities, Frameworks, and Callbacks	455
Passing Behavior as a Parameter	457
Starting Point: Inner Classes	459
Functional Interfaces	460
Example: Functional Interfaces	461
Built-In Functional Interfaces	462
Getting Familiar with java.util.function	463
Example: A Configurable Seller	465
Lambda Expressions	468
Example: Lambda Expressions	470
Grammar for Lambda Parameter Lists	472
Grammar for Lambda Bodies	473
Example: A Configurable Seller	474
Challenges in Writing Lambda Expressions	476
Functional Interfaces in the Collections API	477
Lab 13A: Sorting and Filtering Cars	478
Scope and Visibility	479
Deferred Execution	480
Lab 13B: Using a Generic Data Browser	481
Method References	482
Using Method References	483
Example: Method References	484
Type Arguments in Method References	485
Identifying Instance Methods by Class	486
Creational Methods	487
Null Checks, Type Identification, and Casting	488
Lab 13C: Sorting and Filtering Parts	489
Default Methods	490
Example: Default Methods	492
Example: Stack Traces	494

Chapter 14. Stream Processing505

The Stream Processing Model	507
The Stream API	508
Relationship to Collections	509
Advantages	510
Disadvantages	513
Iterating	514
Filtering	515
Example: Filtering and Counting	516
Mapping	517
Primitive-Type Streams.....	518
Demo: Mapping and Filtering	519
Aggregate Functions and Statistics	524
Demo: Averaging.....	526
Sorting.....	528
Lab 14A: Test Scores	529
Generating.....	530
Limiting	531
Reducing.....	532
Joining.....	533
Example: Generating, Limiting, and Reducing	534
Finding and Matching	535
Grouping	536
Demo: Grade Distribution	537
Flattening and Traversing	539
Example: Calling All Cars	540
Lab 14B: Car Queries	542
Parallel Processing.....	543
Example: Sequential vs. Parallel Processing.....	544



CHAPTER 1

THE JAVA ENVIRONMENT

OBJECTIVES

After completing “The Java Environment,” you will be able to:

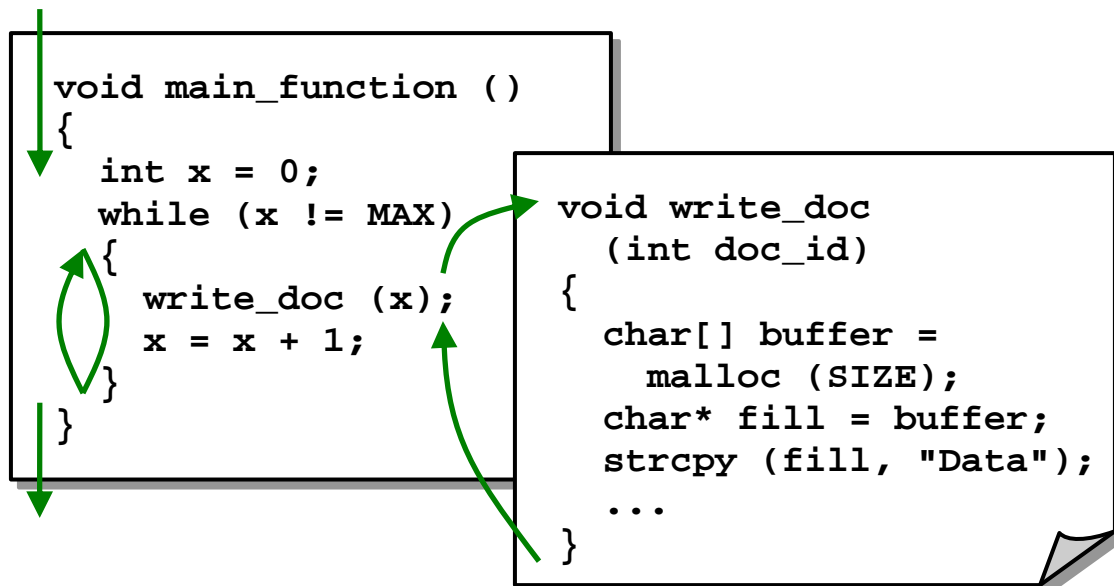
- Explain how Java achieves portability over multiple platforms by compiling to an intermediate bytecode and relying on a Core API implemented per target platform.
- Describe the practical advantages of the Java architecture over other current language and architectural choices.
- Describe the Java software development cycle, especially as it distinguishes Java from other popular languages.
- Build and test a simple, pre-coded Java application.
- Observe bugs and other incorrect behavior in a simple application, and make simple code fixes and observe correct behavior.

The Java Language

- When Java 1.0 was released in 1995, it was positioned as an improvement over the **C** and especially the **C++** programming languages, which were dominant in business software development at that time.
- As such Java is a **structured** language, like C, with program code organized in callable **methods** and a consistent flow of program control through a method's instructions, conditionals, and loops.
- It is an **object-oriented** language, like C++, with all code gathered into **classes** that encapsulate logically related state and behavior, and **objects** as instances of those classes that can be created and caused to interact with each other.
- In more recent versions it is also a **functional** language, offering a number of techniques for passing definitions of behavior to methods – or, in a sense, passing one method to another.
 - In this evolution Java begins to look more like popular modern languages including **JavaScript**.

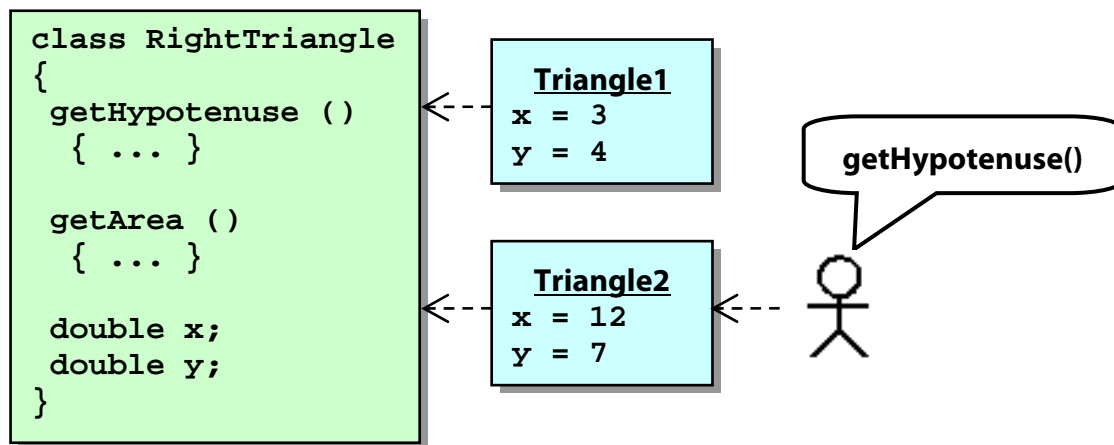
Structured Programming

- Structured programming promotes code robustness, reuse, and maintainability through techniques including:
 - **Organization** into reusable **functions** that can be called from different sources
 - **Declaration** of variables prior to use
 - **Single entry and exit points** for code passages
- The classic structured programming language is **C**.



Object-Oriented Programming

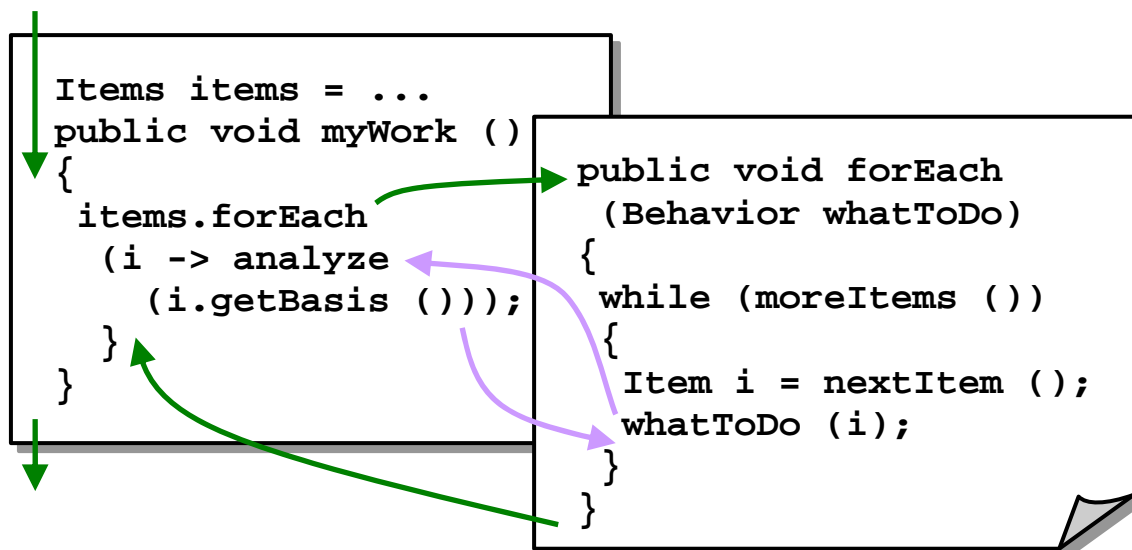
- Object-oriented programming, or **OO**, promotes **encapsulation** of related state and behavior into a coherent **class**.
 - You'll hear many related abbreviations, probably the most popular of which is **OOAD**, which strictly speaking refers to OO **analysis and design**, rather than implementation.
- **Objects** of various classes are brought into being in memory, and their interactions make up the activity of the program.



- OO languages and processes offer a few advantages:
 - Better **reuse** at the class level
 - More **maintainable** code, as encapsulation establishes "a place for everything" and puts "everything in its place."
 - Better **adaptability** to change
- The most popular OO language prior to Java was **C++**, which is actually a superset of the C language.

Functional Programming

- Strictly speaking, pure functional languages are radically different from OO languages:
 - OO calls for classes to hold state and offer related behavior, and **changes in state** will naturally change related behaviors.
 - In OO, that's good; in functional languages, that's called a "**side effect**," and it's bad.
- It's not quite right, then, to say that Java is a functional language.
- But, especially with the release in 2014 of its eighth edition, Java facilitates a programming style in which largely **stateless** methods can take definitions of **behavior as parameters**.



- This gives an OO design better tools for **decoupling** unrelated concepts – such as how to iterate over a complex data structure vs. what you might want to do with each element in that structure.
- This in turn can make classes **easier to use**, and often more **reusable** as well.

Strong Typing and Strict Checking

- Java inherits another aspect of structured-programming from C, namely a strong **type model**.
- In a **strongly typed** language such as Java, all data in memory is of a specific type.
 - There are **primitive types** such as a 4-byte **integer**, a **boolean** value, or a **floating-point** decimal.
 - And there are **object references**, and the objects to which they refer are of specific **classes**.
- When a Java program is **compiled** to bytecode and made ready for execution, the compiler enforces rules for each type, and especially having to do with conversion from one type to another.
- This is known as **strict type-checking** or **strict checking**, and among other things it means that ...
 - You must **declare** a variable and its type before using the variable.
 - You must assign values in the legal **value space** for a variable's declared type – for example a string is not a number, and you can't make a number behave as a true/false boolean.
 - You must **convert** a value from one type to another only in ways that **don't lose precision** – such as storing a single byte in a 4-byte integer, but not the other way around.
- Strong typing can make for frustrating code-authoring at times.
- But it generally has a positive effect on the **robustness** of code, and tends to stamp out simple programming errors and typos before you try to run your program.

Compilation and Interpretation

- Java software is created by a development process different from that of languages that precede it historically – and especially from traditional compiled languages.
- C, C++, and other **compiled languages** produce **executable files** and **libraries** of object code, all of which can be run **directly on the target platform**.
 - This makes for very **efficient** executable code.
 - However, the products of compilation are platform-dependent, requiring **cross-compilation** (and cross-testing) if they are to be run on multiple operating systems.
- BASIC, SQL, JavaScript, and other **interpreted languages** are translated directly from their source code into executable machine code at runtime.
 - This offers great **portability**, because different **runtime environments** can exist for different underlying operating systems and hardware platforms.
 - It can be **slower**, due to the need to parse the source code on the fly – though with modern processors and runtime architectures, for some purposes at least this is not an issue.
 - It can be more **error-prone**, because without a compiling process and a strong type model (which are two different things but often go hand-in-hand), simple coding mistakes only appear at runtime.

Compilation and Interpretation

- Java relies on a combination of compilation and interpretation and tries to get the best of both worlds.
- It is compiled to an intermediate **bytecode** – which in most cases is not the actual instruction set for the physical processor.¹
 - It nevertheless has the **efficiency** of traditional machine code, representing instructions with single bytes and arguments to those instructions in one or two more bytes.
 - And it represents the results of a compilation process that includes **strict type-checking**, so that most basic coding errors are “pushed to the front” of the development process, and never see runtime.
- This bytecode is then interpreted – that is, translated into the machine code on an actual computer.
 - This gives Java the **portability** that most compiled languages lack.
 - It also includes **bytecode verification**, which re-asserts some of the language rules first enforced by the compiler.
 - This is partly for **robustness** and partly for **security** – to avoid some of the common low-level vulnerabilities of C and C++, such as buffer overruns.

¹ There are actually such things as Java CPUs – physical chips designed to run the Java instruction set, or optimized for it. On such a chip the “interpretation” step is just $X \rightarrow X$. Such chips are not in common use.

Forms for Java Software

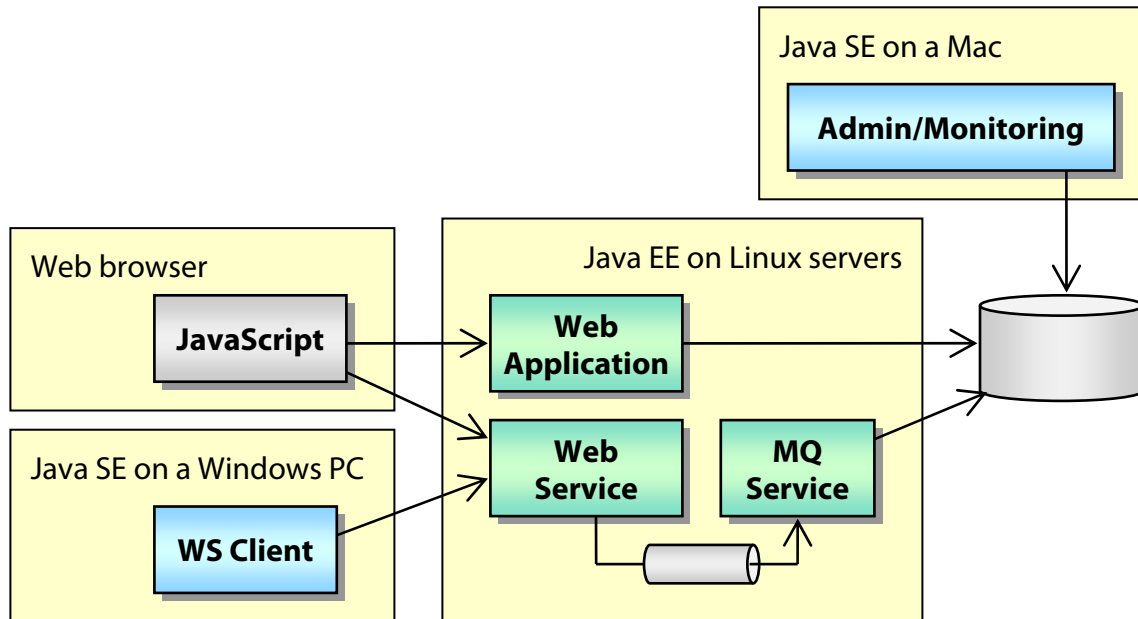
- **Java software can take many forms:**
 - **Console applications** – our traditional programs, started manually or as part of a batch process, and with a discrete job of work to perform
 - **GUI (graphical user interface) applications** – application software in the tradition of Windows desktop applications
 - **Web applications** – usually **hosted** by a **server** process that interprets **HTTP** requests and responses for Java classes
 - **Web services** – components that are available over HTTP as well, but with other pieces of software as their clients, rather than interactive users
 - **Messaging components** – listening for messages on FIFO queues or publish-and-subscribe topics and triggering specific processing
 - **Applets** – a somewhat outdated model by which a small Java component is downloaded from a Web server into a browser and hosted on the client side

Three Java Platforms

- To organize these diverse types of software into a clear architecture, architects have defined three distinct Java platforms.
- The **Java Platform, Standard Edition**, or **Java SE**, is the oldest and most common edition in use.
 - It supports Java application, applet, and component programming for desktop and larger systems – primarily PCs – which may or may not be networked.
 - Common uses of Java SE are standalone GUI and console applications, batch processors, and web-service clients.
- The **Enterprise Edition**, or **Java EE**, extends the standard edition with APIs for so-called “enterprise features.”
 - Java EE implementations support Web service via **servlets** and **JSPs**, persistent data with **JPA**, distributed messaging with **JMS**, and transactional systems via **EJB** – to name a few technologies.
 - Java EE components are more strictly meant for the server side of large-scale systems: processing power, memory and storage space are presumed to be plentiful and expandable.
- The **Micro Edition**, or **Java ME**, is much less used.
 - Java ME supports a variety of “micro” devices, such as phones, tablets, and embedded devices.
 - Under Java ME, CPU power, memory, storage, and connectivity are all assumed to be limited, perhaps severely.
 - It offers a **subset** of the Java SE runtime capabilities.

The Big Picture

- So Java can be the language and runtime environment that supports simple and small systems up to massive, enterprise-wide, distributed business applications.
- Here is one common architecture:



- Java SE components are shown in blue: the web-service client and administrative/monitoring application.
- Java EE components are shown in green: web application, web service, and messaging component.
- The one place where Java doesn't appear is, oddly, the place for which it was initially marketed by Sun Microsystems way back when, and that is in the browser.
 - **JavaScript** is the de-facto standard for browser-hosted applications, and it can communicate with both web applications and web services over HTTP.

The Java Virtual Machine

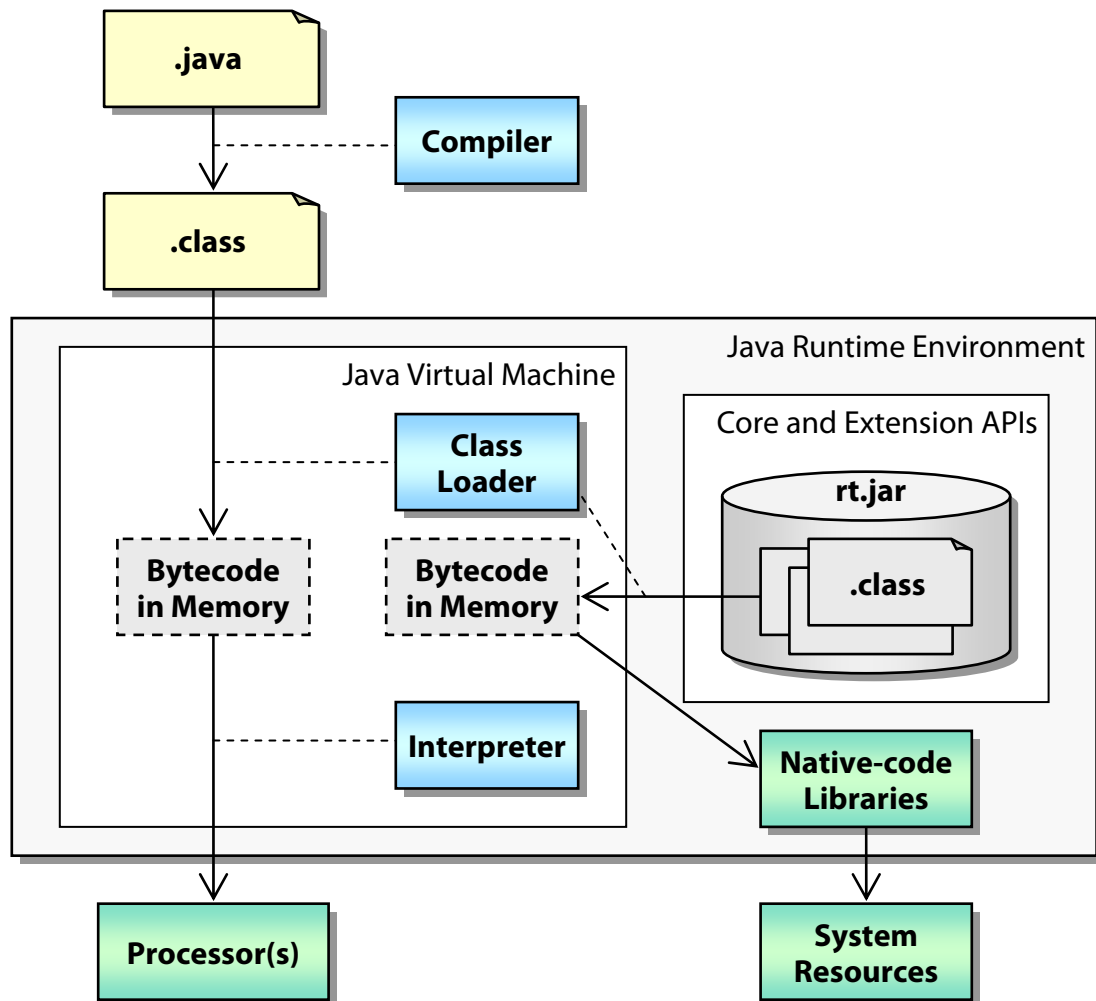
- The key to the portability of Java software is the **Java Virtual Machine**, or **JVM**.
- A virtual machine is just what the name implies: it provides an environment for Java software that emulates a real, physical computer.
 - It has its own machine language or instruction set, known as **Java bytecode**.
 - It implements a **process** and offers Java software a model of multiple **threads** running on a “virtual CPU.”
- The JVM is implemented for a particular type of concrete hardware, and also is dedicated to a particular operating system.
- Java source files are compiled to Java bytecode, which can then be interpreted by the JVM.
 - The JVM translates bytecode instructions to machine language for the real CPU, and then submits sets of these instructions to be executed.
 - The process-and-thread model is similarly mapped to abstractions in the target operating system.
- In this way the JVM isolates compiled Java programs from the details of a particular platform (hardware plus operating system) while allowing them to run on many different platforms without being rewritten.

The Core API

- Like most high-level languages, Java comes with a prepared library of code to perform common tasks.
- This is much more than a convenience.
- The **Core API** not only provides utility methods; it is in fact the only means for Java programs to interact with the outside world.
 - That is, in order to touch any real device, such as the display, mouse, keyboard, file system, network interface, etc., Java programs must make calls on the Core API.
 - Contrast this to C or C-like languages, in which it is possible and even encouraged to write directly to device memory or to make calls directly on an operating system's own API.
- Over the years Java has acquired a number of **extension APIs**, which are not technically part of the core but are expected to be available in a compliant runtime environment.
 - So when we say “core,” we often mean “core and extensions.”
- Like the JVM, the Core API is implemented as part of the Java environment and is dedicated to a particular platform.
 - Thus, like the JVM, the Core API isolates Java programs from OS-specific semantics by providing a consistent interface to the OS.
- It establishes portability over different platforms for high-level behaviors including writing files, building graphical interfaces, printing, and making network connections.

The Java Runtime Environment

- The combination of JVM and Core API makes up most of the **Java Runtime Environment, or JRE**.
 - To run a Java program, a machine must have a JRE installed.
- Here is a summary of development, deployment, and execution of a Java program, as described on the following page:



Development Cycle

- Java software begins as source code in files with **.java** as the filename extension.
- The code in these files always defines **classes**: there will be one or more classes defined per source file.
- The source files are **compiled** into **bytecode**.
 - This bytecode is far more compact than the source code, and is not easily readable to the human eye.
 - It is not composed of CPU instructions, but of instructions in a platform-independent Java instruction set.
- **Compilation** takes **.java** files as input and organizes the resulting bytecode into **.class** files, one for each class represented in the source code.
 - These files are the basic units in which the software is distributed from developer to user – or to another developer for reuse in another application.
 - They can be used directly at runtime, downloaded over the Internet, or packaged in archives such as ZIPs and JARs.
- At runtime, the class files are **loaded** into memory and **interpreted** into CPU instructions specific to a target platform – this is the job of the JVM.
- High-level calls on the Core API ultimately devolve to invocation of **native code** that is dedicated to the platform on which the code is running.

Who Owns Java?

- Sun Microsystems released Java 1.0 in 1995, with the stated goal of keeping the architecture open to participation by other companies, individuals and organizations, while keeping a centralized process for Java to mature without fragmenting.
 - To this end they created a formal and open **Java Community Process**, or **JCP**.
 - The JCP was hosted by Sun but acted as a representative democracy, forming expert groups to define new versions of language specs, new APIs and new uses of Java.
- Oracle Corporation acquired Sun Microsystems in 2010 – and with it the collection of intellectual property that is Java.
 - Oracle now does much of the heavy lifting, especially in building the reference implementations of Java SE specifications – the standard JREs and JDKs.
 - They have also acquired many previously competing application-server and database products, including BEA **WebLogic**, **GlassFish**, and **MySQL**.
- But the JCP is still in force and many companies and non-profit organizations lend their voices to Java's development.
- Among the latter group, one notable is open-source pioneer the **Apache Software Foundation**, which is responsible for many reference implementations of Java EE standards and for many libraries of useful, freely-available code.

Tools

- **Our primary tools for hands-on exercises in this course are:**
 - The **Java 17** developer's kit
 - An integrated development environment (IDE) of your choice
- **The lab software supports two modes of operation:**
 - Integrated building, deployment, and testing in your IDE
 - **Command-line** builds using prepared scripts
- **We'll provide instructions for each approach separately on the next few pages.**
- **Most students will prefer to use the IDE for hands-on exercises, and where there are differences in practice we will lean in the direction of using the IDE.**
- **But, especially in the first few chapters of the course, we'll emphasize command-line usage, as a way of getting familiar with the basic mechanisms of Java development.**

Using a Command Shell

- To compile and run your code projects from DOS, PowerShell, bash, or similar, you'll need to be sure that your JDK is set up correctly, with a **JAVA_HOME** environment variable and the **bin** folder in your executable path.

- For Example, for Windows:

```
set JAVA_HOME=c:\Java17
PATH="%JAVA_HOME%\bin";...
```

- Your JDK location will be specific to your machine: this is often found under **c:\Program Files\Java\...**
- Note that environment changes in an open console don't affect the operating system as a whole.
 - This can be a good thing, especially if you have another Java environment set up for work. Anything you do in a DOS or PowerShell window is isolated.
 - But remember that any new consoles will need the same setup – or you can spawn one from another and inherit the environment settings, for instance with **start** in DOS.

Using an IDE

- You can open any of this course’s Java projects (look for a folder with an **src** subfolder) in your IDE of choice.
- Some IDEs will call this “opening” a project, and some will call it “creating” a project on the existing code.
- Most IDEs are set up to build open projects automatically, so the “build” step mentioned in some instructions in the course will not be an explicit action on your part.
- Then you can run a given Java class as an application.
 - The user interface for this varies by IDE.
 - This boils down to invoking the **java** launcher in order to run the **main** method in that class.

The Java Class Path

- A Java program is not necessarily executed from a single file in a specific location.
- Rather, the JVM uses a more flexible process called **class loading** to find Java bytecode in **.class** files.
 - This process starts with the **launching** of a designated class – this is the **application class**.
 - Once this class is loaded, its **main** method is invoked. **main** might call other methods within the application class.
 - Any classes referenced in the flow of execution will be loaded by the JVM as needed ... and so on for each new class as referenced.
- The class loader finds classes by searching the **class path**.
 - The class path is a sequence of locations on the file system, which can include **directories** as well as archive files: **ZIP** archives and especially Java ARchives or **JAR** files.
 - The class path is most often expressed as a string in which these locations are separated by an OS-specific token: a colon for UNIX and a semicolon on Windows.
 - The class path can be defined globally, as an environment variable, but more often it is set specifically for a given launch of a JVM.
 - The Core API classes do not need to be included explicitly on the class path; they are always available to the JVM by way of a default **boot class path**.

Searching the Class Path

- For example, say the class path is defined on a Windows PC as:

```
java -classpath c:\Dev\Java;c:\Vendor\Classes
```

- To run an application class using the **java** launcher, the **.class** files for that application must be located in one of those two directories:

1. **c:\Dev\Java\MyClass.class**

2. **c:\Vendor\Classes\MyClass.class**

- If the file can be found at one of those locations, the launcher can be used to run the application from any working directory.
- The class loader will load whatever class file it finds first.
- If it is not found in either, the launcher will report an exception:

```
java MyClass
```

```
Exception in thread "main"
```

```
java.lang.NoClassDefFoundError: MyClass
```

- In fact it is one step more complicated than this, in typical practice, because of **packages**.
 - Java classes are organized into packages, with names of the form **a.b.c**.
 - This also maps to the file system, such that for a class **a.b.c.MyClass** the file will be found in a **subdirectory**, the complete path being **a/b/c/MyClass.class**.
 - The class loader actually seeks out the file at this **relative path** under each node of the **class path**.
- We'll start working with packages later in the course.

Hello, Java!

EXAMPLE

- In **Hello/Step1**, there is a dead-simple Java application that prints a greeting to the console.
- The source code is as follows, though we will not worry about any details of Java coding for a little while yet.
 - You can see this in **src/Hello.java**:

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

- What we will consider for the moment:
 - How to build and test the application
 - How the application works for a given platform

Hello, Java!

EXAMPLE

- As with all projects in this course, two scripts have been prepared:
 - A **compile** script (**compile.bat** for Windows/DOS or **compile** for Linux, Mac, and other UNIX systems) to compile the source code in **.java** files to bytecode in **.class** files
 - A **run** script (**run.bat** or **run**) to launch the application class in a new JVM

1. Open a console and navigate to **Hello/Step1**.
2. View a listing of the files in that directory, and the subdirectory **src**:

```
dir or ls
compile
compile.bat
run
run.bat
src
```

```
dir src or ls src
Hello.java
```

3. See that the **compile** script just runs **javac**, and that the **run** script runs the **java** launcher, with a class path including the **src** folder:

```
type compile.bat or cat compile
...
javac src\*.java or javac src/*.java
```

```
type run.bat or cat run
...
java -classpath src Hello
```


Hello, Java!

EXAMPLE

4. Run the **compile** script. View the **src** listing again, and you'll see the new file **Hello.class**.
5. Run the **run** script and see that the application class runs and prints the greeting back to you:

```
run or ./run
Hello, Java!
```

6. Let's experiment a little with the class path and class loading. Try moving the class file to the main project directory:

```
move src\Hello.class . or mv src/Hello.class ./
```

7. Now try **run** again:

```
run or ./run
Error: Could not find or load main class Hello
Caused by: java.lang.ClassNotFoundException: Hello
...
```

8. To load the class now, you'll have to set a different class path. Invoke the **java** launcher explicitly as follows:

```
java -classpath . Hello
Hello, Java!
```

How Does It Work?

- How does this application work on your platform?
- Java software achieves portability on two levels.
- Rather than compiling to machine code for a specific CPU, Java is compiled to portable bytecode.

– Here's a listing of the bytecode in **Hello.class**:

```
0:  getstatic      #2; // gets the printing stream
3:  ldc           #3; // loads the greeting string
5:  invokevirtual #4; // calls the print method
8:  return
```

- This is interpreted to machine instructions, say for an Intel Core i5 chip, at runtime, by the JVM.
- It would of course be interpreted differently for an AMD processor, RISC, ARM, etc.
- Where the application needs to interact with the physical system, it makes Core API calls such as **System.out.println** to write to the console output.
 - The Core API implementation for your platform holds the logic for connecting to the console's standard output stream and producing characters to that stream.
 - The **operating system** is ultimately responsible for implementing the rendering of characters on the screen.

Performance of Java Bytecode

- Java's reliance on an interpreting stage trades speed for portability.
- Java bytecode will never be as fast as native code comprised of machine-runnable instructions.
- The Java architecture includes an optional component called a **Just-in-Time compiler**, which can be plugged into the runtime environment.
- JIT compilers optimize code performance using one or more techniques, none of which is mandatory:
 - Compile or recompile a class file into a so-called “**fat**” class file, which includes both the original bytecode as well as native (thus directly runnable) code for one or more target platforms.
 - Generate native code from bytecode as it is loaded, algorithmically determining which methods are used frequently and caching the associated native code.
- JIT compilers can dramatically enhance the performance of Java software, and are commonplace today as parts of Java runtime environments.
- An algorithm implemented in Java will typically run more slowly than the same algorithm in a purely compiled language, but not by a tremendous margin.

A Java Source File

- Here is an excerpt from a Java source file:

```
package cc.language;

import java.io.*;

public class Translator
    extends OutputStream
{
    private int middle;

    public static void main (String[] args)
    {
        // content removed here
    }

    public void write ()
    {
        // ... and here
    }
}
```

- Elements of this source code, which we'll investigate in this and the next few chapters:
 - An optional **package** declaration, which assigns the contents of the source file to a specific package – this helps to partition the Java namespace
 - An optional **import** declaration, which identifies external packages to which this source file will refer
 - A **class** definition, including (in this case) one **field** and two **method** definitions
- All other Java source code is contained in a class, in some way.

Classes

- The **class** is the primary means of organizing Java code for any purpose.
 - All code, except for package and import statements and comments, is contained in some class – in both source-code and bytecode forms.
 - All forms of deployed Java software contain at least one class, and typically contain several of them.
- Java is an **object-oriented** language.
 - The term **class** will be familiar to programmers of other OO and pseudo-OO languages, such as C++, Ada, Smalltalk, and C#.
- We will defer study of Java as an OO language until Chapter 6, and in the next few chapters will focus on basic language elements and procedural programming.
- For now, we need only to understand that a Java class contains two types of top-level elements: **methods** and **fields**.
 - Methods are blocks of code that can be invoked by defined **signatures** of parameters and return types. Methods are much like “functions” in non-OO languages, and “member functions” in some OO languages.
 - We’ll have no use for fields until Chapter 6.

Defining a Class

- Typically there will be a single class per source file.
- The name of this class and the name (without the **.java** extension) of the source file must match.
 - This helps the compiler find source files for classes when they're referenced by other classes.
- Define the class to be **public** – this means it is accessible to anyone.

```
public class MyClass
{
}
```

- We'll talk about the **visibility** modifiers, including **public**, when we get to object-oriented topics.
- Then populate your class with methods and fields, according to your software design.
 - For the moment we'll confine ourselves to methods.
 - Further, to stay out of the object-oriented space, we'll make all our methods **static**. This means that they can be called without creating an instance of the class first – they are something like “globals” from other languages.

```
public class MyClass
{
    public static void doWork () {...}
    public static int sum (int x, int y) {...}
}
```

Java Applications

- In some languages, a deployed application is an executable file.
- In Java, any class can be an application – again, by exposing a **main** method:

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

- The method signature for **main** is strictly checked.
 - It must be **public**, so that the launcher can call it.
 - It must be **static**, so that the launcher can call it without creating an instance of the enclosing class.
 - It must have return type **void** – meaning does not return a value to the caller.
 - It must accept a single parameter which is an array of **Strings**. By convention this is called **args**, but it could be anything. This array will be populated with any **command-line arguments** passed to the launcher.
- It is not necessary for an application class to confine itself to the definition of a **main** method.
 - It can define other methods that **main** can call.

The Java Compiler (javac)

- A Java compiler parses source code according to the grammar of the language and produces bytecode in “class files” – with the extension **.class**.
- As with most of the Java development tools, there is a public specification to which a Java compiler must conform.
 - There are many Java compilers on the market.
 - We will use the reference implementation that comes with the standard JDK – this is called **javac**.
 - Note that it is also possible to write a compiler that produces Java bytecode from **other source languages** – and this practice is on the increase with languages such as **Scala** and **Groovy**.
- Provide one or more source filenames to **javac**:

```
javac MyClass.java
javac Class1.java Class2.java Class3.java
javac project\src\*.java
```

- You can direct the output of the compiler to a target directory other than the working directory, using the **-d** switch:

```
javac -d c:\Dev\Classes MyClass.java
```

- You can tell the compiler to observe a different classpath for the duration of the compile with the **-classpath** switch. Recall our experiments with class paths from the previous chapter.

```
javac -classpath . MyClass.java
javac -classpath c:\MyClasses *.java
```


The Java Launcher (java)

- A JRE can be instantiated in many ways.
- In starting Java SE applications, the JRE is typically created by a **launcher** that performs several tasks:
 - Creates a JVM
 - Loads the Core API into that JVM's memory
 - Loads a Java class as directed by the user
 - Looks for a **main** method on that class and invokes it
- The JDK's launcher is called, simply, **java**.

```
java MyClass
```

```
java -classpath c:\MyClasses MyClass
```

Built-In I/O Streams

- Recall **Hello** printing a greeting to the console.
- This is the simplest possible example of interaction between Java code and a system device – in this case through the abstraction of an **output stream**.
 - Modern operating systems consistently offer three system streams: **input**, **output**, and **error**.
 - Java models each of these separately, as **System.in**, **System.out**, and **System.err**.
- Throughout this course, we'll make extensive use of **System.out** to produce program output, and to a lesser extent of **System.in** to gather user input.
- These, and command-line arguments to the launcher, will be our means of interacting with Java programs.
- Complete treatment of Java I/O streams is beyond the scope of this course, and is covered in *Advanced Java Programming*.

Application Input and Output

- A program can use either of the output streams to produce output to the user, primarily using the **print** and **println** methods:

```
System.out.print ("a ");  
System.out.println ("bc");  
System.out.println (); // prints a blank line  
System.out.println ("d" + "e");
```

- This produces:

a bc

de

- An application can parse the complete command that was used to launch it.
 - The array of strings passed to **main** contains tokens parsed from the command line.

```
System.out.println ("You passed: " + args[0]);
```

- It does not include the name of the Java class – so it's possible that the array will be empty.

- **System.in** can be read by Java code, primarily using the **read** method:

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

- **System.in** buffers a line of input at a time.
- So many calls to **read** will block until the user hits ENTER.
- Then the buffer will be flushed, so that many calls to **read** will run, un-blocked, in rapid succession.

Command-Line Development Process

- From this point forward, we'll adopt a consistent process of building and testing applications.
- Two scripts will be available in each project directory:
 - The **compile** script will run **javac** on the appropriate source files for the project, directing output to a subdirectory **build** – which it will also create if necessary and clean out if it already exists, prior to compilation:

```
javac -d build src\*.java
```

- The **run** script will include this directory in the class path, and will pass all its command-line arguments along to the launcher:

```
java -classpath build MyClass %*
```

- Some projects will involve variations to these two scripts, and these will be described in the relevant demo, example, or lab exercise.

Integrated Development Process

- If you prefer to work in an IDE ...
 - A Java compile is triggered automatically – when you save a source file, or just-in-time when you run an application.
 - You'll also see errors and warnings immediately, as you edit your source code, because the IDE has a **syntax checker** that marks up the source proactively.
- Where a console application takes command-line arguments, you will need to create a **run configuration** to provide them.
 - The easiest way to do this is first to run the application without arguments, as described above.
 - You may see a failed run of the application, with an error or a usage statement.
 - But then you can edit the run configuration that is created automatically, and add command-line arguments, or make other tweaks to the way the application is run.
 - You can also create multiple run configurations for the same class, with different arguments, making repeated testing a bit easier.
- Keep in mind that you can mix development environments, and a sweet spot is often found by coding in the IDE but testing from a command console.
 - You must **compile** from the command line before a **run**.

Getting Ambitious

LAB 1

Suggested time: 15 minutes

In this lab you will experiment with building and testing a slightly more complex Java application. **Ambitious** expands on **Hello** by offering options for what greeting it should produce. It can be terse or verbose, and in any of several languages.

However, **Ambitious** has a couple problems! You'll build and test the application and see it fail in a few ways, and fix the bugs as directed until it works correctly. This lab is not designed to take us into the details of the Java language! Rather we're hoping to get familiar with some of the common occurrences of building and testing Java applications, so we can recognize the common causes and work more effectively over the next few chapters.

Detailed instructions are found at the end of the chapter.

SUMMARY

- Java is more than just a programming language.
- Java software enjoys a number of fundamental advantages in development as well as deployment, including portability and efficiency.
- The Java language has been designed to greatly reduce bugs in development.
- While not the equal of traditional compiled languages in its runtime performance, Java has closed the gap dramatically over the last few releases, and now offers at least comparable speed.
- The Java Virtual Machine and important tools in the development process, especially the compiler, are precisely specified in their behaviors and hence can be implemented variously without compromising the portability of Java source code and bytecode.
- All Java software is organized into classes.
 - For our purposes over the next few chapters, we will work with single-class applications and focus on the **main** method.
 - Later, we'll explore the full power of classes as expressions of the object-oriented approach to building software.

Getting Ambitious

LAB 1

In this lab you will experiment with building and testing a slightly more complex Java application. **Ambitious** expands on **Hello** by offering options for what greeting it should produce. It can be terse or verbose, and in any of several languages.

However, **Ambitious** has a couple problems! You'll build and test the application and see it fail in a few ways, and fix the bugs as directed until it works correctly. This lab is not designed to take us into the details of the Java language! Rather we're hoping to get familiar with some of the common occurrences of building and testing Java applications, so we can recognize the common causes and work more effectively over the next few chapters.

Lab project: **Ambitious/Step1**

Answer project(s): **Ambitious/Step2**

Files: * to be created
src/Ambitious.java

Instructions:

1. Try to compile the starting code by running the **compile** script. You will get a compiler error. (Or, see this error immediately upon opening the project in an IDE, and “red ink” markup on the source file **Ambitious.java** when you open it.)

compile

```
Ambitious.java:29: error: <identifier> expected
    public void main (String[])
                      ^
```

1 error

2. Note that the compiler gives you the source file and line number. The error says that an identifier was expected at a specific point: method parameters in Java must be named. This is one example of Java's strict approach to compilation, the aim being to push potential errors to the earlier stages of development. Open the source file and fix this by naming the parameter **args**:

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


Getting Ambitious

LAB 1

3. Try compiling again. You should get a clean compile. Now try running the class as an application, using the **run** script as below. What do you get?

run

Error: Main method is not static in class Ambitious, please define the main method as:

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

Or, try running as a Java application in your IDE – you’ll see that this isn’t an option, because the IDE doesn’t yet recognize this class as runnable.

4. Take a look at the declaration of **main** in the source file. Remember that there is a strict definition for the signature of the main method for a Java application class: it must be public, static, return nothing and take a String array as its argument. What is missing here?
5. Add the **static** keyword to the list of method modifiers:

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

6. Compile and run again. Well, this time it did actually run, but the results are a bit disappointing. It just says "Hello" – how ambitious is that? The application is designed to offer two new features: multiple languages and a switch between terse and verbose greetings. Let's try that out, as shown below, or choose **Run | Run configurations** and add the arguments **-v French** there.

run -v French

Comment allez-vous?

7. Better! Try out other options, though, and you’ll note that the application has a lingering problem:

run German

(no output at all)

8. Look in the source file and find the literal string “German”. You don’t need to know Java to debug this one! The string is misspelled in the source code as “Germen”. Fix the string, rebuild and test:

compile

run German

Guten tag

Note that the compiler can’t help you with this sort of thing. It can check method names, data types, basic expression syntax, and so on, but once you put things in quotes, it has no way of knowing whether the string you create is “right” or “wrong.” In fact a fair amount of language and application design have been dedicated to putting more of your code in forms that the compiler can check. Later in the course we’ll see a technique for representing strings such as “French” and “German” as compiler-checkable identifiers.



CHAPTER 2

LANGUAGE FUNDAMENTALS



OBJECTIVES

After completing “Language Fundamentals,” you will be able to:

- Write well-formed Java code using **expressions** and **declarations**.
- Write expressions that include unary, binary, and ternary operators and object method calls.
- Convert string values to numbers.
- Write, compile and test a simple Java application.

Zooming In (Ignorance is Bliss)

- We're going to approach the Java language gradually; we'll focus tightly on some of the simplest aspects of the language first.
- Over this and the next two chapters we'll consider:
 - Basic **expression grammar**
 - **Data types** and **type conversion**
 - **Flow control** for procedural/algorithmic programming
- This means ignoring many language features in the early going.
- However, some of these features will be evident in the source code anyway.
 - For instance, it's unavoidable that we code in **classes**, but we don't want to jump into the many object-oriented features of Java just yet; one thing at a time!
- For these next few chapters we'll couch all our Java code in a simple framework that leaves OO out of the picture:
 - A single Java class will offer a **main** method.
 - There may be other methods in the class – like **main**, these will be **static**. Otherwise, **main** would not be able to call them, for reasons we'll consider in a later chapter on OO Java.

An Expression-Based Language

- Like most high-level languages, Java is “expression-based,” and the bulk of any Java source file consists of various types of **expressions**.
- Expressions can be as simple as “ $x + y$ ” or they can be very complex indeed.
 - The result of one expression can be used as part of another – this is called **nesting** and results in **nested expressions**.
 - “ $5 * (x + y)$ ” is a nested arithmetic expression.
- Even top-level method calls are considered a kind of expression, as they can potentially be nested in other expressions.
 - Many code constructs that we might be tempted to call “statements” because they don’t seem to return any value are actually expressions that are simply being executed without their values being captured.
- We’ll look at the formal grammar for expressions in a moment.
- First, let’s consider the most basic building blocks of Java source code: **identifiers**, **literals**, and **operators**.

Identifiers

- An **identifier** is a bit of text that uniquely identifies some logical entity useful to the source code.
 - The something could be a class, a method, a field, or a local variable, and there are several other possibilities we've not seen yet.
 - The scope under which an identifier must be unique depends on the thing being named.
- An identifier as source text must follow certain rules:
 - It must have at least one character, but otherwise is of **unlimited length**.
 - It must **begin** with either a **letter**, the **underscore** character `_`, or the **dollar-sign** character `$`. This distinguishes it from various literal types, many of which begin with digits, and from operators, which use other characters.
 - Any remaining characters must be letters, underscores, dollar signs, and **digits**.
 - The identifier must not be one of Java's **reserved words**. A list of these is available near the end of the chapter.
 - Note that any Java token can include **Unicode** characters, so identifiers can be defined using non-ASCII characters, and definitions of "letter" and "digit" are as per Unicode.
 - Also, identifiers are evaluated and compared in a **case-sensitive** manner; this is true throughout Java.

Examples of Identifiers

- Legal identifiers:

```
me
myShadow
myShadow2
__guyDownTheStreet$andHisLittleDog_too
Renée
garçon
```

- Illegal identifiers:

```
myShadow.2    // Embeds the . separator
2MyShadow     // Begins with a digit
char          // This is a reserved word
```


Literals

- A **literal** is a value stated directly in the source code.
- Base-ten numeric literals are defined in Java using digits, optional plus/minus sign, and decimal points:

5	5.0	0
8933	0.23332	-70

- Other bases are recognized (for integers only) by the starting character(s):

0b11010111	// binary	(b or B is okay)
023	// octal	
0xFF14	// hexadecimal	(x or X is okay)

- Character literals use single quotes, and can use escape sequences including tabs, new-line characters, and Unicode:

'a'	' '	'\"'
'\t'	'%'	

- String literals can include zero-to-many characters; these use double quotes:

"Hello"	"Unicode string\u78D6"
""	

- There are some literals defined by reserved words:

true	false	null
------	-------	------

- We will study the exact syntax of literals per data type in the next chapter.

Operators

- Java defines a finite set of **operators** which can be used to operate on one, two, or three **operands**, thus forming a single expression.
 - There are binary **mathematical** operators: `+` `-` `*` `/` `%`
 - Unary operators to increment or decrement: `++` `--`
 - There are **comparison** operators which evaluate to a boolean value: `<` `==` `>` `<=` `>=` `!=`
 - There are **bitwise** operators that operate on integral numbers: `&` `|` `~` `^` `<<` `>>` `>>>`
 - These should not be confused with **logical** operators, which combine and return boolean values: `&&` `||` `!`
 - **Assignments** are operators, too: `=` as well as several derivatives such as `+=` `-=` `*=` that perform some other operation and store the result in the left-hand operand.
 - There is one ternary **flow-control** operator `?:`, which evaluates the first operand as a boolean condition and executes either the second or third operand, but not both, based on the state of the condition.
- The dot **separator** evaluates to a reference to a member (right-hand operand) of an object or class (left-hand operand), as in `myObject.doSomething()`.
- See the reference at the end of the chapter for a complete ordered list of Java operators and their evaluation precedence.

Declaring and Invoking Methods

- A method **declaration** is composed of:
 - Optional **modifiers** such as **public**, **private**, **static**
 - The **return type**, which can be any Java data type or **void**
 - The method **name**
 - A **parameter list** of zero to many parameters, each of which has a type and name
 - Optionally, declarations that the method may throw certain **exceptions** – we will cover exceptions later in the course
- The declaration is followed by a block of Java code that will be carried out when the method is **invoked**.

```
public class MyClass
{
    void doSomething () { ... }
    public int calculate (int x) { return x * x; }
}
```

- To invoke a method, state its name followed by a pair of parentheses.
 - If the method takes any **arguments**, provide values in the parentheses as a comma-separated list.
 - If the method returns something other than **void**, it can be used as an operand for an operator, or as an argument to another method call.

```
x = returnSomeNumber ();
y = calculate (5);
```

Variable Parameter Lists

- A method can be defined to take an indeterminate number of arguments.
- This feature is commonly known by the shorthand term **varargs**.
- It is accomplished via a new syntax: the final (or only) parameter in a method's list can be defined with a type name followed immediately by marks of ellipsis – three dots.

```
public void addAll (Object... elements);  
public static String concat  
    (String first, String... allOthers);
```

- The caller can then list out zero to many arguments in the method invocation.

```
myCollection.addAll (thisObj, thatObj);  
String total = concat ("Total is ", val, ".");
```

- Processing of variable parameter lists will be a subject for a later chapter; first we need to know how to process arrays ...

Expressions

- Literals and identifiers can form expressions by themselves.
- An expression can also be formed by combining other expressions:
 - By performing an **operation** on one, two, or three other expressions
 - By calling a **method** that accepts arguments, since these arguments are themselves expressions
- All expressions have a type to which they evaluate, and can be nested to arbitrary depth.

```
5
x
x + 5
x = 5
y = x = 5;
y = (x = 5) + 4;
x < y
x == y
myPoint;
someObject.doSomething ();
otherObject.doSomethingElse (5, x);
otherObject.doSomethingElse
    (5, calculate (x, y));
```

Evaluating Expressions

- Expressions are **evaluated** at runtime based on rules of **precedence**.
 - Every means of combining two or more other expressions has an assigned precedence.
 - Those with higher precedence are evaluated first. For instance, multiplication and division operators precede addition and subtraction.
 - If operators in a series have equal precedence, the interpreter will work left-to-right.
 - Parentheses can be used around an expression to force it to the highest precedence.
 - Thus the two expressions below evaluate differently:

$5 + 4 * 6$
 $(5 + 4) * 6$

- See the reference page at the end of this chapter for a complete rundown of operators in order of precedence.

Declarations

- One of the relatively few code constructs that is not an expression is the **declaration**.
- A programmer declares a variable in order to store or refer to a value, defining the type of the variable and giving it a name.
 - The name must be – you guessed it – a legal identifier.
 - There are many possibilities for type. We'll see some examples in this chapter and in the next chapter consider data types in depth.
 - An initial value may or may not be assigned:

```
int x;  
boolean result = trySomething (max (5, y = 4) + 3);
```

- A declaration is not an expression because it cannot be evaluated to a value, nor be used in another expression.
 - Expressions can be used in declarations, however, as we see above.
 - And an **assignment** is indeed an expression: **y = 4** above evaluates to 4, and this becomes the second argument in the call to the method **max**.

Grammar for Method Code

- Building up from there: method code is always a series of declarations and expressions, separated by semicolons, along with flow-control constructs.
- Flow control is still a couple chapters off; but one important element of flow control worth covering now is the **return** from a method.
 - Methods whose return type is anything other than **void** must return a value from **all possible control paths**.

```
return 5;  
return true;  
return (x + y);  
return success ? "Success!" : "Failure!";
```

- The value must be of the return type declared in the method signature. That is, a method declared to return a number can't actually return a string, and the compiler will check for this.
- A **void** method can use **return** simply for flow control – for instance to terminate the execution of method code prematurely if a requirement is not met – or it can leave it out entirely.

Expressions

EXAMPLE

- The **Expressions** project includes a single Java application class.

– See `src/Expressions.java`:

```
public class Expressions
{
    private static int five () { return 5; }

    private static int increment (int x)
    {
        return x + 1;
    }

    public static void main (String[] args)
    {
        System.out.println ("Expressions:");
        System.out.println (" 5 = " + 5);
        System.out.println (" 5 + 4 = " + (5 + 4));
        System.out.println
            (" 5 * 4 + 2 = " + (5 * 4 + 2));
        System.out.println
            (" 5 * (4 + 2) = " + (5 * (4 + 2)));
        System.out.println
            (" 5 + five () = " + (5 + five ()));
        System.out.println (" 5 + increment (1) = " +
            (5 + increment (1)));
        ...
    }
}
```

- As you can see, the **main** method simply follows a script of increasingly complex expressions, displaying the expression syntax and then the runtime value.

Expressions

EXAMPLE

- Take a few minutes to review the output from this code, and to explore the various types of literals and operators used here.

Expressions:

```
5 = 5
5 + 4 = 9
5 * 4 + 2 = 22
5 * (4 + 2) = 30
5 + five () = 10
5 + increment (1) = 7
increment (5) = 6
increment (five ()) = 6
increment (increment (five ())) / 2 = 4
(5 == 4) = false
(5 > 4) = true
!(5 == 4) = true
((5 == 4) && (5 > 4)) = false
((5 == 4) || (5 > 4)) = true
(!(5 == 4) && !(5 > 4)) = false
```

```
x = 1
y = 2
x + 1 = 2           (x = 1)
x++ = 1             (x = 2)
++x = 3             (x = 3)
increment (x) = 4    (x = 3)
increment (x--) = 4  (x = 2)
increment (--x) = 2   (x = 1)
increment (x = five ()) = 6 (x = 5)
(x = 4) = 4
(x = 4) + 1 = 5
(x *= 7) = 28
(y = (x = 3) + 2) = 5
y % 3 = 2
```

Comments

- You will have noticed one other type of code construct in this and previous source files: **comments**.
- Comments are used to document the code, in order to explain what a method does or how it does it.
- Comments are ignored by a Java compiler.
- Java provides three different syntaxes for comments:
 - The “C++-style” comment starts with a **double-slash**. Anything remaining on that text line is considered a comment.

```
x = getTheValue (); // Defaults to -1
```

- The “C-style” sequence starting with `/*` and ending with `*/` can mark off text which may include line breaks. It’s most useful for longer passages, or to comment out something in the middle of a text line.

```
/*  
Start off with one and let the recursive method  
do its stuff:  
*/  
x = 1;  
myAlgorithm (x);
```

- The “doc comment” starts with `/**` and ends with `*/`. This is like a C-style comment but flags the comment text for use in automatically generating HTML documentation from the source file using a tool called **javadoc**.

Annotations

- A Java **annotation** modifies the package, class, interface, method, field, parameter, or local variable that follows it in the source file.
- Annotations declare **metadata** – that is, facts that can't be captured in the primary type model of Java.
- The annotations model is extensible.
 - There are **Java SE standard annotations**.
 - There are **Java EE standard annotations**.
 - Any API or application can define **custom annotations** and then apply them to other entities.
- We'll see just a little bit of this annotations model in play through this course.
- For the moment, we only catalog it as part of the Java syntax.
 - We identify an annotation based on the leading @ symbol:

```
@MyAnnotation public void foo ();
```

- An annotation precedes the thing it annotates.
- Annotations can have **attributes**, stated within parentheses:

```
@MyAnnotation (a1="string", a2=OtherClass.class);  
public class MyClass  
...
```

Lambda Expressions and Method References

- As of Java 8 there are some new syntax options, having to do with functional programming.
- We won't treat these in depth until much later in the course, but we mention them here for completeness from the perspective of legal Java syntax.
- In certain cases where a reference to a **functional interface** is expected, you can write a **lambda expression**.
 - This is a shorthand for method parameters and method body, and is identifiable by its use of the `->` operator:

```
callSomeMethod (() -> myExistingValue);  
callSomeMethod ((x) -> x + 1);  
callSomeMethod (x -> x + 1);  
callSomeMethod ((x,y) -> x + y);  
callSomeMethod ((x,y) -> { ++x; return x + y; });
```

- The compiler expands a lambda expression to a complete **anonymous class**, with a single **method** implemented as defined in the lambda expression's body.
- In similar circumstances you can provide a **method reference**.
 - This is yet another way of promoting a method to act as a complete, anonymous class.

```
callSomeMethod (MyClass::myMethod);  
callSomeMethod (myObject::myMethod);
```

- You may also see code that refers, indirectly, to a class' constructor by using the keyword **new**:

```
callSomeMethod (MyClass::new);
```

Converting Strings to Numbers

- Just as a practical matter for this chapter's lab exercise, we'll now learn a basic technique for interpreting user input.
- Command-line arguments are passed to your program as an array of strings, as described in the previous chapter.
- What if you want to process these arguments as numbers?
- For integers, the magic words are

`Integer.parseInt (anyString)`

- This is a call on a static method of the Core API class **Integer**, and it takes a string and returns an integer.
- Thus, to parse command-line arguments as numbers, one would write code like the following:

```
int x = Integer.parseInt (args[0]);
```

- We'll come back to this in much greater depth later in the course.
 - There are other classes such as **Long**, **Double**, etc.
 - They have other utility methods, such as one to produce a string representation of a number.

Basic Arithmetic

LAB 2

Suggested time: 30 minutes

In this lab you will build a Java application class called **DoMath**. This will be the first exercise in building a Java class from scratch – a process that will become second-nature after a while. The **main** method will parse the command line looking for two numeric arguments, perform a number of simple mathematical calculations on them, and report the results. For this lab, all numbers will be of integer type (**int**); in the next chapter we will consider different numeric types and means of converting between them.

Detailed instructions are found at the end of the chapter.

Reference – Java Operators and Precedence

- Java observes the following operators, in order of precedence:

++	Pre- or post-increment
--	Pre- or post-decrement
+, -	Unary operators for sign
~	Bitwise complement
!	Logical NOT
(type)	Any type conversion
*	Multiplication
/	Division
%	Modulo
+	Addition
-	Subtraction
+	String concatenation
<<	Bitwise shift left
>>	Bitwise shift right, sign extension
>>>	Bitwise shift right, zero extension
<, <=, >, >=	Comparisons
instanceof	Type identification
==	Equivalence
!=	Inequivalence
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
&&	Logical AND
	Logical OR
?:	Conditional evaluation
=, +=, -=, *=, /= >>=, >>>=, &=, =, ^=	Assignment operators

Reference – Java Separators

- **Separators** are characters whose use is formalized as part of the Java language grammar.
- They are not operators *per se*.
- These include curly braces { }, which are used to define class contents and blocks of code.
- The period or dot is another example of a separator.

Reference – Java Reserved Words

- Avoid using any of the following as identifiers in your code, as they are reserved as part of the Java language and would be interpreted as such:

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>package</code>	<code>this</code>
<code>assert</code>	<code>default</code>	<code>goto*</code>	<code>private</code>	<code>throw</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>protected</code>	<code>throws</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>public</code>	<code>transient</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>requires</code>	<code>try</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>var</code>
<code>catch</code>	<code>exports</code>	<code>int</code>	<code>short</code>	<code>void</code>
<code>char</code>	<code>extends</code>	<code>interface</code>	<code>static</code>	<code>volatile</code>
<code>class</code>	<code>final</code>	<code>long</code>	<code>strictfp</code>	<code>while</code>
<code>const*</code>	<code>finally</code>	<code>module</code>	<code>super</code>	
	<code>float</code>	<code>native</code>	<code>switch</code>	
		<code>new</code>	<code>synchronized</code>	

* Not implemented, just reserved.

- Also beware of the following tokens, which are literal values and thus are off-limits as identifiers:

<code>true</code>	<code>false</code>	<code>null</code>
-------------------	--------------------	-------------------

SUMMARY

- **Basic grammar, identifier syntax, operators, literals, and keywords form the basis for a more complete understanding of the language.**
 - There is much more to discover, both in procedural programming and object-oriented techniques.
 - Still, a great deal of Java code does take the form of simple-to-complex expressions, and understanding the precise process of evaluating those expressions is critical to writing effective code.
- **We will proceed to study data types and flow control, which will provide a clearer picture of the procedural aspects of the language.**

Basic Arithmetic

LAB 2

In this lab you will build a Java application class called **DoMath**. This will be the first exercise in building a Java class from scratch – a process that will become second-nature after a while. The **main** method will parse the command line looking for two numeric arguments, perform a number of simple mathematical calculations on them, and report the results. For this lab, all numbers will be of integer type (**int**); in the next chapter we will consider different numeric types and means of converting between them.

Lab project: **Math/Step0**

Answer project(s): **Math/Step1**

Files: * to be created
 src/DoMath.java

Instructions:

1. Create a new source file **DoMath.java**. From the command line, create the new file, and write the skeletal code for a class **DoMath** with a **main** method. Save this file and assure that you have a clean compile so far.
2. First add code to parse the command line for two arguments into new variables **one** and **two**:

```
int one = Integer.parseInt (args[0]);  
int two = Integer.parseInt (args[1]);
```

3. Write the result of one-plus-two to the console, as follows:

```
System.out.println ("Sum of numbers = " + one + two);
```

It may seem odd, but this will work, because (a) the expression starts with a literal string, and (b) Java interprets the + operator differently when the left-hand operand is a string: it performs concatenation. So the whole expression will be a string, and this is satisfactory for a call to **println**.

4. Now test it out by running the application. Hmm ... can Java not do math?

```
run 2 2  
Sum of numbers = 22
```

Basic Arithmetic**LAB 2**

5. The problem is operator precedence. We need the sum of the numbers to be performed prior to the string concatenation, but since it's all plus signs the interpreter moves from left to right – how does it know that's not what we wanted? Thus we get a (string-plus-two)-plus-two as the string-plus-22.
6. Fix this by parenthesizing the numeric part of the expression, which will force it to be evaluated first and nested in the string-concatenation process:

```
System.out.println ("Sum of numbers = " + (one + two));
```

7. Build and run again, and you should get a decent result.

run 2 2

```
Sum of numbers = 4
```

8. Now start writing out results of other computations: difference, product, division, and finally modulus, or remainder, using the % operator. Build and test – correct output will look something like this:

run 7 3

```
Sum of numbers = 10  
Difference of numbers = 4  
Product of numbers = 21  
Dividing numbers gives 2  
Modulus/remainder is 1
```




CHAPTER 3

DATA TYPES



OBJECTIVES

After completing “Data Types,” you will be able to:

- Identify and use in code all the Java primitive types.
- Describe the issues of converting values from one type to another:
 - What is a **widening** conversion?
 - What is a **narrowing** conversion?
 - How does the compiler treat each of them?
- Use enumerated types for finite sets of possible values.
- Use formatted output to produce clean, readable text for the end user.
- Describe the function of an object reference as a data type and/or variable.
- Create and manipulate Java strings.
- Create and populate arrays.

Strong Type Model

- Java compilers enforce a strong type model.
 - This means that every variable has a declared type, and that the compiler will check the type of any expression to which source code tries to assign that variable.
 - For instance the following will produce a compiler error:

```
String x = 5;
```

- The same strong type checking is applied to **fields**, method **parameters**, and method **return values**.
- Also some operators only work on certain primitive types. For example one can't logically combine numbers:

```
5 && 0 // illegal -- logical AND is for booleans  
5 & 0  // legal  -- this is a bitwise AND operation
```

- By comparison to the “weak typing” or “dynamic typing” common in interpreted languages, Java offers the advantage of aggressive type checking in compilation.
 - This can be irritating when getting used to the language, but it has the ultimate effect of **pushing errors to the front** of your development process.
 - That is, on the rare occasions when you do make a mistake and mis-assign something, the compiler will catch it instead of allowing it to occur at runtime, when it would be harder to debug, and possibly even damaging in some way.

Primitive Types

- Java supports as **primitive** types the following:
 - **Integral numbers** – there are four types here
 - **Floating-point numbers** – two more types
 - **Characters**
 - **Boolean** values
- **Primitive variables and expressions are always passed by value to methods, and returned by value as well.**
 - Thus a primitive-type method parameter can be modified within the method code with **no effect** on the variable that might have been used to provide the argument value.
 - This is technically true for object references, but perhaps misleading: the reference is passed by value, but that means that the object itself is passed **by reference**. More on this when we get to Java as an object-oriented language.
- **When used as fields, primitives are initialized by default to some zero representation when declared:**
 - Numerics will be **zero**.
 - Characters will be **ASCII zero**.
 - Booleans will be **false**.
- **There is no default initialization for primitives when used as local variables; they must either be initialized or assigned before their first use in the method.**

Type Conversion

- There are some sorts of type conversions that are viable, such as from one size of number to another.
- Java defines some type conversions as **widening** and some as **narrowing**.
- These terms mean different things for different types, but the general idea is that any type conversion moves from a **value space** of a certain size to another value space that is either larger or smaller.
 - A **widening** conversion can't possibly lose information, because the value being placed in a larger value space that is a **superset** of its source value space.
 - Compilers will perform widening conversions **implicitly**.

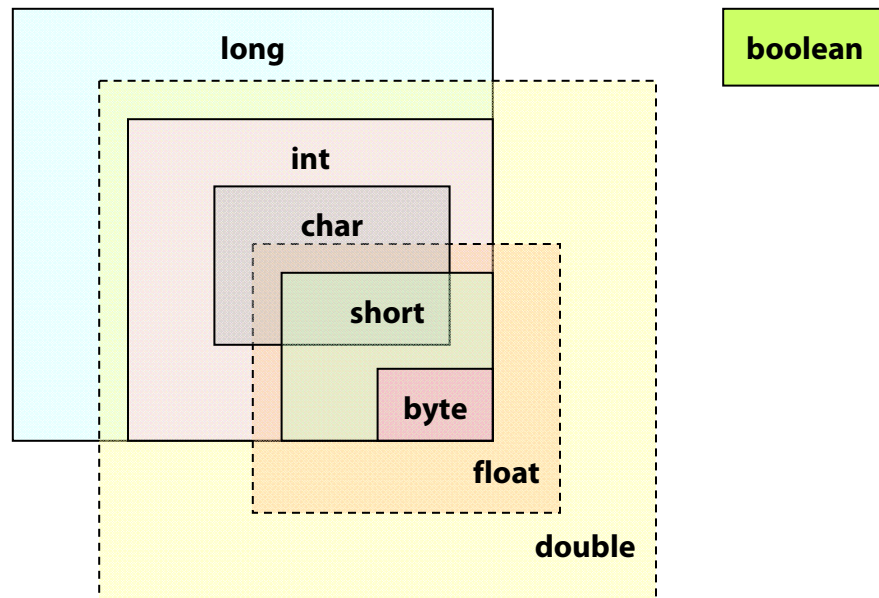
```
byte yourByte = 3;  
int myInteger = yourByte;
```

- A **narrowing** conversion is considered more dangerous, because there is a **loss of precision** in moving to a smaller value space that might not be able to accommodate a value from the larger source space.
- Compilers will perform narrowing conversions, but only **explicitly**; the source code must execute a **type cast** that assures the compiler that the value at runtime won't be truncated or corrupted.

```
yourByte = myInteger;           // Compiler complains  
yourByte = (byte) myInteger;    // If you say so ...
```

Value Spaces

- For any Java type there is a **value space**, which is the set of all possible values of that type.
 - The value space of the **byte** type includes all integers from -128 to +127, inclusive of both.
- We'll look at the value spaces and conversion rules for some of the primitive types, and then see some examples of implicit and explicit conversions.
- A summary diagram (not to scale!) is below.



- So for example any **byte** value can be captured as a short, **int**, or **long**; but not all **long** values can be held in an **int**.
- Note that the **boolean** value space is completely separate.
- This doesn't tell the whole story; there are other differences between the primitive types than just the values they can hold.
- But it's a pretty good guideline to what can convert to what.

Integral Types

- Integers of several sizes (or **widths**) are supported as the following primitive types in Java:
 - **byte**, which has a width of 1 byte, or 8 bits
 - **short**, which has a width of 2 bytes, or 16 bits
 - **int**, which has a width of 4 bytes, or 32 bits
 - **long**, which has a width of 8 bytes, or 64 bits
- All integral types in Java are **signed**: the value space runs from -2^{n-1} to $(2^{n-1} - 1)$ where n is the size in bits.
 - For instance a **byte** can be anywhere from -128 to 127 .
- Literally-expressed integers are assumed to be of type **int**, unless suffixed with an upper- or lower-case L.
 - Lowercase 'l' looks a lot like the digit '1' in code fonts, and so is generally discouraged.
- Conversions between integral types can be made.
 - A **widening** conversion is from a smaller numeric type to a larger one, and can be performed implicitly:

```
int x = 5;
long y = x;
```
 - A **narrowing** conversion goes the other way, and must be performed explicitly, with the use of a typecast:

```
long x = 5L;
int y = (int) x;
```

Floating-Point Types

- Integers of several sizes (or **widths**) are supported as the following primitive types in Java:
 - **float**, which occupies 4 bytes
 - **double**, which occupies 8 bytes
- These types are also signed, and their value spaces are comparable to the same-sized integral types, with a couple of expansions:
 - They can support digits to the right of the decimal.
 - They support IEEE-754 constants for certain real-not-rational numbers, including “not-a-number” and positive and negative infinity.
- Literally-expressed floating-point numbers are assumed to be of type **double**, unless suffixed with an upper- or lower-case F
 - here lower-case is preferred.

Floating-Point Types

- The same rules apply to widening and narrowing conversions for floating-point numbers as to integers:

```
float x = 5f;  
double y = x;  
float z = (float) y;
```

- Converting a floating-point value to an integer is a narrowing conversion – naturally, because the part of the value to the right of the decimal point will surely be lost.
- Converting the other way is a subtler thing.
 - The compiler treats any conversion to a floating-point type as a widening conversion.

```
double a = 123456456165212345L;  
float b = 123456789;  
float c = 123456456165212345L;
```

- The last example above is especially surprising, since a long can be 8 bytes and a float is only 4!
- And, since float and double support **not-a-number** and **infinity** representations, clearly they can't hold every possible int and long value as well.
- So there is a caveat here: any conversion to floating-point is considered a widening conversion – so doesn't require any explicit type cast – but is also understood to be **imprecise**!
- The **least-significant digits** of the original value may be only **approximate** in the resulting floating-point representation.

Characters

- Java represents individual characters with the `char` type.
- The size of a **char** is two bytes, because it represents a Unicode character.
- Characters can be converted to larger integral types implicitly, and vice-versa explicitly.

```
char c = 'A';  
int x = c;  
char d = (char) x;
```

- Although characters and short integers both occupy two bytes, they have different value ranges.
 - Characters are **unsigned** – representing only zero and positive integers – so as to support the entire Unicode character set.
 - Therefore a character could hold a value greater than the maximum value for a **short**, and a **short** can hold negative numbers, which a character can't.
 - Even the smaller **byte** can represent negative numbers, and **char** values are all positive.

Booleans

- Java represents boolean values, which can either be true or false, with the **boolean** type.
- Booleans in Java are completely distinct from all numeric types.
- You cannot convert a number or object reference to a boolean expression – even explicitly.

```
boolean b = true;
if (b)
{
    int x = 5;
    int y;
    if (x) // ILLEGAL
        y = 3;
    if ((boolean) x) // NO GOOD EITHER
        y = 3;
    if (x != 0) // fine
        y = 3;
}
```

- There are two boolean literals: **true** and **false**.

Enumerated Types

- Java also supports an **enumerated type**.
- This allows an application to define a finite set of values that are useful in its work.
 - The enumeration itself, and then each value, can be given a unique name, rather than an arbitrary integer or string value.

```
enum CMYKColor { Cyan, Magenta, Yellow, Black };
```

- These names can be checked by the compiler, where an integer or string could not.
- So code that mistypes or misuses a value will be flagged with a compile-time error, rather than passing the compiler and then failing at runtime.

```
void processLayer (CMYKColor color, int[][] layer)
{
    if (color != CMYKColor.Black)
        applyColorAdjustment (layer);
    ...
}
```

- This is known as **type-safe enumeration**.
- An enum can be defined at the “top level” – that is, in its own source file as a peer of public classes – but more commonly it is defined as a member of a class that uses it.
- We’ll consider enums in greater detail in a later chapter.

Data Types

EXAMPLE

- The application in **DataTypes** goes through a series of exercises on each of the types we've discussed so far.
 - It declares a variable of the appropriate type.
 - It explores legal assignments to the variable.
 - Some illegal assignments are commented out.
 - It prints the final value to the console – for example:

```
short s = 7;
s = 32767; // this is 2 ^ 15 - 1
s = -32768; // this is negative 2 ^ 15
// Would not compile: s = 32768; (this is 2 ^ 15)
// Would not compile: s = -65000;
System.out.println
("Final value of short    s = " + s);
```

- The application then exercises an enumerated type **Color**, which is declared at the bottom of the source file:

```
enum Color { red, green, blue, black, white };
```

- Finally, the application summarizes the legal value spaces for each numeric type.

Data Types

EXAMPLE

- Build and run the application – the output is:

```
Final value of byte      b  = -128
Final value of short     s  = -32768
Final value of int       i  = -2147483648
Final value of long      l  = -9223372036854775808
Final value of float     f  = -5.4E-5
Final value of double    d  = 1.844674407370955E7
Final value of char      c  = *
Final value of boolean   bo = false
String value of Color   co = blue
Ordinal value of Color   co = 2
```

Legal value ranges:

```
Byte      -128 to 127
Short     -32768 to 32767
Integer   -2147483648 to 2147483647
Long      -9223372036854775808 to
9223372036854775807
Float     1.4E-45 to 3.4028235E38
Double    4.9E-324 to 1.7976931348623157E308
```

Data Types

EXAMPLE

- Take a few minutes to review this code and assure that you understand it – especially why certain assignments are legal and some are not.

- Note the use of binary literals to initialize the byte **b**:

```
byte b = 0;  
b = (byte) 0b01111111;    // this is  $2^7 - 1$   
b = (byte) 0b10000000;    // this is negative  $2^7$ 
```

- Also, see the odd new practice, legal for Java 7, of embedding underscore characters in integer literals – supposedly for clarity, though it looks quite odd!

```
1 = 9_223_372_036_854_775_807L;  
1 = -9_223_372_036_854_775_808L;
```

Type Conversions

EXAMPLE

- A second application in **Conversions** explores implicit and explicit type conversions.
 - First it creates one variable of each integral type, and then assigns values one way and then the other. This illustrates the difference between widening (the first set) and narrowing (the second), as far as the compiler is concerned.

```
byte b = 5;
```

```
short s = b;
```

```
int i = s;
```

```
long l = i;
```

```
i = (int) l;
```

```
s = (short) i;
```

```
b = (byte) s;
```

- Then a value that actually wouldn't fit in the smaller types is set into the long `l`. The successive conversions and assignments succeed, because the compiler has been told that they're safe, but at runtime the value will actually be truncated successively.

```
l = 0x1111111111L;
```

```
i = (int) l;
```

```
s = (short) i;
```

```
b = (byte) s;
```

```
System.out.println ("Long: " + l);
```

```
System.out.println ("Converted to integer: " + i);
```

```
System.out.println ("Converted to short: " + s);
```

```
System.out.println ("Converted to byte: " + b);
```

Type Conversions

EXAMPLE

- It does the same for the floating-point types.
 - Note the last section here, in which both conversions between **float** and **long** are considered narrowing and require a type cast:

```
l = (long) f;    // might lose fractional part
f = l;           // might lose overall precision,
                  // but not range
```

- The next section exercises the **Color** enumeration.
 - In neither direction is conversion between the enum and an integer allowed.
 - The integer representation of a given enumerated value's (zero-based) **ordinal position** is available:

```
i = color.ordinal ();
```

- So is a string representation:

```
String colorName = color.name ();
```

- The final section of this example concerns object types and conversions – we'll come back to this in a few chapters.
- Build and run the application – output is:

```
Long: 4581298449
Converted to integer: 286331153
Converted to short: 4369
Converted to byte: 17
Ordinal position of green is 1
Starting object reference conversions ...
Object reference conversions complete.
```

Formatted Output

- Java provides a simple means of formatting output to a stream.
- The **PrintStream**, which you'll recall is the type of **System.out**, offers a method **format**.
 - This is another example of varargs: the **format** method takes a formatting string as its first parameter, and then any number of arguments after that.

```
System.out.format ("%8d %s%n", 77, "Hello");
```

- The call shown above produces the following – note the leading spaces to observe an eight-character “width” for the numeric argument:
77 Hello
- The **String** class offers a **format** method as well: it works the same way but **returns** the string instead of producing it to any particular stream or output channel.

Formatting Strings

- The formatting string passed to **format** consists of literal text and **fields**.
- Each field in the string must correspond to an argument passed to the method – with a few exceptional field types that produce special characters and don't need a value to be provided.
- Common field types are
 - **%d**, which formats an integer
 - **%s**, which formats a string
 - **%f**, which formats a floating-point number
 - **%n**, which produces an end-of-line sequence appropriate for the runtime operating system
- Most fields can be modified in a few simple ways:
 - A number preceding the one-letter field code defines the field width, which is a minimum number of characters to be produced by the field.
 - A hyphen preceding the number indicates the field will be **left-justified**, where **right-justified** is the default, if a width has been specified – even for strings.

Formatting Examples

- The following pairs of lines show the Java code to format certain values and the resulting output:

```
System.out.format ("%d %s %s%n",  
    77, "Hi", "there");  
77 Hi there
```

```
System.out.format ("%4d %8s %8s%n",  
    77, "Hi", "there");  
    77          Hi          there
```

```
System.out.format ("%4d %-8s %-8s%n",  
    77, "Hi", "there");  
    77 Hi          there
```

```
System.out.format ("%16s $%,8.2f",  
    "Dog bone", 70.0);  
Dog bone                $          70.00(and no line break!)
```

```
System.out.format ("The value is $%,-1.2f.", 70.0);  
The value is $70.00.
```

- You can find a good tutorial on formatted output here:

```
https://docs.oracle.com/javase/tutorial  
/java/data/numberformat.html
```

- Formatted output is quite simple to use, but for many purposes it is still simpler and easier to **print** or **println**.
- Formatted output is especially helpful in establishing regular widths for fields, and in justifying the output text, so that columns of values are neatly aligned for the reader of the output.
- We will use a combination of the two practices in this course.

More Math

LAB 3A

Suggested time: 30 minutes

In this lab you will refine your **DoMath** application to use types other than **int** to capture certain values. This will enhance the precision of certain computations, and it will make new ones possible that use utility methods in the Core API – such as calculating the hypotenuse of a right triangle and getting the circumference of a circle.

Detailed instructions are found at the end of the chapter.

Core API Documentation

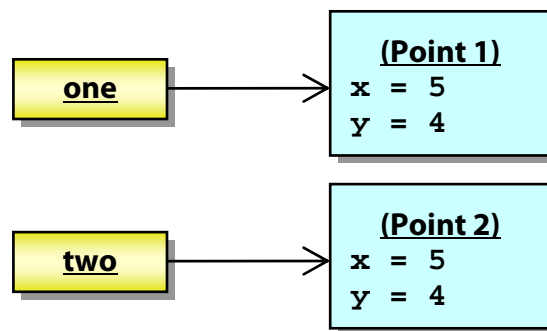
- In the previous lab, we introduced a new Core API class, **Math**, and used a few of its utility methods.
- How would you know about this class if the lab instructions hadn't told you?
- How would you know what methods it offers, their signatures, what they do, etc.?
- Oracle provides a full package of **javadocs** for the Core API.
 - This is HTML documentation generated from the Core API's source code, using the **javadoc** utility that we discussed in the previous chapter.
 - You can find it online – for example, for Java 16:

`https://docs.oracle.com/en/java/javase/16/docs/api/`

- It is also embedded in IDEs, and typically is provided proactively as a popup or tooltip when you mouse-hover on a symbol, or use certain (tool-specific) keyboard shortcuts.

Object References

- There is one and only one type in Java to support the use of objects, which is the **object reference**.
- Any declaration of a variable whose type is a class results in the allocation of an object reference.
- The reference may refer to an object in memory, or it may be the literal **null**.



- For the most part, we'll leave object references alone in this chapter.
- However, strings are implemented in the Java Core API as objects, whose type is the class **String**.
- We'll look at working with object references a bit, so as to avoid a few all-too-common pitfalls in handling strings in source code.

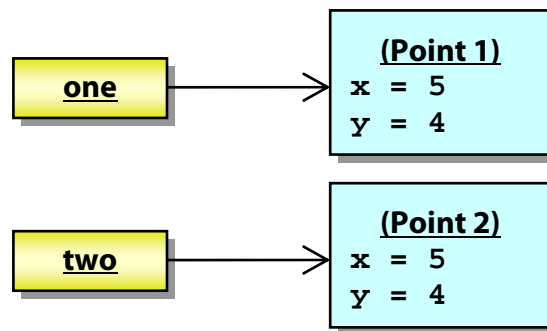
Comparing References

- Object references are variables in their own right, and can be passed to methods, assigned, and compared.
 - They are initialized to **null** where primitives would be initialized to zero.
- If you use the equivalence operator to compare two object references, do not be fooled into thinking that you are in any way comparing the underlying objects.

- Consider the following code:

```
Point one = new Point (5, 4);  
Point two = new Point (5, 4);  
if (one == two)  
    System.out.println ("one equals two");
```

- What do you think will happen?

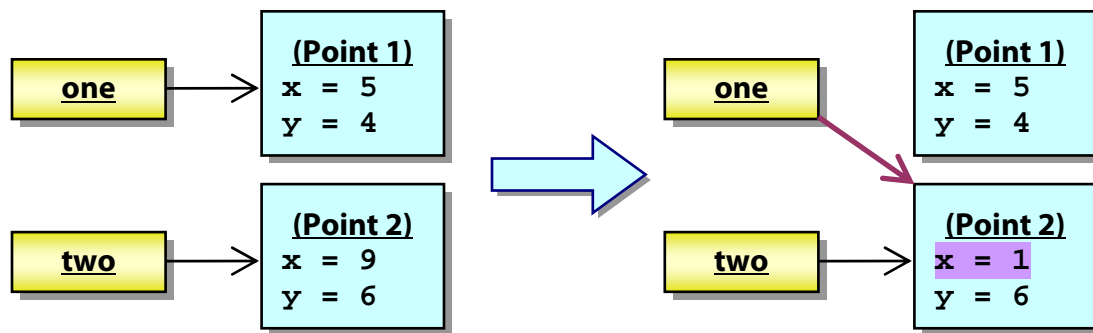


- The two objects are **equivalent**.
- But they are not **identical**.
- Comparing **object references** with **==** and **!=** can only establish **identity**.

Assigning References

- Similarly, it is important to understand the difference between assigning object references and copying object state.
- Consider the following code – what will be printed?

```
Point one = new Point (5, 4);  
Point two = new Point (9, 6);  
one = two;  
one.x = 1;  
System.out.println ("two.x = " + two.x);
```



- Assignment of an object reference does nothing at all to the previously referenced object. It reassigns the reference to another object. Here **two.x** will be 1.
- For testing object equivalence and copying of object state, there are methods on the root class **java.lang.Object**, which we will study in the next module.

Strings

- Java implements strings using a class **String**, but the compiler recognizes **Strings** as behaving in certain special ways.
- You can assign a **String** reference to a string literal without explicitly creating a new **String** instance – this is not possible with any other class:

```
String greeting = "Hello";
```

- String concatenation is possible using the `+` operator directly, as we've seen.
 - The compiler works some magic behind the scenes and the expression evaluates correctly.
 - This is also true for the assignment operator `+=`.

```
greeting += ", Java!";
```

- All other types, primitives and objects, can be implicitly converted to a string representation.

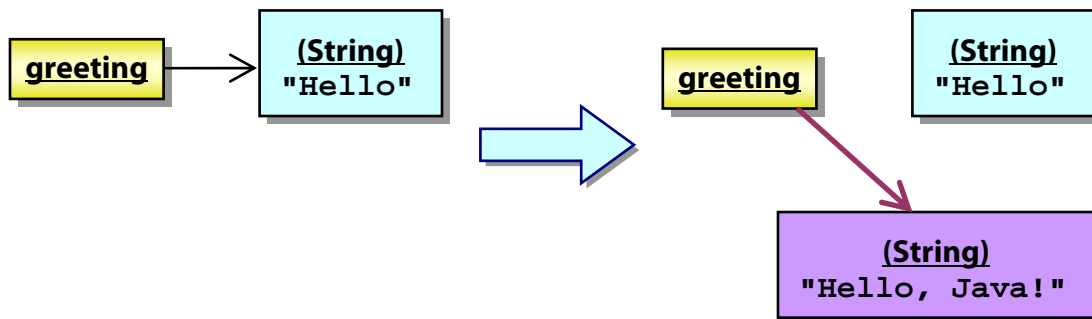
```
String report = "x = " + 5;
```


Strings are Immutable

- Perhaps the most slippery of these special behaviors is that strings are **immutable**.
 - Once allocated, a string will never change its value.
- This may be hard to process: after all, haven't we already seen strings being changed, by concatenating strings to them?

```
String greeting = "Hello";  
greeting += ", Java!";
```

- Well, yes, **greeting** now refers to a new value, "Hello, Java!"
- But, no, the original string was not changed!
- Rather, a new string was allocated in memory, with the new value, and **greeting** – again, an object reference, not an object itself – was re-assigned to that new string.



- Remember this: you are never changing a string in place, even if the code seems to suggest that this is happening:

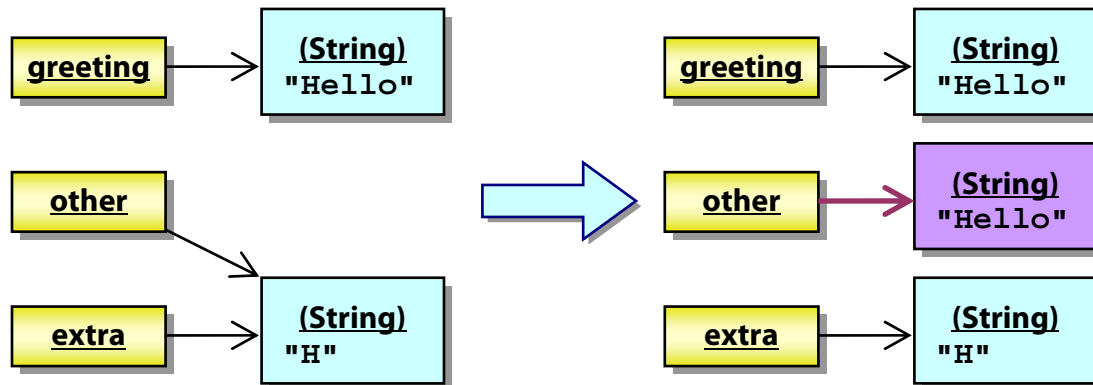
Comparing and Assigning Strings

- You can assign **Strings** to other strings and to literals easily.
- But as with ordinary object references, beware of what an assignment is actually accomplishing.
- Likewise, and more dangerously, you can compare two **String** references directly.

```
String greeting = "Hello";
String other = "H";
String extra = other;
other += "ello";
System.out.println ("Result: " +
    (greeting == other));
System.out.println ("Result: " +
    (other == extra));
```

- This looks natural, but will not give the results you might expect.
- Again, this code is comparing the **references**, which is to say it is evaluating **identity**, not **equivalence**.

Comparing and Assigning Strings



- First, **greeting** and **other** initially refer to two separate string literals, “Hello” and “H”.
 - Then **other** is assigned to a newly-computed string value – which happens to be the same as the value of **greeting**.
 - Still, they are two different objects, and so the first comparison would evaluate to **false**.
- Then, you might think that **extra** and **other** would refer to the same string object.
 - After all, one was assigned directly to the other!
 - But the concatenation operation on **other** results in a new string in memory, while **extra** still refers to the old string value.
- To compare two **Strings** by their contents – that is, to test for equivalence, and not just for identity – use the **equals** or **equalsIgnoreCase** methods.

```
System.out.println ("Result: " +  
    (greeting.equals (other)));
```

Asset Management

LAB 3B

Suggested time: 30 minutes

In this lab you will implement a simple application that combines some numerical calculation with string-building to produce a report on a single asset as described on the command line.

Detailed instructions are found at the end of the chapter.

Arrays

- Java supports arrays as the most direct means of collecting and indexing many values of similar type.
- Arrays are declared using square brackets after either the type or the variable or field name:

```
int[] integers; // preferred
int integers[];
```

- Read or write an element in the array by using square brackets after the array name, supplying a zero-based index:

```
integers[0] = 5;
int x = integers[2];
```

- You can check the length of an array using a built-in pseudo-field called **length**:

```
integers[integers.length - 1] // last element
args.length // number of command-line arguments
```

Allocating Arrays

- Allocate the elements of an array by:

- Initializing the values in the declaration time:

```
int integers[] = { 0, 1, 1, 2, 3, 5, 8 };
```

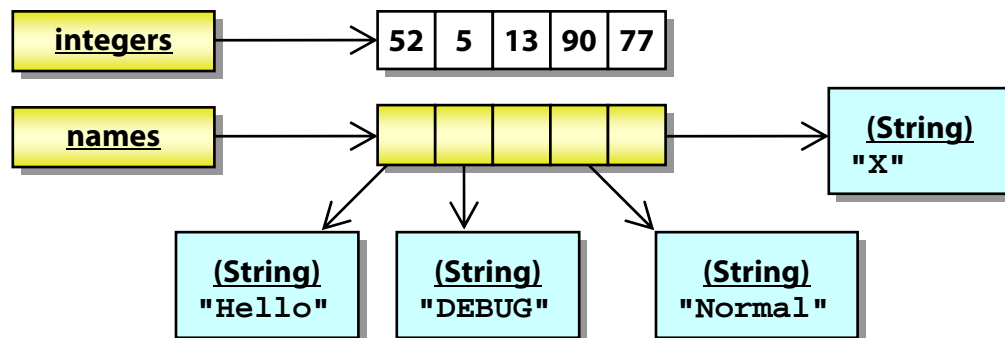
- Using the **new** operator – this can be done in the declaration or in any later assignment:

```
integers = new int[10];  
String[] names = new String[5];
```

- Note that while the array can report its length, **the length is not part of the type** Of an array reference.

- A reference can be assigned and re-assigned to arrays of different lengths – as we see with **integers**, above, which is initialized to an array of seven elements and then assigned to a new array of ten elements.
- Methods can take array references as parameters, and any actual array, of whatever size, can be passed as an argument.

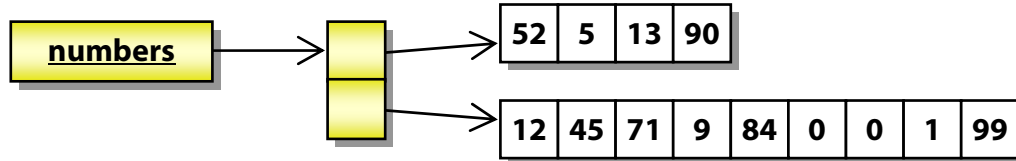
- Arrays are sometimes known as **pseudo-objects**, and like objects they are read and manipulated through references:



Allocating Arrays

- Multidimensional arrays are supported, including **non-rectangular** or **jagged** arrays:

```
int[][] numbers = new int[2][];  
numbers[0] = new int[4];  
numbers[1] = new int[9];
```



- Keep in mind when dealing with arrays of objects that you must explicitly allocate the array (space for the object references) and the objects themselves.

- This allocates an array, all of whose elements will be set to **null**:

```
String[] names = new String[3];
```

- These lines set values into the elements – leaving the third **null**:

```
names[0] = "Hello";  
names[1] = someOtherString;
```

Processing Arrays

- Almost all code that works with arrays needs to put every element in the array through some consistent process.
 - For instance you might want to print every value.
 - You might need to perform an operation on each element, or derive an aggregate value such as a sum.
- For the moment we'll let this problem sit.
- What we need are loops, and these are covered in the next chapter on flow control.

A Mileage Table

LAB 3C

Suggested time: 30 minutes

In this lab you will implement a jagged array to capture the data for a table of driving distances between a few cities. This is a natural application for non-rectangular arrays because the data set itself is really triangular – we have a grid of city-to-city but we don't need the distance from X to X, nor do we need to see the distance from Y to X once we've captured the distance from X to Y.

In this lab you will write the code to initialize the jagged array with mileage values, and do a quick test of finding a mileage in the array and printing it out. In the next chapter you will add more complex processing that uses loops to respond to user requests.

Detailed instructions are found at the end of the chapter.

SUMMARY

- Java supports several primitive types for numbers, characters, and boolean values.
- It is important to take care to differentiate between the manipulation or inspection of an object reference and of the object itself.
- Strings are represented in Java using the **String** class, which is for most intents an ordinary object type, but which the compiler recognizes and treats as a special case in some usages.
- Arrays can be allocated and populated in a variety of ways, including non-rectangular allocations.

More Math

LAB 3A

In this lab you will refine your **DoMath** application to use types other than **int** to capture certain values. This will enhance the precision of certain computations, and it will make new ones possible that use utility methods in the Core API – such as calculating the hypotenuse of a right triangle and getting the circumference of a circle.

Lab project: **Math/Step1**

Answer project(s): **Math/Step2** (intermediate)
Math/Step3 (final)

Files: * to be created
src/DoMath.java

Instructions:

1. As you noticed when testing the previous chapter's lab, using only integers loses precision over some operations. For instance, we know that 7 divided by 3 is not 2! To get a more precise result a floating-point type is needed. You can coerce the expression to type **double** by converting just the first of the two operands, as shown below. That is, a **double** divided by an **int** evaluates to a **double**.

```
System.out.println ("Dividing floating-point conversions = " +  
    (((double) one) / two));
```

2. Build and test at this point. The output is more precise – but maybe too precise!

```
run 7 3
```

```
...
```

```
Dividing floating-point conversions = 2.3333333333333335
```

3. This is a good application for formatted output. Change the final **println** to **format**, separate the text from the numeric expression as two distinct parameters, and add formatting fields, as shown. This will produce at least one digit overall – which is moot in this case – and exactly four to the right.

```
System.out.format ("Dividing floating-point conversions = %1.4f%n",  
    ((double) one) / two));
```

More Math**LAB 3A**

4. Build and test again, and you should see that the result has been formatted more cleanly:

```
run 7 3
...
Dividing floating-point conversions = 2.3333
```

5. Now build a new method into the class. Just after the class declaration and the initial open brace, declare the method **hypotenuse** to take two integers and return a **double**.

```
public class DoMath
{
    private static double hypotenuse (int a, int b)
    {
    }
```

6. Implement the method by first getting the sum of the squares of **a** and **b**. Then use the **Math** utility class (available as part of the Core API) to return the square root, as shown below.

```
        return Math.sqrt (sumOfSquares);
```

7. Call the method from the bottom of **main**, passing **one** and **two** as the arguments, and **format** the results to the console:

```
        System.out.format
            ("Dividing floating-point conversions = %1.4f%n",
             ((double) one / two));
        System.out.format ("Hypotenuse with sides given = %1.4f%n",
                           hypotenuse (one, two));
    }
}
```

8. Build and test at this point – you should see output like this:

```
run 7 3
Sum of numbers = 10
Difference of numbers = 4
Product of numbers = 21
Dividing floating-point conversions gives 2.3333
Hypotenuse with sides given = 7.6158
```

More Math**LAB 3A**

9. Add a second function **circumference** to compute the circumference of a circle with a given **radius**. This parameter can be an integer, and the method should return a **double**.
10. Implement the method to return 2 times pi times the radius, this time using a static field **Math.PI**.
11. Call the method twice – once for each of **one** and **two** – from the bottom of **main**, and format the results. You can continue a formatting string with a combination of literal text and formatting fields, as in:

```
System.out.format ("Circumferences of circles are %1.4f, %1.4f%n",  
    circumference (one), circumference (two));
```

12. Build and test again, and you should see output like this:

```
run 7 3
```

```
Sum of numbers = 10
```

```
Difference of numbers = 4
```

```
Product of numbers = 21
```

```
Dividing floating-point conversions = 2.3333
```

```
Hypotenuse with sides given = 7.6158
```

```
Circumferences of circles are 43.9823, 18.8496
```

This is the intermediate answer in **Step2**.

More Math**LAB 3A****Optional Steps**

13. If you like, experiment more with formatting the output: could you produce a listing that lined up all the numbers in a clean column? Feel free to jump into it with what you've seen so far, or look to the next few steps for hints and instructions. The desired output for **one**=7 and **two**=3 is shown below, with nine characters allocated to the integral part and four to the decimal part of the numbers:

Sum of numbers	=	10
Difference of numbers	=	4
Product of numbers	=	21
Dividing numbers	=	2.3333
Hypotenuse with sides given	=	7.6158
Circumference of circle one	=	43.9823
Circumference of circle two	=	18.8496

14. Start by changing all output statements to **format** calls. So each **format** call will take three arguments: a format string, a literal string (such as "Sum of numbers"), and a numeric value.
15. For your integer values, use a formatting string "`%-27s = %9d%n`". This will left-justify the label but then pad it to 27 characters; produce a space, equals sign and space; format the integer to nine characters, right-justified, and produce an end-of-line character or sequence.
16. For floating-point values, use the same string but change "`%9d`" to "`%14.4f`". Why not just "`%9.4f`"? The first number is the total character width of the field, including the decimal point and the four additional digits to the right. This will allow those characters to run past the end of the integer-value lines, which is what we want.

This is the final answer code in **Step3**.

Asset Management

LAB 3B

In this lab you will implement a simple application that combines some numerical calculation with string-building to produce a report on a single asset as described on the command line.

Lab project: Finance/Step1

Answer project(s): Finance/Step2

Files: * to be created
src/Asset.java

Instructions:

1. Open the source file **Asset.java** to find that the class **Asset** is already stubbed out, including a **main** method.
2. Gather three command-line arguments: the **name** of the asset as a string, the **numberOfShares** as an **int**, and the **sharePrice** as a **double**. (You can use **Double.parseDouble** the same way you used **Integer.parseInt** in the previous lab.)
3. Format four values to the console: **numberOfShares** (%d), **name**(%s), **sharePrice**(%f), and total value as **sharePrice * numberOfShares**(%f). The floating-point fields should be set up to produce two digits to the right of the decimal. Another nice formatting trick you might try at this point is to insert a comma in the field for a number, right after the % sign: this will cause a locale-appropriate separator to appear every three digits in large numbers. Finally, remember a %n to get an end-of-line production!

Asset Management**LAB 3B**

4. Now produce the results of a quick test – see if the **name** is “MMEnterprises”.

```
System.out.println  
    ("Name is MMEnterprises? " + (name == "MMEnterprises"));
```

5. Build and test. You should see results like this:

```
run Coke 5 6.6  
5 shares of Coke at $6.60 -- asset value = $33.00  
Name is MMEnterprises? false
```

6. Now try “MMEnterprises” as the name – what do you think will happen?

```
run MMEnterprises 5000 6.6  
5000 shares of MMEnterprises at $6.60 -- asset value = $33,000.00  
Name is MMEnterprises? false
```

7. What’s wrong with the program so far?

8. Add a second test that uses the **equals** method:

```
System.out.println  
    ("Name is MMEnterprises? " + (name.equals ("MMEnterprises")));
```

9. Build that and test it out:

```
run MMEnterprises 5000 6.6  
5000 shares of MMEnterprises at $6.60 -- asset value = $33,000.00  
Name is MMEnterprises? false  
Name is MMEnterprises? true
```


A Mileage Table

LAB 3C

In this lab you will implement a jagged array to capture the data for a table of driving distances between a few cities. This is a natural application for non-rectangular arrays because the data set itself is really triangular – we have a grid of city-to-city but we don't need the distance from X to X, nor do we need to see the distance from Y to X once we've captured the distance from X to Y.

In this lab you will write the code to initialize the jagged array with mileage values, and do a quick test of finding a mileage in the array and printing it out. In the next chapter you will add more complex processing that uses loops to respond to user requests.

Lab project: Mileage/Step1

Answer project(s): Mileage/Step2

Files: * to be created
src/Mileage.java

Instructions:

1. Open **Mileage.java** and see that the class and **main** method are already implemented, and that there are two simple **String** arrays initialized with the names of four cities. **origins** has values in alphabetical order, while **destinations** runs in reverse-alphabetical order. This is so that we can capture data in a triangle packed up against index (0, 0). The cities and mileages look like this, with origins down the left and destinations across the top:

	Washington	Philadelphia	NYC	Boston
Boston	439	306	216	
NYC	226	93		
Philadelphia	139			
Washington				

A Mileage Table**LAB 3C**

2. Declare an array of arrays of integers, called **mileage**, and set it to hold three arrays:

```
int[][] mileage = new int[3][];
```

3. Initialize the values of the array, one by one:

```
mileage[0][0] = 439; mileage[0][1] = 306; mileage[0][2] = 216;  
mileage[1][0] = 226; mileage[1][1] = 93;  
mileage[2][0] = 139;
```

4. Try building and running at this point. If everything's okay, the application will produce no output.

run

```
Exception in thread "main" java.lang.NullPointerException  
    at Mileage.main(Mileage.java:25)
```

What's the problem? The exception occurs in the first of your mileage assignments. Whenever you see this in working with individual array elements, check to be sure that the elements have been allocated in memory – this doesn't happen automatically, as discussed already. Remember, too, that a two-dimensional array is really an array of arrays. You've got a line of code that allocates this array of arrays – that's one dimension. What about the individual arrays?

5. Add code to initialize each of the three arrays. To allocate the triangular shape we want here, make the first array three elements, the next one two, and the third one just one element.

```
int[][] mileage = new int[3][];  
mileage[0] = new int[3];  
mileage[1] = new int[2];  
mileage[2] = new int[1];  
mileage[0][0] = 439; mileage[0][1] = 306; mileage[0][2] = 216;  
mileage[1][0] = 226; mileage[1][1] = 93;  
mileage[2][0] = 139;
```

6. Build and test again, and you should get a silent, but error-free, run.
7. Now, at the bottom of the method, initialize two integers: **testOrigin** to zero, and **testDestination** to one.
8. Print the distance at this index. Build and test – the desired output is:

Distance from Boston to Philadelphia is 306 miles.



CHAPTER 4

FLOW CONTROL



OBJECTIVES

After completing “Flow Control,” you will be able to:

- Invoke code in a separate method, perhaps repeatedly.
- Build Java code to behave differently based on one or more conditions.
- Iterate over a block of code using a variety of looping constructs.
- Perform some action on every element in an array.
- Loop over all values of an enumerated type.
- Process variable argument lists in method code.
- Nest conditionals and loops as needed for a given algorithm.
- Use **break** and **continue** to refine the flow of execution through a single loop or nested loops.
 - Label nested loops to allow **break** and **continue** to work at any desired scope.
- Implement recursive methods.

Getting Started

- Java applications all begin their execution in the **main** method, as found and invoked by the launcher.
- At its simplest, the flow of execution will run from the top to the bottom of **main**, each declaration and expression being performed in a sequence.
 - When the **main** method ends, the launcher tears down the JVM and kills the supporting process.
- Most applications require a much more complex flow of execution than that.
- There are various techniques for flow control in Java:
 - **Calling** a method
 - **Returning** from a method
 - **Conditional** execution of code blocks
 - **Iterative** execution of code blocks – loops
 - Throwing and catching **exceptions**
- In this chapter we'll study all of these techniques in depth – with the exception of exceptions, which we'll defer to a later chapter.
- Flow-control techniques finally open up the full power of algorithmic programming in Java.
 - We'll pursue a broad range of hands-on exercises in this chapter, to reinforce all of the lessons up to this point.

Method Declarations

- We've considered the method declaration thus far from the basic perspective of language syntax.
- A method declaration is a small definition of a contract between the caller and the implementer.
- The implementer **declares**:
 - That a method is available by a certain **signature** – name and parameter list, although parameter names are only useful to the implementer, and are not binding on the caller in any way
 - That it will return a value of a certain type when invoked, or that it returns no value at all (a **void** method)
 - That it may throw certain **exceptions** (see Chapter 11)

```
public double hypotenuse (double x, double y)
```

- The caller invokes the method simply by naming it in an expression, and by passing **arguments** to the method in a parenthesized, comma-separated list.

```
double z = hypotenuse (7.0, 12.0);
```

- Each argument, in order, must be a legal value of the type of the corresponding, declared parameter in the method signature.
- The compiler will take several different approaches to **coerce** the argument to the appropriate type if it is not exact. These are the widening conversions discussed in the previous chapter, and other tricks yet to be discussed.

Calling Methods

- Calling a method is one simple form of flow control.
- Execution will flow from the point of the call to the first line of code in the called method.
- The compiler builds code that will pass the caller's arguments to the method code, visible using the parameter names.
- To return the execution point to the point of the call, use **return**.
 - If the method is declared with **void** as the return type, nothing else is necessary (or permitted).
 - If not, you must return an expression of the appropriate type, the evaluation of which will be returned to the caller as the evaluation of the method call.
- You can return from any or many points in a method.
- The compiler will insist that all possible control flow paths within a method find their way to a valid return.
- Methods of return type **void** needn't have any **returns** in them; it will be assumed that the method returns when completed.

Structured Programming

- We'll ultimately come to appreciate Java as an **object-oriented**, or **OO**, language.
- OO theory focuses on decomposition and reusability, but it was hardly the first theory to do so.
- **Structured programming** first developed the ideas of:
 - Limited **scope** – that not all variables in a program should be global
 - Explicit **declaration** of variables to be used in a given scope
 - Descriptive variable **names**
 - **"Top-down"** design and code organization: application code arranged in a **hierarchy** of source units (now classes), methods, and data variables
- This last idea is a lot like decomposition in OO.
 - We'll soon see that OO organizes software by **encapsulations** of data and method code – these are **classes**.
 - To oversimplify: structured programming gets halfway there by focusing on structure and reusability of program code, and in the process organizing data around **functions** or **methods**.

Structured Programming in Java

- Classes are the key to object-oriented Java programming, and we'll get to that soon.
- Methods are the key to structured Java programming.
- Application design should recognize various, distinct **functions** – things the application needs to do.
- Code then will consist of many **methods** – the name “method” does imply a means of solving a problem.
- We've seen a small example of structured programming already, in the Math application.
 - Though it is mostly an unstructured script, Math needs to calculate circumference twice.
 - Why do this repeatedly in the **main** method?
 - Better to break it out to a separate method that can be invoked as needed.
 - Parameters are critical to method reusability, and hence to making structured programming feasible.
- So control flow in Java commonly passes from method to method – sometimes in series, but sometimes with method A calling B which calls C which calls D, and so on.
 - This is often called a method **call stack**, and the stack grows with each call and shrinks with each return.

Flow Control

EXAMPLE

- The next section of the chapter explains the most commonly used flow-control features in Java: **conditionals** and **loops**.
- Simple examples are given in these pages.
- A comprehensive example of flow-control techniques is implemented in **FlowControl**.
- You may want to open the source file **src/FlowControl.java** now and have it available for reference as different constructs are discussed over the next few pages.
- At the end of the section, or whenever you prefer, try building and running the application and studying the results of various code fragments.

Testing Conditions

- To invoke different behaviors in different situations, the basic construct in Java is **if-else**:

```
if (condition)  
    <true-block>  
[else  
    <false-block>]
```

- The condition must be a boolean expression.
- The code blocks may be single expressions or blocks of code enclosed by curly braces.
- The **else** part is optional.

- You can nest these:

```
if (x == 4)  
{  
    System.out.println ("Special case.");  
    if (y == 4)  
        doSomethingFantastic ();  
    else  
        doSomethingBoring ();  
}  
else return;
```

- You can chain them as well:

```
if (x == 4)  
    System.out.println ("Hello");  
else if (x == 5)  
    System.out.println ("Yo");  
else  
    System.out.println ("Greetings");
```

Handling Multiple Cases

- If you get too many of these going the coding may be annoyingly repetitive, and for comparing one variable against many values there is a better construct: **switch-case**.

```
switch (expression)
{
case label-1:
    <label-1-code>
case label-N:
    <label-N-code>
[default:
    <default-code>]
}
```

- The evaluated expression is tested against the listed case values in sequence; when a hit is found, the following code is executed.
- If no hit is found, and there is a default label, that code is executed.
- You must use **break** if you wish to break control out of the **switch-case** system.
 - There is no automatic break at the end of a case; if you leave it out, control will flow through all remaining code.
 - By the same token, several cases can trigger the same code.
- You do not need to enclose multiple, semicolon-separated expressions in curly braces under a case, although you may do so.

switch-case Examples

EXAMPLE

- Consider the following code – this one is actually drawn from **FlowControl** – and predict the value of **result** for each possible value of **howMuch**:

```
String result = "";
switch (howMuch)
{
case 0:
    result = "I'm shy. ";
    break;

case 3:
    result = "Let me tell you something. ";

case 2:
    result += "I can talk a streak. ";

case 1:
    result += "I'm not so shy. ";
}

switch (howMuch)
{
case 2:
case 3:
    result += "Sorry, am I blabbering?";
}
```

switch-case Examples

EXAMPLE

- In the first switch:
 - Execution will break out of the 0 case only.
 - The 1 case will only add one string and fall through the bottom.
 - The other cases will add more than one string.
 - Note the use of multiple cases over the same code in the second switch.
 - This second system is logically identical to:
 - This second system is logically identical to:
- ```
if (howMuch == 2 || howMuch == 3)
 result += "Sorry, am I blabbering?";
```
- The final values of **result** would be:

|   |                                                                                         |
|---|-----------------------------------------------------------------------------------------|
| 0 | I'm shy.                                                                                |
| 1 | I'm not so shy.                                                                         |
| 2 | I can talk a streak. I'm not so shy. Sorry, am I blabbering?                            |
| 3 | Let me tell you something. I can talk a streak. I'm not so shy. Sorry, am I blabbering? |

## switch-case Examples

### EXAMPLE

- Immediately below this example is another, in which the test expression for the **switch** is a string.

```
String[] input = { "ALL", "1", "2", "UNKNOWN" };
for (String test : input)
{
 System.out.format ("%8s", test);
 switch (test)
 {
 case "ALL":
 System.out.println ("All of the above");
 break;

 case "UNKNOWN":
 System.out.println ("Value is unknown");
 break;

 default:
 System.out.println ("100 * value is " +
 100 * Integer.parseInt (test));
 }
}
```

- Output from this passage of code is:

```
ALL All of the above
1 100 * value is 100
2 100 * value is 200
UNKNOWN Value is unknown
```

## Short-Circuit Evaluation

---

- When the compiler encounters a boolean expression, and that expression includes evaluation of logical operators, the compiler will stop evaluating as soon as it is certain of the result.
- This is much more than a performance optimization!
- It means that if the first expression in a series, when evaluated, makes the remaining expressions irrelevant, they will not be executed at all.

- For `(x && y)`, Y will be ignored if X is **false**.

- For `(x || y)`, Y will be ignored if X is **true**.

- A common way to take advantage of this feature is to check array boundaries prior to indexing the array:

```
// Safe use of args:
if (args.length != 0 && args[0].equals ("Command"))
 doCommand ();
```

- Beware, however: any mutations that might occur to variable or object state in latter expressions may or may not occur:

```
// x may or may not be incremented:
if (myCondition && ++x > myThreshold)
 doTheDo ();
```



## Loops

---

- There are several looping constructs in Java:
  - The **while** loop
  - The **do-while** loop
  - The **for** loop, which is available in a standard and also a new simplified syntax
- In all of these constructs, the loop can iterate over a single expression or over a block of code.
- Note that any declarations in a block of code defined under a loop have the scope of that block.
  - This is true for all nested code blocks, but it is tempting when dealing with loops to allocate something within and reference it when the loop has completed. This is not possible.
  - Instead, declare the element prior to beginning the loop; assign or modify the element in the loop; and test or otherwise use the element after the loop completes:

```
boolean itWorked = true;
<any looping construct here>
{
 // derive a reference to thisTask
 if (thisTask.failed ())
 itWorked = false;
}

if (itWorked)
 celebrate ();
```

## The while Loop

---

- The **while** loop is the simplest, logically: it iterates so long as its condition is met.

```
while (condition)
 <loop-block>
```

- The condition will be checked at the outset and after each iteration until it fails, at which point control will flow to the expression following the loop.
- A common technique is to use a unary decrement or increment operator as part of the condition:

```
public void freeAll (Object[] someArray)
{
 int n = someArray.length;
 while (n-- != 0)
 {
 System.out.println
 ("Releasing object: " + someArray[n]);
 someArray[n] = null;
 }
}
```

- It is also common to establish a potentially infinite loop using **while (true)**.
  - Why would one do this?
  - Hold that thought! and we'll come back to it when we see how to **break** out of loops.

## The do-while Loop

---

- The **do-while** loop is a variation on the **while** loop.

```
do
{
 System.out.println ("Hello?");
 waitTenSeconds ();
}
while (!responseReceived ());
```

- The only logical distinction is that it always executes once.
  - The condition is only tested after the first iteration.
  - The above code will be sure to prompt the user at least once for a response.
  - If none comes in a certain period of time, the loop will cause the prompt to be repeated.

## The for Loop

---

- The **for** looping construct is the most complex and powerful one, allowing you to define three important expressions:

```
for (initializer; boundary; iteration)
 <loop-block>
```

- The **initializer** is executed first, and only once. It can include declaration and initialization of a new variable.
- The **boundary** condition is checked at the outset and prior to each iteration.
- The **iteration** expression is evaluated (that is, executed) once after each iteration.

- An example of using **for** to look for factors of a number *n*:

```
for (int i = 1; i <= n; ++i)
 if (n % i == 0)
 System.out.println ("Factor: " + i);
```

- None of the terms is strictly required.

- The initializer can be absent, for instance if initialization has already occurred:

```
for (; stillGoing (); iterate ()) makeMerry ();
```

- The iterator can be absent if something in the loop block changes the boundary condition.
- Leaving out the boundary is rare, but legal, like **while (true)**.

## Processing Arrays

---

- A simplified version of the **for** loop is available for the common task of iterating over all values of an array.

```
for (type iterator : array)
 <loop-block>
```

- Example – given an array of **doubles**:

```
for (double number : someArray)
 doSomethingTo (number);
```

- This is exactly equivalent to:

```
for (int i = 0; i < someArray.length; ++i)
 doSomethingTo (someArray[i]);
```

- The syntax is obviously simpler and nicer, but some control is lost.

- Use the full, formal syntax to:

- Loop from the end to the beginning of an array:

```
for (int i = someArray.length - 1; i >= 0; --i)
 doSomethingTo (someArray[i]);
```

- Loop over same-sized arrays in parallel, where the index is needed to work on both at once, for instance to copy from one to the other:

```
for (int i = 0; i < array1.length; ++i)
 array1[i] = array2[i] * 2;
```

- Loop over only part of the array; process every other member; etc.

## Looping and Enumerated Types

---

- The **enum** type defines a finite set of named values.
- What if we wanted to loop over all those values?
- Any enumerated type provides a static method, **values**, which returns an array containing all its possible values, in the order of their declaration.
- This then is another good application for the simplified **for** loop:

```
System.out.print ("Colors are ");
for (Color color : Color.values ())
 System.out.print (color.name () + " ");
```

- It may seem odd at first that the type **Color** appears twice in the **for** loop.
- This is an oddity of enumerated types, which will come clearer when we study them in depth: the type itself provides some utilities, such as the **values** method.
- Another common usage for getting the enumerated value that corresponds to a given string is:

```
Color color = Color.valueOf ("red");
```

## Processing Variable Argument Lists

---

- Recall, also from our language-fundamentals discussion, the **varargs** feature, which allows a method to declare that it takes any number of arguments of a certain type.

```
public static String concat
 (String first, String... allOthers);
```

- How are these arguments passed to the method code?
  - We can't define a unique name for each, as in **element0**, **element1**, **element2**, ...
  - This would be inelegant, and at any rate we don't know at code-and-compile time how many arguments there will be.
- Rather, the values are passed to the method in an array of the appropriate type.
- Thus we have another good application for the simplified **for** loop.
- **concat** might look like this, since **allOthers** will be an array of **Strings**:

```
public static String concat
 (String first, String... allOthers)
{
 String result = first;
 for (String next : allOthers)
 result += next;

 return result;
}
```

## Collecting Statistics

### EXAMPLE

- In **Stats** there is a small application that accumulates statistics on scores.
- In order to remember these numbers from one call to the next, we need fields to store them.
  - In this way we're dipping our toes into the object-oriented waters, just a little ahead of schedule.
  - See `src/Stats.java`:

```
private int sampleSize = 0;
private double mean = 0.0;
```

- The **addScores** method modifies these with each new score received, by a tricky little expression that modifies both values:

```
mean = ((mean * sampleSize) + score) /
 (++sampleSize);
```



## Collecting Statistics

### EXAMPLE

- The method offers a sort of batch-processing mechanism by which the caller can pass any number of scores at once.
  - It then processes the whole list as an array of integers:

```
public void addScores (int... scores)
{
 System.out.print ("Adding scores:");
 for (int score : scores)
 {
 System.out.print (" " + score);
 mean = (mean * sampleSize) + score) /
 (++sampleSize);
 }
 System.out.println ();
}
```

- The **main** method exercises this and a **report** method a few different ways:

```
public static void main (String[] args)
{
 Stats stats = new Stats ();
 stats.addScores (100, 62, 80, 93, 82);
 stats.report ();

 stats.addScores (99);
 stats.report ();

 stats.addScores ();
 stats.report ();
}
```

## Collecting Statistics

### EXAMPLE

- Notice that it's legal to call the method with no arguments!
  - This is harmless but also pointless; if it were important to prevent this sort of call the method signature would have to be

```
public void addScores
 (int score, int... moreScores)
```

- ... and the method would have one parameter for the first score and a second one with an array of any others given.
- Build and test the application and observe the following output:

```
Adding scores: 100 62 80 93 82
Sample size: 5
Mean score: 83.4
```

```
Adding scores: 99
Sample size: 6
Mean score: 86.0
```

```
Adding scores:
Sample size: 6
Mean score: 86.0
```

## The Ternary Flow-Control Operator

---

- Expression grammar also has a means of flow-control built into it: this is the ternary operator `?:`.
- This operator has much the same effect as an **if-else** construct.
  - The following two passages are logically equivalent:

```
if (x == 0)
 System.out.println ("zero");
else
 System.out.println ("non-zero");

System.out.println (x == 0 ? "zero" : "non-zero");
```

- **if-else** systems can direct processing over entire blocks of code, including other conditionals, loops, and declarations.
- The `?:` operator can be nested in the midst of a complex expression.
  - It can also simplify code where both potential paths for an **if-else** do very nearly the same thing, as in the above example.
- Both result expressions (the second and third operands) must evaluate to the same type.
  - This limitation does sometimes force one to use **if-else**, all by itself.

## Test Scores

**LAB 4A**

### Suggested time: 30 minutes

In this lab you will implement an application that processes test scores, as already primed into two arrays: **scores** and **names**. You will assign grades to each student based on his or her score, thus creating and populating a third array **grades**. You will then print the complete results to the console.

Detailed instructions are found at the end of the chapter.

## DNA

**LAB 4B****Optional****Suggested time: 30 minutes**

In this lab you will refine an existing algorithm for storing sequences of DNA codons in a byte array. The starter code simply copies each character of a string into a byte array of the appropriate length. You will compress the storage of this data by recognizing that there are only four distinct characters in the string – ‘G’, ‘C’, ‘A’, and ‘T’ – and converting each character to a two-bit number that is then packed into one-fourth of a byte in the array.

This will give some additional exercise in looping and working with arrays, and also allow you to explore the less-commonly-used operators `<<` and `>>` for bitwise rotation of integer values.

This lab is recommended for experienced algorithmic coders. Those with limited experience in bitwise arithmetic, especially, are advised to study the instructions and completed lab answer, or to expect to take extra time working through the lab steps.

Detailed instructions are found at the end of the chapter.

## Overriding Loop Flow

---

- One can refine flow control through a loop using **break** and **continue** statements.
  - Both immediately terminate the current iteration.
  - **break** ends the loop; that is, it sets the execution point to immediately after the loop.
  - **continue** proceeds immediately to the next iteration.
- An example of using break to optimize a search – why keep looping when the item has been found?

```
for (int i = 0; i < myArray.length; ++i)
 if (myArray[i] == targetNumber)
 {
 System.out.println ("Index = " + i);
 break;
 }
```

- An example of using continue to avoid the latter part of a block of code in some of the iterations:


```
for (int n : doubles)
{
 System.out.println ("Value: " + n);
 if (n < 0)
 continue;

 System.out.println
 (" Square root: " + Math.sqrt (n));
}
```

## Issues with break and continue



---

- These methods of interrupting the flow of a loop block have their uses, but they do run afoul of a key tenet of structured programming.
  - This theory asserts that there should be one and only **one way in** to a block of code, one and only **one way out**.
  - **break** and **continue** violate this – by design in fact.
  - In complex code bases, this makes it easier for a programmer – especially one who didn't write the original code – to err when modifying the code.




```
for (int n : doubles)
{
 System.out.println (n);
 if (n < 0)
 continue;


 System.out.println (Math.sqrt (n));
 System.out.println (n * n); //oops!
}
```



- Thus some discourage the use of these techniques, preferring a more rigorous structured approach:



```
for (int n : doubles)
{
 System.out.println (n);
 if (n >= 0)
 System.out.println (Math.sqrt (n));
}
```



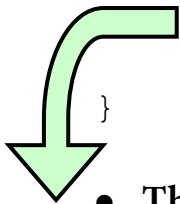
- ... though it's not always that simple!

## Controlling Nested Loops

---

- Loops and conditionals can be nested to any depth.
  - Two nested loops are often called the **outer** and **inner** loop.
- **break** and **continue** both apply to the current scope – that is, to the narrowest scope.
- What if the algorithm design calls for breaking out of an outer loop under some conditions?
- Java provides a means for **labeling** a loop with a single identifier.
- The label can then be referenced by either **break** or **continue** to indicate the scope at which either should operate:

```
outer: for (int x = 0; x < matrix.length; ++x)
{
 inner: for (int y = 0; y < matrix[x].length; ++y)
 if (myArray[i] == targetNumber)
 {
 System.out.println ("Found it!");
 break outer;
 }
}
```



- This is a more compelling case for using **break**.
  - There is still the problem of multiple exit paths.
  - Practically, though, the advantage is greater, because it's harder to manage control flow through nested loops with proper structured-programming practices.



## Statistics

**LAB 4C**

**Suggested time: 30 minutes**

In this lab you will enhance the **Scores** application from Lab 4A to produce some statistics about the distribution of test scores. This will provide exercise in working with arrays and nested loops.

Detailed instructions are found at the end of the chapter.

## Looking up Driving Distances

**LAB 4D****Optional**

**Suggested time: 45 minutes**

In this lab you will complete the **Mileage** application you began in Lab 3C by adding code to look up the distance between two user-supplied cities. This will take you through some interesting gymnastics as you: gather user input; find the cities in the origin and destination arrays; rule out error conditions in the input choices; deal with the triangular shape of the array, since the array cell you really want might be the mirror image of the one the user asked for; and finally produce the requested output.

This is an intermediate lab; instructions are less specific than in most earlier labs, and you are challenged to come up with some of the tactical-level solutions on your own. It is meant to help reinforce some of the basic techniques you've seen over the last few chapters.

Detailed instructions are found at the end of the chapter.

## Recursion

---

- It is entirely legal in Java for a method to call itself.
- This is an important algorithmic technique known as **recursion**.
  - We say that flow of execution **recurses** through several calls on the method in question.
  - Recursion through multiple methods is also possible, as in A calls B, B calls C, C calls A.
- Whenever recursion is implemented, it is essential that some **boundary condition** be defined and enforced that will **terminate the recursion after some number of calls**.
  - This may be a number known at compile time, or it may be expressed in terms of method parameters, whose values are only known at runtime.
  - The unacceptable scenario is called **infinite recursion**, which will hang the calling thread and eventually crash the JVM.
  - Each method call demands a certain amount of memory called **stack space**, and infinite recursion will exhaust available memory by consuming it per recursive call.
  - A simple example of infinite recursion would be:

```
public static int myMethod (int x)
{
 return myMethod (x - 1) * x;
}
```

## “Long” Division

### EXAMPLE

- In **LongDivision** is an example of recursive processing: the method **divide** produces the result of dividing one integer by another.

- It does so by recursively subtracting!

```
public static int divide
(int dividend, int divisor)
{
 System.out.println (" dividend = " + dividend +
 "; divisor = " + divisor);

 if (dividend >= divisor)
 return divide (dividend - divisor, divisor) + 1;
 else
 return 0;
}
```

- The **dividend** is reduced by the amount of the **divisor**.
  - The result is simply a count of the number of times the method is called.
  - The boundary condition in this case is when the remaining amount in **dividend** is less than the **divisor**.
- Build and test on the arguments 7 and 3.

**run 7 3**

```
dividend = 7; divisor = 3
dividend = 4; divisor = 3
dividend = 1; divisor = 3
7 / 3 = 2
```

## Sorting Algorithms

**LAB 4E****Optional**

**Suggested time: 60 minutes**

In this lab you will implement one or more common sorting algorithms to a list of the names of the fifty US states. This is the final challenge lab for this section of the course, and step-by-step instructions are not provided, although starter code is. Each of three sorting algorithms – bubble, insertion, and merge – is described briefly, and you are asked to implement it from scratch using whatever techniques are appropriate. The final one in particular – the merge sort – is an exercise in managing a recursive method. You will also have the option of implementing a binary search.

Even if you do not do this lab, you may find a review of the answer code interesting, as several common Java techniques are illustrated throughout. Also, there is a rough performance profile of the sorting algorithms themselves, based on clocking the answer application for various sizes of data sets. This data really illustrates the power of recursion – at least as a stack-space-for-speed tradeoff.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- The Java language provides a range of constructs for conditional execution of method code.
- Likewise there are several options for building loops to iterate over certain code.
- Labeling of loops gives greater control over the flow of execution through nested loops.
- With flow-control techniques understood, the lessons of the last two chapters can finally be brought to bear on a wide range of practical problems.
  - Expressions and data types are important concepts by themselves.
  - Only with conditionals, loops, and method calls can a programmer attack most practical problems, however.
- Now that we have a good handle on procedural programming in Java, we're going to leap into the world of object-oriented software.

# Test Scores

**LAB 4A**

In this lab you will implement an application that processes test scores, as already primed into two arrays: **scores** and **names**. You will assign grades to each student based on his or her score, thus creating and populating a third array **grades**. You will then print the complete results to the console.

**Lab project:** Scores/Step1

**Answer project(s):** Scores/Step2 (intermediate)  
Scores/Step3 (final)

**Files:** \* to be created  
src/Scores.java

## Instructions:

1. Open **Scores.java** and see that the **main** method begins by initializing the two arrays **scores** and **names**, each with ten elements. You will do your work at the bottom of **main**.

2. First, print out a list of student names. You can use the simplified **for** loop, as in:

```
for (String name : names)
 System.out.println (name);
```

3. Build and test this new code:

```
Suzie Q
Peggy Fosnacht
Boy George
Flea
Captain Hook
Nelson Mandela
The Mighty Thor
Oedipa Maas
Uncle Sam
The Tick
```

4. Now you'll evaluate each numerical score as a letter grade. For this the simple **for** syntax won't cut it, because you'll need to work in parallel through three arrays. Start by declaring the third of these, an array of strings called **grades**. Initialize it to a set of ten new **String** references. A good practice here is to rely on the length of one of the existing arrays in dimensioning this new one; this way, if the set of scores grows or shrinks in the future, this line of code will not need to be changed. The same can be said of the rest of the code you're about to write – we'll avoid "hard-coding" the size 10 anywhere.

```
String[] grades = new String[scores.length];
```

**Test Scores****LAB 4A**

5. Print a header to the console such as “Student Score Grade”. You can pad whatever amount of space seems to be appropriate in between the words here – take a quick look at the list of names to get a better idea what will work.
6. Create a **for** loop over the **scores** array, using the full syntax and declaring an **int student** as the loop index. Create a blank code block for the loop, and do the remaining steps inside that block.
7. Get this student’s score as a local **int** called **score** – this will be **scores[student]**.
8. Declare a local string **grade** and set it to the empty string – that is, **“”**.
9. Write code to set the correct value for **grade** based on **score**. You can do this with a series of **if-else if-else**. The **grade** should be set to “A” for scores 90 or over; “B” for 80-89, and so on, with an “F” given for a score less than 60.
10. Set **grades[student] = grade**.
11. Print the student name, score and grade to the console.
12. Test now to see if your logic is working. Correct results will look something like this:

... (student names)

| Student         |    | Score | Grade |
|-----------------|----|-------|-------|
| -----           |    | ----  | ----- |
| Suzie Q         | 76 | C     |       |
| Peggy Fosnacht  | 91 | A     |       |
| Boy George      | 80 | B     |       |
| Flea            | 55 | F     |       |
| Captain Hook    | 71 | C     |       |
| Nelson Mandela  | 98 | A     |       |
| The Mighty Thor | 70 | C     |       |
| Oedipa Maas     | 88 | B     |       |
| Uncle Sam       | 69 | D     |       |
| The Tick        | 60 | D     |       |



**Test Scores****LAB 4A**

13. Now, as shown above, your output probably isn't very clean-looking. The student names vary in length, of course, so you wind up showing the scores and grades in staggered columns. You could fix this using formatting, but let's take this as an opportunity for more exercise in loops – how would you go about it? You need to know how to do a few things: get the length of a String, loop, and print single spaces. Well, the length of a string is **someString.length ()**; you already know how to loop; and you can **System.out.print (' ');** to get a space without a line break following.

14. So, to clean up the output, start by breaking out the name to an initial statement:

```
System.out.print (names[student]);
System.out.println (" " + score + " " + grade);
```

15. Between the two statements, insert a **for** loop that will print the right number of spaces. That is, the loop should run an index from **names[student].length ()** to however many spaces there are in total before the first column of scores should be printed. Print a single space each time through the loop.

16. Build and test, and you should see output like this:

| Student         | Score | Grade |
|-----------------|-------|-------|
| -----           | ----- | ----- |
| Suzie Q         | 76    | C     |
| Peggy Fosnacht  | 91    | A     |
| Boy George      | 80    | B     |
| Flea            | 55    | F     |
| Captain Hook    | 71    | C     |
| Nelson Mandela  | 98    | A     |
| The Mighty Thor | 70    | C     |
| Oedipa Maas     | 88    | B     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |

This is the intermediate answer in **Step2**.

**Optional Steps**

17. Here's another little challenge: could you simplify the code you're using to calculate each grade? After all, the first four grades are a smooth sequence of letters; the scores on which they're based are a regular sequence of multiples of ten; and you can convert characters to integers!

18. Start by removing most of your score tests. You should have only a single test, now: is the score greater than or equal to 60? If it is, set **grade** using an expression that will automatically resolve any score between 60 and 99 to a capital letter between A and D. (Hint: the numerical ASCII code for capital letters starts with 65 for an 'A'.) If it's less than 60, simply set **grade** to "F" – you probably already have this code.

19. Build and test your updated code – the output should be the same.

This is the final answer in **Step3**.

# DNA

**LAB 4B****Optional**

In this lab you will refine an existing algorithm for storing sequences of DNA codons in a byte array. The starter code simply copies each character of a string into a byte array of the appropriate length. You will compress the storage of this data by recognizing that there are only four distinct characters in the string – ‘G’, ‘C’, ‘A’, and ‘T’ – and converting each character to a two-bit number that is then packed into one-fourth of a byte in the array.

This will give some additional exercise in looping and working with arrays, and also allow you to explore the less-commonly-used operators `<<` and `>>` for bitwise rotation of integer values.

This lab is recommended for experienced algorithmic coders. Those with limited experience in bitwise arithmetic, especially, are advised to study the instructions and completed lab answer, or to expect to take extra time working through the lab steps.

**Lab project:** **DNA/Step1**

**Answer project(s):** **DNA/Step2**

**Files:** \* to be created  
**src/DNA.java**

**Instructions:**

1. Build and test the application. Notice that it accepts a string from the command line, but if none is provided it has a default value ready for testing purposes. The application prints this string, packs it one-character-per-byte into an array, and then reconstitutes a new string from the byte array, and prints the result. (The idea of the byte array is that this might become the format for persistent storage of the string, say in a database or flat file.)

```
GCCATTGGCACCTAA
GCCATTGGCACCTAA
```

**DNA****LAB 4B**

2. Open **DNA.java** and begin by defining a new **char[]** called **map**, and priming it with the four characters 'A', 'C', 'G', and 'T'. Do this just after the declaration of the **compressed** byte array.
3. Change the dimension of **compressed** from the full length of the string to the length, plus three, then divided by four. Each byte of this array will accommodate four characters of the string, and by adding three before dividing we assure that there will be a full byte available for one, two or three characters at the end if the length is not a multiple of four.
4. The new storage structure will provide one byte for each four characters in the string. This is possible because a byte is eight bits, and because there are only four possible characters, which therefore can be identified by just two bits (a number from zero to three). The trick will be to pack these two-bit numbers in bits 0-1, 2-3, 4-5, and 6-7 of the total byte, and then to get them back out.
5. Remove the one statement that forms the body of the first **for** loop and replace it with a code block. Begin this block by declaring two integers: **index** will be the index of the byte in the **compressed** array that holds the value for this character, and **rotation** will be the number of bits to the left by which the character's code value will be rotated left, in order to fit it into its slot in the indexed byte. Get **index** as **a / 4** and **rotation** as **(a % 4) \* 2**.
6. Declare a third integer, **find**, and set it to zero. You will use this to express the code value of the current character.
7. Create a **while** loop that checks two conditions: it should proceed only when **find** is less than **map.length** and when **map[find]** does not equal the current character (which is **codons.charAt(a)**). Each time through the loop, execute **++find**.
8. When the loop terminates, **find** will be between zero and four inclusive. If it's four, the current character wasn't found, so you should print an error message and quit – use **System.exit(-1)** to shut down the application at once.
9. Otherwise, **find** is the code value you want. Now, to pack this into the array, bitwise OR it into the correct byte of the array, after rotating it left by **rotation**:

```
compressed[index] |= find << rotation;
```

10. You might want to **compile** at this point, just to be sure you have no syntax errors. You could test, too, although the results would be strange, because you've changed your encoding algorithm but not the decoding algorithm, so you get a four-character result:

```
GCCATTGGCACCTAA
-?Q♥
```

**DNA****LAB 4B**

11. Rip out the body of the decoding loop – keep the declaration of **reconstituted**, though, and the definition of the loop itself. (Notice that this is the one place in the example where we can use the more convenient for-each syntax. Everywhere else we need to use that zero-based index somehow, but here we just need to process each byte in sequence.)
12. Above the loop, declare an **int** called **totalLength**, and set it to zero.
13. Nest a second loop inside the first one that iterates a rotation value **r** starting from zero and incrementing by 2 each time. Also increment **totalLength** by one each time – recall the comma separator when you want to perform multiple updates in the iterator section of a **for** loop. The boundary condition is that **r** is less than eight and that the **totalLength** printed is less than **codons.length ()**.
14. Now, how to rediscover that character? You want to bitwise AND the value of byte **b** with the number 3 rotated left by **r** – that will mask out all but the two bits you want – and then rotate the result right by **r** to get it down to the code value in the range zero to three. Then use that as an index to the **map** array, and that's your character! Append that to **reconstituted**. That's a long way of saying:

```
reconstituted += map[(b & (3 << r)) >> r];
```

15. Now print **reconstituted** as you did before. Build and test, and the default string should be decoded correctly from the smaller array:

```
GCCATTGGCACCTAA
GCCATTGGCACCTAA
```

## Statistics

**LAB 4C**

In this lab you will enhance the **Scores** application from Lab 4A to produce some statistics about the distribution of test scores. This will provide exercise in working with arrays and nested loops.

**Lab project:** **Scores/Step3**

**Answer project(s):** **Scores/Step4**

**Files:** \* to be created  
**src/Scores.java**

### Instructions:

1. If you didn't complete Lab 04A, review the instructions there, and build and test the application in your lab directory, which is the completion of that lab.
2. Open **Scores.java** and start working at the bottom of the **main** method. First, declare an integer called **total** and set it to zero.
3. Create a **for** loop labeled **Student**. Iterate an index **student** over the **scores** array.
4. Inside the code block for this array, add that student's score to **total**.
5. After the loop, print the mean test score to the console: this will be **total** divided by the length of **scores**.
6. Build and test. You should find that the mean score is 75.8.
7. What's that? You got 75? Why do you suppose that is? If you got 75 and not 75.8, you need to convert **total** to a **double** before dividing it – this will preserve double precision in the arithmetic and will give you the precise result. Build and test again.
8. Now, before the **Student** loop, declare and populate an array of strings called **possibleGrades** with the five possible string values "A", "B", "C", "D" and "F".
9. Also declare and initialize an array of five integers called **frequency** to a set of five zeroes. You will increment values in this array based on the contents of **grades**.
10. Inside the loop, before adding to **total**, define a second loop labeled **Grade**. This should iterate an index **g** over the **possibleGrades** array.
11. For each index value, test to see if this student's grade equals the possible grade in question. (A few things to remember: you have two indices **student** and **g**, and one should be used on each of the arrays; and be sure not to test two strings against each other with the **==** operator – unless you're looking for their identity as objects.) If the values do match, bump the value of **frequency[g]**, and for performance break out of the inner loop.

**Statistics****LAB 4C**

12. In a second loop at the bottom of **main** (no label needed), iterate over **possibleGrades** again, and print out the number of times each grade occurred.

13. Build and test, and you should see results like these:

```
...
Statistics:
 There were 2 As given.
 There were 2 Bs given.
 There were 3 Cs given.
 There were 2 Ds given.
 There were 1 Fs given.
```

The mean score was 75.8

This is the final answer in **Step4**.

**Optional Steps**

14. Try changing the **break** statement you just wrote to **break Student;** – what do you think will happen? Build and test ...

```
...
Statistics:
 There were 0 As given.
 There were 0 Bs given.
 There were 1 Cs given.
 There were 0 Ds given.
 There were 0 Fs given.
```

The mean score was 0

15. You killed off all the processing after finding the first score.

16. Now try **continue Student;** – what do you think will happen now?

```
...
Statistics:
 There were 2 As given.
 There were 2 Bs given.
 There were 3 Cs given.
 There were 2 Ds given.
 There were 1 Fs given.
```

The mean score was 0

**Statistics****LAB 4C**

17. Now you allow complete processing of occurrences, but you never accumulate anything in **total**, so the mean value is zero.
18. Finally, try leaving the label out of the statement, so that it's just **break**; – now what?
19. You should see the correct output as with the final answer step, because **break** and **continue** both default to the “most local” scope, meaning the inner loop.

```
...
Statistics:
 There were 2 As given.
 There were 2 Bs given.
 There were 3 Cs given.
 There were 2 Ds given.
 There were 1 Fs given.

The mean score was 75.8
```

## Looking up Driving Distances

**LAB 4D****Optional**

In this lab you will complete the **Mileage** application you began in Lab 3C by adding code to look up the distance between two user-supplied cities. This will take you through some interesting gymnastics as you: gather user input; find the cities in the origin and destination arrays; rule out error conditions in the input choices; deal with the triangular shape of the array, since the array cell you really want might be the mirror image of the one the user asked for; and finally produce the requested output.

This is an intermediate lab; instructions are less specific than in most earlier labs, and you are challenged to come up with some of the tactical-level solutions on your own. It is meant to help reinforce some of the basic techniques you've seen over the last few chapters.

**Lab project:** **Mileage/Step2**

**Answer project(s):** **Mileage/Step3**

**Files:** \* to be created  
**src/Mileage.java**

**Instructions:**

1. If you did not complete Lab 3C, build and run the starter application, which is the completion of that lab. The mileage table is initialized and a quick test is performed to look up one of the values:

Distance from Boston to Philadelphia is 306 miles.

2. Open **Mileage.java** and start work at the bottom of the **main** method by deleting the declarations of **testOrigin** and **testDestination**. Leave the **System.out.println** call, but open up some space before it; you will do your work prior to this call and it will be the last statement of the completed application.
3. Implement a check that the user has provided at least two command-line arguments. If not, print a usage statement, and then quit.
4. Declare two strings **origin** and **destination**, and initialize each to one of the command-line arguments.
5. If the two strings are the same, print an error and quit – no point producing the distance from X to X.
6. Declare two integers **originIndex** and **destinationIndex**, both initially zero.



## Looking up Driving Distances

## LAB 4D

7. Search through the **origins** array and find the right value for **originIndex**. This is the same pattern as used in Lab 4B to find the index of the current character in **map**: that is, use a **while** loop to continue to increment **originIndex** as long as it's less than the length of the array and the string at that point in the array is not the requested **origin**.
8. Do the same for **destinationIndex**.
9. If either index equals the length of the corresponding array, the requested city was not found; print an error and quit.
10. Now the remaining trick is that while the array is triangular in shape to avoid stating distances redundantly, the user doesn't know this, and if she did she still wouldn't know in which order to place the two city names. For instance if the origin index were 2 (Philadelphia) and the destination were 2 (NYC), your lookup would go out of bounds – but the value you really want is at origin 1, destination 1 – NYC to Philadelphia. How can you check for this condition, and correct it?
11. Your code can check in one of two ways. The lookup will go out of bounds if **mileage[originIndex].length <= destinationIndex** – by definition. Another way to check is to recognize that the shape of the array can be expressed as the constraint that the sum of the two indices must be less than **origins.length – 1**. Perform either of these tests, and if the index would be out of bounds, open a code block and perform the following two steps.
12. To rearrange things so that the distance will be found, first swap **origin** and **destination** strings to be the command line arguments 1 and 0, respectively, where they were 0 and 1. This will make sure that the output is clean and help you to check when this block of code is executed and when it isn't.
13. Swapping the index values is tougher. Because of the nature of the data, the two arrays **origins** and **destinations** have the same size but run in opposite directions. Thus you will want to do the usual three-step swap of values – temporary value equals origin, origin equals destination, destination equals temporary value – but set **originIndex** to the length of **origins** minus **destinationIndex** minus one! and similarly for **destinationIndex** based on your temporary value. This isn't as crazy as it will first seem, but it requires a little extra thought.
14. Now, modify the final **System.out.println** statement to use **origin**, **destination**, **originIndex** and **destinationIndex**.

**Looking up Driving Distances****LAB 4D**

15. Build and test, and you should be able to look up any of the values in the array based on any of the twelve combinations of city names “Boston”, “NYC”, “Philadelphia”, and “Washington”. If you misspell a city, you should get a clear error; if you provide the same city twice, you should get an error; and you should get the same result regardless of the order of two city names. As in:

**run**

Usage: java Mileage <origin> <destination>

**run Boston Philadelphia**

Distance from Boston to Philadelphia is 306 miles.

**run Boston NYC**

Distance from Boston to NYC is 216 miles.

**run Bosstown Warshington**

Couldn't find one or both cities.

**run Boston Boston**

Distance is definitely zero!

**run Philadelphia Boston**

Distance from Boston to Philadelphia is 306 miles.

# Sorting Algorithms

**LAB 4E****Optional**

In this lab you will implement one or more common sorting algorithms to a list of the names of the fifty US states. This is the final challenge lab for this section of the course, and step-by-step instructions are not provided, although starter code is. Each of three sorting algorithms – bubble, insertion, and merge – is described briefly, and you are asked to implement it from scratch using whatever techniques are appropriate. The final one in particular – the merge sort – is an exercise in managing a recursive method. You will also have the option of implementing a binary search.

Even if you do not do this lab, you may find a review of the answer code interesting, as several common Java techniques are illustrated throughout. Also, there is a rough performance profile of the sorting algorithms themselves, based on clocking the answer application for various sizes of data sets. This data really illustrates the power of recursion – at least as a stack-space-for-speed tradeoff.

**Lab project:** **Sort/Step1**

**Answer project(s):** **Sort/Step2** (intermediate)  
**Sort/Step3** (final)

**Files:** \* to be created  
**src/Sort.java**

**Instructions:**

1. Open **Sort.java** and see that several methods are implemented – not all of them completely. **main** calls **buildNamesArray**, which returns an array of the names of 50 states running roughly from west to east. It then sorts using **bubbleSort** and prints the results. It does the same with **insertionSort** and then **mergeSort**. Finally, with the last sorted array, it calls **indexOf** to try to find a name provided on the command line. **indexOf** should implement a binary search over the passed array. So far, all three sort methods and the search method do nothing.
2. Implement one or more of the sorting algorithms as described on the following page, and/or the binary-search method, also described. Test your code and assure that the names are printed in alphabetical order, and that you can find a given state name (if you implemented **indexOf**.)

## Sorting Algorithms

## LAB 4E

### Bubble Sort

A bubble sort is the simplest sorting algorithm to implement, but is also the slowest.

Make a pass through the array from 0 to length minus two, inclusive. For each element, check if the element after it is less than the element itself (for strings this means `A.compareTo (B) < 0`). If so, swap the two elements in the array – typically using a temporary variable to hold value one, moving two to one, and then filling the temporary value in for value two.

Keep making passes until you don't perform any swaps in a pass. Then you're done.

### Insertion Sort

The insertion sort starts from the second element in the array and works to the last one. For each element, it treats the entire array preceding that element as already successfully sorted. This is true for the second element, since there is only one element before it! For each element, then, start by setting it into a temporary variable, and treat its slot in the array as an open space. Then work back through the preceding elements, shuffling them forward to fill the space so long as they are greater than the temporary value. When you find an element whose value is less than the temporary value, the temporary value should be inserted in the space after that element, instead of shuffling that element forward.

### Merge Sort

This one is hard! The idea is to break the array up into roughly equal halves or subsections, and to recursively sort each subsection and then merge the two. This sounds like magic ... and it works like magic, compared to the previous two, if you get it right. The trick is that eventually these subsections are broken down to only one or two elements in length. One element is always in the right order! Two elements can simply be tested and swapped.

To merge two arrays that are already sorted, set an index to the start of each, and an index into a new array that's the size of the two sources put together. Loop while both indices are still in range for their source arrays, and each time through, copy whichever indexed element is less, and increment only that index. When the loop falls through, one and only one of the two arrays will be exhausted; copy the remaining elements from the other one.

A hint to get started on this one is that **mergeSort** won't have most of the code for this answer. Rather you will define a method **mergeHelper** that takes not only the target array but also two integers **left** and **right** that define the subsection you want to sort. **mergeSort** will **mergeHelper** right off the bat, providing 0 and length-minus-one for the two new arguments, and then **mergeHelper** will call itself – twice – unless the size of the subsection is one or two.

## Sorting Algorithms

## LAB 4E

### Binary Search

A binary search is really the payoff for sorting a collection in the first place. It is only possible on a sorted array. Start in the middle of the array and check if the element is what you want. If not, see if it's less than or greater than your desired value. This will rule out the other half of the array! You can then jump to halfway between the far end and the thing you just tested, and try again. This is a very fast search because it can safely skip over so much content, and it scales very well.

1. Whether or not you go through the work of implementing any of these methods, review, build and test the answer code. Discuss the implementations with your instructor. Here's the expected output:

#### **run Nevada**

Bubble sort:

Bubble sort took 39 passes and performed 1785 assignments.  
Alabama, Alaska, Arizona, Arkansas, California, Colorado, Connecticut,  
Delaware, Florida, Georgia, Hawaii, Idaho, Illinois, Indiana, Iowa,  
Kansas, Kentucky, Louisiana, Maine, Maryland, Massachusetts, Michigan,  
Minnesota, Mississippi, Missouri, Montana, Nebraska, Nevada, New  
Hampshire, New Jersey, New Mexico, New York, North Carolina, North  
Dakota, Ohio, Oklahoma, Oregon, Pennsylvania, Rhode Island, South  
Carolina, South Dakota, Tennessee, Texas, Utah, Vermont, Virginia,  
Washington, West Virginia, Wisconsin, Wyoming

Insertion sort:

Insertion sort performed 640 assignments.  
Alabama, Alaska, Arizona, Arkansas, ...

Merge sort:

Alabama, Alaska, Arizona, Arkansas, ...

Nevada is a state.

2. Finally, try the **Step3** answer. This does not do the binary search, but it does the three sorts on a much bigger set of strings – 2500 well-mixed values based on the state names – and it times each of the three methods. See the file **SortResults.txt** in **Step3** for sample results over this and a much larger set of 125,000 strings.

Results over a well-mixed set of 125,000 items (3 nested loops):

Bubble sort:

Done in 7338723 milliseconds.

Insertion sort:

Done in 1372864 milliseconds.

Merge sort:

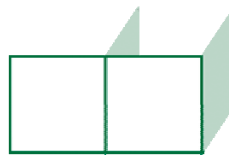
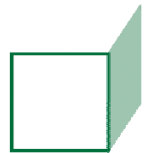
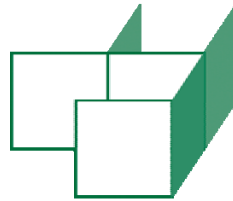
Done in 611 milliseconds.





## CHAPTER 5

# OBJECT-ORIENTED SOFTWARE



## OBJECTIVES

*After completing “Object-Oriented Software,” you will be able to:*

- Describe the benefits of using object-oriented analysis and design techniques to solve complex software problems.
- Employ decomposition and abstraction to break a complex system into discrete concepts and collaborating roles.
- Express those concepts and roles as classes.
- Identify relationships between classes in the resulting system.
- Decide what pieces of an object model should be made visible to which other pieces, or to the user of that system or the outside world in general.
- Undertake an iterative approach to the software design and development process, using OO concepts and the Unified Modeling Language.



## Getting Our Bearings

---

- Thus far we've studied Java largely as a procedural programming language.
- Now we undertake a study of Java as an **object-oriented** language, beginning to get the full value from class and object relationships and inheritance.
- For those who have not been exposed to a great deal of object-oriented analysis and design, or **OOAD**, this chapter serves as an introduction.
  - We will look at the motivations for OO analysis and design, without specific reference to the Java language.
  - We will consider one important part of OOAD, known as **static analysis**, which is concerned with identifying the **things** involved in solving a problem (things can be tangible, intangible, or in fact mere intellectual constructs).
  - Another major sort of analysis which we will leave alone, because it doesn't help us much in understanding Java, is **dynamic analysis**, concerned with first understanding the functioning of the required system.
- Having acquired basic OO concepts in this chapter, we'll proceed over the next few chapters to see how these concepts map to the Java language.
  - We'll also continue to expand on OOAD in later chapters, synchronizing this with study of the specific ways in which Java implements additional OO features, and building up a case-study application as we go.

## Complex Systems

---

- The primary point of OOAD is to aid in building complex systems, and one typically begins the OO process with a **decomposition** of a complex problem.
  - **Analysis** is the craft of breaking a problem into logical parts.
  - It is by no means a trivial task; in many cases there are tricky judgment calls to make in decomposing a real-world software problem, and many factors come into play.
- The goal is to progressively separate a problem into smaller problems by identifying potential subsystems and the relationships between them.
- Eventually, the parts come into focus as individual, more or less self-sufficient **concepts**.
  - **Abstraction** refers to the process of identifying the important concepts in a potential solution.
  - As the term implies, one must move from a real-world description of something to an abstract concept of that thing.
  - This may sound like going in the wrong direction!
  - But software is soft, after all: it is important to define what parts of a real (tangible or intangible) entity are relevant to the problem, and which are mere distractions.
  - Not every aspect of a real entity need be modeled in the corresponding software.

## Case Study – A Car Dealership

---

- We'll now begin to design an application that models a car dealership.
  - This will provide a practical example to motivate the OO concepts discussed in this chapter.
  - We'll then build the application over the next few chapters, as we see these concepts expressed in the Java language.
- The car dealership has a stock of cars, some new and some used, and also a parts department.
  - This is the basic data relevant to the application; in OO terminology this is the application's **state**.
  - The idea of a car dealership also implies **behavior**: actions such as listing inventory, searching for cars, buying and selling cars, taking cars for test drives, and so on.
  - The core concept of **objects** as an organizing principle for software design is that we should be able to find ways to marry elements of state and behavior that are logically related.
  - For example the data sticker price and price paid by the dealership clearly play into the behavior of selling the car.
  - How can we connect these things logically in design and implementation, so as to facilitate development and maintenance of the application?
- We'll now study specific OO features and apply them to building up a design for the car dealership.

## Analysis

---

- We can identify several distinct concepts in the dealership:
  - The **dealership** as a whole
  - An **inventory** of cars and parts
  - Individual **cars**
  - Individual **parts**
  - The **customer**
  - One or more **sellers** who can sell cars
- How do these concepts relate?
  - There are **cardinal** relationships, for instance **one dealership to many cars** and many parts.
  - Probably there are **many sellers to many cars**, which is to say that any seller might wind up selling any car.
  - There are **dependencies** such as a customer relying on the behaviors of a seller in order to buy a car.
  - There are **specializations**: new and used cars might share a number of features but then be different in other ways.
  - Also, cars and parts can both be seen as inventory, so there might be a **generalization** of the two for purposes of producing inventory reports.

## Advantages of Decomposition

---

- After analysis, one big problem has been broken into several smaller ones.
  - Each concept addresses a small problem.
  - They collaborate to solve the larger problem.
  - It should be easier to solve several little problems than one big and complex one.
  - This is important in initial development, but the real rewards are seen when it comes time to maintain the software, as individual concepts help to isolate various features of the application that might need to change.
  - Thus a huge advantage of OOAD is that it produces software that **adapts well to change**.
- Also, in the system under analysis, and possibly in future development, there is a better chance that work once done will not need to be done again: a better chance for meaningful **reuse** of software.
  - A clearly recognized concept, well decoupled from others in the original problem space, may become the basis for a new system, or be plugged in to an emerging one.
  - This is much less likely for larger, **monolithic** designs.
  - Even if a defined concept doesn't exactly fit the new problem space, it might be close enough to be specialized for that new problem.

## What is a Car?

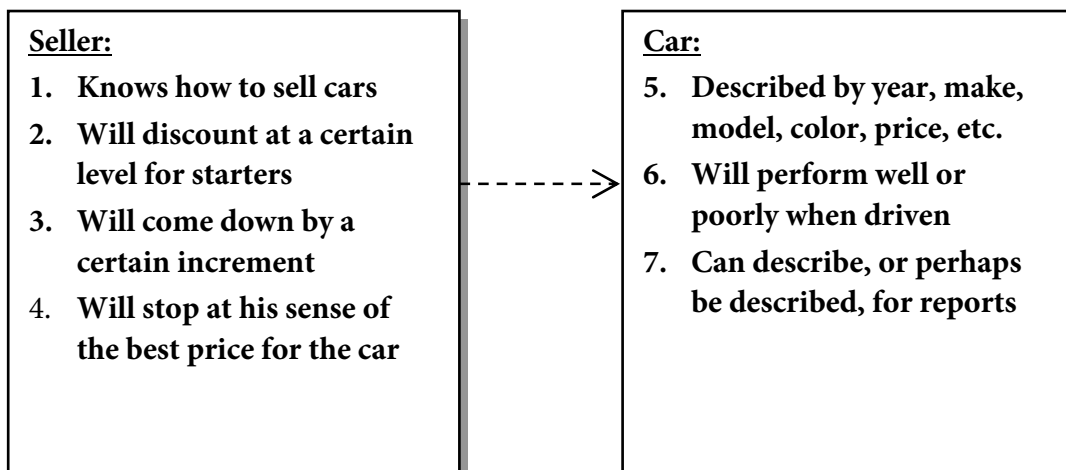
---

- Consider several possible abstractions of the concept of a car.
- In a manufacturing management and control application, a car might be conceptualized as something to be built and shipped.
  - A car has many component parts which must be assembled.
  - It may have an associated status as it proceeds along an assembly line.
- In an application designed to predict traffic flow for hypothetical detours on a street map, a car is something that moves toward a destination.
  - It has speed and direction.
  - If we infer the presence of a driver, the car can be seen as making decisions based on stimuli.
  - Here we see the notion that an abstraction can identify more than a set of nouns associated with a concept; the abstraction can include the notion of associated behaviors.
- In our application, a car is something to stock and hopefully to sell.
  - A car has state including make, model, color, and price.
  - Behaviors include description for reporting purposes, taking a test drive, entering into a negotiation to sell the car.

## Classes

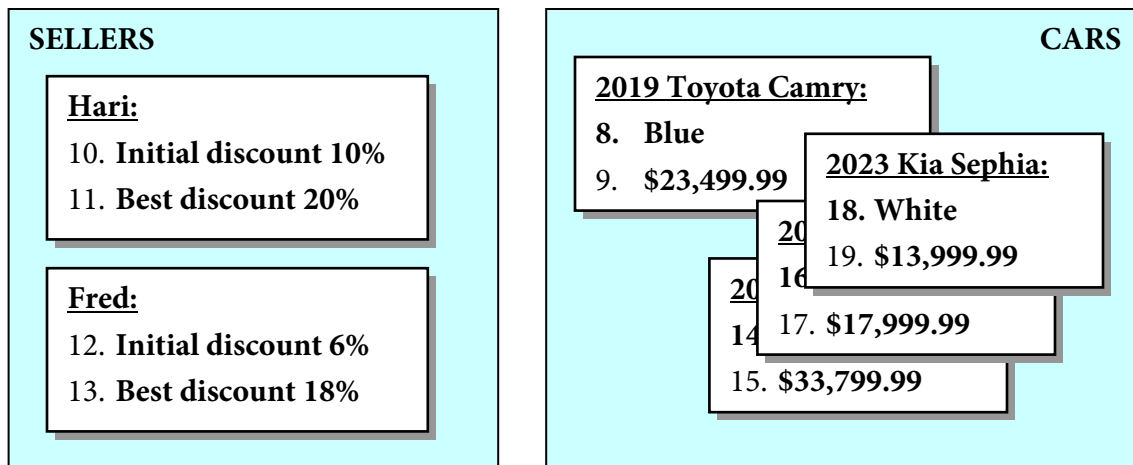
---

- A **class** is a formal expression of a concept.
  - It is more than just a data record; a class gathers related definitions of **state** and **behavior**, and makes them easily accessible to each other.
  - Therefore we say that the class **encapsulates** the concept.
  - Other terms for **class** are **classification** and **type**.
- Multiple classes compose a system.
  - A class is considered to have **responsibilities**, to play a particular role in solving a problem.
  - Many classes, in order to meet their responsibilities, will be defined to delegate to other classes, called **collaborators**.
  - Collaboration occurs through **relationships** such as dependency, association, aggregation, composition, specialization and generalization.



## Objects

- Any individual thing that can exist in the real-world system might be modeled, at runtime, by an **object** in memory.
  - There is one concept of a car, but we expect there will be many individual cars on the lot.
  - In the application, there will be one class for the car concept, but there will be many **objects** of that class, or type.
  - So a class **defines** state and behavior, and an object holds the actual state of an individual thing and can apply behavior to that specific state, according to the definitions of the class.
  - A class **classifies** the possible objects in the system.





## Advantages of Encapsulation

---

- A class, by encapsulating state and behavior, can be designed to be responsible for its own instances' integrity.
  - The class will provide a **public** interface to its capabilities or functionality.
  - It will keep its state **private**, however. This is known as **data hiding**, and it assures that the class code is the only code that can make state changes on objects of that class.
  - This helps to keep order in a complex system, in which many people will be working on sometimes overlapping tasks.
  - It encourages those not responsible for a given class' code to work with objects of that class as intended.
  - Also, as changes to the class' **implementation** are required, they can often be made without changes to the **interface**, thus saving users of the class from the need to recode or rebuild.
- Each class in a properly decomposed system meets a well-defined set of responsibilities, has well-defined capabilities.
  - Any other classes that require those capabilities can get them in one place, and one place only.
  - This is known as a **single point of maintenance**, which as the term suggests is useful because initial development as well as ensuing debugging, maintenance and changes due to new requirements are effected in one place.

## What is a Date?

---

- Consider another example: in just about any business system, there will be a need to create, manipulate and represent calendar dates.
- What are the responsibilities of a hypothetical **Date** class?
  - Hold state information for a specific calendar date that can be reliably expressed as day-month-year.
  - For one thing the year had better support more than 2 digits!
  - Beyond this, though, a **Date** class would be asked to validate parameters whenever a user of the class attempts to set new state information – to ensure its own integrity as a meaningful datum.
  - It would be asked to manage calendar arithmetic, to answer questions like, “What is the date that’s two weeks after February 22, 1999?”

## A Date Class

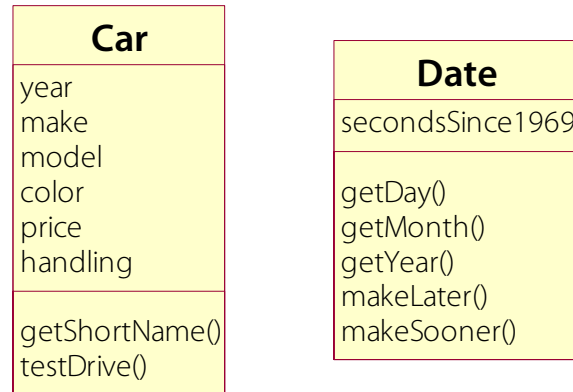
---

- What **methods** would a **Date** class need to offer?
  - Methods to set its value or individual parts of its value, known formally as **mutators** and colloquially as “setters.”
  - There are specific sorts of methods or messages that can be defined on a class just for initialization, called **constructors**, and this would be an excellent place for one.
  - Then we might not want or need individual methods to set year, month, day.
  - We would surely want methods to get the whole state, or part of it; these are known as **accessors** or “getters.”
  - More sophisticated functions like date arithmetic, formatting the date into strings like “Wednesday, April 21, 1999”.
- What **attributes** would a **Date** class define?
  - That’s up to us! This is an **implementation detail** that may or may not be relevant to analysis and design, and more importantly, it is no one’s business but the class designer’s and eventually the implementer’s.
  - One possible approach is to keep separate numbers for year, month, day.
  - Another is to keep a single count of days since a certain uniform starting date.
  - Note that one is easier to express and to parse (to decode and encode), but the other is easier to manipulate.

## Unified Modeling Language

---

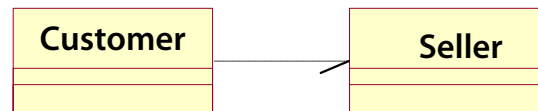
- To continue our study of OOAD, and as a useful notation through the rest of the course, we will take a brief look at the **Unified Modeling Language**, or **UML**.
- UML represents classes as shown below.



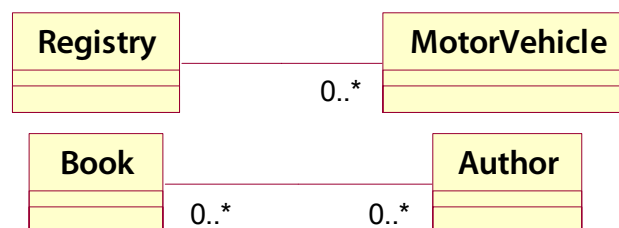
- The main rectangle comprises three **compartments**, for the class' **name**, **attributes**, and **operations**.
- That is, the name, the state, and the behavior.
- UML can express **visibility** of attributes and operations; the formal terms are as follows:
  - **Public** members are just that; anyone can use them.
  - **Private** members can only be seen from inside the class.
  - **Protected** members can only be seen from inside the class and from any specializations of the class.
- Though the concepts are important, for simplicity we'll mostly leave visibility notes out of our diagrams.
  - The general rule will be that attributes are private and operations are public.

## UML Relationships

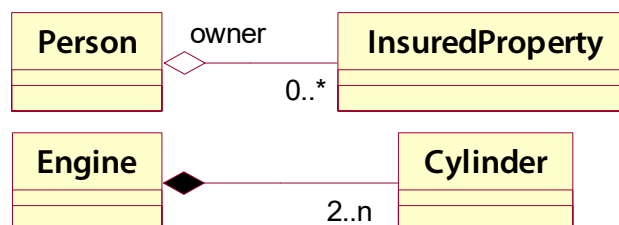
- In order to fully describe a solution to a software problem, we must be able to show classes and the relationships between them.
- UML recognizes a few basic relationships, each of which suggests something about the interaction between classes as implemented and at runtime.
  - If objects of one class use objects of another, the classes are related by semantic **dependency**:



- A class that relates to another by a clear cardinal relationship – one-to-one, one-to-many, etc. – relates to that other class by **association**. Cardinality and **role** names can be attached to the simple solid line that connects associated classes.



- A more specific sort of association is the **aggregation**, which implies that one concept is really defined in terms of another. Some aggregations are also logical **compositions**.



## UML Relationships

---

- One good instinct to develop: when looking at a tentative class diagram, identify and count the required relationships.
- There is no set mathematical relationship that should hold between number of classes and number of relationships.
- If you have too few and you are likely not solving the problem – or it was not a very complex problem.
  - See if there aren't relationships that will indeed be required as the software is implemented.
- Trickier is the problem of too many relationships.
  - Each relationship implies a dependency of some sort.
  - Dependencies translate to reliance on specifics of another class' definition, especially that class' public operations.
  - The stronger the relationship, the deeper the dependency, although language specifics will dictate the costs.
  - For instance if a class aggregates another and that second class' interface changes, your class will need to be recoded and rebuilt.
  - Any complex system requires some dependencies, but too many will make the software **brittle** and difficult to adapt.
  - See if there are responsibilities that could be more cleanly encapsulated in one place, not several.

## Iterative Development

---

- OOAD is much more than a set of techniques and notation; it is a true **methodology** for software development.
- At the core of OO thinking is the idea that complex software must be built **iteratively** to succeed in the modern business climate.
- Nearly all software today must adapt to rapid change driven by a fast-paced, competitive economy.
- Yet technological advances and economies of scale have driven the demand for increasingly complex systems:
  - Handling vast quantities of data
  - Transacting business at great volume and frequency
  - Performing difficult computations
  - Enforcing complicated business rules
- Thus even the requirements for software are prone to change while the software is being built, and certainly after it's completed in a version 1.0 as well.

## Embracing Change

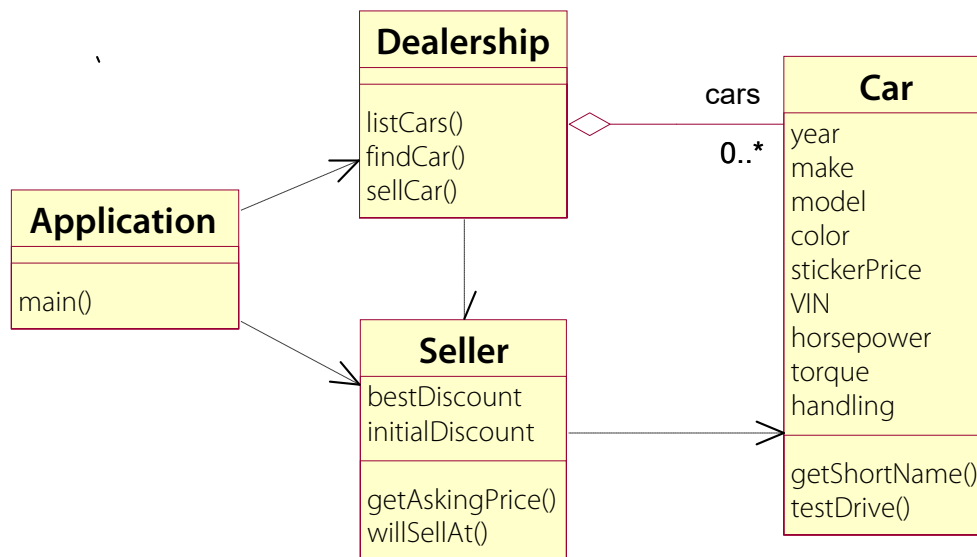
---

- The OO philosophy embraces rapid change.
- It rejects the older “waterfall” methodology.
  - This traditional approach called for total control of a long development process from end-to-end in one comprehensive cycle from functional requirements through design and implementation to testing and release.
  - It just isn’t practical for most business problems today.
- Instead it is possible, and even imperative, that change be incorporated into the development process.
- OO software is built in **iterations**, all but the last of which is incomplete by design!
  - Each iteration, from the earliest prototype, goes through requirements definition, analysis, design, implementation and testing.
  - This makes for much more robust software as the development proceeds; there are no great surprises waiting to be ferreted out after many months of waterfall development.
- Done well, iterative development reduces risk.
- The trick is to define iterations sensibly:
  - What problems should be attacked immediately?
  - What can safely be left open for now?



## Car Dealership Design – First Pass

- We'll take an iterative approach to our case study over this course, though not for the usual practical reasons.
  - Rather, we are undertaking a learning process, and want to move incrementally through concepts.
  - Still, the general iterative approach – and UML's strength in sketching out early solutions that mature into the complete design – will serve us well.
- Here's a UML diagram of the first iteration of the system, which we'll begin to implement in the next chapter:



- We're focusing on an inventory of cars, the ability to test-drive them, and sellers who can engage the customer to sell them.
- We leave out parts and general inventory for the moment; this is a simple example of allowing incompleteness in UML diagrams and development cycles.

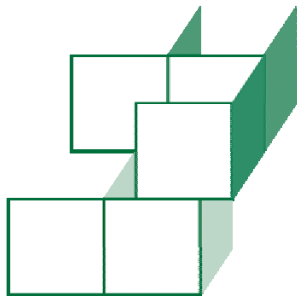
## SUMMARY

- **Modeling software as classes and objects offers benefits:**
  - In addressing complex problems methodically
  - In robustness and maintainability of the resulting code
- **A class is defined to have a clear set of responsibilities which in turn define that class' role in solving the problem at hand.**
- **To solve a complex problem, classes will require collaboration from other classes; these too should be clearly defined, since a collaborator assumed on one class translates into a responsibility on another.**
- **With a few simple relationships it is possible to model systems of almost any level of complexity.**
- **Iterative development is key to successful development in a climate of rapid requirements change.**
  - The OO methodology embraces change and takes advantage of an iterative approach to improve the reliability of software as it develops in tandem with evolving ideas about the requirements of the application.
- **We'll consider addition features of object-oriented systems in later chapters, synchronizing these with study of the Java language and our case-study application.**



# CHAPTER 6

## CLASSES AND OBJECTS



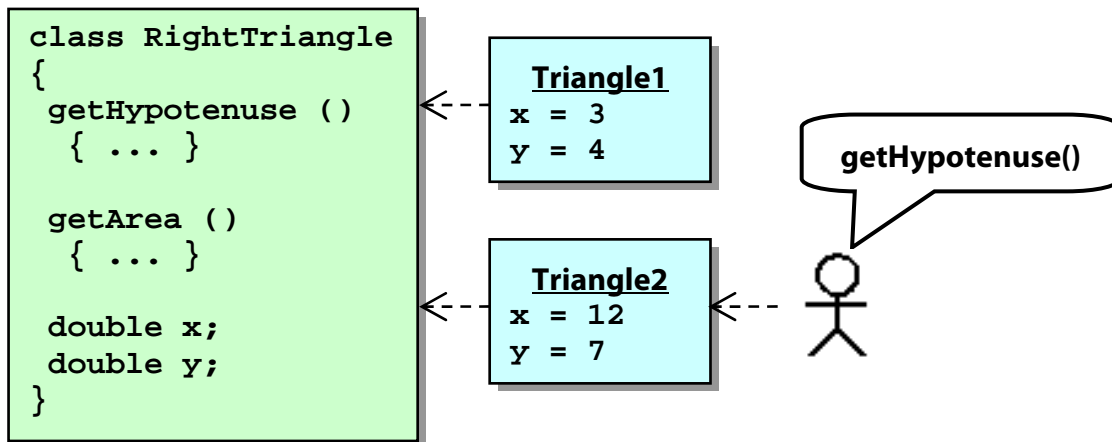
## OBJECTIVES

*After completing “Classes and Objects,” you will be able to:*

- Implement a class design in Java using classes, fields, and methods.
- Define constructors to properly initialize class instances.
- Write methods to use the implicit **this** reference as well as to make references outside the class or object scope.
- Define field and method visibility according to class design and Java package design.
- Overload methods to provide semantic convenience to the client code.
- Implement class relationships using fields and arrays.
- Pack Java classes into a JAR and run applications from a JAR on the class path.

## The Java Class as an Encapsulation

- Java models classes using the **class** construct.
- Classes can be **instantiated** one or many times in the run of an application; the instances are **objects**.
  - Objects occupy their own memory, primarily to capture state.



- **Fields** express UML attributes – elements of state.
  - Each field occupies part of the object's memory allocation, and is thus evaluated distinctly for each object of a class.
- **Methods** express UML operations – definitions of behavior.
  - When a method is called, it is called on a particular object, using the dot separator: **stock.getPrice**, **thisCar.testDrive**.
  - The object on which the method was called is then implicit in the method's behavior; the method reads and writes fields on that object, and not some other object.

## The Java Class as an Encapsulation

---

- We've seen that variables of primitive type – numbers, booleans, characters – can be assigned directly to values of that type.

- Objects must be explicitly instantiated using **new**:

```
RightTriangle triangle1 = new RightTriangle ();
triangle1.setX (3);
triangle1.setY (4);
```

- Then they can be used through references such as **triangle1**.
- Call methods or reference fields on an object using the dot separator, as shown above.
- Strings are a special case.
  - They are objects of class **String**, but the compiler treats strings specially such that one rarely has to create a string using **new**.
- **triangle1** is not an object!
  - It is an object reference, or perhaps “a name for an object.”
  - An object reference defaults to **null** – no object is referenced.
- It is possible to pass arguments in a **new** construct:

```
RightTriangle triangle1 = new RightTriangle (3, 4);
```

  - These arguments can be used to initialize that class.
  - The compiler will pass them along to an appropriate **constructor**.

## Constructors

---

- Java compilers recognize a specialized method definition known as a **constructor**.
- Constructors are defined much as methods are, with some basic distinctions.
  - Constructors always bear the name of the class itself.
  - Constructors do not return a value.

```
public class Point
{
 public Point (int x, int y) { ... }
}
```

- Constructors are called implicitly on object creation, after memory has been allocated for the object's fields, to allow the object to initialize that memory.
  - If no constructor is explicitly defined for a class, the compiler will supply a public **default constructor** – one that takes no arguments.
  - If any constructors are defined, this default constructor will not be built by the compiler. It is a good idea to define a default constructor if you think you will use one.
  - When the object is created with **new**, any arguments supplied at that time are passed to the constructor.

```
Point myPoint = new Point (4, 5);
```

## Garbage Collection

---

- Once it's created, there is no explicit way to destroy an object in Java!
- Instead, the JVM tracks all references to the object once it is created.
  - When a second or third reference is assigned to the object, its **reference count** increases accordingly.
  - When those references all go out of their various scopes, and the object's reference count comes to zero, the object's memory is made available for **garbage collection**.
- There is a **garbage collector** thread running at low priority in the JVM that frees such memory when it has the chance.
  - The process can also be invoked explicitly on an application thread, by calling **System.gc**.
- Neither does Java recognize a **destructor** as an opposite to constructors.
- Java does define a special method called a **finalizer** which is at least roughly analogous to a destructor.

`protected void finalize ()`

- Implementing **finalize** is unusual.
- Typical cleanup code would explicitly delete objects referenced by the dying object – which is unnecessary in Java! That's the point of garbage collection.



## An Ellipsoid Class

**EXAMPLE**

- In **Ellipsoid/Step1** there is a class **Ellipsoid** which encapsulates a three-dimensional ellipsoid.
- See **src/cc/math/Ellipsoid.java**:
  - It holds three independent semi-axis values **a**, **b**, and **c**.

```
private double a;
private double b;
private double c;
```

- It can compute its own volume and classify itself according to equivalence relationships between the semi-axes.

```
public double getVolume () { ... }
public String getType () { ... }
public String getDefinition () { ... }
```

- A separate application class **EllipsoidCLI** provides a command-line interface for testing **Ellipsoid**.

```
Ellipsoid bean = new Ellipsoid ();
bean.setA (Double.parseDouble (args[0]));
bean.setB (Double.parseDouble (args[1]));
bean.setC (Double.parseDouble (args[2]));
...
System.out.println
 ("Three-dimensional ellipsoid -- properties:");
System.out.println
 (" Volume: " + bean.getVolume ());
System.out.println
 (" Type: " + bean.getType ());
...
```

## An Ellipsoid Class

**EXAMPLE**

- Build and test the application as follows.
  - Command-line usage is shown below; or create one or more run configurations in your IDE.

**run 3 4 4**

Three-dimensional ellipsoid - properties:

Semi-axis A: 3.0

Semi-axis B: 4.0

Semi-axis C: 4.0

Volume: 201.06192982974676

Type: Oblate spheroid

Definition:

A "squashed" ellipsoid with a single axis of symmetric rotation that is shorter than the other two axes (which are equal).

**run 4 4 4**

Three-dimensional ellipsoid -- properties:

Semi-axis A: 4.0

Semi-axis B: 4.0

Semi-axis C: 4.0

Volume: 268.082573106329

Type: Sphere

Definition:

Where  $A = B = C$ , offering perfect rotational symmetry in all three dimensions.

## Naming Conventions and JavaBeans

---

- There are some simple conventions for capitalization of various types of identifiers in Java.
  - Package names are all lowercase: **com.mycom.view**
  - Class names are word-capitalized, as in **MyClass**.
  - Method and field names are also word-capitalized, but start with lowercase: **numberOfSomething**, **jobToDo**.
  - Static fields which serve as defined constants (and we will look at these later) are all uppercase, words separated by underscores, as in **MINIMUM\_BALANCE**.
- Beyond this there are conventions for naming methods, especially accessors and mutators.
  - It is so common and useful to group a field with this pair of methods that the **JavaBeans** specification standardizes it:

```
public type getName ();
public void setName (type x);
```

- These two methods imply, by convention, that there is a **property** called **name** of a specific **type** on the class.
- This is useful in general-purpose Java coding.
  - It makes one's code easier to read and to comprehend.
  - Some Java technology uses JavaBeans conventions as a way of making it possible for non-Java code to interact with a class – for instance code in a scripting language on a Web page.

## Ellipsoid as a JavaBean

### EXAMPLE

- **Ellipsoid** follows the JavaBeans conventions.
- In fact it exposes all of its logic through properties.
- Three properties **a**, **b**, and **c** are read/write – meaning there are both accessor and mutator methods, as in:

```
public double getA () { return a; }
public void setA (double newValue)
 { a = newValue; }
```

- The fact that these methods are “backed” by a private field is unimportant to the JavaBeans standard.
  - In fact what is important is that the methods could be implemented any number of ways, and that the choice of implementation is no business of the caller.
  - The information could live in a database, in a file, or perhaps in a delegate class elsewhere in memory.
- Three fields **volume**, **type**, and **definition** are read-only – only the accessor method is provided.

```
public double getVolume ()
{
}
}
```

- This is natural since the value is derived from other properties on the class.
- This is another example of a property not backed by a field on the class itself.

## The Car Class

### EXAMPLE

- In **Cars/Array/Step1**, there is a prototypical version of the car dealership that includes an implementation of the **Car** class from the first-iteration design.
- There are read-only JavaBeans properties implemented for each of the UML attributes in the original design.

– See `src/cc/cars/Car.java`:

```
private String make;
..
public String getMake ()
{
 return make;
}
```

| <<JavaBean>><br>Car            |  |
|--------------------------------|--|
| year : int                     |  |
| make : String                  |  |
| model : String                 |  |
| color : String                 |  |
| stickerPrice : double          |  |
| VIN : String                   |  |
| horsepower : int               |  |
| torque : int                   |  |
| handling : enum                |  |
| getShortName() : String        |  |
| testDrive() : TestDriveResults |  |

- In fact, to clarify that those attributes are not simple private fields but rather properties, and thus imply the presence of public methods, we define a UML **stereotype** **<<JavaBean>>**, on which attributes map to properties by default.

## The Car Class

**EXAMPLE**

- UML operations **getShortName** and **testDrive** are implemented to use various fields on the class.

```
public String getShortName ()
{
 return "" + year + " " + make + " " + model +
 " (" + color + ")";
}

public TestDriveResults testDrive ()
{
 String feedback = "";
 double factor = 1.0;

 if (horsepower > 240)
 factor = 1.1;
 else if (horsepower < 160)
 factor = 0.9;

 if (torque > 300)
 factor *= 1.1;
 else if (torque < 200)
 factor *= 0.9;

 if (factor > 1.15)
 feedback += "Feels very powerful! ";
 else if (factor > 1.0)
 feedback += "Feels powerful ... ";
 else if (factor == 1.0)
 feedback += "Power is not bad ... ";
 else if (factor >= 0.9)
 feedback += "Feels a bit weak. ";
 else
 feedback += "Feels very weak! ";
 ...
}
```

## Building the Inventory

### EXAMPLE

- In **Cars/Array/Step2** code in **src/cc/cars/Persistence.java** creates two cars and packs them into an array, by reference.

```
public Car[] loadCars()
{
 Car[] result = new Car[2];

 result[0] = new Car ("Toyota", "Prius", 2015,
 "ED9876", "Silver", 28998.99, 220, 180,
 Car.Handling.FAIR);
 result[1] = new Car ("Subaru", "Outback", 2015,
 "PV9228", "Green", 25998.99, 250, 260,
 Car.Handling.GOOD);

 return result;
}
```

- Note that the **Car** class exposes a non-default constructor.
  - It takes several parameters and initializes its properties accordingly.
  - This is essential for **Car**, since it doesn't expose mutator methods for the properties!
- Because the constructor is available, it is possible to pass the long string of arguments in using **new**.
  - Calling **new** is then much like calling a method, except that instead of an explicit method name you get the constructor.
  - But as with methods, the argument list in the call and the parameter list in the method signature must match.

## Building the Inventory

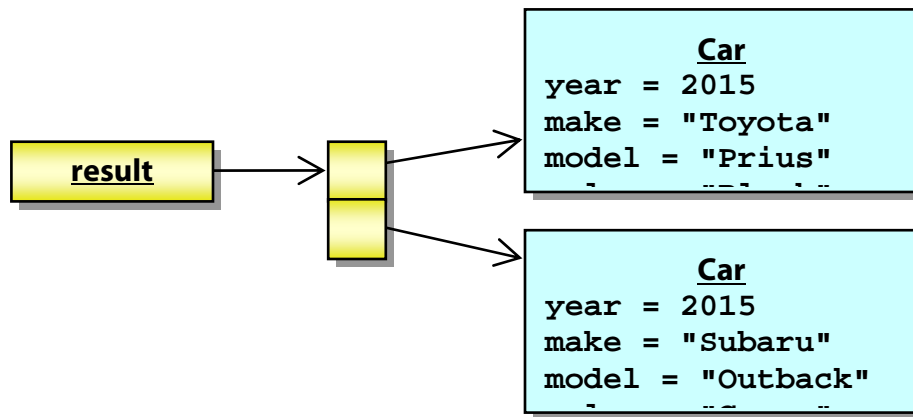
### EXAMPLE

- Also, note the difference between allocating an array of object references ...

```
Car[] result = new Car[2];
```

- ... and creating an actual object ...

```
result[0] = new Car ("Toyota", "Prius", 2015,
 "ED9876", "Silver", 28998.99, 220, 180,
 Car.Handling.FAIR);
```



- What would happen if you did one but not the other?
  - If you forgot to allocate the object references, then **result[0]** and **result[1]** would not exist.
  - In fact the array itself would not exist and **result** would be **null**; when you tried to assign **result[0]**, you'd get a **NullPointerException**.
  - If you forgot to create the objects, then **result[0]** and **result[1]** would exist, but would be **null**.
  - Whenever you tried to use them, as in **result[0].getMake**, you'd get a **NullPointerException**.



## new Objects vs. new Arrays

---

- Note that we have seen the **new** operator in two closely related syntaxes:

```
someObject = new ThatClass(5);
someArray = new ThatClass[5];
```

- Assure yourself that you are clear on the difference between the two:
  - The first construct creates a single new object of type **ThatClass** – passing the number five to the constructor.
  - The second one creates a contiguous block of five references to objects of type **ThatClass**. It does not create any objects!
- In fact, the two are commonly combined.
  - Recall that an array of objects requires two allocations: the object references and the objects themselves.

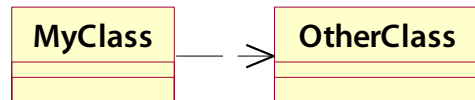
```
ThatClass[] someArray = new ThatClass[2];
someArray[0] = new ThatClass (34);
someArray[1] = new ThatClass (45);
```

## Implementing Relationships

---

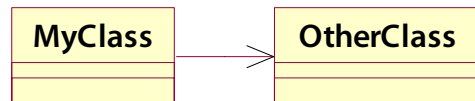
- All relationships in a class design will be expressed in Java through some use of object references.
  - Creating, using, and discarding objects of the second class.
  - Accepting references to that class as method parameters:

```
public void myMethod
 (OtherClass collaborator)
{
 collaborator.helpMeOut ();
}
```



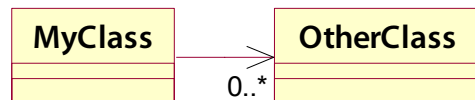
- Association of one class with another is usually implemented by adding a field to the associating class of the associated type:

```
public void myMethod ()
{
 collaborator.helpMeOut ();
}
private OtherClass collaborator;
```



- One-to-many aggregations can be implemented using arrays of object references:

```
private OtherClass[] others;
```



- Later we'll learn how to use **collections** for this same purpose.

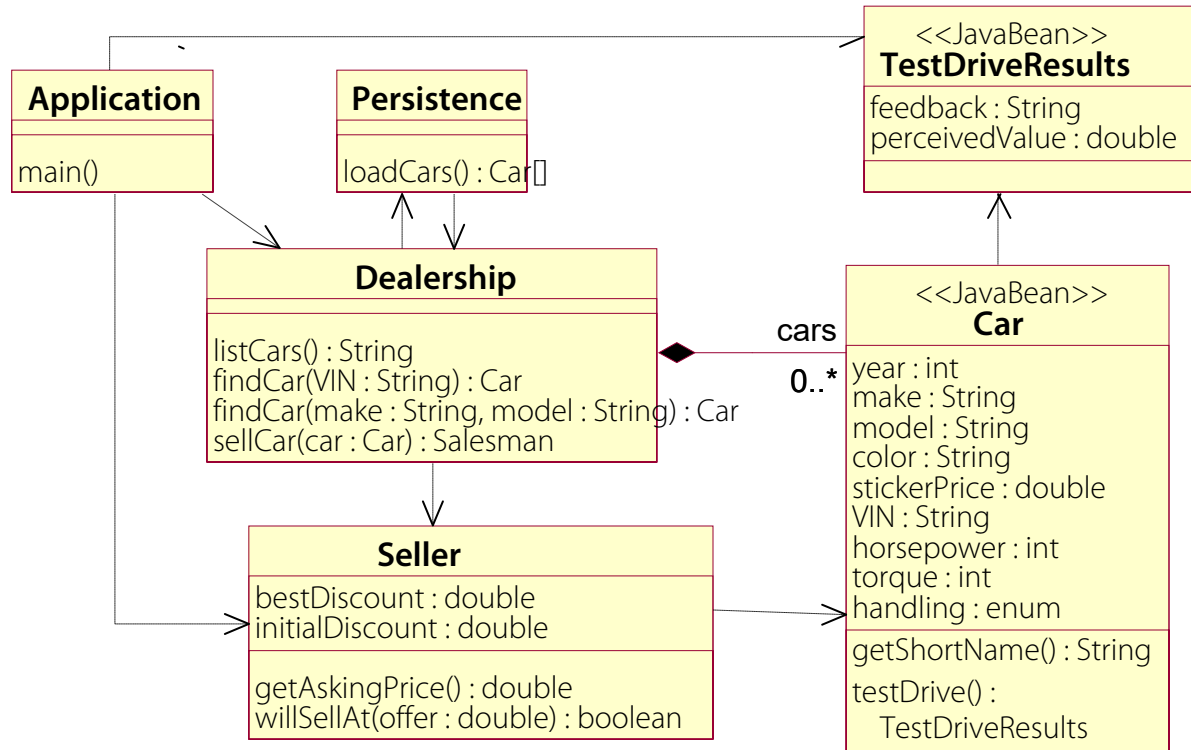
## Car Dealership Design – Second Pass

---

- As we get a better sense of what Java can do in the OO realm, we'll refine the design from the previous chapter in a few small ways.
- The **Car.testDrive** method will need to return both a perceived value (in dollars and cents) and a feedback string to print to the console.
  - Since only one value can be returned from a method, we define a class `TestDriveResults` to capture the full result set.
  - Objects of this type will be returned from **Car.testDrive** and their properties read individually by the caller.
- The new **Persistence** class handles initializing data for the **Dealership**.
  - A good design practice is to decouple **business logic** – basic application behaviors – from **persistence logic**.
  - **Persistence** just hard-codes a few **Cars** right now, but in later steps it will perform file-based persistent storage.
  - Note that the reason for a separate **Application** class is to follow another, similar rule: decoupling business logic from **presentation**, which in this case is console input/output.

## Car Dealership Design – Second Pass

- An updated UML diagram follows – note that more specifics about field types and method signatures are provided, as well:



## Using this

---

- **Methods use fields on the called object implicitly – this is a key part of Java’s encapsulation capability.**
  - The compiler manages a reference to the called object, which can also be used explicitly – it is called **this**.
- **A method may need to pass the current object reference to another method as an argument.**
  - Here the **this** reference can be used just like any other object reference – perhaps on a datebook-event object:

```
public makeAppt (Datebook book, String desc)
{
 book.add (this, desc);
}
```

- **Also, **this** can be used to override scoping rules.**
- It is legal for a class member and a method parameter to have the same name.
- In this case the compiler sees the unqualified name in code as referring to the method scope.
- You can refer to the hidden class member using **this** – an especially useful technique in constructor code, so that it’s not necessary to come up with different names:

```
public Point (int x, int y)
{
 this.x = x;
 this.y = y;
}
```

## Visibility

---

- The visibility of a member determines what other pieces of the system can see and use it.
- Object-oriented analysis and design in general recognize three levels of visibility, and in Java each is defined by a corresponding keyword:
  - **public** members are just that; anyone can use them.
  - **private** members can only be seen from inside the class.
  - **protected** members can only be seen from inside the class and from any **subclasses** – a concept we'll pursue in the following chapter – or any classes in the same package – one we'll consider right after the upcoming lab.
- Define member visibility using the **public**, **private**, and **protected** keywords.
- Note that visibility applies to classes, not to objects: two objects of the same type can see each other's private or protected members.

## Listing and Finding Cars

**LAB 6A**

**Suggested time: 45 minutes**

In this lab you will implement the listing and lookup features of the car dealership. The **Car** class is already in place and fully implemented, as are the **Persistence** and **TestDriveResults** classes. You will implement methods on the **Dealership**: **listCars** and two overloads of **findCar**. Then you will add code to the **Application** class to call these methods based on command-line input.

Detailed instructions are found at the end of the chapter.

## Packages

---

- Before going much further with classes and objects it's important to understand the concept of **packages**.
- Java uses packages to **partition the global namespace**.
  - That is, a class in one package may have the same simple name as a class in another package, because the packages **qualify** the class names.
  - Without packages, complex software systems would often run into problems with name collisions: two different programmers each contributing a class called **Record**, each of which is really meant to do something different.
  - Earlier languages ran into this difficulty and programmers were forced into self-qualifying names using prefixes and suffixes: **TkNetFirewallSettings**, or **CRM\_GUI\_Window\_Main**.
  - Even if name collisions can be avoided internally, when companies or software projects merge, collisions become more likely.
- Packages are a more elegant solution because they logically **separate the partitioning from the simple name of the class**.
  - Java provides a mechanism that makes it easy to concentrate on the simple name for a class in most source code.
  - The fully-qualified name that ultimately identifies the class is always understood in any source-code context, however.



## Package and Class Names

---

- Package names consist of a sequence of tokens.
  - Packages can contain zero to many classes.
  - They are entirely about naming and with grouping classes; they have no code of their own.
- In Java source code, a package name is represented using a dot to separate the tokens in sequence:

```
mypackage
mypackage.subsystem.part1
```

- Classes are referenced by **fully-qualified names** that begin with their package names, include another dot, and conclude with the simple class name:

```
mypackage.MyClass
mypackage.subsystem.part1.ThatOtherClass
```

- A class name must be unique within its package, hence the following class names can be distinguished by a Java compiler or runtime:

```
java.io.File
mypackage.File
```

## Organizing Class Files by Package

---

- As classes map to files, packages map to directories.
  - Thus when a class in a package is compiled, the class file will be placed in a subdirectory of the **-d** target directory, according to its package declaration.
  - The class **cc.math.EllipsoidCLI** can be compiled as follows:  

```
javac -d c:\classes EllipsoidCLI.java
```
  - The resulting class file is found at:  

```
c:\classes\cc\math\EllipsoidCLI.class
```
- In this way the logical organization of classes into packages that is recognized in Java source code is mapped cleanly to an organization of class files on the file system.
- It's considered good practice to organize source files into package-aligned directories, as well.
- It is straightforward to find a desired class based on its full class name:
  - The JRE searches the class path as usual.
  - For each directory in the path, it looks for a subdirectory tree according to the package name, as in **/cc/math**.
  - It then seeks a class file based on the simple class name.
- When launching a packaged class, use the dot syntax, not the slash syntax for directories:

```
java -classpath c:\classes cc.math.EllipsoidCLI
```

## Package Naming and Design

---

- Package names beginning with the token **java** are reserved for the Core API:

```
java.io
java.awt.event
```

- The token **javax** is reserved for **extension packages**, which are not part of the Core but many of which are deployed with the standard JRE.
- A convention for packaging your classes is as follows.
  - Start by inverting your company's Internet domain name back-to-front, as in **com.mycompany**.
  - Define a hierarchy of packages for different scopes within the domain – product or project, subsystem, etc.
- It also makes sense to gather in one package classes whose responsibilities are closely related, especially if they will need to refer to one another extensively.
- Later we will see that packages also define a fourth possible scope for member visibility.

## Imports

---

- When one class makes reference to another explicitly in its code, it can use the fully-qualified class name:

```
java.util.Vector all = new java.util.Vector ();
```

- This can be unwieldy, and repetitive.
- Java compilers recognize an **import** directive, which can be used to simplify out-of-class references.
- The directive can be used to import a single class name from another package, or to import all the class names in a package:

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

- An imported class can then be referenced more directly, by using only the class name.

```
Vector all = new Vector ();
```

## Imports

---

- You must be careful when importing whole packages: a compile-time **name collision** can result if a referenced class name is found in more than one imported package ...

```
import java.awt.*;
import java.util.*;
...
 List<Integer> listOfIntegers = new List<> ();
 // ERROR: "reference to List is ambiguous"
```

- ... or is imported explicitly more than once:

```
import java.awt.List;
import java.util.List;
 // ERROR: "java.awt.List is already defined
 in a single-type import"
```

- It is legal (and viable) to import a single symbol and another entire package that happens to contain that symbol:

```
import java.awt.*;
import java.util.List;
...
 List<Integer> listOfIntegers = new List<> ();
```

- Whether this is a good practice is a separate matter.

## Best Practices with Imports

---

- There's a dilemma here: on one hand, importing using package-level wildcards is easy on the author of the source code.
- On the other, it's a burden on anyone who wants to read the source code and to understand it.
  - The reader is stuck searching for any unfamiliar class names in any number of packages – just as the compiler must do.
- It's also challenging to keep the list of **import** statements up to date with the code itself.
  - The compiler complains if one is missing – but not if an imported symbol is unused.
  - Most IDEs will flag unused imports as warnings.
  - This can lead to junk imports being left behind.
- There are two answers to the best-practice question that this dilemma implies:
  - First, best practice is to import explicitly and not to use wildcards – i.e. go the extra mile and make your code more readable and maintainable.
  - Second, this is very easy to do! Your friendly neighborhood IDE can do all of this indexing for you and generate necessary imports.
  - Many will also weed out unnecessary ones left behind from earlier work; and even alphabetize and paragraph the code for you!

## Packaging Hello, Java

**DEMO**

- As we experimented with the Hello application's class-file location in an earlier chapter, now we'll look at how this location is affected by "packaging" the class.
  - Work in **Hello/Step1**.
  - A re-organized version of the application is in **Hello/Step2**.
- The class is currently unpackaged – it may also be said to be in the **default package**.
- We will migrate it into a package, **cc.basics**.
- Run this demo first from the command line.

1. Build and test the application as usual.

```
compile
run
Hello, Java!
```

2. Open **src/Hello.java** and add a package statement to the top:

```
package cc.basics;
```

3. Save the file and compile. Try to **run** it.

```
compile
run
Error: Could not find or load main class Hello
```

## Packaging Hello, Java

**DEMO**

4. Examine both the build and run scripts – what’s the problem?

```
javac src*.java
java -classpath src Hello
```

5. We’ll have to fix a few things in sequence. First off, we’ve changed the class name to **cc.basics.Hello**, but we’re still trying to run it as **Hello**. Change the **run** script:

```
java -classpath src cc.basics.Hello
```

6. Try again:

```
run
Error: Could not find or load main class Hello
```

7. Not much better. In fact we’ve taken a step back to go forward, because right now the **compile** script isn’t putting the class file in the right place. So before we were actually finding **Hello.class** in the working directory, but it held a class of the “wrong” name: **cc.basics.Hello**. Now, with the right name, we can’t even find the class file!
8. The problem with the build is that it’s not putting the class file in a subdirectory **cc/basics**. The reason for this is mundane; it’s just that the **javac** compiler will only direct output in this way if it’s given the **-d** switch and an output directory. So even if we direct output explicitly to the working directory, we’ll get different behavior. Change the **compile** script to be more like our general process, with a new **build** directory:

```
md build or mkdir build
javac -d build src*.java
```



## Packaging Hello, Java

**DEMO**

9. Change **run.bat** accordingly:

```
java -classpath build cc.basics.Hello
```

10. Build and test again:

```
compile
```

```
run
```

```
Hello, Java!
```

- Notice that **Hello.class** now lives in **build/cc/basics**.
- If you like, try the same experiment, using your IDE, in **Hello/Step1**.
  - Some IDEs will flag the **package** declaration at the top of the source file as an error.
  - So while **javac** will go ahead and compile the source file, regardless of its location, the IDE is enforcing its idea of best practice and insisting you align the logical package with the physical location.
- 11. The IDE may offer a “quick fix” by which it moves the source file into place for you. Hover over the error and when the Quick Fix popup appears, choose “Move ‘Hello.java’ to package ‘cc.basics’.”
  - You’ll be back in business.

## Car Dealership Packages

---

- The car dealership is designed with classes in multiple packages.
- The primary package is **cc.cars**, and you have done all your work here thus far. This includes:
  - **Car**
  - **Seller**
  - **Dealership**
  - **Application**
  - **Persistence**, which is not in the first-iteration design but will be added in this chapter
  - **TestDriveResults**, which also is added in this chapter
- A second package **cc.tools** holds generic utilities.
  - Right now this package is empty.
  - Later a utility class **UserInput** will appear, to facilitate gathering purchase price offers from the interactive user.
- A third package **cc.cars.sales** will appear many steps later.

## Package Visibility

---

- Java adds fourth level to the three that are common to OO or UML-friendly languages; this is known as **default** or **package** visibility.
  - It is called **package** visibility because it dictates that the member can be seen only by classes that share the enclosing class' package.
  - It is called **default** visibility because there is no keyword used to define it; it applies by default.
- The potential use of this level can have a great deal of impact on choices regarding the best package design.
  - A group of related classes may be defined to share a package.
  - Only classes in that group can see package-visible fields and methods that they declare.
  - One class might define a utility method, or a logging object, or other shared resource, for the general use of package members.
- Note that package visibility does not extend to so-called “subpackages.”
  - A package-visible member defined in package **A** cannot be seen from code in package **A.B**, nor vice-versa.
  - This is a surprise to most programmers, but – to give a concrete example – the packages **java.awt** and **java.awt.event** have no special relationship except in the way they are named.

## Overloaded Methods

---

- A method is not distinguished solely by its name.
- You can provide multiple **signatures** with the same method name – or for a constructor.
  - This is called **overloading** a method.
  - The caller can call the method with arguments to match any of the overloaded parameter lists.

```
public class HumanResources
{
 public Employee findEmployee (long ID);
 public Employee findEmployee (String name);
 ...
}
```

- It is assumed that the synonymous methods will do more or less the same thing, based on different input.
- One common use of overloading is to provide default values for some parameters in the original signature.
  - One method with fewer parameters can call another overload of the method, providing the default arguments.

```
public class Dialer
{
 public void dial (int number, int retries) {...}
 public void dial (int number)
 {
 dial (number, 3);
 }
}
```

## Overloading Constructors

---

- Multiple constructors can be defined for a class.
- This is essentially a form of overloading, since there is no method name to distinguish one constructor from another.
  - Only the parameter list can identify a constructor.
- Sometimes one constructor will fully initialize the fields of the class based on a long parameter list, and another one will do nothing, leaving initialization to later calls to mutator methods.
  - **Ellipsoid** has only this latter approach, while **Car** has only the former one.
- It is also possible for one constructor to invoke another one explicitly, just the way a method can call a different overload.
  - Use the **this** keyword as a method name to invoke a different constructor on the same class for the same object.
  - Define default values for different constructors in this way:

```
public class Dialer
{
 public Dialer (int number, int retries) { ... }
 public Dialer (int number)
 {
 this (number, 3);
 }
 public void dial () { ... }
}
```

## Rules for Overloading

---

- Overloads must be differentiated by their signatures – that is, names and parameter lists.
  - Different overloads can have different return types.
  - But return type alone cannot be used to disambiguate method signatures when the compiler is trying to build a call.
- Consider the following code, meant to allow the caller to get the **SSID** attribute as a number or as a string:

```
public class MyClass
{
 public int SSID ();
 public String SSID ();
}
```

- The compiler would not be able to decide for sure which method signature should be invoked based on the call as shown, so it disallows the overload.
- Some good news: the compiler can decide on a method using widening (implicit) type conversions, which it will favor over narrowing conversions:

```
public class MyClass
{
 public void sendIt (short number);
 public void sendIt (long number);
}
...
myInstance.sendIt (5L); // long
myInstance.sendIt ((short) 5); // short
myInstance.sendIt (5); // long!
```

## Finding More Cars, and Driving Them

**LAB 6B**

**Suggested time: 30 minutes**

In this lab you will implement an overload of **findCar** to allow the user to seek out a car by make and model. You will also implement the test-drive feature of the application, calling the simpler **findCar** and then calling **Car.testDrive** to get results for the user.

Detailed instructions are found at the end of the chapter.

## Selling Cars

**LAB 6C**

**Suggested time: 30 minutes**

In this lab you will create the Seller class to complete this iteration of car-dealership development. You will build this class from scratch, implement its two methods, and then write code to create and call it from the existing application framework.

Detailed instructions are found at the end of the chapter.



## The Java ARchive

---

- The **Java ARchive**, or **JAR**, offers an alternative to placing class files directly on the local file system.
  - JARs can include class files and other resources, such as data files, sound clips, images, etc.
  - A JAR may be built with compression or not.
  - A JAR is a single file that is much easier to deploy elsewhere than the local file system: to ship as part of an application or toolkit, for instance.
- JARs can be placed on the class path, just like directories.
  - A JAR captures an internal directory structure, and so package subdirectories such as **cc/math** or **cc/cars** can be preserved when creating the archive.
  - The JVM can load classes out of subdirectories of the JAR.
- In fact, the entire Core API is captured in one immense JAR.
  - **rt.jar** is deployed with the JRE.
  - It is implicitly loaded as part of the “boot class path.”
- Various Java tools and APIs are deployed in JAR form.

## Working with JARs

---

- Build JARs with the **jar** tool, included in the JDK.
  - The **c** option directs the tool to create an archive.
  - The **x** option directs it to extract files.
  - The **t** option can be used to inspect an archive's table of contents.
  - The **f** option says you'll be supplying filenames or wildcards.
- For instance, to pack up all the class files in a subdirectory of the working directory:

```
jar cf MyNewJar.jar cc\cars*.class
```

- The JAR can then be sent around the world – emailed, published and downloaded, etc. – and placed in a local classpath:

```
java -classpath MyNewJar.jar cc.cars.Application
```

## Java in a JAR

**LAB 6D****Optional**

**Suggested time: 15 minutes**

In this lab you will experiment with creating and using JARs.

This exercise too you should do entirely from the command line. There are means of getting IDEs to produce JARs as build targets, but it's beyond the scope of the course to walk through that process.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- Java provides complete support for implementation of object-oriented designs.
- In fact, Java relies heavily on classes for its basic functionality as an environment, partially through enforcement of the rule that all classes ultimately inherit **java.lang.Object**.
- We will study this more carefully in the next chapters, in which we focus on inheritance and polymorphism as supported by the Java language.
- Java is pointedly object-oriented, and all code in a Java application must live in some class.
  - It is possible to write non-OO applications in Java – as we’ve seen in previous examples and labs.
  - However the real purpose of Java is to enable OO implementations in which the application at runtime exists as a system of objects in memory, with relationships to each other that allow them to collaborate to solve the problem at hand.

## Listing and Finding Cars

**LAB 6A**

In this lab you will implement the listing and lookup features of the car dealership. The **Car** class is already in place and fully implemented, as are the **Persistence** and **TestDriveResults** classes. You will implement methods on the **Dealership**: **listCars** and one version of **findCar**. Then you will add code to the **Application** class to call these methods based on command-line input.

**Lab project:** Cars/Array/Step3

**Answer project(s):** Cars/Array/Step4

**Files:** \* to be created  
src/cc/cars/Dealership.java  
src/cc/cars/Application.java

### Instructions:

1. Review the starter code, including the **Car** and **Persistence** classes.
2. Open **Dealership.java**, and see that it holds a skeleton for the class, with just a field **cars** defined to hold the inventory.
3. Implement a public default constructor to create a new **Persistence** object and to call **loadCars**, storing the result in our **cars** array:

```
public Dealership ()
{
 Persistence persistence = new Persistence ();
 cars = persistence.loadCars ();
}
```

4. Define the **listCars** method as shown in the UML diagram: it takes no parameters and returns a **String**.

```
public String listCars ()
{
}
```

### Dealership

```
listCars() : String
findCar(VIN : String) : Car
findCar(make : String, model : String) : Car
sellCar(car : Car) : Salesman
```

**Listing and Finding Cars****LAB 6A**

5. The method should create a **String** called **result**, and then iterate over the **cars** array using a **for** loop – you can use the simplified form in this case. For each element in the array, call **getVIN** and **getShortName** and add these to **result**. Format the result – note that the **String** class has its own **format** method! – to put a colon after the VIN and add and end-of-line after the short name.
6. Return **result** from the method.
7. Open **Application.java** and see that there is a skeleton **main** method and a helper method **die** that prints a complete usage statement and terminates the program.
8. Start your **main** implementation by checking the length of the **args** array; if there's nothing there at all, call **die**.
9. Create a new **Dealership** and call it **dealership**.
10. Now test **args[0]** to see if it's equal to "list". (Remember that you can't do this with the **==** operator!) If it is, call **dealership.listCars**, and print the result to the console. If it's not, call **die**. (You will add **else if** clauses in between this **if** and the final **else** to support additional commands; thus, as you proceed, any call to a command not yet implemented will print a usage statement.)
11. Save both files and test the application as follows. You should see output as shown – if not, check for code errors and try again, or consult your instructor.

```
run list
ED9876: 2015 Toyota Prius (Silver)
PV9228: 2015 Subaru Outback (Green)
...
IM3110: 2014 Ford Escape Hybrid (Silver)
PU4128: 2015 Audi A3 (Blue)
```

Later steps also have code to prompt the user for a command, which makes testing from an IDE a little smoother.

**Listing and Finding Cars****LAB 6A**

12. That's the list feature; now you'll tackle finding cars. Back in **Dealership.java**, create a **findCar** method based on the first of the UML-diagrammed ones: it should take a single **String** called **VIN**, and return a **Car**.
13. Again, loop through the **cars** array. This time, for each element, see if **getVIN** equals the passed **VIN**. If it does, return that element of the array as the found car.
14. If the loop "falls through" – that is, completes without finding a match – then return **null**.
15. In **Application.java**, add a test for "price" after your implementation of the "list" command:

```
if (args[0].equals ("list"))
{
 System.out.println (dealership.listCars ());
}
else if (args[0].equals ("price"))
{
}
else die ();
```

16. Start your implementation of "price" by checking to see that there is at least a second command-line argument; if **args.length** is less than two, call **die**.
17. Create a local **Car** reference called **found**, and initialize it to a call to **dealership.findCar**, passing **args[1]** as the argument.
18. If **found** is **null**, print a statement saying the car could not be found by the given VIN.
19. Otherwise, print the VIN and short name of the car, and then produce the results of **found.getStickerPrice**.
20. Build and test at this point:

```
run price XYZ
Car not found.
```

```
run price ED9876
ED9876: 2015 Toyota Prius (Silver) $28,998.98888889
```

21. You may or may not get the longer, uglier price representation shown above. To improve the looks of the price, wrap **found.getStickerPrice** in a call to **System.out.format**, using a formatting string **"\$%,1.2f"** for the currency value. Test again:

```
run price ED9876
ED9876: 2015 Toyota Prius (Silver) $28,998.99
```

## Finding More Cars, and Driving Them

**LAB 6B**

In this lab you will implement an overload of **findCar** to allow the user to seek out a car by make and model. You will also implement the test-drive feature of the application, calling the simpler **findCar** and then calling **Car.testDrive** to get results for the user.

**Lab project:** Cars/Array/Step4

**Answer project(s):** Cars/Array/Step5

**Files:** \* to be created  
src/cc/cars/Dealership.java  
src/cc/cars/Application.java

### Instructions:

1. Implement the second overload of **findCar** from the UML design: the one that takes a make and model instead of a VIN. Add that method to **Dealership** and implement it almost the same way the first version is implemented. The difference is that you test **getMake** against the make and **getModel** against the model, instead of testing the VIN. As with the first method, this one should return immediately if it finds a hit and return **null** if the loop falls through.
2. In **Application.java**, modify your “price” handler to call one overload or the other depending on how many command-line arguments exist. If **args.length** is two, call the one-argument method as you’re already doing, but if it’s three, call the two-argument method, passing **args[1]** and **args[2]**. You could do this with **if-else**, but especially when initializing a new local variable, the ternary flow-control operator **?:** can be very handy, as in:

| Dealership                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------|
| listCars() : String<br>findCar(VIN : String) : Car<br>findCar(make : String, model : String) : Car<br>sellCar(car : Car) : Salesman |

```
Car found = (args.length == 2)
 ? dealership.findCar (args[1])
 : dealership.findCar (args[1], args[2]);
```

3. Build and test – you should be able to find by make and model as follows:

**run price Subaru Outback**

PV9228: 2015 Subaru Outback (Green) \$25,998.99



**Finding More Cars, and Driving Them****LAB 6B**

4. Implement a final command “drive” in **Application.main**. You won’t need any additional support from the dealership on this one, because you already know how to find a car, and test-driving is a behavior on **Car**. Add a new **else if** to the command-processing chain in **Application.main**, and start your implementation by checking that there are at least two command-line arguments; **die** if not.
5. Then call **dealership.findCar** as you did in your original “price” code, again assigning the result to a local **Car** reference called **found**. If **found** is **null**, print an error message; otherwise, call **found.testDrive**, and capture the results in a local **TestDriveResults** reference called **result**.
6. Print the results to the console using **result.getFeedback** and **result.getPerceivedValue** – you’ll want to format this second expression using **System.out.format** as it’s a dollar-and-cents value.
7. Build and test – you should see results like this:

```
run drive ED9876
Feels a bit weak. Handling is fair.
Seems like this car is worth $26,099.09.
```

```
run drive PV9228
Feels powerful ... Handling is good.
Seems like this car is worth $31,198.79.
```

# Selling Cars

**LAB 6C**

In this lab you will create the `Seller` class to complete this iteration of car-dealership development. You will build this class from scratch, implement its two methods, and then write code to create and call it from the existing application framework.

**Lab project:** Cars/**Array/Step5**

**Answer project(s):** Cars/**Array/Step6**

**Files:** \* to be created  
**src/cc/cars/Seller.java** \*  
**src/cc/cars/Dealership.java**  
**src/cc/cars/Application.java**

## Instructions:

1. Create the new source file **cc/cars/Seller.java**. Define a public class **Seller** in package **cc.cars**.

2. Define three fields on the class:

```
private Car car;
private double initialDiscount;
private double bestDiscount;
```

3. Define a public constructor that takes three parameters to match the three fields:

```
public Seller (Car car, double initialDiscount, double bestDiscount)
{
}
```

4. Implement the constructor by assigning each field to the corresponding argument. Remember that when a field and a method parameter have the same name, you can reference the parameter directly and the field using **this**, as in:

```
public Seller (Car car, double initialDiscount, double bestDiscount)
{
 this.car = car;
 this.initialDiscount = initialDiscount;
 this.bestDiscount = bestDiscount;
}
```

5. Define the methods designed for `Seller`, as shown in the UML above.
6. Implement **getAskingPrice** by getting the sticker price of the **car** field and discounting it by **initialDiscount**. (Hint: simplest way to do this is multiply the sticker price by **(1 – initialDiscount)**.)

| Salesman                                                          |
|-------------------------------------------------------------------|
| bestDiscount : double<br>initialDiscount : double                 |
| getAskingPrice() : double<br>willSellAt(offer : double) : boolean |

**Selling Cars****LAB 6C**

7. Implement **willSellAt** to test the offer price against the sticker price discounted by **bestDiscount**. If the offer is at least as good as this “best price,” return **true**, otherwise **false**. (Hint: you don’t need an **if-else** to do this! Remember that you can return a boolean expression directly, and that’s the right return type for this method.)
8. Add the **sellCar** method to **Dealership**. It should take a single parameter of type **Car**, and return a **Seller**. Implement this to create a new **Seller**, passing the car and standard values for the discounts of 10% and 15%:

```
public Seller sellCar (Car car)
{
 return (new cc.cars.Seller (car, .1, .15));
}
```

9. In **Application.java**, add another command, “buy”. Start by checking that there are at least two command-line arguments, then find the car by the passed VIN and print an error message if it’s not found.
10. Call **dealership.sellCar**, passing the **found** car and capturing a reference to the returned **Seller** in a variable **seller**.
11. Print the asking price of the car based on **seller.getAskingPrice**.
12. Solicit an offer from the user with the new **UserInput** utility. To use this, just call the method **UserInput.getString** – this will wait for the user to type a line of text into the console, and return the string the user typed. Convert this string to a **double** using **Double.parseDouble**, and pass it to **seller.willSellAt**.
13. If **willSellAt** returns **true**, say that the car has been sold, and if **false**, say there’s no deal.
14. Build and test at this point. You should now be able to run the “buy” command, see the asking price, and make an offer, which should be accepted if its within 15% of the sticker price:

```
run buy ED9876
"I can sell the car for $26,099.09."
10000
"Nope -- can't do it."

run buy ED9876
"I can sell the car for $26,099.09."
26000
"Sold!"
```

## Selling Cars

## LAB 6C

### Optional Steps

15. Enhance the “buy” command to allow the user to make multiple offers, in a loop, until either the car is sold or the user decides to walk away by typing “quit”. You’ll need to enclose some of your existing code in a loop, with conditions that terminate the loop when the user enters “quit” and when the car is sold. You could take a strictly structured approach to this using a flag such as **carSold**, or you could use **break** to exit the loop in certain cases. (The answer code goes the former route.)
16. Build and test, aiming for behavior as follows:

```
run buy ED9876
"I can sell the car for $26,099.09."
23000
"Nope -- can't do it."
24000
"Nope -- can't do it."
26000
"Sold!"
```

## Java in a JAR

**LAB 6D**

In this lab you will experiment with creating and using JARs.

This exercise too you should do entirely from the command line. There are means of getting IDEs to produce JARs as build targets, but it's beyond the scope of the course to walk through that process.

**Lab project:** **PackageAndJAR/Step1** (optional)06  
**PackageAndJAR/Step2**

**Answer project(s):** **PackageAndJAR/Step3**

**Files:** \* to be created  
**src/Translator.java**  
**compile.bat** or **compile**  
**run.bat** or **run**

### Instructions:

#### Optional Starter Steps

1. If you would like more exercise in dealing with Java packages, start your work in **PackageAndJAR/Step1**. The code here is not yet in a package – you can take it through the same transition as the earlier demonstration did for **Hello.java**.
2. First, give the application a quick test – it translates English input to pig-Latin output. Hit **Ctrl-C** to quit:

```
a peck of pickled peppers
aay eckpay ofay ickledpay epperspay
```

3. Add a package statement to **Translator.java** to put its class in the package **cc.language**.
4. Make a new **build** directory, and change the **compile** script to direct compiler output to it.
5. Change the **run** script to run **cc.language.Translator** from the **build** directory. Now the code is in its own package and you should be able to build and re-test to see the same results.

This is the intermediate version in **PackageAndJAR/Step2**. Continue your work, or simply pick up in **PackageAndJAR/Step2**.

**Java in a JAR****LAB 6D**

6. Add commands to the **compile** script to create a JAR with the class file in it:

```
javac -d build src\cc\language*.java
cd build
jar cf ../MyApps.jar cc\language*.class
cd ..
```

7. Run the script and see that you have a new JAR file in the lab directory. Run the **jar** utility again to show a table of contents of the JAR:

```
jar tf MyApps.jar
META-INF/
META-INF/MANIFEST.MF
cc/language/Translator$Vowel.class
cc/language/Translator.class
```

8. You can see the class file and that the subdirectory structure has been preserved. (The **MANIFEST.MF** is a standard part of a JAR file – it can be used to provide meta-data about the contents of the JAR, such as which classes should be loaded as starting points for execution, which are enterprise components of certain types, etc.)
9. Delete the entire **build** directory. Try running, and of course it will fail, because the class file is missing from the class path.

10. Modify the **run** script to include the JAR on the class path:

```
java -classpath MyApps.jar cc.language.Translator
```

11. Test this out and you should get the usual results.

This is the final answer version in **PackageAndJAR/Step3**.

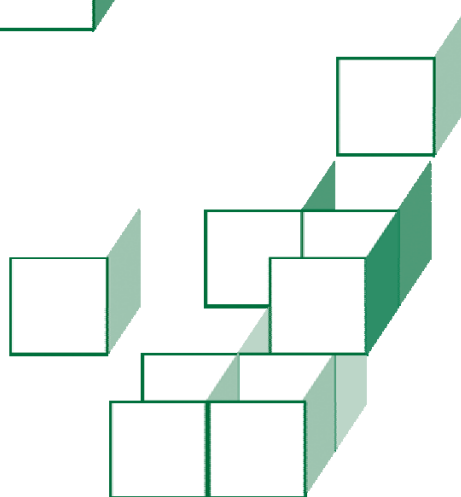
**Optional Final Step**

12. Try building a JAR of the current car-dealership application and test it out.



## CHAPTER 7

# INHERITANCE AND POLYMORPHISM



## OBJECTIVES

*After completing “Inheritance and Polymorphism,” you will be able to:*

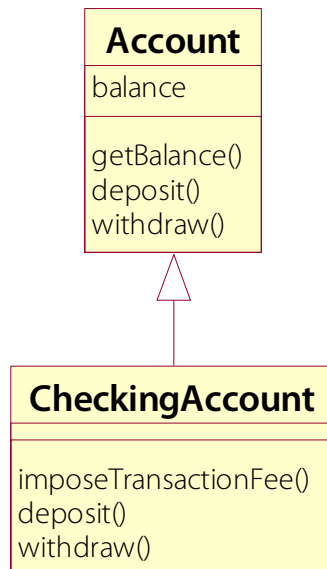
- Implement specialization relationships in class designs by extending Java classes.
- Identify the runtime type of an actual object and make appropriate reference typecasts so as to be able to take full advantage of most-derived semantics.
- Override superclass method implementations to achieve polymorphism as called for by a class design.
- Make explicit reference to the superclass where necessary, such as in leveraging superclass implementations of overridden methods.



## UML Specialization

---

- Often some or all of the process of analyzing a complex problem comes down to finding a useful way to **classify** the concepts that are emerging.
- For the manufacturing abstraction of a car, for instance, there may be many kinds (classes) of cars.
  - These classes may have a lot in common, and for a number of reasons it is good to pull those commonalities together.
  - Then the things that distinguish various makes of car, such as different engines, wheels, body panels, interiors, etc., can be expressed as **specializations** of the general concept.
- The OO-speak for this identifies the general concept as a **base class** or **supertype**, and the specializations as **derived classes** or **subtypes**.



## Advantages of Specialization

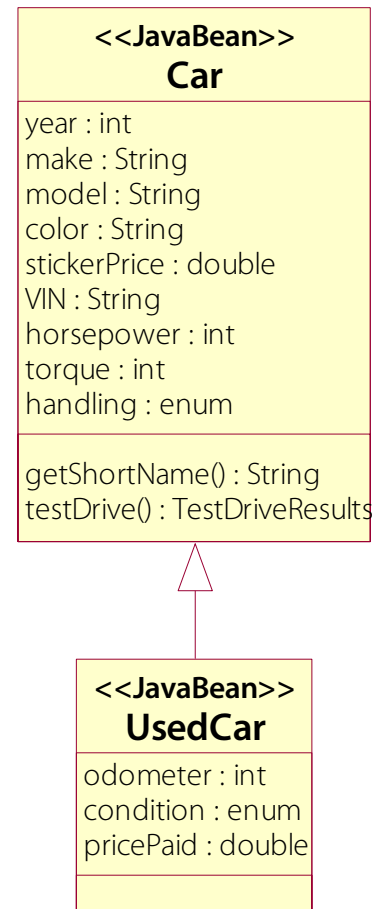
---

- **Why is specialization useful in design?**
  - So much of OO design is “finding a place for everything and putting everything in its place.” In other words we want **single points of maintenance**.
  - Specialization (with its implied opposite, **generalization**) provides one important way to avoid expressing the same concept repeatedly in the design.
  - Things that are common to related concepts are captured in the generalized type, and are expressed once – and implemented once, and maintained in one place, not in several.
- **Why is specialization useful in implementation?**
  - The same benefits of single maintenance points apply to implementation as to design – with greater impact, if anything.
  - Also, OO languages, Java included, have features that make it easy to work with specialized objects.
  - It is possible for code written to use a generalized type to interact with a specialization of that type at runtime – without being rewritten just for the specialized type.
  - That is, references to class **Base** can be assigned to objects of type **Base** or **Derived** at runtime.
  - This extends the advantage of common features in the base type to the caller, who doesn't need to write special code for each specialized type to use these general features.

## Used Cars

### EXAMPLE

- We'll begin another iteration of car dealership design by adding the requirement to support used and new cars.
- Clearly, new and used cars have a lot in common:
  - All of the fields we currently implement – **year, make, model, price, handling**, etc. – are relevant to used cars.
  - The existing behaviors – **getShortName** and **testDrive** – are also applicable.
- But used cars bring some new concerns:
  - We'll say that a used car must be modeled to include its mileage, general condition, and the price we paid for it. (Let's assume that dealers get new cars at standard markups from the manufacturer, so price-paid hasn't been an issue before now.)
  - For the moment, used cars have no new behaviors, but we'll return to that question later in the chapter.
- Thus we model the **UsedCar** as a specialization of the **Car**.



## Extending a Class

---

- To implement a specialization relationship in Java, you will **extend** one class to create another.
  - Use the **extends** keyword in defining the class:
- The extending or **derived** class (or **subclass**, or **subtype**, etc.) then automatically holds the same fields and methods as the **base** class (or **superclass**, etc.).
  - A class can have only one base class – “single inheritance.”
- The extending or **derived** class (or **subclass**, or **subtype**, etc.) then automatically holds the same fields and methods as the **base** class (or **superclass**, etc.).
  - It can then be defined to add its own fields and methods.
  - This gives an immediate and obvious benefit in code reuse.
- Note that constructors are **not** inherited.
  - For reasons having to do with the mechanics of building a new object in memory, distinct constructors must be defined for each subclass in an inheritance tree.
  - This means that a derived class must redefine constructors.
  - Use the keyword **super** to invoke a specific base-class constructor from within a derived-class constructor.

```
public Derived (int x, int y, int z)
{
 super (x, y);
 this.z = z;
}
```

## Implementing UsedCar

**EXAMPLE**

- In **Cars/Array/Step7**, the **UsedCar** class has been mostly implemented.
  - See **src/cc/cars/UsedCar.java**:

```
public class UsedCar
 extends Car
{
 private int odometer;
 private Condition condition;
 private double pricePaid;

 public enum Condition
 { EXCELLENT, GOOD, FAIR, POOR, WRECK };

 public UsedCar (String make, String model, ...)
 {
 super (make, model, year, VIN, color,
 stickerPrice, horsepower, torque, handling);
 this.odometer = odometer;
 this.condition = condition;
 this.pricePaid = pricePaid;
 }

 public int getOdometer ()
 { return odometer; }

 public Condition getCondition ()
 { return condition; }

 public double getPricePaid ()
 { return pricePaid; }
}
```

## Implementing UsedCar

**EXAMPLE**

- Do a quick build of the application.
- Note especially the constructor for **UsedCar**.

```
public UsedCar (String make, String model,
 int year, String VIN, String color,
 double stickerPrice, int horsepower, int torque,
 Handling handling, int odometer,
 Condition condition, double pricePaid)
{
 super (make, model, year, VIN, color,
 stickerPrice, horsepower, torque, handling);

 this.odometer = odometer;
 this.condition = condition;
 this.pricePaid = pricePaid;
}
```

- The derived class defines three new fields.
- To construct a used car, then, one needs to provide all of the basic values for a car, and also **odometer** reading, **condition**, and **pricePaid**.
- The constructor explicitly invokes the superclass constructor as its first step, using **super**.
- It is illegal to invoke **super** after any other code in the constructor
  - it must be the first step.
- So there is some rewriting to do here, in order to get the benefits of the base class **Car**.
- But then the **Car** constructor handles assigning all the fields, and all its accessor methods are available for a used car.

## Implementing UsedCar

### EXAMPLE

- Try deleting the constructor code and compile again.
- Naturally, there are several errors. Most of these have to do with the **Persistence** class trying to invoke a constructor that's no longer there.
- But notice the one on **UsedCar** itself:

```
src\cc\cars\UsedCar.java:14: error: constructor Car
 in class Car cannot be applied to given types;
public class UsedCar
 ^
 required: String,String,int,String,String,
 double,int,int,Handling
 found: no arguments
 reason: actual and formal argument lists differ
 in length
```

- This is one of the Java compiler's more infuriating error messages.
  - It's saying that we're trying to invoke the default **Car** constructor – but the only constructor **Car** defines takes a long list of parameters.
  - It points to the definition of **UsedCar** as the offending source code!

## Implementing UsedCar

### EXAMPLE

- The **UsedCar** source doesn't define a constructor.
- Therefore the compiler generates a public default constructor automatically.
- What we haven't seen yet is that this generated constructor must somehow invoke one of the constructors on the base class, to keep the process of object creation working correctly.
  - But which one?
  - Since the default constructor has no parameters, it couldn't provide any in calling **super**. So it looks for a base-class constructor that also takes no arguments ...
  - ... and **Car** doesn't have one!
- If the code for the generated constructor were written into the source file, it would look like this:

```
public UsedCar ()
{
 super ();
}
```

- Hence the error message saying that we tried to invoke **super** without providing arguments, and the lack of any real source code to point us to in spelling out the error.
- This is a common occurrence in Java development.



## Using Derived Classes

---

- Instantiating a derived class is no different from instantiating any other class.

```
UsedCar used = new UsedCar (1999, "Saturn", ...);
```

- The derived-type object that results offers both its own fields and methods and those of the base class.

```
System.out.println (used.getShortName ());
```

- It is also possible to assign a base-type reference to a derived-type object, or to pass a derived-type object to a method that takes a base-type reference as one of its parameters.

```
Car anyCar = new UsedCar (1999, "Saturn", ...);
```

- The reference can be used to call any base-type methods on the object, but not to call derived-type methods.

```
anyCar.getStickerPrice ();
anyCar.getOdometer (); // won't compile!
```

- The compiler won't allow the second line shown above, because while it's certain that anything referenced through **anyCar** is a **Car**, it can't guarantee that the object is a **UsedCar**.
- This is structurally similar to what we've already seen with primitive types, such as **ints** and **longs** – one can safely be treated as the other, but not vice-versa.

## Type Conversion

---

- You can cast an object reference to a different type using the same style employed in casting primitives such as numbers.
- Java compilers view a cast from derived to base type as a **widening** conversion.
  - Such a conversion makes only safe assumptions: all the members that might be legally referenced through a base-class reference are sure to be there on the derived type.
  - Therefore the compiler will perform this sort of conversion implicitly when called for:

```
anyCar = used;
Car anyCar = new UsedCar (1999, "Saturn", ...);
```

- ... but then it will prohibit derived-type calls to go through the base-type reference, as it always does.
- Casting from base-class to derived-class reference is a **narrowing** conversion.
  - Here the compiler cannot guarantee that the referenced object can do everything it might be asked to do through the derived-type reference, and this makes compilers nervous.
  - Therefore it will insist that the cast be made explicitly:

```
usedCar = (UsedCar) anyCar;
```

- When you make an explicit **downcast** of a reference, you are assuring the compiler that calls through that reference will be safe.

## Type Identification

---

- If you do not know the exact runtime type of an object, you can find it in several ways.
- The **instanceof** operator provides a simple test that an object is either of a given type or a type derived from it.

```
boolean isOfTypeA = myObject instanceof A;
```

- The operator takes a left-hand argument of type **Object** and a right-hand argument that is a class name the compiler knows, because it is in the package or imported into the source file.
- It returns **true** if the object is of the specified type or of a type that **extends** or **implements** the specified type, and **false** otherwise.
- You can (and should) use **instanceof** to assure that your downcasts are safe.

```
if (anyCar instanceof UsedCar) {
 UsedCar used = (UsedCar) anyCar;
 System.out.println (used.getCondition ());
}
```

- If you don't check with **instanceof**, the compiler will allow the downcast – you're overruling it, essentially – but at runtime you might get a **ClassCastException** if the runtime type of the object is not as expected.

## Type Conversions

### EXAMPLE

- We've seen the example in **Conversions** illustrating narrowing and widening conversions between primitive-type values.
- The last section of the **main** method also exercises conversions on various object references.
  - Two dead-simple types are defined at the bottom of the file:

```
class Base
{
 public int x;
}
```

```
class Derived
 extends Base
{
 public int y;
}
```

- **main** uses them as follows:

```
Derived myObject = new Derived ();
Base myBaseRef = myObject;
```

```
Derived myDerivedRef = null;
if (myBaseRef instanceof Derived) // will be true
 myDerivedRef = (Derived) myBaseRef;
```

```
Base baseRef = new Base ();
Derived derivedRef2 = null;
if (baseRef instanceof Derived) // will be false
 derivedRef2 = (Derived) baseRef; // would fail!
```

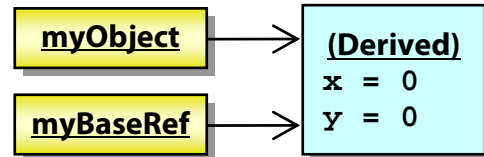
## Type Conversions

### EXAMPLE

- It's worth a moment to walk through this code: see what's happening in memory and note the tension between compile-time types of object references and run-time types of objects.

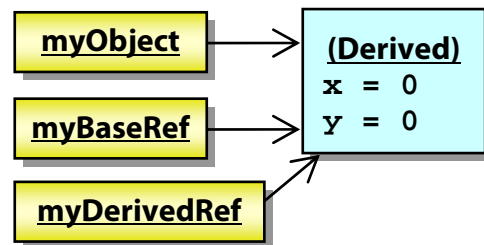
```
Derived myObject = new Derived ();
Base myBaseRef = myObject;
```

- We have two object references to a single object.



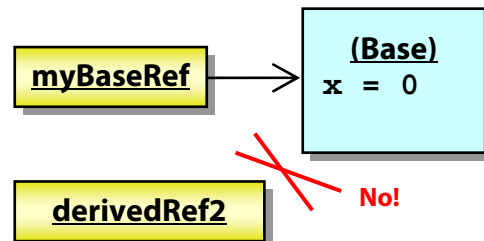
```
Derived myDerivedRef = null;
if (myBaseRef instanceof Derived) // will be true
 myDerivedRef = (Derived) myBaseRef;
```

- Now **myDerivedRef** is assigned to the object as well, since the object (not the reference) is of type **Derived**.



```
Base baseRef = new Base ();
Derived derivedRef2 = null;
if (baseRef instanceof Derived) // will be false
 derivedRef2 = (Derived) baseRef; // would fail!
```

- Now **baseRef** is assigned to a **Base**-type object. The **instanceof** fails, fortunately, so a bad assignment from **myDerivedRef2** to this object is avoided.



- Otherwise, what would happen if we tried to read **myDerivedRef2.y**?

## Type Identification and Casting

---

- Since identification and casting so often go together, there is a shorthand that simplifies the process.

- Instead of checking and then casting, like this ...

```
if (obj instanceof MyClass) {
 MyClass mine = (MyClass) obj;
 ...
}
```

- ... you can instead supply a variable name in the **instanceof** check, which will be populated with a derived-type object reference if the check succeeds:

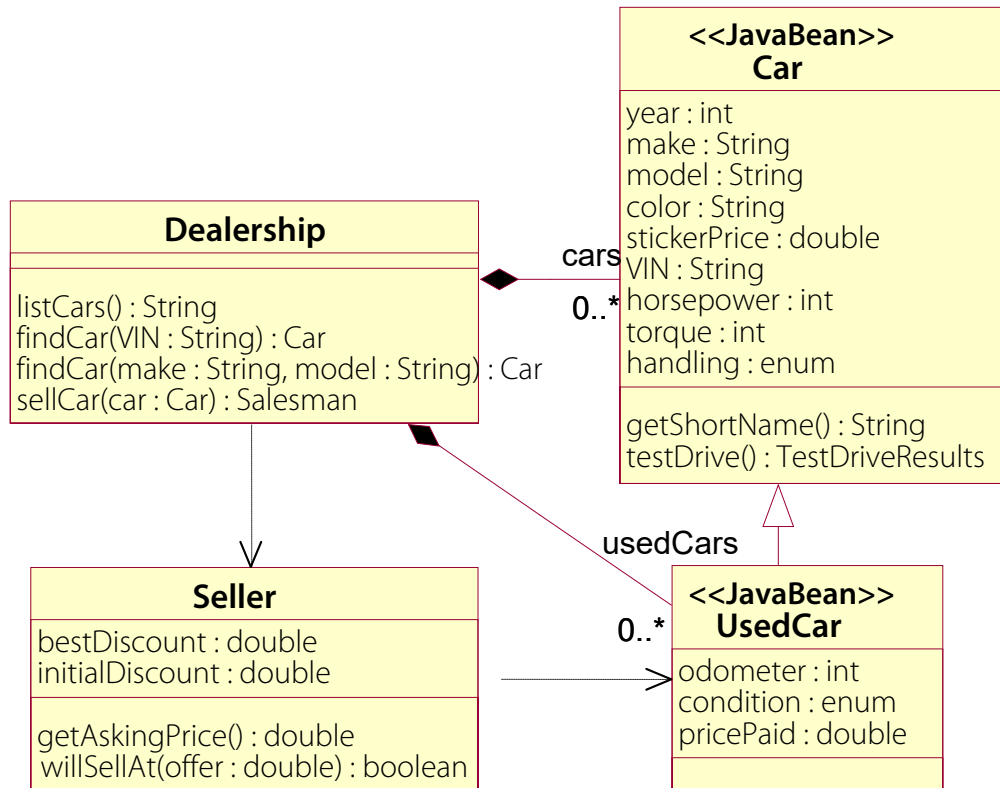
```
if (obj instanceof MyClass mine) {
 ...
}
```

- It's also safe to use this check-and-cast syntax in a short-circuited logical expression:

```
if (obj instanceof MyClass mine &&
 mine.getMyValue() != 0) { ... }
```

## Car Dealership Design – Third Pass

- We add **UsedCar** to the design for this chapter.



- Dealership** now holds a second array **usedCars** to collect a few instances of this class along with its new cars in **cars**.
- Dealership.listCars** must iterate over this array now, too.
- Dealership.findCars** (both overloads) need to search both arrays. To help with this you will implement a new helper function **getAllCars** that builds a single array from the two fields.
- Seller** still takes generic **Cars** in its constructor, but uses **instanceof** to check if it's selling a used car. If so, it avoids selling at any price less than **pricePaid**.

## Listing and Selling Used Cars

**LAB 7A**

**Suggested time: 45 minutes**

In this lab you will implement the next iteration of functionality on the car dealership, as diagrammed on the previous page. First you will add used cars to the **Dealership**'s listings by iterating over the **usedCars** array, and flagging each such listing as "USED." You'll create the helper method **getAllCars** and revise the **findCar** methods to use that. Then you'll add code to **Seller** to exercise caution when selling a used car so as not to discount below the price paid for the car.

Detailed instructions are found at the end of the chapter.



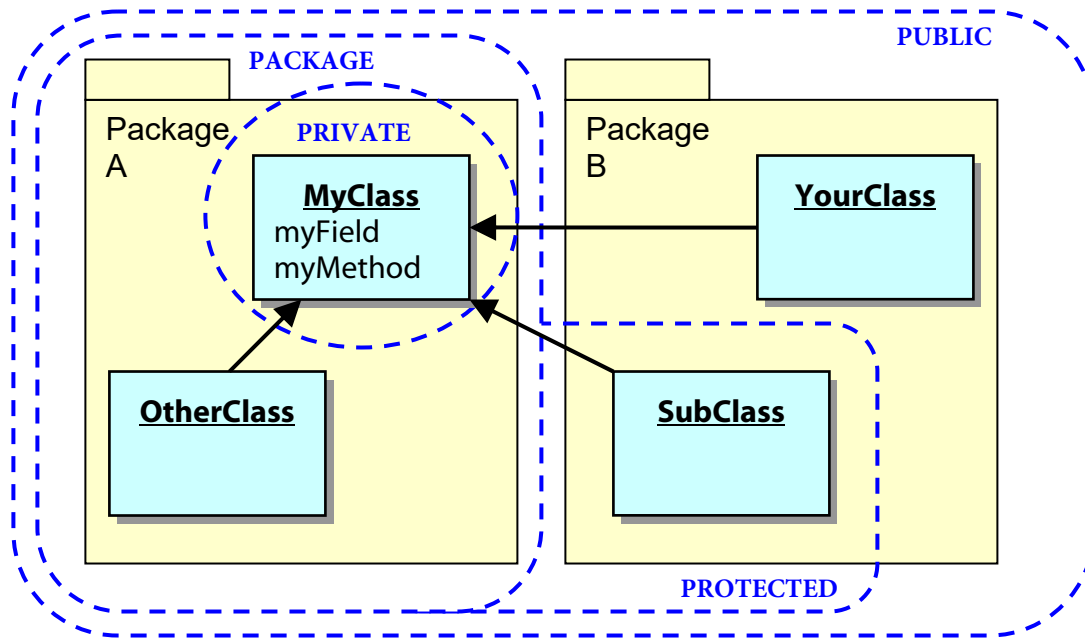
## Protected Visibility

---

- Now we can reconsider the **protected** modifier with a full understanding of its implications.
- This is a common means by which a class can share information or features with any derived classes.
  - A protected field can be read and written directly by a subclass.
  - A protected method can be invoked by a subclass, and might operate on a private field on the subclass' behalf.
- Unlike most OO languages, Java defines **protected** to imply package visibility as well.
  - This has not had the best reception in OO circles; to many purists it seems overly broad.

## Protected Visibility

- The following diagram illustrates the four visibility options for a class member:



- Only the class itself can see its private members.
- Any class in the package can see members with default visibility.
- Subclasses and package members can see protected fields and methods.
- Public members are available to anyone.
- Note, finally, that classes themselves can be either public or package-visible.
  - Naturally, if one can't see the class, one can't see its members – even if they are public.

## Polymorphism

---

- Object-oriented software offers a quite potent capability called **polymorphism**.

- Polymorphism means, loosely, “many (possible) shapes.”
- Consider one piece of code calling a method on an object through an object reference.

```
public void myMethod (Base thing)
{
 thing.doSomethingBasic ();
}
```

- We know that there may be different types for the object reference and the object itself: the object reference may be to a base type, and the object may be of a derived type.

```
myMethod (new Derived ());
```

- Polymorphism is achieved when behavior is different for different runtime object types, as called through the same object-reference types in the client code.
- Polymorphism can be achieved in a number of ways, but the most direct is through specialization.
  - Base-type methods may be **overridden** by new implementations provided by derived classes.
  - The calling code only knows the semantics of the base type, not the type of the actual object handling the call.
  - This allows the derived class to offer different functionality without requiring any changes to the client code.

## Overriding Methods

---

- The derived class can override methods as defined by the base class.
  - It simply provides its own implementations of those methods, using signatures identical to those used in the base class.
  - The derived-class implementations will be called in place of the base class implementations – for objects of the derived type only.
- This is not to be confused with **overloading** methods.
  - Overloading is a syntactic device by which two methods can share the same name using different parameter lists.
  - Overriding means providing a different implementation for a base-class method, without changing the method signature.
- One can mix and match the techniques.
  - You can add an overload of a method in a derived class.
  - If a base class defines multiple overloads of a method, you can override any number of them in a derived class.
- It is critical that the override have **exactly** the same signature as the base-class method.
  - If not, you will wind up **overloading** by mistake! and your method will not be called when you want it to.
  - The compiler has no way of knowing which technique you mean to use and so will not warn you.

## Rules for Overriding

---

- **Generally speaking, the full method declaration must be preserved in an override of the method.**
  - If name and parameter list don't match, it's not an override.
  - It is illegal to reduce the visibility of the method through overriding, though one may make, say, a private method public.
  - Return type must match, or be a subtype of the base-class method's return type.
  - The override cannot add any exceptions to the declaration.
- **In all of the above ways, the compiler is drawing the line at any change that could trip up a caller using the base-class reference.**
  - If the caller expects a method to be available, a derived class can't make it unavailable.
  - A derived type of the expected return type is safe; in fact the base-class method might be returning an object of a type derived from its own published return type.

## The @Override Annotation

---

- As mentioned earlier, it's all too easy to overload a method when you intended to override it.
- This can be a particularly frustrating mistake, because:
  - The compiler can't help you.
  - At runtime, your method probably won't be called when you expected it to – a very quiet failure that will leave you pulling out your hair and instrumenting your code with “print statements” to confirm what code is executing and what isn't.
- The compiler supports a standard annotation **@Override**.

### @Override

```
public void onSomeEvent (SomeEvent ev) { ... }
```

- **@Override** can only be applied to methods.
- It advises the compiler of your intent to override a base-class method; the compiler responds by treating any failure to observe the overriding rules as an **error**, instead of falling back to a method overload.
- This can save a lot of time and trouble and is strongly recommended.

## Referencing the Superclass

---

- Code in a derived class can refer to the superclass using the keyword **super**.
  - We’ve seen this for invoking a superclass constructor.
  - It can also be used to clarify which implementation of a method to invoke.
- If a derived class overrides a given method, and wants only to add something to the base class behavior, call the base class’ implementation using **super** as an object reference.

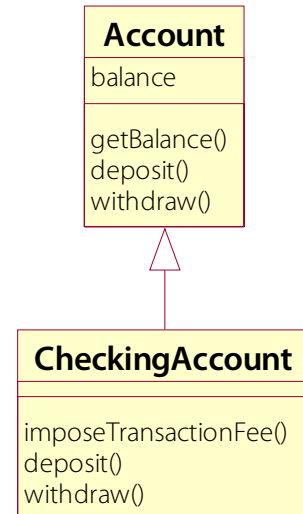
```
@Override
public void deposit (double amount)
{
 imposeTransactionFee ();
 super.deposit (amount);
}
```

- The call to the base-class method can occur at any point in the derived-class code – it could be called first, last, in the middle, or even multiple times if that’s the point of the override.
- You can call a different base-class method than the one you’re overriding, too.
- Note that without the use of **super** you will get infinite recursion!
- This is a surprisingly common coding mistake, and of course the compiler can’t possibly catch it.

## Bank Accounts

### EXAMPLE

- In **Bank** there are two types of bank accounts, one extending the other.
  - **Account** does the basic math of deposits and withdrawals.
  - **CheckingAccount** specializes **Account** to set a minimum balance and to impose a fee per transaction if that minimum is not maintained.



```
protected void imposeTransactionFee
()
{
 if (getBalance () < MINIMUM_BALANCE)
 super.withdraw (TRANSACTION_FEE);
}
```

```
@Override
public void deposit (double amount)
{
 imposeTransactionFee ();
 super.deposit (amount);
}
```

```
@Override
public void withdraw (double amount)
{
 super.withdraw (amount);
 imposeTransactionFee ();
}
```

```
private static final double MINIMUM_BALANCE = 100.0;
private static double TRANSACTION_FEE = 1.50;
```



## Bank Accounts

**EXAMPLE**

- The **Test** application class creates one account of each type and passes it through a simple **exercise** method:

```
public static void exercise (Account account)
{
 account.deposit (50);
 System.out.format
 ("After deposit, balance is $%,1.2f%n",
 account.getBalance ());
 account.withdraw (50);
 System.out.format
 ("After withdrawal, balance is $%,1.2f%n",
 account.getBalance ());
}
...
Account account = new Account (90);
exercise (account);

account = new CheckingAccount
 (account.getBalance ());
exercise (account);
```

- The accounts behave differently as called from the **exercise** method – because their runtime types are different.

```
Creating new account with balance of $90.00
After deposit, balance is $140.00
After withdrawal, balance is $90.00
```

```
Creating a new checking account on
 old account balance.
After deposit, balance is $138.50
After withdrawal, balance is $87.00
```

## The All-Important

**LAB 7B**

**Suggested time: 30 minutes**

In this lab you will implement an override of the **testDrive** method for **UsedCar**. You will use the base-class implementation, but then modify the results to take the **condition** of the car into account. You will test this out on various cars and see polymorphism in action between **Car** and **UsedCar** instances.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- Java enables specialization using class extension.
- Extending classes get all the fields and methods of their base classes “for free.”
- The compile-time type of an object reference and the runtime type of the object itself can be two different types.
  - The compiler will allow implicit conversion from a derived-type reference (or new object) to any base type, but will require an explicit type cast to go the other way.
  - If different, the runtime object type must always be a subtype of the object reference type.
  - Otherwise a **ClassCastException** will be thrown when the reference is assigned.
- Java type identification via the **instanceof** operator can be used to protect typecasts against unexpected runtime types.
- Java enables polymorphism by letting methods be overridden.
  - The caller of a method, through an object reference, does not know the runtime type of the object, and needn’t know.
  - The appropriate implementation of a called method will be found based on the runtime object type.

## Listing and Selling Used Cars

**LAB 7A**

In this lab you will implement the next iteration of functionality on the car dealership. First you will add used cars to the **Dealership**'s listings by iterating over the **usedCars** array, and flagging each such listing as "USED." You'll create the helper method **getAllCars** and revise the **findCar** methods to use that. Then you'll add code to **Seller** to exercise caution when selling a used car so as not to discount below the price paid for the car.

**Lab project:** Cars/Array/Step7

**Answer project(s):** Cars/Array/Step8

**Files:** \* to be created  
src/cc/cars/Dealership.java  
src/cc/cars/Application.java  
src/cc/cars/Seller.java

### Instructions:

1. Open **Dealership.java** and note the new **usedCars** array. (You may want to take a look at **Persistence**, too, and see the used cars that are primed into this array.)
2. Add the used cars to the application's list output by adding code to **listCars**. You can copy the existing code that loops through **cars**, paste a clone of it just below, and then modify that loop to work through **usedCars**.
3. Also, when listing used cars, add " -- USED" to the formatting string, before the %n.
4. Build and test and you should see the full twenty-car listing:

```
run list
ED9876: 2015 Toyota Prius (Silver)
PV9228: 2015 Subaru Outback (Green)
...
KM0962: 2004 Saab 9000 (Silver) -- USED
HL4735: 2003 Saturn Ion (Plum) - USED
```

5. If you search for one of the used cars, though, you won't find it:

```
run price RP5191
Car not found.
run price Geo Metro
Car not found.
```

**Listing and Selling Used Cars****LAB 7A**

6. Add a new **protected** method to the **Dealership** class called **getAllCars**. It should take no parameters and return a **Car** array.
7. Implement the method by creating a new array of **Car** references called **result**. Initialize this to an array of **new Car** references whose length is the sum of the lengths of **cars** and **usedCars**.
8. Initialize a local integer **r** to zero.
9. Now loop through each array, one after the other. In each loop, for each element, copy the reference into **result[r]** and bump the value of **r**. (So that **r** will equal **c**, throughout the first loop, but will begin the next loop at **cars.length** while **u** is still zero.) A useful trick here: the iterator expression in a **for** loop can take a comma separator between multiple expressions, as in
 

```
for (int c = 0; c < cars.length; ++c, ++r)
```
10. Return **result**.
11. Change both **findCar** methods to loop over the result of **getAllCars**, instead of **cars**.
12. Build and test, and you should be able to find any of the ten cars, of either type.

**run price RP5191**

RP5191: 1978 Renault Le Car (Yellow) \$1,499.99

**run price Geo Metro**

RG0504: 1991 Geo Metro (Midnight) \$1,098.99

13. Open **Application.java** and enhance the “price” command handler to flag a car as used in its listing. We get this for free from **dealership.listCars**, but we’re on our own with a single car. How can we find out if the returned **Car** is a used car or not? Use the **instanceof** operator to check, and thus provide one more string to be formatted, which will either be “ -- USED” or the empty string:

```
System.out.format ("%s: %s - $%,1.2f%s%n",
 found.getVIN (), found.getShortName (),
 found.getStickerPrice (),
 found instanceof UsedCar ? " -- USED" : "");
```

14. Build and test to see the new flag showing up in the output:

**run price RP5191**

RP5191: 1978 Renault Le Car (Yellow) \$1,499.99 -- USED

**run price Geo Metro**

RG0504: 1991 Geo Metro (Midnight) \$1,098.99 -- USED

**Listing and Selling Used Cars****LAB 7A**

15. Open **Seller.java** and make sure it won't sell at a price that undercuts what was paid for the car. First, in **getAskingPrice**, check that the **result** isn't below the price paid before returning it. Since you don't know at the moment whether this is a used car or not, you must test **instanceof** prior to calling **getPricePaid**, and if it isn't a used car in the first place, just let the **result** stand.

```
double result = car.getStickerPrice () * (1.0 - initialDiscount);
if (car instanceof UsedCar used && used.getPricePaid () > result)
 result = ((UsedCar) car).getPricePaid ();

return result;
```

16. Add a similar test to **willSellAt**, although here you don't need to modify a value but just return false if either the offer is lower than the best-discounted price (already done) or lower than the price paid if it's a used car.
17. Build and test. You should find that when you try to buy the Renault Le Car, you can't buy it for \$1300, even though this is higher than the old lowest price of 15% below sticker, because the dealership paid \$1350 for it. Hard to move those Le Cars these days.

```
run buy RP5191
"I can sell the car for $1,350.00."
1300
"Nope -- can't do it."
1350
"Sold!"
```

# The All-Important

**LAB 7B**

In this lab you will implement an override of the **testDrive** method for **UsedCar**. You will use the base-class implementation, but then modify the results to take the **condition** of the car into account. You will test this out on various cars and see polymorphism in action between **Car** and **UsedCar** instances.

**Lab project:** Cars/**Array/Step8**

**Answer project(s):** Cars/**Array/Step9**

**Files:** \* to be created  
**src/cc/cars/UsedCar.java**

## Instructions:

1. Build the starter code and do a quick test drive of the Subaru Outback, Ford Pinto and the El Camino:

```
run drive PV9228
Feels powerful ... Handling is good.
Seems like this car is worth $31,198.79.
```

```
run drive AR7993
Feels a bit weak. Handling is fair.
Seems like this car is worth $0.89.
```

```
run drive WQ0227 (third character is a zero)
Feels powerful ... Handling is fair.
Seems like this car is worth $2,308.89.
```

2. Open **UsedCar.java** and create an override of the **testDrive** method. Remember to use the **@Override** annotation – this isn't mandatory but it's a very good idea.
3. Start your implementation by calling the superclass method and storing the results in a local variable:

```
TestDriveResults results = super.testDrive ();
```

**The All-Important****LAB 7B**

4. Now check the **condition** of the car against each of the enumerated values **excellent**, **good**, **fair** and **poor**. Adjust the feedback and value accordingly:
  - If **EXCELLENT**, call **results.addFeedback** to add a string saying something good. Pump up the perceived value by 25%, using the accessor and mutator methods.
  - If **GOOD**, just add some nice feedback but don't change the value.
  - If **FAIR**, say that it "shows its age" and deduct 25% from the value.
  - If **POOR**, say so and deduct 50% from the value.
  - Otherwise, it's a wreck! In this case call **results.setFeedback** to say that it's a wreck – this won't append your string but overwrite whatever was there before, because it no longer matters! Set the perceived value to zero.
5. Return the **results** object.
6. Build and re-test, and you should get different results now for your used cars. The new cars are unaffected, naturally.

**run drive PV9228**

Feels powerful ... Handling is good.  
Seems like this car is worth \$31,198.79.

**run drive AR7993**

It's a wreck! Couldn't get it to move.  
Seems like this car is worth \$0.00.

**run drive WQ0227**

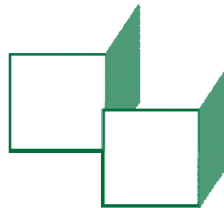
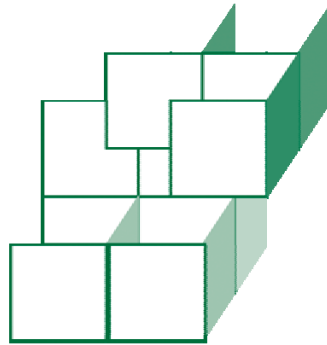
Feels powerful ... Handling is fair. Seems to be in great shape.  
Seems like this car is worth \$2,886.11.





## CHAPTER 8

# USING CLASSES EFFECTIVELY



## OBJECTIVES

*After completing “Using Classes Effectively,” you will be able to:*

- Use **static** fields and methods effectively in Java designs and class implementations.
- Implement static initializer blocks that run when a class is first loaded into the JVM.
- Import static fields defined by other classes for simpler, direct use in your code.
- Prohibit further inheritance of a class.
- Understand the costs of object creation in Java, and take appropriate steps to minimize object creation:
  - Use **factory** classes to control creation.
  - Use pools of immutable objects where appropriate.
  - Use **StringBuffer**s and **StringBuilder**s instead of **Strings** where a high volume of string operations will occur.
- Use enumerated types effectively, including stateful and behavioral enumerations.

## Class Loading

---

- In this chapter we'll consider some of the finer points of using classes in Java.
- There may be many objects of a class in memory, and these are created when requested explicitly by code – that is whenever **new** is used to build a new object.
- What about the class itself?
- The class must map into memory at some point:
  - To provide the bytecode for various methods
  - To define the shape of objects in memory based on the fields
- We're also about to discover that the class itself can have state! So that must be held in memory, too.
- Classes are loaded into the JVM at first reference.
  - The application class is identified first, for instance as a command-line argument to the **java** launcher.
  - Then, as code in **main** references other classes, these classes are loaded, and so on through their running code.
- An entity called the **class loader** is responsible for this function.
  - The class loader is a pluggable feature, and different implementations can exist.
  - The standard one in a Java SE JVM is the **system class loader**.

## Static Fields

---

- The modifier **static** can be applied to any class member – field or method.
- A static field is a state element that is represented only once for the class, not once per object as usual.
- Static fields can be mutable or immutable (constant).
  - Insist that a static field cannot be changed after initialization by also declaring it **final**.
- Use static fields to define values that must be shared by all objects of the class – statics are the closest thing Java has to “globals.”
  - A shared reference to an external resource or utility.

```
private static Database DB;
```

- A common operating mode or threshold value.

```
private static double MINIMUM_BALANCE;
```

- Constant values that will be checked in many places through the class' code.

```
public static final String VERSION = "1.2.1_01";
```

- As **non-static** fields (a/k/a **instance** fields) are initialized when an object is created, static fields are initialized when the class is loaded.

## Static Methods

---

- Static methods are a little trickier to grasp.

```
public class MyClass
{
 public static String convert (String starter) ...
}
```

- A static method is a method that must be called without reference to an object of the defining class.
  - When invoked from outside the class scope, it is called not on a particular object, but on the class itself:  
`MyClass.convert ("Starting value");`
  - It has no **this** reference – even when called from a non-static method on the same class that does have a **this** reference.
  - This is actually helpful when no object reference of the appropriate type is handy.
- Use static methods to define functionality that relates to the class, and not to an object of the class.
  - Utility methods that perform some common chore
  - The **main** method on any application class
  - Methods that operate on static fields
- A class might consist entirely of static members; in UML this is known as a **class utility**.
  - Such a type is never instantiated; what would be the point?

## Crossing the Boundary

---

- Instance methods can use both static and instance fields.
- Static methods must not attempt to interact with instance fields, however.
  - They are invoked with no implicit object reference – no **this** – and so cannot evaluate references to instance fields.
  - They can use static fields and call other static methods.
- Consider the **Bank** example from the previous chapter.
  - **CheckingAccount** defines a minimum balance and a transaction fee:

```
private static final double MINIMUM_BALANCE = 100;
private static double TRANSACTION_FEE = 1.50;
```

- The instance method `imposeTransactionFee` reads both:

```
protected void imposeTransactionFee ()
{
 if (getBalance () < MINIMUM_BALANCE)
 super.withdraw (TRANSACTION_FEE);
}
```

- If we mistakenly declared this method **static**, it would still be able to read these two values.
- However, it would fail to understand either **getBalance** (which is non-static because every account balance is different) or **super.withdraw**.

## Fixed Discounts

### EXAMPLE

- In **Cars/Array/Step10**, the **Seller** class has been enhanced to define default values for initial and best discounts.

```
private static final double
 DEFAULT_INITIAL_DISCOUNT = .10;
private static final double
 DEFAULT_BEST_DISCOUNT = .20;
```

- It is still possible to set these for a specific object, but there is also a simpler constructor that uses the defaults.

```
public Seller (Car car)
{
 this (car, DEFAULT_INITIAL_DISCOUNT,
 DEFAULT_BEST_DISCOUNT) ;
}
```

- Thus the non-static fields **initialDiscount** and **bestDiscount** can still vary by instance.

## Static Initializer Blocks

---

- What if the initialization of a static field is too complicated to provide right in its definition?
  - Perhaps an algorithm must be run to generate a value.
  - One might require a series of statements to prime an array.
- For an instance field, one just uses the constructor.
- For static fields, initialization in the constructor will occur every time a new object is created.
  - This is inefficient at the very least.
  - It may also produce unwanted results, if the static field has changed over time and a new instance resets it.
- Java defines a simple but peculiar syntax to address this need – the **static initializer block**.

```
static
{
 x = someComplexComputation ();
 myArray = new int[4][]; ...
}
```

- This block is placed as an ordinary class member.
- Any code in such a block is executed just once, when the class is loaded.
- It is like a cross between a static method – it is static and can't touch instance fields – and a constructor – it's anonymous and invoked implicitly at “class creation time.”



## Static Imports

---

- You can import a static member of a class, so that you can identify that member by its simple name, instead of having to qualify it with package and class.
- This is known as a **static import**, and it's distinguished from ordinary imports by use of the keyword **static**.
- This is a great option when your class makes heavy use of constants defined by another class.
- As with ordinary imports you can identify a single symbol or use a wildcard – and the wildcard is more common in static imports:

```
import static com.me.MyClass.SOME_CONSTANT;
import static com.me.MyClass.*;
```

```
...
```

```
public void foo (int x)
{
 if (x == SOME_CONSTANT) { ... }

 return SOME_OTHER_CONSTANT;
}
```

## Calendar Constants

### EXAMPLE

- In **Seasons** is a simple application that produces a string based on the month in which it is run, using **static finals** defined in the class **java.util.Calendar**. See **src/WhatShouldIWear.java**:

```
import java.time.LocalDate;
import java.time.Month;
import static java.time.Month.*;

public class WhatShouldIWear
{
 public static void main (String[] args)
 {
 Month month = LocalDate.now ().getMonth ();

 String what = "";
 switch (month)
 {
 case JANUARY:
 case FEBRUARY:
 what = "Black and brown, perhaps white " +
 "accessories for skiing.";
 break;
 ...
 }
 ...
 if (month == JULY || month == AUGUST)
 System.out.println
 ("And don't forget the sunscreen!");
 }
}
```

- Note that the **case** expressions would work even without the static import: they are evaluated with the switch expression **month** in scope.

## The Object Class

---

- Earlier it was stated that a Java class can have at most one superclass.
- In fact, every Java class, except one, has exactly one superclass.
- If you don't explicitly state that your class extends another, then your class is automatically a subclass of **java.lang.Object**!
- This class defines a few methods that are so key to object behavior that all classes must provide them:
  - **toString** produces a string representation of the object state.
  - **equals** can be used to test for equivalence to another object – we recognize this from using it on **Strings**, and it turns out it's more generic than that, although **String** implements it specifically for its own comparison algorithm.
  - **hashCode** allows objects to live in **hashed** collections – we'll consider the Collections API in a later chapter.
  - **getClass** can report the runtime type of the object.
  - **wait** and **notify** are used in multi-threading.
- In upcoming versions of the car dealership, we'll use **getClass** to clarify the runtime type of a seller, which will be able to vary over several new classes:

```
System.out.println ("Seller class: " +
 seller.getClass ().getName ());
```

## Prohibiting Inheritance

---

- The keyword **final** in Java is used in a few related but ultimately disparate ways.
  - We’ve seen it used to make a static field immutable.
  - It can be used for local variables in a method as well, with the same effect – the variable can’t be changed once initialized:

```
public void layout ()
{
 final int margin = 5;
 int x = margin;
 for (int n = 0; n < stuff.length; ++n)
 {
 stuff[n].setPosition (x);
 x += stuff[n].getWidth () + margin;
 }
}
```

- It can also make a **method** immutable – that is, a **final** method cannot be overridden in any derived classes.

```
private final double criticalBitOfCode (int x);
```

- Finally (sorry), it can be used to assert that an entire class should not be extended at all:

```
public final class LastOneInHierarchy
```

## Costs of Object Creation

---

- Objects are wonderful things, but they're relatively expensive to create, to maintain, and to destroy.
  - When **new** is invoked, memory must be allocated.
  - The constructor must be invoked; in fact a sequence of constructors running from **Object** down to the type being created must be executed.
  - Polymorphism comes at a price, too: a **virtual pointer** from the object to a **virtual table** of code pointers must be managed to assure that the right implementation of a method is invoked when called.
  - Various behind-the-scenes management of the object must be set up, including reference counting for garbage collection and a lock for multi-threaded use.
- All this to say, it's certainly fine to use objects, but it's also worth keeping an eye out for situations in which objects are being created and destroyed very frequently or in great quantities.
- We'll consider first a very common cause of unnecessary object creation – our unassuming friend the **String** class!
- Then we'll look at some general techniques to control the amount and frequency of object creation.

## The Secret Lives of Strings

---

- Strings enjoy a special relationship with the compiler.
- The compiler will automatically create a **String** object to wrap a string literal encountered in code.

– That means that this line of code:

```
String x = "Hello";
```

– ... is functionally equivalent to this line of code:

```
String x = new String ("Hello");
```

- **String** is the only class for which the compiler will do this.
- Note also that **Strings** are **immutable**.
  - For various rather detailed reasons of performance, Java strings implement a **copy-on-write** strategy that allows them to be shared easily without using a lot of memory.
  - The upshot is that once a **String** is created, it cannot change its value.
  - When you think you're changing a string, you're actually creating a new one! So this line of code:

```
x = x + ", Java!";
```

– ... is functionally equivalent to this:

```
x = new StringBuffer (x).append ("", Java!")
.toString ();
```

## Performance of Strings

---

- These features are very convenient for most coding.
- However there is a hidden performance penalty, because new objects are being created that you wouldn't necessarily guess at from the code.
  - A new string is created when you assign a literal.
  - Two new objects are created when you append a string to another: a **StringBuffer** and a new **String** with the concatenated value.
  - There can also be costs of re-allocating buffers in memory to accommodate the longer strings; re-allocating memory is roughly as expensive as object creation.
- This is no big deal in most cases.
- Problems arise when ordinary **String**-based algorithms are invoked at high volume.
- A classic example is a loop that builds a string progressively from values parsed from some outside source, such as another string, an input stream, a file, or a database.
  - For each new value, the loop incurs the cost of a new **StringBuffer**, possible memory re-allocation, and a new **String**.
  - It also drops a lot of dead objects on the garbage collector in quick succession, and that has to get cleaned up sometime.

## DNA

### EXAMPLE

- Consider **DNA/Step/Step 2**, which resumes the progression from an earlier lab.

- The last passage of code “reconstitutes” a readable RNA string from a packed byte array:

```
String reconstituted = "";
int totalLength = 0;
for (byte b : compressed)
 for (int r = 0;
 r < 8 && totalLength < codons.length ();
 r += 2, ++totalLength)
 reconstituted += map[(b & (3 << r)) >> r];
```

- This is a pretty good example of the worst thing you can do!
  - This code will create at least  $2r$  objects, and will transfer  $r(r-1)/2$  bytes.
  - Performance is **quadratic** – which is a fancy way of saying it will scale terribly as the string gets longer.



## DNA

### EXAMPLE

- In **DNA/Step 3**, the application has been modified to do some crude profiling.
  - There is now a static method **test** that does what **main** used to do, except that instead of reading the command-line or setting a default value for **codons**, it synthesizes a test string of a given length.
  - **main** now invokes **test** multiple times, asking for longer and longer strings.
  - The string lengths grow linearly, from 24000 to 120000.
  - **test** no longer prints out the strings; instead, it times itself and prints a report of how long it took to build the reconstituted string.

```
long start = System.currentTimeMillis ();
for (byte b : compressed)
 for (int r = 0; ...)
 reconstituted += map[...];
System.out.println ("String of length " +
 codons.length () + " took " +
 (System.currentTimeMillis () - start) +
 " milliseconds");
```

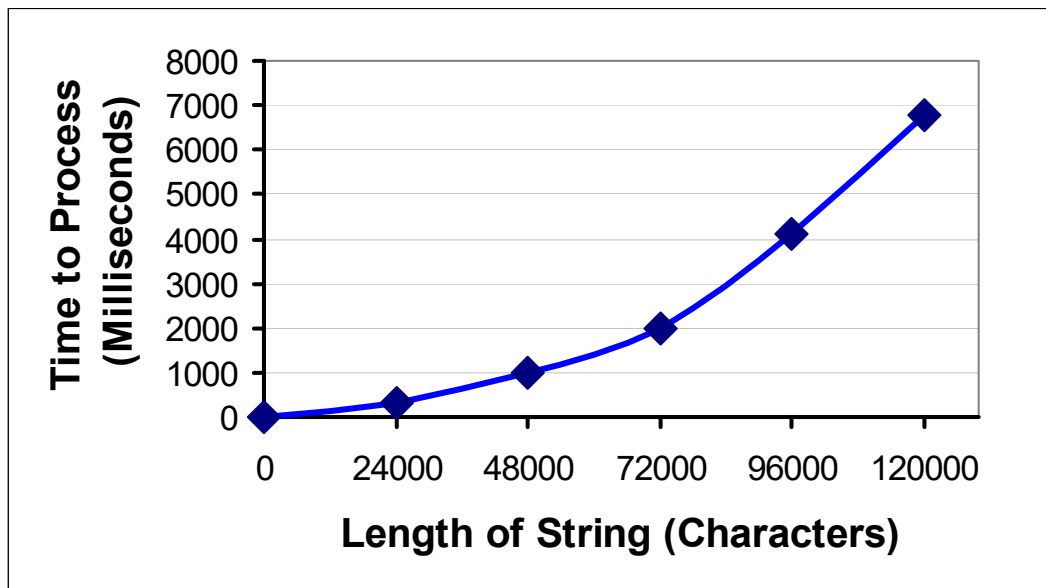
## DNA

### EXAMPLE

- Build and test, and you'll see how performance degrades:

```
String of length 24000 took 297 milliseconds
String of length 48000 took 1015 milliseconds
String of length 72000 took 1986 milliseconds
String of length 96000 took 4126 milliseconds
String of length 120000 took 6767 milliseconds
```

- We'd like to see something close to **linear** performance metrics for something like this; instead, for twice the characters, it takes about four times as long – that's **quadratic** performance.



## StringBuffer and StringBuilder

---

- The key to better performance when manipulating character strings in Java is to take advantage of the built-in **StringBuffer** and **StringBuilder** classes.
- **StringBuffer** is the class the compiler is invoking implicitly when you do string arithmetic with the **String** class.
  - You can use it explicitly yourself, and in the process take control of object creation and destruction.
  - Rather than creating a new buffer, appending, translating to string and throwing it away, each time, you can create a buffer once, append many times, and translate once.
- **StringBuilder** is nearly identical in functionality, but trades thread safety for even better performance than **StringBuffer**.
  - You can declare a local variable of type **StringBuilder**, and take advantage of its better performance, because that local variable cannot be shared between executing threads.
  - An **instance field**, **static field**, or a **method parameter** of this type should be given a second look, as it might be shared between threads and would not work correctly. It's usually better to use **StringBuffer** in those cases.
- Create a **StringBuilder** explicitly, based on a starting string value, or a starting capacity:

```
StringBuilder builder1 = new StringBuilder ();
StringBuilder builder2 = new StringBuilder ("One");
StringBuilder builder3 = new StringBuilder (4096);
```

- Default capacity is 16 characters.

## Using StringBuilders

---

- **Manipulate strings by calling methods on the builder:**
  - **append** is the most heavily used method, and is overloaded to accept all the primitive types, plus strings, string buffers and builders, **Objects**, and arrays of characters.
  - **insert** is similarly overloaded and allows insertion at any point in the existing string.
  - **delete** and **deleteCharAt** are the main ways of removing content
- **Most methods return the string builder object itself, which is convenient when several operations must be performed in sequence.**
  - Calls can be chained without having to restate the builder object reference.
  - One common construct is to chain calls to **append** – this is roughly equivalent to chaining values using the **+** operator when working with **Strings**:

```
builder.append ("The value is ")
 .append (number)
 .append (".");
```

- **When the heavy lifting is done and you need to carry a normal string forward into other parts of the application, simply call **toString** on the builder.**

```
String result = builder.toString ();
operateOnAString (result);
```

## Tuning Up DNA

**EXAMPLE**

- In **DNA/Step 4**, the application has been modified in a few simple ways (profiling code not shown):

```
StringBuilder reconstituted =
 new StringBuilder (codons.length ());
int totalLength = 0;
for (byte b : compressed)
 for (int r = 0;
 r < 8 && totalLength < codons.length ();
 r += 2, ++totalLength)
 reconstituted.append
 (map[(b & (3 << r)) >> r]);

//System.out.println (reconstituted.toString ());
```

- The accumulator is now a **StringBuilder**, not a **String**.
  - We call **append** instead of using **+=**.
  - We would produce the string using **reconstituted.toString**.
- **Build and test, and check the performance ...**

**run**

```
String of length 24000 took 0 milliseconds
String of length 48000 took 0 milliseconds
String of length 72000 took 0 milliseconds
String of length 96000 took 0 milliseconds
String of length 120000 took 0 milliseconds
```

- How's that?
- If you were to profile it, you'd actually still see a slight hitch in the creation of the test strings, because that code too uses progressive concatenation, where it should use a builder.

## Working with Strings

---

- The **String** class itself does offer some handy methods.
- There are **substring**, **startsWith**, **endsWith**, **charAt**, and lots of other “string arithmetic” functions.
  - See the API docs for all the details.
- Less obvious is the **split** method, which “tokenizes” the called string into an array of substrings.

```
String[] words = "one fish two fish".split (" ");
```

- You provide a **delimiter** string, and **split** recognizes each occurrence of your string as a boundary.
- The delimiter will be excluded from all results.
- Note that the delimiter is a **regular expression** (“regex”).
- Regex usage is beyond our scope; but for example notice that the following two calls will get different results, because the + character is seen as a modifier that means one or more occurrences of the previous character, so it will split on any number of commas:

```
"a,,b,c".split (","); // yields "a", "", "b", "c"
"a,,b,c".split (",+"); // yields "a", "b", "c"
```

- Also, beware of trying to use regex special characters such as a dot in your delimiter. It’s okay to do this, but you must escape such characters – using a **double-slash** because both Java and the regex evaluator see the reverse-slash as an escape character:

```
"one+two+three".split ("\\+");
```

## Working with Strings

---

- **String** also offers the **replace** and **replaceAll** functions, which can be quite powerful.

```
String new = old.replace ("Sam", "Fred");
String new = old.replaceAll ("\\.txt", "");
```

- Why two methods? They seem to do the same thing, right?
- The naming isn't as clear as it could be, but ...
  - **replace** assumes that the first parameter is an **exact string**, and matches it verbatim.
  - **replaceAll** assumes that the first parameter is a **regular expression**, and matches it accordingly.
  - This means that the second parameter can include **replacement groups**, as well – although this and further study of regexes is beyond the scope of this course.
- Just remember which of the two methods you're calling!

## Controlling Object Creation

---

- Beyond the special case of managing strings, there are a few general strategies and patterns to bear in mind when confronted with potentially large volumes of objects, or with frequent creation and destruction.
- The **factory** pattern defines a separate factory class that is responsible for creating objects of a given type.
  - This pattern has many advantages, but for our purposes here it's helpful because it can **hide** the process of object creation.
  - Using a factory, the client code will call a factory method such as **newInstance**, instead of using **new** directly.
  - You can't write code for **new**! (Actually, in some languages, you can, but not in Java.)
  - You can write **newInstance** yourself, though.
- Typically you will also make the use of **new** impossible, by defining all constructors as non-public or perhaps by defining an **abstract type**.
  - We'll consider abstract types in the next chapter.
- Then your factory method can act as if it's creating new objects each time it's called.
  - But in reality it might be reusing objects from a fixed-size **pool**.
  - Such objects can be recycled, avoiding both a great volume of objects in memory and frequent creation and destruction.



## Understanding Enumerated Types

---

- Thus far we've used enumerated types as simple sets of unique names.
- In fact, the members of these sets are not simple values; they are objects.
- An **enum** is actually a specialized sort of class – and another interesting exercise in controlling object creation.
- The compiler translates an **enum** definition into a class of the same name that defines a set of static final members, whose names are those given in the enum definition.
  - This class closely follows a design pattern known as the **flyweight**, in which objects are desired but there can only be a finite set of possible object states.
  - Thus it is worthwhile to limit the number of possible instances to one for each possible value.
  - A flyweight hides its constructor, and rather than provide a factory method it simply offers a full set of public, pre-constructed instances, ready to use.
  - All callers share these instances – there is only one **Color.red** object, which is fine because it's immutable anyway.

## Disassembling Color

### EXAMPLE

- It may be illuminating to see a disassembly of an enumerated type.

- Consider the simple Color enum defined

**DataTypes/src/DataTypes.java:**

```
enum Color { red, green, blue, black, white };
```

- Build the project, and then set the class path to the build directory:

```
compile
set classpath=build
```

- Run the **javap** tool – this is a crude disassembler that was used to produce the bytecode listing for **Hello.main**, back in Chapter 1.

**javap Color**

Compiled from "DataTypes.java"

```
final class Color extends java.lang.Enum<Color>{
 public static final Color red;
 public static final Color green;
 public static final Color blue;
 public static final Color black;
 public static final Color white;
 public static final Color[] values();
 public static Color valueOf(java.lang.String);
 static {};
```

```
}
```

## Disassembling Color

### EXAMPLE

- Aha! There are many of the symbols we've been using as if they were native language features:
  - **Color.red**, **.green**, etc. – notice that these are instances of the **Color** class itself
  - The **values** array
  - The **valueOf** method that converts a string to an instance
- We've seen a few other methods – **name**, **ordinal**, etc.
- Where are they?
- Notice that **Color** is found to extend the class **Enum**.
- Let's disassemble that!

```
javap java.lang.Enum
```

```
Compiled from "Enum.java"
```

```
public abstract class java.lang.Enum extends
java.lang.Object implements
java.lang.Comparable, java.io.Serializable {
 public final java.lang.String name();
 public final int ordinal();
 public java.lang.String toString();
 public final boolean equals(java.lang.Object);
 public final int hashCode();
 protected final java.lang.Object clone()
 throws java.lang.CloneNotSupportedException;
 public final int compareTo(java.lang.Enum);
 public final java.lang.Class
 getDeclaringClass();
 ...
}
```

## Stateful and Behavioral Enumerations

---

- Seeing that an **enum** is actually a specialized **class** might get your creative juices flowing ...
- Could an enumerated type define more than just an ordered list of values?
- Yes, and in this way Java enumerations are surprisingly powerful.
  - Enums can have **state**, beyond a single ordinal value.
  - They can define **behavior** – methods, just like any class.
- We now consider the full grammar of **enum** definitions, of which our uses so far have just been the simplest case:

```
enum Name
{
 value1[(initializers)],
 value2[(initializers)], ...
 valueN[(initializers)] [;

 constructor! fields! methods!
}
```

- Only the **enum** declaration itself, and the value list at the beginning, are treated differently by the compiler.
  - From there out, ordinary member definitions will be compiled as usual.
  - Let's look at some of the possibilities this opens up.

## Car Handling

### EXAMPLE

- In **Cars/Array/Step10**, the **Car** class has expanded its definition of the **Handling** enum to simplify its method code.
- The old **testDrive** method had to switch over the possible values and update the feedback string and perceived-value factor accordingly:

```
feedback += "Handling is " + handling + ".";
if (handling.equals (Handling.excellent))
 factor += .2;
else if (handling.equals (Handling.good))
 factor += .1;
else if (handling.equals (Handling.poor))
 factor -= .1;
```

- This is not very object-oriented!
- If enums are classes, couldn't those classes encapsulate things like how to update the factor?

## Car Handling

**EXAMPLE**

- The new version defines **Handling** to do just that:

```
public enum Handling
{
 EXCELLENT(.2), GOOD(.1), FAIR(0), POOR(-.1);

 private Handling (double adjustment)
 {
 this.adjustment = adjustment;
 }

 public double getAdjustment ()
 {
 return adjustment;
 }

 private double adjustment;
};
```

- The compiler extrapolates this to a full-fledged flyweight class.
- Now **testDrive** can rely on **getAdjustment** instead of mapping enum member to value factor.

```
feedback += "Handling is " +
 handling.toString().toLowerCase() + ".";
factor += handling.getAdjustment ();
```

- And so can anyone else, which is the point of encapsulation: we get reuse of a mapping that's inherent to the **Handling** type in the first place.

## Condition as a Behavioral Enumeration

**LAB 8**

**Suggested time: 30 minutes**

In this lab you will enhance the **UsedCar.Condition** enumeration. Where **Car.Handling** is stateful, **UsedCar.Condition** will go beyond this to define useful behavior right in the enum. This will further simplify the **testDrive** implementation for **UsedCar**.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- Class loading is managed by the JVM, but the behavior is pluggable.
  - Different class loaders can be provided to handle different persistent formats for class bytecode – such as JARs or object databases – and different means of reading the source – such as network connections to HTTP and FTP resources.
- Static fields capture state for the class itself, not for individual objects of the class.
- Static methods are invoked without reference to any object, and can only act on static fields.
- Instance methods can act on both static and instance fields.
- Static initializers are like constructors for the class; they are invoked at class-loading time.
- Object creation and maintenance are relatively expensive in the JVM.
- **StringBuilders** offer an important escape from the performance problems related to object creation when using strings.
- For general purposes, object creation and maintenance in memory can be controlled using the **factory pattern** and managed pools of objects.



## Condition as a Behavioral Enumeration

**LAB 8**

In this lab you will enhance the **UsedCar.Condition** enumeration. Where **Car.Handling** is stateful, **UsedCar.Condition** will go beyond this to define useful behavior right in the enum. This will further simplify the **testDrive** implementation for **UsedCar**.

**Lab project:** Cars/Array/Step10

**Answer project(s):** Cars/Array/Step11

**Files:** \* to be created  
src/cc/cars/UsedCar.java  
src/cc/cars/Persistence.java

### Instructions:

1. Open **UsedCar.java** and review the **testDrive** method. Notice the long chain of conditionals that adjusts feedback and value based on the **condition** field:

```
if (condition.equals (Condition.EXCELLENT))
{
 results.addFeedback ("Seems to be in great shape.");
 results.setPerceivedValue (results.getPerceivedValue () * 1.25);
}
else if (condition.equals (Condition.GOOD))
...

```

2. Again, not a very object-oriented solution! Almost always, a chain like this, or a **switch-case** construct, is a sign that something could be more cleanly encapsulated in classes. The code here would have to be replicated anywhere else one wanted to deal with conditions, feedback, and perceived value. In this application, **testDrive** may be the sole, ideal encapsulation of all of this logic, but as a general design principle it makes best sense to keep the behavior as close to the state as possible.
3. So we'll migrate this to the **enum** itself. This will require more than just a read-only property, as in **Car.Handling**, because there are different behaviors for different conditions – more than can be parameterized in numbers or strings. (Specifically, for some conditions we concatenate to the feedback string, but for some we replace it.)

## Condition as a Behavioral Enumeration

**LAB 8**

4. Start by opening up some space to work in the **Condition** enum, and add a trailing semicolon to the value list:

```
enum Condition
{
 EXCELLENT,
 GOOD,
 FAIR,
 POOR,
 WRECK;
}
```

5. Give the class (yes! that's what it is) two private fields: a **double multiplier** and a **String feedback**.
6. Add a constructor that initializes both fields based on matching parameters.
7. Create a method **adjustFeedback** that takes a **TestDriveResults** object as its parameter. Have it adjust the perceived value by **multiplier**, and then either add **feedback** to its feedback string or, if the condition is **WRECK**, replace it completely with our own **feedback**. **TestDriveResults** has methods **setFeedback** and **addFeedback**, just for these purposes.

How do you test to see if the condition is **WRECK**? Remember, you're in a method on an ordinary class that has only a limited set of instances, each of which is defined as a static member ... Simply test if **this == WRECK**. Enums are unusual in many ways, and one of them is that testing for object identity really is the same as testing for equivalence. The **equals** method would work, too.

8. Finally, enhance the initializer list to give each enumerated value the correct state. You can get the appropriate values from the current **testDrive** code, or just refer to the following:

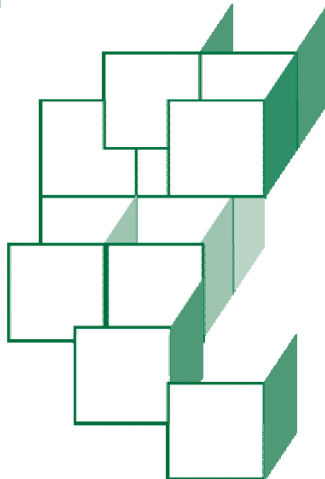
```
public enum Condition
{
 EXCELLENT (1.25, "Seems to be in great shape."),
 GOOD (1.0, "It's been well kept."),
 FAIR (.75, "Shows its age."),
 POOR (.5, "What a mess!"),
 WRECK (0.0, "It's a wreck! Couldn't get it to move.");
}
```

9. Now, you can dramatically simplify the code in **testDrive**: remove the whole set of **if-else-if** tests, and instead just call **condition.adjustFeedback**, passing the **results** object.
10. Rebuild and retest, and you should find that the application behavior is unchanged. But we now have a more robust OO solution to the problem, and better reusability.



## CHAPTER 9

# INTERFACES AND ABSTRACT CLASSES



## OBJECTIVES

*After completing “Interfaces and Abstract Classes,” you will be able to:*

- Design systems using abstract types.
- Define interfaces as definitions of behavior without implementation, and implement those interfaces in separate classes.
- Create abstract classes to provide partial implementations that then must be specialized to be used at runtime.

## Interface vs. Implementation

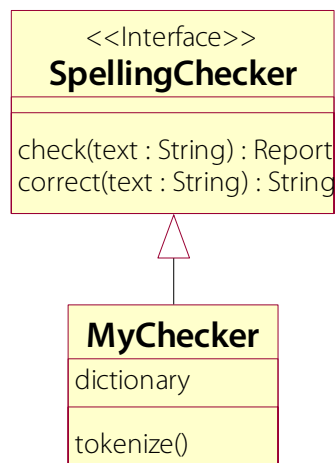
---

- For many reasons in many contexts, it's generally good practice to think of an encapsulation in terms of both **interface** and **implementation**.
- The public interface of a class defines how the rest of the world interacts with that class.
  - Any class in a UML design that requires **collaboration** with another class must call methods on that class' public interface.
- The implementation of the class is best hidden behind that interface.
  - We've discussed the benefits of **data hiding**.
  - Other implementation choices are also prone to misuse by the outside world, if exposed publicly: things like how data is derived, dependencies on certain files or databases, dependencies on other classes, and so on.
- Any class separates interface from implementation, just by the nature of the Java language.
  - The implementation code for even a public method can't be manipulated or used by a caller, in any way other than calling the method by its defined signature – which is the interface.
- However for some purposes it's useful to go a step farther and separate the interface and implementation as logical entities, as types, and even as objects in memory.

## UML «Interface» and Realization

---

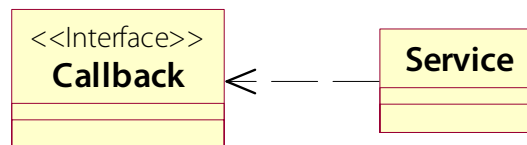
- UML recognizes this general OO principle by offering the «Interface» stereotype.
- An encapsulation marked in this way is understood to carry no implementation at all.
  - It exists solely for the purpose of specifying a public interface – to something.
  - The implementation must exist somewhere, of course, and typically there will be one or more classes that provide implementation for a given UML interface.
  - Although from the perspective of type information an implementation may seem to **specialize** an interface, it doesn't exactly; the thinking is that an implementation could only specialize another implementation.
  - So we say an implementation **realizes** an interface – makes it **concrete**, while the interface is called an **abstract** type.



## UML «Interface» and Realization

---

- Use interfaces in OO designs to capture common elements of **semantics** but not common behavior.
- For most Java SE software, this feels a lot like specialization, and is usually motivated by the same sorts of concerns.
  - It's a special case in which only method signatures are truly common between the would-be specializations.
  - It's also a perfect technique for defining semantics that you have no intention of implementing – such as the **outbound** semantics of a callback interface for a tool or component.
  - Your design in this case says, “When certain things happen, I’ll contact you in the following ways. You’re responsible for whatever is supposed to happen then.”



- In designs for distributed systems, the interface is often available remotely, while the implementation exists only on a server.
  - RMI and EJB systems insist on separate interfaces, and use these to generate both server- and client-side code for remote method invocation, object pooling, security, and transaction control.
  - Web services map Java interfaces to interoperable service descriptions.

## The Java Interface

---

- Java allows for the definition of behavior without corresponding implementations in a construct called an **interface**.
- Interfaces are like classes but have no implementation code, and everything is public.
- Interfaces can contain only the following members:
  - Methods with no implementations, known as **abstract** methods.
  - Static, final fields; these are typically definitions of standard values like strings and enumerations.
- The **public** and **abstract** modifiers for methods are optional; no other choices are legal.
  - A good practice is to state **public** but leave out **abstract**:

```
public interface SpellingChecker
{
 public Report check (String text);
 public String correct (String text);
 public static final int MAX_LENGTH = 4096;
}
```

- This makes it easy to reuse the method signature verbatim in an implementing class.
- Interfaces are **abstract** types as a whole – this means that they cannot be instantiated, as classes can.

```
Runnable thread = new Runnable (); // ILLEGAL
```



## Implementing Interfaces

---

- A class can realize an interface type by using the **implements** keyword – similar to **extending** a class:

```
public class MyChecker
 implements SpellingChecker
```

- The class must then provide implementations of the methods defined on the interface.
  - This is done as if one were overriding a base-class method: restate the method signature and provide code.

```
public class MyChecker
 implements SpellingChecker
{
 public Report check (String text)
 { ... }

 public String correct (String text)
 { ... }
}
```

- Do not use the **@Override** annotation, though; technically this isn't an override, since there must first be a base-class implementation before a derived class can override it.
  - Also there isn't the same concern about the compiler failing to catch any mistakes in the method signature, because you must accurately declare all your implementations, or the compiler will raise an error saying your class is not concrete.
  - Also, there is no useful meaning for **super** in this context.

## Single Implementation Inheritance

---

- Java supports a **single-inheritance** model, in which each class has one and only one superclass.
  - You cannot **extend** more than one class, by any syntax.
  - If you do not supply a base class, your class implicitly extends **java.lang.Object**. This is the only class that has no superclass and is known in some circles as the **root inheritable**.
- This means that you can only inherit one **implementation**.
- However, a class can implement multiple interfaces.

```
public class BeAllAndEndAll
 extends Mediocre
 implements Wonderful, Sublime, Impressive
```

- Interfaces can **extend** other interfaces, as well.
  - In UML we'd call this one interface specializing another.
  - So UML specialization always maps to **extends** in Java, and UML realization always maps to **implements**.
- Any methods defined on multiple base interfaces will simply be merged by the compiler.
  - There is no question of “which one to use?” because the compiler can decide if they're equivalent semantically.
  - If they are, they're really one method; and if not, they're overloads!

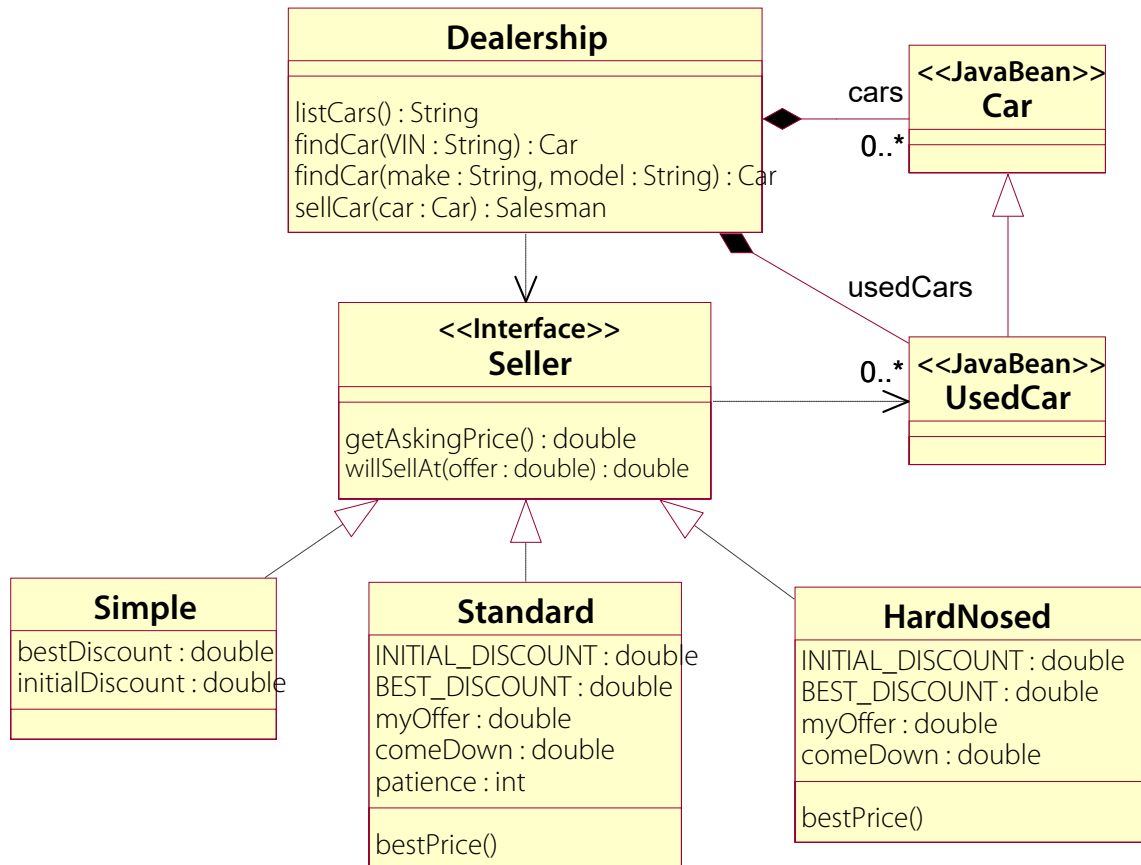
## Interfaces in the Core API

### EXAMPLE

- The interface-implementation split occurs frequently in the Java Core API.
- We have yet to delve into much of this API so far.
  - We've used **String** and **StringBuffer**, the various number classes such as **Integer** and **Double**, and **System**, all from the **java.lang** package.
  - We've used utilities from **java.util** including **Math**.
- We'll start seeing interfaces a lot over the rest of the course:
  - The Collections API defines interfaces such as **Iterator** and **Collection**, which are implemented differently for linked lists, vectors, stacks, maps, and so on.
  - Java's threading model is expressed in an interface **Runnable** and an implementing class **Thread**.
  - Console and file input and output, and network communications, are implemented using **streams**, which are abstracted to interfaces such as **DataInput** and **DataOutput**.
  - The Serialization API uses an interface **Serializable** to mark types which are able to participate in serialization and deserialization.

## Car Dealership Design – Fourth Pass

- We'll refine our ideas about how to sell cars in this next iteration of the car dealership.



- Seller** is now an interface, and the basic implementation that was in this class is now in **Simple**.
- Additional types **Standard** and **HardNosed** implement different strategies for selling cars.
- All implementations live in a new package **cc.cars.sales**.
- The **Dealership** will hand out different sellers for different situations – although we'll only get as far as randomly selecting a seller for each sale in our implementation.

## Simple Seller

**DEMO**

- In **Cars/Array/Step12** there is a version of the car dealership – similar to what we’ve seen in an earlier lab, plus a slight change to one of the method signatures for selling cars.
  - We will work through a demonstration of how to re-factor an existing class – **Seller** – into an interface with a separate implementation class **Simple**.
    - The completed demo is found in **Cars/Array/Step13**.
    - In the upcoming lab you will add new implementations of the **Seller** interface: **Standard** and **HardNosed**.
  - The following instructions will only make sense if you work in a simple editor and make changes from the command line.
    - IDEs offer code-refactoring features that will simplify this process.
    - Try it outside of the IDE first, and then you may want to experiment with IDE refactoring features from there.
1. Create the new directory **src/cc/cars/sales**.
  2. Copy **src/cc/cars/Seller.java** to **src/cc/cars/sales/Simple.java**.
  3. Open **Seller.java** and change it from a class to an interface:  

```
public interface Seller
```
  4. Remove both constructors.

## Simple Seller

**DEMO**

5. Remove the implementations of the methods and add a trailing semicolon to each method signature:

```
public double getAskingPrice () ;
{
 double result = car.getStickerPrice () *
 (1.0 - initialDiscount);
 if (car instanceof UsedCar used &&
 used.getPricePaid () > result)
 result = ((UsedCar) car).getPricePaid ();

 return result;
}
```

6. Note that **willSellAt** now returns a **double** instead of a **boolean**. The seller can now return a counter-offer; if the caller's offer is good then the counter-offer will equal that number, and otherwise it will be higher.
7. Remove the fields from the bottom of the class, so all that's left is the two method signatures:

```
public interface Seller
{
 public double getAskingPrice ();
 public double willSellAt (double offer);
}
```

8. Save that file, and open the new **Simple.java**.

## Simple Seller

**DEMO**

9. Change the package to **cc.cars.sales**, and import **cc.cars**, since you'll be using **Seller** and **Car** from that package and you're no longer working from inside it:

```
package cc.cars.sales;

import cc.cars.*;
```

10. Rename the class to **Simple**, and make it implement **Seller**:

```
public class Simple
 implements Seller
```

11. Rename the constructors, leaving the rest of the signature and code alone:

```
public Simple (Car car) { ... }
public Simple (Car car, double initialDiscount,
 double bestDiscount) { ... }
```

12. That's it! Since we copied the original code we know the method signatures will match up so our implementation is clean.

13. Change **compile.bat** or **compile** to include the new package in compilation – here's an updated **compile.bat** for example:

```
javac -d build src\cc\cars*.java
 src\cc\cars\sales*.java src\cc\tools*.java
```

14. Quick quiz: if we build right now, what will the results be?

## Simple Seller

**DEMO**

15. Try it:

**compile**

```
src\cc\cars\Dealership.java:90: cc.cars.Seller is
 abstract; cannot be instantiated
 return (new cc.cars.Seller (car));
 ^
```

1 error

16. The one thing that never moves automatically is the type designated for object creation: constructors aren't inherited, and the client code must always explicitly ask to create a specific type.

17. Open **src/cc/cars/Dealership.java** and change **sellCar** to instantiate the new **Simple** class – note that you don't need to change the method's return type:

```
public Seller sellCar (Car car)
{
 return new cc.cars.sales.Simple (car);
}
```

18. Now you can build cleanly, and the application will behave exactly as it did before. In the upcoming lab, you will implement additional **Seller** subtypes, and thus get polymorphism in the sales process from the perspective of the interactive user as customer.



## System Properties

---

- The JVM can be configured for various purposes by way of a set of name/value pairs known as **system properties**.
- These can be set from the **java** command line, using the **-D** switch.

```
java -Dprop1=value1 -Dprop2=value2 com.me.MyClass
```

- In an IDE, you set these in a run configuration.
  - Generally these are referred to as “VM arguments.” Note that they are not the same as command-line or “program arguments”.
- Then, you can both get and set system properties programmatically, via methods on the **System** class.

```
System.setProperty ("prop1", "value1");
String value2 = System.getProperty ("prop2");
String value2 =
 System.getProperty ("prop2", "default value");
```

- The one-argument overload of **getProperty** will return **null** if the property is not defined.
  - The two-argument version will return the supplied **default value**.
- System properties allow you to **externalize** any number of decisions, which means that you avoid encoding them in your Java source files.
- This is great for situations where you want to let a user or administrator make a specific choice at runtime.
- We’ll use it to control the choice of **Seller** implementation in the upcoming lab.

## The Art of the Deal

**LAB 9A**

**Suggested time: 30-60 minutes**

In this lab you will build additional implementations of the **Seller** interface, rounding out this iteration of the design with a **Standard** seller and a **HardNosed** seller. This is firstly an exercise in interface implementation, but also an opportunity to implement some more interesting application logic. These new sellers are each a bit more complex than the **Simple** one we've been using so far, and between the two of them they vary slightly. The updated **Dealership** will hand out one of the two new implementations at random.

Detailed instructions are found at the end of the chapter.

## Abstract Classes

---

- Sometimes a design calls for an encapsulation in which part of the implementation can be specified generally, but some cannot.
  - Often this common part can be encapsulated in one class, but can only be implemented by delegating to methods that must be implemented differently by derived classes.
- This mixing of implementation and specification is modeled in Java as an **abstract class**.
  - The common implementations that can be implemented on this base class are called **concrete methods**.
  - Those that are not implemented are **abstract methods**.
  - This is the same idea as in an interface, but in an interface all methods are abstract; here one typically defines some of each.
- Abstract classes cannot be instantiated, only subclassed.
- A class that defines at least one abstract method must be declared an abstract class.
  - It is also legal to declare a class abstract even if all its methods are concrete.
- You can never extend multiple classes, even if one of the would-be base classes is abstract.

## Designing with Abstract Classes

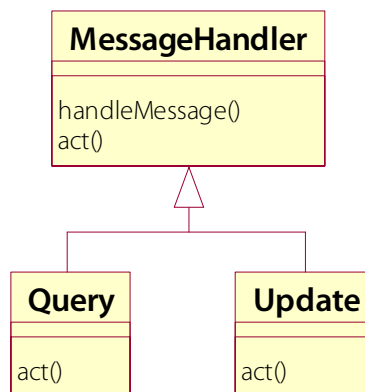
---

- Use abstract classes when you see a potential generalization whose implementation will necessarily be incomplete.
- Think of it as a form to fill out, where the blanks are the code you'd have to write for the class' methods.
  - If you can fill out the whole form, you're probably designing a **concrete class** – the type we've been working with all along in this course.
  - If you can't fill out any of it, you're looking at an **interface**.
  - If you can fill in some of the blanks, but not all of them, you're probably on your way to using an **abstract class**.
- Very often, the concrete methods on an abstract class – the blanks you can fill in – will be implemented in terms of the abstract methods.
  - You're recognizing that if a derived class can say how to do X, Y, and Z, then you can implement A, B, and C based on those methods, and do it the same way for all derived classes.
  - This saves the derived classes from having to implement A, B, and C repeatedly, and preserves a useful generalization of all the methods for public use.

## Message Cracking

### EXAMPLE

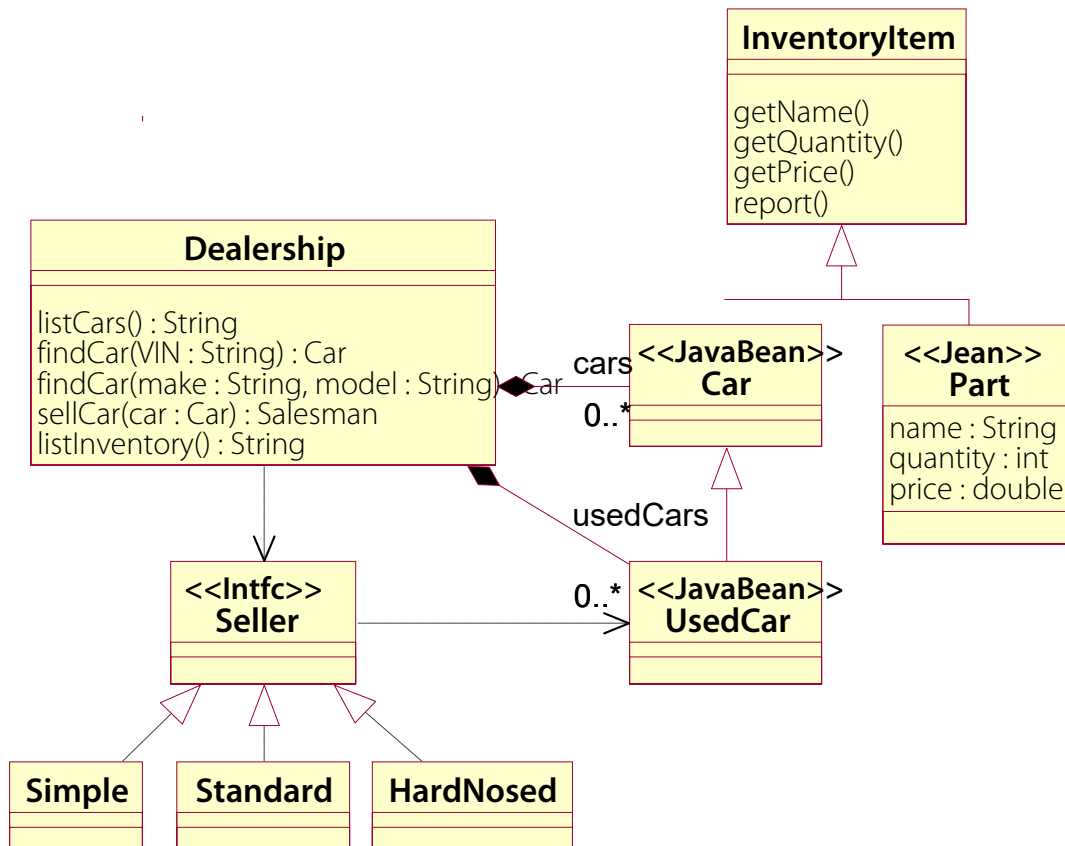
- Consider a hypothetical application that needs to process messages of a certain grammar.
  - This could be a binary format such as ASN.1, a plain-text protocol such as SMTP, or an XML format such as SOAP
- There will be different message types with different meanings, and the design will include different classes dedicated to handling them.
- But the basic behavior of parsing the message grammar and “cracking” out the interesting values is generic (more or less, depending on the grammar).
- A likely design to solve this problem would be:



- The base class is abstract; it offers the public method **handleMessage**, which knows how to parse the grammar and extract values.
- This method calls protected method **act**, which is implemented by concrete subclasses; these classes know nothing of the message grammar! which makes them potentially portable to new protocols.

## Car Dealership Design – Fifth Pass

- We now flesh out the part of the design that supports inventory listings that include different types of objects: cars (and used cars) and parts.



- The new abstract class **InventoryItem** can produce a formatted line for a report, based on name, quantity, and price.
- Each of these values is derived a little differently by the different inventory types.

## Polymorphic Inventory

**LAB 9B**

**Suggested time: 30 minutes**

In this lab you will complete the implementation of the inventory feature for the car dealership. A good deal of new code has been added for this lab: the new **InventoryItem** type is in place and partially implemented, the new **Part** class is completely implemented, and **Dealership** knows how to produce a report based on a heterogeneous collection of items. You will complete **InventoryItem** and then make **Car** an extension of it, which will mean implementing new methods on **Car**.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- **Interface-implementation split is an important technique in all varieties of Java software.**
  - Interfaces define semantics, which are all the caller should be concerned with anyway.
  - In distributed software (Java EE), interfaces as type definitions can travel from server to client, and facilitate code generation or dynamic invocation, while the implementations can live on the server side only.
- **Java treats interfaces and classes in a consolidated type model.**
  - Both classes and interfaces have a specialization mechanism, using the **extends** keyword.
  - Classes realize interfaces using the **implements** keyword.
  - Users of interfaces and classes handle object-reference types, object types, and typecasting the same way when dealing with interface or class types.
- **Abstract classes provide a middle solution between pure-interface and pure-implementation.**
  - An abstract class typically knows how to meet some, but not all, of its responsibilities as designed.
  - Concrete subtypes complete the implementation in specific ways.



# The Art of the Deal

**LAB 9A**

In this lab you will build additional implementations of the **Seller** interface, rounding out this iteration of the design with a **Standard** seller and a **HardNosed** seller. This is firstly an exercise in interface implementation, but also an opportunity to implement some more interesting application logic. These new sellers are each a bit more complex than the **Simple** one we've been using so far, and between the two of them they vary slightly. The updated **Dealership** will hand out one of the two new implementations at random.

**Lab project:** Cars/Array/Step13

**Answer project(s):** Cars/Array/Step14

**Files:** \* to be created  
src/cc/cars/sales/Standard.java \*  
src/cc/cars/sales/HardNosed.java\*  
src/cc/cars/Dealership.java

**Instructions:**

1. Copy the source file **src/cc/cars/sales/Simple.java** to **src/cc/cars/sales/Standard.java**. Change the class name to **Standard**.
2. Keep the two static final fields.
3. Get rid of the instance fields **initialDiscount** and **bestDiscount**, but keep **car**.
4. Add three new private fields: a **myOffer** and a **comeDown** of 30% – both **doubles** – and an integer **patience** that's initialized to 3.
5. Get rid of the three-argument constructor. Rename the one-argument constructor, and re-implement it to initialize **myOffer** to **car.getStickerPrice**.

| Standard                                                                                                       |
|----------------------------------------------------------------------------------------------------------------|
| INITIAL_DISCOUNT : double<br>BEST_DISCOUNT : double<br>myOffer : double<br>comeDown : double<br>patience : int |
| bestPrice()                                                                                                    |

## The Art of the Deal

## LAB 9A

Here is a summary of the standard sales strategy. The numbered steps below will walk through implementation in detail.

- Implement **getAskingPrice** based on a standard initial discount of 10%.  
(*Example: on a new car with a sticker price of \$10000 we'll ask for \$9000.*)
  - Internally, know that our best price is a 20% discount – or the price paid when selling a used car, whichever is higher. (*Best price would be \$8000.*)
  - Successive calls to **willSellAt** will remember the latest offer (starting with that asking price) . If the customer's offer is at or above the seller's latest offer, we have a deal.
  - A threshold is defined that's exactly as far below best price as the latest offer was above it. (*If best price is \$8000 and the seller has offered \$9000, this threshold would be \$7000.*) The customer has to be offering at least this much; otherwise we'll calculate a counteroffer as if they had. (*So if the customer offers \$7500, we'll work toward that, but if they offer \$6000, we'll pretend they offered \$7000.*)
  - The seller will come down on his price by 30% of the gap between the seller's last offer and this target value. (*So the response to \$7500 would be \$8700; the response to anything less than \$7000 would be \$8600.*)
  - Finally, the seller will only come down three times. After that his patience is exhausted and he will stick to his latest price until the customer buys or walks away.
6. Start by implementing a protected method **bestPrice** that returns either the sticker price minus **BEST\_DISCOUNT**, or the price paid for a used car, whichever is higher. (Hint: use the static method **Math.max** to get the higher of two numbers.)
  7. Implement **getAskingPrice** to return the 10% discount. Set **myOffer** to this value before you return it.
  8. Rip out the existing implementation of **willSellAt**. Start a new one by checking if **yourOffer** is greater than or equal to **myOffer**. If it is, return **yourOffer** to close the deal.
  9. If **patience** is zero, return the **myOffer**. If not, decrement **patience** and proceed.
  10. Find the target price as the maximum of **yourOffer** and **(2 \* bestPrice () - myOffer)** . (This latter expression is the price that's exactly as far below best price as the latest offer is above it.) Assign the higher of the two numbers to a new double called **lower**.
  11. Reduce **myOffer** by the difference between **myOffer** and **lower**, multiplied by **comeDown**. Return that value.

**The Art of the Deal****LAB 9A**

12. Modify **Dealership.sellCar** to create a **Standard** seller instead of a **Simple** one.
13. Build and test, and you should get responses as shown below, with annotations in italics explaining the behavior – note that the application now prints out the runtime class of the seller it gets back, for clarity in testing:

```

compile
run price ED9876
ED9876: 2015 Toyota Prius (Silver) $28,998.99

run buy ED9876
Seller is class cc.cars.sales.Standard
"I can sell the car for $26,099.09."
23000
"Well, I can't do that, but I could sell it for $25,169.36."
24000
"Well, I can't do that, but I could sell it for $24,818.55."
24500
"Well, I can't do that, but I could sell it for $24,722.99."
24600
"Well, I can't do that, but I could sell it for $24,722.99."
(Patience exhausted - won't move any farther.)
24700
"Well, I can't do that, but I could sell it for $24,722.99."
24725
"Sold!"

run buy ED9876
Seller is class cc.cars.sales.Standard
"I can sell the car for $26,099.09."
15000
"Well, I can't do that, but I could sell it for $24,359.15."
(Too low an offer; seller continues as if we'd offered more.)
20000
"Well, I can't do that, but I could sell it for $23,663.18."
21000
"Well, I can't do that, but I could sell it for $23,384.79."
23400
"Sold!"

```

The answer code for this class includes additional checks to assure that we never calculate or offer a price that's below the **pricePaid** for a **UsedCar**.

Notice, by the way, that you haven't had to change **Application** lately? **Dealership** creates the seller object, but it's **Application** that interacts with it through **getAskingPrice** and **willSellAt**, by representing the user's input and printing the seller's responses. That class hasn't had to change since **willSellAt** switched to returning a **double** – that's an interface design change, not about implementation. We can now plug in any valid **Seller** implementation and **Application** will work with it seamlessly.

**The Art of the Deal****LAB 9A**

14. You might want to experiment with different cars and offers, or modify some of the parameters in **Standard** to see how it behaves.
15. Now, you have multiple **Seller** implementations, but the application can only use one at a time. How best to choose? We might like to have a random selection – something like what happens when you actually walk into a car dealership to buy a car, you never know who you'll get.

But it's good to have deterministic behavior, too – for testing and diagnostics if nothing else. So we'll combine these approaches.

16. Remove the existing implementation of **sellCar**.
17. Start a new one by getting a system property:

```
String type = System.getProperty("cc.cars.Seller.type");
```

18. If **type** is **null**, choose a seller type randomly, based on the least-significant digits of the system time. Note that on many systems time is reported to a precision of 10 milliseconds, not one millisecond. So ...

```
if (type == null)
 type = (System.currentTimeMillis () / 10) % 2 == 0
 ? "Standard" : "HardNosed";
```

You could also use **java.math.Random** or **java.security.SecureRandom** – but for this purpose either of those would probably be overkill.

19. Now, based on the type name, create an object of the appropriate type:

```
switch(type)
{
 case "Simple": return new Simple(car);
 case "Standard": return new Standard(car);
}
```

20. If this falls through, just return **null** – for now. In a later chapter we'll improve the error-handling in this method, for the case in which a system property has been configured but isn't recognized.

**The Art of the Deal****LAB 9A**

21. Now you can test by setting the system property. From the command line, enter the following command or edit your **run** script so that it does so:

```
java -classpath build -Dcc.cars.Seller.type=Standard (all on one line)
cc.cars.Application buy ED9876
```

In an IDE, edit your run configuration, and be sure to put the following text in the VM arguments to:

```
-Dcc.cars.Seller.type=Standard
```

When you test with the property set one way or the other, you should now be able to control the choice of seller type; and when you leave it undefined you should get a more or less even distribution between **Simple** and **Standard** objects.

**Optional Steps – most students will prefer simply to review this code:**

22. Notice that the standard seller actually gives a better price when you lowball him. Maybe when the offer is too low, the seller should pull back a bit, and only come down maybe 5%. Implement that and test.

23. Implement a third seller type, **HardNosed**, whose strategy is similar to **Standard** with the following differences:

- Best discount is just 15%.
- Until the customer's offer is at least the seller's best price, he won't budge at all. Then he'll close the gap by just 20%.
- He does however have infinite patience!

| HardNosed                                                                                    |
|----------------------------------------------------------------------------------------------|
| INITIAL_DISCOUNT : double<br>BEST_DISCOUNT : double<br>myOffer : double<br>comeDown : double |
| bestPrice()                                                                                  |

24. Change **Dealership** to use the new class.

25. Build and test – here are expected results with **HardNosed**:

```
compile
run buy ED9876
Seller is class cc.cars.sales.HardNosed
"I can sell the car for $21,599.09."
18000
"Well, I can't do that, but I could sell it for $21,599.09."
19000
"Well, I can't do that, but I could sell it for $21,599.09."
20000
"Well, I can't do that, but I could sell it for $21,599.09."
21000
"Well, I can't do that, but I could sell it for $21,479.27."
21000
"Well, I can't do that, but I could sell it for $21,383.42."
quit
```

## Polymorphic Inventory

**LAB 9B**

In this lab you will complete the implementation of the inventory feature for the car dealership. A good deal of new code has been added for this lab: the new **InventoryItem** type is in place and partially implemented, the new **Part** class is completely implemented, and **Dealership** knows how to produce a report based on a heterogeneous collection of items. You will complete **InventoryItem** and then make **Car** an extension of it, which will mean implementing new methods on **Car**.

**Lab project:** Cars/Array/Step15

**Answer project(s):** Cars/Array/Step16

**Files:** \* to be created  
src/cc/cars/InventoryItem.java  
src/cc/cars/Part.java  
src/cc/cars/Dealership.java  
src/cc/cars/Persistence.java  
src/cc/cars/Application.java  
src/cc/cars/Car.java

### Instructions:

1. Review the starter code, which has new types and also new code in old classes. The **InventoryItem** is the base for **Part**, and the **Dealership** now keeps an array of **parts** along with **cars** and **usedCars**. **Persistence** primes six entries into this new array. **Dealership** implements a new public method **listInventory** that produces a formatted report by iterating over **parts** and getting a single line of text for each item from the method **InventoryItem.report**. (This method isn't available yet! and so the project currently won't build. That's the first thing you're going to do.) Finally, **Application** implements another command "inventory" that prints out the full report.
2. Open **InventoryItem.java** and add a concrete method **report**, taking no arguments and returning a **String**. Start by creating a **StringBuffer** called **report**, and initialize it to **getName**.
3. Append **getQuantity**, then **getPrice**, and a total value which is the product of the two. Finally, append **'\r'** and **'\n'** to end the line. (Just **'\n'** on non-Windows systems.)
4. Return **report.toString**.
5. In **Dealership.listInventory**, replace the placeholder string with a call to **item.report**.

**Polymorphic Inventory****LAB 9B**

6. Build and test the “inventory” command – you should see something like this:

**run inventory**

| Item            | Quantity | Price | Value  |
|-----------------|----------|-------|--------|
| Brake hose      | 8        | 30.49 | 243.92 |
| Rearview mirror | 1        | 54.99 | 54.99  |
| Distributor cap | 3        | 14.99 | 44.97  |
| Headlight       | 4        | 49.99 | 199.96 |
| Fuse            | 16       | 2.99  | 47.84  |
| Spark plug      | 16       | 4.99  | 79.84  |

7. Clearly, the output could stand to be cleaned up a bit. Use **String.format** to create even columns in the report. (This will also get you to a platform-neutral way of dealing with the end of the line: just use the **%n** field in the format string.) Build and test again – desired output is:

**run inventory**

| Item            | Quantity | Price | Value  |
|-----------------|----------|-------|--------|
| Brake hose      | 8        | 30.49 | 243.92 |
| Rearview mirror | 1        | 54.99 | 54.99  |
| Distributor cap | 3        | 14.99 | 44.97  |
| Headlight       | 4        | 49.99 | 199.96 |
| Fuse            | 16       | 2.99  | 47.84  |
| Spark plug      | 16       | 4.99  | 79.84  |

8. So **report** is doing the work of producing and formatting the values, and the math for the final one, but **InventoryItem** leaves it open how the name, quantity and price might be derived. **Part** provides simple wiring for this. What about **Car**?

9. Open **Car.java** and make the class extend **InventoryItem**. Try compiling right there, and what do you expect to see?

**compile**

```
src\cc\cars\Car.java:10: cc.cars.Car is not abstract and does not
override abstract method getPrice() in cc.cars.InventoryItem
public class Car
 ^
```

1 error

10. That is, it's okay to extend an abstract class, but then you must either implement the abstract methods or be an abstract class yourself. Implement the three methods for **Car**: for the name, concatenate **year**, **make** and **model**; for quantity just return one; and for price return the sticker price.

11. Now you need to get the cars into the list. In **Dealership.java**, un-comment the helper method **getFullInventory**, which works much like **getAllCars**, but combines three arrays and returns an array of **InventoryItems**.

**Polymorphic Inventory****LAB 9B**

12. Change **listInventory** to use **getFullInventory** instead of **parts**.

```
for (InventoryItem item : getFullInventory ())
```

13. Build and test, and you should see the complete inventory:

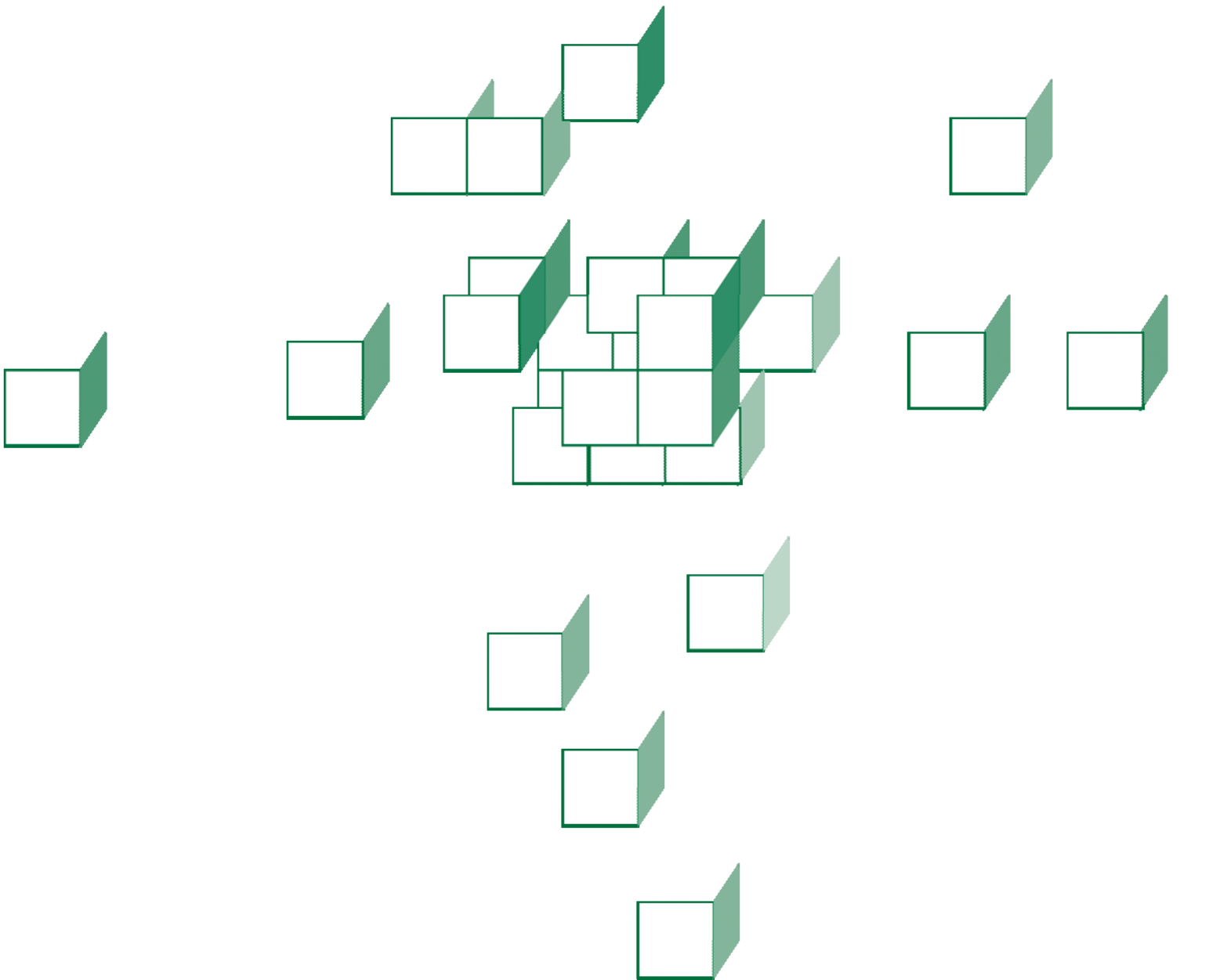
**run inventory**

| Item                      | Quantity | Price     | Value     |
|---------------------------|----------|-----------|-----------|
| -----                     | -----    | -----     | -----     |
| 2015 Toyota Prius         | 1        | 28,998.99 | 28,998.99 |
| 2015 Subaru Outback       | 1        | 25,998.99 | 25,998.99 |
| 2014 Ford Taurus          | 1        | 32,499.99 | 32,499.99 |
| 2015 Saab 9000            | 1        | 34,498.99 | 34,498.99 |
| 2014 Honda Accord         | 1        | 29,999.99 | 29,999.99 |
| 2014 Volkswagen Jetta TDI | 1        | 23,899.99 | 23,899.99 |
| 2015 Kia Sonata           | 1        | 21,999.99 | 21,999.99 |
| 2015 Ford F-150           | 1        | 28,999.99 | 28,999.99 |
| 2014 Ford Escape Hybrid   | 1        | 23,999.99 | 23,999.99 |
| 2015 Audi A3              | 1        | 39,999.99 | 39,999.99 |
| 1977 AMC Pacer            | 1        | 1,998.99  | 1,998.99  |
| 1974 Ford Pinto           | 1        | 0.99      | 0.99      |
| 1978 Renault Le Car       | 1        | 1,499.99  | 1,499.99  |
| 1991 Geo Metro            | 1        | 1,098.99  | 1,098.99  |
| 1972 Ford El Camino       | 1        | 2,098.99  | 2,098.99  |
| 2004 Toyota Prius         | 1        | 8,998.99  | 8,998.99  |
| 2004 Subaru Outback       | 1        | 6,998.99  | 6,998.99  |
| 2003 Ford Taurus          | 1        | 9,499.99  | 9,499.99  |
| 2004 Saab 9000            | 1        | 11,498.99 | 11,498.99 |
| 2003 Saturn Ion           | 1        | 4,598.99  | 4,598.99  |
| Brake hose                | 8        | 30.49     | 243.92    |
| Rearview mirror           | 1        | 54.99     | 54.99     |
| Distributor cap           | 3        | 14.99     | 44.97     |
| Headlight                 | 4        | 49.99     | 199.96    |
| Fuse                      | 16       | 2.99      | 47.84     |
| Spark plug                | 16       | 4.99      | 79.84     |



# CHAPTER 10

## GENERIC AND COLLECTIONS



## OBJECTIVES

*After completing “Generics and Collections,” you will be able to:*

- Explain the advantages of **generics** in the Java language, and use generic types in your code.
- Use the Collections API to manage dynamic collections of objects.
  - Add values to a collection, including primitives.
  - Inspect values in a collection.
- Choose between Core API implementations for lists, sets, and maps.
- Use iterators instead of directly manipulating collections, for better separation between interface and implementation.
- Take advantage of sorting and searching features in the API to optimize your own algorithms.

## Limitations of Arrays

---

- When managing series, sets, and groups of data, arrays are not always the best solution.
- Arrays capture snapshots of data fairly well.
- They do not excel where data is volatile – especially when the size of the data set can fluctuate.
  - To insert an element into an array is to slide every element above the insert point over some number of bytes in memory.
  - This assumes that the array has been allocated with extra space at the end; if it hasn't, then a new array must be allocated and all the values must be copied over to it.
- More generally, arrays expose low-level memory-management issues to the application programmer.
- When used as return values or parameters on public methods, arrays also expose something of a class' implementation strategy to the outside world.
- To wit: if one wants to provide access to a private array, one has a few awkward choices:
  - Make the array itself available through an accessor method. The array could then be manipulated in ways the class designer wouldn't like.
  - Provide an interface for iteration over the array, with a **first** method, a **next** method, etc.
  - Return a deep copy of the array, which is inefficient.

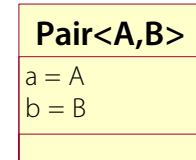
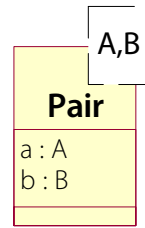
## Dynamic Collections

---

- A **dynamic collection** is a class which can hold zero, one, or many instances of some other class(es).
- By contrast to arrays, dynamic collections:
  - **Hide the data** storage away from the caller, preserving the OO principle of (you guessed it) data hiding
  - Use **more memory and processing**, because they are instances of classes, with all the usual overhead
  - **Handle insertion, deletion, and resizing better** – even though many implementations will be backed by arrays at some level
  - Can themselves be hidden behind **iterators**, which fosters general-purpose **algorithmic programming**
  - Can offer rich **object-oriented interfaces** to their underlying data, and thus can be easier for client code to use
  - Can take advantage of OO **polymorphism**, such that a client knows only the interface and the implementation can be tuned for certain performance advantages
- Java has always had dynamic collections.
- Through Java 1.4, they were only available in a **weakly-typed form**.
  - That is, the collection classes – such as **Vector**, **LinkedList**, **HashMap** – managed **Objects**, and not any specific type.
  - This was a weakness compared to strongly-typed arrays.

## Generic Types

- As of Java 5 the language supports **generic types**.
  - UML calls this a **parameterized type**: a type defined partially in terms of another type or types.
  - Formally these are represented as shown above right, with type parameters in a dashed rectangle; but many designers find it simpler to use the angle-bracket notation shown below right.
  - C++ popularized the use of parameterized types, and it is the **C++ template syntax** that generics borrow.
  - But, C++ coders, beware! These are not templates, but a more limited feature. More on this at the end of the chapter.
- Generics address a certain pattern of design challenges, in which even the method signatures must vary for different related types.
  - Does this sound familiar? Dynamic collections in Java underwent a revolution between Java 1.4 and 5.0, and all are now generic types.
  - That is, we speak not just of a “list,” but of “a list of strings;” not just a “map,” but “a map from integers to instances of **MyClass**.”



```
List<String>
Map<Integer,MyClass>
```

- For collections, this removes one of the major disadvantages as compared to using arrays.
  - The other, loss of efficiency, is a completely natural tradeoff; ultimately both arrays and dynamic collections have their places.

## Declaring Generic Classes

---

- It will be beyond our scope in this course to get into creating our own generic types.
- Still, it's worth a little while to understand the basic syntax on the declaring side, and then we'll focus on how to use generics as defined in the Core API.
- The compiler recognizes a class as a generic type by the presence of an angle-bracketed list of predicate types following the class name.
  - There must be at least one such type; if more, separate them with commas.

```
public class Things<T>
public class OtherClass<A,B,X>
```

- The predicate types are understood to be placeholders for other classes – not primitives or arrays of things.
  - The class code then uses the defined placeholder (such as **T**) in its code, and the compiler treats references to **T** simply as references to some object.
  - Only when the generic class is used in client code is **T** replaced with a “real” type such as **String** or **Car**.
- It is also possible to define a generic that can only work on subtypes of a certain other class.
  - The predicate is declared to **extend** that other class:

```
public class PostalDelivery<T extends USAddress>
```

## Using Generics

---

- When a **generic** is put to a **specific** use, the compiler can replace the predicate types throughout, and apply its usual strict type checking to everything.

```
Things<String> stringThings = new Things<String>();
```

- Say **Things<T>** defines a method **addThing** that takes a parameter of type **T**; the compiler can assure that it is called safely, and so would allow this:

```
stringThings.addThing ("Hello");
```

- ... but flunk this:

```
stringThings.addThing (new YourClass ());
```

- It works the same way with return types, but now the advantage is less in avoiding silly mistakes and more in the convenience of not having to downcast a return type.
  - If **Things<T>** also offers a method **getBestThing** that returns an instance of **T**, there is no need to downcast it to **String** when working with **Things<String>**.

```
String best = stringThings.getBestThing ();
```

- Nonetheless, **Things<T>** could be used in different code to operate on instances of **Car**, **Ellipsoid**, or any other class.

## Implicit Type Arguments

---

- As of Java 7, the compiler will try to deduce your type arguments when constructing new objects.
- This makes it possible to leave out the arguments in certain common constructs, where they're obvious:

```
List<String> myList = new ArrayList<> ();
Pair<Integer,String> myPair = new Pair<> (5, "Hi");
```

- Note that it is not the constructor arguments that determine the type arguments – though the compiler could probably be taught to use those, too!
- Rather it is the context in which the expression is being evaluated.
  - That is, the compiler is working “**outside-in**” in a way that it did not do in the earliest versions of Java.
  - Another example of outside-in evaluation is the auto-boxing feature we'll discuss later in this chapter.



## A Generic Pair of Objects

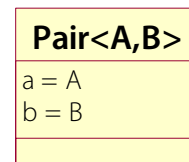
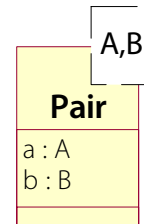
### EXAMPLE

- Though the Collections API provides much of the motivation for generic types, we'll consider some abstract generic examples on their own, before concentrating on collections for most of the chapter.
- A very simple example – in fact a primitive sort of collection class itself – is in **Generics**.
  - See the source file `src/cc/generics/Pair.java`:

```
public class Pair<A,B>
{
 public A a;
 public B b;

 protected Pair () {}

 public Pair (A a, B b)
 {
 this.a = a;
 this.b = b;
 }
}
```



- This is just a class that holds a reference to two other objects, **a** and **b**.
- But the types **A** and **B** are not known when compiling this class – there is no implication that actual classes **A** and **B** exist when **Pair<A,B>** is compiled.
- These placeholders are only replaced with specific classes when other code uses the **Pair<A,B>** class.

## A Generic Pair of Objects

### EXAMPLE

- The class **PairOfPairs** exercises this generic type:

```
public class PairOfPairs
{
 public static void main (String[] args)
 {
 Pair<String,Integer> score1 =
 new Pair<String,Integer>
 ("Will", new Integer (60));

 // This would not compile:
 //score1.a = new Integer (60);
 //score1.b = "Will";

 Pair<String,String> sPair =
 new Pair<> ("This", "That");
 sPair.b = "Will";
 }
}
```

- **score1** is a pair of string and integer.
  - The compiler will enforce this fact, replacing **A** in the generic definition of the class with **String**, and **B** with **Integer**.
  - Thus the user of the class is protected from the understandable mistake of swapping A and B when reading or writing values – an **Object**-based **Pair** would not do that.
- **sPair** is a pair of strings.
  - Where it's concerned, **b** is a **String**!
  - The two invocations of **Pair<A,B>** are as different to the compiler as **Car** and **Part**.

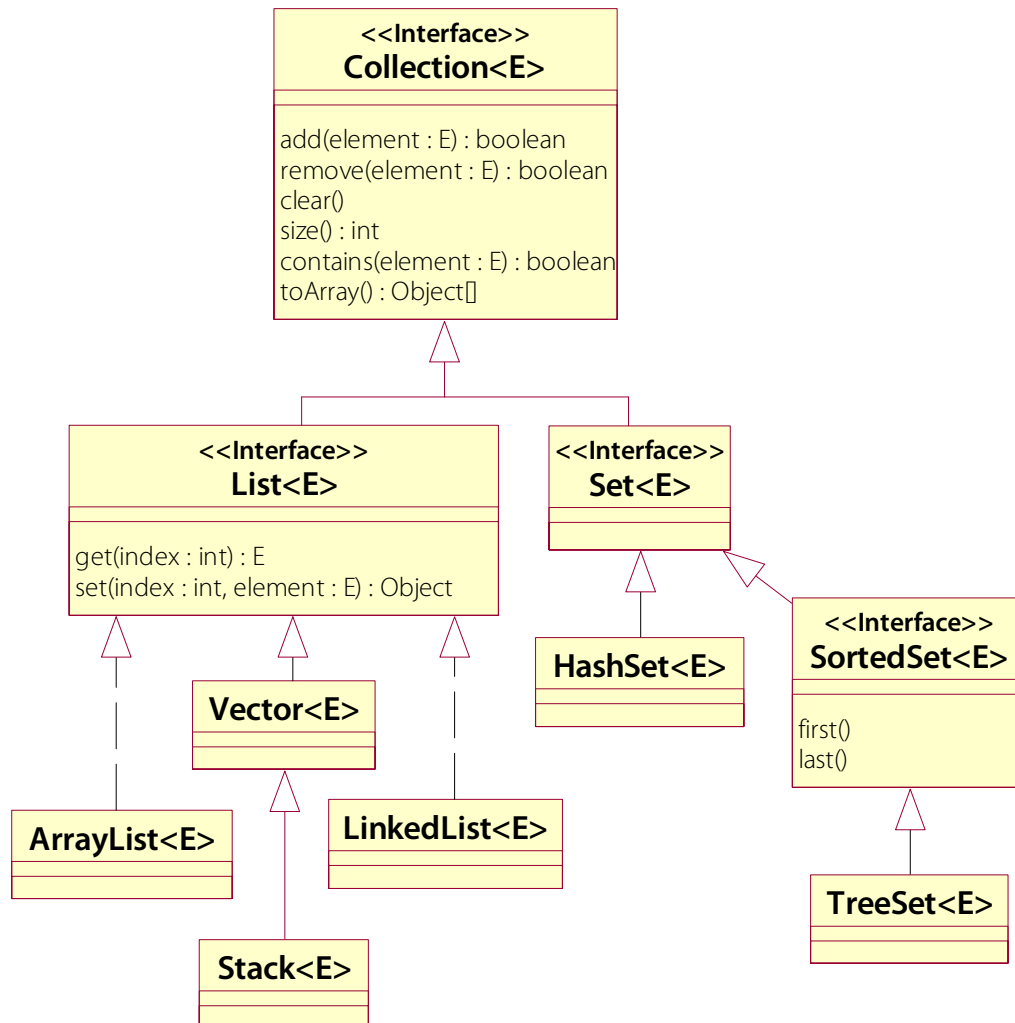
## The Collections API

---

- The Java Core API includes a comprehensive set of dynamic collection classes called the **Collections API**.
  - The whole API is implemented in package **java.util**.
- The API includes:
  - **Ordered collections** such as **ArrayList**, **LinkedList**, and **Stack**. Different implementations offer different interfaces and performance characteristics.
  - **Maps** that index **values** under **keys**, including **HashMap**.
  - Variants of the above that assure **uniqueness** of elements, including **HashSet**.
  - **Iterators** that abstract the ability to read and write the contents of a collection in loops, and isolate that ability from the underlying collection implementation.
  - Several other specializations: weak-reference collections, types meant for working with enumerated values, and so on.
- There is also a class utility called **Collections**, which offers a very useful set of algorithms.

## Collection Types

- The collection types are classified under the root interface **Collection<E>**.



- It's conventional to use a capital E as the type parameter here – indicating “Element” rather than the more general T for “Type”.
- Ordered collections implement **List<E>**.
- Collections that assure uniqueness implement **Set<E>**.
- Sorted collections implement **SortedSet<E>**.

## The Collection<E> Interface

---

- All collections implement the interface **Collection<E>**:

```
public interface Collection<E>
{
 // Partial listing of methods:
 public int size ();
 public void clear ();
 public Object[] toArray ();
 public boolean add (E element);
 public boolean remove (E element);
 public Iterator iterator ();
}
```

- Thus they can all perform certain functions:
  - Add and remove elements
  - Clear to an empty set
  - Report their size
  - Convert their data to an array of **Objects**
- Additional properties of a given collection are defined by its implementation of one of the sub-interfaces of **Collection<E>**.

## The List<E> Interface

---

- A specialization of **Collection<E>**, **List<E>** promises that the collection is ordered, and offers additional methods based on an ordinal index:

```
public interface List<E>
 extends Collection<E>
{
 public void add (int index, E element);
 public E remove (int index);
 public E get (int index);
 public void set (int index, E element);
 public ListIterator listIterator ();
}
```

- Lists are the most commonly used collection types.
  - They are also the most like arrays, which are also ordered and can be indexed, but do not offer any of the other, extended features we've discussed, such as uniqueness or sorting.
- Though the interface offers essentially random access to data, not every implementation will be efficient in that usage pattern.
  - Traditionally, though, it's the job of the implementer to anticipate what client code will do, and to provide the appropriate collection type for the anticipated usage.
  - Failing that, it is the use of interfaces such as **List<E>** that allow the implementer the flexibility to swap in different collection types as actual usage patterns and performance are observed.

## The `ArrayList<E>` Class

---

- Maybe the most popular concrete type in the API is the **`ArrayList<E>`**, which, as the name suggests, implements the **`List<E>`** over an array of **`E`**.
- **`ArrayLists`** perform best in “random access” to their elements, because the underlying arrays are good at that, too.
- Their weakness, as with arrays, is insertion and deletion.
  - However they are smart about allocating in blocks, so that only occasional resizing of the backing array is required.
- Create an **`ArrayList`** with no arguments and begin adding elements to it.
- You can also specify the initial capacity as an integer.

```
List<String> myList = new ArrayList<String> ();
List<Car> cars = new ArrayList<Car> (6);
```

- Capacity and size are not the same thing!
  - Size is the number of elements currently in the collection.
  - Capacity is the current allocation of “slots” for elements.
  - Capacity is always greater than or equal to size, and it grows and shrinks by intervals typically larger than one – ten is the default.

## The **LinkedList<E>** Class

---

- A radically different means of achieving a scalar collection is the **LinkedList<E>**.
  - Each element in a linked list is discrete in memory.
  - It holds a pointer to the next element and the previous one.
- Linked lists excel at insertion and deletion, because there is no need to shuffle any of the existing list elements when a new one is added.
  - Rather, an existing link is broken and two new ones are formed.
  - Deletion is just the opposite process.
- Iterating over a linked list is a little slower, however, and random access to an arbitrary index is **LinkedList's** real weak spot.
  - Indexing right into the backing array used by **ArrayList** is replaced by a walk from element one to element N.



## Building Collections

---

- The first thing to remember about collections is that you must create them explicitly.
  - This is true of arrays as well.
  - It is a common mistake to declare a reference to an **ArrayList** or **LinkedList** and just assume that the object is there and ready for action.
  - This doesn't work with any other classes, either! but for some reason with collections it's a much more common mistake.
- Once a collection object is created, simply add elements to it.
  - Use **add** to append the new element to the end.
  - On a list, use the **add** overload that takes an index to insert at a certain point in the collection.
  - **remove** an element by identifying it, or on a list by its index.
- Any Java object of the specific type that replaces E – or of a subtype of E – can be placed in any collection.
  - Most collections will hold objects of the same type – these are **homogenous** collections.
  - **Heterogeneous** collections are viable, though.
  - Often there's a bit of both: all elements will share a base type, but will vary over types derived from that base.

## Reading Elements

---

- Adding objects to a collection is a simple matter of calling **add** and passing the object, as in:

```
List<MyClass> myList = new LinkedList<MyClass> ();
myList.add (new MyClass (5));
```

```
List<String> stringList = new ArrayList<> ();
stringList.add ("literal");
stringList.add ("figurative");
```

```
List<Car> cars = new ArrayList<> (6);
cars.add (someCar);
cars.add (someOtherCar);
cars.add (1, someUsedCar);
```

- Notice the Java-7 usages above: the compiler can figure the expected type argument from the expression context.

- Read objects from a list using **get**.

```
String whichMeaning = stringList.get (1);
```

- **get** will return an object of type **E** – or a subtype.
- You may choose to downcast this reference to a subtype of **E** – with the same caveats as with any object reference.

```
Car someCar = cars.get (1);
if (someCar instanceof UsedCar used) {
 ...
}
```

- Through the **Collection<E>** interface, there is no such simple means of getting individual elements.

## Looping Over Collections

---

- A more common problem is to process every element in a collection, rather than drawing one arbitrary element.
- For a list, anyway, you could do it like this:

```
for (int i = 0; i < myList.size (); ++i)
 doSomethingWith (myList.get (i));
```

- Does this look familiar?
- This is very much like the “poor man’s” loop over arrays.

- For arrays there’s a simplified **for** loop; would the same technique work for collections?

- Oh, yes!

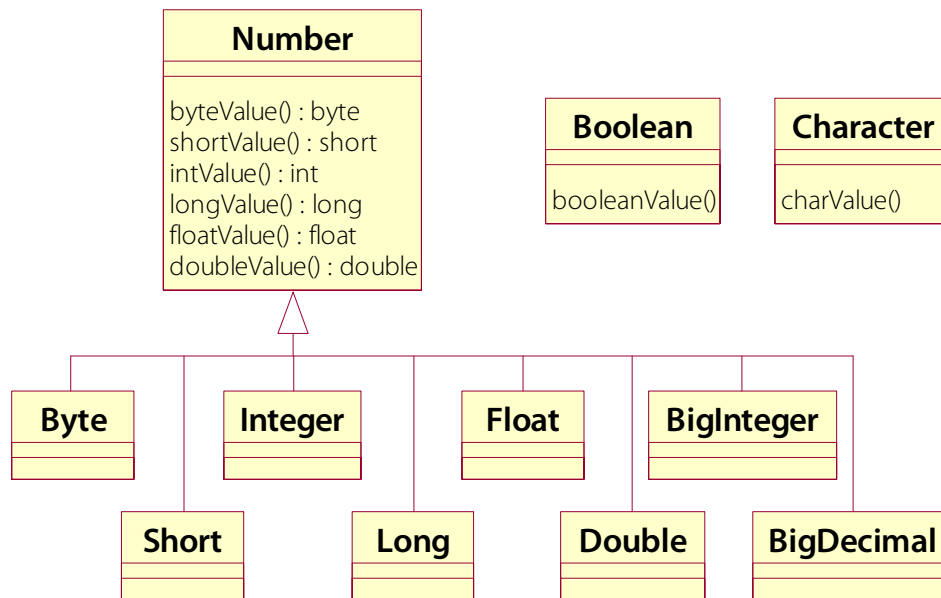
```
for (String whichMeaning : stringList)
 interpretAs (whichMeaning);
```

```
for (Car car : cars)
 if (car.getVIN ().equals (VIN))
 return car;
```

- The exact requirement is that the collection implement the interface **Iterable<E>**, which most collection types in the API do.

## Collecting Primitive Values

- Primitive values cannot be placed directly in a collection.
  - Generics only work for classes; **List<int>** is impossible.
- A library of classes is available in the **java.lang** package to wrap primitives:



- We’ve seen these in use for converting strings to numbers.
- They also serve the basic purpose of “boxing” primitive values: the primitive is held in an object of the corresponding wrapper type, and that can be placed in a collection.

```
List<Integer> intList = new ArrayList<Integer> ();
```

```
intList.add (new Integer (5));
```

```
intList.add (new Integer (4));
```

```
int y = intList.get (0).intValue ()
 + intList.get (1).intValue ();
```

## Auto-Boxing

---

- But most boxing and un-boxing will occur implicitly:
  - Promotion of a primitive to the corresponding wrapper object
  - Demotion of a wrapper back to the corresponding primitive
- For example, the code on the previous page can be simplified:

```
List<Integer> intList = new ArrayList<Integer> ();
```

```
intList.add (5);
```

```
intList.add (4);
```

```
int y = intList.get (0) + intList.get (1);
```

- Note that you still need to cast generic **Number** objects to the specific types, such as **Integer** and **Double**, before auto-boxing will “kick in.”
  - Working with a heterogeneous collection such as **ArrayList<Number>**, explicit casts are necessary:

```
List<Number> numList = new ArrayList<Number> ();
```

```
numList.add (5); // fine, clearly an int literal
```

```
numList.add (4.5); // also fine, a double
```

```
int y = ((Integer) numList).get (0);
```

```
 // cast is necessary so runtime can choose
```

```
 // the correct primitive type for conversion
```

## xxxAll Methods

---

- There are a number of convenience methods in various collections that will, say, add or remove all elements in another collection.

- For these to work on a generic type such as **List<E>**, they need to take advantage of another, stranger generics syntax:

```
public void addAll (Collection<? extends E>);
public void removeAll (Collection<? extends E>);
public boolean containsAll
 (Collection<? extends E>);
```

- The question mark is called a **wildcard**.
  - Wildcards allow limited type conversion, not between the types of elements in collections, but between the collections themselves.
  - A method like **addAll** wants to accept as an argument any collection of any type convertible to **E**.
  - Then why not a parameter type **Collection<T extends E>**?

## Convertibility of Generics

---

- This gets us into a fairly mind-bending exercise in how generics work (and don't work).
- The root issue can be expressed in a trick question: for some generic type **X** and two types **Base** and **Derived** that share an inheritance relationship: is **X<Derived>** convertible to **X<Base>**?

- Instinct immediately says, “yes, of course it is!”
- But in fact it is not – not safely.
- Consider what might happen:

```
List<Derived> dList = new LinkedList<Derived> ();
List<Base> bList = dList; // seems safe
bList.add (new Base ()); // hmm ...
dList.get (0).someDerivedMethod (); // OOPS!
```

- See the problem?
- Once the compiler allows the second line to pass, it can't prevent the fourth line from blowing up at runtime.
- So the compiler makes the second line impossible, because, at bottom, it is not safe to treat a collection of **Derived** as a collection of **Base**.

## Wildcards

---

- It would be safe, though, if we could guarantee that the base-type reference **bList** would never modify the collection.
- And we have designs that call for methods like **addAll** and **removeAll**, which want to accept collections of their own type **E** or some derived type.
- How can they be declared?

```
public void addAll (Collection<E>)
```

- ... won't accept a collection of E-derived type.

```
public void addAll (Collection<T extends E>)
```

- ... won't pass, for exactly the reason on the previous page.

- This is the motivation for the wildcard syntax.

```
public void addAll (Collection<? extends E>)
```

- This syntax strikes a deal with the compiler:
  - The compiler agrees to accept collections of **E** or collections of E-derived types – even though it makes the poor boy nervous.
  - In return, the programmer agrees not to modify the collection it finds through the object reference.
  - A method such as **addAll** only needs to read elements from the source collection, and there's no difficulty there.



## Car Dealership

### EXAMPLE

- A major revision of the car dealership has been broken out into a new tree of steps – see **Cars/List/Step1**.
- There are no design changes in this revision.
- It's all implementation strategy: the arrays in **Dealership** have been replaced with **Lists**.

– See `src/cc/cars/Dealership.java`:

```
List<Car> cars;
List<UsedCar> usedCars;
List<Part> parts;
```

– The constructor still initializes these by delegating to the **Persistence** class, and the methods there now return lists:

```
public Dealership ()
{
 Persistence persistence = new Persistence ();
 cars = persistence.loadCars ();
 usedCars = persistence.loadUsedCars ();
 parts = persistence.loadParts ();
}
```

## Car Dealership

**EXAMPLE**

- **getAllCars** combines two of the lists into a new one, and returns an iterator on that.

```
protected List<Car> getAllCars ()
{
 ArrayList<Car> result = new ArrayList<Car>
 (cars.size () + usedCars.size ());
 result.addAll (cars);
 result.addAll (usedCars);

 return Collections.unmodifiableList (result);
}
```

- The last line converts the temporary collection into an unmodifiable collection, using a utility method we'll study a little later in the chapter.
- **findCar** runs one loop over this merged collection:

```
public Car findCar (String VIN)
{
 for (Car car : getAllCars ())
 if (car.getVIN ().equals (VIN))
 return car;

 return null;
}
```

## Car Dealership

**EXAMPLE**

- **src/cc/cars/Persistence.java** has been reworked to use the dynamic collections

- Helper methods **addCar** and **addUsedCar** create and add to a given result list. Here's **addCar**:

```
private static void addCar (List<Car> result,
 String make, String model, int year,
 String VIN, String color, double price,
 int horsepower, int torque,
 Car.Handling handling)
{
 result.add (new Car (make, model, year, VIN,
 color, price, horsepower, torque, handling));
}
```

- Then the public methods such as **loadCars** can call helpers many times as needed:

```
public List<Car> loadCars()
{
 List<Car> result = new ArrayList<>();

 addCar (result, "Toyota", "Prius", 2015,
 "ED9876", "Silver", 28998.99, 220, 180,
 Car.Handling.FAIR));
 ...
 addCar (result, "Audi", "A3", 2015,
 "PU4128", "Blue", 39999.99, 280, 230,
 Car.Handling.EXCELLENT));

 return result;
}
```

- Note that there is no longer a need to call out the size of the data set ahead of time, or to use fixed indices when setting values.

## Better Management of Test Scores

**LAB 10A**

**Suggested time: 30-45 minutes**

In this lab you will overhaul the **Scores** application to use the Collections API instead of various arrays. This will be a two-step process, as you will replace the primary collection of names, scores and grades in this lab, and then take a different approach to capturing occurrence statistics in Lab 10B.

Detailed instructions are found at the end of the chapter.

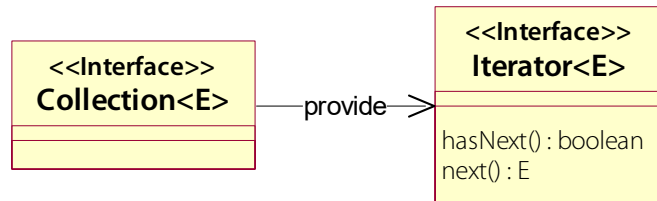
## The `Iterator<E>` Interface

---

- The `Iterator<E>` interface isolates iteration over a collection from the collection class itself.

```
interface Iterator<E>
{
 public void remove ();
 public boolean hasNext ();
 public E next ();
}
```

- An iterator can be derived from a collection using the `iterator` method.



- Then, the common looping construct is:

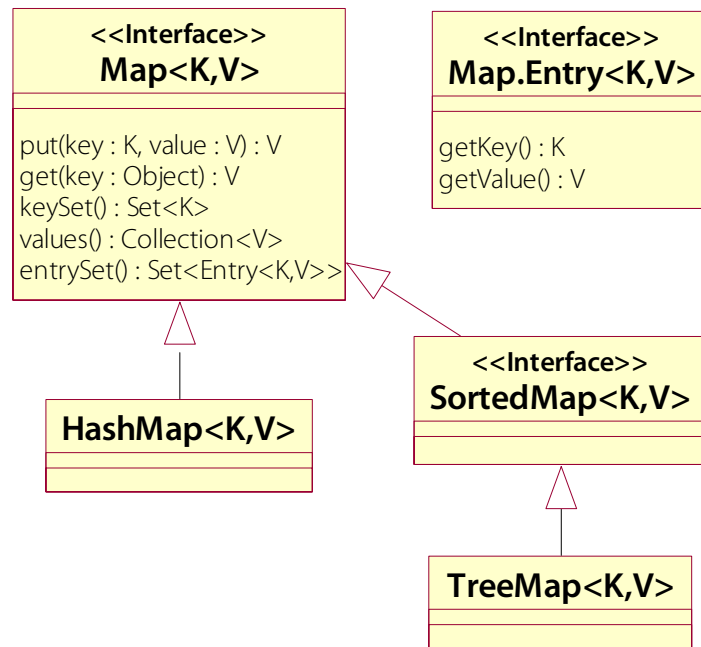
```
while (iterator.hasNext ())
 doSomethingWith (iterator.next ());
```

- Iterators offer a clean way to expose collections to the outside world without being tied to the collection type.
- An application is free to implement its own iterator class, which might do any sort of gymnastics to present a smooth stream of objects to the caller.
  - There might be multiple collections behind a single iterator.
  - Some of the data may not even be known when the iterator is requested – the iterator might use lazy evaluation, caching, etc.

## Maps

---

- Maps are not collections, per se.
- They are objects that implement the interface **Map<K,V>** and thus map **key** objects to **value** objects.



- Keys must be unique.
- There can be only one value per key, and it may not be **null**.

## The Map<K,V> Interface

---

```
interface Map
{
 public int size ();
 public void clear ();
 public V get (Object key);
 public V put (K key,V value);
 public V remove (Object key);
 public boolean containsKey (Object key);
 public boolean containsValue (Object value);
 public Set<K> keySet ();
 public Collection<V> values ();
 public Set<Map.Entry<K,V>> entrySet ();
}
```

- Maps do provide **views** of their data as collections:
  - **keySet** provides just the keys.
  - **values** provides the values.
  - **entrySet** provides a **Set** of **Map.Entry** objects, each of which is a pair of key and value. (Map.Entry is an **inner class**, a feature we'll study in a later chapter.)

## Post Offices

**EXAMPLE**

- The following code snippets exercise a **HashMap** object by storing the street addresses of US Postal Service offices in the Boston area, keyed by ZIP code:

```
Map<String,String> map = new HashMap<> ();
map.put ("02130", "655 Centre Street");
map.put ("02131", "303 Washington Street");
map.put ("02118", "34 Stuart Street");
map.put ("02116", "92 Boylston Street");
map.put ("02107", "1 Franklin Street");

System.out.println (map.get ("02116"));

// Either:
for (String ZIP : map.keySet ())
 System.out.println (ZIP + ": " + map.get (ZIP));

// Or:
for (Map.Entry<String,String> entry :
 map.entrySet ())
 System.out.println
 (entry.getKey () + ": " + entry.getValue ());
```



## Sorted Sets and Maps

---

- Collections that implement **SortedSet<E>** provide automatic insertion sorting whenever elements are added.
  - Thus the collection is always well ordered.
  - The only Core implementation is **TreeSet<E>**, which stores its elements in a binary tree.
- Sorting is based on a definition of how two objects of a given type **compare**.
  - Strings are sorted alphabetically.
  - **Number** instances are compared numerically, including type conversions to compare numbers of different sizes.
- Most other classes require some sort of comparison criterion to be defined for them.
  - This can be done on the class itself by implementing **java.lang.Comparable**, which has one method **compareTo**.
  - Or a class can be responsible for making comparisons between objects of other classes, by implementing **Comparator**.
- The **SortedMap<K,V>** interface defines similar semantics for maps.
  - The only Core implementation is **TreeMap<K,V>**.

## 50 States

### EXAMPLE

- In **SortedSet**, the fifty state names primed into an array of strings are sorted in alphabetical order.
  - Recall that this was the subject of an earlier challenge lab, and you were asked to implement various sorting algorithms by hand using only arrays.
  - This code uses the **TreeSet** class, which implements sorting with good performance over frequent inserts and deletions by storing elements in a binary tree.

```
String[] names = buildNamesArray ();
SortedSet<String> set = new TreeSet<> ();
for (String name : names)
 set.add (name);
```

- It prints the contents of the sorted set:

```
for (String name : set)
 System.out.print (name + " ");
System.out.println ();
```

- Instead of a binary search algorithm using arrays, we simply call **contains** to search for a given string value:

```
String fullName = args[0];
for (int i = 1; i < args.length; ++i)
 fullName += " " + args[i];

System.out.println (fullName +
 (set.contains (fullName)
 ? " is a state."
 : " is not a state."));
```

## Gathering Statistics in a Map

**LAB 10B**

**Suggested time: 30 minutes**

In this lab you will rebuild the latter part of the **Scores** application to get rid of the remaining two arrays, **possibleGrades** and **frequency**, and to replace them with a map.

Detailed instructions are found at the end of the chapter.

## The Collections Class Utility

---

- Since dynamic collections are so neatly polymorphic, and generic-therefore-type-safe, it is possible for the Core API to include a set of high-quality **algorithms** that operate on the collection types.
  - An algorithm isolates a generally useful behavior, such as iterating, searching, or sorting, from the underlying collection type.
  - It can also vary its behavior according to parameters: sort in a particular order, or search based on a certain criterion.
- The Collections API includes a single class utility **Collections**, with several very handy algorithms:
  - **sort** will sort an existing **List<E>**, with an overload that takes a **Comparator** for the type **E**.
  - **reverseOrder** will “invert” an existing **Comparator** or provide one that inverts the default comparator for a type.

```
Collections.sort (myList,
Collections.reverseOrder ());
```

- Several other methods will re-order an existing list: **reverse**, **shuffle**, **swap**, and **rotate**.
- **min** and **max** will get a single object from a collection, and **frequency** will tell how many times an object occurs.
- **binarySearch** will find a given object on a given list (which must already be sorted).

## Algorithms

**EXAMPLE**

- In **Algorithms**, there is an application that puts a list of integers through its paces:
  - See `src/Algorithms.java`:

```
List<Integer> numbers = new ArrayList<> ();
Collections.addAll (numbers, 1, 2, 3, 4, 5);
 // Another nice one! Using the varargs feature.
printList ("Start", numbers);
Collections.reverse (numbers);
printList ("Reverse", numbers);
Collections.shuffle (numbers);
printList ("Shuffle", numbers);
Collections.swap (numbers, 0, 4);
printList ("Swap", numbers);
System.out.format ("% -16s %d\n", "Minimum:",
 Collections.min (numbers));
System.out.format ("% -16s %d\n", "Maximum:",
 Collections.max (numbers));
Collections.sort (numbers);
printList ("Sort", numbers);
...
numbers.addAll (Collections.nCopies (3, 3));
printList ("Expanded", numbers);
System.out.format ("% -16s %d\n", "Frequency of 3:",
 Collections.frequency (numbers, 3));
Collections.replaceAll (numbers, 3, 9);
printList ("Replaced", numbers);
System.out.format ("% -16s %d\n", "Frequency of 3:",
 Collections.frequency (numbers, 3));
Collections.sort (numbers,
 Collections.reverseOrder ());
printList ("Descending", numbers);
...
System.out.format ("% -16s %d\n", "Position of 4:",
 Collections.binarySearch (numbers, 4));
```

## Algorithms

**EXAMPLE**

- Build and run the application to watch the show:

```
Start: 1 2 3 4 5
Reverse: 5 4 3 2 1
Shuffle: 1 2 5 3 4
Swap: 4 2 5 3 1
Minimum: 1
Maximum: 5
Sort: 1 2 3 4 5
Expanded: 1 2 3 4 5 5 4 3 2 1 3 3 3
Frequency of 3: 5
Replaced: 1 2 9 4 5 5 4 9 2 1 9 9 9
Frequency of 3: 0
Descending: 9 9 9 9 9 5 5 4 4 2 2 1 1
Position of 4: 7
Position of 4: -1
```

## Sorting Scores

### EXAMPLE

- Another, simple example is in **Scores/Step8**.
  - The code that sifted the original scores into a new **ArrayList** is replaced by a call to **sort** in reverse order:

```
List<Record> sorted = new ArrayList<> (scores);
Collections.sort
 (sorted, Collections.reverseOrder ());

System.out.println ("Student Score Grade");
System.out.println ("----- ----- -----");
for (Record record : sorted)
 System.out.format ("%24s %3d %1s%n",
 record.name, record.score, record.grade);
```

- For this to work there must be a way of comparing **Record** objects
  - Java doesn't know a **natural order** for these things.
  - So Record implements Comparable<Record>:

```
class Record
 implements Comparable<Record>
{
 ...
 public Record (String name, int score)
 {
 this.name = name;
 this.score = score;
 }

 public int compareTo (Record other)
 {
 return Integer.compare(score, other.score);
 }
}
```

## Conversion Utilities

---

- Also in **Collections** are several sets of methods that convert from one sort of collection to another.
- The most interesting are the **unmodifiableXXX** methods: **unmodifiableList**, **unmodifiableSet**, etc.
  - Each of these creates an immutable “view” of an existing, mutable collection.
  - This is a great utility, one that addresses a design problem that comes up almost constantly: how to expose a collection for reading without risking unwanted modification by the client code?
  - The view will throw an **UnsupportedOperationException** if any attempt is made to modify the source collection through it, or through an iterator that it hands out.
- Another set of methods **synchronizedXXX** offer views of source collections that are thread-safe.
  - It may be a rude surprise to discover that most of the basic collection types are not thread-safe already!
  - The trade-off is made for performance in single-threaded use.
  - Where safety is needed, these methods can provide it.
- Finally the **checkedXXX** methods provide views that are “dynamically type-safe.”
  - This is strictly for integrating generics with legacy code – see Appendix B on Compatibility and Migration.



## SUMMARY

- Dynamic collections can make many coding tasks much easier, and the resulting code more maintainable.
- Java Generics offer excellent type safety, and also make code more self-documenting.
- Different collection types offer different performance characteristics, and each excels at a certain pattern of usage.
- Iterators offer a layer of objects that isolate the semantics of traversing collections from the underlying implementation.
- Maps provide an easy way to build collections of name/value pairs; these are also weakly typed.
- Sorted sets and maps assure that all values or keys are kept in their natural ascending order.
- The **Collections** class utility offers a very handy set of algorithms and conversion utilities.
- In all, the **java.util** package is well named!
  - Use generic collections, iterators and algorithms liberally in your code.
  - Use arrays where collections of values or object references can be expected to stay the same once created, or where the very best possible performance is required.

## Better Management of Test Scores

**LAB 10A**

In this lab you will overhaul the **Scores** application to use the Collections API instead of various arrays. This will be a two-step process, as you will replace the primary collection of names, scores and grades in this lab, and then take a different approach to capturing occurrence statistics in Lab 10B.

**Lab project:** **Examples/Scores/Step5**

**Answer project(s):** **Examples/Scores/Step6**

**Files:** \* to be created  
**src/Scores.java**

### Instructions:

1. Open **Scores.java** and note that the hard-coded data is provided in a new form. At the bottom of the source file there is a second, package-visible class **Record**, which is a simple data structure holding **name**, **score** and **grade**. What used to be three separate arrays for these three sets of values is now a single **List<Record>** – initialized to use an **ArrayList** as the list implementation. The list is now initialized with a series of **add** calls that create new **Records** that wrap the data. The rest of the code has not been adjusted to use the new collection, however; it's the old code that used the arrays. Therefore the code will neither compile nor run correctly. Let's fix that!
2. The method prints a header and loops over the old array. Change this to a simplified **for** loop:

```
for (Record record : scores)
```

**Better Management of Test Scores****LAB 10A**

3. Change the rest of the loop to get and set values on **record**:

```
for (Record record : scores)
{
 int score = record.score;
 String grade = null;
 if (score >= 60)
 grade = "" + (char) (68 - (score - 60) / 10);
 else
 grade = "F";

 record.grade = grade;
 System.out.print (record.name);
 for (int fill = record.name.length (); fill < 24; ++fill)
 System.out.print (" ");
 System.out.println (" " + score + " " + grade);
}
```

4. Skip down to the nested loop that populates the **frequency** array. Make the same change to the outer loop (the one labeled **Student:**) as you did at the top of the method, using the simplified **for** loop syntax.
5. In testing for a matching grade inside the inner loop, change **grades[student]** to the grade in the current **record**:

```
if (record.grade.equals (possibleGrades[g]))
```

6. Make a similar change to the code that accumulates the **score** in **total**.
7. Build and test at this point. You should get the same behavior as before:

| Student         | Score | Grade |
|-----------------|-------|-------|
| -----           | ----- | ----- |
| Suzie Q         | 76    | C     |
| Peggy Fosnacht  | 91    | A     |
| Boy George      | 80    | B     |
| Flea            | 55    | F     |
| Captain Hook    | 71    | C     |
| Nelson Mandela  | 98    | A     |
| The Mighty Thor | 70    | C     |
| Oedipa Maas     | 88    | B     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |

Statistics:

```
There were 2 As given.
There were 2 Bs given.
There were 3 Cs given.
There were 2 Ds given.
There were 1 Fs given.
```

The mean score was 75.8

**Better Management of Test Scores****LAB 10A****Optional Steps**

8. After computing the grades, but before generating the statistics, add a middle section that performs an insertion sort on the **scores**, placing them in descending order in a new **ArrayList<Record>**. You'll do this over the next few steps; to start with, declare a **List<Record> sorted** and initialize to a new **ArrayList<>**.
9. Now build a nested loop: the outer loop (label it **Source:**) will iterate over **scores**, as usual, and the inner loop will iterate over **sorted**. This inner loop will need to use the full syntax of the **for** loop, because we need an integer index to operate on specific elements of the **sorted** collection.

```
Source: for (Record record : scores)
{
 Destination: for (int s = 0; s < sorted.size (); ++s)
 {
 }
}
```

10. At the top of the inner loop, declare a local **Record** called **test** and set it to **sorted.get(s)**.
11. If the **record.score** is greater than the **test.score**, call **sorted.add**, passing the index **s** and **record** (this overload of **add** inserts at a given index), and then **continue Source;**
12. If the inner loop falls through, then **record** belongs at the end of the growing **sorted** vector; pass it in a call to **sorted.add** (the overload that appends to the end of the list).
13. After the nested loops, print a new header and iterate over **sorted**, printing a formatted line for each record, just as you do in the original code. (Or, if you like, use **System.out.format** this time, instead of the filling loop. The answer code takes this approach.)

**Better Management of Test Scores****LAB 10A**

14. Build and test, and your complete output should be:

| Student         | Score | Grade |
|-----------------|-------|-------|
| Suzie Q         | 76    | C     |
| Peggy Fosnacht  | 91    | A     |
| Boy George      | 80    | B     |
| Flea            | 55    | F     |
| Captain Hook    | 71    | C     |
| Nelson Mandela  | 98    | A     |
| The Mighty Thor | 70    | C     |
| Oedipa Maas     | 88    | B     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |

| Student         | Score | Grade |
|-----------------|-------|-------|
| Nelson Mandela  | 98    | A     |
| Peggy Fosnacht  | 91    | A     |
| Oedipa Maas     | 88    | B     |
| Boy George      | 80    | B     |
| Suzie Q         | 76    | C     |
| Captain Hook    | 71    | C     |
| The Mighty Thor | 70    | C     |
| Uncle Sam       | 69    | D     |
| The Tick        | 60    | D     |
| Flea            | 55    | F     |

Statistics:

There were 2 As given.  
There were 2 Bs given.  
There were 3 Cs given.  
There were 2 Ds given.  
There were 1 Fs given.

The mean score was 75.8

## Gathering Statistics in a Map

**LAB 10B**

In this lab you will rebuild the latter part of the **Scores** application to get rid of the remaining two arrays, **possibleGrades** and **frequency**, and to replace them with a map.

**Lab project:** [Examples/Scores/Step6](#)

**Answer project(s):** [Examples/Scores/Step7](#)

**Files:** \* to be created  
**src/Scores.java**

### Instructions:

1. Open **Scores.java** and start by removing the two arrays **possibleGrades** and **frequency**, near the bottom of the **main** method. (Note one of the advantages of this mapping approach is that you will not need to define a list of possible grades ahead of time; the map will begin to accumulate whatever grades do occur.)
2. In their place, initialize a **Map<String,Integer> grades** to a new **HashMap<>**.
3. Rip out the inner loop labeled **Grade:**, but leave the last line that accumulates values in **total**.
4. Create an **Integer** reference called **frequency** and initialize it by looking up any frequency entry that may exist under the current record's grade:

```
Integer frequency = grades.get (record.grade);
```

5. Either there will be something there already for this grade, or there won't. If **frequency** is **null**, then this is the first encounter with this grade, so call **grades.put**, passing **record.grade** and the value 1. (Thanks to auto-boxing, you don't need to instantiate a new **Integer** here – the compiler will do it.)
6. Otherwise, call **grades.put** and pass **record.grade** and **frequency + 1**. (Lots of auto-conversion here! Down to **int**, then add one, then back up to **Integer**.) Note that this will overwrite the value you just looked up, which means you've incremented the existing count.

**Gathering Statistics in a Map****LAB 10B**

7. After the code that prints the “Statistics:” header to the console, replace the loop over **possibleGrades** with an iteration over **grades.entrySet**:

```
for (Map.Entry<String,Integer> entry : grades.entrySet ())
```

8. For each entry, print a line of text formatted to show the grade and the number of times it occurred, which will be **entry.getKey** and **entry.getValue**.
9. Build and test, and you should see that you’re gathering statistics accurately. However, you may have lost the A-through-F ordering – which was in place before thanks to the manual ordering of **possibleGrades**.

```
Statistics:
 There were 2 Ds given.
 There were 2 As given.
 There were 1 Fs given.
 There were 3 Cs given.
 There were 2 Bs given.
```

10. How can you get sorting into your code? This couldn’t be much simpler: change **HashMap** to **TreeMap**, build, and test again. This map type automatically sorts as keys are added, so:

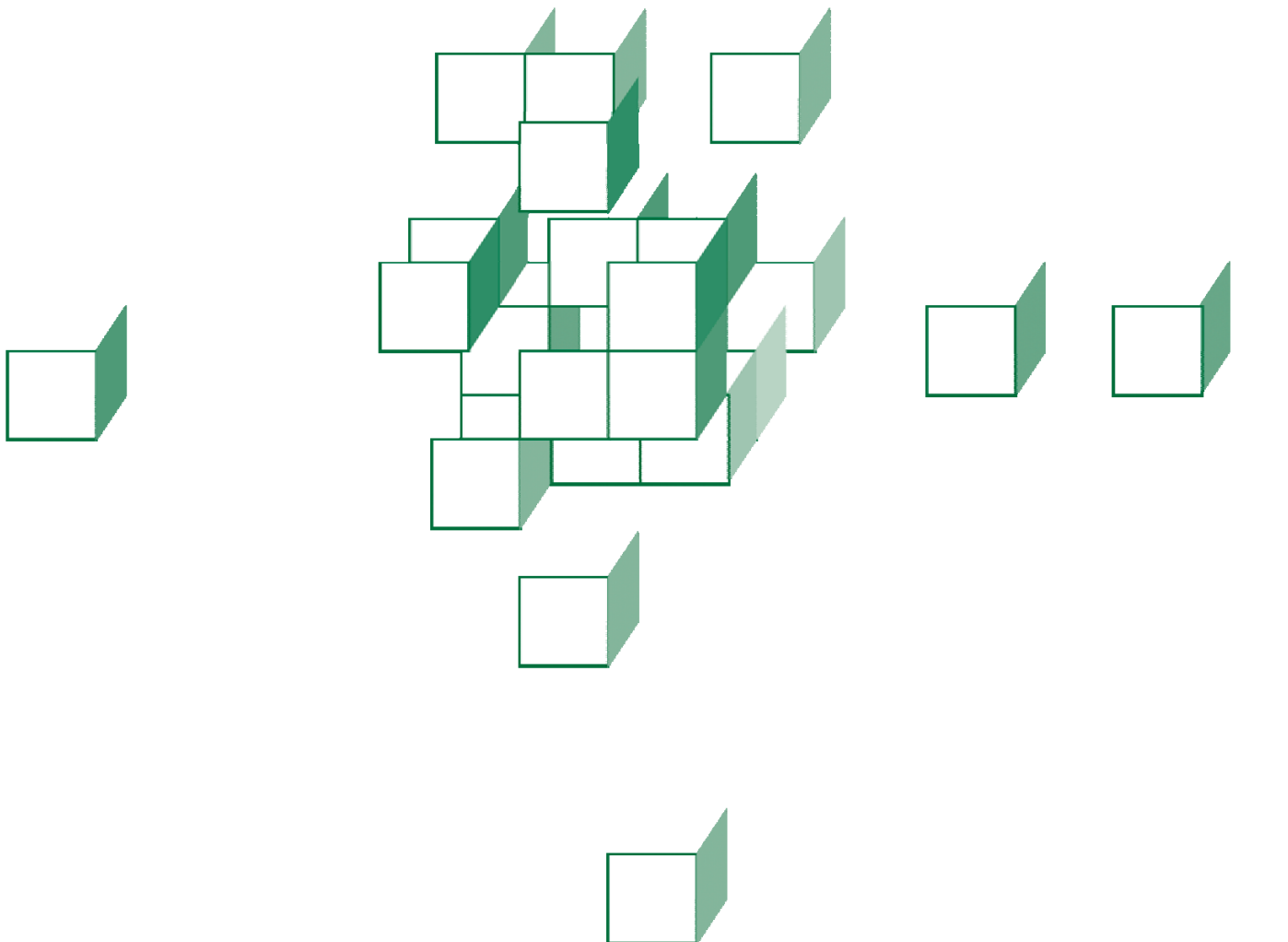
```
Statistics:
 There were 2 As given.
 There were 2 Bs given.
 There were 3 Cs given.
 There were 2 Ds given.
 There were 1 Fs given.
```





## CHAPTER 11

# EXCEPTION HANDLING AND LOGGING



## OBJECTIVES

*After completing “Exception Handling and Logging,” you will be able to:*

- Distinguish object-oriented exception handling from traditional error handling strategies.
- Implement exception handling in Java applications:
  - Throw exceptions from methods.
  - Try code that might throw exceptions and catch those exceptions so as to handle them appropriately.
  - Assure that cleanup code is executed even when code earlier in a procedure fails.
- Distinguish between checked and unchecked exceptions, and design code accordingly.
- Use the Java Logging API to produce diagnostic information that can be filtered and redirected at runtime.

## Reporting and Trapping Errors

---

- Sometimes it is possible and appropriate to define method signatures such that exceptional conditions – such as invalid parameters or incomplete object initialization – can be reported via the return value.
- However, such a reporting mechanism can become more trouble than it is worth.
  - Perhaps the return type of a certain method can't naturally support expressions of error conditions.
  - If an error occurs many nested method calls away from the code that would most properly handle the error, then every method along the way must find a way to accurately report the error, which can be cumbersome and difficult to maintain.

- Consider the following call stack:

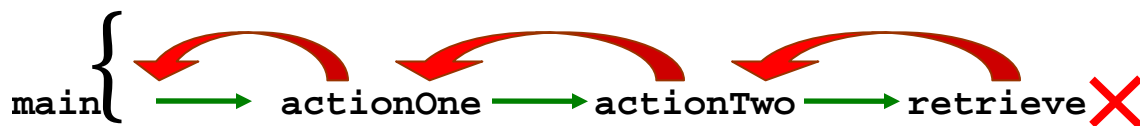
`main` → `actionOne` → `actionTwo` → `retrieve` ❌

- Method **main** calls method **actionOne**.
  - Method **actionOne** calls method **actionTwo**.
  - Method **actionTwo** calls method **retrieve**.
  - Method **retrieve**, in the course of processing, finds that something is fundamentally wrong; perhaps a database connection has timed out or died abnormally.
- How might method *main* be advised of the problem?

## Exception Handling

---

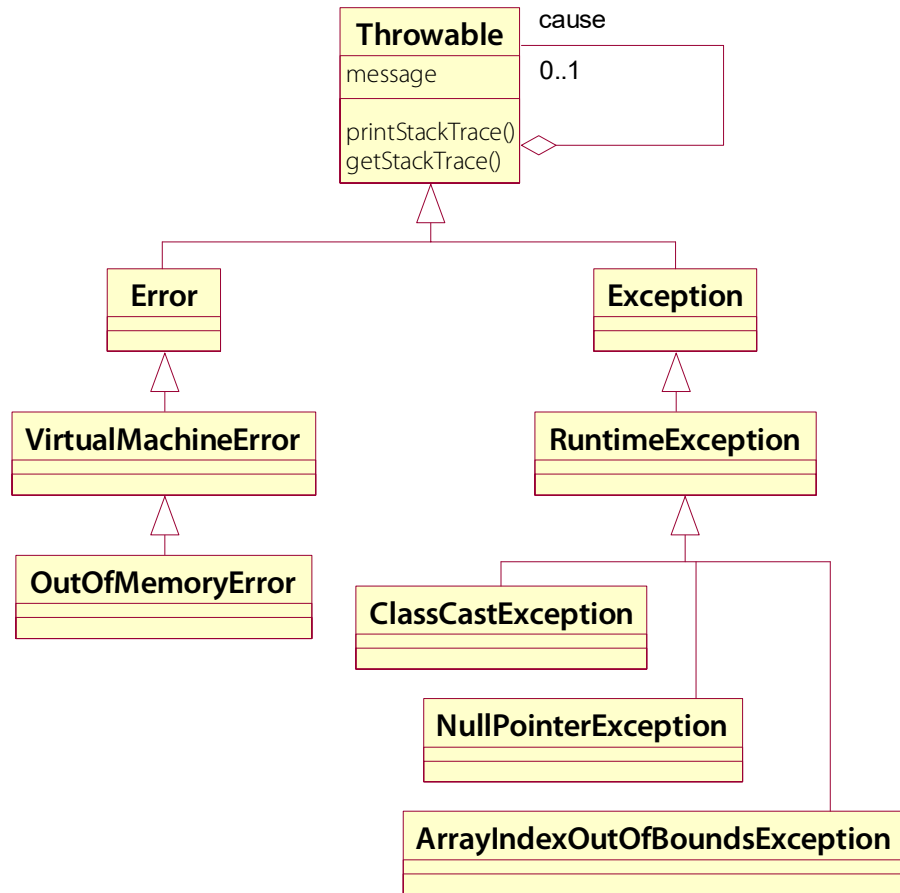
- **Exception handling** provides a straightforward means of assuring that error notifications will flow from the source of the error (or point of detection) to whatever point in the code is best suited to deal with it.
  - Where an exceptional condition occurs, an **error** or **exception** object is **thrown**.
  - Each successive calling method must either handle the exception or throw it along.
  - Eventually the exception falls through to a method call that was couched in a **try** block, and can thus be **caught**.
- Consider again the example from the last page:



- Method **retrieve** could declare that it **throws** the **BadDBConnection** exception.
- On failure, the method would create an instance of this type and **throw** it.
- Intermediate methods like **actionOne** and **actionTwo** would also declare that they might throw the exception.
- Method **main** would **try** its call to method **actionOne**, and **catch** the **BadDBConnection** exception.
- When the exception were thrown, the **catch** block in **main** would be invoked, and passed the exception object as a parameter. It could then take appropriate actions.

## Exceptions

- **Exception** in Java is in most respects an ordinary class.
- To be **thrown**, an object's type must extend **java.lang.Throwable**.



- Exceptions can and often do have state elements, used to capture information about what went wrong.
  - **Throwable** offers a **message** property – a simple string.
  - It also carries a **call-stack trace** around with it, and can print this stack trace to the console or other print stream.

## Throwing Exceptions

---

- When an exceptional condition does arise, instantiate an exception of the appropriate type and **throw** it:

```
public void doSomething ()
 throws MyException
{
 boolean b = hasThePaperBeenDelivered ();
 if (!b)
 throw new MyException ();
 goReadThePaper ();
}
```

- Control will jump from the point at which the exception was thrown.
  - It will pass down the call stack, stopping when code involved in the stack **catches** it.
  - Each method in the stack that doesn't catch it declares that it also can **throw** it.
  - Eventually the exception could propagate down through the **main** method and into the JVM thread that calls **main**.
  - At this point the launcher will print a stack trace to the console and terminate – we've seen this occasionally so far.

### Exception in thread "main"

#### **java.lang.NumberFormatException:**

```
For input string: "x"
at java.lang.NumberFormatException
 .forInputString (NumberFormatException.java:48)
at java.lang.FloatingDecimal
 .readJavaFormatString (FloatingDecimal.java:1207)
...
```

## Unchecked Exceptions

---

- The Java compiler takes a strict view of exception handling, as we'll see.
- But some exceptions defined in the Core API are known as **unchecked** exception types.
  - These are errors that are so fundamental, and hence so hard to predict, that they might occur anywhere.
  - For this reason the usual rules are suspended – and we'll see what those are and how they apply in a moment.
- Unchecked exception types are all subclasses of **Error** or **RuntimeException**. All other **Throwable** subtypes are known as **checked exceptions**.
- Commonly-encountered **RuntimeException** subtypes:
  - **ClassCastException**
  - **NullPointerException**
  - **ArrayIndexOutOfBoundsException**
  - **ArithmeticException** (including divide-by-zero errors)
  - **IllegalArgumentException**
  - **IllegalStateException**
- Commonly-encountered **Error** subtypes – these represent really fundamental problems in the virtual machine:
  - **ThreadDeath**
  - **OutOfMemoryError**

## Declaring Exceptions

---

- The Java compiler generally takes a strict view of exception handling.
- If a method includes code to **throw** a checked exception, it must declare that it **throws** the exception as part of its signature.

```
public void doSomething ()
 throws MyException
```

- By doing this the programmer asserts that the method may throw an object of the named type.
  - In fact the method may throw any subtype of a method it declares that it throws.

```
public void myMethod ()
 throws Base
{
 boolean success = workIt ();
 if (!success)
 throw new Derived ("We blew it!");
}
```

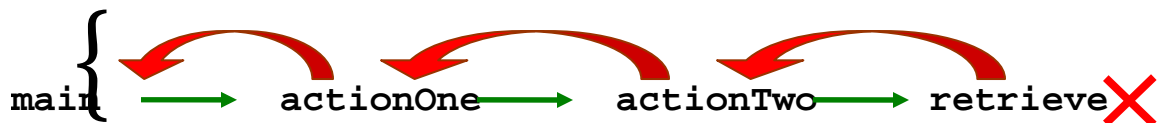
- Again, this does not apply to unchecked exceptions.
  - It is still legal to declare **throws IllegalStateException**, for example, but it is not necessary and not commonly done.



## Trying and Catching

---

- Strict checking also applies whenever a checked exception could be generated by a passage of code.
  - This includes exceptions that propagate to the method from methods it calls.
- Thus any method A that calls another method B must address any checked exceptions declared by B.
  - Method A can handle the exception with a **catch** block.
  - If it does not, it must itself say that it **throws** the exception.
  - Although it didn't create the problem, from the perspective of any method that calls A, it is the source of the problem!
- Recall our early example:



- All methods except main must declare **throws** **BadDBConnection**.
- This strict checking of signatures requires some extra attention, but it provides a guarantee:
  - You will not be surprised by any checked exceptions showing up on your doorstep!
  - Above, the coder of the **main** method knows in advance what could happen at runtime, and can plan for all contingencies.

## Catching Exceptions

---

- To handle exceptions, build a system of code blocks:
  - Enclose the code that might result in the exceptional condition in a **try** block.
  - This must be immediately followed by any number of **catch** blocks; each defines a different exception type and will catch exceptions of that type or derived types.
  - Optionally, the system can terminate with a single **finally** block, which will be executed in all cases.
  - There must be at least one **catch** or **finally** in the system.

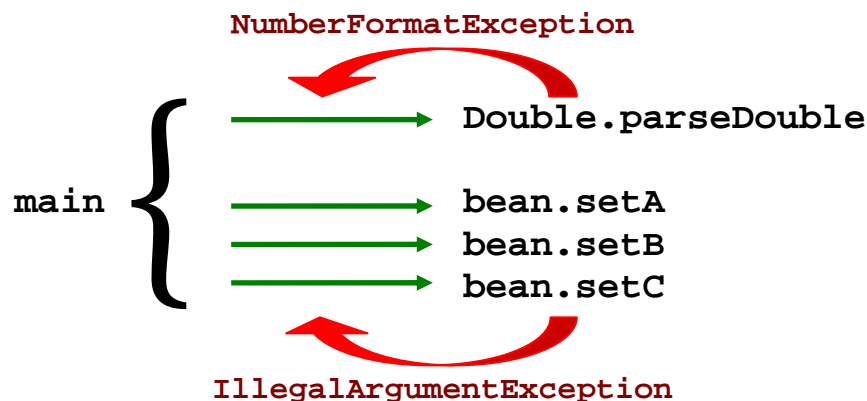
```
public void makeHimDoSomething ()
{
 bumbleAlong ();
 try
 {
 doSomething ();
 System.out.println ("Success!");
 }
 catch (MyException ex)
 {
 System.out.println ("Caught one.");
 }
 finally
 {
 cleanUp ();
 }
}
```

- The “Success!” line will be printed only if the **doSomething** method returns normally.

## Bad Geometry

DEMO

- There are three ways in which the user can mess up when launching **Ellipsoid/Step2** – the previous version of the Ellipsoid application:
  - Provide too few arguments – we already check the length of the **args** array and handle this case.
  - Provide an argument that isn't a number. The application chokes in this case, because a **NumberFormatException** would propagate through **main** to the JVM.
  - Provide a negative number, or zero. The application actually runs to completion in this case, but the data is invalid and the computations irrelevant – an ellipsoid can't have negative or zero dimensions.
- We'll add exception handling for these latter two cases – the final system is implemented in **Ellipsoid/Step3** and works like this:



## Bad Geometry

**DEMO**

1. Build and test the starter code, and observe a successful run and each of the three failures mentioned above.
  - As in earlier chapters, we're showing the command-line usage here; in the IDE you can instead place arguments such as "4 5 5" in the **Program arguments** field of a run configuration.

**run 4 5 5**

```
Three-dimensional ellipsoid -- properties:
 Semi-axis A: 4.0
 Semi-axis B: 5.0
 Semi-axis C: 5.0
 Volume: 418.8790204786391
 Type: Oblate spheroid
 ...
```

**run**

```
Usage: java cc.math.Ellipsoid <a> <c>
```

**run x y z**

```
Exception in thread "main"
java.lang.NumberFormatException:
 For input string: "x"
```

**run 4 5 -5**

```
Three-dimensional ellipsoid -- properties:
 Semi-axis A: 4.0
 Semi-axis B: 5.0
 Semi-axis C: -5.0
 Volume: -418.8790204786391
 Type: Triaxial ellipsoid
 ...
```

## Bad Geometry

**DEMO**

2. Open `src/cc/math/EllipsoidCLI.java`. We'll begin by handling the case of an ill-formed number. We can't fix the problem, exactly, and that's okay! Exception handling is not usually about fixing problems, but about handling errors gracefully when they occur.
3. Find the code that sets the semi-axis properties on the **bean**, and enclose it in a **try** block:

```
Ellipsoid bean = new Ellipsoid ();
try
{
 bean.setA (Double.parseDouble (args[0]));
 bean.setB (Double.parseDouble (args[1]));
 bean.setC (Double.parseDouble (args[2]));
}
```

4. Follow this immediately with a block to **catch** the **NumberFormatException**:

```
Ellipsoid bean = new Ellipsoid ();
try
{
 bean.setA (Double.parseDouble (args[0]));
 bean.setB (Double.parseDouble (args[1]));
 bean.setC (Double.parseDouble (args[2]));
}
catch (NumberFormatException ex)
{
 System.out.println
 ("All three arguments must be real numbers.");
}
```

## Bad Geometry

**DEMO**

### 5. Rebuild and test:

**run x y z**

All three arguments must be real numbers.

Three-dimensional ellipsoid -- properties:

Semi-axis A: 1.0

Semi-axis B: 1.0

Semi-axis C: 1.0

Semi-axis C: 1.0

Volume: 4.1887902047863905

Type: Sphere

Definition: ...

- Hmm. It's ... better ... but why are we seeing a report on an ellipsoid with dimensions (1, 1, 1) after our error message?
- It's an easy mistake to make, thinking that catching an error will abort processing in some way.
- But of course it doesn't: after we print our message, control falls through the **catch** block and resumes with all of those print statements that are meant for normal operation.
- The (1, 1, 1) dimensions just happen to be the defaults, written right into the **Ellipsoid** class.
- We could call **System.exit** from our **catch** block, but this is not a structured approach and generally makes the code structure more complicated and hard to read or to maintain.

## Bad Geometry

**DEMO**

6. Instead, expand the scope of the **try** block by putting all of those print statements in there, as well.
  - So anything that should only happen on successful parsing goes in the **try** block.
  - Only code that should be executed after some sort of recovery would stay outside/after the **try/catch** system.
7. Run again, and see a cleaner result.

**run x y z**

All three arguments must be real numbers.

8. Now open **src/cc/math/Ellipsoid.java**, and find the method to set the property **a**. Add code to test the input value, and if it is not positive, throw an **IllegalArgumentException**:

```
public void setA (double newValue)
{
 if (newValue <= 0)
 throw new IllegalArgumentException
 ("All semi-axis lengths must be " +
 "positive numbers.");

 a = newValue;
}
```

## Bad Geometry

**DEMO**

9. Do the same for methods **setB** and **setC**.

10. Rebuild and test – you’ll see that the exception is thrown and propagates down through **main**:

```
run 4 5 -5
```

```
Exception in thread "main"
java.lang.IllegalArgumentException:
 All semi-axis lengths must be positive numbers.
 at cc.math.Ellipsoid.setC(Ellipsoid.java:73)
 ...
```

11. Back in **EllipsoidCLI.java**, add a second **catch** block for all **Exceptions**. This will be a general-purpose user-interface handler that stops the exception from propagating and simply prints the associated message.

```
catch (NumberFormatException ex)
{
 System.out.println
 ("All three arguments must be real numbers.");
 System.exit (-1);
}
catch (Exception ex)
{
 System.out.println (ex.getMessage ());
 System.exit (-1);
}
```

12. Build and test one last time:

```
run 4 5 -5
```

```
All semi-axis lengths must be positive numbers.
```



## Multiple catch Blocks

---

- When several **catch** blocks accompany a **try** block, they will be checked in sequence.
  - The first one that catches exceptions of the type being propagated – or any of its base types – will be invoked.
  - The others will be ignored; thus you can't implement overlapping handlers for various types and expect them all to be triggered.
- Because exception handling is object-oriented, you must exercise care in organizing your **catch** blocks.
  - If block 1 catches a certain type **Base** and block 3 catches a type **Derived** that extends **Base**, then block 3 is dead code. It can never be invoked.
  - Fortunately the compiler will catch this for you.
- As an example, try swapping the order of the **catch** blocks in the completed demo.
- Build this and you will see a compile error:

**compile**

```
src\cc\math\EllipsoidCLI.java:30: exception
java.lang.NumberFormatException has already been
caught
```

```
 catch (NumberFormatException ex)
 ^
```

```
1 error
```

## Catching Multiple Exception Types

---

- Also, as of Java 7, there is a new option to catch multiple exception types in a single block.
  - List any number of exception types, separated by the vertical bar character:

```
public void makeHimDoSomething ()
{
 bumbleAlong ();
 try
 {
 doSomething ();
 System.out.println ("Success!");
 }
 catch (MyException | YourException ex)
 {
 System.out.println ("Caught one.");
 }
}
```

- This is useful where you want to do basically the same things in response to several possible sorts of failure – IO, network, syntax failure, etc. – as part of a complex process.
- The compile-time type of the argument is the most-derived common base type of those in the list.
  - If you need to work with specifics of the individual exception types, use **instanceof** testing, or just go back to separate **catch** blocks for each.

## Catch-and-Release

---

- It is completely legal to **throw** an exception from inside a **try**, **catch**, or **finally** block.
  - A method can catch its own exceptions! This can be a convenient way of managing complex flow control: throw exceptions in the **try** block and catch them right below.
  - From a **catch** block, you might want to handle one type of exception by throwing a different type.
  - Or you might want to “partially handle” the exception: allow the exception to propagate but do some fix-up as it goes by. So you can **catch** exception X, do your thing, and then simply **throw** it again.
  - Throwing from **finally** is rare, but legal and appropriate for some conditions.
- You can also nest exception-handling systems in any of these blocks, perhaps to recover from one particular error and carry on while still watching for another.

```
Connection con = null;
try
{
 con = getDBConnection ();
 doStuffWith (con);
}
finally
{
 try { con.close (); }
 catch (SQLException ex) {}
}
```

## Chaining Exceptions

---

- There are some drawbacks to strict checking for exception signatures.
- Especially, it threatens to foil good abstraction in systems design.
  - A subsystem of an application may trigger a certain type of exception endemic to that system.
  - Probably it defines the exception type itself.
  - Not all such exceptions can be handled within the subsystem; some will necessarily propagate to the outside world.
  - Does everyone who uses this system have to catch its exception types? That would be exposing important (maybe volatile) implementation details through the public interface.
- A way around strict checking – to be used judiciously! – is the **cause** feature in **Throwable**.
  - An exception can be **wrapped** in a new exception of a different type that might be more appropriate to methods further down the call stack.
  - This can be done any number of times, creating a **chain** of exceptions that can be followed back to the root cause.



## Exceptions in the Car Dealership

**LAB 11A**

**Suggested time: 30 minutes**

In this lab you will go around the car dealership application and enhance the code to make it more robust in the face of bad data or exceptional conditions.

- As in **Ellipsoid**, you will guard against non-numeric inputs by the interactive user.
- Where enumerated types have been defined, code that checks for possible values and fails to match an enumerated value will throw a new, application-defined **BadDataException**.

Detailed instructions are found at the end of the chapter.

## try-with-resources

---

- Java 7 introduces a new option for managing resources that would not simply be garbage-collected in case of failure due to a thrown exception.
- The traditional means of dealing with such a resources is a **finally** block.
  - This is guaranteed to be called in any case at runtime, and so is a good place for cleanup code.
- The more facile approach now is to “try with resources,” initializing local variables for your *try* block in a special parenthesized statement list:

```
try (OutputStream out =
 new FileOutputStream ("myfile.txt"))
{
 out.write (...);
 ...
}
```

- With this new construction, there is no need for a **finally** block to clean up the stream and underlying file object.
- Indeed, this is the only way in which a **try** block can stand alone, with no **catch** or **finally** (though those are still legal).
- For this to work, all local variables thus initialized must be of a type that implements a new interface **java.lang.AutoCloseable**.
- That interface’s **close** method will be called automatically on any exit from the **try** block – normal or exceptional.

## Auto-Closing

### EXAMPLE

- We'll see more interesting examples of this feature once we have some resources worth closing!
- But see **AutoClose** for a trivial example.
- **src/cc/ex/MyResource.java** defines a class that implements the necessary interface:

```
public class MyResource
 implements AutoCloseable
{
 public void parse (String text)
 {
 char c = ' ';
 for (int i = 0; i < 5; ++i)
 c = text.charAt (i);
 }

 public void close ()
 {
 System.out.println
 ("I was closed automatically!");
 }
}
```

- The **parse** method is dangerous: it will fail if given a string of less than five characters.
- The class implements the **close** method as required.

## Auto-Closing

### EXAMPLE

- In **src/cc/ex/MyCaller.java**, a block of code uses an instance of this resource class, and takes advantage of try-with-resources to be sure that it will be closed when done:

```
public class MyCaller
{
 public static void main (String[] args)
 {
 String text =
 args.length != 0 ? args[0] : "Long enough";

 try (MyResource res = new MyResource ();)
 {
 res.parse (text);
 }
 }
}
```

- The call to **parse** will succeed in some runs of the program, fail in others.
  - But the resource will be cleaned up regardless.
- Give this a quick test:

**run**

I was closed automatically!

**run bop**

I was closed automatically!

Exception in thread "main"

java.lang.StringIndexOutOfBoundsException: String  
index out of range: 3



## Logging

---

- **System.out.println** and **.format** are such easy ways of producing console output that it's tempting to use them to log warnings and errors as they occur in an application.
  - The **Throwable.printStackTrace** method is equally attractive: a quick method call that dumps lots of useful information at runtime.
- This is not good practice, however.
- It fails to provide for different expectations at development time and production time.
  - The developer wants to see diagnostic information.
  - The end user typically does not!
  - In either situation, more or less diagnostics might be desirable over time: the developer needs less and less; the user needs none – until something goes wrong!
- The best practice that has evolved is the use of a configurable **logging** API.
  - Diagnostic messages can be sent to a logging provider.
  - At runtime, the final destination(s) for this information – files, consoles, GUIs, etc. – its format, and frequency or severity, all can be configured for a given situation.

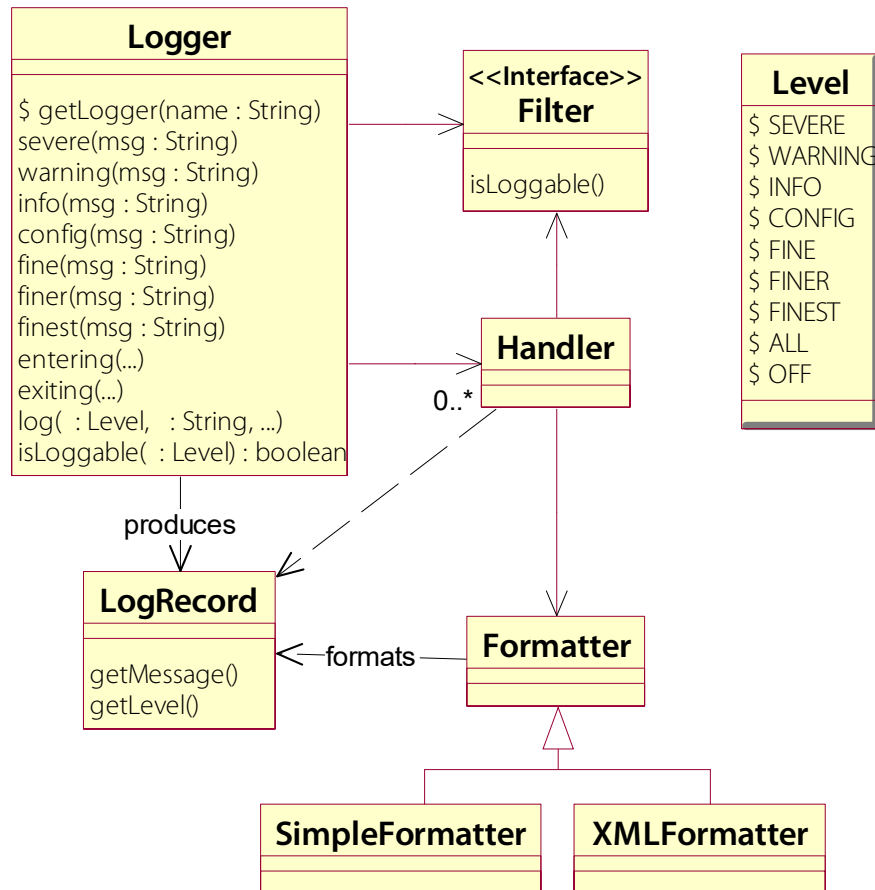
## Logging APIs

---

- There are several popular logging APIs for Java.
  - **Log4J** from Apache has been the most popular, is excellent, and is in wide use today.
  - As of Java 1.4, the Core API offers a **native logging API**.
  - A abstraction layer called **Jakarta Commons Logging** is also widely used; it can be backed by Java standard logging, Log4J, or other providers.
- We'll focus on the Java standard API, which offers the following features:
  - A **factory pattern** for creating logs
  - **Multiple logs**, identified by string names
  - **Severity levels** to distinguish various diagnostic output – this is typically used by the logging implementation to **filter** the actual log output based on runtime configuration
  - Configurable **handlers** to produce messages to console, files, or other output destinations

## The Java Logging API

- The Java standard logging API is implemented in package **java.util.logging**:



- The entry point for most application code is the **Logger**.
- Most of the rest of the API can be left to automatic use by the JRE.
  - Based on a logging **configuration file**, the **LogManager** will create **Handler** instances and assign them to various logs.
  - This will cause messages to be filtered, and to appear at certain destinations in certain formats.

## Using a Logger

---

- Use the factory method **getLogger** to derive reference to a specific log.
- Typically, a class that needs logging will initialize a static field to an appropriate **Log** instance.
  - It will then call logging methods such as **warning**, **severe**, and **info** to produce diagnostics to the log.

```
package com.me;
import java.util.logging.Logger;

public class MyClass
{
 public void foo ()
 {
 log.info ("Doing something well.");
 if (!doSomethingWell ())
 log.warning ("Did something poorly!");
 }

 private static Logger log
 = Logger.getLogger ("com.me.MyClass");
}
```

- When handling exceptions, use the **log** method to provide severity level, message, and the source exception:

```
try
{
}
catch (SomeException ex)
{
 log.log (Level.SEVERE, "Bad thing!", ex);
}
```

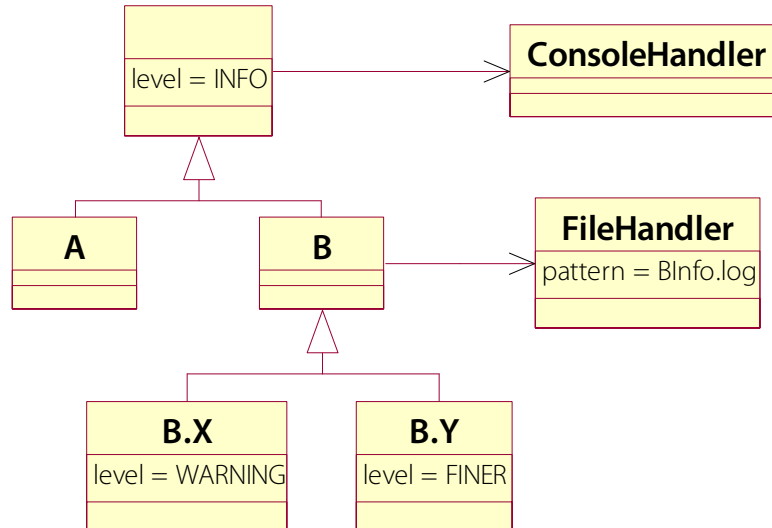
## Logging Levels

---

- The logging API defines seven standard levels for log messages, as static finals in the class *Level*; in descending order of severity, they are:
  - **SEVERE**, for errors and fatal errors
  - **WARNING**, for definite problems that may or may not have significant impact on the system
  - **INFO**, for significant but normal operating conditions and events
  - **CONFIG**, for runtime configuration details
  - **FINE**, for debug-only information
  - **FINER**, for trace-level information (**entering** and **exiting** methods automatically use this level)
  - **FINEST**
- Custom levels can be defined, as well.
- A log message will be produced through a given path if all filtering levels are at its own level or a lower one.
- Use **Logger.isLoggable** to determine ahead of time if a given message would get through or be thrown out.
  - Skip time-consuming diagnostic routines in the latter case.
- There are also pseudo-levels **ALL** and **OFF**, which usually function as master switches to enable all logging or to disable it completely.

## Log Hierarchy

- Java logging defines a hierarchical structure for loggers, and a corresponding naming pattern.
- There is a **root logger**, whose name is “”.
- From there, one can define any number of logs, with names composed of simple tokens separated by dots – often, class or package names are used.
- The hierarchy is derived from the log names:
  - The root is the parent of all logs with single-token names.
  - Any log “A” is the parent of any log “A.B”.
  - Here, the root log is the parent of logs “A” and “B,” and “B” is the parent of “B.X” and “B.Y.”



- The hierarchical structure matters for filtering levels and what handlers are assigned to what logs.
  - Both features can be inherited from parent logs.

## Logging and Configuration

### EXAMPLE

- In **Logging** is a Java application **Chattering**, which carries out a simple script:

- Initialize two logs, one called **root** based on the name "", and one called **log** based on the name of the class itself:

```
static Logger root = Logger.getLogger ("");
static Logger log = Logger.getLogger
 (Chattering.class.getName ());
```

- Write messages to **log** at **FINE** and **INFO** levels:

```
try
{
 log.fine ("Debugging just for us");
 log.info ("Information just for us");
```

- Log messages to **root** at several levels, and then at **SEVERE** level based on catching an exception:

```
 root.fine ("A debugging message");
 root.info ("An informational message");
 root.warning ("A warning message");
 root.severe ("An error message");

 Object x = null;
 System.out.println (x.getClass ().getName ());
}
catch (Exception ex)
{
 root.log (Level.SEVERE, "Exception", ex);
}
```

## Logging and Configuration

**EXAMPLE**

1. Build and run the application and see that the logging messages appear directly in the console:

```
Jun 20, 2022 10:57:00 PM Chattering main
INFO: Information just for us
Jun 20, 2022 10:57:00 PM Chattering main
INFO: An informational message
Jun 20, 2022 10:57:00 PM Chattering main
WARNING: A warning message
Jun 20, 2022 10:57:00 PM Chattering main
SEVERE: An error message
Jun 20, 2022 10:57:00 PM Chattering main
SEVERE: Exception
java.lang.NullPointerException: Cannot invoke
"Object.getClass()" because "<local1>" is null
 at Chattering.main(Chattering.java:27)
```

- But only messages at **INFO** level and above are seen.
- This is the standard logging configuration for Java, which is found in **%JAVA\_HOME%\jre\lib\conf\logging.properties**.

```
#####
Default Logging Configuration File
handlers= java.util.logging.ConsoleHandler
.level= INFO
```

- The **handlers** line directs all log records to the console.
- The **.level** line sets the root (or default) level to **INFO**.



## Logging and Configuration

**EXAMPLE**

2. There are two configuration files provided in the demo directory.  
**logging1.properties** sets the default level higher and the level for the **Chattering** log lower:

```
.level=WARNING
.handlers=java.util.logging.ConsoleHandler
```

```
Chattering.level=FINE
...
```

3. Run the application with this logging configuration (note that a rebuild is unnecessary – this is pure runtime configuration):

```
java -classpath build
-Djava.util.logging.config.file=logging1.properties
Chattering
```

```
Jun 20, 2022 10:57:01 PM Chattering main
FINE: Debugging just for us
Jun 20, 2022 10:57:01 PM Chattering main
INFO: Information just for us
Jun 20, 2022 10:57:01 PM Chattering main
WARNING: A warning message
Jun 20, 2022 10:57:01 PM Chattering main
SEVERE: An error message
Jun 20, 2022 10:57:01 PM Chattering main
SEVERE: Exception
java.lang.NullPointerException: Cannot invoke
"Object.getClass()" because "<local1>" is null
 at Chattering.main(Chattering.java:27)
```

- Now the **INFO** message to the **root** log is filtered out, but the one to the **log** object is passed, and so is the **FINE** one.

## Logging and Configuration

**EXAMPLE**

4. **logging2.properties** preserves the levels from the previous configuration, and adds a **FileHandler** assignment for the **Chattering** log:

```
.level=WARNING
.handlers=java.util.logging.ConsoleHandler

Chattering.level=FINE

java.util.logging.ConsoleHandler.formatter=
 java.util.logging.SimpleFormatter
java.util.logging.ConsoleHandler.level=FINE

Chattering.handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level=FINE
java.util.logging.FileHandler.pattern=Log.xml
java.util.logging.FileHandler.limit=50000
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=
 java.util.logging.XMLFormatter
```

## Logging and Configuration

**EXAMPLE**

5. Thus when we run with this configuration we see the same things in the console, and a file **Log.xml** is also generated:

```
<?xml version="1.0" encoding="windows-1252"
standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
 <date>2014-11-04T12:31:58</date>
 <millis>1415122318244</millis>
 <sequence>0</sequence>
 <logger>Chattering</logger>
 <level>FINE</level>
 <class>Chattering</class>
 <method>main</method>
 <thread>10</thread>
 <message>Debugging just for us</message>
</record>
<record>
 <date>2014-11-04T12:31:58</date>
 <millis>1415122318275</millis>
 <sequence>1</sequence>
 <logger>Chattering</logger>
 <level>INFO</level>
 <class>Chattering</class>
 <method>main</method>
 <thread>10</thread>
 <message>Information just for us</message>
</record>
</log>
```

- Note that only the records logged to **Chattering** appear in this file.
- The root log records only go the console.

## Logging in the Car Dealership

**LAB 11B**

**Suggested time: 30 minutes**

In this lab you will add appropriate logging to several classes in the Cars application.

Detailed instructions are found at the end of the chapter.

## SUMMARY

- **Exception handling provides an elegant way to assure that error information flows smoothly from source to handler.**
- **Java exception handling is completely object-oriented:**
  - Exceptions are ordinary Java objects, defined by classes.
  - Signature checking and catching of exceptions takes specialization relationships into account.
- **Strict checking of most exceptions makes coding a bit more laborious, but there is a significant payoff in confidence for any given method – it knows exactly what could and could not happen.**
- **The system is very flexible and allows for many different designs.**
  - Exceptions can propagate through methods quietly.
  - They can be caught in complex systems of **catch** and **finally** blocks.
  - These systems can be nested for more precise control of flow.
- **The Java Logging API provides a flexible and configurable structure for diagnostic information.**
  - This is an especially good fit to exception-handling code: get in the habit of logging some sort of information in your **catch** blocks.

## Exceptions in the Car Dealership

**LAB 11A**

In this lab you will go around the car dealership application and enhance the code to make it more robust in the face of bad data or exceptional conditions.

- As in **Ellipsoid**, you will guard against non-numeric inputs by the interactive user.
- Where enumerated types have been defined, code that checks for possible values and fails to match an enumerated value will throw a new, application-defined **BadDataException**.

**Lab project:** Cars/List/Step2

**Answer project(s):** Cars/List/Step3

**Files:** \* to be created  
**src/cc/cars/Application.java**  
**src/cc/cars/Persistence.java**  
**src/cc/cars/BadDataException.java**  
**src/cc/cars/Car.java**  
**src/cc/cars/UsedCar.java**

### Instructions:

1. Build and test the starter application in two ways. First, try to buy a car, and offer “a million dollars”:

```
run buy BA0091
Seller is class cc.cars.sales.HardNosed
"I can sell the car for $29,249.99."
a million dollars
Exception in thread "main" java.lang.NumberFormatException: For input
string: "a million dollars"
 at ...readJavaFormatString(FloatingDecimal.java:2043)
 at ...parseDouble(FloatingDecimal.java:110)
 at java.lang.Double.parseDouble(Double.java:538)
 at cc.cars.Application.main(Application.java:141)
```

2. Well, for such a generous offer, the seller certainly could have been nicer to us! The words are being misinterpreted as a number, and the application chokes.
3. Now try driving that same car:

```
run drive BA0091
Exception in thread "main" java.lang.NullPointerException
 at cc.cars.Car.testDrive(Car.java:162)
 at cc.cars.Application.main(Application.java:114)
```

**Exceptions in the Car Dealership****LAB 11A**

4. Oops! If you look in **Persistence.java**, you'll see that a bad value has been introduced into the code for this car: **handling** can't be **null**. (And, of course, in the real world we couldn't fix this just by fixing this bit of source code; it would live in a database somewhere, or have been supplied to us interactively or over a network connection.)
5. We'll fix these two problems in sequence. First, open **Application.java** and find the loop in handling the "buy" command where the application calls **UserInput.getString** and converts to a number to pass to **seller.willSellAt**.

```
while (!carSold && !(offer = UserInput.getString ())
 .equalsIgnoreCase ("quit"))
{
 double money = Double.parseDouble (offer);
 double counter = seller.willSellAt (money);
 if (counter == money)
 {
 System.out.println ("\Sold!\n");
 carSold = true;
 }
 else
 System.out.format ("\Well, I can't do that, but " +
 "I could sell it for $%,1.2f.\n", counter);
}
```

6. Where would you inject exception handling to deal with bad user input most gracefully?
7. You could wrap the call to **parseDouble** in a **try** block, but then what would you do? Exit the application? That's going overboard – we should be able to tell the user how to do better and let him or her try again. We could print a message and then **continue** the loop ... not bad, but if you think about how **try** and **catch** work, they really offer us the flow control we want without having to explicitly **continue**.

**Exceptions in the Car Dealership****LAB 11A**

8. The trick is just to wrap the entire code block under **while** in a **try** block. Then **catch** the **NumberFormatException**, print an error message ... and you're done. What will happen when **catch** falls through? The while loop will resume! which is exactly what we want to happen. So add code like this:

```
while (!carSold && !(offer = UserInput.getString ())
 .equalsIgnoreCase ("quit"))
try
{
 double money = Double.parseDouble (offer);
 double counter = seller.willSellAt (money);
 if (counter == money)
 {
 System.out.println ("\Sold!\n");
 carSold = true;
 }
 else
 System.out.format ("\Well, I can't do that, but " +
 "I could sell it for $%,1.2f.\n", counter);
}
catch (NumberFormatException ex)
{
 System.out.println ("\Sorry, I need a monetary offer. "
 + "We don't barter here.\n");
}
```

9. Build and test that first use case again, and you should see the application handles the bad input beautifully:

```
run buy BA0091
Seller is class cc.cars.sales.HardNosed
"I can sell the car for $29,249.99."
10000
"Well, I can't do that, but I could sell it for $29,249.99."
two chickens and a plough
"Sorry, I need a monetary offer. We don't barter here."
quit
```

10. Now we'll address the problem of invalid data. Review the new **BadDataException** class and see that it can capture what object has the bad data and a string description of it. It extends **Exception** and uses that class' ability to capture a **message** by passing one of its arguments to the superclass constructor:

```
public BadDataException (Object source, String message)
{
 super (message);
 this.source = source;
}
```



**Exceptions in the Car Dealership****LAB 11A**

11. Open **Car.java** and find the constructor, and add code to check the **handling** value before initializing the object's state:

```
public Car (...)
{
 if (handling == null)
 throw new BadDataException
 (this, "Handling can't be null");

 this.make = make;
 ...
}
```

12. You'll see that the IDE's syntax checker flags this code with an error. Or, try to **compile** at this point – but of course you can't, because you're breaking the exception-handling rules.

**compile**

```
src\cc\cars\Car.java:64: unreported exception cc.cars.BadDataException;
must be caught or declared to be thrown
```

```
 throw new BadDataException (this,
 ^
```

1 error

13. Declare that the constructor **throws BadDataException**. Two other classes will light up with red errors in your IDE, or the next time you **compile**:

**compile**

```
src\cc\cars\UsedCar.java:72: unreported exception
cc.cars.BadDataException; must be caught or declared to be thrown
 super (make, model, year, VIN, color, stickerPrice, horsepower,
 ^
```

1 error

So, when a base-class constructor throws a checked exception, derived-class constructors must handle that exception – just as calling methods must.

14. In this case, best to declare that the **UsedCar** constructor **throws** the exception as well – and you might want to do a similar check in this constructor, to be sure that the **condition** is not **null** either.

This was not a problem in **Ellipsoid** with **IllegalStateException**. Why not? If you don't understand the difference between the two situations, ask your instructor.

**Exceptions in the Car Dealership****LAB 11A**

15. . If you try compiling again, you see that you will have to follow the possible call stack down to **Persistence.java**, because this exception is now unhandled there, in both the **addCar** and **addUsedCar** methods. For example you'll see:

```
src\cc\cars\Persistence.java:29: unreported exception
cc.cars.BadDataException; must be caught or declared to be thrown
 result.add (new Car (make, model, year, VIN, color, price,
```

16. In the offending **addCar** method, wrap the call to **result.add** in a **try** block. Catch the exception and do nothing. This is not good practice! If you leave code like this, it's called "swallowing" the exception, and it will drive other programmers crazy. So we will improve this code in the following lab.

But for now, this technique assures that any invalid objects will quietly be excluded from inventory, which is what we want.

```
try
{
 result.add (new UsedCar (make, model, year, VIN, color, price,
 horsepower, torque, handling, odometer, condition,
 pricePaid);
}
catch (BadDataException ex)
{
}
```

17. Make a similar change to **addUsedCar**.

18. Build – you should get a clean compile now – and test:

**run list**

```
ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
HJ5599: 2015 Saab 9000 (Pearl) $34,498.99
ME3278: 2014 Honda Accord (Red) $29,999.99
...
```

See that the problem car has been excluded from the inventory – it used to be the third car in the list – and that any of the cars on the lot can indeed be taken for test drives, without adverse effect.

# Logging in the Car Dealership

**LAB 11B**

In this lab you will add appropriate logging to several classes in the Cars application.

**Lab project:** Cars/List/Step3

**Answer project(s):** Cars/List/Step4

**Files:** \* to be created  
**src/cc/cars/Persistence.java**  
**src/cc/cars/Application.java**  
**src/cc/cars/Dealership.java**  
**src/cc/cars/sales/Simple.java**  
**src/cc/cars/sales/Standard.java**  
**src/cc/cars/sales/HardNosed.java**  
**run or run.bat**  
**Cars.logging.properties**

## Instructions:

1. First, let's address the issue we left hanging in the previous lab. Open **Persistence.java** and define a logger for the class:
2. Now, in each of the two methods **addCar** and **addUsedCar**, log the exclusion from inventory, including the exception that caused the exclusion:

```
private static final Logger LOG =
 Logger.getLogger (Persistence.class.getName ());
```

```
catch (BadDataException ex)
{
 LOG.log (Level.WARNING,
 "Excluding " + VIN + " from inventory.", ex);
}
```

The choice of logging level here is subjective, and depends on the importance of consistent data and a complete inventory to the application. We might consider this to be SEVERE, or an INFO-level item; below INFO would probably not be right.

**Logging in the Car Dealership****LAB 11B**

- Run the application, and see that your warning shows up in the console:

```
run list
```

```
Oct 31, 2014 2:09:43 PM cc.cars.Persistence addCar
WARNING: Excluding BA0091 from inventory.
cc.cars.BadDataException: Handling can't be null
 at cc.cars.Car.<init>(Car.java:65)
 at cc.cars.Persistence.addCar(Persistence.java:35)
 at cc.cars.Persistence.loadCars(Persistence.java:76)
 at cc.cars.Dealership.<init>(Dealership.java:39)
 at cc.cars.Application.main(Application.java:80)

ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
...
```

- Open **Dealership.java** and give it a logger as well.
- Anywhere you use external information to configure the application, it's usually a good idea to log information on the process of reading the configuration and setting things up. In the **sellCar** method, reflect the current configuration of seller type, if any:

```
if (type == null)
{
 type = (System.currentTimeMillis () / 10) % 2 == 0
 ? "Standard" : "HardNosed";
 LOG.config
 ("No seller type configured; randomly chose " + type);
}
else
 LOG.config ("Configuration calls for seller of type " + type);
```

- Also, if a type is configured, but you can't translate it to a known class, say so, before the method returns an otherwise unexplained **null**:

```
LOG.severe ("Unrecognized seller type: " + type);
return null;
```

- Run the application and try to buy a car:

```
Seller is class cc.cars.sales.HardNosed
"I can sell the car for $26,099.09."
quit
```

You don't see any of your logging output, do you? Remember that the default logging level is INFO. (The **Application** class has been printing out the type of the seller for a while now, but just by **System.out.println**.)

**Logging in the Car Dealership****LAB 11B**

8. Run the application with a specific seller type by setting the system property as follows:

```
-Dcc.cars.Seller.type=NonExistent
```

You can do this in a run configuration – be sure to put the above text in the VM arguments, and not the normal program arguments. Or, if working on the command line, edit the script that you’re using – **run** or **run.bat** – to include the above switch.

9. Run again, and you should see that (a) the application can’t proceed, and (b) it shows you why, as the SEVERE-level logging you included in **sellCar** does come through:

```
Oct 31, 2014 2:13:42 PM cc.cars.Dealership sellCar
SEVERE: Unrecognized seller type: NonExistent
Exception in thread "main" java.lang.NullPointerException: Cannot
 invoke "cc.cars.Seller.getClass()" because "seller" is null
 at cc.cars.Application.main(Application.java:132)
 ...
```

10. Open **Cars.logging.properties** and add settings there for the root logging level and the console handler’s logging level:

```
.level=CONFIG
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=CONFIG

java.util.logging.FileHandler.formatter=...
...
```

11. Remove your **cc.cars.Seller.type** setting, and instead set the logging properties to use your file:

```
-Djava.util.logging.config.file=Cars.logging.properties
```

12. Run the application normally to buy a car. Now you can see the CONFIG-level output from **sellCar** as well.

```
Oct 31, 2014 2:18:17 PM cc.cars.Dealership sellCar
CONFIG: No seller type configured; randomly chose Standard
Seller is class cc.cars.sales.Standard
"I can sell the car for $26,099.09."
quit
```

**Logging in the Car Dealership****LAB 11B**

13. One part of the application that may leave a developer confused is the selling process – why was a certain price calculated, why was a certain offer accepted or refused? Open **Simple.java** and define a **LOG** field.

14. In **getAskingPrice**, explain the decision-making as you go. This would be appropriate for (at most) FINE level:

```
public double getAskingPrice ()
{
 double result =
 car.getStickerPrice () * (1.0 - initialDiscount);
 LOG.fine
 ("Giving initial discount of " + initialDiscount + "%.");

 if (car instanceof UsedCar used &&
 used.getPricePaid () > result)
 {
 result = ((UsedCar) car).getPricePaid ();
 LOG.fine
 ("Adjusting to the price paid for the (used) car.");
 }

 return result;
}
```

15. Put similar logging in the **willSellAt** method, explaining the rationale for the counteroffer that the method returns. We'll leave off with specific code listings at this point and let you arrange the logging output as you think is appropriate. But note that a method such as this will often need to be re-structured, to allow logging statements to occur in the middle of what was a single, fluent expression.
16. You can run the application and try to buy a car – but, it won't surprise you that the FINE-level logging doesn't appear in the console, since you've configured for CONFIG and up.

If you like, tweak the logging properties to show FINE, FINER, or FINEST, and you will see these log entries as well.

## Logging in the Car Dealership

## LAB 11B

But let's say we don't want FINE as a master level setting. After all, many components log quite a lot of information at this level, down to debug-only instrumentation and even method-call tracing. Let's configure the logging system a little more precisely, so we can observe the sales logic without committing to FINE across the board.

17. Start by setting the level to FINE only for logs at or under **cc.cars.sales**:

```
cc.cars.sales.level=FINE
```

18. Now connect the pre-defined file handler to these logs:

```
cc.cars.sales.handlers=java.util.logging.FileHandler
```

19. If you run the application now, with **-Dcc.cars.Seller.type=Simple** to force the use of the **Simple** seller, any car-buying you undertake will be logged to a file **Cars.log.txt**, in the project directory.

It will not be written to the console, because even though the loggers are now passing FINE entries, the **ConsoleHandler** is still set to ignore everything below CONFIG.

### Optional Steps

20. Though external configuration is generally preferred, you can also configure the logging system from application code. Open **Application.java** and, at the top of the **main** method, add the following code:

```
Level level = Level.FINE;
Logger.getLogger("").setLevel(level);
for (Handler handler : Logger.getLogger("").getHandlers())
 if (handler instanceof ConsoleHandler)
 handler.setLevel(level);
```

The loop is necessary in order to find the configured handler that performs console output; there is for example no **getConsoleHandler** method. So the above code sets both the main logging level and the console output level to FINE.

21. Run again and you should see your FINE-level output in both the console and the log file.
22. Finally, if you have time, give **Standard** and **Hard-Nosed** sellers FINE-level logging on their decision-making, and test that out as well.

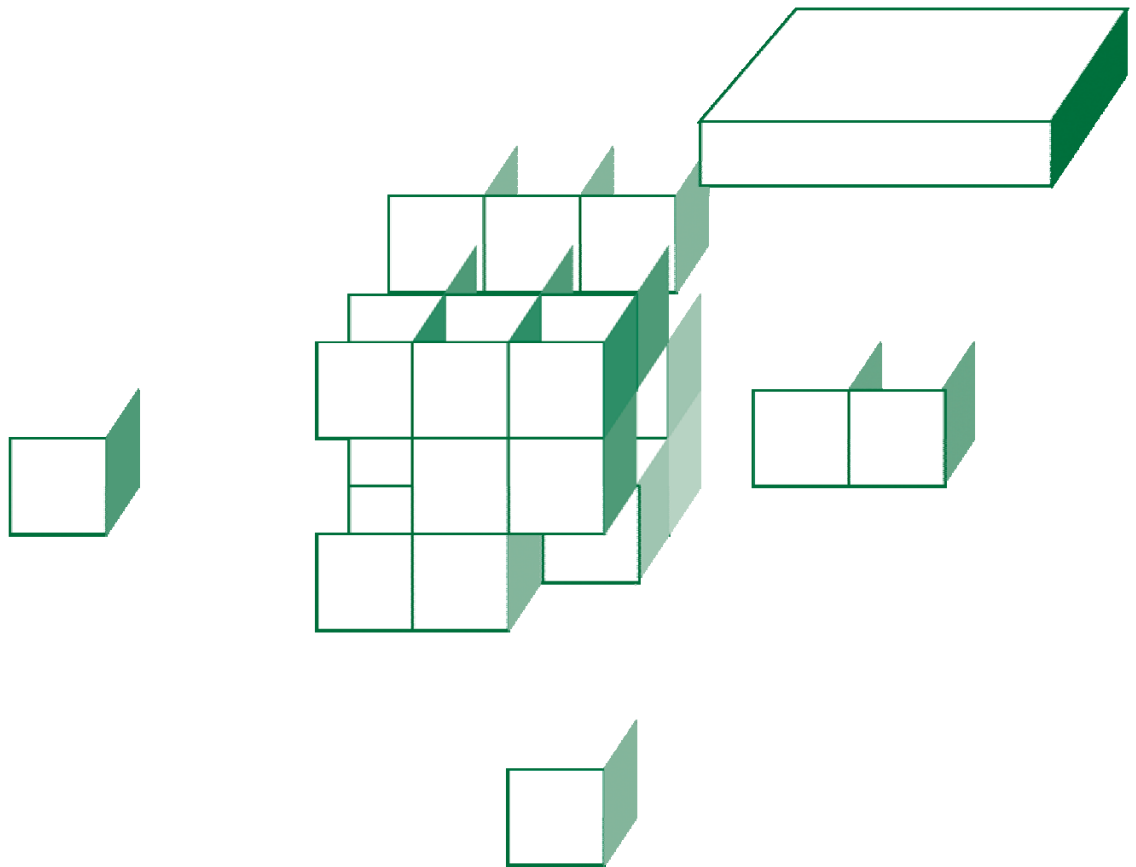
**Cars/List/Step5** puts a couple of things back in place for later work: the **handling** of the Ford Taurus is back to **Handling.GOOD**, and the **Application** reads an external logging configuration file, so we're not looking at FINE-level logging by default in later exercises.





# CHAPTER 12

## NESTED CLASSES



## OBJECTIVES

*After completing “Nested Classes,” you will be able to:*

- Understand the motivation for inclusion of inner classes in the Java architecture.
- Create and use named inner classes.
- Implement inner classes to refer to outer-class members, and to explicitly reference the outer object.
- Create and use anonymous inner classes.

## Classes Everywhere!

---

- Thus far we've worked entirely with classes that are defined in their own source files.
- It is possible in Java to define one class as a member of another, and this is known as a **nested class**.
- If marked **static**, a nested class just provides a narrower **scope** at which to define state and behavior.
  - For example you might want to encapsulate a data type that you would use as part of the **internal logic** of a top-level class, or as a compound **return type** from some of your methods.
- If not marked **static**, a nested class is an **inner class**.
  - This also provides a narrower scope for state and behavior.
  - It also establishes a special relationship between **instances** of the inner and outer classes, which can be useful for certain kinds of **event handling** and **callback** logic.
- In fact, you can define a class inside any block of code – this is known as a **local class**.
  - Uses for local classes are few and far between, but for example you might want to encapsulate a frequently-used piece of a method's computations, and keep it at that method's scope.
- Perhaps most strangely, you can define a class and instantiate it, all in one passage of code and without giving the class a name. This is an **anonymous class**.
  - This is a mechanism for defining **inner classes** more easily and directly, while compromising any potential for **reuse**.

## Nested and Static Classes

---

- At their simplest, **nested classes** are just classes defined inside of other classes, and a nested class looks a lot like top-level classes:

```
public class OuterClass
{
 private static class StaticClass

 private class InnerClass

 public interface Callback { ... }
}
```

- It's placed inside the **outer class**, and becomes a **member** of that class, on par with fields and methods.
  - A nested class may be **private**, **protected**, **public**, or have default visibility, while a top-level one can only be public or default.
  - Nested **interface** definitions are fine, too.
  - Nested classes can have **constructors**, **fields**, **methods** ... and their own **inner classes**, although more than one level of nesting is unusual.
  - The fully-qualified name of a nested class is of the form **package1.packageN.OuterClass.InnerClass**.
  - Nested classes are compiled to separate class files with names that use the \$ separator, as in **OuterClass\$InnerClass**.
  - Nested classes can **extend classes** and **implement interfaces**, and in fact this is a big part of the motivation for inner classes.
- A **static class** is just a nested class marked **static**.

## Using Static Classes

---

- A static class behaves exactly as would a top-level class, only subject to the rules of visibility as applied to its location.
- As with other static members, static classes are visible from inside the enclosing class, and can be referenced by their simple names.

```
public class OuterClass
{
 private static class StaticClass
 {
 private int x;
 public void foo ();
 }

 private void bar ()
 {
 StaticClass obj = new StaticClass ();
 obj.foo ();
 }
}
```

- From outside the enclosing class, any references to the nested class must be qualified:

```
public class OtherClass
{
 public void baz ()
 {
 OuterClass.StaticClass obj =
 new OuterClass.StaticClass ();
 obj.foo ();
 }
}
```

- You can **import** nested classes, and **import static** as well.

## A Private Record Type

### EXAMPLE

- In **Scores/Step9**, the **Record** class we used in the previous chapter's labs has been moved inside the **Scores** class.

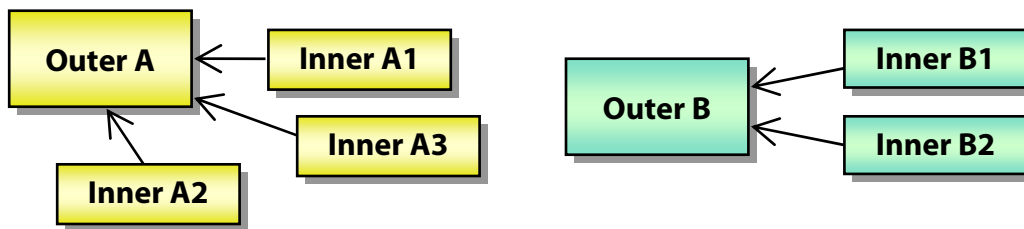
```
public class Scores
{
 ...
 private static class Record
 implements Comparable<Record>
 {
 ...
 }
}
```

- This is appropriate from a **scoping-and-visibility** perspective, because the record type is an internal encapsulation.
- **Scores** happens to be the only class in the application!
- But if it weren't, other classes would have no business seeing or using **Record**.
- Or, if it were useful for other classes to be able to add records to the data set, or receive them as method results, the class could be made **public**, and then referenced from outside as **Scores.Record**.

## Inner Classes

---

- An **inner class** is a nested class that is not marked **static**.
- In many ways it behaves as does a static class, and most of the rules apply that we've seen for static classes: visibility, scoping, name qualification, etc.
- But inner classes define a special relationship between their instances and those of the enclosing class.
  - Every **inner object** holds a reference to its **outer object**.



- This is a **unidirectional, many-to-one** relationship.
  - If the outer object needs a reference to one or more of its inner objects, you must capture that and store it explicitly.
  - An outer object can have multiple inner classes/objects.
  - An inner object can have only one outer object – although it could have a chain of outer object that belongs to another object, and so on. This is legal, but not common:



- Because inner classes are intended to relate specifically to instances of outer classes, they cannot define static members.
  - Instead, they rely on statics as defined by their outer classes.

## Instantiating Inner Classes

---

- You must instantiate inner classes with reference to a specific outer object.
- From within the class this usually doesn't look any different:

```
public class OuterClass
{
 private class InnerClass { ... }

 public void foo ()
 {
 InnerClass obj = new InnerClass ();
 }
}
```

– The outer object for **obj**, above, is implicitly **this**.

- From a static method, however – or from outside the defining class – a new syntax is needed to identify that outer object:

```
public static void baz ()
{
 OuterClass outer = new OuterClass ();
 InnerClass inner = outer.new InnerClass ();
}
```

– The object reference to the left of the dot identifies the outer object for the new instance of the inner class.



## Visibility to the Outer Class

---

- Code in an inner-class method can “see” its own members and the members of its outer class(es).
- This visibility is implemented by way of a new “this” reference.
  - Though you rarely need to use the symbol explicitly, it is written as **this\$0** – or **this\$1** to refer to an outer object’s outer object, etc.
  - So code such as the following ...

```
public class OuterClass
{
 private String name;

 private class InnerClass
 {
 private int x;

 public void foo (String one, int two)
 {
 name = one;
 x = two;
 }
 }
}
```

- ... expands to this during compilation:

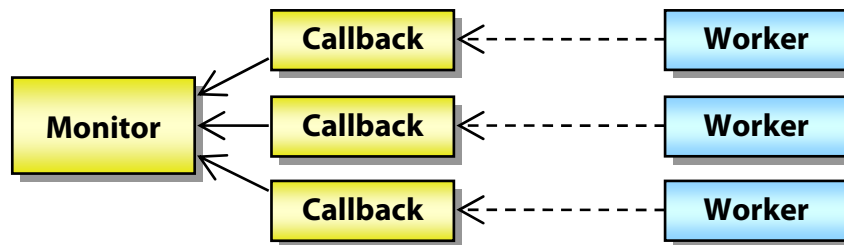
```
public void foo (String one, int two)
{
 this$0.name = one;
 this.x = two;
}
```

- Symbols defined in the inner class will hide identical ones defined on the outer class.
- In this case you can just write e.g. **this\$0.hiddenSymbol**.

## Using Inner Classes

---

- So inner classes offer two useful capabilities that combine nicely:
  - An inner class can specialize another type – by extending a class or by implementing an interface.
  - An inner object can see and operate upon the outer object.
- So a common use of inner classes is for handling event notifications from other classes or systems.
- This sort of **observer pattern** occurs in many contexts and at many scopes, from GUI programming to distributed systems.

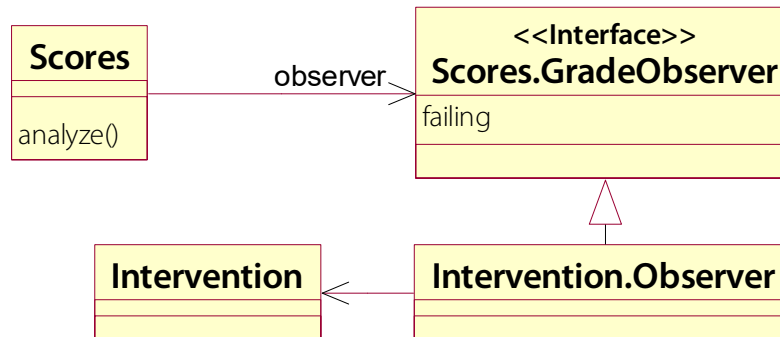


- An event source defines a class or **callback interface** through which it's willing to send notifications – as minute as a mouse movement, or as infrequent as a server status alarm.
- A primary object may want to listen for events or notifications from **multiple sources** – but it can't implement the same callback interface more than once, itself.
- It can instantiate many **inner objects** that implement the callback interface, and each one can update state on the outer object.

## Monitoring Scores

**EXAMPLE**

- In **Scores/Observer**, the Scores application has been re-organized to a more conventional component form.



- The analysis of score data now occurs in a method **analyze**, and this must be called on an instance of **Scores**.
- The caller passes score data to the method.
- The object supports an **observer** to which it will send notifications of failing grades by calling the **failing** method on a defined callback interface.
- The new class **Intervention** does not implement this interface itself, but defines an inner class that does – which allows it to monitor more than one **Scores** analysis, if necessary.

## Monitoring Scores

**EXAMPLE**

- See **Scores.java**: the **Record** class is still there, but has been declared **public**:

```
public class Scores
{
 public static class Record
 implements Comparable<Record>
 ...
```

- Here is the callback interface, and a property **observer** of this type:

```
public interface GradeObserver
{
 public void failing (String name, int score);
}
```

```
private GradeObserver observer;
```

- The **main** method now passes the initialized names-and-scores list to the **analyze** method on an instance of **Scores**.

```
public static void main (String[] args)
{
 List<Record> scores = new ArrayList<> (10);
 scores.add (new Record ("Suzie Q", 76));
 ...
 Scores worker = new Scores ();
 Intervention intervention =
 new Intervention (worker);
 worker.analyze (scores);
 intervention.sendLetters ();
}
```

- In the process, it creates an **Intervention** object and gives it a chance to observe the analysis, and when done it calls this object to produce its own report.

## Monitoring Scores

**EXAMPLE**

- In **Intervention.java** we see the outer and inner classes that receive notifications of failing grades:

```
public class Intervention
{
 private List<String> failingGrades =
 new ArrayList<> ();

 private class Observer
 implements Scores.GradeObserver
 {
 public void failing (String name, int score)
 {
 failingGrades.add (name);
 }
 }
}
```

- The **Intervention** constructor instantiates and registers an **Observer** with the **Scores** object:

```
public Intervention (Scores worker)
{
 worker.setObserver (new Observer ());
}
```

## Monitoring Scores

### EXAMPLE

- The **sendLetters** method doesn't actually do anything except print out the accumulated student names; the rest is //TODO ...

```
public void sendLetters ()
{
 if (failingGrades.size () != 0)
 System.out.println ("Failing grades:");
 for (String name : failingGrades)
 {
 System.out.println (" " + name);
 //TODO ...
 }
}
```

- Run the **Scores** application to see that just one letter would need to be sent ...

Student	Score	Grade
-----	-----	-----
Suzie Q	76	C
...		

```
Statistics:
 There were 2 As given.
 There were 2 Bs given.
 There were 3 Cs given.
 There were 2 Ds given.
 There were 1 Fs given.
```

The mean score was 75.8

```
Failing grades:
 Flea
```

## Local Classes

---

- You can define a class from within a method.

```
public class AnyClass
{
 public void foo ()
 {
 class LocalClass
 {
 public int x;
 public void calculate () { ... }
 }

 LocalClass helper = new LocalClass ();
 helper.calculate ();
 ...
 }
}
```

- This can be a useful technique when a method needs to execute some common functionality many times, but when that functionality would be useless outside the method.
  - Thus a separate class member is a less-attractive solution.
  - If the functionality truly requires an object-oriented solution – state and behavior encapsulated somehow – and is properly scoped to the method itself, this is a perfect technique.
  - These are rare cases, however.

## Anonymous Classes

---

- The remaining class usage is the **anonymous class** – so called because the technique is to define a class with no name.
- By way of a fairly disturbing syntax, you can define, instantiate and pass a new class as an argument, completely on the fly.

```
mainWindow.addWindowListener
(new java.awt.event.WindowAdapter ()
{
 @Override
 public void windowClosing
 (java.awt.event.WindowEvent ev)
 {
 System.exit (0);
 }
});
```

- The symbol following **new** is not the name of your class, but implicitly the name of a **base class** or **implemented interface**.
- After the opening and closing parentheses, you define the body of the class, more or less normally; but this is typically a single method implementing an interface method or overriding a base implementation.
- The resulting class is compiled to a separate file, whose name will be of the form **OuterClass\$N** where N is a unique number for the outer class.



## Using Anonymous Classes

---

- Anonymous classes had their initial heyday as a way of writing event handlers in GUI programming – especially with Java **AWT** (Abstract Windowing Toolkit) and **JFC** (Java Foundation Classes, a subset of which is a/k/a Swing).
  - A **window** class might have reference to many different controls: **buttons**, **text fields**, **list boxes**, and so on.
  - It couldn't implement all the different event callbacks effectively by itself, and so would use inner classes.
  - But to define the class in one place and then use it in another seemed awkward to many GUI programmers – especially those familiar with **JavaScript** and the use of **closures**.
  - Anonymous classes arguably fit the GUI-programming model more naturally, so that in one passage of code can be seen the control, the event, and the logic to handle the event.
- **Callback interfaces have proliferated in Java APIs over the years:**
  - The **Collections API** uses them for things like **sort criteria**.
  - Persistence frameworks use them to let you plug in object-specific logic such as how to map a **database row to an object**.
- **Inner and anonymous classes have turned out to be Java's best means of providing specifics to such APIs, in a style known as functional programming ...**
- **... at least until Java 8 provided new tools, which will be the focus of study in the following chapter.**

## Named vs. Anonymous

### EXAMPLE

- In **InnerClasses**, there are two applications, almost identical in the way they apply a custom comparator to the *sort* algorithm.
  - Rather than sorting alphabetically, they want to sort first from shortest string to longest, only resolving differences between same-length strings by alphabetical comparison.
- **src/StringSortNamed.java** does this by defining a static inner class **SortByLength**, and passing it to **sort**:

```
public class StringSortNamed
{
 public static class SortByLength
 implements Comparator<String>
 {
 ...
 }

 public static void main (String[] args)
 {
 List<String> strings =
 new ArrayList<String> ();
 Collections.addAll (strings,
 "alpha", "beta", "gamma", "delta", "epsilon");
 printList ("Start", strings);

 Collections.sort (strings);
 printList ("Sort", strings);

 Collections.sort
 (strings, new SortByLength ());
 printList ("Sort by length", strings);
 }
}
```

## Named vs. Anonymous

### EXAMPLE

- **StringSortAnonymous** pulls the inner-class definition into an anonymous class, created “on the fly” and used only for this single invocation of **sort**:

```
Collections.sort
(strings, new Comparator<String> ()
{
 public int compare (String one, String two)
 {
 ...
 }
});
printList ("Sort by length", strings);
```

- Run either application and see the same results:

**compile**

**run StringSortNamed**

```
Start: alpha beta gamma delta epsilon
Sort: alpha beta delta epsilon gamma
Sort by length: beta alpha delta gamma epsilon
```

**run StringSortAnonymous**

```
Start: alpha beta gamma delta epsilon
Sort: alpha beta delta epsilon gamma
Sort by length: beta alpha delta gamma epsilon
```

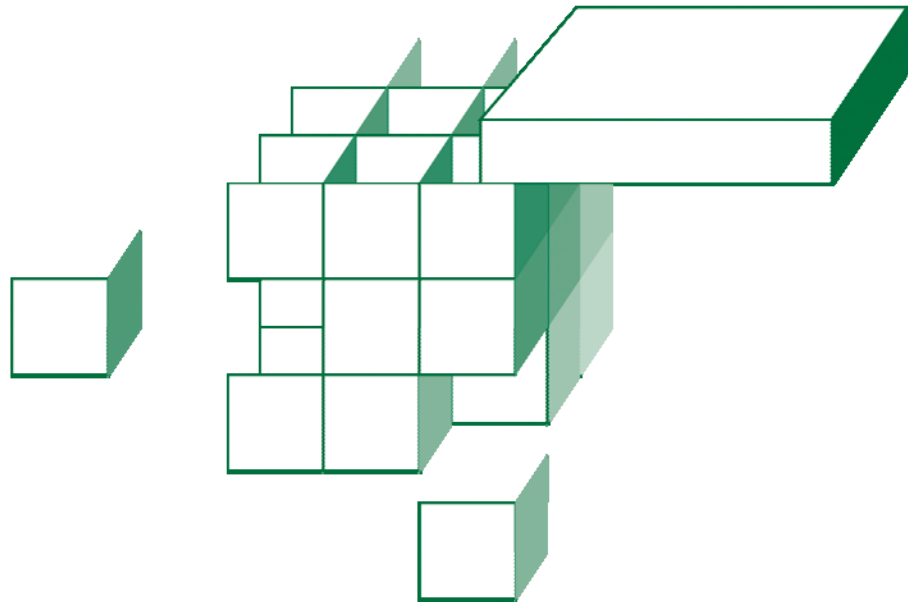
- This example is more to illustrate the difference in syntax than to address design issues.
- If this were a practical exercise, the static inner class would probably be the better design choice.
  - The sorting logic is sufficiently reusable that hacking it out in a one-use-only anonymous inner class seems short-sighted.

## SUMMARY

- **Static classes provide a convenient means of defining functionality which is only relevant to a particular class but which needs to be passed elsewhere in the system.**
- **Inner classes also provide a neat mechanism that marries the “inner object” to the “outer object” and allows it to act effectively on that outer object’s behalf.**
  - This means that the inner object must be created by the outer object – which is not usually a problem – or with reference to the outer object in external code.
- **Local classes offer potentially miniscule scopes for use and reuse of state and behavior; their use is rare.**
- **Anonymous let you define event-handling or plug-in logic at a limited scope and in a non-reusable way.**

# CHAPTER 13

## FUNCTIONAL PROGRAMMING



## OBJECTIVES

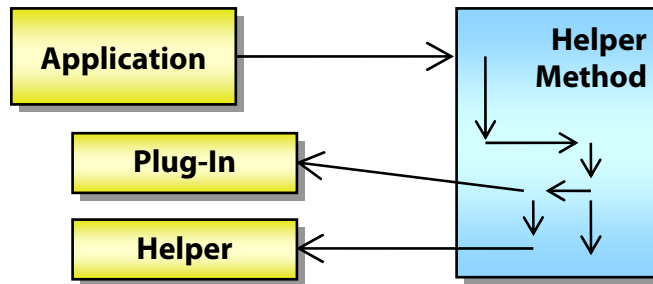
*After completing “Functional Programming,” you will be able to:*

- Describe functional programming, and identify means of passing behavior definitions between methods.
- Use inner classes for functional programming.
- Define and use functional interfaces.
- Write effective lambda expressions, and understand the subtleties of enclosing scope and visibility.
- Use method references effectively.
- Understand the significance of default methods.

## Utilities, Frameworks, and Callbacks

---

- In object-oriented development, reusable classes – utilities, frameworks, what-have-you – can offer methods or APIs to be called as needed by client code.
- Increasingly, the most effective utilities not only are called, but also call their client code back.



- Sophisticated processing can be carried out based on your provided **data** – such as record sets to analyze, modes in which to operate, and threshold values to observe.
- Then it often requires you to provide specific **behavior** as well – such as what to do with each of a set of calculated results, or how to map a value of one type to an appropriate value of another type.
- Various design approaches also call for this sort of behavior parameterization.
  - **Algorithmic programming** lets you separate the process of traversing a complex data structure from the actions to be taken – “for each ...” and “first that ...” sorts of plug-ins.
  - Design patterns such as **Strategy** and **Template Method** break more specific behaviors out via interfaces through which a more general-purpose implementation can make method calls.

## Utilities, Frameworks, and Callbacks

---

- The go-to technique in all of these contexts is the **callback** or **helper interface**, which often will have a single method.
- Examples from the Java Core API include:
  - The **Collections API** lets you define sort criteria through the **Comparable<T>** and **Comparator<T>** interfaces.
  - **Java Swing** defines the **ActionListener** interface to advise you of a button click in a desktop application, and **JavaServer Faces** defines a similar interface for actions in HTML forms.
  - A **Thread** is created with reference to a **Runnable**, which defines work for the thread in a **run** method.
- Examples from Java EE and open-source frameworks include:
  - All Java EE component standards – Servlets, JSF, JPA, web-services APIs, and more – define **lifecycle hooks** as ways to let the application plug in code to be run at specific times such as when a component has been created or is about to be destroyed.
  - The **Spring** framework uses callbacks liberally, for lifecycle events, filtering, and for example in supporting database components that use **JDBC**, by letting the application plug in relational-to-object and object-to-relational mapping methods.



## Passing Behavior as a Parameter

---

- Various programming languages, past and current, have found different ways in which to pass definitions of behavior as parameters to methods.
  - In **Smalltalk**, a block of code can be passed as an argument, verbatim.
  - In **C** we use the more pedestrian **function pointer**.
  - **C++ templates** can be used creatively to instantiate an algorithm on a particular class with a known method name.
  - **JavaScript**, **Groovy**, and many other languages offer the **closure**, and Java anonymous inner classes are closely patterned on JavaScript closures.
  - **C#** and other .NET languages support **lambda expressions**,
  - **Scala** has something that looks similar, known as the **anonymous function**.

## Passing Behavior as a Parameter

---

- Java has developed increasingly effective ways of passing behavior over the years, though until Java 8 none have been quite as fluid as the lambda expression.
  - **Interfaces** allow a consistent behavior to be defined and implemented in various classes, and instances of any of these can be passed to a given method that knows the interface.
  - The **Reflection API** is a quite sophisticated tool for dynamic invocation, and offers the closest thing Java has had to a **function pointer** – but it's code-intensive and prone to error at runtime.
  - **Inner classes** and especially **anonymous classes** are type-safe, flexible, and provide a solid programming model – something like **closures** in JavaScript or Groovy, though a bit clumsy by comparison.
- Java 8 brings new tools to the language:
  - **Functional interfaces**, which are essentially single-method interfaces for which there can be no question as to which method is being implemented – even if that method isn't named or formally overridden by conventional syntax
  - **Lambda expressions**, which are (often) brief expressions of a method implementation, from which the method and indeed an enclosing class can be extrapolated
  - **Method references**, which let you promote existing methods as implementations of functional interfaces

## Starting Point: Inner Classes

---

- The **inner class** provides a serviceable tool for implementing callback functionality.
  - The **outer class** can define as many inner classes as needed, implementing different interfaces or extending classes; and can instantiate each as many times as needed.
  - The **inner object** can see and **manipulate the state** elements of the **outer object** in response to method calls.
- Inner classes can be defined on the fly as **anonymous classes**, and in that usage they are something like **closures** in other languages.
  - The source of the expected callback and the code to be executed when the callback is invoked are all spelled out together.
  - Compare this JavaScript ...

```
myButton.onclick = function (ev)
{
 alert ("Pressed button!");
};
```

- ... to this Java Swing code:

```
myButton.addActionListener (new ActionListener ()
{
 public actionPerformed (ActionEvent ev)
 {
 JOptionPane.showMessageDialog
 (parent, "Pressed button!");
 }
});
```

- But inner classes are more verbose and clumsy than closures.

## Functional Interfaces

---

- As of Java 8, a **functional interface** is, simply, any interface that defines exactly one method.
- The annotation **@FunctionalInterface** may be applied to such an interface, but it is not necessary in order to make the interface a functional interface.
  - It is more like the **@Override** annotation: it is advice to the compiler, stating your intent as a programmer.
  - If a method annotated as a **@FunctionalInterface** doesn't abide the rules, the compiler will treat it as an error.
- What's the big deal about defining exactly one method?
- One of the shortcomings of inner classes for functional programming is that they still must be defined as classes, even though they are really just wrappers around single method implementations.
- Functional interfaces don't solve this problem, by themselves, but they do solve part of it.
  - They put us in a position to skip the class wrapping, or more accurately to let it be implied.
  - Now, we can start to think in terms of implementing a callback interface by implementing its one and only method – perhaps without all the trimmings and trappings of nested classes ...

## Functional Interfaces

**EXAMPLE**

- In **Functional/Step1** is an introductory example of a functional interface; see **src/cc/functional/Functional.java**:

```
public class Functional
{
 public static interface Callback
 {
 public void notifyMe (String event);
 }
}
```

- The outer class defines one method that takes the functional interface as one of its parameters:

```
public static void doSomethingAndNotifyMe
 (String thingToDo, Callback callback)
{
 callback.notifyMe (thingToDo);
}
```

- The **main** method exercises this system by way of a simple anonymous class:

```
public static void main (String[] args)
{
 doSomethingAndNotifyMe
 ("Hop", new Functional.Callback ()
 {
 public void notifyMe (String ev)
 {
 System.out.println
 ("Anonymous class received: " + ev);
 }
 });
}
```

## Built-In Functional Interfaces

---

- After writing a lot of callback interfaces and methods, you may well have noticed that there are only a handful of common parameter lists.
  - The method name may vary – but for the moment let's imagine that the method name is not quite so important (because soon it will be just about irrelevant).
  - The typical parameter lists are zero, one, two, or maybe three of the common types **String**, **int**, **long**, **double**, **boolean**, and maybe some specialized classes.
- So, another nice thing about functional interfaces is that the Core API now includes a whole package of built-in functional interfaces that cover a lot of these common cases.
- Yes, the fine folks at **java.lang.function** are ready to meet many of your callback needs ...

```
IntBinaryOperator.applyAsInt (int, int): int
IntUnaryOperator.applyAsInt (int): int
IntFunction<T>.apply (int): T
```

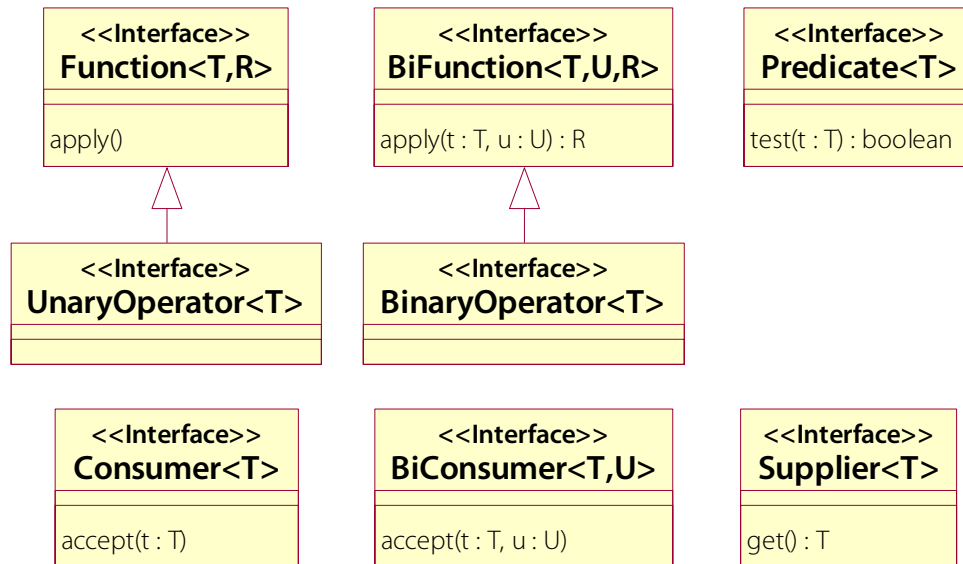
```
DoubleBinaryOperator.applyAsDouble
 (double, double) : double
DoubleUnaryOperator.applyAsDouble (double) : double
DoubleFunction<T>.apply (double) : T
```

```
BinaryOperator<T>.apply (T, T) : T
BiFunction<T,U,V>.apply (T, U) : V
```

- And, as we get into lambda expressions and method references, you'll see that the only thing that may not be a fit for you – the method name – will not matter!

## Getting Familiar with java.util.function

- The **java.lang.function** interfaces break down as follows – and we start to see that it's all about method signature:



- A **Function**'s method takes a parameter and returns a value. A **UnaryOperator** does, too, but restricts the parameter and return type to be the same, where they can be different for **Function**.
- A **BinaryFunction** takes two parameters and returns a value, and is restricted in type by **BinaryOperator**.
- **Consumer** and **BiConsumer** take parameters but return nothing.
- A **Supplier** takes no parameters and provides a return value.
- **Predicate** is a special **Function** that always returns **boolean**.
- As you get deeper into functional programming in Java, you'll be surprised how few interfaces of your own you'll need to define.
- For example our home-grown **Callback** interface is functionally equivalent to a **Consumer<String>**.

## Getting Familiar with `java.util.function`

---

- The interfaces shown on the previous page are the generic types.
- If any of the parameter or return types you need are primitives, these won't work for you – unless you are willing and able to use the boxing types such as **Integer** and **Double** as your type arguments, which in many cases is fine.
- But for cases where you really want to stick with primitives – maybe to avoid the costs of creating a lot of transient objects – there are solutions for you, too.
  - In fact the majority of interfaces in the package are interfaces built specially to handle primitive types **boolean**, **int**, **long**, and **double**.
  - We won't enumerate them all here, but if you hunt through **java.util.function**, you'll find the flavor of interface you need.
- Finally ... what about a function that takes no parameters and returns nothing?
  - There is no such functional interface in **java.util.function**.
  - An appropriate interface would be **Runnable**.

<<Interface>> <b>Runnable</b>
run()

- This has been in the Java Core API since Java 1.0, and now is explicitly annotated as a functional interface.



## A Configurable Seller

**EXAMPLE**

- In **Cars/List/Step5**, a new **Seller** implementation is found in **src/cc/cars/sales/Configurable.java**.

```
public class Configurable implements Seller {
```

- We implement each of the interface's methods by calling an implementation of a functional interface, which can be passed to the constructor and held as an instance variable:
- For asking price, we use a **DoubleUnaryOperator** – so we'll pass a **double** (sticker price), and get a **double** back (asking price).
- For the counter-offer logic, we define a custom interface, because we want to pass a lot of information to the pluggable implementation:

```
@FunctionalInterface
public interface CounterOfferCalculator {
 public double calculateCounterOffer
 (double sticker, double myOffer,
 double yourOffer, int rounds);
}
```

- We keep track of these in instance variables – along with the car we're supposed to sell, and negotiation state ...

```
private Car car;
private DoubleUnaryOperator askingLogic;
private CounterOfferCalculator counterLogic;

private double myOffer;
private int rounds;
```

## A Configurable Seller

**EXAMPLE**

- ... and let the caller initialize them via the constructor.

```
public Configurable(Car car,
 DoubleUnaryOperator askingLogic,
 CounterOfferCalculator counterLogic) {
 this.car = car;
 this.askingLogic = askingLogic;
 this.counterLogic = counterLogic;
}
```

- Then we implement the interface, calling the method on the appropriate implementation.

```
public double getAskingPrice() {
 return myOffer = limit
 (askingLogic.applyAsDouble(car.getPrice()));
}

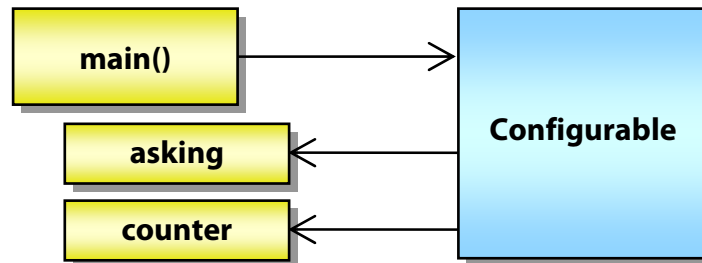
public double willSellAt(double yourOffer) {
 if (yourOffer >= myOffer) {
 return yourOffer;
 }
 return myOffer = limit (counterLogic
 .calculateCounterOffer(car.getPrice(),
 myOffer, yourOffer, ++rounds));
}
```

- We take care of some things so the pluggable implementation doesn't need to worry about them:
  - Recognizing a sale and closing the deal
  - Keeping track of the offers for next time.
  - Limiting the offered prices for used cars

## A Configurable Seller

**EXAMPLE**

- We could use this class to experiment with different price-negotiation logic, plugging in different formulas as anonymous classes.



```
Seller test1 = new Configurable(
 new DoubleUnaryFunction() {
 public double applyAsDouble(double sticker) {
 Return sticker * .95;
 }
 },
 new CounterOfferCalculator() {
 public double calculateCounterOffer
 (double sticker, double myOffer,
 double yourOffer, int rounds) {
 return mine - 100;
 }
 }
));
```

- But, again, we see how unwieldy these anonymous classes are – and local classes, inner classes, or top-level classes wouldn't be any better.

## Lambda Expressions

---

- Java 8 also introduces the **lambda expression** to the language.
- A lambda expression defines a method implementation – without either the traditional method signature or a class in which to define the method.
  - If you are calling a method that takes a parameter whose type is a **functional interface**, such as this ...

```
public void someMethod (IntUnaryOperator op);
```

- ... then instead of this ...

```
public class CallbackImpl
 implements Callback
{
 public int operateOn (int x)
 {
 return x + 1;
 }
}
someMethod (new CallbackImpl ());
```

- ... you might just write this:

```
someMethod (x -> x + 1);
```

- So there's our big step forward: all of the class and method framing around the implementation simply falls away.
  - And this is possible because the interface, being a functional interface, leaves no doubt as to which method you mean to implement with your expression.

## Lambda Expressions

---

- So, at first blush, the lambda expression appears to be a much easier way to write an anonymous inner class.
- It is that; but it offers another, less obvious advantage, which is that it does not define its own **this** and **super** references, the way instance methods of a class do.
- Why is that a big deal?
- It means that the implementation of the lambda expression – known as the **body** – **shares the enclosing scope**.
  - Now this is a whole lot more like a closure as known in other languages, and it has huge advantages in many event-handling and callback contexts.
  - Often, the job is to register a callback with an object, for some **future invocation**, that will trigger logic based on **present state**.
  - If you've ever written an inner class to implement a callback, and then realized that it needs to carry state from the moment of its creation to the moment at which it is invoked ... and then given it fields and a constructor just to carry that information around ... well, you're going to like lambda expressions!

## Lambda Expressions

**EXAMPLE**

- See **Functional/Step2**; in **src/cc/functional/Functional.java** the **Callback** interface has been annotated:

```
@FunctionalInterface
public static interface Callback
{
 public void notifyMe (String event);
}
```

- Again, this annotation doesn't change the possible use of the interface, one way or the other.
- But it's good documentation since we're now implementing the interface with a lambda expression, and it protects against any later feature-creep that would compromise those lambda uses.

## Lambda Expressions

**EXAMPLE**

- In **main**, after using an anonymous class, we call the same method with a lambda expression implementing the callback.

```
doSomethingAndNotifyMe ("Skip",
 ev -> System.out.println
 ("Lambda 1 received: " + ev));
```

- Though they are not often reused, note that it is possible to capture the result of a lambda expression – which, remember, is an instance of an anonymous class that implements the expected functional interface – and use it later:

```
Callback impl1 = ev -> System.out.println
 ("Lambda 2 received: " + ev);
doSomethingAndNotifyMe ("Jump", impl1);
```

- Run this version of the application and see that both lambdas are invoked:

```
Anonymous class received: Hop
Lambda 1 received: Skip
Lambda 2 received: Jump
```

## Grammar for Lambda Parameter Lists

---

- The general form for a lambda expression is this:

*(type1 param1[, typeN paramN]) -> body*

- But just about everything to the left of the `->` operator – the **parameter list** – is optional.
  - Parentheses are unnecessary for single-parameter methods.
  - Type identifiers are optional.
- Of course you can't have nothing to the left of the `->` operator: one or more of the optional pieces must be there.
- Practical options are more like this:

`() -> ...`

`(x) -> ...`

`x -> ...`

`(int x) -> ...`

`(MyClass x) -> ...`

`(x, y) -> ...`

`(String x, int y) -> ...`



## Grammar for Lambda Bodies

---

- To the right of the `->` operator lies the lambda expression's **body**.
  - The body may use all **method parameters**, as named in the parameter list, and all symbols available in the **enclosing scope**.
  - It must return a value consistent with the expected return type of the method defined on the corresponding functional interface.
- The body is either a single expression or a block of code.
  - If an expression, this expression is considered to be the **return** value of the generated method – unless the expression evaluates to **void**, in which case the method must also return **void**.

```
x -> -x
() -> ++count // modifies and returns the number
(x) -> System.out.println ("Value is " + x);
(x, y) -> String.format
 ("There were %d instances of %d.", y, x)
```

- With blocks, though just about anything is legal, we generally keep it brief.

```
x -> { trackingX = x; return x + 1; }
() -> { ++count; } // modifies, but returns nothing
(x, y) -> { myMap.put (x, y);
 return myMap.keySet ().size (); }
```

- Complex processing probably merits a traditional method and/or class. Consider **method references**, which we'll cover later in the chapter.

## A Configurable Seller

**EXAMPLE**

- In **Cars/List/Step5**, the **Dealership**'s **sellCar** method is rewritten to use the **Configurable** seller in lieu of **Simple**, **Standard**, and **HardNosed** classes (which have been removed from the project).
  - It replaces each with a **Configurable** instance, plugging in equivalent logic as lambda expressions.
  - Notice that, while lambda expressions are often written as method parameters, they can be captured as variables, too.
  - All three of those classes started with a 10% discount, so we can set that once:

```
DoubleUnaryOperator askingLogic =
 price -> price * .9;
```

## A Configurable Seller

**EXAMPLE**

- Then we can set a variable to refer to the appropriate counter-offer implementation ...

```
Configurable.CounterOfferCalculator counterLogic;
switch(type)
{
case "Simple":
 counterLogic = (price, mine, yours, round) ->
 yours > price * .8 ? yours : mine;
 break;

case "Standard":
 counterLogic = (price, mine, yours, round) ->
 mine - (round <= 3 ? (mine - Math.max
 (yours, price * 1.6 - mine)) * .3 : 0);
 break;

case "HardNosed":
 counterLogic = (price, mine, yours, round) ->
 mine - (yours > price * .85
 ? (mine - yours) * .2 : 0);
 break;

default:
 LOG.severe ("Unrecognized seller type...");
 return null;
}
```

- ... and use the two variables to initialize the **Configurable** seller that will behave as desired:

```
return new Configurable
 (car, askingLogic, counterLogic);
```

- We lose the logging from those classes, but not much else.

## Challenges in Writing Lambda Expressions

---

- In one of its many triumphs of understatement, regarding lambda expressions the Java language specification says,  
*The syntax has some parsing challenges.*<sup>2</sup>
- In other words there's a lot of inference involved in figuring out your lambda expression and turning it into an implementation of the expected functional interface.
- Mostly this is a problem for compiler vendors! But it shows up in the daily life of the application developer in the form of some truly inscrutable error messages.
- Especially, when your lambda doesn't fit to the expected method signature, the best the compiler can usually do is to tell you something like "there is no overload of the method that takes a () -> {} and a (Object) -> {}."
  - The problem in this case may actually be with the expected method signatures: does your **parameter list** line up correctly with the expected method signature? Are you **returning** something, and if so is it of the appropriate type?
  - But it can also turn out to be **unrecognized symbols** or even **mis-matched parentheses**.
  - Look for confusing **type conversions**.
  - Try adding **explicit types** to your parameters. It may not solve the problem, but often it will shake loose a clearer error message!

---

<sup>2</sup> Gosling et. al., *The Java Language Specification, Java SE 8 Edition*, section 15.27, p. 597.

## Functional Interfaces in the Collections API

---

- Functional programming has been integrated into much of the Core API, and is especially accessible as part of the Collections API.

- **Collection** offers a `forEach` method, taking a **Consumer<T>**.

```
names.forEach(n -> System.out.println(n));
```

- It's actually a **Consumer<? Super T>**, and we'll elide the wildcard uses in the API for simplicity in this text.
- And it's actually defined on **Iterable<T>**, and it's part of how **for** loops work!

- The venerable **removeAll** methods are joined by a new method **removeIf**, which takes a **Predicate<T>** as a parameter.

```
numbers.removeIf(n -> n < 0);
```

- **Comparator<T>** is a functional interface, so a lambda can be used anywhere a **Comparator** is expected.

```
Collections.sort(students,
 (a,b) -> a.getScore() - b.getScore());
```

- Even better, there's a utility method **comparing** that takes a **Function<T,U>**, which lets you simply identify a property of an object that should be used as the sorting key.

```
Collections.sort(students,
 Comparator.comparing(s -> s.getScore()));
```

## Sorting and Filtering Cars

**LAB 13A**

**Suggested time: 45 minutes**

In this lab you will explore the use of functional interfaces in the Collections API, and write some utility code of your own that takes functional interfaces as parameters.

Detailed instructions are found at the end of the chapter.

## Scope and Visibility

---

- The body represents the implementation of what will be a generated class that implements the functional interface expected, based on the context in which the lambda expression appears.
- But – even if you define the body as a block of code – there is essentially no **this** reference, and this leaves you free to refer to the enclosing scopes in your code.
  - This includes fields of the enclosing class(es), and parameters of the enclosing method.
  - Even local variables defined before the use of the lambda expression are fair game.
- There are restrictions on parameters and variables, having to do with deferred execution of the expression.
  - Essentially, you **can't modify** values on the **stack** from within a lambda expression.
  - If the generated method were to be executed **synchronously** with your method call, there would be no problem.
  - But in the case of **deferred execution**, the thing being modified might no longer exist! And the compiler can't be sure of the timing of the callback invocation(s).

## Deferred Execution

---

- Consider the code below:

```
public void foo (int x)
{
 bar (() -> ++x);
 List<Integer> list =
 Collections.singletonList (x);
 new DeferredExecutor ((y) -> list.add (y));
}
```

- There are a few possible issues here. For one thing, we're not entirely sure what value will be placed as the initial value in **list**: will it be **x**, or **x + 1**?
  - The answer depends on whether **bar** calls the generated method immediately or at some later time.
- Much worse, if **bar** defers execution of our expression, then **x** may well have been popped off of the call stack.
  - There would be no value **x** to read, and no place to store **x + 1**.
- So, the rule: you may read locals, but you may not modify them.
  - The generated method will work with a **capture** of the value at the time the lambda is evaluated – so you can read it, even later on.
  - But if you try to modify a variable or parameter, the compiler will complain that the target is “not final or effectively final.”
  - So it's not insisting that you declare the variable to be **final**, but in the lambda it will treat it as if it is.
  - Note that you may modify object state through a captured object reference; you just can't re-assign the reference. That's why the **list.add** call in the second lambda expression is okay.



## Using a Generic Data Browser

**LAB 13B**

**Suggested time: 30 minutes**

In this lab you will write client code to use an existing **Browser** class that manages the user interface for page-by-page browsing of a set of information. The **Browser** doesn't know what the data is or how to paginate it, so it offers a set of callbacks:

- Four callbacks for next-, previous-, first-, and last-page commands
- A callback to show the data for the current page
- A callback to get a label or heading for the current page

So, your client code is expected to be stateful, to the extent that it knows what is “current” and how to move around the data set. Then it must also provide the logic to display and label a given page.

You'll work through a couple of implementations of the client, so that you can experiment with different styles and strategies, and see what works and what doesn't.

Detailed instructions are found at the end of the chapter.

## Method References

---

- Java 8 offers another way to implement a functional interface, which is to use or reuse an existing method.
- The new syntax expresses a **method reference** by use of a double-colon operator `::` with a class or object to the left and a method name to the right.
- Some more specific options for the left-hand operand:

- A simple class name

```
MyClass::myMethod
String::format
```

- An object reference

```
myObject::myMethod
```

- An expression that evaluates to an object reference

```
foo (x)::toString
Collections.singleton (x)::addAll
(date != null ? date : new Date ())::getTime
```

- The **super** keyword, in order to reference a superclass implementation in favor of a derived-class implementation

```
super::equals
```

- The right-hand operand is simply the name of the method.
- There's more to the grammar, and we'll dig further in a moment.

## Using Method References

---

- Supply a method reference anywhere you might use a lambda expression – which is to say, anywhere a functional interface is required.

```
doSomethingAndNotifyMe (System.out::println);
```

```
table.combine (Math::addExact);
table.combineAndTransform
 (Math::addExact, Math::abs);
```

```
Consumer<String> consumer = PrintStream::print;
```

- Using a given method by reference is equivalent to calling it explicitly from within a lambda expression:

```
doSomethingAndNotifyMe
 ((x) -> System.out.println (x));
```

- It follows that, if you find yourself writing a lambda expression such as the above, a simple method reference is probably the better choice.
- Both techniques involve a lot of type inference, and can be tricky to get right.
- But once resolved a method reference is the simpler, clearer, more type-safe, and more maintainable option.

## Method References

**EXAMPLE**

- We've added a method and some method references to **Functional/Step3**. See **src/cc/functional/Functional.java**.
  - The new method is static, and takes a single **String** parameter:

```
public static void notificationSink
 (String event)
{
 System.out.println
 ("Static method received: " + event);
}
```
  - At the bottom of the **main** method, we invoke **doSomethingAndNotifyMe** two more times, passing the method reference each time to satisfy the functional-interface requirement:

```
doSomethingAndNotifyMe
 ("Skate", Functional::notificationSink);

Callback impl2 = Functional::notificationSink;
doSomethingAndNotifyMe ("Pass", impl2);
```
  - As with lambdas, we show that the method reference really does evaluate to something, and can be captured by reference to the functional interface.
- **Run the application and see results as follows:**

```
Anonymous class received: Hop
Lambda 1 received: Skip
Lambda 2 received: Jump
Static method received: Skate
Static method received: Pass
```

## Type Arguments in Method References

---

- When working with generics, you can include type arguments in your method references.
- Identify a parameterized type as the class of a method reference, using the usual syntax:

`MyGeneric<String>::myMethod`

- You may not use wildcards.

- Identify a parameterized method by a less familiar syntax in which the type arguments are stated prior to the method name:

`MyClass::<String,Integer>convert`

- Don't mistake this second technique for a means of **disambiguating overloads** of a method.
- The traditional method-signature matching happens based on the expected functional interface and its method.
- This syntax is meant specifically for parameterized methods, to let you indicate what should replace "type T."

## Identifying Instance Methods by Class

---

- There is one quite surprising trick you can play: you can identify an instance method using a class name.

`List<String>::add`

- Generally, as we've seen, it's either **Class::staticMethod** or **object::instanceMethod**.
  - In this special case, the method you identify must have **one fewer parameter** than the functional method you are matching.
  - The idea is that the first parameter on the functional method will be treated as the **target** on which the instance method will be invoked.
  - Then, any remaining parameters will be passed to the referenced method.
- Let's say you need to implement a callback whose type is **BiConsumer<List,String>**.
    - You want to cause the second parameter to be added to the first.
    - Except for this special syntax, there's no way to do this – you would need to write a lambda expression such as  
`(list, string) -> { list.add (string); }`
    - This little trick with method references lets you use the method reference shown above, instead.

## Creational Methods

---

- Finally, there are some specialized uses of the method-reference operator that allow you to direct object creation.
- The simplest of these places the **new** keyword to the right:

```
Supplier<MyClass> factory = MyClass::new;
Supplier<TreeSet<Double>> treeFactory =
 TreeSet<Double>::new
```

- The left-hand operand must be a **class identifier**.
- It must not be **abstract**, and it must not be an **enum**.
- The constructor(s) of the identified class are searched for a match to the **parameter list** of the functional method.

```
Function<Car, Seller> sellerFactory = Standard::new;
```

- You can do this with array types, too:

```
double[]::new
MyClass[]::new
Runnable[]::new
```

- Notice that abstract types are okay here, because you're not expecting to create the objects – just an array of references to them.
- So the **Runnable[]::new** example would create an array of references to **Runnable** – which might refer to objects of any concrete subtype of **Runnable**.

## Null Checks, Type Identification, and Casting

---

- Some quite useful methods are available in the Core API, but not all that well publicized, and they're especially handy in functional programming
- We'll show each by comparison to the lambda expression you'd be tempted to write otherwise ...

– Testing for null:

<code>x -&gt; x == null</code>	<code>x -&gt; x != null</code>
<code>Objects::isNull</code>	<code>Objects::nonNull</code>

– Testing for type:

<code>x -&gt; x instanceof MyClass</code>
<code>MyClass.class::isInstance</code>

– Casting:

<code>x -&gt; (MyClass) x</code>
<code>MyClass.class::cast</code>

- Not to be forgotten, either: if you want to express a predicate that's true if the given value is equal to an expected value, you can refer to the **equals** method:

<code>x -&gt; x.equals("Hello")</code>
<code>"Hello"::equals</code>



## Sorting and Filtering Parts

**LAB 13C**

**Suggested time: 30 minutes**

In this lab you will add to your work from the first lab in the chapter, working with the parts inventory and exploring the use of method references. You'll also see that you can refactor some of that prior lab's code from lambdas to method references.

Detailed instructions are found at the end of the chapter.

## Default Methods

---

- A **default method** is a method that ...
  - Is defined on an **interface** – not on either a concrete or abstract class,
  - Is marked with the keyword **default**, and
  - Is **implemented**!
- Yes, interfaces can now implement methods. Let that one sink in for a moment.
- A default method is often implemented in terms of other, non-default methods – much the way an abstract class might implement a concrete method in terms of one or more abstract ones, letting the derived types fill in the blanks.
- Default methods can be overridden, just like concrete methods.
- A default method does not count for purposes of identifying an interface as being **functional**.
  - So a functional interface has **exactly one non-default** method and zero-to-many **default** methods.
  - In fact this is a big part of the advantage, because the caller of a functional interface may enjoy a relatively rich interface of several useful methods, while the implementation of the interface can implement just one – meaning we still get to use lambda expressions and method references.

## Default Methods

---

- The decision to add default methods to the language has been controversial.
- The rationale has to do with functional interfaces, and also with backward compatibility.
  - If we want to add **new functionality** to an interface model in a new version of Java – as they very much wanted to do in Java 8 – we're a bit stuck, because that means adding **new methods**.
  - **Old implementations** of that interface would no longer be valid.
  - By adding a **default method** we can avoid this trap: all existing implementations get the benefit of the default implementation, and new implementations can **override** as necessary.
- There are concerns that the addition may be destabilizing.
  - For one thing, default methods for the first time introduce the mixed blessing of **multiple inheritance** to Java.
  - What if a class implements **two interfaces**, or **a class and an interface**, with competing implementations of the same method?
  - Now the derived class must **disambiguate** between the two available implementations, or **override** the method itself.
  - More generally this language features seems to blur the line between **interface** and **abstract class** – for each of which there have always been clear rules of conduct.
- Implementing your own default methods is an advanced practice, and out of our scope.
- We will take a quick look at how default methods behave ...

## Default Methods

**EXAMPLE**

- In **Functional/Step4** there is a new default method on the **Callback** interface –see **src/cc/functional/Functional.java**:

```
@FunctionalInterface
public static interface Callback
{
 public void notifyMe (String event);

 public default void notifyWithTimestamp
 (String event)
 {
 notifyMe
 (LocalDateTime.now() + ": " + event);
 }
}
```

- On the enclosing class, a new method takes advantage of the default method on the callback:

```
public static void
doSomethingAndNotifyMeWithATimestamp
 (String thingToDo, Callback callback)
{
 callback.notifyWithTimestamp (thingToDo);
}
```

## Default Methods

**EXAMPLE**

- Note that a caller to the new method, which invokes the default, can still provide an implementation in lambda or method-reference form – because the default is implemented in terms of the functional method:

```
doSomethingAndNotifyMeWithATimestamp ("Shoot",
 ev -> System.out.println
 ("Lambda 3 received: " + ev));
doSomethingAndNotifyMeWithATimestamp
 ("Shoot", impl2);
```

- Run the application and see output as follows:

```
Anonymous class received: Hop
Lambda 1 received: Skip
Lambda 2 received: Jump
Static method received: Skate
Static method received: Pass
Lambda 3 received: 2014-10-17T11:35:47.977: Shoot
Static method received:
 2014-10-17T11:35:47.977: Shoot
```

## Stack Traces

**EXAMPLE**

- A final example in **Functional/Step5** gives a little insight into the impact of lambda expressions and method references at runtime.
- Another “referenceable” method has been added to **src/cc/functional/Functional.java** that shows a stack trace:

```
public static void showStackTrace (String path)
{
 System.out.println
 ("Calling via " + path + ":");
 new Throwable().printStackTrace ();
}
```

- We invoke it, both by reference and via a lambda expression:

```
doSomethingAndNotifyMe ("a static method",
 Functional::showStackTrace);
doSomethingAndNotifyMe ("a lambda expression",
 ev -> Functional.showStackTrace (ev));
```

## Stack Traces

**EXAMPLE**

- Run and note the generated classes in both stack traces:

```
Anonymous class received: Hop
Lambda 1 received: Skip
Lambda 2 received: Jump
Static method received: Skate
Static method received: Pass
Lambda 3 received: 2014-10-17T11:36:27.573: Shoot
Static method received:
 2014-10-17T11:36:27.588: Shoot
```

Calling via a static method:

```
java.lang.Throwable
 at cc.functional.Functional.showStackTrace
 (Functional.java:82)
 at cc.functional.Functional$$Lambda$10
 /951007336.notifyMe(Unknown Source)
 at cc.functional.Functional
 .doSomethingAndNotifyMe(Functional.java:51)
 at cc.functional.Functional.main
 (Functional.java:119)
```

Calling via a lambda expression:

```
java.lang.Throwable
 at cc.functional.Functional.showStackTrace
 (Functional.java:82)
 at cc.functional.Functional.lambda$3
 (Functional.java:122)
 at cc.functional.Functional$$Lambda$11
 /1528902577.notifyMe(Unknown Source)
 at cc.functional.Functional
 .doSomethingAndNotifyMe(Functional.java:51)
 at cc.functional.Functional.main
 (Functional.java:121)
```

## SUMMARY

- Java has been edging toward functional programming ever since the introduction of inner classes in (whew) Java 1.1.
- Java 8 leaps into modern functional programming with both feet.
- The key challenge to enabling functional programming in Java is to provide a means of treating a method or implementation as a complete class – and the key solution is the functional interface.
- Functional interfaces allow both lambda expressions and method references to be supplied where an object reference is expected – thus freeing the client programmer from the drudgework of building class and method framing around the basic logic.
- Lambda expressions have the additional capability to capture the enclosing scope of a method call for purposes of implementing a callback – even if invocation of that callback is deferred.
- The default method is a somewhat controversial feature that allows familiar interfaces to acquire new methods without breaking backward compatibility – and new functional interfaces to carry helper functions.



## Sorting and Filtering Cars

**LAB 13A**

In this lab you will explore the use of functional interfaces in the Collections API, and write some utility code of your own that takes functional interfaces as parameters.

**Lab project:** Cars/**List/Step5**

**Answer project(s):** Cars/**List/Step6**

**Files:** \* to be created  
**src/cc/cars/Inventory.java**

### Instructions:

1. Open **Inventory.java** and see that there is a helper method **show** **that prints a line describing a given Car**, and a little starter code in a **main** method.
2. Let's start by creating a second helper method, **showAll**, that can print out a report on a collection of cars. Declare this method to be **public** and **static**, to return nothing, and to take a string label and a **Collection<Car>** as its two parameters.
3. In the method body, print a line of hyphens, and then the given label on another line, and then a blank line.
4. Now, you could use a simple **for** loop to print each car in the given collection. But let's take a functional approach: call **forEach** on the collection, and pass a lambda expression that takes a single parameter **c** and calls **show**, passing that car.
5. Then print another blank line.
6. In **main**, call **showAll** and pass "New cars" and **cars**.
7. Run the class as a Java application now, and you should see something like this:

```

New cars

ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
BA0091: 2014 Ford Taurus (Black) $32,499.99
HJ5599: 2015 Saab 9000 (Pearl) $34,498.99
ME3278: 2014 Honda Accord (Red) $29,999.99
UI4456: 2014 Volkswagen Jetta TDI (Green) $23,899.99
WE9394: 2015 Kia Sonata (White) $21,999.99
XY1234: 2015 Ford F-150 (Black) $28,999.99
IM3110: 2014 Ford Escape Hybrid (Silver) $23,999.99
PU4128: 2015 Audi A3 (Blue) $39,999.99
```

8. Call **showAll** again, passing "Used cars" and **usedCars**, and test that out as well.

**Sorting and Filtering Cars****LAB 13A**

9. Now, sort the list of new cars, using a custom comparator:

```
Collections.sort(cars,
 (a,b) -> Double.compare(a.getPrice(), b.getPrice()));
```

10. Call **showAll** again, with the label “New cars by price” and the sorted **cars** list. Test the code at this point.

11. Now sort the list again, this time by model year, using the **Comparator.comparing** utility method:

```
Comparator<Car> byYear = Comparator.comparing(c -> c.getYear());
```

12. Show again, with an appropriate label. Test again.

13. Sorting by model year is not too satisfying, since there are a lot of cars with the same year. Let’s sort by year and then price. This will require more complex logic than would usually go into a lambda expression, but it is possible, and we’ll go this route just to explore the grammar:

```
Collections.sort(cars, (a,b) -> {
 int result = Integer.compare(a.getYear(), b.getYear());
 if (result == 0) {
 result = Double.compare(a.getPrice(), b.getPrice());
 }
 return result;
});
```

14. Show again, with another, distinct label. Run the program again.

15. Now, let’s say that you want to filter out all of the Fords from the new-cars inventory, before sorting and printing. Call **cars.removeIf**, passing a predicate that will return **true** for any car with “Ford” as its **make** property.

16. Test again, and see that you’re down to seven new cars:

-----

New cars by year and price

```
UI4456: 2014 Volkswagen Jetta TDI (Green) $23,899.99
ME3278: 2014 Honda Accord (Red) $29,999.99
WE9394: 2015 Kia Sonata (White) $21,999.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
ED9876: 2015 Toyota Prius (Silver) $28,998.99
HJ5599: 2015 Saab 9000 (Pearl) $34,498.99
PU4128: 2015 Audi A3 (Blue) $39,999.99
```

## Sorting and Filtering Cars

## LAB 13A

17. What if we want to move those three Fords over to a new collection, and then remove them from the main one? There is no **addIf**, or **getIf**, anything like that on **Collection**. So, we'll write our own! Define a **public, static** method **extractIf** that takes a collection of cars and a **Predicate<Car>**.
18. Implement the method to create a new **ArrayList** of cars, to loop through the given collection, and to add any car for which a call to the predicate's **test** method returns **true**. Then return the compiled list.
19. In the **main** method, before your call to **removeIf**, call **extractIf**, passing **cars** and the same lambda expression that you pass to **removeIf**. Capture the results in a local variable **fords**.
20. Call **showAll** to print the list of "Fords" before moving on to remove them and sort the main collection.
21. Test and see that this is working.
22. Now, you've probably noticed that you're repeating yourself a bit here: it's just the lambda expression, not a lot of code, but still it would be nice to define that just once. And you can do that: before calls to **extractIf** and **removeIf**, define a variable **isAFord**, of type **Predicate<Car>**, and initialize it to your lambda expression.
23. Use **isAFord** as the predicate parameter in your calls to **extractIf** and **removeIf**.

### Optional Steps

24. Define a second method, **stringify**, that takes a collection of cars and a **Function<Car,String>** called **transformation**. This will work a lot like **extractIf**, but instead of adding to the new list conditionally, it will always add something: not the original object, but the results of transforming it to a string, by calling **apply** on the supplied function.
25. Use your new method to print a list of the VINs of all new cars. Note that, after calling **stringify**, you can directly call **forEach** on the returned list, passing a lambda that prints the string on a line.
26. Do the same thing, with just a different lambda expression, to print a list of all of the names of the cars – that is, call **getShortName** on each, instead of **getVIN**.

## Using a Generic Data Browser

**LAB 13B**

In this lab you will write client code to use an existing **Browser** class that manages the user interface for page-by-page browsing of a set of information. The **Browser** doesn't know what the data is or how to paginate it, so it offers a set of callbacks:

- Four callbacks for next-, previous-, first-, and last-page commands
- A callback to show the data for the current page
- A callback to get a label or heading for the current page

So, your client code is expected to be stateful, to the extent that it knows what is “current” and how to move around the data set. Then it must also provide the logic to display and label a given page.

You'll work through a couple of implementations of the client, so that you can experiment with different styles and strategies, and see what works and what doesn't.

**Lab project:** **Browser/Step1**

**Answer project(s):** **Browser/Step2** (intermediate)  
**Browser/Step3** (final)

**Files:** \* to be created  
**src/cc/tools/Browser.java**  
**src/cc/tools/Pages.java**

### Instructions:

1. Open **Browser.java** and see the constructor, which takes a list of six functional-interface parameters.

```
public Browser (Runnable next, Runnable previous, Runnable first,
 Runnable last, Runnable show, Supplier<String> getLabel)
```

Most of what it needs to tell us as the user browses our data is in the nature of a simple command: no parameters, and it doesn't need anything back from us, but just fires a notification to us so we can do as we please. For those first five, the parameter type is **Runnable**. Then it needs us to provide a string as a heading for the current page, and for that it requests a **Supplier<String>**.

**Using a Generic Data Browser****LAB 13B**

2. Open **Pages.java** and see that there's very little here thus far: just the constants **LENGTH** and **PAGES** setting the dimensions of the data we will be providing to the user.
3. Declare an **int** variable **offset**, which will start at **1** and then will move around as the user browses.
4. Create a new **Browser** called **browser**. Start by providing **null** for all six arguments.
5. Call **browser.browse**, passing **PAGES** to set the limit of page numbers the browser will observe as the user moves around.
6. Now, this compiles, but of course it won't get far at runtime. It's time to fill in logic for the commands and other callbacks. Start with the handler for "next page," replacing the **null** for the first argument to the **Browser** constructor with an expression like this:

```
() -> offset += LENGTH
```

7. Hmm ... this doesn't seem to fly. The error message indicates that – as we discussed – you can't modify a local variable. In our case, the user interactions are synchronous with the call to **browse**, and so the **offset** variable would abide on the stack. But in the general case it's not safe to make a local variable available for modification like this.
8. Okay, we need a new strategy. Get rid of **offset**, and instead, at the top of the method, create a local class **Holder**, with a single, **public int** field called **value**.
9. Create an instance of the class called **holder**.
10. Set **holder.value** to **1**.

11. Change your first lambda expression to:

```
() -> holder.value += LENGTH;
```

This flies where **offset** didn't, because **holder** is an object reference, which can safely be captured into the lambda context and treated as invariant. What changes is a field of the referenced object, and that won't disappear, even if the calling method terminates and has its local variables cleaned off the stack.

12. Fill in the next three arguments with lambda expressions for previous page, first page, and last page. The final one is just a bit tricky:

```
() -> holder.value -= LENGTH,
() -> holder.value = 1,
() -> { holder.value = 1 + (PAGES - 1) * LENGTH; },
```

Why the curly braces around the last expression? Try it without the braces, and see that the compiler gets confused. This is what they mean by "The syntax has some parsing challenges."

**Using a Generic Data Browser****LAB 13B**

13. For the fifth argument, you need an expression that will show a page of data. We need to print the numbers in the given range – for example from 1 to 10, or from 31 to 40 – in a comma-separated list; and then we need a blank line of text.

The exact implementation tactics are up to you, but note that you will need a block of code for this one, rather than a simple expression.

14. For the final expression, produce a string of the form

Numbers between <offset> and <offset + LENGTH - 1>

15. Now you should be able to run the application. Initial output will be something like this:

```
Page 1 of 5: Numbers between 1 and 10
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Enter "next", "prev", or "quit".

Enter navigation commands and see that the browser and your client code are working together to present the right information:

**next**

```
Page 2 of 5: Numbers between 11 and 20
11, 12, 13, 14, 15, 16, 17, 18, 19, 20
```

Enter "next", "prev", or "quit".

**last**

```
Page 5 of 5: Numbers between 41 and 50
41, 42, 43, 44, 45, 46, 47, 48, 49, 50
```

Enter "next", "prev", or "quit".

**quit**

This is the intermediate answer in **Browser/Step2**.

16. If you don't like the **Holder** approach, there is a simpler strategy. Remove this class from the top of the **main** method, and get rid of the **holder** variable as well.
17. Instead, declare a **private static int** field called **offset**, and initialize that to **1**.

If you like, you can move **LENGTH** and **PAGES** to class scope as well.

18. Change your existing **holder.value** expressions to **offset**.

19. Test your application again, and see consistent results.

This is the final answer in **Browser/Step3**.

## Sorting and Filtering Parts

**LAB 13C**

In this lab you will add to your work from the first lab in the chapter, working with the parts inventory and exploring the use of method references. You'll also see that you can refactor some of that prior lab's code from lambdas to method references.

**Lab project:** Cars/**List/Step7**

**Answer project(s):** Cars/**List/Step8**

**Files:** \* to be created  
**src/cc/cars/Inventory.java**

### Instructions:

1. Open **Inventory.java** and review the code there. This is just about exactly the answer code from the earlier lab, but the two utility methods **extractIf** and **stringify** have been made generic, so you'll be able to use them on, for example, **Part** objects as well as on **Cars**. **stringify** has also been renamed to the more general **transform**.
2. First, take a look over the rest of the code, and see if you can identify lambda expressions that could be simplified to method references. Basically, any lambda that takes a single parameter and just calls a method on it, or passes it to a static method, can be done as a method reference.

There are at least six such opportunities: see if you can find them, and refactor the code. Test to prove that the output is consistent.

3. Let's get a list of the **Part** objects in the dealership. Start by creating a new **Dealership** object, using the no-argument constructor.
4. Call **extractIf**, passing the results of a call to **dealership.getFullInventory** and a predicate that's true only for **Parts**:

```
Collection<Object> extracted = extractIf
 (dealership.getFullInventory(), Part.class::isInstance);
```

Notice that you have to capture this as a collection of objects, because just testing to see that something is a **Part** doesn't automatically cast it to that type.

5. You'll need to do that yourself, by calling **transform: pass your new** extracted collection and **Part.class::cast** as the transformation. This can be captured as a collection of parts.
6. Define a local collection of strings, and initialize it to the results of a call to **transform**, passing your collection of parts and a reference to the **getName** function on **Part**.
7. Call **forEach** on your collection of strings, and pass **System.out::println**.

**Sorting and Filtering Parts****LAB 13C**

8. Test this code, and see that you get a clean list of the names of all parts in the dealership's inventory:

Part names:

```
Brake hose
Rearview mirror
Distributor cap
Headlight
Fuse
Spark plug
```

9. Now let's calculate a sum of all of the "prices paid" for the used cars in the inventory. Start by initializing a **double** to zero.
10. You already have a **usedCars** collection, so you can pass that to **transform**, along with a references to the **getPricePaid** method. And you can put that whole expression on the right side of the colon in a simple **for** loop, and in the loop just add the value to your accumulator.

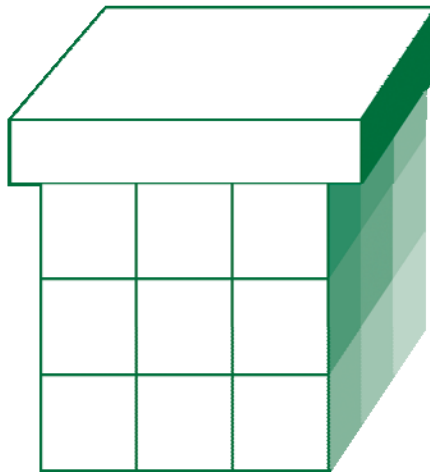
11. Print the results. If you test this you should see something like:

```
Total outlay for used cars is $42,700.00.
```



# CHAPTER 14

## STREAM PROCESSING



## OBJECTIVES

*After completing “Stream Processing,” you will be able to:*

- Explain the Stream API and describe the relationship between Stream and Collections APIs.
- Derive streams from collections.
- Use streams to process information, in a variety of ways:
  - Perform **computations**.
  - **Filter** data and **find** individual elements.
  - **Transform** data values, types, and “shapes.”
  - **Sort** data.
  - Execute **queries** over data sets, and use **aggregate functions**.
  - **Group** data into subsets and derive aggregate values per subset.
- Generate streams algorithmically.
- Direct parallel stream processing, and be aware of the subtle nature of decision-making between sequential and parallel streams.

## The Stream Processing Model

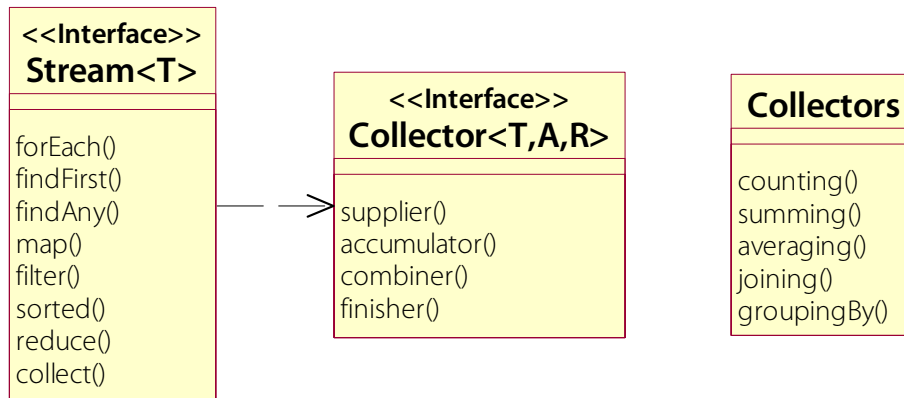
---

- Java 8 introduces a new API for **stream processing**.
- First, we need to separate the two uses of the term “stream” in Java: these are not input/output streams as found in the **java.io** package.
- Rather, we use “**java.util.Stream**” and a few related types to process data – arrays and lists and sets of data, in ways already familiar from work with the Collections API:
  - **Iterating** over data
  - **Finding** things
  - **Filtering** and **transforming** data
  - Deriving **statistics** about the data
  - **Ordering** and even **grouping** the data
- The above actions are also common to work with databases, and for example we can do all of them in **SQL**.
- Stream processing brings a more modern, fluent approach to these traditional tasks, and more of an orientation toward the querying style of SQL even when working with data in memory.
- It also offers benefits in efficiency.
  - There are **fewer deep copies** of large data sets, because most operations are carried out **on-demand**.
  - Streams support **parallel processing**, which can bring major benefits on **multi-core** platforms.

## The Stream API

---

- The interface model for stream processing is remarkably simple, with just about all of the important semantics captured in two key interfaces:



- The **Stream<T>** itself encapsulates a sequence of objects of type **T**, and offers the ability to mutate the stream and to produce some sort of result.
  - Methods such as **map** or **filter** are **intermediate operations**, meaning that each returns a **Stream** – possibly the same one, possibly another, maybe of the same type, maybe of a new type.
  - Then some of its methods are **terminal operations** – returning not streams but **collections**, **maps**, single-value **aggregates**, and other types – or in some cases nothing at all.
- The **Collector<T,A,R>** encapsulates a few related functions that work together to develop a (usually non-**Stream**) result from a stream's members.
- The **Collectors** class utility provides factories for useful **Collector** implementations, including typical SQL aggregate functions and even the ability to group data into a map.

## Relationship to Collections

---

- Streams and collections can both be used to process volumes of information.
- A collection – such as **ArrayList<T>** – both stores information and gives you methods by which to process the information.
- A stream, somewhat like an **Iterator**, provides a processing API but doesn't take responsibility for storing data in memory.
  - You typically **derive a stream** from a collection or an array – or from another stream: there is a method **stream** on **Collection** itself that provides a new stream, ready to use.

```
Stream<String> myStream = myList.stream ();
```

- Streams are **single-use** objects, so we typically create them, use, and release them – whereas we often hold on to collections.
- So more common than the above code would be direct and immediate use of the derived stream:

```
myList.stream().map(...).filter(...).forEach(...);
```

- **Stream<T>** does offer a method **iterator**, so you can for example take a stream from an unknown source and plug it into a traditional **for** loop – though it's more complicated than you might expect to do so:

```
for (String s :
 (Iterable<String>) myStream::iterator) { ... }
```

## Advantages

---

- Anything you can do with a **Stream**, you can do with arrays and/or the Collections API.
- So, why do we have a whole other way of doing the same jobs?
- There are several advantages to the Stream API – and any one or two of these might be enough reason to use a stream instead of a simple for loop or an iterator for a given job.
- Probably the most obvious is the fluent API design, which allows the programmer to chain stream method calls in what is known as a **pipeline**:

```
int count = myList.stream ()
 .map (x -> Math.abs (x))
 .filter (x -> x % 2 == 1)
 .count ();
```

- As you can see, **functional programming** is a big part of the approach, and the Stream API was a big motivator for other Java-8 language features such as lambda expressions and method references.
- Some of this is **style**, and there are times when the gain in concision seems to be outweighed by a loss of clarity.
- But the fluent style is a nice fit to the sorts of jobs we do with streams, because all method return values, except for the final result, are **throwaway objects**.

## Advantages

---

- The API builds in a good bit of functionality commonly implemented by application code in Java 7 and prior. So you don't need to build yet another **for** loop to ...
  - **Search** for an element that meets some criterion
  - **Count** or take **sums** and **averages** of numeric data
  - **Concatenate** or otherwise synthesize a compound result
- The API is also designed to provide SQL-like querying over data as modeled in memory.
  - You can **transform** and **filter** based on mapping and predicate functions.
  - You can **order** results.
  - You can **group**! It's quite a pleasant surprise the first time you come across this capability, and the **groupingBy** collectors are remarkably handy for a range of programming problems.

## Advantages

---

- These are mostly ease-of-use arguments; and ease of use matters.
- But it's important to remember that the Stream API's real value proposition has more to do with efficiency – even where it seems like there's no big win in the way you've coded something.
- And a big piece of this efficiency value proposition is parallelism.
  - **Stream** encapsulates a lot of logic that you would otherwise have to write yourself – and we first see that as ease of use.
  - But it also allows the implementation to organize processing in a way that is open to **parallel processing** – perhaps by **multiple threads**, and perhaps by **multiple processors**.
  - So when you implement processing – even simple algorithms such as a **find-first** search – using streams, you make your code friendly to parallel processing where available.
- Even on a single thread, stream processing can be more efficient:
  - It uses **less memory** thanks to the **on-demand** design.
  - We've not seen these yet but, like the **java.util.function** package, **java.util.stream** defines alternative types dedicated to supporting large sets of **primitive values** – thus saving the considerable costs of object creation for all the **boxing objects** that would be needed otherwise.



## Disadvantages

---

- To be objective, let's also consider some of the down side of using the Stream API.
- Code written in the fluent-API style can be harder to read.
- Between this fluency and the heavy reliance on type inference inherent in Java-8 functional programming, it can be troublesome to write, as well.
  - **Error messages** as specified and as implemented by major vendors might need to mature a bit yet.
  - There are a number of head-scratching cases, including many errors that apparently concern type-matching and ultimately come down to mis-matched parentheses!
- The promised efficiency gains may not be realized for smaller data sets and in single-threaded (a/k/a “sequential”) processing.
  - Streams involve some overhead, and the break-even may be larger than you'd expect.
- Parallel processing can easily backfire on you.
  - For one thing, don't just make a stream parallel and assume that everything will be better and faster. In **single-core** contexts it may well be **slower**.
  - Not all stream logic can support parallel processing, and if you are not careful you can **queer your results** by switching from sequential to parallel – and this is a very bad thing.

## Iterating

---

- Okay, let's get down to cases here: we'll walk through the most common stream operations over the next few pages and see some working examples as we go.
- Among the simplest things you can do is to operate in some way on each element in the stream.

– This would print each element in the stream to the console:

```
myStream.forEach (System.out::println);
```

– This would replace each object's **name** property with an all-lowercase version of its previous **name**:

```
myStream.forEach
 (x -> x.setName (x.getName ().toLowerCase ()));
```

- Since we so often get streams from existing arrays or collections, it's hard just yet to see the value of this approach.

– Why do this ...

```
circles.stream ().forEach
 (circle -> circle.setCircumference
 (Math.PI * 2 * circle.getRadius ());
```

– ... instead of the old, reliable:

```
for (Circle circle : circles)
 circle.setCircumference
 (Math.PI * 2 * circle.getRadius ());
```

- There are the efficiency and parallelism arguments.
- But the above is not a typical scenario; **forEach** is a terminal operation, usually preceded by other, more interesting logic.

## Filtering

---

- You can filter the contents of a stream using the intermediate operation **filter**.
  - The method takes a **Predicate<? super T>**, which can be provided by a lambda expression or method reference (or, as usual, by more traditional methods such as an inner-class instance).

```
myStrings.filter (name -> name.contains ("-"))
myStrings.filter (String::isEmpty)
myNumbers.filter (x -> x > 0)
myDepartments.filter
 (d -> d.getPayroll () < 100000)
```

- The result is a **Stream<T>** – that is, a stream of the same type as the one on which the method was called – that itself will represent only those source elements for which the predicate is **true**.
- It's often a good idea to put your filtering as early in the pipeline as you can, so as to cut down on unnecessary processing of elements that then get filtered out.
- **Stream<T>** also offers the simple method **distinct**, which returns a **Stream<T>** that represents only one of each set of equivalent elements in the source – as evaluated by the **equals** method.

```
myStrings.distinct ()
```

- This method often follows at least one **mapping** function; for example the list of distinct ZIP codes among a list of customers might be captured as

```
customers.map (c -> c.getZIP ()).distinct ()
```

## Filtering and Counting

### EXAMPLE

- In **Streams/Step1** there are a few examples of typical data processing, each problem being worked by traditional means of **for** loops, arrays, and collections; and then again using streams.
- We'll look at one piece of **src/cc/streams/Streams.java** at a time, starting with code that counts the names in a list that include initials followed by periods.

```
List<String> names = new ArrayList<> ();
Collections.addAll (names, "James T. Kirk", ...);
```

```
String regex = ".*[A-Z]\\..*";
```

- With the collections API this would be:

```
int initials = 0;
for (String name : names)
 if (name.matches (regex))
 ++initials;
```

```
System.out.println
 ("Using the Collections API: " + initials);
```

- With streams:

```
System.out.println ("Using the Stream API: " +
 names.stream ()
 .filter (n -> n.matches (regex))
 .count ());
```

- Run the application and see the count done both ways (at the top of the longer output of the program):

```
Names with initials:
Using the Collections API: 3
Using the Stream API: 3
```

## Mapping

---

- You can transform every element in a stream with a call to **map**.
  - This method takes a **Function<? super T, ? extends R>**, which means that it can transform not only the **value** but also the **type** of the element.

```
myNumbers.map (x -> -x)
// e.g. Stream<Integer> to Stream<Integer>
```

```
myStrings.map (s -> s.length ())
// Stream<String> to Stream<Integer>
```

```
myDepartments.map (d -> d.getManager ())
// Stream<Department> to Stream<Employee>
```

- The method returns a **Stream<R>** – so the new stream type may be the same or may be different.
- Type **T** is known as the type argument on the source stream, and type **R** is inferred from the provided **Function** implementation.

## Primitive-Type Streams

---

- As with functional interfaces such as **Supplier** and **Consumer**, **Stream** itself is implemented in several forms.
- **Stream<T>** works for any non-primitive type **T** – and so can work with boxing types **Integer**, **Long**, **Double**, and so on.
- When possible it's usually better to work with one of the primitive-type streams:
  - **IntStream**
  - **LongStream**
  - **DoubleStream**
- These offer the same range of features, but with method signatures dedicated to working with one specific primitive.
  - This avoids a lot of unnecessary boxing and un-boxing costs, in memory and processing time.
  - There are still parameterized methods, for example to allow transformation from a primitive-type stream to an object-type stream.
  - There is also a method **boxed** that returns a **Stream<T>** of the appropriate boxing type.
- **Stream<T>** also has a number of method overloads in order to transform to or otherwise to handle primitive types.

## Mapping and Filtering

**DEMO**

- We're going to start building stream-based alternatives to more processing problems in **Streams.java**.
  - Do your work in **Streams/Step1**.
  - Code showing the completed work for this and a few later segments of the demonstration is in **Streams/Step2**.
- The existing code seeks phone numbers that might be attractive to businesses as being easy to remember.

```
long[] numbers = [1234567890L, ...];
```

```
for (long number : numbers)
{
 String asString = Long.toString (number);
 Set<Character> distinct = new TreeSet<> ();
 for (int i = 0; i < asString.length (); ++i)
 distinct.add (asString.charAt (i));

 if (distinct.size () <= 4)
 System.out.println
 (" " + padNumber (asString));
}
```

- The logic to format the number in a user-familiar way is just busy enough that we place it in a helper method:

```
public static String padNumber (String number)
{
 return number.substring (0, 3) + "-" +
 number.substring (3, 6) + "-" +
 number.substring (6);
}
```

## Mapping and Filtering

**DEMO**

- So, with streams, we need to ...
  - Develop a **LongStream** from the provided numbers
  - **Map** them to string representations
  - **Filter** them by the occurrences of distinct digits
  - **Map** those that pass the filter to the user-friendly form
  - **For each**, print to the console
- 1. Run the application to see results as produced by the existing code:

```
Attractive phone numbers:
 Using array processing:
 800-818-1800
 858-558-8555
 617-616-1000
 212-222-3300
 877-227-2477
 Using the Stream API:
```

- 2. After the existing code, get a **LongStream** of the numbers:

```
System.out.println (" Using the Stream API:");
LongStream.of (numbers)
```

- 3. For starters, just jump to the end of the process and print them:

```
.forEach (n -> System.out.println (" " + n));
```



## Mapping and Filtering

**DEMO**

4. Try running this, and of course you'll see incorrect results – all the numbers print, and without any formatting:

Attractive phone numbers:

Using array processing:

```
800-818-1800
858-558-8555
617-616-1000
212-222-3300
877-227-2477
```

Using the Stream API:

```
1234567890
8008181800
5123437900
8585588555
6176161000
2122223300
1234512345
8772272477
4133145000
```

5. Map the **long** values to **Strings**:

```
LongStream.of (numbers)
```

```
 .mapToObj (String::valueOf)
```

```
 .forEach (n -> System.out.println (" " + n));
```

- If you run again at this point, the results will be the same.

## Mapping and Filtering

**DEMO**

- Now we need to apply a filter.
- We could capture the filter logic from the traditional implementation in a lambda expression, or break it into another helper method and refer to that.
- But, getting a count of distinct results ... streams are good at that! Could we get a stream of the characters of each candidate string?
- It turns out we can – **String** has a new method **chars** that does just what we want.

6. Apply a filter as follows:

```
LongStream.of (numbers)
 .mapToObj (String::valueOf)
 .filter (n -> n.chars ().distinct().count() <= 4)
 .forEach (n -> System.out.println (" " + n));
```

7. Test now ...

Attractive phone numbers:

Using array processing:

```
800-818-1800
858-558-8555
617-616-1000
212-222-3300
877-227-2477
```

Using the Stream API:

```
8008181800
8585588555
6176161000
2122223300
8772272477
```

- So, the filtering works, but we still need to format the numbers.

## Mapping and Filtering

**DEMO**

8. We can easily tap into the **padNumber** function, by method reference:

```
LongStream.of (numbers)
 .mapToObj (String::valueOf)
 .filter (n -> n.chars ().distinct ().count () <= 4)
 .map (Streams::padNumber)
 .forEach (n -> System.out.println (" " + n));
```

9. Run the finished implementation and see identical results for both approaches:

Attractive phone numbers:

Using array processing:

```
800-818-1800
858-558-8555
617-616-1000
212-222-3300
877-227-2477
```

Using the Stream API:

```
800-818-1800
858-558-8555
617-616-1000
212-222-3300
877-227-2477
```

## Aggregate Functions and Statistics

---

- The Stream API makes available the set of aggregate functions familiar to SQL programmers, as terminating methods on the stream types:
  - `count`
  - `sum`
  - `average`
  - `min`
  - `max`
- All the primitive streams implement all of these.
- **Stream<T>** doesn't offer **sum** or **average**; you would **mapToInt/Long/Double** somehow before calculating these.
- It does provide **min** and **max**, taking a **Comparator** to determine what is the "greater" element and what is the "lesser."
- Beware that the **average**, **min**, and **max** methods return a wrapping type known as an **optional** – to account for the contingency that there is no value.
  - You can count **zero elements**, but you can't get an average or say what was the highest or lowest value.
  - The return type will be **OptionalInt/Long/Double**, or **Optional<T>** for **Stream<T>**.
  - Call **getAsInt/Long/Double** if you're certain that there is a value – or just **get** on **Optional<T>**.
  - Call **isPresent** first, if not sure.

## Aggregate Functions and Statistics

---

- You can derive the same aggregates using collectors provided by the **Collectors** utility:

```
counting
summingInt/Long/Double
averagingInt/Long/Double
minBy
maxBy
```

- The **sum** and **average** functions take **mappers**, and so combine the mapping and aggregate calculation in one shot.
- There are also methods to derive sets of statistics:

- On the primitive streams these are:

```
summaryStatistics
```

- On the **Collectors** utility, these methods return collectors:

```
summarizingInt/Long/Double
```

- First, beware of the similarity to **summingInt/Long/Double**.
- These collectors return **Int/Long/DoubleSummaryStatistics**, which in turn have read-only properties **count**, **sum**, **average**, **min**, and **max**.
- With this approach you can get all statistics in a **single pass** through the stream, rather than having to create and use another stream for each value.

## Averaging

**DEMO**

- Continuing work in **Streams/Step1** ... the third segment of **Streams.java** shows calculation of average response time – let's say to a broadcast email that requests that each member of a group fill out a form with information.

```
final SimpleDateFormat formatter =
 new SimpleDateFormat ("h:m");
Date request = formatter.parse ("9:00");
List<Date> responses = new ArrayList<> ();
Collections.addAll (responses,
 formatter.parse ("10:00"), ...);
```

- Typical calculation in Java 7 might run like this:

```
long requestInstant = request.getTime ();
long total = 0L;
for (Date response : responses)
 total += response.getTime () - requestInstant;

System.out.println ("Using the Collections API: " +
 ((int) total / responses.size () / 1000 / 60) +
 " minutes.");
```

10.Run to see the baseline calculation, and a placeholder for our result:

```
Average response time:
Using the Collections API: 107 minutes.
Using the Stream API: minutes.
```

## Averaging

**DEMO**

11. Now let's build a stream-based solution. Start by getting a stream from the collection, mapping each date to a long which is the response time in milliseconds, and taking the average of that:

```
System.out.format("... Stream API: %d minutes.%n",
 responses.stream ()
 .mapToLong (r -> r.getTime () - requestInstant)
 .average ().getAsDouble() / 1000 / 60);
```

12. Test that much – hmm, a couple of issues:

Average response time:  
Using the Collections API: 107 minutes.  
Using the Stream API: 107 minutes.

- Remember that **average** returns an **optional**.
- And, we need to get from milliseconds to minutes.

13. Call **getAsDouble** to shed the **OptionalDouble**, and then divide to get seconds and then minutes, converting the results to an **int**:

```
System.out.println ("Using the Stream API: " +
 ((int) responses.stream ()
 .mapToLong (r -> r.getTime () - requestInstant)
 .average ()
 .getAsDouble () / 1000 / 60) +
 " minutes.");
```

14. Run this final implementation to see the answer we want:

Average response time:  
Using the Collections API: 107 minutes.  
Using the Stream API: 107 minutes.

## Sorting

---

- Call **sorted** on a stream to derive a new stream with elements in a given order.
- All stream types offer this method.
  - On the primitive streams the order will be **ascending numerical**.
  - On **Stream<T>**, it will be the “natural order” for type **T**, as determined by that type’s implementation of **Comparable<T>**.
- **Stream<T>** also offers an overload that accepts a **Comparator<T>** as an argument.
  - This makes it easy to sort objects based on one of their common properties.
  - There is no such flexibility on the primitive stream types.



## Test Scores

**LAB 14A**

### **Suggested time: 30 minutes**

In this lab you will re-implement most of the processing of test scores in the Scores application, using streams. The one piece we'll leave for later in the chapter is analysis of grade distribution – which will turn out to be a nice application of grouping. For now you'll do the grade assignments, sort the results a few ways, and find the mean score.

Detailed instructions are found at the end of the chapter.

## Generating

---

- You most often derive streams from arrays and collections.
- But you can synthesize a stream in other ways.
  - Call the static method **generate** to create a stream based on a **Supplier** of the appropriate type:

```
LongStream.generate (mySequence::next)
IntStream.generate (SecureRandom::nextInt)
```

- Call the static method **iterate** to create a stream based on initial state and a mapping function:

```
IntStream.iterate (0, x -> x + 1)
Stream<boolean[][]>.iterate (square, Life::algo)
```

- **IntStream** and **LongStream** offer the **range** and **rangeClosed** methods to provide sequences:

```
IntStream.rangeClosed (1, 100);
```

- **range** sets up a sequence that includes the start number but **excludes the end** number.
  - **rangeClosed** includes both.
- You can define a stream **of** any number of predefined values:

```
IntStream.of (1, 1, 2, 3, 5, 8, 13);
Stream<Car>.of (theElCamino);
```

- Finally, **empty** will produce an empty stream.

```
Stream<String>.empty ();
```

## Limiting

---

- This brings us for the first time to the possibility of **infinite streams**.
  - There are use cases for infinite streams of data – though usually we’re really saying that something is finite, but we just don’t know what the boundaries are.
  - In other words, programs that run until shut down may be written as if they will run forever.
- So an infinite stream may indeed be what you want.
- If not, consider the **limit** function, which fixes the maximum number of elements that can be pulled from the stream.

```
IntStream.generate (SecureRandom::nextInt)
 .limit (100)
```

## Reducing

---

- A **reducing operation** is one that transforms a stream of data into a single result.
  - The result type might be something **multi-valued**, such as a collection that we want to keep, return to the caller, etc.
  - Or it might be a **single value** – and all the aggregate functions are just specialized reducers.
- If the built-in aggregates and collectors don't do what you need to do, call **reduce** and provide your **reducing operator**.
  - This will be a **BinaryOperator<T>** for **Stream<T>**, or an **Int/Long/DoubleBinaryOperator** for the primitive streams.

```
myIntegers.reduce ((x,y) -> x ^ y)
```

- Your operation doesn't need to be **commutative**, but be aware that the first parameter will be the result of the prior operation.
- There is an overload that takes an **initial value** (called the “identity value”) as the first parameter, and the operation as the second.

```
myIntegers.reduce (-1, (x,y) -> x ^ y)
myObjects.reduce
 (new ArrayList<MyClass>, List::add)
```

## Joining

---

- For string concatenation, also consider **Collectors.joining**, each of the overloads of which return a **Collector** that concatenates string representations of the source stream's elements.

```
String codonString =
 codons.stream ().collect (Collectors.joining ());
```

- You can provide a **delimiter**, and/or **prefix** and **suffix**.

```
System.out.println (names.collect
 (Collectors.joining (" ")));
```

```
System.out.println (comments.collect
 (Collectors.joining (" ", "/* ", " */")));
```

## Generating, Limiting, and Reducing

**EXAMPLE**

- The final passage in **Streams.java** computes a checksum over a sequence of integers.

– See this in **Streams/Step2**.

```
final int LIMIT = 60606;
```

– Traditional looping code for this is:

```
byte checksum = 0;
for (int i = 0; i < LIMIT; ++i)
 checksum += i;
System.out.println
 ("Using a for loop: " + checksum);
```

– With streams it is:

```
System.out.println ("Using the Streams API: " +
 IntStream.iterate (0, x -> x + 1).limit (LIMIT)
 .reduce ((x,y) -> (byte) (x + y)).getAsInt ());
```

```
System.out.println (" ... or: " +
 IntStream.range (0, LIMIT)
 .reduce ((x,y) -> (byte) (x + y)).getAsInt ());
```

- Output from this final passage is:

Checksum:

Using a for loop: 35

Using the Streams API: 35

... or: 35

## Finding and Matching

---

- You can derive a reference to one element from a stream by calling one of the finding methods.
  - Both return an **Optional<T>**.
  - They don't take a **Predicate**, as you might have guessed; we already have **filters** that do that, so we just pick from what's left:

```
Optional<Record> record = scores.stream ()
 .filter (r -> r.score > 89).findFirst ();
```

```
Optional<String> longEnough = pwds.stream ()
 .filter (p -> p.length () > 8).findAny ();
Optional<String> found = tokens.stream ()
 .filter (String::empty).findAny ();
```

- As the names suggest, **findFirst** guarantees to give you the first element in the stream to meet your criteria.
  - **findAny** will get you a match, but it might not be the first. This can perform better on a parallel stream, but it doesn't guarantee even that it will find the **same result** every time.
- You can find out if a certain predicate would be matched, without getting the matching element(s), with two more methods.
  - These do take predicates, directly:

```
boolean safe =
 pwds.stream ().allMatch (p -> p.length () > 8);

boolean valid =
 cluster.stream ().anyMatch (s -> s.isMaster ());
```

## Grouping

---

- An especially neat utility available from **Collectors** is a set of collectors that will perform grouping.
- There are three overloads of **groupingBy**, and all will turn a **Stream** into a **Map** where keys are determined by applying a **classifier** function.

- One gives you the full group membership for each key value, as **Map<K, List<T>>**. Here are employees partitioned by department:

```
employees.stream ().collect (Collectors.groupingBy
 (e -> e.getDepartment ().getID ()))
```

- Another gives you aggregate values, in a **Map<K,D>**, based on a second **Collector** that computes the aggregates. Here is each department's payroll:

```
employees.stream ().collect (Collectors.groupingBy
 (e -> e.getDepartment ().getID (),
 Collectors.summingInt (e -> e.getSalary ())))
```

- The third also performs aggregation, and gives you the opportunity to **supply** the map. Here's the payroll map again, but this time sorted by department name:

```
employees.stream ().collect (Collectors.groupingBy
 (e -> e.getDepartment ().getID (), TreeSet::new,
 Collectors.summingInt (e -> e.getSalary ())))
```

- The only downside to the grouping collectors is that they can be a real handful – especially with all the type arguments to one or two functions and a secondary collector.



## Grade Distribution

**DEMO**

- Let's finish the job we started in the previous lab, and restore the logic for analyzing distribution of letter grades in the Scores application.
  - Do your work in **Scores/Step11**.
  - The completed demo is found in **Scores/Step12**.

1. Open **src/Scores.java** and add console output for a section with the grade analysis:

```
System.out.println ();
System.out.println ("Statistics:");

System.out.println ("The mean score was " + ...);
```

2. Develop a grouping operation over the records, as follows:

```
scores.stream().collect(Collectors.groupingBy
 (r -> r.grade, Collectors.counting()))
```

- So we're classifying records by **grade** – that will be the key in the resulting map – and for a value we're just getting a **count** of students with a given grade.

3. Now, what to do with this result? It's going to be a **Map<String,Long>** ... so we can iterate over the results, and here we turn to a good, old-fashioned loop:

```
for (Map.Entry<String,Long> entry :
 scores.stream().collect(Collectors.groupingBy
 (r -> r.grade, Collectors.counting()))
 .entrySet ())
 System.out.format
 (" There were %1d %1ss given.%n",
 entry.getValue (), entry.getKey ());
```

## Grade Distribution

**DEMO**

4. Run the application, and see that you've definitely got the grouping and counting operations working:

Statistics:

```
There were 2 As given.
There were 2 Bs given.
There were 3 Cs given.
There were 2 Ds given.
There were 1 Fs given.
```

5. Thanks to the simplicity of the strings we're using as keys, the results show up in alphabetical order. But the map we're given, by default, is a **HashMap**. To assure sorted results, you can use the third overload of **groupingBy**:

```
for (Map.Entry<String,Long> entry :
 scores.stream().collect(Collectors.groupingBy
 (r -> r.grade, TreeMap<String,Long>::new,
 Collectors.counting()))
 .entrySet ())
 System.out.format
 (" There were %1d %1ss given.%n",
 entry.getValue (), entry.getKey ());
```

## Flattening and Traversing

---

- You can also simply flatten a multi-level data structure, using **flatMap** and variants thereof.
  - This method takes a **Function** that in turn accepts an object of type **T** and produces a **Stream<T>**.

```
long employeeCount =
 departments.stream ()
 .flatMap (d -> d.getEmployees ().stream ())
 .count ();
```

- It derives the “**sub-stream**” for each element in the source stream.
  - But rather than putting each sub-stream in a map, it **flattens** the results into a single stream.
- You can also sew together multiple streams of like type into a single stream, using **Stream.concat**:

```
Stream.concat
 (hotDrinks.stream (), coldDrinks.stream ())
 .forEach (Drink::serve);
```

- This is an elegant answer to the problem of **traversing** multiple collections, without needing to create an intermediate collection in memory just to represent the concatenation.

## Calling All Cars

**EXAMPLE**

- In **Cars/List/Step9**, the **Dealership** class has been reworked to take advantage of streams – almost entirely for the benefit of concatenating its three lists of **cars**, **usedCars**, and **parts**.
- See **src/cc/cars/Dealership.java**:

- Near the bottom of the file, see that **getAllCars** now returns a **Stream<Car>**, and uses **concat** to produce a result:

```
protected Stream<Car> getAllCars ()
{
 return Stream.concat
 (cars.stream (), usedCars.stream ());
}
```

- **getInventory** goes one better, merging that stream with parts:

```
protected Stream<InventoryItem>
 getFullInventory ()
{
 return Stream.concat
 (getAllCars(), parts.stream ());
}
```

## Calling All Cars

**EXAMPLE**

- Various other methods on the class are revised to work with streams instead of collections:

- **findCar** methods now use stream filtration – for example:

```
public Car findCar (String VIN)
{
 Optional<? extends Car> found = getAllCars ()
 .filter (car -> car.getVIN().equals(VIN))
 .findFirst ();
 return found.isPresent () ? found.get() : null;
}
```

- Listing methods could use **forEach** and then call printing methods, but they go a different route by using the **joining** collector; for example here's **listInventory**:

```
public String listInventory ()
{
 return InventoryItem.getHeader () +
 getFullInventory().collect
 (Collectors.mapping (item ->
 item.report (), Collectors.joining ()));
}
```

## Car Queries

**LAB 14B**

**Suggested time: 30-45 minutes**

In this lab you will implement a set of queries over the set of cars available in the car-dealership application, using the Stream API.

This is a challenge lab, with relatively high-level instructions and the expectation that you will invent your own solutions to the problems it poses – and indeed there are multiple ways to solve most of these.

Detailed instructions are found at the end of the chapter.

## Parallel Processing

---

- Since the Stream API is so motivated by the promise of parallel processing, we'd be remiss not at least to consider it in this chapter.
- But, ultimately, parallelism is a challenging area.
  - It is not “settled science” that the automatic (auto-magic?) parallelism offered by the Java 8 runtime is the best general-purpose strategy – and certainly not obvious that it's the best strategy for your specific application and processing challenges.
  - Parallel processing can go wrong in many ways, and it should be considered an advanced technique.
  - It is nowhere near as simple as ... well, as what we're about to do, which is to add the method **parallel** to a stream setup before carrying out a process over it.

## Sequential vs. Parallel Processing

### EXAMPLE

- In **Parallel** there is an application that runs a fairly simple, high-volume computation: it takes a running average of integers running from 0 to 999.
- It does this five times, five different ways, in order (let's say semi-scientifically) to explore the effects of parallel vs. sequential processing.

- Here is the core method, **chew**, which offers three switches:

```
public static void chew (boolean countThreads,
 boolean slowDown, boolean parallel)
{
 System.out.print
 ("Processing " + LIMIT + " numbers");
```

- It sets up a stream of numbers from 0 to **LIMIT** - 1:

```
 IntStream source = IntStream
 .iterate (0, x -> x + 1).limit (LIMIT);
```

- If told to process in parallel, it calls **parallel** on the stream, re-assigning the variable **source** to represent the parallel stream:

```
 if (parallel)
 {
 source = source.parallel ();
 System.out.print (", in parallel");
 }
```



## Sequential vs. Parallel Processing

**EXAMPLE**

- If told to count threads, it inserts into the processing pipeline a reference to a helper method that adds the calling thread to a **Set<Thread>**, to be queried later for its size.

```
if (countThreads)
{
 source = source.map
 (ChewThroughNumbers::countThreads);
 threads.clear ();
 System.out.print (" , counting threads");
}
```

- If told to slow the process down, it inserts a helper method that sleeps the calling thread for 10 milliseconds:

```
if (slowDown)
{
 source = source.map
 (ChewThroughNumbers::slowDown);
 System.out.print (" , slowing down");
}
```

```
System.out.println (" ...");
System.out.println ();
```

## Sequential vs. Parallel Processing

### EXAMPLE

- Then it computes the average by completing the pipeline with the terminal method **average**. It produces the result and a calculation of time consumed in processing, and if asked to count threads it shows the count:

```
long start = System.currentTimeMillis ();
double mean = source.average ().getAsDouble ();
long time =
 System.currentTimeMillis () - start;

System.out.println ("Result: " + mean);
System.out.println
 ("Done in " + time + " msec.");
if (countThreads)
 System.out.println
 ("Used " + threads.size() + " threads.");
System.out.println();
}
```

## Sequential vs. Parallel Processing

**EXAMPLE**

- The **main** method calls **chew** with five combinations of switches:

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

- Sequentially, with no modifications – the way we’ve done all of our processing in this chapter:

```
 chew (false, false, false);
```

- Counting threads used in sequential processing:

```
 chew (true, false, false);
```

- Counting threads but switching to parallel processing:

```
 chew (true, false, true);
```

- Counting threads, sequential processing again, and now artificially slowing down the process:

```
 chew (true, true, false);
```

- Counting threads, slowing down, in parallel:

```
 chew (true, true, true);
```

```
}
```

## Sequential vs. Parallel Processing

**EXAMPLE**

- Drum roll, please ... run the application!
- Let's consider the results. First, sequential processing takes no measurable time – after all, it's ultimately just 1000 very simple computations:

```
Processing 1000 numbers ...
```

```
Result: 499.5
```

```
Done in 0 msec.
```

```
Processing 1000 numbers, counting threads ...
```

```
Result: 499.5
```

```
Done in 0 msec.
```

```
Used 1 threads.
```

## Sequential vs. Parallel Processing

### EXAMPLE

- Running the same process in parallel ... takes longer:

```
Processing 1000 numbers,
 counting threads, in parallel ...
```

```
Result: 499.5
Done in 953 msec.
Used 4 threads.
```

- Keep in mind that throwing multiple threads at a computation does not guarantee faster execution.
- There are **overheads** to multi-threading, including context-switching between running threads, synchronization, and combination of results.
- On a single-processor system, the best motivations for multi-threading are **slow resources** (file system, network, peripherals) and **responsiveness** (to users and perhaps incoming remote requests).
- Multiple threads all trying to use the processor for computation will just wind up in line behind one another, and you pay a high price to manage the threads while getting no benefit.
- Now, if we had a multi-processor or multi-core system and were to administer the JRE to take advantage of that, then everything would change, and it could easily be a win (although with 1000 numbers to crunch, a hard win to measure!) to use parallel processing.

## Sequential vs. Parallel Processing

### EXAMPLE

- This is why we've included the "slow down" switch.
- Let's look at a comparison of sequential vs. parallel where the action taken on every element involves 10 msec of down time:

```
Processing 1000 numbers,
 counting threads, slowing down ...
```

```
Result: 499.5
Done in 15626 msec.
Used 1 threads.
```

```
Processing 1000 numbers,
 counting threads, slowing down, in parallel ...
```

```
Result: 499.5
Done in 3907 msec.
Used 4 threads.
```

- Now that's more like what we expected to see.
- The four threads allocated by the JRE still queue up behind each other – every 10 milliseconds – but then **all four get to run** before waiting again, so four times as much processing occurs over time.
- Time spent is still more than 25% of the single-threaded time, reflecting the **overheads** discussed on the previous page.
- But now it's a **net win**, where for raw computation it's a net loss.
- The takeaway from all this should be:
  - Understand that parallel processing is an advanced science.
  - Don't just call **parallel** on all your streams and assume that this will be beneficial.

## SUMMARY

- At first blush, a lot of Stream API examples will look like just a different syntax to solve a problem we can already solve using **for** loops, conditionals, arrays, and the Collections API.
- But streams bring a number of advantages:
  - A **fluent API**, a better fit with **functional programming** as enabled in Java 8, and generally easier use
  - A larger body of **built-in logic**, especially for querying
  - Support for parallel processing and an API that makes it easy to switch this feature on and off
- The Stream API – in concert with the functional API – is meticulously designed and implemented to be efficient and type-safe when working with common primitive types.
- Parallel processing is a big motivator for the Stream API, but that doesn't mean it should be used liberally or without deep analysis of costs and benefits for a particular system.

# Test Scores

## LAB 14A

In this lab you will re-implement most of the processing of test scores in the Scores application, using streams. The one piece we'll leave for later in the chapter is analysis of grade distribution – which will turn out to be a nice application of grouping. For now you'll do the grade assignments, sort the results a few ways, and find the mean score.

**Lab project:** Scores/Step10

**Answer project(s):** Scores/Step11

**Files:** \* to be created  
src/Scores.java

### Instructions:

1. Open **Scores.java** and see that – eek! – all the logic has been ripped out. Now there are just a few pieces, similar to the starter sets for earlier labs: the data, as a **List** of **Record** objects; and a couple of helper functions **header** and **report** to simplify console output.

The **main** method already produces a series of headers, but there's no data in any of these sections. So if you run the application you'll see:

All grades:

Student	Score	Grade
-----	-----	-----

Sorted highest-to-lowest:

Student	Score	Grade
-----	-----	-----

Sorted alphabetically by name:

Student	Score	Grade
-----	-----	-----

The mean score was

2. Right after the population of the **scores** list, derive a stream from the list, and call **forEach** on it.



**Test Scores****LAB 14A**

3. As your operation in **forEach**, place the code (and you may want to refer to **Scores/Step9** for this, and perhaps copy some fragments) that calculates the grade based on the score – but now as a lambda expression. You’re implementing a **Consumer<Record>** with this expression here, and you can get the **score** field from the given **Record** and make your calculations. Then store this in the record’s **grade** field.
4. Now, after the first header (“All grades:”), you need to show the data. This one’s pretty easy: develop another **forEach** and just provide the **report** method, by reference, as your operation.
5. If you run now you should see the grades assigned correctly:

All grades:

Student	Score	Grade
-----	-----	-----
Suzie Q	76	C
Peggy Fosnacht	91	A
Boy George	80	B
Flea	55	F
Captain Hook	71	C
Nelson Mandela	98	A
The Mighty Thor	70	C
Oedipa Maas	88	B
Uncle Sam	69	D
The Tick	60	D

6. Now you need to show the records, sorted from high score to low score. Remember that **Record** already implements **Comparable<Record>**, and so has a natural order based on the test score.

This task doesn’t require that you make any changes to the underlying collection, though in earlier versions of the application we stored these results in a second collection. Since all we’re going to do is to produce these to the console, and then forget them, you can just sort another scores stream – use **sorted** and pass it the results of calling **Collections.reverseOrder** – and then chain your **forEach** to call **report** again.

**Test Scores****LAB 14A**

7. Test and see that this does what we want it to do:

Sorted highest-to-lowest:

Student	Score	Grade
-----	-----	-----
Nelson Mandela	98	A
Peggy Fosnacht	91	A
Oedipa Maas	88	B
Boy George	80	B
Suzie Q	76	C
Captain Hook	71	C
The Mighty Thor	70	C
Uncle Sam	69	D
The Tick	60	D
Flea	55	F

8. Now sort and report again, but this time sorting names alphabetically. Now you will need to write your own **Comparator<Record>** implementation – which you can do as a lambda expression, taking two parameters and comparing their **name** fields.

9. Test and see output like this:

Sorted alphabetically by name:

Student	Score	Grade
-----	-----	-----
Boy George	80	B
Captain Hook	71	C
Flea	55	F
Nelson Mandela	98	A
Oedipa Maas	88	B
Peggy Fosnacht	91	A
Suzie Q	76	C
The Mighty Thor	70	C
The Tick	60	D
Uncle Sam	69	D

10. Finally, calculate the mean test score. You can do this in at least two ways. First, try mapping each **Record** to an **int** which is that record's score value – you will need to call **mapToInt** to get an **IntStream** as a result – and then calling **average** on that stream. Remember to **getAsDouble** at the end, or you'll see the string representation of an **OptionalDouble** instead of just the number.

11. Test and see that you get ...

The mean score was 75.8

We'll look at a second approach to averaging that uses a prepared **Collector**, later in the chapter.

## Car Queries

**LAB 14B**

In this lab you will implement a set of queries over the set of cars available in the car-dealership application, using the Stream API.

This is a challenge lab, with relatively high-level instructions and the expectation that you will invent your own solutions to the problems it poses – and indeed there are multiple ways to solve most of these.

**Lab project:** Cars/**List/Step9**

**Answer project(s):** Cars/**List/Step10**

**Files:** \* to be created  
**src/cc/cars/Queries.java**

### Instructions:

1. Open **Queries.java** and see two helper functions **show** – one taking a **Car** and one taking an **Optional<Car>**, and both designed to help in producing query output. Both are **static** and so might be called explicitly or passed to stream methods via method reference.

See also that the **main** method carries out a few of the early steps of the **Application** class that you’ve been running throughout most of this course: it creates a **Persistence** helper and uses it to load lists of **cars** and **usedCars**. These two lists will be the source data for all your queries.

2. At the bottom of the **main** method, use the Stream API to find a car by VIN. As with all of these exercises, you can provide query parameters as hard-coded values – in this case one appropriate value would be “ME3278”. Note that you should be able to find the car whether it is new or used.

Run your class as a Java application, or use the prepared script **runQueries**. Expected output is:

```
ME3278: 2014 Honda Accord (Red) $29,999.99
```

3. Now find all new cars of a specific model year, 2015, and print each to the console. Expected output:

```
ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
HJ5599: 2015 Saab 9000 (Pearl) $34,498.99
WE9394: 2015 Kia Sonata (White) $21,999.99
XY1234: 2015 Ford F-150 (Black) $28,999.99
PU4128: 2015 Audi A3 (Blue) $39,999.99
```

**Car Queries****LAB 14B**

4. Find new cars with a price less than \$30,000. Expected output:

```
ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
ME3278: 2014 Honda Accord (Red) $29,999.99
UI4456: 2014 Volkswagen Jetta TDI (Green) $23,899.99
WE9394: 2015 Kia Sonata (White) $21,999.99
XY1234: 2015 Ford F-150 (Black) $28,999.99
IM3110: 2014 Ford Escape Hybrid (Silver) $23,999.99
```

5. Find all cars whose **make** is “Ford”, and sort the results from lowest to highest **price**.  
Expected output:

```
AR7993: 1974 Ford Pinto (Dust) $0.99 -- USED
WQ0227: 1972 Ford El Camino (Blue and tan) $2,098.99 -- USED
QL3314: 2003 Ford Taurus (Gold) $9,499.99 -- USED
IM3110: 2014 Ford Escape Hybrid (Silver) $23,999.99
XY1234: 2015 Ford F-150 (Black) $28,999.99
BA0091: 2014 Ford Taurus (Black) $32,499.99
```

6. Show all new cars, grouped by model year. Expected output:

Model year 2014:

```
BA0091: 2014 Ford Taurus (Black) $32,499.99
ME3278: 2014 Honda Accord (Red) $29,999.99
UI4456: 2014 Volkswagen Jetta TDI (Green) $23,899.99
IM3110: 2014 Ford Escape Hybrid (Silver) $23,999.99
```

Model year 2015:

```
ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
HJ5599: 2015 Saab 9000 (Pearl) $34,498.99
WE9394: 2015 Kia Sonata (White) $21,999.99
XY1234: 2015 Ford F-150 (Black) $28,999.99
PU4128: 2015 Audi A3 (Blue) $39,999.99
```

7. Show average sticker price for new cars by model year. Expected output:

```
Model year 2014: $ 27,599.99
Model year 2015: $ 30,082.82
```