

# Core Java 8 Programming

TT2105: Student Guide



## Trivera Technologies LLC

Collaborative IT Training, Mentoring & Courseware Solutions

Trivera Technologies LLC is a premier provider of **IT education and training, structured mentoring programs, custom courseware design & licensing and specialized education** services and solutions. Our extensive team of subject matter experts, engaging instructors and courseware designers brings years of current, practical, real-world experience into every classroom.

**Our goal is to make you a success.** Whether you're a manager arranging training for your company, a trainer or training firm using our courseware materials at your valued client site, or a student attending a course presented by one of our experts, **our core mission is to develop and deliver exactly the program you need to achieve your goals**, held to the highest industry quality standards, with complete support and assistance before, during and after your event.

We are a skills-oriented services firm that offers introductory through advanced level training in hundreds of IT topics. Our courses are fully customizable to suit your unique requirements, and can be presented live in your classroom, virtually for private delivery for your team, or in public training courses available around the globe. We **wholly-own** the majority of the courses we teach, providing us with the ultimate flexibility to create the most effective, targeted programs. Our expert delivery and management team services your event from end to end, producing and delivering targeted events with limited burden on your own firm, providing you with the most value for your training dollar.

### Trivera Technologies Services Include

- **World Wide Instructor Led Training offered Onsite, Online, Self-Paced & Blended Solutions**
- **Extensive Open Enrollment Public Schedule**
- **Collaborative Mentoring Programs, Skills Immersion Camps & Code Academy Programs**
- **Skills Assessment Services; Gap Training Solutions and Corporate Training Plans**
- **Corporate Training & Vendor Management Solutions**
- **Custom Course Development Services**
- **Courseware Licensing – Corporate or OnDemand**
- **Flexible Pricing Programs and Options**
- **Satisfaction Guaranteed Services**

### Trivera Technologies

#### Educate. Collaborate. Accelerate!

Training • Mentoring • Courseware • Programs  
In-Person • Online Live • Public Schedule • Custom

Training@triveratech.com | +609.647.7572 phone  
[www.triveratech.com](http://www.triveratech.com)



100% Woman-Owned Small Business

### A Sample of our Course Offerings includes....

#### > Application Development, Programming & Coding:

Intro to Programming • Java / JEE • C# / .Net • ASP.Net • VB.Net • C++ • C • COBOL • R • Scala • SQL • Cloud • Spring • Spring Batch / Boot / Security / Cloud • Hibernate • JSF • Struts • Secure Coding • Microservices • Web Services • MVC • RESTful Services • Design Patterns • Cloud

#### > Agile & TDD: Agile Development • TDD / BDD • Unit Testing • Frameworks • Scrum • SAFE • Kanban

#### > Web Development & Design: HTML5 / CSS3 • JavaScript • JQuery • Angular • React / Redux • MEANStack / MERNStack • Node.js • UX / UI • BootStrap • XML / XSLT • Responsive Design • Python • Perl • PHP • XML

#### > Python: Python for Web • Python Networking / Sys Admin • Python Data Science • Python for Security Pros

#### > Security: CyberSecurity • Secure Software Design • Secure Coding • Secure Web Development • Database Security • OWASP • STIG • CCSK

#### > Mobile: Android • IOS • PhoneGap • Kotlin • Swift • Mobile Application Testing • Secure Mobile Development

#### > Big Data / Data Science, AI , Machine Learning, & Deep Learning: Machine Learning • Hadoop Admin • Hadoop Development • Pig / Hive / MapReduce • Scala • Spark • R for Data Science • Data Science • Python for Data Science • AI – Artificial Intelligence

#### > SOA: SOA • Architecture • Analysis • Design • Governance • Implementation • SOA Security

#### > Databases: DB Design • DB Security • DB2 • SQL • PLSQL • Oracle • MySQL • SQLServer • MongoDB • NoSQL • MariaDB • Cassandra • Oracle • JDBC

#### > MainFrame: C • COBOL • DB2 • Assembler • Transition to Web • Java for COBOL

#### > O/S & SysAdmin: Windows • Linux • UNIX • Mainframe • Z/OS • Administration • IOS

#### > Oracle: DBA • SQL • PL/SQL • New Features • OBIEE

#### > Tools • Eclipse • Ant / Maven • Oracle • Oracle BI • IntelliJ • NetBeans • Tableau • Selenium • Cucumber

#### > DevOps: DevOps • GIT • GITHUB • Jenkins • JIRA • Docker

#### > IT Skills: Agile • Scrum • ITIL • Project Management • Leadership • Soft Skills • End User

#### > Software Architecture, Design & Engineering: Architecture • Analysis • Requirements • Estimation • Use Cases • UML • OO • BDD • Data Modeling & Design • Software Design

#### > Software Testing: QA • Test Automation • Unit Testing • TDD • Selenium • Cucumber • Gherkin

Please visit [www.triveratech.com](http://www.triveratech.com) for the complete course catalog.

All written content, source code, and formatting are the property of Trivera Technologies LLC. No portion of this material may be duplicated or reused in any way without the express written consent of Trivera Technologies LLC. For information please contact [Info@triveratech.com](mailto:Info@triveratech.com) or visit [www.triveratech.com](http://www.triveratech.com).

All software or hardware products referenced herein are trademarks of their respective holders. Products and company names are the trademarks and registered trademarks of their respective owners. Trivera Technologies has used its best efforts to distinguish proprietary trademarks from descriptive names by following the capitalization style used by the manufacturer.

**Copyright © 2019 Trivera Technologies LLC. All rights reserved.**



# Table of Contents

## Core Java 8 Programming

**Session: Java: A First Look .....** **13**

**Lesson: The Java Platform.....** **14**

Lesson Agenda .....	15
Defining Java™ .....	16
Java Provides Several Platforms .....	17
Note on Terminology .....	18
Java SE .....	19
Java SE Overview .....	21
Java SE Development Kit (JDK) .....	22
Executing Programs .....	23
Lifecycle of a Java Program.....	24
Responsibilities of JVM .....	25
Java is Dynamic: The Runtime Process .....	26
Garbage Collection.....	27
Documentation and Code Reuse .....	28
JavaDoc Provides Documentation Delivery .....	29
Example of Top-Level Documentation .....	30
Documentation at the Package Level .....	31
Documentation at the Class Level .....	32
Documentation at the Lowest Level.....	33
In-Line Comments are Translated.....	34
...into HTML Rendering .....	35
Lesson Review and Summary .....	36

**Lesson: Using the JDK .....** **37**

Lesson Objectives .....	38
Setting up the Environment.....	39
The Development Process.....	40
Compiling Package Classes .....	41
Source and Class Files .....	42
Java Applications .....	43
Student Activity (10 minutes) – Your Java .....	44
Lesson Review and Summary .....	45
Exercise: Exploring MemoryViewer .....	46
Exercise: Exploring ColorPicker .....	47

**Session: Getting Started with Java .....** **48**

**Lesson: Writing a Simple Class .....** **49**

Lesson Objectives .....	50
Classes in Java .....	51
What Is a Class? .....	52

Basic Java Syntax for a Class.....	53
Defining the Class .....	54
Access Modifiers .....	56
Class Instance Variables.....	57
A Class Can Define Data Structures.....	58
Primitives in Java .....	59
Primitives vs. Object References .....	60
Creating Objects.....	61
The main Method .....	62
Basic Java Syntax for a main Method.....	63
Using the Dot Operator .....	64
Writing Output .....	65
The Employee class.....	66
Java Keywords.....	67
Lesson Review and Summary .....	68
Exercise: Create a Simple Class.....	69
<b>Lesson: Adding Methods to the Class .....</b>	<b>70</b>
Lesson Objectives .....	71
Instance Methods .....	72
Passing Parameters Into Methods.....	73
Returning a Value From a Method .....	74
Overloaded Methods .....	75
Overloaded Methods Diagram .....	76
Constructors .....	77
Defining a Constructor.....	78
Optimizing Constructors .....	79
Lesson Review and Summary .....	80
Exercise 2: Create a Class with Methods .....	81
<b>Lesson: Language Statements .....</b>	<b>82</b>
Lesson Objectives .....	83
Operators .....	84
Short hand increment/decrement operators .....	85
Comparison Operators .....	86
The if statement.....	87
Logical Operators .....	88
The if / else statements .....	89
The ternary operator.....	90
Looping: The for Statement.....	91
Looping: The while Statement.....	93
Looping: The do Statement.....	94
Continue and Break Statements .....	95
The switch Statement.....	96
Lesson Review and Summary .....	98
Exercise 3: Looping.....	99
Exercise 4: Language Statements .....	100

<b>Lesson: Using Strings .....</b>	<b>101</b>
Lesson Objectives .....	102
Strings .....	103
String Methods .....	104
String Equality .....	105
String substring (Java 6) .....	106
Java substring (Java 6) .....	107
StringBuffer .....	108
String, StringBuffer, and StringBuilder .....	109
Lesson Review and Summary .....	110
Exercise 5: Fun with Strings.....	111
Exercise 6: Using StringBuffers and StringBuilder.....	112
<b>Lesson: Specializing in a Subclass.....</b>	<b>113</b>
Objectives: Specializing Subclasses.....	114
Extending a Class .....	115
The extends Keyword.....	117
Upcasting .....	118
Downcasting.....	119
Overriding Superclass Methods .....	120
Method Overriding.....	121
Calling Superclass Methods From Subclass .....	122
The Object Class.....	123
The instanceof Keyword.....	124
Use instanceof Before Downcasting .....	125
The instanceof Compiler Check .....	126
Object Equality .....	127
The Object.equals Method .....	128
Overriding the equals Method .....	129
The equals and hashCode Methods .....	130
Default Constructor .....	131
Implicit Constructor Chaining .....	132
Passing Data Up Constructor Chain .....	133
A Common Programming Mistake .....	134
Lesson Review and Summary .....	135
Exercise 7: Creating Subclasses .....	136
<b>Session: Essential Java Programming .....</b>	<b>137</b>
<b>Lesson: Fields and Variables.....</b>	<b>138</b>
Lesson Objectives .....	139
Fields vs. Variables .....	140
Instance vs. Local Variables: Usage Differences.....	141
Data Types and Variables .....	142
Default Values .....	143
Block Scoping Rules .....	144
Using this.....	145
Final and Static Variables .....	146

Static Variable Diagram .....	147
Static Fields.....	148
Simple Example of Static Fields.....	149
Static Methods.....	150
Lesson Review and Summary .....	151
Exercise 8: Field Test.....	152
<b>Lesson: Using Arrays .....</b>	<b>153</b>
Objectives: Using Arrays.....	154
Arrays .....	155
Accessing the Array .....	156
Multidimensional Arrays .....	157
Copying Arrays.....	158
Variable Argument Support.....	159
Varargs Rules.....	161
Lesson Review and Summary .....	162
Exercise 9: Creating an Array .....	163
<b>Lesson: Java Packages and Visibility.....</b>	<b>164</b>
Lesson Agenda .....	165
The Problem.....	166
Packages.....	167
Class Location of Packages.....	168
The package Keyword.....	169
Importing Classes.....	170
Executing Programs .....	171
Accessibility/Visibility .....	172
Java Naming Conventions .....	173
Lesson Review and Summary .....	174
Student Activity (10 minutes) – Refactoring in Your IDE .....	175
<b>Session: Advanced Java Programming.....</b>	<b>176</b>
<b>Lesson: Inheritance and Polymorphism .....</b>	<b>177</b>
Lesson Agenda .....	178
Polymorphism.....	179
Polymorphism: The Subclasses.....	180
Derived Classes as the Superclass .....	181
Casting to the Derived Class.....	182
Using instanceof for Downcasting.....	183
Upcasting vs. Downcasting .....	184
Calling Superclass Methods From Subclass .....	185
The final Keyword.....	186
Using final Variables.....	187
‘Blank final’ Fields and Variables .....	188
Lesson Summary .....	189
Exercise 10: Salaries - Polymorphism .....	190

<b>Lesson: Interfaces and Abstract Classes .....</b>	<b>191</b>
Lesson Agenda .....	192
Separating Capability from Implementation .....	193
Abstract Classes .....	194
Shape as an Abstract Class.....	195
Polymorphism With Abstract Classes .....	196
Interfaces.....	197
Implementing an Interface.....	199
Extending Interfaces.....	200
Polymorphism With Interfaces .....	201
Type Checking .....	202
Abstract Classes vs. Interfaces .....	203
Lesson Review and Summary .....	204
Exercise 11: Mailable - Interfaces .....	205
<b>Lesson: Exceptions .....</b>	<b>206</b>
Lesson Agenda .....	207
What is an Exception?.....	208
Exception Architecture .....	209
Handling Exceptions.....	210
The Throwable Class .....	211
Exception Hierarchy .....	212
The try Block .....	213
The catch Block.....	214
Handling Multiple Exceptions .....	215
The finally Block .....	216
Automatic Closure of Resources.....	217
Full Example of Exception Handling .....	218
Generalized vs. Specialized Exceptions .....	219
Overriding Methods .....	220
Creating Your Own Exceptions .....	221
Throwing Exceptions .....	222
Re-throwing an Exception .....	223
Checked vs. Unchecked Exceptions.....	224
Exception Hierarchy .....	225
Checked vs. Unchecked Exceptions.....	226
Lesson Review and Summary .....	227
Exercise 12: Exceptions .....	228
<b>Session: Java Developer's Toolbox .....</b>	<b>229</b>
<b>Lesson: Utility Classes .....</b>	<b>230</b>
Objectives: Utility Classes.....	231
Wrapper Classes.....	232
The Number Class .....	233
Numbers and Strings.....	234
Autoboxing/Unboxing .....	235
Autoboxing/Unboxing Issues.....	236

Big Decimal .....	237
Decimal Formatting .....	238
The Date Class.....	239
Lesson Review and Summary .....	240
Exercise 13: Using Primitive Wrappers.....	241
<b>Lesson: Enumerations and Static Imports .....</b>	<b>242</b>
Lesson Agenda .....	243
Rationale for Enumerations.....	244
Enumeration Syntax .....	245
Implementing Enums.....	246
Enumeration Example .....	247
Enumeration and Abstract Methods.....	248
Enumerations as a Better Class Type.....	249
When You Should Use Enumerations.....	250
Using Enumeration in Switch Statements.....	251
Static Imports .....	252
Static Imports - Fields.....	253
Static Imports - Methods .....	254
When (Not) to Use Static Imports .....	255
Lesson Review and Summary .....	256
Exercise 14: Enumerations .....	257
<b>Lesson: Introduction to Annotations .....</b>	<b>258</b>
Objectives: Intro to Annotations .....	259
Annotations Overview .....	260
@Deprecated .....	262
@Override .....	263
@SuppressWarnings .....	264
@FunctionalInterface (Java 8) .....	266
Meta-annotations.....	267
Writing a Custom Annotation .....	269
Annotation Members .....	270
Default Member Values.....	271
Java Reflection API .....	272
Obtaining Annotation Information.....	273
Annotations Applied .....	275
Annotations Applied (Java EE) .....	276
Annotations Applied (JUnit).....	277
Lesson Review and Summary .....	278
Exercise 15: Annotations .....	279
<b>Session: Collections and Generics .....</b>	<b>280</b>
<b>Lesson: Introduction to Generics .....</b>	<b>281</b>
Lesson Agenda .....	282
Introduction to Generics .....	283
Writing a List.....	284

Using the List Class.....	285
Generic Class.....	287
Using Generic Classes.....	288
Type Parameter Naming Conventions.....	289
Bounded Types .....	290
Bounded Types (Upper Bound) .....	291
Bounded Types .....	292
Raw Types .....	293
Generic Methods.....	294
Generics and Subtypes.....	296
Generics and Subtypes (Example).....	298
Wildcards.....	299
Bounded Wildcards .....	301
Lesson Review and Summary .....	302
Tutorial: Setup JUnit 4 Project library in IntelliJ .....	303
Exercise 16: DynamicArray .....	304
<b>Lesson: Collections .....</b>	<b>305</b>
Lesson Agenda .....	306
The Collections Framework .....	307
Characterizing Collections .....	308
Indexed, Linked or Hashed Implementations.....	309
Features of the Collection Implementation Classes.....	310
Collection Interface Hierarchy .....	311
Top Level Collection Interfaces .....	312
Collection<E> Interface .....	313
Optional Methods .....	314
Collections Class.....	315
java.util.Iterator Interface.....	316
The Set<E> Interface .....	317
SortedSet<E> Interface.....	318
Comparable<T> Interface .....	319
Comparator<T> Interface .....	320
NavigableSet<E> Interface .....	321
The List<E> Interface .....	322
ListIterator<E> Interface.....	323
Queue<E> Interface .....	324
Map Interfaces.....	325
Creating and Using a HashMap<K,V> .....	326
Feature Comparison.....	327
Using the Right Collection.....	328
Collections and Multithreading .....	330
Optimizing Collection Constructors – Initial Size .....	331
Lesson Review and Summary .....	332
Exercise 17: Using Hashtable and HashMap .....	333
Exercise 18: Collections Poker .....	334

**Session: Lambda Expressions; Collections and Streams ..... 335****Lesson: Introduction to Lambda Expressions ..... 336**

Lesson Agenda .....	337
Functional vs OO Programming .....	338
Functional Interfaces .....	339
The FunctionalInterface Annotation .....	340
Introduction to Lambda Expressions.....	341
Lambda Expression Syntax .....	342
Utility Methods.....	343
Implementing Functions .....	344
Inner Classes .....	345
Anonymous Inner Classes .....	346
Transforming to Lambda .....	347
'Optimizing' Lambda Expression .....	350
Lambda Expression Parameters.....	352
Lambda Expression Body .....	353
Functional Interfaces Which Return Data .....	354
Functional Interfaces and Generics .....	355
java.util.function Package .....	357
Default methods on Functional Interfaces .....	359
Final Variables.....	360
Method References.....	361
Static method references .....	363
Instance method references.....	364
Reference method of specific instance .....	365
Constructor References .....	366
Lambda Expression vs Anonymous Class.....	367
Lesson Review and Summary .....	368

**Lesson: Java 8 Collection Updates ..... 369**

Lesson Agenda .....	370
Collections and Lambda Expressions.....	371
Iterable Interface .....	372
Spliterator Interface .....	373
Spliterator Characteristics .....	375
Methods Added to Collection .....	376
List Interface.....	377
Comparator Static Methods .....	378
Comparator Methods.....	380
The Map Interface .....	382
The Map Interface.....	384
ConcurrentHashMap Class .....	385
ConcurrentHashMap forEach.....	386
ConcurrentHashMap search .....	388
ConcurrentHashMap Reducing .....	389
Lesson Review and Summary .....	390
Exercise 19: Functional Collections .....	391

<b>Lesson: Streams .....</b>	<b>392</b>
Objectives: Streams .....	393
Collections of Data .....	394
Declarative ‘Programming’ .....	395
‘Garbage Variables’ .....	396
Large Collections.....	397
Java Stream API .....	398
Collections and Streams .....	399
The Flight Example .....	400
java.util.stream.Stream Interface .....	401
Intermediate Operations.....	402
Terminal Operations.....	403
Collectors .....	404
Collectors Static Import .....	405
Filtering.....	406
Truncating .....	407
Skipping Elements.....	408
Mapping.....	409
Filter Using distinct.....	410
Matching .....	411
Finding Elements.....	412
Optional Overview .....	413
Stream ‘Sources’ .....	414
Numeric Streams.....	415
Static Methods on Numeric Streams.....	416
Reducing .....	417
forEach Terminal Operation .....	418
Lesson Review and Summary .....	419
Exercise 20: Working with Streams .....	420
<b>Lesson: Collectors .....</b>	<b>421</b>
Lesson Agenda .....	422
Introduction to Collectors .....	423
Collectors and Collector .....	424
Creating collection(s) from Stream.....	425
Grouping with Streams.....	426
Joining Elements .....	427
Partitioning .....	428
Counting and Summing .....	429
Lesson Review and Summary .....	430
Exercise 21: Collecting.....	431
<b>Session: Supplemental Materials (Time Permitting).....</b>	<b>432</b>
<b>Lesson: The new Date/Time API .....</b>	<b>433</b>
Objectives: Date and Time API .....	434
What’s Wrong With java.util.Date?.....	435
java.time .....	436

Core Classes of Date/Time API .....	437
Date/Time Hierarchy .....	438
DayOfWeek Enum.....	439
Month Enum .....	441
The LocalDate Class .....	442
TemporalField and ChronoField.....	444
Instant Class.....	445
Duration Class.....	446
Manipulating Dates.....	447
TemporalAdjuster Interface .....	448
Using the TemporalAdjusters Class.....	449
Creating Custom Adjusters .....	450
Formatting Date/Time.....	451
Working With Time Zones .....	452
Legacy Date Code.....	454
Lesson Review and Summary .....	455
Exercise 22: Agenda .....	456
<b>Lesson: Formatting Strings .....</b>	<b>457</b>
Objectives: String Formatting .....	458
java.util.StringJoiner .....	459
printf .....	460
Format Specifier Syntax .....	461
Type Conversion Categories.....	462
Type Conversions .....	463
The Formatter Class.....	464
System.out.printf / System.out.format.....	466
String.format.....	467
Formatting a String .....	468
Formatting Numeric Types .....	469
Argument Index .....	470
Formatting Date/Time.....	471
Formatting Times .....	472
Formatting Dates.....	473
Formatting Date/Time.....	474
Lesson Review and Summary .....	475
<b>Lesson: Java Data Access JDBC API .....</b>	<b>476</b>
Lesson Agenda .....	477
What is JDBC? .....	478
Structured Query Language (SQL) .....	479
Connecting to the Database.....	480
Initializing java.sql.DriverManager .....	481
Connecting to the Database (Java SE).....	482
java.sql.Statement.....	483
java.sql.PreparedStatement.....	484
java.sql.ResultSet.....	485
Executing Inserts, Updates and Deletes .....	486

Controlling Transactions .....	487
Mapping SQL Types to Java Types .....	488
Obtaining Database Metadata .....	489
Obtaining ResultSet Metadata .....	490
Stored Procedures Defined.....	491
Stored Procedure Parameters .....	492
javax.sql.RowSet.....	493
Connected vs Disconnected RowSets .....	494
javax.sql.rowset.JdbcRowSet .....	495
Executing a Statement .....	496
Navigating Through RowSet .....	497
JdbcRowSet – Retrieving Data .....	498
javax.sql.rowset.CachedRowSet.....	499
CachedRowSet Example .....	500
Lesson Review and Summary .....	501
Tutorial: Setup The Derby Database.....	502
Exercise 23: Reading Table Data .....	503
Exercise 24: Using JdbcRowSet.....	504
Exercise 25: Executing within a Transaction .....	505
<b>Lesson: Unit Testing Fundamentals .....</b>	<b>506</b>
What is Unit Testing? .....	507
What Is Unit Testing? (white-box/black-box) .....	509
Purpose of Unit Testing.....	510
Why Do We Write Unit Tests? .....	511
Good Unit Tests .....	512
Test Stages .....	514
Unit Test Stage.....	515
Integration Test Stage .....	516
Unit Testing vs Integration Testing.	517
Functional Testing .....	518
Non-Functional Testing .....	519
Understanding Unit Testing Frameworks.....	520
So What is a "Good" Test?.....	521
Good: Readable Equates to Maintainable .....	522
Proper Organization and Structure .....	523
Test the Right Thing .....	524
Run in Solitude .....	525
Reduce and Manage Dependencies.....	526
Reliability .....	527
To Sum Up.....	528
Keys to Success.....	529
<b>Lesson: Jumpstart: JUnit 4.x .....</b>	<b>530</b>
Lesson Agenda .....	531
JUnit Overview .....	532
JUnit Design Goals.....	533
JUnit Features .....	534

Reasons to Use JUnit.....	535
How JUnit Works.....	536
Test Case using JUnit .....	537
Exploring JUnit .....	538
Writing the TestCase.....	539
Test Result Verification (Assertions).....	540
Launching Tests .....	541
Failures vs. Errors .....	542
Introducing Class Message .....	543
Creating Class MessageTest .....	544
The First Test Implementation Steps .....	545
Testing the Constructor .....	546
Running a Test in an IDE .....	547
Running a Test From the Command Line .....	548
Seeing Results of a Test: JUnit View .....	549
Using the Results of a Test .....	550
Seeing Results of a Successful Test.....	551
Test Suites .....	552
Composing Tests Using Suite.....	553
JUnit Test Fixture .....	554
JUnit Method Lifecycle .....	555
Managing Resources with Fixtures .....	556
Share Expensive Setups.....	557
Review.....	559

# Core Java 8 Programming

**TT2105**

## **TT2105 Core Java 8 Programming**

All written content, source code, and formatting are the property of Trivera Technologies LLC. No portion of this material may be duplicated or reused in any way without the prior express written consent of Trivera Technologies LLC.

All Products and company names are the trademarks of their respective owners. Trivera Technologies has used its best efforts to distinguish proprietary trademarks from descriptive names by following the capitalization style used by the manufacturer.

**Copyright © 2019 Trivera Technologies LLC. All rights reserved.**

**Version 20190527**

**Trivera Technologies | Collaborative IT Education Solutions**

**Training | Mentoring | Courseware**

In-Person | Online Live | Self-Paced | Private Onsite | Public Schedule | Custom Programs

## About This Course

- **Core Java 8 Programming is a fast-paced, quick start to Java 8 training course geared for experienced developers who have some exposure to object-oriented programming language.**

Welcome to Core Java 8 Programming

Thank you for choosing us to provide you with the very highest standard of technical training!

We take great pride in our attention to detail in designing the very best quality courseware in the industry.

This course is the result of many developer-months of planning, design, development, and review. Every course is very carefully designed using proper guidelines for instructional technology, including the use of specific performance objectives, relevant laboratory exercises, and professional quality content layout to ensure the most effective transfer of knowledge.

Because we realize that no work is ever perfect, we are constantly striving to improve our materials. Throughout this course you will have ample opportunity to provide us with your feedback on all aspects of this course. Please do take the time to give us your input. It is extremely valuable to us.

Again, many thanks for entrusting us with your educational needs. Enjoy the course!

# Workshop Agenda

- **Session: Java: A First Look**
  - **Lesson: The Java Platform**
  - **Lesson: Using the JDK**
    - ◆ **Exercise: Exploring MemoryViewer**
    - ◆ **Exercise: Exploring ColorPicker**
- **Session: Getting Started with Java**
  - **Lesson: Writing a Simple Class**
    - ◆ **Exercise: Create a Simple Class**
  - **Lesson: Adding Methods to the Class**
    - ◆ **Exercise: Create a Class with Methods**
  - **Lesson: Language Statements**
    - ◆ **Exercise: Looping**

## Workshop Agenda (cont'd)

- **Session: Getting Started with Java (cont'd)**
  - **Lesson: Language Statements (cont'd)**
    - ◆ **Exercise: Language Statements**
  - **Lesson: Using Strings**
    - ◆ **Exercise: Fun with Strings**
    - ◆ **Exercise: Using StringBuffer and StringBuilder**
  - **Lesson: Specializing in a Subclass**
    - ◆ **Exercise: Creating Subclasses**
- **Session: Essential Java Programming**
  - **Lesson: Fields and Variables**
    - ◆ **Exercise: Field Test**
  - **Lesson: Using Arrays**

## Workshop Agenda (cont'd)

- **Session: Essential Java Programming (cont'd)**
  - **Lesson: Using Arrays (cont'd)**
    - ◆ **Exercise: Creating an Array**
  - **Lesson: Java Packages and Visibility**
- **Session: Advanced Java Programming**
  - **Lesson: Inheritance and Polymorphism**
    - ◆ **Exercise: Salaries - Polymorphism**
  - **Lesson: Interfaces and Abstract Classes**
    - ◆ **Exercise: Mailable - Interfaces**
  - **Lesson: Exceptions**
    - ◆ **Exercise: Exceptions**

## Workshop Agenda (cont'd)

- **Session: Java Developer's Toolbox**
  - **Lesson: Utility Classes**
    - ◆ **Exercise: Using Primitive Wrappers**
  - **Lesson: Enumerations and Static Imports**
    - ◆ **Exercise: Enumerations**
  - **Lesson: Introduction to Annotations**
    - ◆ **Exercise: Annotations**
- **Session: Collections and Generics**
  - **Lesson: Introduction to Generics**
    - ◆ **Tutorial: Setup JUnit 4 Project library in IntelliJ**
    - ◆ **Exercise: DynamicArray**
  - **Lesson: Collections**

## Workshop Agenda (cont'd)

- **Session: Collections and Generics (cont'd)**
  - **Lesson: Collections (cont'd)**
    - ◆ **Exercise: Using Hashtable and HashMap**
    - ◆ **Exercise: Collections Poker**
- **Session: Lambda Expressions; Collections and Streams**
  - **Lesson: Introduction to Lambda Expressions**
  - **Lesson: Java 8 Collection Updates**
    - ◆ **Exercise: Functional Collections**
  - **Lesson: Streams**
    - ◆ **Exercise: Working with Streams**
  - **Lesson: Collectors**
    - ◆ **Exercise: Collecting**

## Workshop Agenda (cont'd)

- **Session: Supplemental Materials (Time Permitting)**
  - **Lesson: The new Date/Time API**
    - ◆ **Exercise: Agenda**
  - **Lesson: Formatting Strings**
  - **Lesson: Java Data Access JDBC API**
    - ◆ **Tutorial: Setup The Derby Database**
    - ◆ **Exercise: Reading Table Data**
    - ◆ **Exercise: Using JdbcRowSet**
    - ◆ **Exercise: Executing within a Transaction**
  - **Lesson: Unit Testing Fundamentals**
  - **Lesson: Jumpstart: JUnit 4.x**

## Workshop Agenda (cont'd)

- **Supplemental Exercises (Time Permitting)**

- ◆ **Exercise: Defining the Student Subclass**
- ◆ **Exercise: Defining the Student Array**
- ◆ **Exercise: Define Java packages**
- ◆ **Exercise: Adding Generics to Dynamic Array**
- ◆ **Exercise: Writing a Collection**

# Formalities

- **Introductions**
- **Logistics (breaks, facilities, lunch, etc.)**
- **Rules of Classroom Engagement**
- **Let's Get Started!**

## Student Introductions

- Name
- Company
- Role
- Previous projects of interest
- Current project
- Languages/Areas of expertise
- Why are you taking this workshop?

### Getting to Know You

The best way for your instructor to help you in learning is for him/her to understand a little bit about your background. When it's your turn, please don't be shy! Speak up so everyone in the class can hear you. You will probably find that there is at least one other person in the class with a background and set of goals similar to your own. Much of the value in taking a class is the networking that can occur between developers.

When each of the class members have had the opportunity to talk about their background and goals, the instructor will spend a few minutes sharing his/her own background, including qualifications for teaching this course.

## Questions before we begin?

Ready?

Before moving on into the course contents take a moment to digest everything you've seen so far. Feel free to ask your instructor any questions that will make you feel more comfortable with the class.

## **Session: Java: A First Look**

**The Java Platform  
Using the JDK**

## **Lesson: The Java Platform**

**The Java Platform**  
Using the JDK

## Lesson Agenda

- **Understand what Java is**
- **Understand which tools to use**
- **Understand different versions of Java**

## Defining Java™

- **Java is a platform not just a language**
- **Java Language is part of Java Platform**
- **API to the platform is accessed in java**
- **Java Platform can run on:**
  - Other OS
  - Cars
  - Cards
  - Rings
  - PDAs
- **When running on another OS, you need a Virtual Machine**

## Java Provides Several Platforms

- **Java Standard Edition (Java SE)**
  - Used on desktop computers and high end small devices
  - Contains some APIs that are essential to enterprise applications (e.g., JDBC and RMI)
- **Java Enterprise Edition (Java EE)**
  - Used to build large scale multi-tier enterprise applications
  - Built on top of Java SE
  - Contains enterprise-specific APIs such as EJB
- **Java Micro Edition (Java ME)**
  - Used on small devices such as PDAs (e.g., Palm Pilots) and cell phones
  - Includes subset of Java SE
  - Sub-divided into profiles

## Note on Terminology

- The terminology changed during the transitions from Java 1.4 to Java 5:
  - J2SE → Java SE
  - J2EE → Java EE
  - J2ME → Java ME
- Cleaning up the version numbers
  - J2SE 1.4.2 →
    - ◆ Java SE 5, Java SE 6, Java SE 7, Java SE 8 and Java SE 9
  - Java EE 1.4 →
    - ◆ Java EE 5, Java EE 6, Java EE 7 and Java EE 8

Note that from here on out, we will be using the latest terminology.

## Java SE

- **The Java Platform, Standard Edition is at the core of Java technology**
- **Support for many areas of functionality**
  - **Essentials:** Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on
  - **Applets:** Set of conventions used by applets
  - **Networking:** URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses
  - **Internationalization:** Help for writing programs that can be localized for users worldwide

## Java SE (cont'd)

- **Security:** Both low level and high level, including electronic signatures, key management, access control, and certificates
- **Software components:** JavaBeans can plug into existing component architectures
- **Object serialization:** Allows lightweight persistence and communication via RMI
- **Java Database Connectivity (JDBC):** Uniform access to a wide range of relational databases
- **APIs for 2D/3D graphics, accessibility, servers, collaboration, telephony, speech, and more ...**

Java SE is essentially a specification and various entities have provided their implementations of that specification in the form of Java Development Kits (JDKs) and Java Runtime Environments (JREs).

## Java SE Overview



From the Java API documentation which can be found at  
<http://docs.oracle.com/javase/7/docs/>

## Java SE Development Kit (JDK)

- The JDK is a set of software and software tools, supplied by Oracle, that includes all the basic components needed to build Java applications, applets and servlets
- The major components include:
  - The Java Core API (Application Programming Interface)
  - javac - Java compiler
  - java - Java bytecode interpreter (Java Virtual Machine)
  - javadoc - Generates documentation from source code
  - jar - Create and extract Java Archives
  - javap – Disassembles class files
  - jdb - Basic command line debugger
  - ...

Tools are documented in the docs directory of the JDK under

<JAVA\_HOME>\docs\tooldocs\tools.html

javac - The Java compiler

java - The Java bytecode interpreter (Java Virtual Machine)

The Java Core API (Application Programming Interface) classes - The set of class libraries supplied with Java

javadoc - A tool to generate documentation from Java source code

jar - A tool to create and extract archive files, modeled after the UNIX tar utility

javap - Disassembles class files

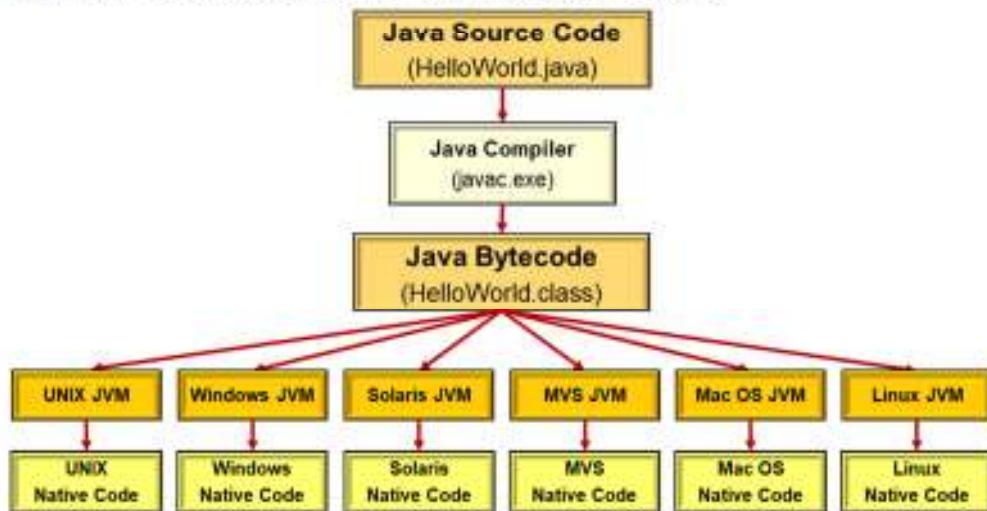
jdb - A basic command line debugger for Java programs

## Executing Programs

- **Java platform "runs" classes**
- **Classes are compiled java sources**
- **Compiled class is byte code**
- **Byte code is native compiled code for the Java Platform**

## Lifecycle of a Java Program

- Bytecode is partially compiled code
- JVM converts it to native code at runtime

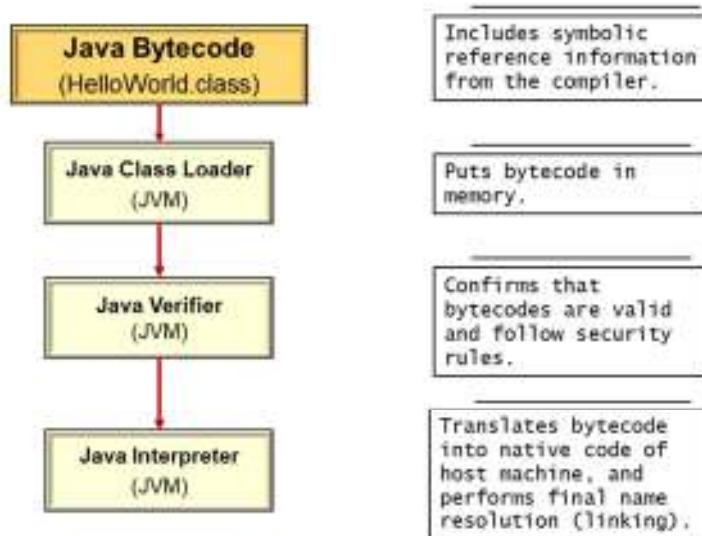


The JVMs for Solaris and for Windows are available from the Oracle site.  
Other platforms are available from their respective vendors.

## Responsibilities of JVM

- **Loading of classes**
  - Loading the byte code, class verification, preparation, resolution, and initialization
- **Garbage collection**
  - Freeing memory that is not being used anymore
- **Security**
  - Protects against un-initialized memory, illegal type conversions and the allowed stack operations
- **Interfacing to native (platform-specific) code**

## Java is Dynamic: The Runtime Process



## Garbage Collection

- **Background thread in JVM**
- **Searches for unreachable objects**
- **When unreachable objects are found**
  - The object's finalize method may be called
  - The object is re-examined to see if it is still unreferenced
  - If still unreferenced, the object memory is marked for reclamation
- **The garbage collector cannot be triggered explicitly**

...  
`System.gc();`  
...

- **This is only a request for garbage collection, not a command**

### Forcing Garbage Collection

Sorry, you can't. You can supply strong hints to the garbage collector like setting an object reference to null. You can request the garbage collector to run, but you cannot force it to run.

When the garbage collector runs, it will decide how much memory to reclaim, and which objects to reclaim. Again, you cannot control this.

## Documentation and Code Reuse

- Object-oriented programming languages have been around since the 1950s holding the promise of code reuse
  - Never really got there
- In order to reuse code, it must be documented in a familiar and easy to use manner
  - Must be easy to develop and maintain as well
- Java has provided Javadocs to support such documentation

## JavaDoc Provides Documentation Delivery

- Javadoc tool comes with the JDK utilities
- Tool compiles a hierarchical set of comments by extracting them from the source code itself
- Tool then generates an HTML-based representation of those comments
- Since the delivery of the documentation is in HTML, it is universally accessible
- Since the organization and presentation of the documentation is consistent, stakeholders develop a high comfort level with it

## Example of Top-Level Documentation

The screenshot shows the Java Platform, Standard Edition 8 API Specification documentation. The left sidebar lists packages such as java.util, java.awt, and javax.swing. The main content area is titled "java™ Platform, Standard Edition 8 API Specification" and contains a brief introduction: "The document is the API specification for the Java™ Platform, Standard Edition". Below this is a "Profiles" section with three items: JavaSE-11, JavaSE-14, and JavaSE-15. The main content area also features a "Packages" section with a table mapping package names to their descriptions:

Package	Description
java.awt	Provides the classes necessary to create an output and the classes for displaying input or communication with the system.
java.awt.image	Provides classes for image I/O.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.print	Provides all of the classes for creating print operations and for printing graphics and images.
java.awt.event	Provides classes for user input.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.image	Provides all of the classes for creating print operations and for printing graphics and images.
java.awt.print	Provides interfaces and classes for printing with different types of printers bind by AWT components.
java.awt.event	Provides classes and interfaces relating to events.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on regions related to two-dimensional geometry.
java.awt.image	Provides classes and interfaces for the image and font framework.
java.awt.print	Provides interfaces that enable the development of print facilities that can be used with any Java™ printer and monitor.
java.awt.image	Provides classes for creating and manipulating images.

## Documentation at the Package Level

**Package java.lang**

Contains classes that are fundamental to the design of the Java programming language.

See Description

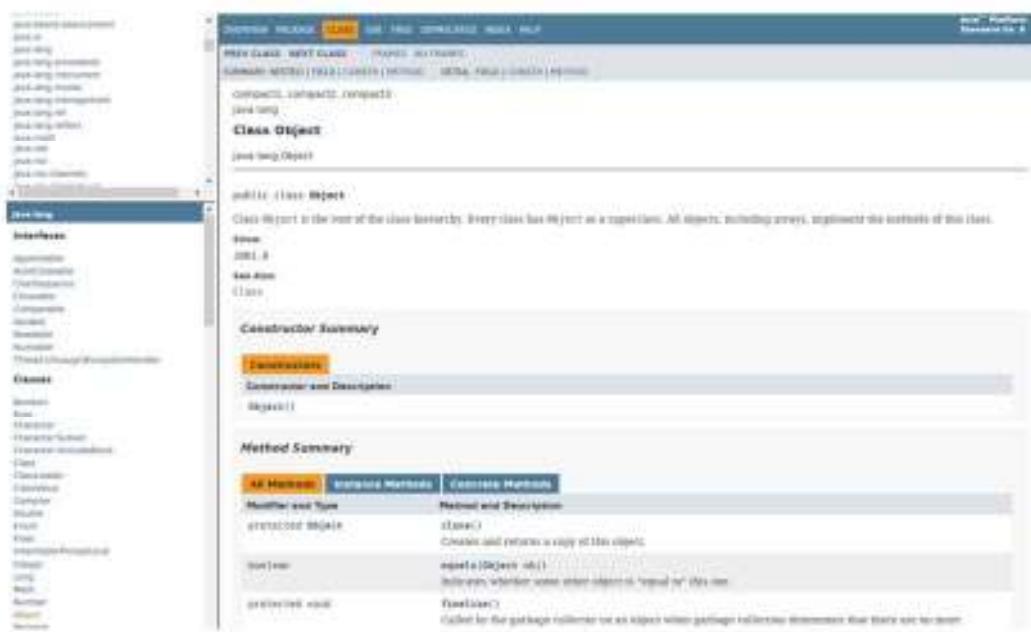
**INTERFACE SUMMARY**

Interface	Description
<code>Comparable</code>	An object to which the <code>compareTo</code> method is applied.
<code>Cloneable</code>	An object that has field <code>cloneable</code> (such as the <code>Serializable</code> field) is cloned.
<code>CharSequence</code>	A CharSequence is a readable sequence of char values.
<code>Comparable&lt;T&gt;</code>	A class implements the Comparable interface to indicate to the <code>Object.compareTo</code> method that it is equal to or positioned in relation to other objects of that class.
<code>Serializable</code>	Implements this interface to define a source of characters.
<code>Serializable</code>	The Serializable interface should be implemented by one class, whose instances are intended to be received by a client.
<code>Thread.UncaughtExceptionHandler</code>	Specifies the handler thread behavior if UncaughtException occurs. This is a stronger exception.

**CLASS SUMMARY**

Class	Description
<code>Boolean</code>	The Boolean class wraps a value of the primitive type boolean in an object.
<code>Byte</code>	The Byte class wraps a value of primitive type byte in an object.
<code>Character</code>	The Character class wraps a value of the primitive type char in an object.
<code>Integer</code>	Registers of this class represent primitive values of the <code>int</code> class.
<code>Long</code>	Registers of this class represent primitive values of the <code>long</code> class.
<code>Number</code>	Registers of this class represent primitive values of the <code>double</code> class.
<code>Short</code>	Registers of this class represent primitive values of the <code>short</code> class.

## Documentation at the Class Level



## Documentation at the Lowest Level

**equals**

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects. That is, for any non-null reference values `x` and `y`, `x.equals(y)` returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the same value).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method which states that equal objects must have equal hash codes.

**Parameters:**  
`obj` — the reference object with which to compare.

**Returns:**  
true if this object is the same as the `obj` argument; false otherwise.

**See also:**  
`hashCode()`, `finalize`

## In-Line Comments are Translated...

```
package com.triveratech.xtensil.ui.toolbox;

import java.awt.BorderLayout;

/**Main window is the central point of the application display. It is basically
 * set up as a tabbed pane interface with each tab accessing different
 * functionality.
 * @author      Dan Corsberg
 * @version     4.0
 */

public class XtensilMainWindow extends JFrame{

    //Set up path for help files.
    protected String helpFile;
```

## ...into HTML Rendering

com.triveratech.xtensil.api  
**Class XtensilMainWindow**

```
java.lang.Object
  ↳ java.awt.Component
    ↳ java.awt.Container
      ↳ java.awt.Window
        ↳ java.awt.Frame
          ↳ javax.swing.JFrame
            ↳ com.triveratech.xtensil.api.toolbox.XtensilMainWindow
```

**All Implemented Interfaces:**

```
java.awt.image.ImageObserver, java.awt.MimicContainer, java.io.Serializable, java.accessibility.Accessible, javax.swing.RootPaneContainer,
java.awt.WindowConstants
```

```
public class XtensilMainWindow
extends javax.swing.JFrame
```

Main window is the central point of the application display. It is basically set up as a tabbed pane interface with each tab accessing different functionality.

**Versions:**

4.0

**Author:**

Dan Cederberg

**See Also:**

[SmokedForm](#)

### Field Summary

```
static com.triveratech.xtensil.api.toolbox.AboutFrame ABOUTFRAME
```

The tab (or tray in the toolbox) for About

## Lesson Review and Summary

- 1) Is the Java language platform independent?**
- 2) Name the steps to make a java program run?**

- 1) No, Java programs only run on the Java Platform
- 2) Code, compile (using javac) and execute (using java)

## **Lesson: Using the JDK**

The Java Platform  
**Using the JDK**

## Lesson Objectives

- After completing this lesson, you will be able to:
  - Use the command line compiler to compile a Java class
  - Use the command line Java interpreter to run a Java application class
  - Correctly set up the CLASSPATH environment variable to allow the compiler and interpreter to run correctly

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Setting up the Environment

- **JAVA\_HOME**
  - Points to your installation directory
- **PATH settings**
  - Should have JAVA\_HOME \bin
- **CLASSPATH settings**
  - Location of class files in order to be able to execute
  - JRE classes (depends on java version)
  - Any libraries that are not part of JDK (especially Your classes-root)
- **CLASSPATH can also be set when invoking java (covered later)**

## The Development Process

- We write source code
  - Kept in a .java file
  - File name is case sensitive and matches class name
- We compile the source code into byte code
  - Compiler reads .java file(s) and writes .class file(s)
  - The base name remains the same

```
PROMPT> javac MyClass.java
```

- We execute the .class file

```
PROMPT> java MyClass
```

### Developing Java Applications

We must conform to some basic rules for representing Java source code.

Java source code is kept in a file with a suffix of .java. We can place multiple classes into the same file, but only one of those classes can be public. There can also be only one package keyword in the file, and must be placed at the top of the file.

When the source file is compiled, one or more files with the suffix of .class will be created. These files contain the bytecode, or instructions, that can be read by the Java interpreter. For each class defined in the source file, there will be a corresponding .class file. The base name of the class file is the name of the Java class

The class(es) can be tested by running the Java class that contains a main method.

## Compiling Package Classes

- Compiled class files are written to the current directory using full package path
  - The following option can change the output directory

```
javac -d /output/dir myPackage/MyClass.java
```
  - Otherwise, class files must be moved to appropriate location
- Rules for java files:
  - Each file may contain as many classes/interfaces you wish
  - Only one class may be public
  - If the source file contains a public class or public interface called X, the name of the file must be X.java
  - Each class or interface forms its own .class file
- One way to manage your files is to keep the source file in the same directory as the class file
- Another way is to always use `-d <classDir>` to maintain separate tree of class files

### Compiling to a Location in your CLASSPATH

To make it easier to maintain your class files in the correct location, the compiler has an option which allows you to specify the output directory of the compiler.

## Source and Class Files

- A Java source file must be named <ClassName>.java
  - The source file may contain many class definitions, but there can only be one public class in the source file
- When compiled, each class is stored in a separate .class file

### Source and Class Files

Java source files must be placed within files with the extension .java. Every .java file has an associated package. The classes defined in the file are members of that package. Each class within a package must have a distinct name. A .java file that specifies no package is associated with the anonymous package.

If more than one class is defined in a .java file, only one class may be public. The public class provides the filename (<ClassName>.java). Other classes in the file are used internally to the package and cannot be used outside of it.

When compiled, each class is stored in a separate .class file, with the classes name providing the file name.

## Java Applications

- Have the following method

```
public static void main(String[] args) { ... }
```

- To execute from the command line

```
Prompt> java MyMainClass
```

- Usually has access to run various system commands

```
void System.setIn(InputStream in);  
void System.setOut(PrintStream out);  
Properties System.getProperties();
```

### Applications

A Java application is represented by a class that has a main method. The application is executed by a runtime interpreter, generally the standard Java interpreter that is typically invoked from the command line.

A Java application can be graphical or non-graphical in nature. It generally has no security limitations, so it can perform such tasks as running system commands, making and receiving network connections, accessing the file system, and performing various operations on the Java runtime.

The disadvantage of a Java application is that it must usually be installed manually on a runtime system, such as copying from a CD or accessing it using a network file server.

## Student Activity (10 minutes) – Your Java

- Java is currently installed on your computer
- Open a DOS window and type "java" at the command line
- You should see a response that indicates the various options that are available in working with the Java runtime
- Determine the version of Java that you are running using "java –version"
- Type "java test" at the command line.
  - This will cause an error to be generated. What is the error and what is causing the error to occur
- Invoke the Java compiler and examine the options that are available for that tool

## Lesson Review and Summary

- 1) What is the name of the compiler?**
- 2) What is the name of the interpreter?**
- 3) How does the run-time environment find classes?**

### Lesson Review Answers:

1. Java source files are compiled into bytecode, or class files, using the `javac` utility
2. Application class files are executed using `java`
3. Using the Classpath environment variable

## **Exercise: Exploring MemoryViewer**

Please refer to the written lab exercise and follow the directions as provided by your instructor

## **Exercise: Exploring ColorPicker**

Please refer to the written lab exercise and follow the directions as provided by your instructor

## **Session: Getting Started with Java**

**Writing a Simple Class  
Adding Methods to the Class  
Language Statements  
Using Strings  
Specializing in a Subclass**

## **Lesson: Writing a Simple Class**

### **Writing a Simple Class**

Adding Methods to the Class

Language Statements

Using Strings

Specializing in a Subclass

## Lesson Objectives

- After completing this lesson, you will be able to:
  - Write a Java class that does not explicitly extend another class
  - Define instance variables for a Java class
  - Implement a main method to create an instance of the defined class

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Classes in Java

- All code in Java is implemented within a **class**
- Objects are defined by **classes**
- A class is composed of **fields and methods**
- Multiple **object instances** (or just **objects**) can be created, or **instantiated**, for a class
  - Each object instance of a class has its own copy of the set of fields
  - Objects of a class share methods
- Object oriented programming is very different from procedural programming
  - Class defines shared behavior
  - Object instance provides unique state
  - Data can be hidden within the object

### Objects Are Everything

Unlike procedural languages, Java forces us into the object oriented world. Everything in Java exists inside a class definition, which is used to create objects. A program is made up of a collection of dynamically created objects which send messages to one another.

## What Is a Class?

- A class is a template for creating objects
- It is composed of:
  - Data
  - Methods
- A simple example

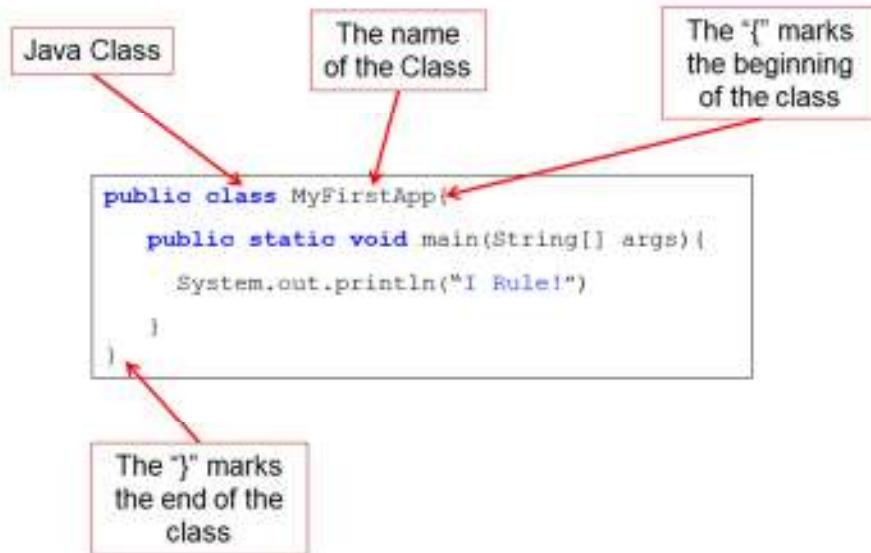
```
class Employee{  
    int salary;  
    int getSalary() { // gets the salary  
        return salary;  
    }  
    void setSalary(int newSalary) { // sets the salary  
        salary = newSalary;  
    }  
}
```

### Definition

A class is a template for building objects. A class encapsulates all of the features of an object, that is, its data elements and its instance methods. When you write a class, you are defining an object's structure and its behavior.

An object's structure is determined by its fields, while its behavior is determined by its methods. When you create a new object from a class, you are creating an instance of that class. Each instance of a class has its own instance variables, and therefore is independent of other objects of that class.

## Basic Java Syntax for a Class



## Defining the Class

- Let's break down the class definition

- A class is defined using the class keyword

```
class Employee{ }
```

- Instance variables are declared in the class block, outside of the methods

```
String name;  
int salary;
```

- Methods are defined within the class block

```
void setSalary(int newSalary) {  
    salary = newSalary;  
}
```

### Simple Example of a Class

A class is defined using a class block. All fields and methods are listed within this block.

Directly inside the class block are the fields of the class. These define the structure of data as a data record. These fields can be declared as any valid Java type.

Also within the class block are the method blocks. Each method is implemented similarly to functions in other languages. Methods may take arguments, and may provide a return value.

A special type of method, called a constructor, can be defined to specify how the object is initialized when it is created.

## Defining the Class (cont'd)

- A class may have one or more constructors
  - initialization methods with the same name as class name

```
class Employee{  
    ...  
    Employee() {  
        name = new String("Kimberly");  
        salary = 2000000;  
    }  
}
```

## Access Modifiers

- Access Modifiers define visibility of class, field or method
  - Determines whether member can be used by other class
- A class can be defined as **public** or **package-private**
  - Using **public** makes class visible to other classes
  - Using package-private (no modifier) limits visibility to current package

```
public class Employee {...}
class Employee {...}
```
- Fields and methods can also be defined as **private** or **protected**
  - Using **protected** makes member visible to subclasses
  - Any **private** member is only visible to owning class

```
private String name;
private int salary;

public void setSalary(int newSalary) {
    salary = newSalary;
}
```
- Fields are most often defined as **private**
  - Encapsulating state of object

Access modifiers can be applied to class definitions, methods and fields to specify their visibility. When a (top-level) class is defined as ‘public’ the class definition can be ‘seen’ by other classes, while a class that has been defined without access modifier (package-private) can only be seen by other classes within the same package.

When defining members of a class (fields or methods) two additional access modifiers of a class can be added. By defining a field or method as private, the member becomes only visible to the class in which it has been defined, while defining it as protected also makes the member visible to subclasses.

## Class Instance Variables

- **Instance variables (or just *fields*):**
  - Hold the object's state
  - Can be used by all of the methods of that object instance
  - May be accessible by external objects
- **Every field is declared to be a specific *type***
  - A primitive type, such as an `int` or a `char`
  - An object type, a reference to another object instance
- **Instance variables are automatically set to their default value when declared if not initialized explicitly**
- **The `null` keyword:**
  - An explicit reference type value
  - A reference of any type can be set to `null`
  - Not the same as an *uninitialized reference*

### Instance Variables

Instance variables hold the state of the object and are visible to all of the methods in the class. Instance variables are set to their default values when the object is created, or to the value you specify the instance variables to be set to.

Instance variables define the structure of an object. Instance variables are typically defined at the start of the class definition. We have declared the fields public which allows anyone to have access to them. As you will see later, this is usually undesirable. When we discover how to write instance methods, we will declare the fields private and create public accessor methods

## A Class Can Define Data Structures

```
public class Employee {  
    private int age = 0; // instance variable  
    private String firstName = "unknown"; // instance variable  
    private String lastName; // instance variable  
    //getters and setters...}
```

```
Employee me = new Employee();  
me.setAge(35);  
me.setFirstName("Fred");  
me.setLastName("Flinstone");
```

```
Employee you = new Employee();
```

**Employee**  
firstName="Fred"  
lastName="Flinstone"  
age=35

**Employee**  
firstName="unkown"  
lastName=null  
age=0

## Primitives in Java

- Java defines eight primitive types
  - Types are predefined and reserved keywords
  - Simple and singular values stored on the stack

Type	Size	Min	Max
byte	8-bit	-128	+127
char	16-bit	Unicode 0	Unicode $2^{16}-1$
boolean	1-bit	true or false	
short	16-bit	-32,768	32,767
int	32-bit	$-2^{31}$	$+2^{31}-1$
long	64-bit	$-2^{63}$	$+2^{63}-1$
float	32-bit	32-bit IEEE 754	
double	64-bit	64-bit IEEE 754	

### Primitive Types

Java provides a few non object oriented concepts. The first of these are the primitive types. These are very similar to the types in a language like C. They hold a value and have no class properties.

## Primitives vs. Object References

- A primitive field type holds a value
  - Fields not set explicitly will be given a default value

```
private int age = 13;
private int salary; // = 0;
```
- An object reference field type can only hold a reference to an object

```
private Person student; // Person is an object type
```
- Comparing to C/C++
  - C/C++ has both pointers and structs
  - Java has only strongly typed pointers - object references
  - Java objects are instantiated similarly to dynamic memory allocation in C/C++

```
student = new Person("Kimberly"); // Continued from above
private Person teacher = new Person("Robert"); // all at once
```

Java object references hold the address of the object to which they refer. The type of the reference is the type "reference to object" where the object is the class it refers to.

It is said that Java has no pointers. This is not entirely accurate as these references are pointers. The difference with Java is that its pointers are strongly typed, and these references cannot be manipulated in such ways as incrementing the reference values, etc.

## Creating Objects

- Objects are normally created (or instantiated) by using the **new** operator
  - The class name on the right matches the field's reference type on the left
  - The new object instance is initialized with a value of "Kimberly"

```
private Person student;  
student = new Person("Kimberly");
```

- This Person instance will remain in memory until student no longer references it

```
student = new Person("Jim");
```

- Or even 

```
student = null; // no instance is referenced
```

### Creating Objects

The new operator is used to create a new object. It causes the invocation of a class constructor, and returns a reference to that object.

A class constructor is a method with the same name as the class. If the constructor requires parameters then they are passed as method arguments.

When we create a new object, the new operator creates the object using a class constructor, then it returns a reference to that object. The returned reference is then assigned to the object reference through the use of the assignment operator (=).

When an object is no longer referenced, it is said to be eligible for Garbage Collection.

## The main Method

- Any class may have a method with the following signature

```
public static void main(String[] args) { ... }
```

- A class that has this method may be invoked from the command prompt, or shell, using the *Java interpreter*
  - The interpreter will invoke the `main` method

```
PROMPT> java MyClass
```

- Class would be referred to as *Java application or a driver*
  - Most classes are not drivers and do not have a `main` method

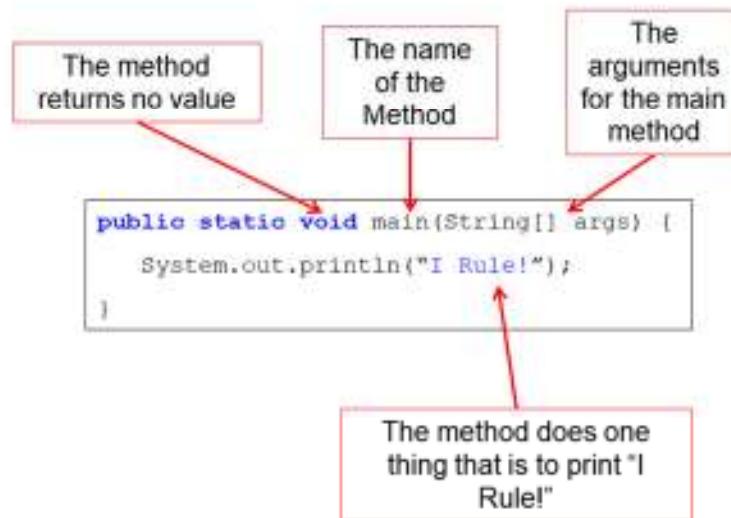
### The main Method

The main method is a special method in a class. It allows the class to become a starting point for running an application.

The Java interpreter, which is the application that runs our Java code, is executed as an operating system command. We must pass to this command the name of a class to start executing. The class provided must have a main method; it is the well known method that the Java interpreter will use as an entry point to begin execution.

We'll be looking at this method in more detail later in the course.

## Basic Java Syntax for a main Method



## Using the Dot Operator

- To access an object's method or variable, you can use the dot operator

- Syntax: `instance.variableName` or `instance.methodName()`

- The data type is the type of the variable or of the return type

- Invoking a method that is in an object

```
private String personsName = student.getName();
```

- Accessing a field that is in an object

```
student.age = 21;
```

- Note:** Convention and best practice would use setters and getters to access a field in another object

- Dot operations may be chained

```
if (student.getLastName().equals("Kindred")) {...}
```

- Notes:**

- Chaining dot operations can make debugging challenging since several access operations may take place in a single statement

- The visibility of methods or fields referenced must support access from the calling code (covered later)

### Dot Notation

The dot operator is used on object references to access fields and methods of that object. Most commonly it is used to invoke the methods of an object instance.

As shown here, fields of the object can also be accessed using the dot operator.

The dot is generally referred to as an operator. However, in looking more closely at the Java language specification, it would probably be more accurately defined as a separator.

## Writing Output

- A Java program can write messages to the console
  - `System.out` – points to standard output
  - `System.err` – points to standard error output

```
-
System.out.println("Got to this line in my code!");
-
System.err.println("An error occurred!");
-
```

### `System.out.println()`

This illustrates the use of a class name (`System`), an object reference name (`out`), and a method call (`println()`). This example takes a `String` argument and writes it to the standard output stream.

Don't worry about how all this works for right now. We'll be using the code illustrated above in many of our upcoming lab exercises. Later in the workshop we'll cover output streams in detail.

## The Employee class

```
public class Employee {  
    private int age = 0; // instance variable  
    private String name = "unknown"; // instance variable  
  
    //getter and setter method definitions  
    public void setName(String newName) {  
        this.name = newName;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setAge(int newAge) {  
        this.age = newAge;  
    }  
  
    public int getAge() {  
        return this.age;  
    }  
}  
  
public class MySecondApp {  
  
    public static void main(String args[]) {  
        Employee me = new Employee();  
        me.setAge(35);  
        me.setName("Troy");  
  
        Employee you = new Employee();  
        you.setAge(60);  
        you.setName("Dan");  
  
        System.out.println(me.getName());  
        System.out.println(you.getName());  
    }  
}
```

## Java Keywords

- The Java programming language defines a number of keywords
  - Can not be used for variables, methods, class names, etc.

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

The Java programming language defines a number of keywords. These keywords are used to define the language constructs and are therefore not allowed to be used as variable names, method names, class names, etc.

The `goto` and `const` keywords are defined as keyword but is not used in the Java programming language.

The `strictfp` was added in version 1.2 of the programming language and is used to define the way the JVM handles floating point operations

The `assert` keyword was added in version 1.4 of the programming language

The `enum` keyword was added in version 1.5 (now called version 5) of the programming language

## Lesson Review and Summary

- 1) Do fields contain values in classes or in objects?**
- 2) Must instance variables declare a type?**
- 3) Are instance variables automatically initialized?**
- 4) What are the two categories of field types?**
- 5) What is the purpose of the `main` method?**
- 6) What is the name of the operator that creates object instances?**
- 7) How can you display text to the screen?**

- 1) Objects
- 2) Yes
- 3) Yes... instance fields are automatically initialized when the object instance is created
- 4) Primitive types and object references
- 5) A main method can be defined to allow the class to be invoked from the command line
- 6) Objects are created from classes using the `new` operator
- 7) Text can be written to standard output using `System.out.println("my text");`

## Exercise: Create a Simple Class

`~/StudentWork/code/SimpleClass/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

## Lesson: Adding Methods to the Class

Writing a Simple Class

**Adding Methods to the Class**

Language Statements

Using Strings

Specializing in a Subclass

## Lesson Objectives

- After completing this lesson, you will be able to:
  - Write a class with accessor methods to read and write private instance variables
  - Write a constructor to initialize an instance with data
  - Write a constructor that calls other constructors of the class to benefit from code reuse
  - Use the `this` keyword to distinguish local variables from instance variables

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Instance Methods

- A method is similar to a function or subroutine
  - Methods can provide access to an object's behavior or data

```
access-modifier returnType methodName ( arguments ) {  
    local variable declarations  
    method code  
    more local variable declarations  
    more method code  
    return returnType;  
}
```

- The key components of a method are:
  - A modifier such as public or private (covered later)
  - A return type may be any class, interface, primitive type, or void
  - The method's name
  - Input parameters
  - The method's body

### Instance Methods

Instance methods define an objects behavior. A method in Java is equivalent to a function, procedure, or subroutine in other languages. A parameter is declared by stating the object's type, followed by the object's name (e.g. String myName)." Methods must be declared within a class definition. In other words, there are no "global" methods.

#### Five Key Components of a Method

modifier e.g. public, private, protected, default (you will see more of these later).

returnType This is either a primitive data type, an object, or void. A method may return at most one thing.

methodName The name of the method should describe what the method does, e.g. calculateRangeDifference().

arguments A method may have any amount of arguments, including zero.

method body The method body contains the code which is executed when the method is called

## Passing Parameters Into Methods

- A method can have zero or more parameters
  - The method name and parameter types make up its signature

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

- The this keyword
  - Clarifies specifies the use of the instance variable
- When the method is called:
  - Each parameter is a local variable of the method
- The void keyword is used to indicate the return type for a method that returns nothing

```
public String getName() {  
    return this.name;  
}
```

### Argument(s)

A method may have zero or more arguments (or parameters). When you create a method, you declare the parameters which you wish to pass into the method. The parameter declared by stating the objects type, followed by the objects name. e.g. String myName. Parameters are separated with a comma within the parameter list.

When the method body is executed, each formal argument becomes a local variable within the scope of the method with the value(s) of the object passed in. For example, if we wanted to change the name of an object, we could call its setName() method. This method takes one parameter, a String. Within the body of the method, this string is called newName. If we were to call this method and pass in a string with the value "Bob", the variable newName would contain the same value. We would then be able to assign the value of the newName object to the original object's name field.

## Returning a Value From a Method

- If a return type other than void is specified, the method must return a value of that type
- If a return type of void is specified, the method will return at the final brace without an explicit return statement
- You can return from a method at any point by explicitly calling return

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

public String getName() {
    if (name != null)
        return name;

    System.out.println("Unknown name");
    return name;
}
```

### Return Value

A method within Java must have zero or one return values. The return value may be a primitive data type (e.g. boolean, int, ...), an object (e.g. String, Person, ...) or if there is no return value, void.

The return type is declared after the access modifier and prior to the method name.

The return statement, usually located as the last statement in a method, is the mechanism used to return objects from a method. You may place expressions after a return statement, but they must evaluate to the object that is to be returned. e.g. return (3 + 4 + x), where the return type is int and the variable x is an int.

## Overloaded Methods

- Two or more methods within a class with same name
  - Parameter types must be different

void setSalary(int newSalary) { ... }  
void setSalary(String newSalary) { ... }
- Method signature must be unique
  - Defined by method name and number/type of parameter(s)
  - Return type and exceptions are not part of signature

String getName() { ... }  
char[] getName(int maxLength) { ... }
- Can make code easier to understand and maintain
- Overloading is not the same as overriding
  - Overloaded methods may be overridden in subclass
  - Overridden methods may be overloaded in subclass

### Overloading

Rather than invent new names for methods, method overloading can be used when the same operation has different implementations. In the example below, two implementations of the setSalary method are as follows:

```
void setSalary (int newSalary) {}  
void setSalary(String newSalary) {}
```

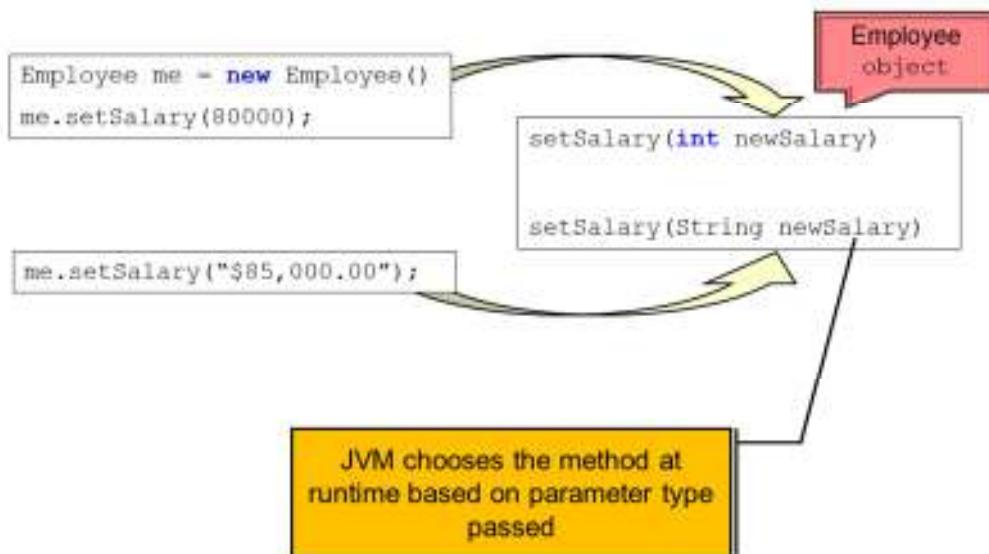
### Overriding vs. Overloading

Method overriding must not be confused with method overloading.

Overriding: requires the same signature (name and parameters) and the same return type and that the original method is inherited from its superclass.

Overloading: requires different parameters but the same method name. The parameters must differ in type, order, or number.

## Overloaded Methods Diagram



## Constructors

- A constructor is a special method that is called automatically when an object is created
  - Has the same name as the class
  - Has no return type
- Steps in the construction of an object:
  - Create an instance of the object
  - Initialize all the variables to their default values
  - Invoke any explicit initialization
  - Execute the code in the constructor

### Constructors

A constructor is a special method defined within the class.

A developer may provide one or more constructors to the class to control the initialization of the object instances of that class.

The constructor is a method with the same name as the class. Unlike regular methods it does not have a defined return type.

### Field Initialization

Fields of a class have default values that are used when an instance of the class is created. The constructor can change this by explicitly initializing the instance variables. In most cases this is the primary function of the constructor.

## Defining a Constructor

- **Each class may have zero or more constructors**
  - If a class has no constructor, Java provides a default constructor which does nothing

```
public class Employee{  
    private String name;  
  
    public Employee() { // no argument constructor  
        name = new String("Kimberly");  
    }  
  
    public Employee(String aName) {  
        name = aName;  
    }  
}
```

### The Default Constructor

It is not necessary to define a constructor for a class. If no constructor is defined, the compiler will generate a constructor that takes no arguments.

Knowing this fact is not really all that critical for a developer learning Java. However, there is a case where this implicit behavior can cause a problem.

If you create a class with no constructor and later use it by instantiating it using the default constructor, it will work because the compiler implicitly gave you the no-argument constructor. If you later add a constructor that takes arguments, the compiler no longer implicitly creates the no-argument constructor for you, and the code that instantiates the class using the no-argument constructor will not compile.

## Optimizing Constructors

- One constructor can invoke another to complete its task
  - Invocation must be the first line in the constructor

```
public class Employee{  
    private String name;  
    private int age;  
  
    public Employee(String aName) {  
        this(aName, 0);  
    }  
  
    public Employee(String aName, int anAge) {  
        this.name = aName;  
        this.age = anAge;  
    }  
}
```

### Code Reuse is Key

In the object oriented programming, code reusability is key. You want to use existing code as much as possible. The same holds true for constructors. If there is a constructor that can be used by another constructor, then use it. This makes maintenance simpler.

## Lesson Review and Summary

- 1) Give an example of a method access modifier?
- 2) What are the valid return types for a method?
- 3) Can a method be written without input arguments?
- 4) Can a constructor be written without arguments?
- 5) What is the purpose of the `this` keyword?
- 6) Can one constructor invoke another?
- 7) Can two methods have the same name?

### Lesson Review Answers:

- Two access modifiers are `public` and `private`
- A method can return an object reference, a primitive, or `void`
- Methods and constructors can be written with or without parameters
- Multiple methods and constructors can be written with the same name, provided their argument types are different (overloading)
- A constructor can invoke another constructor using `this(...)`, provided it is the first line in the calling constructor
- The `this` keyword specifies a name defined in the class, and is used to distinguish fields from method variables
- Yes, the methods are said to be overloaded if they share the same name but different parameters

## **Exercise 2: Create a Class with Methods**

`~/StudentWork/code/Methods/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Language Statements

Writing a Simple Class  
Adding Methods to the Class  
**Language Statements**  
Using Strings  
Specializing in a Subclass

## Lesson Objectives

- After completing this lesson, you will be able to:
  - List the four arithmetic operators
  - List the three operators to increment and decrement numbers
  - List the six comparison operators
  - List the two logical operators
  - Name the return type of the comparison and logical operators
  - Use one of the increment operators to increment an integer
  - Write a `for` loop that can iterate a specified number of times

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

# Operators

- **Assignment**

```
int x = 5;
```

- **Arithmetic operators**

+	add
-	subtract
*	multiply
/	divide

- **Short hand increment/decrement operators**

x += y;	short for x = x + y
x -= y;	short for x = x - y
x++, ++x;	short for x = x + 1
x--, --x;	short for x = x - 1

- **There are more advanced operators, such as bit operators**

## Operators

Java has the complete C operator set, with one minor addition to the bit shift operators. Operator precedence is the same as C, and may be altered with the use of parenthesis. If you are unsure of evaluation order, use parenthesis to make the meaning of your code clear.

If you know C, Java will be very familiar. All statements can be multi-line and are terminated with a semicolon. Unique to Java is convenient syntax for dealing with strings. If you don't know C, follow the code examples carefully.

There is an interesting book called "Learn Java by Association" by Dillip Dedhia, ISBN 0-9679469-0-5. The book has side by side programs in Cobol, Fortran, Pascal, Basic, C, and Java, which we highly recommend to non C programmers.

## Short hand increment/decrement operators

- Notice difference between prefix and postfix operator

```
public void start() {
    int x = 10;
    printInteger(++x);
    System.out.println(x); //Prints 11
}

public void printInteger(int value){
    System.out.println(value); //Prints 11
}
```

```
public void start() {
    int x = 10;
    printInteger(x++);
    System.out.println(x); //Prints 11
}

public void printInteger(int value){
    System.out.println(value); //Prints 10
}
```

## Comparison Operators

- Comparison operators

- All evaluate to a Java Boolean value of either true or false

= =, !=, >=, <=, >, <

```
public void process(int value){  
    boolean greater = value > 15;  
    ...  
}
```

### Java's Comparison Operator

Java's comparison operators always result in a boolean value. This is different from C or C++, and eliminates the misuse of = when == was intended.

## The if statement

- The if-then **statement**

- Unlike C, expression must be boolean, not int

```
public void process(int value){  
    if(value > 21){  
        System.out.println("Greater");  
    }  
}
```

- When conditional operation is a single statement curly brackets are optional

```
if(value > 21) System.out.println("Greater");
```

If statements may be nested, and if-else trees are supported. In general, it is good practice to overuse braces to make the meaning of your code clear to yourself and others.

# Logical Operators

- **Logical operators**

&& logical and

```
// x must be 5 and y must be 10
if (x == 5 && y == 10)
```

|| logical or

```
// either x must be 5 or y must be 10
// both would be okay,
//only one is necessary
if (x == 5 || y == 10) // then true
```

- Single character logical operators ( & and | ) result in both sides of the expression to be evaluated!

```
int i = 5;
int j = 5;
if(i++ > 3 || j++ > 5){
    System.out.println("...");}
System.out.println(i); // 6
System.out.println(j); // 5
```

```
int i = 5;
int j = 5;
if(i++ > 3 | j++ > 5){
    System.out.println("...");}
System.out.println(i); // 6
System.out.println(j); // 6
```

&&	logical AND stops processing on the first step that evaluates false
	logical OR stops processing on the first step that evaluates true
&	logical AND always processes all steps
	logical OR always processes all steps

## The if / else statements

- The if-then-else statement

```
if (booleanExpression) {  
    ...  
} else {  
    ...  
}
```

- The else-if statement

```
public void process(int value){  
    if(value > 21){  
        ...  
    } else if(value > 11){  
        ...  
    } else{  
        ...  
    }  
}
```

## The ternary operator

- The ternary (or conditional) operator can be used for value assignment
  - Syntax:  
`variable = testCondition ? valueIfTrue : valueIfFalse;`
  - Examples:

```
minVal = (a < b) ? a : b;  
absValue = (a < 0) ? -a : a;
```

## Looping: The for Statement

- **The for statement**

- Is used to perform a repetitive code block while a condition is true
- Allows for variable initialization and increment/decrement

```
for(initial statement; condition; iteration expression) {  
    // do something until condition is false  
}
```

- **Example:**

```
for(int a = 0; a < 10; a++) {  
    System.out.println("a=" + a);  
}
```

- Notice that the variable defined in the for statement is part of the scope of the loop block

### The for Statement

The for statement is used to control looping. The general form is (starting values; ending condition test; increment value). You may initialize multiple starting values and have multiple increment expressions.

## Looping: The for Statement (cont'd)

- Needs name of the variable used to hold the individual element reference and the collection variable

```
public void testEnhancedFor() {  
    String myStringArray[] = new String[5];  
  
    myStringArray[0] = new String("String number one");  
    myStringArray[1] = new String("String number two");  
    myStringArray[2] = new String("String number three");  
    myStringArray[3] = new String("String number four");  
    myStringArray[4] = new String("String number five");  
  
    for (String element : myStringArray) {  
        System.out.println(element);  
    }  
}
```

Java 5 introduced the 'enhanced' for-loop.

When a collection of data is to be iterated the enhanced for loop can be used. The syntax is  
for ( <Type of object in array> <name of individual element> : <name of array reference> )

## Looping: The while Statement

- The while control loop will execute a block of code 0 or more times

```
while(booleanExpression) {...};
```

- Note that the test occurs before each pass through loop

```
while (buffer.containsMoreData()) {  
    // do something  
}
```

### The while Statement

A while statement may execute zero or more times, as the test is performed at the top of the loop.

## Looping: The do Statement

- The do **statement will execute a loop 1 or more times**
  - do { // loop code} while (booleanExpression);
  - Note loop gets executed before first test for completion

```
int x = 50;
do {
    x = x/2;
} while (x > 10);
```

- What is x at completion?

### The do Statement

The do statement will execute at least once. The test for completion is done at the end of the loop.

## Continue and Break Statements

- The **continue** statement begins the next iteration of the enclosing loop
- The **break** statement exits the enclosing loop
- Remember there is no **goto** statement, although it is a reserved word

```
x = 0;
while (true) {
    x++;

    if (x > 100) {
        break;
    }

    if (x == 20) {
        continue;
    }
    x *= 2; // skipped if x = 20
}
// we exit here when x > 100
```

### Flow Control

Continue continues the looping, whereas break breaks out of the loop. This behavior is very similar to the constructs in other programming languages.

## The switch Statement

- Replaces complex if/else constructs
- You can switch on
  - byte
  - short
  - char
  - int
  - Enum types
  - Strings

```
public void colorCode(int code) {  
    switch(code) {  
        case 1:  
            System.out.println("colorCode = " + code);  
            break;  
        case 2:  
            System.out.println("colorCode = " + code);  
            break;  
        default:  
            System.out.println("colorCode = " + code);  
    }  
    System.out.println("break sent us here");  
}
```

### The switch Statement

The switch statement eliminates complex if/else code structures by testing an integer value multiple times. If you do not have a break statement at the end of a case, execution will fall into the next case. So by ordering your cases appropriately, you can selectively execute multiple case blocks. This can be very confusing to someone reading your code, so do this very selectively, and again a lot of comments are appropriate.

## The switch Statement (cont'd)

- Multiple values can be ‘checked’
  - When break is omitted

```
public void color(String color) {  
    switch (color) {  
        case "black":  
        case "brown":  
            System.out.println("Dark");  
            break;  
        case "white":  
            System.out.println("Light");  
            break;  
    }  
}
```

- The default statement is optional

When the `break` statement is not added to a `case`, processing will continue down the `case` statements until a `break` statement is encountered.

In the example above, “Dark” is printed for both the value “black” and “brown”

The use of the `default` keyword is optional. When none of the cases meet the criteria, nothing is printed.

## Lesson Review and Summary

- 1) What are four arithmetic operators?**
- 2) What are the six comparison operators?**
- 3) What are two commonly used logical operators?**
- 4) Name four complex language statements used for testing and/or looping?**
- 5) What is the return type of comparison operators?**
- 6) Can the test for a loop condition be an integer?**

Comparison and logical operators return `boolean`

`if`, `while`, and `do-while` statements use `boolean` conditions

The `continue` statement will cause the code to jump to the top of the loop

The `break` statement will cause the code to leave the loop

A `switch` statement can be used to branch to a section of code based on the value of an integer type

## Exercise 3: Looping

`~/StudentWork/code/Looping/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

## Exercise 4: Language Statements

`~/StudentWork/code/LanguageStatements/lab-code`

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

# Lesson: Using Strings

Writing a Simple Class

Adding Methods to the Class

Language Statements

**Using Strings**

Specializing in a Subclass

## Lesson Objectives

- **After completing this lesson, you will be able to:**
  - Create an instance of the `String` class
  - Test if two strings are equal
  - Test if two strings are the same object
  - Get the length of a string
  - Parse a string for its token components
  - Perform a case-insensitive equality test
  - Build up a string using `StringBuffer`
  - Contrast `String`, `StringBuffer`, and `StringBuilder`

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

# Strings

- **String is a class in Java**
- **They can be instantiated like any other class type**

```
String myString = new String("this is my string");
```
- **Common literals are often created in the constant pool for efficient reuse**

```
String myString = "this is my string";
```
- **The compiler will generate appropriate code to support string concatenation statements**

```
String newString = "the first half" + " the second half";
```
- **String class instances are immutable**
  - Value cannot be modified once object is instantiated
  - Any concatenation operation creates a new String object

## About Strings

Directly assigning a string is supported by the compiler and is slightly more efficient than using a constructor. It is equivalent to constructing a new String instance. In addition, the compiler will look for other requests to create the identical String, and will generate the bytecode such that all references actually refer to the same instance of String. This is allowable since an instance of the String class is read-only.

Strings are full fledged objects with extra support for creation and concatenation built into the language. Strings are immutable, so when you concatenate two strings, Java will give you a new object with the value of the two strings as one.

You will see that most classes in Java support strings in special ways. All classes provide a mechanism to convert a class instance to a string, and many classes allow instances of the class to be created from strings, where appropriate.

Internally, strings are stored in Unicode, not as bytes. This means that the String class can inherently support any language that has a Unicode representation, including such languages as Japanese and simplified Chinese.

## String Methods

- The String class provides many useful methods
  - equals - a character by character comparison
  - length - returns the length of the string
  - equalsIgnoreCase - ignores case during the comparison
  - charAt - get the character at an index
  - indexOf - see if the sub string exists within the string
  - lastIndexOf - the last occurrence of the sub string
  - startsWith - does the start match the sub string
  - endsWith - does the string end with the sub string
  - toUpperCase - return the string's value in upper case
  - toLowerCase - return the string's value in lower case
  - trim - returns copy of string with leading and trailing whitespace removed

```
String str1 = new String("Hello");
String str2 = new String("Hi");
if (str1.equals(str2)) return true;
```

### String Objects

Since strings are objects, they have a number of very useful methods. Be sure to use the equals method, not the == operator, when comparing two strings.

Probably the most commonly used methods are equals and length. You will notice that there are no methods to write to the contents stored in the String object instance.

## String Equality

- String is a good example of the difference between == and the equals() method
  - The == operator tests to see if two references refer to the same object
  - The equals() method compares the content of two objects for equality
  - ◆ Each class usually has its own implementation

```
String string1 = new String("Toms River, NJ");
String string2 = string1;
if (string1 == string2) System.out.println("1==2"); // true
if (string1.equals(string2)) System.out.println("1eq2"); // true

String string3 = new String("Toms River, NJ");
if (string1 == string3) System.out.println("1==3"); // false
if (string1.equals(string3)) System.out.println("1eq3"); // true
```

### Equals vs. ==

It is important to understand the results of the == operator versus the equals method of objects. The == operator tests to see if two object references are pointing to the identical object. The equals method compares the contents of two objects and returns true or false depending on the contents. In almost all cases, you probably want to use the equals method, not the == operator.

## String substring (Java 6)

- Original String remains in memory

```
String veryLargeString = "abcdefghijklmnopqrstuvwxyz";
String substring = veryLargeString.substring(4, 5); //e

veryLargeString = null;
```

- Now it does not

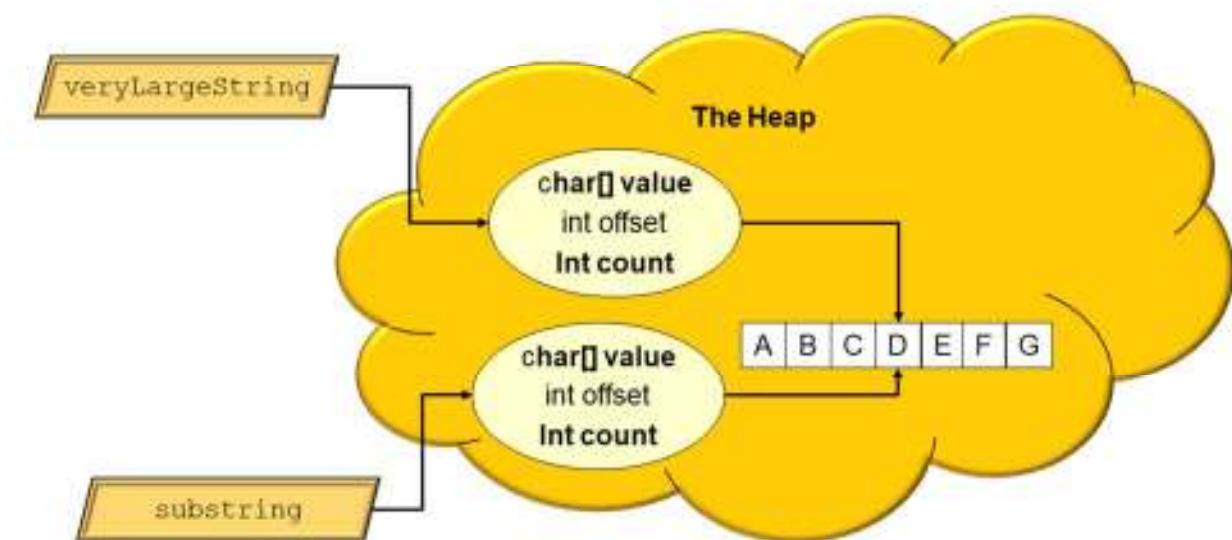
```
String veryLargeString = "abcdefghijklmnopqrstuvwxyz";
String substring = new String(veryLargeString.substring(4, 5));

veryLargeString = null;
```

- In Java 7 this issue is resolved

Until Java 7, using the substring method did not result in a new string to be created. It would 'simply' create a reference to part of the large string. In Java 7 this issue was resolved

## Java substring (Java 6)



## StringBuffer

- The **StringBuffer** class is a **mutable String**
- It is common to use a **StringBuffer** to build up a string value and convert it to a **String** when done

```
StringBuffer buffer = new StringBuffer();
buffer.append("some text");
...
buffer.append("some more text");
...
buffer.append(" and more again");
...
String stringValue = buffer.toString();
```

- Can be far more efficient, since unlike **String**, we don't create multiple read-only **String** instances to create larger **String**
- **StringBuffer** has a number of useful text manipulation methods

### StringBuffers

Because strings are immutable, you can create a lot of smaller String objects when concatenating strings. The StringBuffer class provides a high efficiency solution to this problem.

In addition to most of the methods of the String class, the StringBuffer class contains methods to modify the contents without creating a new object. Strings may be easily converted to StringBuffers and visa versa.

## String, StringBuffer, and StringBuilder

- **String operations are relatively costly, especially when performing concatenations**
  - Each concatenation creates a new String object to represent the newly concatenated string.
  - Series of concatenations cause large number of objects to be created
- **StringBuffer uses a single object to support the growing string**
  - Much greater efficiencies are achieved if you are modifying character strings at all.
- **StringBuilder increases these efficiencies even more**
  - Same operations as StringBuffer but they are unsynchronized
  - Greatly reduced overhead since there is no thread synchronization
- **StringBuilder should be viewed as the default**
  - Unless thread synchronization is needed

## Lesson Review and Summary

- 1) What method is used to modify a String?
- 2) What are the advantages of StringBuffer and StringBuilder over a String?

The `String` class contains a large number of useful string manipulation methods

A `StringBuffer` is more limited in terms of functionality, but supports updating the string content without creating new instances. A `StringBuilder` is not synchronized and is, therefore, faster and more efficient than `StringBuffer`.

## Exercise 5: Fun with Strings

`~/StudentWork/code/Strings/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

## **Exercise 6: Using StringBuffers and StringBuilders**

**`~/StudentWork/code/StringBuilders/lab-code`**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## **Lesson: Specializing in a Subclass**

Writing a Simple Class  
Adding Methods to the Class  
Language Statements  
Using Strings  
**Specializing in a Subclass**

## Objectives: Specializing Subclasses

This lesson covers how to write subclasses in Java.

Specifically, it covers:

- Constructing a class that extends another class
- Correctly implementing equals and toString
- Writing constructors that pass initialization data to the parent constructor as appropriate
- Using instanceof to verify the class type of an object reference
- Overriding subclass methods and use the super keyword to leverage behaviors in the parent
- Safely casting references to a more refined type

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Extending a Class

- A class may *extend* the definition of another class
  - Methods and instance data of the *superclass* are available to the *subclass* (based on superclass method/field visibility)
  - We can add new instance data
  - We can add new methods
- We can *override* methods of the superclass
  - Same name, return type, and calling parameters
  - Different behavior
  - Best practice is to always use the @Override annotation
    - ◆ Not required but helps compiler (and programmers) detect errors

### Code Reuse Through Inheritance

A key concept of object oriented programming is the concept of code reuse through inheritance. We can extend the functionality of an object, while still using all of the code developed in the parent class. Alternatively we can alter the behavior of the parent class by overriding some of the methods defined in the parent class.

## Extending a Class (cont'd)

- The subclass instance can be used anywhere the superclass type is required
- Java does not support *multiple inheritance of classes*
  - A subclass cannot extend more than one superclass
  - Single inheritance actually provides a safer OO model

### About Inheritance

Inheritance in Java is the ability of a subclass to use the structure and behavior of its parent class or superclass. Inheritance is a relationship. That is to say, a chair is a type of furniture, or a cat is a type of mammal.

Java has removed a problem by making multiple inheritance illegal. Multiple inheritance when used well is very powerful but most of the time simply leads to overly complex hard to maintain software. Java has implemented the notion of interfaces which solve the design problems caused by the no multiple inheritance. These will be covered in a later lesson.

As with other OO languages, subclasses know everything about their superclasses, but superclasses know nothing about their subclasses.

## The `extends` Keyword

- Subclassing is expressed using the `extends` keyword

```
class Person {  
    int age; // Primitive data type  
    String name; // Object reference  
}  
  
class Employee extends Person {  
    int empID;  
}
```

- The following code is valid

```
Employee me = new Employee();  
me.age = 21;  
me.name = new String("Kimberly");  
me.empID = 12345;  
Person p = me;
```

- These are not valid

```
Employee you = new Person();  
p.empID = 4242;
```

### Extend to Add

When you define a subclass, you need only provide the code to implement changes in behavior from the superclass. By default, your subclass inherits the capabilities of its superclass.

To change the behavior of an inherited method, the subclass redefines the method in its class definition. This causes the corresponding superclass method definition to be hidden.

We can also extend a class with additional functionality by adding new methods and fields. Another key concept is that an extended class can be treated as its base class so long as you do not use any of the new functionality defined in the extended class.

## Upcasting

- Allows us to tell the compiler to use a different type within the object's class hierarchy
- Upcasting to a more generalized type
  - Also referred to as **casting**
  - Safe since able to be checked by compiler

```
// Employee and Employer are both subclasses of Person
Employee employee = new Employee("Kimberly"); // the subclass
Employer employer = new Employer("Brenda"); // the subclass
Person person; // the superclass
person = (Person)employee; // Explicit cast not required here
```

### Casting

Since Java is a strongly typed language, it always checks for type compatibility at compile time. Some checks, however, are possible at runtime only and cases where an operator would have incompatible operands can often occur. Java demands that a cast be used in such cases.

## Downcasting

- Casting to a more specialized type
  - Also referred to as *casting*
  - Compiler can check that the cast is within the class' hierarchy
  - Compiler cannot verify that referenced object is of that type
  - Could result in a runtime error called a `ClassCastException`

```
void someMethod(Person person) {  
    // The following code may or may not cause a ClassCastException  
    Employer emp = (Employer) person;  
    ...  
}
```

- Primitives can also be cast
  - ◆ Not required if converting to wider type
  - ◆ Required if narrowing

The cast from a Person to an Employer is technically valid, since Employer is a subtype of Person. The compiler will check for this.

However, the person reference could actually refer to an Employee instead of an Employer, and so at runtime this cast would fail.

## Overriding Superclass Methods

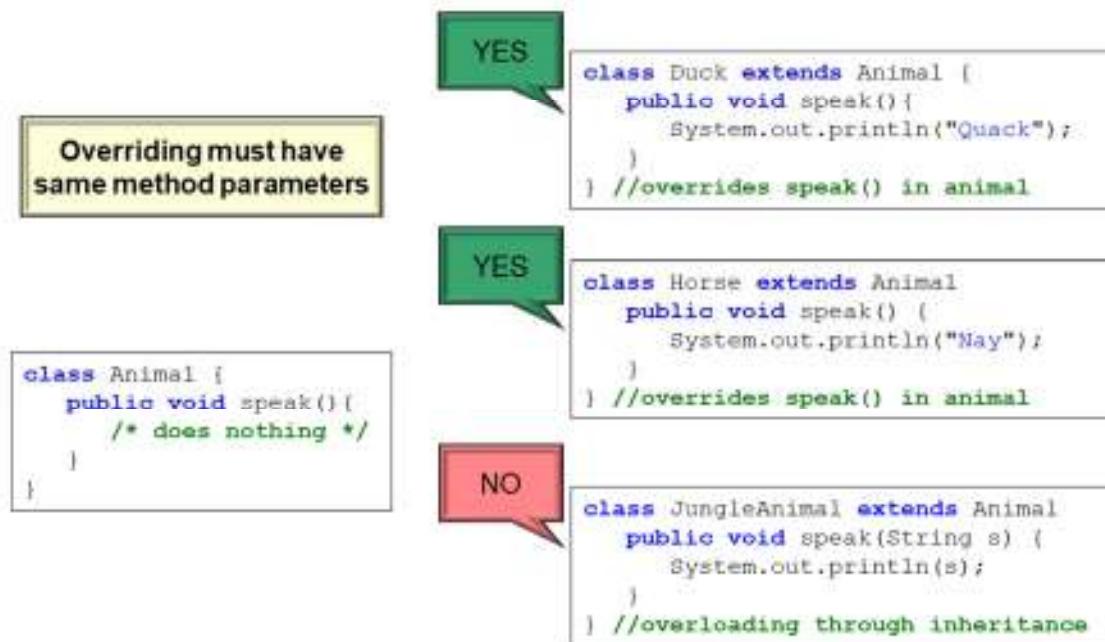
- A subclass may override methods inherited from superclass
  - The signatures of the methods must be the same
  - This hides the method with the same signature in the superclass
  - Note: assumes the method is visible and not final
- Do not confuse method overloading with method overriding
  - Overloading is where you define multiple methods with the same name but with different input parameters (arguments)
  - Overriding is where you define a method with exactly the same signature as a method in a superclass
  - Overriding is used to provide specialized behavior

### Overriding

A subclass inherits all of its superclass' methods. A subclass can, though, override and modify the behavior of a method in its superclass. This is known as overriding.

To override a method, the subclass must define a method with the exact signature and return type as the one in the superclass. Note that the method overridden does not need to be defined in the immediate superclass, i.e. a subclass can override methods in any grandparent class.

# Method Overriding



## Calling Superclass Methods From Subclass

- An overriding method will often build on the implementation of the superclass' overridden method
  - The **super** keyword is used to access methods in the parent

```
class Parent {  
    String name;  
  
    public String toString() {  
        return name;  
    }  
}  
  
class Child extends Parent {  
    String phone;  
  
    public String toString() { // overriding the parent  
        return new String(super.toString() + ":" + phone);  
    }  
}
```

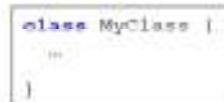
### Leveraging Superclass Implementations

The ability to override a method in a subclass would only be moderately useful if doing so required us to duplicate the work already done in the superclass.

Java allows overriding methods to invoke the overridden method in the superclass using the **super** keyword.

## The Object Class

- Base class from which all others eventually extend
- This class is extending from Object



- Well designed classes should always override:
  - equals
    - ◆ Allows class to determine its own equality test
  - toString
    - ◆ Allows class to intelligently convert itself to a String
  - hashCode
    - ◆ Provides a unique value for unique objects

### java.lang.Object

Every class that is defined in Java is a subclass of Object, even if it is not stated explicitly in the class definition.

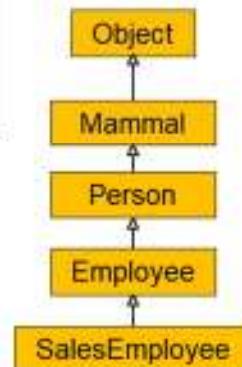
Well written Java classes will override **equals** and **toString** originally provided in the Object class.

While the behaviors of these two methods are somewhat functional as defined in Object, in most cases they are not complete enough to provide fully appropriate behavior in the subclass.

## The instanceof Keyword

- **instanceof is a Java keyword to test if a reference points to an object of a certain type**
  - Returns false when the reference is null

```
Person person = new Person();
Person otherPerson = null;
if (person instanceof Person) { ... } //true
if (person instanceof Mammal) { ... } //true
if (person instanceof Object) { ... } //true
if (person instanceof Employee) { ... } //false
if (otherPerson instanceof Person) { ... } //false
```



The **instanceof** keyword is used to check if an object instance is of a particular type.

In the class hierarchy a Person object is a valid instance of Person, Mammal and Object. In other words, it provides an implementation of all these class definitions.

The person object is not an instance of SalesEmployee.

The reference that is used for the evaluation does not explicitly have to be checked for null, since a null point is (per definition) not an instance of a type.

# Use instanceof Before Downcasting

- Checks if object is an instance of type

```
Person person = new Employee();

if (person instanceof Person) { ... } //true
if (person instanceof Employee) { ... } //true
```

- Often used before downcast
  - Avoid ClassCastException

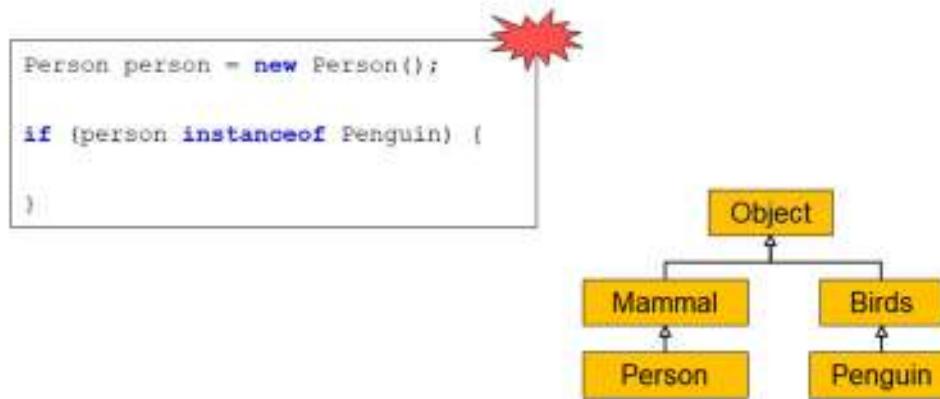
```
private void speak(Person person) {
    if (person instanceof Employee) {
        Employee employee = (Employee) person;
    }
}
```

The **instanceof** check is performed on the object instance to which the reference is pointing, not the reference type!

By using the **instanceof** keyword, you can add a check to your code before you downcast the reference. When the reference is pointing to an **instanceof** Employee it is valid to use an Employee reference to point to this object.

## The instanceof Compiler Check

- Partially checked by the compiler
  - Person reference can never point to Penguin object



When **instanceof** is used against a type that is not within the same class hierarchy, the compiler will throw an error.

In the example, a Person reference can never be used to point to an instance of Penguin, as a result the compiler will immediately mark this statement as an error.

# Object Equality

- If you have two Person instances, how do you determine if these instances represent the same person?
  - By comparing their unique id (primary key)?
  - By comparing each individual property of the instances?
  - By comparing only their social security number?
  
- Answer is different per application...
  - Business rules define when two sets of data represent same person

When you have two instances of Person (each populated with data), how do you determine if these objects represent the same person?

The answer to this question depends on the application you are developing. In one application they are only considered equal when the ID is unique (for example, when the data was read from a database, by comparing their primary key). In this situation two instances with different IDs but the same social security number are considered to resemble two different persons.

Other applications consider two instances equal when the social security number is the same, even when they were read from two different rows in a database.

Again, other applications consider two persons equal when they have the same name (e.g. username).

In other words, to determine if two instances are equal, the developer must provide the logic to determine if they are equal.

## The Object.equals Method

- Every object has an **equals** method
  - Inherited from **Object** class
- Implementation can be overridden
  - **Object.equals()** method checks if the two references are referring to the same object instance

```
public class Person {  
    String name;  
    int socialSecurityNumber;  
    @Override  
    public boolean equals(Object other) {  
        if(! (other instanceof Person)){ return false; }  
        Person otherPerson = (Person) other;  
        if(this.socialSecurityNumber ==  
            otherPerson.getSocialSecurityNumber()) {  
            return true;  
        }  
        return false;  
    }  
}
```

The Object class defines an **equals** method. The default implementation of this method checks if the two references point to the same object, so they do not implement a business rule.

Since the equals method is inherited by all classes, developers can override the default implementation of the equals method to provide an implementation that resembles the business rule for the application they are developing.

The example shown above considers two persons to be equal when the employee id value of both instances is the same.

## Overriding the equals Method

- Can override the equals using different properties...
  - Note we are checking to be sure we are comparing the same type of object, as well as a business rule comparison

```
public class Person {  
    String name;  
    int socialSecurityNumber;  
    @Override  
    public boolean equals(Object other) {  
        if (!(other instanceof Person)) {  
            return false;  
        }  
        Person otherPerson = (Person) other;  
        if (this.name.equals(otherPerson.name)) {  
            return true;  
        }  
        return false;  
    }  
}
```

The diagram shows the Java code for the `Person.equals()` method. A red callout box points to the first `if` statement: "If not instanceof Person it will not be equal". Another red callout box points to the `name.equals()` call: "Using equals method of String class".

Another implementation of the equals method might be to check the name of both **Person** instances.

The first check is to ensure that the object being compared to is of the same type, otherwise it is not considered to be equal.

Once the 'other' reference has been downcasted (it is of the same type) we can compare the name values. Since the name is of type **String**, we can use the equals method of the String class to check if the two names are identical.

## The equals and hashCode Methods

- Every object has a **hashCode** method
  - Inherited from Object class
- If you override the **equals** method, should also override **hashCode** method
  - hashCode must generate equal values for equal objects
  - Multiple calls to hashCode must return same value
    - ◆ As long as none of the property values have changed
  - Both hashCode and equals methods must use the same set of significant fields
  - Same hashCode does **not** mean objects are equal!

```
public int hashCode() {  
    return ((name == null) ? 0 : name.hashCode());  
}
```

Another method that is inherited from the Object class is the **hashCode** method.

When the **equals** method is overridden, you should also override the **hashCode** method, keeping in mind these four rules:

- 1) The **hashCode** method must generate the same value for all instances that are equal (according to the **equals** method)
- 2) Multiple calls to the **hashCode** method must return the same value, as long as the state of the object has not been changed. (no property values have been changed)
- 3) The **hashCode** must be calculated using the same fields that were used to determine if the two objects are equal
- 4) The **hashCode** method may return the same value for two objects even when these objects are not equal. (according to the **equals** method)

## Default Constructor

- If a class has no constructors it is given a default no-arg constructor

```
class MyClass {  
    public static void main(String args[]) {  
        new MyClass(); // this will work  
    }  
}
```

- If a constructor is specified which takes arguments, Java will not provide a default no-argument constructor

```
class MyClass {  
    public static void main(String args[]) {  
        new MyClass(); // this won't work  
    }  
    MyClass(String someArg) {  
    }  
}
```



### The Default Constructor

Many applications that manage your classes for you will make the assumption that there is a default constructor. This should do something reasonable such as setting the variables to meaningful initial values.

Don't forget the issues discussed in the previous lesson regarding default constructors provided by the compiler.

## Implicit Constructor Chaining

- Parent class constructors
  - Child class constructors call no-arg by default

```
public class Employee{  
    private String name;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
  
    public Employee() {  
    }  
}  
  
public class SalesEmployee extends Employee{  
  
    public SalesEmployee() {}  
  
    public SalesEmployee(String name) {}  
}
```

- With this code, name would never be saved

```
SalesEmployee employee = new SalesEmployee("Kimberly");
```

### Invoking Superclass Constructors

For the same reason that we may choose to invoke overridden methods in the superclass, we may also choose to have our constructor pass initialization data to the corresponding superclass constructor.

If the superclass constructor is not explicitly invoked, the compiler will automatically invoke the no-argument constructor of the superclasses regardless of which constructor was invoked in the lowest level class being instantiated.

Following the OO paradigm of encapsulating services and reusing them, constructors can be chained together so each more specialized constructor makes use of the more generalized superclasses constructor. Employee is a more specialized form of Person.

As objects are built from the inside out (the highest superclass is built first, followed by the next, and so on, down to the most specialized class), the invocation of the superclasses constructor must be the first statement in the constructor.

## Passing Data Up Constructor Chain

- Private superclass data must be provided via the constructor
  - Private fields cannot be accessed by subclass
  - Superclass may optionally provide "setter" method

```
public class Employee{  
    private String name;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
  
    public Employee() {...}  
}
```

```
public class SalesEmployee extends Employee{  
    public SalesEmployee() {}  
  
    public SalesEmployee(String name) {  
        super(name);  
    }  
}
```

### Providing Initialization Data to the Superclasses

In many cases the decision to pass data up to the superclass constructor is not optional.

In this example an employee is constructed with a name. However, name is a private attribute of the superclass. It would be a mistake for the Employee subclass to duplicate the data storage of the name field. The only way to get this information into the Person superclass is to pass it to its appropriate constructor at initialization time.

## A Common Programming Mistake

- Parent class constructor may not have a no-arg constructor
  - Child class must explicitly call an available superclass constructor

```
public class Person {  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    /* public Person(); // if this constructor didn't exist  
     * */
```

- Child class no-arg is invalid, and won't compile

```
public class Employee extends Person {  
    public Employee() {  
        // to fix...  
        // this(); or super();  
    }  
  
    public Employee(String name) {  
        super(name);  
    }  
}
```

### What Not To Do

Remember that if the subclass constructor does not explicitly pick which superclass constructor to call, the compiler will automatically generate the code to call the no-argument superclass constructors.

If the Person class did not have a no-argument constructor, and there were no explicit superclass constructor calls in the subclass constructors, the compiler would automatically attempt to call the no-argument constructor of the superclass.

All of this is apparent. What is often not apparent is that when the subclass constructor that takes a String is invoked, this does not cause the superclass String constructor to be invoked.

If the superclass does not have a no-argument constructor, then every constructor in the subclass must explicitly call a valid constructor in the superclass.

## Lesson Review and Summary

- 1) How may a class subclass from multiple classes?**
- 2) Is it necessary to cast when assigning a subclass reference to an existing superclass reference?**
- 3) What two methods are normally overridden in the subclass?**
- 4) What is the operator that tests the type of an object reference?**
- 5) Is it possible for a constructor to invoke a superclass constructor?**
- 6) Can a class be written without any constructors, and then instantiated?**

1. It may not (multiple inheritance is not possible)
2. No
3. The equals and hashCode methods
4. instanceof
5. Yes
6. Yes, the default (no-argument) constructor is created when no constructors are defined

## Exercise 7: Creating Subclasses

`~/StudentWork/code/SubClasses/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Session: Essential Java Programming**

**Fields and Variables  
Using Arrays  
Java Packages and Visibility**

# Lesson: Fields and Variables

**Fields and Variables**  
Using Arrays  
Java Packages and Visibility

## Lesson Objectives

- After completing this lesson, you will be able to:
  - Initialize instance variables to default values
  - Distinguish between instance variables and method variables within a method
  - Initialize method variables prior to use
  - Explain the difference between the terms **field** and **variable**
  - List the default values for instance variables of type `String`, `int`, `double`, and `boolean`
  - Name the keyword used to create constants in Java

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Fields vs. Variables

- **Fields are variables declared within the scope of the entire object**

```
class MyClass {  
    String myStringField;  
    int myIntField;  
    void someMethod() {  
        ...  
    }  
}
```

- **Local Variables are declared within the scope of a method**

```
...  
public boolean equals(Object other) { // other is a variable  
    String otherStr; // another variable  
    ...  
}
```

- **These are naming conventions used to clarify the scope of a variable**

### Naming Conventions

In Java, a field refers to a member of the class. It is defined in the class block, and is accessible by all the methods of the class.

A local variable is the term used to denote a variable of a method. This includes the method arguments and the primitive and object references declared within the body of the method. These variables cease to exist once the execution returns from the method.

## Instance vs. Local Variables: Usage Differences

- When an object is created, all of its instance variables are initialized
  - Implicitly to their default values
  - Explicitly initialized to some other value
- Local variables are not implicitly initialized
  - The compiler will not let you use a local variable which has not been explicitly initialized

```
class Variables {  
    int x = 1; // Instance variable  
    int y; // Defaults to zero  
  
    public void aMethod() {  
        int x; // hides the instance variable  
        int y = 5; // hides the instance variable  
        y = x; // won't compile, x has no value  
    }  
}
```



### Some Differences

Java states that all instance variables are given a default value or a specified value at initialization. The life of these variables is the life of the object.

However, variables created in methods (automatics or locals) have a much shorter life and have no predefined default value. These variables must be initialized before it is used. The reason for this is that they live on the stack within a stack frame and, for reasons of efficiency, these variables are not initialized automatically. Their scope is from the initial call to the method until the return of that method.

This is a major improvement of Java over C, which did not check to see if a variable was properly initialized before it was used.

# Data Types and Variables

- **Primitive data types**
  - For reasons of efficiency, Java has primitive types as well as objects
  - Java's primitive types are constant across all platforms, making portability possible
- **Object reference – holds reference to an object**
  - Object references declare the class type they will hold
  - Object references are "type safe"
- **Array – hold a list of variables**
  - Can be typed to primitives or object references
- **Variables may be initialized when they are declared**

## Variable Types

Java's variable types are platform independent. They are always the sizes specified above. Object references hold the address of the object, not the object itself. We cannot do any arithmetic on object references, thus, Java has no "pointers". Strings and arrays are true objects.

Booleans are a unique type with values of true or false, not zero versus non-zero. Object references are type safe. You can only store the type of class you declare, or a sub type of that class in an object reference. Unlike C, you cannot do arithmetic on an object reference, so you cannot inadvertently modify memory outside of your control.

## Primitive Data Types

boolean	true or false (not 0 versus non 0)
byte	8 bit signed number
short	16 bit signed number
char	16 bit Unicode character
int	32 bit signed number
long	64 bit signed number
float	32 bit IEEE floating point number
double	64 bit IEEE floating point number

## Default Values

- When declaring instance variable without explicit initialization values, they will be set to their default values
  - integer types → 0
  - float types → 0.0
  - boolean → false
  - char → \u0000 (hex zero)
  - Reference types → null
- Local variables are not set to default values implicitly
  - Must be explicitly initialized before they can be referenced

### Instance Variables

When objects are created, instance variables are automatically set to their default values. Object references are set to null, which is distinct from 0. If you try to use a null object reference, the program will cause a runtime error to occur.

We will learn how to process this error later in the exceptions section of the course.

## Block Scoping Rules

- Examples of code blocks in Java
  - The method (a block within the class)
  - Language statements (if, for, while, etc.)
  - Exception handling code
  - Explicit code blocks (not used often)
- Local variables within blocks "hide" fields of the same name outside the block
- A variable is invalidated when the block in which it was declared is no longer being executed

```
for (int a = 0; a < 10; a++) {  
    System.out.println("a is " + a);  
}  
  
// The following line of code is invalid  
System.out.println("a finished at " + a);
```



### Reminder

Local variables with the same name as instance variables will hide the instance variable from the method. You must use the this reference to see the instance variable if you have an instance variable with the same name as a local variable. Again, you can always see out of a block, but not into a block of code.

## Using this

- The **this** keyword is used to resolve ambiguity between instance variables and same-named local variables

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

- **this** refers to **this** object instance

### The this Pointer

Every object has a this pointer. It is automatically created and initialized at the time the object is created. It is often used to distinguish a field from a local variable by the same name.

## Final and Static Variables

- A **final field is a constant**
  - Often is public to allow external access
  - Can be used with method arguments
  - Must be initialized before it can be used
- A **static field is visible without object instance**
- Constants are often static to allow access without instantiation of the defining class
- Final fields can only be initialized in declaration

```
public final class FontColor {  
    public final static Color BLACK = new Color(0, 0, 0);  
  
    private JTextArea createTextArea() {  
        JTextArea txt = new JTextArea();  
        txt.setFont(new Font("Verdana", Font.BOLD, 12));  
        txt.setForeground(Color.BLACK);  
        return txt;  
    }  
}
```

### Special Field Types

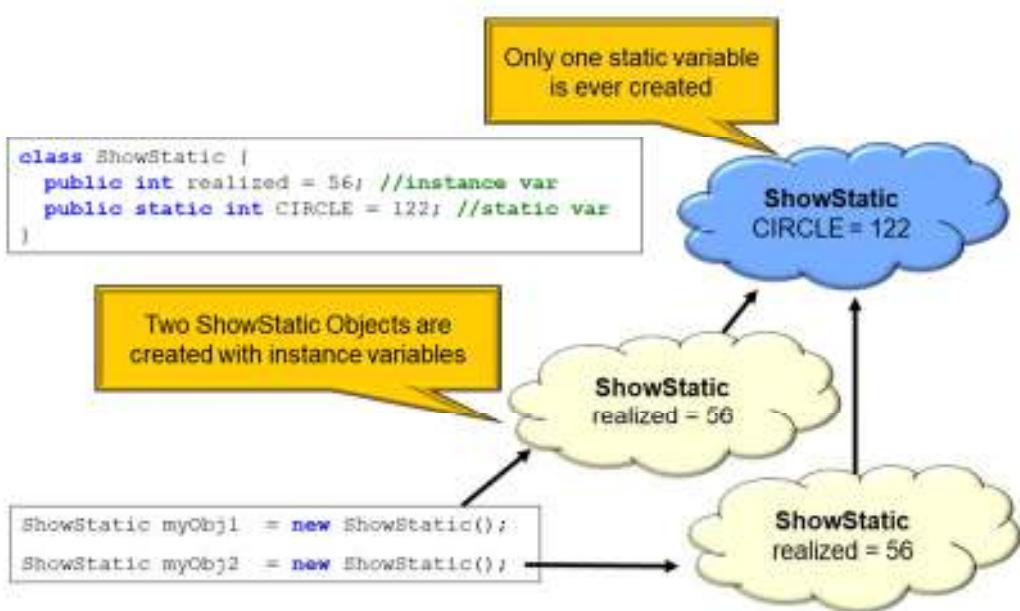
These modifiers can only be used on fields, not local variables in a method.

When a field is declared final, its contents cannot be changed. It becomes a constant. Generally, final fields are named in all upper case letters to easily distinguish them from normal fields.

A static field is a field in which its contents exist within the class definition, not within an object instance. This means that this field can be accessed by all instances of the class, and by other classes if permitted by access visibility.

Often, constants (final) are made static so that their values can be used by other classes. Static fields will be covered in more detail in a later lesson.

## Static Variable Diagram



## Static Fields

- Only one copy per class
  - Instance variables are one copy per object instance
  - Static fields are shared by all instances
  - (Note: This applies per classloader)
- Accessed as *Classname.fieldname*
  - Color is a class name; red is the static field  
`Color.red`
- Two types of static fields are used
  - Mutable fields  
`static int theCount;`
  - Read-only fields (constants)  
`static final int REQUEST_LIMIT = 10;`
    - final fields must be initialized when declared
    - Convention is for final fields names in all upper case

### The Static Field

The static field belongs to the class and not to the instance. It is created when the class is loaded into the JVM. There is only one of these no matter how many instances there are. All the instances share the same single object or method.

Be aware that all instances have access to the class variable and can change its value if it was not declared final. There needs to be agreement and rules concerning access and its mutability.

This applies to methods as well. Static methods cannot call instance methods. This is because there is no guarantee that an instance exists nor, if there are multiple instances, which one to call.

## Simple Example of Static Fields

- Counts all object instances of class

```
public class CountingClass {  
    public static int theCount;  
    public CountingClass() {  
        theCount++;  
        System.out.println("The count is " + theCount);  
    }  
    public void someMethod() {...  
        if (theCount > 5)  
            ... // do something  
    }  
}  
  
public class Tester {  
    public Tester() {  
        System.out.println(CountingClass.theCount);  
    }  
}
```

### Static Fields

Static fields are shared by all instances of a class. They are useful for keeping counts of numbers of objects created, or as a reference to a shared resource by all members of the class.

## Static Methods

- Can be used without creating an object instance
- Useful for utility functions that don't require object state
- Commonly used static method is `main`
  - Tells the JVM where to start executing
  - Must be:

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

- Limitations of static methods
  - Cannot access instance variables (non-static fields)
  - Cannot invoke non-static methods
- Do not overuse static methods

### Static Methods

Static methods can be used without creating an instance of a class. The most familiar of these is the `main` method which is used to bootstrap execution of the program. It is important to remember that a static method cannot use any of the instance variables defined in the class without having an object reference to use.

## Lesson Review and Summary

- 1) Is it necessary to initialize instance variables before using them?
- 2) Is it necessary to initialize local variables before using them?
- 3) Can a local variable be the same name as an instance variable?
- 4) How are constants created?
- 5) Can constants be referenced without instantiating the class?
- 6) What are the default values for instance variables?

Instance variables have default values, and do not need to be initialized before they are used

Local variables must always be initialized before they can be referenced (read)

Local variables hide instance variables of the same name, unless the `this` operator is used to explicitly reference the instance variables

Constants are created using the `final` keyword

Constants can be referenced without instantiating the class by using the `static` keyword

## Exercise 8: Field Test

`~/StudentWork/code/FieldTest/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Using Arrays

Fields and Variables  
**Using Arrays**  
Java Packages and Visibility

## Objectives: Using Arrays

This lesson covers how to use arrays in Java. Specifically, it covers:

- Declaring an array reference
- Allocating an array
- Initializing the entries in an array
- Writing methods with a variable number of arguments

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Arrays

- An array is a fixed sequence of a specific type
  - To declare an array reference

```
String myStringArray[];
```

`int[] myIntArray;`
  - To allocate the array object

```
myStringArray = new String[4];
```

```
Employee emp = new Employee("Elias");
Employee[] e = {emp, new Employee("Jennifer")};
```
  - ◆ Note the use of square brackets
  - The array allocation above has created space for four String references (initialized to null)
    - ◆ Not five String objects
- Filling the array with data

```
myStringArray[0] = new String("String number one");
myStringArray[1] = new String("String number two");
myStringArray[2] = new String("String number three");
myStringArray[3] = new String("String number four");
```

### Arrays

Arrays are true objects. They can hold primitive values or object references. Use of arrays is fail-fast, meaning that if your program attempts to index an array outside of its allocated bounds, an **ArrayIndexOutOfBoundsException** is immediately thrown. Array indexes are 0 based.

### Allocation of Arrays

A program will fully allocate an array in three logical steps. The Java language allows the syntax that can perform these steps in fewer statements. We've illustrated the three steps separately here for clarity.

First, the array reference type is declared. At this point we may have an array of type **String**, yet we haven't yet allocated the space for the N number of references.

In the second stage we allocate the space for the array. Remember that this is an array of references, not an array of objects. At the end of this stage we have N references of the specific type, but no objects of that type yet.

The third stage is where each of the references is set to point to a valid object instance of that type.

## Accessing the Array

- An integer is used to subscript the array

```
System.out.println(myStringArray[0]);
```

- Understanding the Java types

- myStringArray is a String[]
- myStringArray[x] is a String

- Arrays are special Java objects

- length is an attribute of the array

```
for (int a = 0; a < myStringArray.length; a++)  
    System.out.println(myStringArray[a]);
```

- This will throw an **ArrayIndexOutOfBoundsException**

```
int index = 5;  
System.out.println(myStringArray[index]); // past the end
```

### Arrays

Arrays are full fledged objects in Java with special syntax built into the language to support the creation and access to arrays. Arrays are type safe.

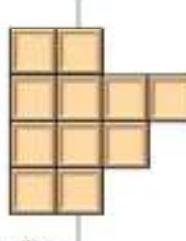
The most common thing invoked on arrays is the length attribute. Regardless of whether the references within the array are valid, this attribute will always return the number of slots in the array. Note that you cannot change the length of an array.

Indexing is done using integer primitives. Java will automatically check that you are not attempting to index past the end of the array (probably the single most pernicious bug in C and C++).

## Multidimensional Arrays

- **Arrays can be multidimensional of one type**
- **To create an array of String arrays**

```
String multi[][];
multi = new String[4][];
multi[0] = new String[2];
multi[1] = new String[4];
multi[2] = new String[3];
multi[3] = new String[2];
multi[0][0] = new String("Hello");
multi[0][1] = new String("What is your name?");
multi[2][0] = new String("Thank you for shopping with us");
...
```



- **Understanding the Java types**
  - **multi[x]** is a **String[]**
  - **multi[x][y]** is a **String**

### Multidimensional Arrays

Multidimensional arrays are essentially arrays of arrays. They do not have to be rectangular as the above example illustrates.

While an interesting idea, in most cases multidimensional arrays are not used very often. To maintain more complex sets of data, we generally use one of the **Collection** classes.

## Copying Arrays

- Copying arrays is very common
  - Reading data from an input stream

```
byte[] data = new byte[5000]; // big enough to hold data
...
int count = in.read(data);
byte[] dataRead = new byte[count];
for (int a = 0; a < count; a++)
    dataRead[a] = data[a];
```

- Array expansion is another case for array copying
- Use `System.arraycopy` instead
  - Is a native method
  - Far more efficient than corresponding Java code shown above

```
byte[] dataRead = new byte[count];
System.arraycopy(data, 0, dataRead, 0, count);
```

### Let the JVM Work

The built in `arraycopy()` method is much faster than any code you could possibly write, even with a JIT. Try it to convince yourself. `arraycopy()` is extremely useful for reallocating the size of an array.

## Variable Argument Support

- Methods might require variable number of arguments
  - Could use an array

```
public void testOldVarargs() {  
    Object[] values = { "string arg", new Date(), new Integer(42)  
    someObject.process(values);  
}
```

Prior to Java SE 5 the implementation of methods that needed a variable number of arguments was done using one or two arrays

## Variable Argument Support (cont'd)

- With variable arguments, the method to be called defines a class type followed by ellipses and an argument name

```
public void process(Object... args) {  
    for(Object o : args) {  
        System.out.println("incoming arg = " + o);  
    }  
}
```

- The caller sends in the arguments in an agreed upon order

```
// Notice the use of autoboxing as well.  
public void testVarargs() {  
    someObject.process("string arg", new Date(), 42);  
}
```

Prior to Java SE 5 the implementation of methods that needed a variable number of arguments was done using one or two arrays

## Varargs Rules

- **Variables arguments must be last in the defined method's argument list**  

```
public void process(Date d, String command, Object... args) {...}
```
- **Be careful how you overload methods that support varargs**
- **Arrays in a variable argument list must be cast to be type Object**
  - Or the compiler will not treat it as a member of the varargs list; it will treat it as the actual variable argument list

```
public void testVarargs() {  
    Object [] list = { "one", new Date() };  
    someObject.process(list); //Not correct for varargs  
    someObject.process((Object)list); //Correct use  
}
```

## Lesson Review and Summary

- 1) What method is used to increase the size of an array?**
- 2) How can you find the number of slots in an array?**
- 3) Is it possible to create multi-dimensional arrays?**
- 4) How are elements in an array set and retrieved?**

Arrays are fixed in size once allocated

The length field can be accessed on an array to determine its size

A multi-dimensional array can be created, but only of one type

The elements in an array are accessed using integer indexing

## Exercise 9: Creating an Array

`~/StudentWork/code/Array/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Lesson: Java Packages and Visibility**

Fields and Variables

Using Arrays

**Java Packages and Visibility**

## Lesson Agenda

- Using the `package` keyword to define a class within a specific package
- Using the four levels of accessibility/visibility
- Using the `import` keyword to declare references to classes in a specific package
- Using the standard type naming conventions when creating packages, classes, and methods
- Correctly executing a Java application class that is defined in a package

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## The Problem

- Consider a class named Patent
  - In a legal library it is a document with the abstract, registration number, etc.
  - In a shoe manufacturing company it is a leather with color, thickness, cost, etc.
- What if we are writing a program for the legal department of a shoe manufacturer?
  - How can we use both classes in a single program or expression?

### Significance of Names without a Context

The English language is rich but not rich enough for computers. Thus we have name clashes such as the one shown above for Patent. Packages take care of isolating one name from another by providing an explicit context.

# Packages

- Every class/interface belongs to a package
  - A means of organizing code into logical groups
    - ◆ Classes in a package tend to change together
    - ◆ Often maintained by the same programmer
  - Helps avoid name clashes and increase safety
    - ◆ A package is a namespace for the artifacts within
    - ◆ Package begins with organization's top level domain in reverse
  - If no explicit package is declared, then the class is in the default package
    - ◆ It is not good practice to leave classes in the default package
- If the Car class is in the package com.foo.vehicle then its qualified class name is com.foo.vehicle.Car
  - By convention, package names are in lower case

## About Packages

Packages are a means of organizing code into logical groups. In Java, packages map directly to a directory structure. In the case of **java.lang.Object**, the **Object** class is located within the directory **java/lang/**.

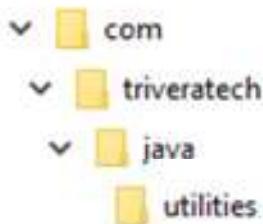
Packages have two distinct sets of classes, those which are made public and those which are hidden inside the package. Hidden classes typically are support classes for the public ones and are not meant to be used outside the context of the package.

The convention in Java is that package names are lower case, class names start with upper case, and field/method names with lower case. This helps disambiguate the dot operators.

An example of this is **java.lang.System.out.println()**, where **java.lang** is a package/subpackage, **System** is a class, **out** is a static field within System (a field), and **println()** is a method in the object referred to by out.

## Class Location of Packages

- Packages are grouped by directory
  - They are hierarchical
  - Convention: map to an Internet domain
    - ◆ Example: com.triveratech.java.utilities



### Mapping Package Paths to Directory Paths

Packages map to the underlying directory structure where the classes are stored, the CLASSPATH.

The plan is that all packages will have a directory structure that maps to the reverse domain name of the producer. If this scheme is adhered to then all packages themselves will not have name clashes. For example the package name might be **com.triveratech.java.utilities**.

## The package Keyword

- A package statement at the beginning of a .java file designates which package its contents belongs to
  - A class may only belong to one package

```
// Note how the package name maps to the directory structure.  
package com.triveratech.java.utilities;  
  
public class MyClass {  
    public static void main(String[] args) {  
        "  
    }  
}
```

- The class compiled from the above would be located in the file  
`com\triveratech\java\utilities\MyClass.class`
- Access to the class requires the fully qualified name:  
`com.triveratech.java.utilities.MyClass`

### The Package Keyword

Each class may belong to only one package. That is, only one package statement is allowed per file. The package statement must be the first statement in the source file.

In the case of our Java development environment, the package keyword is hidden from us because the package location of the class is implied by the active package name selection in the package tree display.

## Importing Classes

- Using the full names of classes can be long and cumbersome
  - ```
java.util.Date aDate = new java.util.Date();
```
- Java provides the import statement to import classes
  - Once a class has been imported it can be referenced by name
    - ```
import java.util.Date;
```

  
...  

```
Date aDate = new Date();
```
  - All members of a package can be imported at once by using the \* wildcard
    - ```
import java.util.*; // imports all classes from java.util
```
  - Beware of importing identically names classes from different packages
- Importing does not add to the program size

### Importing Classes

Since the proper class name of a class is the package path name followed by the class name, we would have to reference this fully qualified name in our source code.

For classes whose actual class name (such as "Date") are unique within our source code, it becomes convenient to refer to the class by its short name.

However, since the class name could exist in other packages we have to tell the compiler from which package we wish to obtain the class. For example, there is a **Date** class in **java.sql** in addition to the **Date** class in **java.util**.

The import statement can be used to direct the compiler to a specific class of a specific package, or to all classes of a package. We can also have multiple import statements in our code.

In the case where there would still be a conflict, say using the **Date** class from both packages listed above, we could import one, but then use the full path name for the other (such as "**java.sql.Date**").

## Executing Programs

- This will fail

```
PROMPT> java MyClass
```



- This will work

```
PROMPT> java com.triveratech.java.utilities.MyClass
```

### Proper Class Names During Execution

Since the proper name of a class is its full name (package name plus class name) we must provide the fully qualified class name to the runtime interpreter.

This is probably the single biggest problem new developers have when trying to run their code.

## Accessibility/Visibility

|           |   | Inside package |             |        | Outside |             |
|-----------|---|----------------|-------------|--------|---------|-------------|
|           |   | Class          | Constructor | Method | Class   | Constructor |
|           |   | •              | •           | •      | •       | •           |
| private   | ✓ | ✗              | ✗           |        |         |             |
| 'default' | ✓ | ✓              | ✓           |        | ✗       | ✗           |
| protected | ✓ | ✓              | ✓           |        | ✗       | ✗           |
| public    | ✓ | ✓              | ✓           |        | ✓       | ✓           |

### Using Visibility

Visibility modifiers allow the programmer to encapsulate or hide parts of a class or the class itself! Typically, as a programmer, you should allow the minimum level of visibility unless there is a reason for making it more visible. This will help prevent others from using the class in ways that it was not intended to be used.

Java supports four levels of visibility:

**public** – Allows anyone access. Typically the class, constructors and worker methods are made public.

**default** – Is used when there is no explicit visibility modifier. Allows only classes inside the same package to access.

**protected** – Allows access to classes inside the same package and subclasses of the protected class.

**private** – Allows access from only the class itself. Typically is used for instance variables and hidden methods. Use this modifier if there is no need for anyone outside of the class to use its methods/variables.

## Java Naming Conventions

- Consider `xxx.yyy.zzz()`
  - It could mean: `package.class.method()`
  - Or `class.field.method()`
- Style rules to minimize confusion:
  - Package name: `lower.case.components`
  - Class name: `MixedStartsUpperCase`
  - Field/Method: `mixedStartsLowerCase`
  - Constants: `ALL_UPPER_CASE`
- Under this convention:
  - `xxx.yyy.zzz()` is probably `class.field.method()`
  - `xxx.Yyy.zzz()` is probably `package.class.method()`
- These conventions make code easy to read and understand
- These rules are not enforced by the compiler

### Use Consistent Naming

It is very important to using consistent naming techniques. You can see from the examples above what a difference it makes. You will also be able to read other code that adheres to the Java convention.

## Lesson Review and Summary

- What are two benefits of defining classes within a hierarchical package namespace?
- What are the three access modifiers that can be placed on methods?
- Does the import operation increase the size of your application?
- Is it necessary to know the package name of an application when invoking it from the command line?
- Are package names generally in upper case or lower case?

Packages provide a hierarchical namespace which helps to eliminate class name conflicts

Packages provide a framework for scope of access when access modifiers are used

The three method and field access modifiers are `public`, `private`, `protected`, plus default (no modifier)

Importing a package tells the compiler to look in that package to resolve relative class name references

The proper name of a class is its package name followed by its class name

Package names should be in all lower case

## Student Activity (10 minutes) – Refactoring in Your IDE

- Startup your IDE and open your workbench. The following instructions are typical for an Eclipse-based IDE. Yours may vary somewhat.
- In the Package Explorer on the left side, right click on the TestProject entry that you created earlier
  - Select Refactor-> Rename
  - Rename the project to TestingProject
- Using the same approach
  - Refactor the package to testing.core
  - Refactor the Java class name to TestingClass
- Note how your IDE ensures that the internals of the class are properly coordinated with the refactoring operations that you are performing
- In the TestingProject
  - Create a new package called dummy
  - Copy the TestingClass (right click option)
  - Paste a copy of TestingClass into the dummy package
  - Run dummy.TestingClass
- Delete the TestingClass in the dummy package
  - Also delete the dummy package

# **Session: Advanced Java Programming**

**Inheritance and Polymorphism  
Interfaces and Abstract Classes  
Exceptions**

# **Lesson: Inheritance and Polymorphism**

**Inheritance and Polymorphism**

Interfaces and Abstract Classes

Exceptions

## Lesson Agenda

- Writing a subclass with a method that overrides a method in the superclass
- Grouping objects by their common supertype
- Utilizing polymorphism by correctly invoking subclass method implementations through superclass references
- Safely casting a supertype reference to a valid subtype reference
- Using the `final` keyword on methods and classes to prevent overriding through subclassing

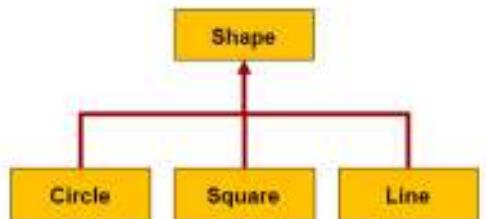
### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Polymorphism

- Consider the following definition of classes



- The Shape class is implemented as follows

```
public class Shape {  
    public void draw() {  
        System.out.println("Drawing Shape");  
    }  
}
```

### About Polymorphism

Polymorphism is one of the most powerful concepts of object oriented programming. The key to polymorphism is deciding at runtime exactly what kind of object we are working with.

Polymorphism comes from the Greek: poly = many, morphism = forms. It allows code to be written that is self modifying so that when new classes are added to an inheritance hierarchy to augment the functionality of the program, no changes need to be made to the existing code.

It depends on the principle of substitutability where a superclass object can hold a reference to itself or any of its subclasses.

## Polymorphism: The Subclasses

- **The subclass implementations:**

- **The Circle class**
- **The Square class**
- **The Line class**

```
class Circle extends Shape {  
    public void draw() {  
        System.out.println("Drawing Circle");  
    }  
}
```

```
class Square extends Shape {  
    public void draw() {  
        System.out.println("Drawing Square");  
    }  
}
```

```
class Line extends Shape {  
    public void draw() {  
        System.out.println("Drawing Line");  
    }  
}
```

### Subclasses

Notice that we have defined special versions of the `draw` method for each of our specialized shapes. The method has the same signature as that in the super class. This is a key requirement for making polymorphism work.

A method signature is the unique combination (for the associated class) of a method name and the names and types of parameters.

Note: Each subclass overrides the parent's `draw` method.

## Derived Classes as the Superclass

- Run time binding will call subclass method, even though it is referenced through superclass type

```
Shape shapes[] = new Shape[4];
// shapes[0] = (Shape) new Circle() - upcasting is implicit
shapes[0] = new Circle();
shapes[1] = new Square();
shapes[2] = new Line();
shapes[3] = new Shape();

for (int a = 0; a < shapes.length; a++)
    shapes[a].draw(); // calling Shape.draw, but will actually
                      // invoke Circle.draw, Square.draw, etc.
```

- The output:



```
PROMPT> java SuperClass
Drawing Circle
Drawing Square
Drawing Line
Drawing Shape
```

### Polymorphism in Action

We now store our specialized objects in the generalized container and call their draw methods. Notice that inside the for loop, we have no idea what kind of shape we are drawing. This is polymorphism in action.

The process of abstraction is key to good object oriented programming practice. Java facilitates this by allowing the specialized class to be referenced as its superclass. This allows us to treat things in a more generalized way and greatly simplifies code.

One of the nice things about the Java™ language is the ability to use subclasses in the place of superclasses. What this means is that you can assign a subclass object to a superclass object reference. Additionally, the subclass does not need to be the immediate child of the superclass, the subclass could extend an object which extends the superclass. Assigning this subclass to an object reference of a "grandparent" is still valid. The reason that this is valid is that the grandchild object (subclass) will still contain all of the characteristics of the grandparent (superclass).

## Casting to the Derived Class

- If an object reference is to a superclass type

```
Shape shape = new Circle(); // the implicit upcast
```

- Then we can't access the methods of the subclass

- The getRadius method doesn't exist for type Shape

```
shape.getRadius()
```



- Must downcast to a the subclass type

- Circle is a subclass of Shape, the compiler allows this

```
Circle circle = (Circle) shape; // explicit cast is necessary
```

- But if shape were initialized as Line, explicit cast would fail at runtime

```
Shape shape = new Line();
Circle circle = (Circle) shape; //fails
```

### Casting

There are times, however, that we need to find out if a generalized reference is really a specialized instance. The **instanceof** operator allows us to test what kind of instance we have. Once we have confirmed the fact that we have a specialized instance, we can then cast the general reference to the actual subtype we have, and access the specialized methods of the instance.

## Using instanceof for Downcasting

- If a downcast is not valid, ClassCastException will be thrown

```
Shape shape = new Circle();  
  
Line line = (Line) shape; // compiler allows this!  
//throws an exception at runtime!
```

- The instanceof operator tests the type of the object referenced (not the declared type of the reference variable itself)
  - shape is declared as a reference to type Shape
  - shape is actually referencing the Shape subtype Circle
- This expression would now evaluate as true

```
if (shape instanceof Circle) {  
    Circle circle;  
    circle = (Circle) shape; // this is now safe  
}
```

### Verifying the Type Before Casting

The Java language includes the keyword **instanceof** to help us determine what the exact class of a subclass is. An example is shown in the slide above. However it is better, where possible, to have code explicitly designed to handle a particular type of object rather than branching within a class based on the type. Testing the object type and then processing based on type is procedural code, whereas having a dedicated object to handle the object is more object oriented.

This aspect of Java will become clear when you learn to handle events from GUI objects.

## Upcasting vs. Downcasting

### • Upcasting

- Runtime-safe, checked by the compiler
- Can be used to fill an array of related types
- Often done to satisfy method signature requirements

```
addShapeToList( (Shape)circle );
```

### • Downcasting

- Can only partially be checked by the compiler
- Used to expose additional subclass methods
- Can convert collection of generic types to more specialized forms

```
void drawShapes(Shape shapes[]) {  
    for (int a = 0; a < shapes.length; a++) {  
        shapes[a].draw(); // makes sense, a shape method  
  
        if (shapes[a] instanceof Circle)  
            ((Circle)shapes[a]).getRadius();  
        if (shapes[a] instanceof Line)  
            ((Line)shapes[a]).getArrowHead();  
    }  
}
```

### Upcasting

We can always upcast, because when we do so, we lose access to specialized behavior. We cannot downcast without checking the type, because the specialized behavior may not exist. If we do a bad downcast, the error will be caught at runtime, not compile time.

Note: Is this good design? **drawShapes** takes a **Shape []**, yet goes beyond **Shape** behavior to get more specialized behavior!

## Calling Superclass Methods From Subclass

- An overriding method in a subclass may wish to call the overridden method from the superclass
  - Superclass

```
public class Person {  
    private int speed = 5; // can't be seen from subclass  
    public int getSpeed() {  
        return speed;  
    }  
}
```

- Subclass

```
public class Athlete extends Person {  
    private int speedFactor = 2;  
    public int getSpeed() {  
        return super.getSpeed() * speedFactor;  
    }  
}
```

### Leveraging Superclass Methods

When a subclass overrides a method of its superclass, the superclass' method is hidden. This causes a problem if we want to use the superclass' method instead of the subclass'.

To access the superclass' method, we can use the **super** keyword. The syntax is, **super.MethodName()**.

## The final Keyword

- The **final** keyword prevents modification

- **final** classes may not be subclassed

```
public final class LotteryPredictor {..}
```

- **final** methods may not be overridden

```
public class SecureProtocol {
    public final boolean verifyCredentials(Credentials c) {
        ...
    }
}
```

- **final** fields cannot have their value changed

```
private final int TAX_RATE = 33;
```

### The **final** Keyword

Sometimes we may wish to ensure that a class or one of its methods are not subclassed or overridden. If a method performs a vital function, such as authentication, it would be very trivial for someone to override this method and implement a non-secure method in its place. We can stop this by placing the **final** keyword before the return type of the method.

## Using **final** Variables

- Method variables can be defined as **final**

```
public void calcPayment() {
    final int value = 15;
    //value= 25; Compilation error
    ...
}
```

- Method parameters can be defined as **final**

- Reference is **final**, object state can still be changed!

```
public void calcPayment(final Employee emp) {
    //emp = new Employee(); //Would cause compiler error
    emp.setSalary(0); //Allowed !
}
```

Variables declared within the scope of a method can also be declared as **final**. Once these variables have been given a value, the value cannot be changed.

As we have seen before, method parameters are local variables to the method. Method parameters can also be defined as **final**. By doing so it becomes impossible to assign a different value (or reference) to the parameter.

Note: While final fields cannot be changed, remember that object references are just references, not a copy of the object, so the underlying object may change even though the reference to it is declared **final**.

## 'Blank final' Fields and Variables

- 'Blank final' variables are not initialized when declared
  - Final instance fields must be initialized in constructor

```
public class Payment {  
    private final int TAX_RATE;  
    public Payment(){  
        TAX_RATE= 33;  
    }  
}
```

- Final method variables can only be set once

```
public void calcPayment(){  
    final int value;  
    ...  
    value = 15; //OK  
    ...  
    //value= 25; Compilation error  
}
```

Final fields and variables do not always need to be initialized when they are declared. Instance fields must be initialized when the object instance is being constructed. When the final field has not been given a value when it was declared, it must be given a value in the constructor of the class (making sure all constructors of the class initialize the final field).

When a method variable is defined as final, it must be given a value before it can be used (identical to non-final method variables). Once it has been given value, its value can not be changed.

## Lesson Summary

This lesson covered how to use inheritance and polymorphism. Specifically, it covered:

- Writing a subclass with a method that overrides a method in the superclass
- Grouping objects by their common supertype
- Utilizing polymorphism by correctly invoking subclass method implementations through superclass references
- Safely casting a supertype reference to a valid subtype reference
- Using the `final` keyword on methods and classes to prevent overriding through subclassing

## Exercise 10: Salaries - Polymorphism

`~/StudentWork/code/Polymorphism/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Lesson: Interfaces and Abstract Classes**

Inheritance and Polymorphism  
**Interfaces and Abstract Classes**  
Exceptions

## Lesson Agenda

- Define supertype contracts using interfaces
- Define supertype contracts using abstract classes
- Implement concrete classes using interfaces
- Implement concrete classes using abstract classes
- Explain one advantage of interfaces over abstract classes
- Explain one advantage of abstract classes over interfaces

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Separating Capability from Implementation

- An object has an *interface* and an *implementation*
  - The programming interface is the set of visible methods
  - The implementation is hidden within those methods
- An object can safely change its implementation provided it does not change its programming interface
- The Shape class is providing a programming interface consisting of one method

```
public class Shape {  
    public void draw() {  
    }  
}
```

- Java provides a formal way of representing these programming interface contracts
  - interface
  - abstract class

### Encapsulation

The most fundamental rule in object oriented programming is that the internal state of an object is not directly accessible from external objects. Instead, aspects of the object's state are accessed through methods. We call this internal hiding of state encapsulation.

With a well defined set of methods on a type, we say that the type provides an interface to external objects. As long as the interface doesn't change, we can change the internal implementation of the object without affecting the rest of the application. This is one of the core principles regarding software reliability and maintainability.

### Java Interfaces

In Java there are two language constructs that allow us to represent an interface, the abstract class and the interface. Keep in mind that this is in addition to the standard Java class which we've seen so far, which is also considered a type.

## Abstract Classes

- An abstract class is a class with zero or more **abstract methods**
  - An abstract method is a method signature with no body

```
// note the semi-colon, no code
public abstract void doSomething();
```
- Allows for a class to provide a partial implementation, while still requiring a complete implementation in a subclass (or concrete class)
- Concrete class – conventional term used to distinguish a class from an abstract class
- An abstract class cannot be instantiated, only extended
  - An abstract class can extend another abstract class
  - The concrete subclass must ensure that all abstract methods are implemented

### Abstract Classes

Abstract classes are sets of rules that the subclasses must implement for the code to compile. The rules are composed of method signatures that require a concrete implementation. `eat()` in the example above.

However, abstract classes can themselves have concrete implementations of methods. These methods are inherited just like a method in a non abstract class. In fact, an abstract class can have no abstract methods although it would not be that useful.

The rule in Java is that if a class has at least one abstract method, then it is itself abstract and must have the `abstract` keyword in its definition.

## Shape as an Abstract Class

- Shape should be an abstract class
  - Is probably too generic to be a real shape
  - The draw method can never be intelligently implemented

```
public abstract class Shape {  
    protected int startX, startY;  
  
    public int getStartX() {  
        return startX;  
    }  
  
    public int getStartY() {  
        return startY;  
    }  
  
    public abstract void draw();  
}
```

- The Circle subclass inherits coordinate functionality, and adds required drawing capability

```
public class Circle extends Shape {  
    public void draw() {  
        ...  
    }  
}
```

### The Shape Class

Since Shape is really too generic to exist on its own, we'll create it as an abstract class. This way, we can provide some functionality in the class, yet not allow it to be instantiated. It can be subclassed, however, and the subclass must provide the implementation of the draw method, which presumably can only be done by a real shape.

## Polymorphism With Abstract Classes

### Let's revisit the Shape classes

```
public abstract class Shape {  
    private String name;  
    public String getName() {  
        return name;  
    }  
    public abstract void draw();  
}  
  
public class Circle extends Shape {  
    public void draw() {  
        System.out.println("Circle");  
    }  
}  
  
public class Square extends Shape {  
    public void draw() {  
        System.out.println("Square");  
    }  
}  
  
Shape shapes[] = new Shape[3];  
shapes[0] = new Circle();  
shapes[1] = new Square();  
shapes[2] = new Line();  
  
for (int a = 0; a < shapes.length; a++) {  
    shapes[a].draw();  
}
```

### Using the Abstract Class as the Common Supertype

We cannot instantiate an abstract class, we can only instantiate its concrete subclasses. But, we can create an object reference to an abstract class. This object reference can "point" or refer to any of its subclasses. This allows us to call methods defined in the abstract class without knowing how the subclass implements them.

# Interfaces

- An interface is similar to an abstract class
  - Defines only abstract method implementations
  - The methods of an interface are always abstract
  - Using the **abstract** keyword is optional
  - Does not define instance variables

```
public interface Drawable {  
    public abstract void draw();  
}
```

- Would be de same as

```
public interface Drawable {  
    void draw();  
}
```

## About Interfaces

An interface is similar to an abstract class, except that it does not instance variables.

All interfaces are explicitly public and abstract. Note that interfaces are typically adjectives. e.g. Runnable, Steerable, Sortable, ...

Any class that implements an interface must implement all of the interface's methods, otherwise the class must be declared abstract.

## Implementing Interfaces

To implement an interface use the **implements** keyword, as shown above. Note that a class can implement more than one interface. If this is done, separate the interfaces with a comma.

If a class cannot implement one or more of the methods defined in the interface(s), then it must be declared abstract.

Note that abstract classes can also implement interfaces.

## Interfaces (cont'd)

- A class implements an interface
  - Indicates a capability more than an inheritance relationship
  - A class that implements an interface is said to be a subtype of its interface

```
public class Circle implements Drawable {  
    ...  
    public void draw() {  
        ...  
    }  
}  
public class Square implements Drawable { ... }  
public class Line implements Drawable { ... }
```

## Implementing an Interface

- A class can implement more than one interface:

```
public interface Drawable {
    public abstract void draw();
}

public interface Printable {
    public abstract void print();
}

public interface Radial {
    public abstract double getRadius();
}

public class Oval implements Drawable, Printable {
    ...
    public void draw() { ... }
    public void print() { ... }
}

public class Circle extends Oval implements Radial {
    ...
    public double getRadius() { ... }
}
```

- Circle is also each of the following types:  
Oval, Drawable, Printable, Radial

### Implementing Interfaces

You can build large hierarchies by combining multiple interfaces. Here we've isolated various capabilities as separate interfaces. A class that has a capability indicates this by implementing that interface.

### Extending Interfaces

The Note: example above also shows a sub-interface. Oval is a class that is both Drawable and Printable, and as a non-abstract class it must implement the methods in those interfaces.

When the Circle class extends the Oval class, it adds its own capability unique to circles, which is the existence of a radius.

## Extending Interfaces

- Shape as an interface

- Would allow concrete shapes to be a Shape, yet inherit from a more "functional" parent class

```
public interface Drawable {
    public abstract void draw();
}

public interface Printable {
    public abstract void print();
}

public interface Shape extends Drawable, Printable {
}

public class Circle extends GraphicalComponent implements Shape {
    public void draw() { ... }
    public void print() { ... }
}
```

- Circle is also each of the following types

- Drawable, Printable, Shape, GraphicalComponent

### Shape as an Interface

In this example we've implemented Shape as an interface. By doing so we're losing the little functionality that Shape had as an abstract class.

However, it does now allow our Circle to use its single inheritance to extend from a much more functional superclass.

## Polymorphism With Interfaces

- An object reference is of one or more types
  - The type may be both class types and interface types
  - If a method has an argument of an interface type, it can be passed a reference to an object of any class which implements that interface type

```
--  
Circle circle = new Circle();  
Square square = new Square();  
  
--  
drawShape(circle);  
drawShape(square);  
  
--  
void drawShape(Shape shape) {  
    shape.draw();  
}
```

### Polymorphism with Interfaces

Since classes, abstract classes, and interfaces are all considered equal Java types, we can apply the rules and benefits of polymorphism to interfaces the same way as we did with classes and abstract classes.

## Type Checking

- Interface types can be used as a tag
  - They may contain no methods
- Methods can specify their arguments as this type

```
public interface Serializable { }
```

- Methods can specify their arguments as this type

```
public class ObjectOutputStream {  
    ...  
    public void writeObject(Serializable o) { ... }  
    ...  
}
```

- A class must implement this interface to be passed to the method

```
public class CustomerRecord implements Serializable {  
    ObjectOutputStream os = ...;  
  
    os.writeObject(this);  
    ...  
}
```

### Type Checking

Using the appropriate interface type on the formal parameters of the methods guarantees that the object being passed will have the correct methods.

### Serializable Objects

Some interfaces are used simply to warn the programmer that they have to construct the class in an appropriate manner. A `Serializable` object is one which can be safely sent over a network to a remote object. It contains no methods but the type is required for the method `writeObject()`. The notion is that the programmer will have been reminded to make sure that no JVM specific information is non-transient. This concept is covered in great detail in the Server Side Java course.

## Abstract Classes vs. Interfaces

- **Abstract classes**
  - You can only inherit from one superclass
  - Abstract classes can contain concrete methods
  - Abstract classes can contain instance variables
  - Abstract classes can contain private, protected, default, and public methods and variables
- **Interfaces**
  - You can implement many interfaces at once
  - Interfaces can contain abstract methods
  - Interfaces can not contain instance variables
  - Interfaces can contain class fields (static)
  - All interface access is public
- **A class (abstract or concrete):**
  - extends zero or one other class
  - implements zero or more interfaces
- **An interface:**
  - extends zero or more interfaces

### Abstract Classes vs. Interfaces

To compare the functionality of abstract classes versus interfaces, remember that you can only inherit from one class, but you can implement as many interfaces as you want. Abstract classes, unlike interfaces can also supply implementations for some methods which access instance variables.

Remember to apply the "is a" versus "can do a" test to see if you should be using an abstract class or an interface.

## Lesson Review and Summary

- 1) Why would you implement a type contract using an interface instead of an abstract class?**
- 2) Why would you implement a type contract using an abstract class instead of an interface?**
- 3) Can an interface be written with no methods?**
- 4) Is there any difference between referencing supertypes that are interfaces vs. abstract classes?**
- 5) Can a class implement two interfaces where each of the two interfaces extend a common interface?**
- 6) Can a class extend an abstract class and not implement all of the required methods?**
- 7) What is the difference between an abstract class and an interface (with default methods)**

1. Creating a supertype as an interface allows the implementer to subclass from another class
2. Creating a supertype as an abstract class allows some instance behaviors to be written and reused
3. An interface can have zero methods
4. Classes, interfaces, and abstract classes all support the identical type model
5. Interfaces can have multiple inheritance
6. An abstract class can extend another abstract class, and is not required to implement its methods. A class can be completely implemented, yet defined as abstract so it cannot be instantiated, only subclassed
7. An Abstract class can contain instance variables, and interface cannot. Classes can only extend from one other class but can implement multiple interfaces

## **Exercise 11: Mailable - Interfaces**

**`~/StudentWork/code/Interfaces/lab-code`**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

# Lesson: Exceptions

Inheritance and Polymorphism  
Interfaces and Abstract Classes  
**Exceptions**

## Lesson Agenda

- Defining a try/catch block that allows methods that throw exceptions to be called
- Correctly implementing try/catch blocks for methods that throw multiple exceptions
- Naming the exception supertype that is not checked for by the compiler
- Defining your own application exceptions
- Correctly implementing a method that throws exceptions
- Correctly overriding a method that throws exceptions

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## What is an Exception?

- An exception is a run-time signal that some unforeseen (or exceptional) condition has occurred
  - Logic errors: invalid array access, divide by zero
  - Device errors: printer off-line, network card problem
  - System errors: hard drive full, out of memory
- Traditionally, functions would return error codes that were interpreted or ignored
- Java uses **exception handling**

### Error Checking

There are many types of errors that can crop up in production programs. Java provides a standard way of handling them. System errors, device errors, data inconsistencies, program errors, etc. can all be handled with exceptions.

Exceptions in Java are very efficient as there is no error checking if all goes well except right at the point where the error could happen. The return up the calling stack does not require error checking on the return from each method. When an exception is thrown and caught, then and only then, does the code have to deal with them.

## Exception Architecture

- Methods raise exceptions instead of returning error codes
  - An Exception object is created and thrown
  - Subtypes of Exception can contain data
- A method that may throw an exception must be called within a try/catch block

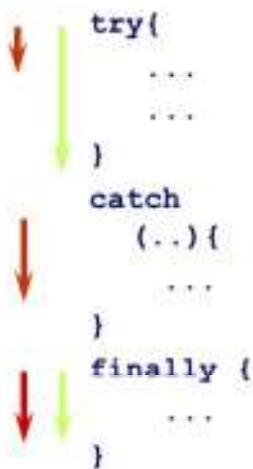
```
...  
try { // code that may throw an exception  
    n = stream1.read();  
} catch ( IOException e ) {  
    // Code to handle the exception  
}  
...
```

- Forces error conditions to be recognized and handled

### About Exceptions

When an error does happen, an **Exception** object is created and this object is thrown back to some point in the code where it is caught. Catching it requires a **try{...}** catch block to be implemented. Throwing an exception is effectively a "non-local goto". Execution jumps to the closest catch block. This jump does not involve walking up the stack call by call but throws away the existing stack back to the try catch block. The intervening stack frames are discarded.

## Handling Exceptions



The illustration shows the keywords and how they work which we explore in the next pages.  
The darker line illustrates an occurrence of an exception, the lighter line illustrates the normal flow of control.

## The Throwable Class

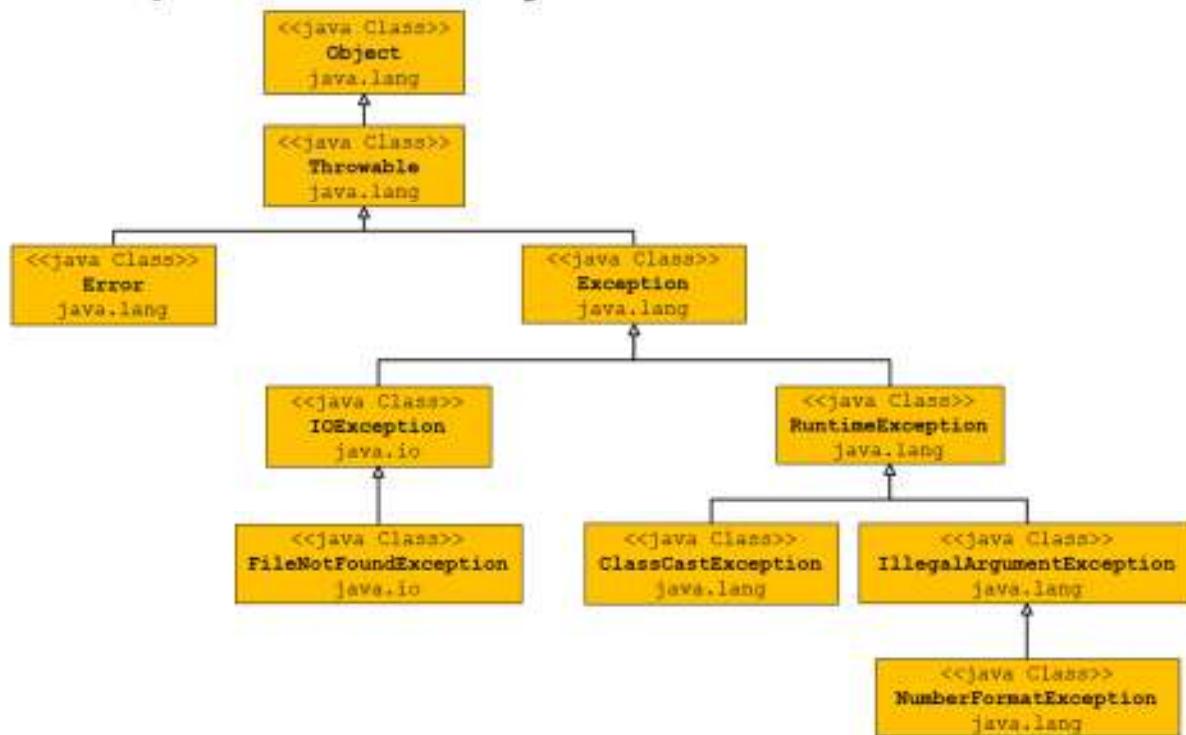
- All errors and exceptions inherit from the **Throwable** class
  - **Error**
    - ◆ Usually unrecoverable low level system problems
    - ◆ Not typically handled by the application
  - **Exception**
    - ◆ All types inherit from **Exception**
    - ◆ All can be handled by the application in some way
    - ◆ May not be recoverable, but application has opportunity to handle gracefully

### Errors vs. Exceptions

An error is an abnormal condition which is typically unrecoverable. This typically includes hardware problems. An error will usually cause the program to crash.

An exception is an abnormal condition which can at least be handled. Handling the exception may just be gracefully shutting the program down.

# Exception Hierarchy



## The try Block

- A **try** block is used to enclose a block of code that may throw an exception

```
public class Sumfile { // sum the bytes in a file
    public static void main(String[] args) {
        FileInputStream fin = null;
        int b = 0;
        int sum = 0;
        try {
            fin = new FileInputStream(args[0]);
            while((b = fin.read()) != -1) {
                sum += b;
            }
        } catch (FileNotFoundException fnfe) {/*code*/}
        System.out.println("Sum of file " + args[0] + " is " + sum);
        ...
    }
}
```

### Syntax

The syntax for catching exceptions is perfectly logical. Within the **try {}** block is the normal code. If no exceptions are thrown then this code executes normally. However, if an exception is thrown then the remaining statements are skipped and the **catch ( ) {}** block is executed.

The "non-local goto", generated when the exception is thrown, starts execution at the catch block.

In the above example, creating the **FileInputStream** could generate a host of errors that will throw exceptions. Look at the JavaDoc documentation for the constructor of the **FileInputStream** class in the package **java.io** to see the possible exceptions that could be thrown.

## The catch Block

- Exceptions are handled by the catch block
  - The catch block can be used to catch multiple exceptions
  - `getMessage()` method of Exception prints meaningful information
  - `printStackTrace()` will tell you where the exception occurred

```
    }
} catch (FileNotFoundException fnfe) {
    System.err.println("An error occurred:" + e);
    e.printStackTrace();
    return;
} finally {/*code*/}
```

### The catch Block

The catch block looks very similar to a method call. The argument is the exception that was thrown. This is a genuine bona fide Java object.

Execution continues into the catch block just as it would in a method.

Once the try block has executed normally or catch block has completed execution, the finally method will be called. The only time a finally block will not execute is if the try block it is associated with exits via `System.exit()`.

## Handling Multiple Exceptions

- Multiple types of exceptions can be caught in a single catch block
  - Exception types are separated by the | symbol

```
public void exceptionCatcher() {  
    try {  
        somethingThatCanThrowThreeExceptions();  
    } catch (ExceptionA eA) {  
        // handle ExceptionA  
    } catch (ExceptionB | ExceptionC eBC) {  
        // Java 7: handle ExceptionB and ExceptionC  
    }  
}
```

With the introduction of Java 7, multiple exception types can now be handled in a single catch block. (Before separate catch blocks had to be defined for each Exception type)

## The finally Block

- Execution may leave a try block by:
  - Completing successfully and falling through the end brace
  - Executing a return statement
  - Throwing an exception
- The finally block guarantees that a block of code is always executed
  - Unless the try block it is associated with exits via `System.exit()`

```
...  
} finally {  
    System.out.println("File access completed");  
}
```

- Used for cleaning up resources, regardless of outcome
- Never throw exceptions out of finally blocks

### The finally Block

You must be extremely careful not to allow a finally block to throw an exception. If code in your finally block can throw an exception you must catch it within the finally block.

## Automatic Closure of Resources

- Many constructs in Java must be closed to release related resources
  - JDBC constructs, I/O streams, and sockets are included
- Resources can be scoped in the try block and they will be closed automatically
  - No longer have to explicitly close the resources
  - Advisable to catch exceptions and handle them

```
try {  
    InputStream in = new FileInputStream(src);  
    OutputStream out = new FileOutputStream(dest)  
}  
{  
    // code using the streams  
} catch (FileNotFoundException ex) {  
    //Handle error condition here  
}
```

All such resources now implement the **java.lang.AutoCloseable** interface

# Full Example of Exception Handling

## ● Example of the full syntax

```
try {
    x = someMethodThatMayThrowAnException();
    y = someMethodThatMayThrowAnotherException();
}

catch (ExceptionType1 e1) {
    System.err.println("Exception 1: " + e1);
}

catch (ExceptionType2 e2) {
    System.err.println("Exception 2: " + e2);
}

finally {
    System.out.println("Thanks for calling me, however it worked");
}
```

### A Fuller Example

The **try{}, catch{}, finally{}** syntax is described in the slide above. Note that you can have any number of catch statements. Also remember that exceptions are objects, and as such a catch statement can catch an exception or any of its subclasses.

## Generalized vs. Specialized Exceptions

- A catch block can specify a supertype exception
- Catch block sequence must start with subtypes

```
public class MyException_A extends Exception { ... }
public class MyException_AB extends MyException_A { ... }
public class MyException_AC extends MyException_A { ... }
public class MyException_B extends Exception { ... }
```

```
try {
    someMethod();
} catch (MyException_AB eab) { // start with subtype first
    ...
} catch (MyException_A ea) {
    // will catch MyException_A and catch MyException_AC
} catch (MyException_B eb) { // okay, since not subtype
    ...
}
```

### Catching Specialized Exceptions

All exceptions are either **java.lang.Exception** or more specialized versions of it. If you have many catch blocks for different exceptions as shown above, how are they treated?

The rule is as follows: the catch blocks will be examined in top down sequential order until a type match is found and that will be the catch block that is executed.

Be careful not to put **Exception** as the top parameter for the catch block. As all exceptions are subclasses of **Exception**, it would always be called. As a rule of thumb, make sure that the exceptions are ordered in the most specialized to the most general. **Exception** will always be the most general.

## Overriding Methods

- A method must declare in its signature all of the exceptions that it may throw

```
public void myMethod() throws FoobarException, ABCException {  
    ...  
}
```

- You can override a method that throws an exception
- The overriding method can only declare to throw the exceptions, or subclasses of the exceptions, which are declared in the overridden method

```
// the overriding method in the subclass  
public void myMethod()  
    throws FoobarException, ABCException, FoobarSubtypeException {  
    ...  
}
```

### Throwing Exceptions in Overriding Methods

If you override a method that throws exceptions, you do not need to throw all of the exceptions which were declared in the overridden method. However, you cannot add exceptions to the overriding method which were not in the overridden method.

It is also allowable for the overridden method to throw no exceptions.

Remember that exceptions are objects, so the overriding method can throw exceptions which are subclasses of the exceptions in the overridden method.

## Creating Your Own Exceptions

- Since **Exception** is a class, we can subclass it or any of its subclasses to create our own exceptions
- Standard exception classes have two constructors
  - Default constructor
  - Constructor that contains a detailed message about problem

### Creating Exceptions

Since **Exception** is a class, we are able to subclass it to create our own exceptions to throw. This is usually done if there is an error (or errors) that you wish to inform the calling code about.

Note that it is not a common practice to subclass neither **Error** nor **RuntimeError**.

## Throwing Exceptions

- You can explicitly throw exceptions by using the **throw** keyword
- You can throw predefined exceptions, or ones that you have made
- Since an exception is an object, you must instantiate it before throwing

```
public void myMethod() throws MyException {  
    if (somethingWentWrong) {  
        throw new MyException("Something went wrong!");  
    }  
    ...  
}
```

### Throwable Objects

You can use the **throw** keyword to throw your own exceptions. These exceptions may be standard system exceptions, or ones which you have created.

The main thing to remember when throwing exceptions is that they are **Throwable** objects. This means that you must instantiate an **Exception** object prior to throwing it.

Unchecked exceptions (subclasses of **java.lang.RuntimeException**) do not need the **throws** clause on the method.

## Re-throwing an Exception

- An exception handler may (for whatever reason) choose to re-throw the exception that it just caught

```
public void myMethod() throws FoobarException {
    try {
        myOtherMethod(); // also throws FoobarException
    } catch (FoobarException fbe) {
        System.err.println("Foobar in myOtherMethod");
        throw fbe;
    }
}
```



### Example

The above example shows how the exception is simply re-thrown and then caught in an outer block.

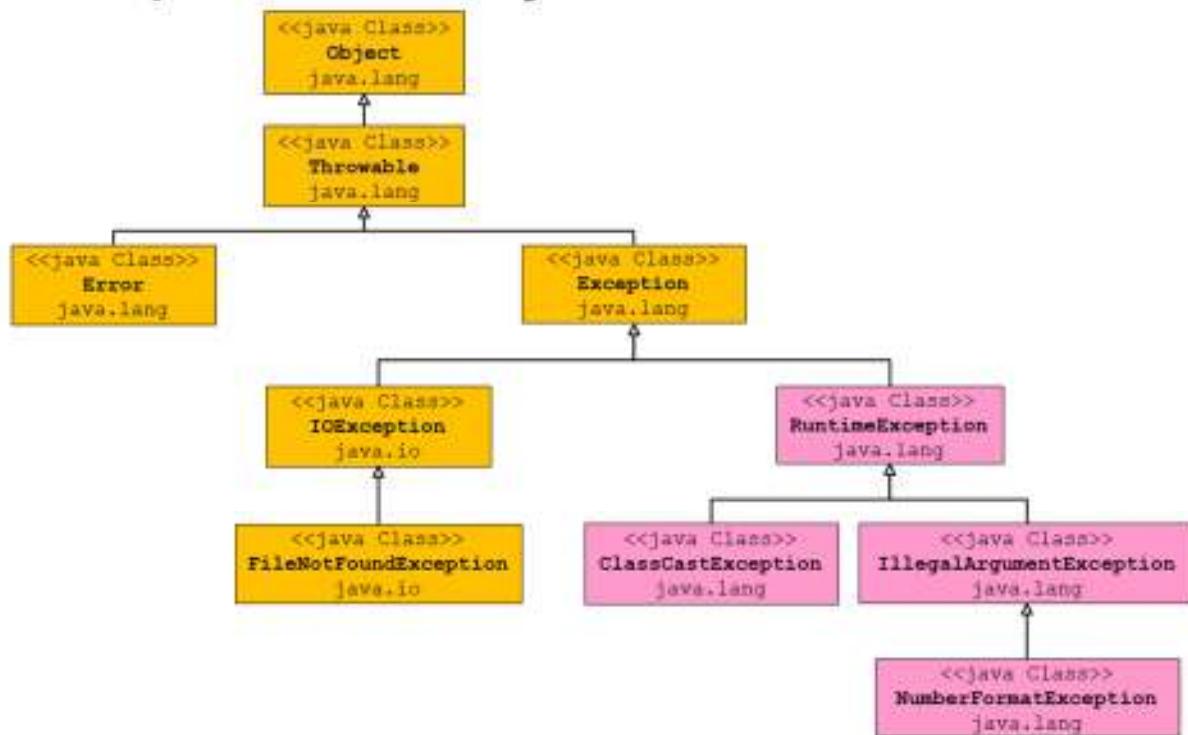
## Checked vs. Unchecked Exceptions

- **Unchecked exceptions**
  - Represent program defects
  - All are subtypes of `RuntimeException`
  - Methods are not required to handle or declare
  - Examples include: `IllegalArgumentException`, `NullPointerException`, etc.
- **Checked Exceptions**
  - Represent defects that are outside immediate control of application
  - Method is required to handle (catch) the exception or declare that it could be thrown
  - Application-specific errors are part of this category

### Checked vs. Unchecked Exceptions

It is expected that unchecked exceptions would be fixed by the programmer. Checked exceptions are expected to be caused by something external to the application, like user input. It is perfectly acceptable to handle unchecked exceptions, and is expected if it would be caused by the user instead of the logic of the program.

# Exception Hierarchy



## Checked vs. Unchecked Exceptions

- Checked exception can be caught at the location where thrown

```
public void someMethod() {  
    try {  
        anExceptionalMethod();  
    } catch (SomeException se) {  
        -  
    }  
}
```

- Or method can declare that the exception may travel up to caller

```
public void someMethod() throws SomeException {  
    anExceptionalMethod();  
}
```

- Unchecked exceptions do not need to be handled

```
public void someMethod() {  
    mayThrowRuntimeException();  
}
```

## Lesson Review and Summary

- 1) Why is an exception better than an error return value on a method?**
- 2) Can custom exceptions be written?**
- 3) What are the two general categories of exceptions?**
- 4) Can runtime exceptions be caught in a try/catch?**

1. Exceptions force the application to handle error conditions
2. Customized exceptions are classes that extend the Exception class
3. Application exceptions can be written
  - To indicate specific application errors
  - To carry additional data
4. Runtime exceptions can be caught, but it is not required by the compiler

## Exercise 12: Exceptions

`~/StudentWork/code/Exceptions/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Session: Java Developer's Toolbox**

**Utility Classes  
Enumerations and Static Imports  
Introduction to Annotations**

# Lesson: Utility Classes

## **Utility Classes**

Enumerations and Static Imports

Introduction to Annotations

## Objectives: Utility Classes

This lesson covers how to use various commonly used utilities classes. Specifically, it covers:

- Converting String representations of primitive numbers into their corresponding wrapper
- Converting String representations of primitive numbers into their primitive types
- Using the Date class to get the current time and date
- Using the Date class to determine if one Date is earlier or later than another

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## Wrapper Classes

- The following extend from `java.lang.Number`
  - `Integer`, `Long`, `Short`, `Double`, `Float`, `Byte`,  
`BigDecimal`, `BigInteger`, `AtomicInteger`, and  
`AtomicLong`
- Other wrapper classes include
  - `Boolean`, `Character`, `Void`, `AtomicBoolean`, and  
`AtomicReference`
  - The "Atomic" wrappers were added in Java 5 and are used  
in concurrent operations
  - Contained in `java.util.concurrent.atomic`

### About Wrapper Classes

There are many times when a primitive is inadequate. The Java language provides wrapper classes that wrap the primitive up into a real object. It still holds the primitive within it but now has all the characteristics of an object plus a host of methods that can act on the primitive.

Review the JavaDoc documentation for a class such `java.lang.Integer`.

Each primitive type has a wrapper class: **byte**, **char**, **short**, **int**, **long**, **float**, **double** and **boolean**.

## The Number Class

- Base class that allows any type to be converted to any primitive type

```
public abstract class Number ... {  
    public int intValue();  
    public long longValue();  
    public short shortValue();  
    public float floatValue();  
    public double doubleValue();  
    public byte byteValue();  
}
```

- All numeric primitive wrappers extend this type

### The Number Class

This is the superclass to the other wrapper classes.

Notice that every wrapper class has the ability to convert content into any of the primitive types. So for example, a wrapper that contains a **float** value can be obtained as an **int**.

## Numbers and Strings

- Each primitive wrapper can be constructed from a String

```
public Integer(String intStr) throws NumberFormatException {  
    —  
}
```

- Also provide static methods to convert from text to the primitive type

```
Integer: public static int parseInt(String str);  
Long: public static long parseLong(String str);  
Double: public static double parseDouble(String str);  
etc.
```

- And to the primitive wrapper

```
Integer: public static Integer valueOf(String str);  
Long: public static long valueOf(String str);  
etc.
```

### Flexibility in Parsing and Conversion

**parseInt()** is another very useful method. Anytime you need to convert from a string into the appropriate type, use the **parseX()** where X is the appropriate type. These are static methods so they can be invoked without having to instantiate a wrapper class.

All the parse methods throw a **NumberFormatException** if the string does not hold a valid number. This is not a checked exception and the code will compile without it. However, due to the potential for a number formatting exceptions, you should always perform these operations within a try-catch statement to more gracefully handle such errors.

## Autoboxing/Unboxing

- The autoboxing and unboxing functionality is meant to remove the explicit wrapping and unwrapping of primitive values in code

- Before autoboxing:

```
List myList = new ArrayList();
myList.add(new Integer(42));
```

- After autoboxing:

```
List myList = new ArrayList();
myList.add(42);
```

```
Integer intObject = new Integer(27);
int i = intObject.intValue();
```

```
Integer intObject = 27;
int i = intObject;
```

- You can treat primitive values and primitive wrappers objects the same way...most of the time

## Autoboxing/Unboxing Issues

- There are a few things to bear in mind when using autoboxing/unboxing
  - Objects are not primitives
    - ◆ If you try to unbox a null you will get a `NullPointerException` at runtime
  - There is a performance penalty due to the creation and garbage collection of objects
  - It can be fast, but should not be used in places where performance is critical
- Use autoboxing/unboxing wisely
  - Use it when an object is required, but only a primitive is available
  - Do not use it in calculation or performance-sensitive code

## BigDecimal

- Provides precise control over operations with decimal points
- Extends Number

```
public BigDecimal setScale(int scale, int round);
public BigDecimal add(BigDecimal bd);
public BigDecimal subtract(BigDecimal bd);
public BigDecimal multiply(BigDecimal bd);
public BigDecimal divide(BigDecimal bd, int round);
```

- Note that arithmetic operations do not affect the original object

### Improved Floating Point Handling

Anyone who has tried to do precise arithmetic with floating point will realize the value of **BigDecimal**. You can precisely calculate any precision you desire. **BigDecimal** is especially useful with SQL databases.

## Decimal Formatting

- You can apply custom format masks using `DecimalFormat`

```
DecimalFormat myFormatter = new DecimalFormat("###,###.###");  
String output = myFormatter.format(value);
```

- The `format()` method can take
  - `double` and `long`
  - `float` upcast
  - Primitive wrappers such as `Float`

### Floating Point Formatting

This class allows you to specify a template for displaying string representations of floating point numbers. Once the formatter has been initialized with a template it can be used to format floating point values and returned as a properly formatted string.

## The Date Class

- Allows you to construct a date that is accurate to the millisecond
- Getting the current time and date

```
Date date = new Date();
```
- Comparing dates

```
boolean after(Date otherDate);  
boolean before(Date otherDate);
```
- Calendar
  - Abstract base class to convert a Date into numbers (such as year, month)
  - It knows about a specific calendar (Gregorian, Lunar type calendar)
- When importing the Date class, ensure you use `java.util.Date` and not `java.sql.Date`

### The Date Class

This class represents a date and time in an easy to use format. By constructing a **Date** with no values, it will become initialized to the current time and date.

## Lesson Review and Summary

- 1) What is the primary purpose of the primitive wrapper classes?
- 2) Do these classes support conversion to and from String?
- 3) What is the default value of an instance of Date?
- 4) Can Date instances be used to perform time and date comparisons?

The wrapper classes allow primitive values to be passed into methods that require object types

All of the primitive wrappers support `String` parsing and `toString`

The current time and date can be determined by creating instance of `Date` with no input values

The `Date` class provides methods to check if one date is earlier or later than another

## Exercise 13: Using Primitive Wrappers

`~/StudentWork/code/PrimitiveWrappers/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Lesson: Enumerations and Static Imports**

Utility Classes  
**Enumerations and Static Imports**  
Introduction to Annotations

## Lesson Agenda

- **Rationale for the creation of type safe enumerations**
- **Enumeration syntax**
- **Enumerations as a better class type**
- **Using static imports**

## Rationale for Enumerations

- Without enumerations there are two ways to mimic enumeration behavior
  - The Constant Interface Pattern
  - ◆ Interface with constants based on primitives

```
public interface Meal {  
    public static final int BREAKFAST = 0;  
    public static final int LUNCH = 1;  
    public static final int DINNER = 2;  
}
```

- The Typesafe Enum Pattern

- ◆ Class with a private constructor and public static final variables to define valid constant objects

```
public class Meal {  
    public static final Meal BREAKFAST = new Meal("Breakfast");  
    public static final Meal LUNCH = new Meal("Lunch");  
    public static final Meal DINNER = new Meal("Dinner");  
    private String _name;  
  
    private Meal(String name) { _name = name; }  
    public String toString() { return _name.toString(); }  
}
```

The Constant Interface Pattern's biggest problem was type safety. Because the constants were primitives it was very difficult to guarantee their use.

The Typesafe Enum Pattern did not suffer from the problems of the Constant Interface Pattern because the constants were ultimately a concrete class type. However, they could be a lot of work and were still considered brittle. Also, they were difficult to use in switch statements.

## Enumeration Syntax

- Java SE defines an enumeration syntax that looks very similar to C/C++

```
[scope] enum [Enumeration Name] ( [value], [...] )
```

- Java enums are full classes
- A list of constants would be defined as:

```
public enum Meal { BREAKFAST, LUNCH, DINNER }
```

In Java 5 the enum keyword was introduced. Besides classes and interfaces, developers can now also develop class files containing enumeration definitions.

Java enumerations are full implementations of classes, as we will see during this lesson.

The example shows the simplest form of Java enumerations, a comma separated list of enumeration values.

## Implementing Enums

- All Java enums implicitly extend `java.lang.Enum`
  - Enums can **not** extend other classes (or enums)
  - Enums can implement one or more interfaces
- Has predefined convenience methods

```
Meal[] values = Meal.values(); //Returns all values  
Meal lunch = Meal.valueOf("LUNCH"); //String must match enum value
```

- Can **not** be instantiated with the new construct
- Can override `toString` method
- Can define additional methods
- Consists of constants that are implicitly public static final

All Java enumerations implicitly extend the `java.lang.Enum` class. As a result a java enum can not extend other classes or enumertions. However, it is possible for an enum to implement one or more interfaces.

Every enumeration has predefined convenience methods. The `values()` method returns an array of all the valid enum values. By using the `valueOf` method, a string value van be converted into an enum value. The string value must be exactly the name of the enum value, otherwise an exception is thrown.

Enumerations can not be instantiated using the new keyword, however it is possible to define additional methods in the enum class of override methods (like `toString()`)

## Enumeration Example

```
public enum CoffeeSize {SMALL(10, "Small"), MEDIUM(20, "Medium"),
LARGE(30, "Large"), EXTRALARGE(40, "Extra Large");

private int ounces;
private String label;

private CoffeeSize(int val, String newLabel) {
    ounces = val;
    label = newLabel;
}

public int getSize() { return ounces; }
public String getLabel() { return label; }
public String toString() {
    return String.format("%s, %d ounces", label, ounces);
}

String s = CoffeeSize.EXTRALARGE.toString(); //Extra Large, 40 ounces
String name = CoffeeSize.EXTRALARGE.name(); //EXTRALARGE
String label = CoffeeSize.EXTRALARGE.getLabel(); //Extra Large
int size = CoffeeSize.EXTRALARGE.getSize(); //40
//iterate over all CoffeeSize values
for (CoffeeSize coffeeSize : CoffeeSize.values()) {
    ...
}
```

Annotations:

- Instance variables: `ounces`, `label`
- Constructor (must be private or default): `private CoffeeSize(int val, String newLabel)`
- Instance methods: `getSize()`, `getLabel()`
- Override `toString()`

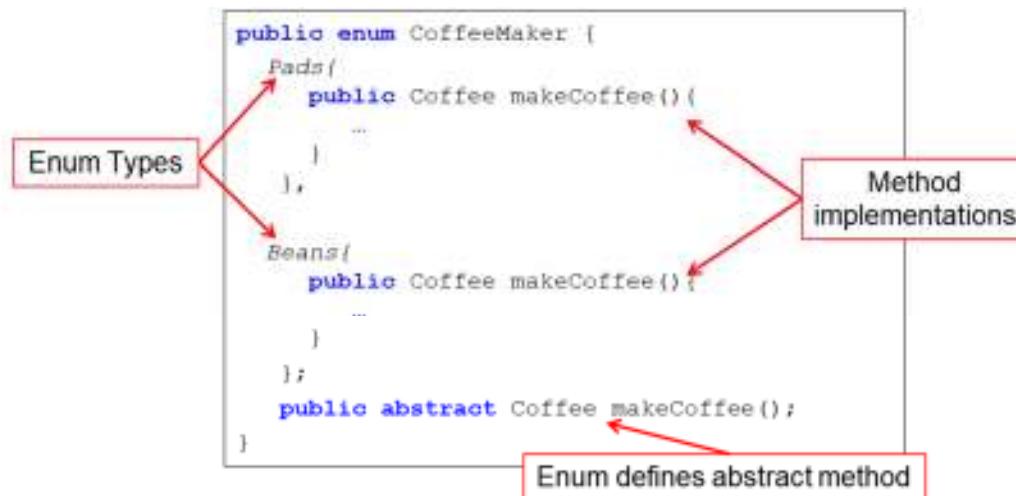
The example shows a more complex enumeration.

An enum can have a constructor that accepts parameters, however the access modifier must be either private or default (no access modifier). When defining possible enum values, each value must be invoking the constructor of this enum. In other words, all parameter values of the constructor must be specified when defining enum values.

The constructor parameters become instance variables of the enum, which can be read using the getter methods, as shown by the second code example.

# Enumeration and Abstract Methods

- Abstract methods can be defined in an enumeration.
  - Must be implemented by all of the enumerated objects



It is even possible to define abstract methods within an enum. However, each enum value that is defined must provide an implementation of this abstract method.

## Enumerations as a Better Class Type

- **Encapsulation**
  - Enumerations can have data and behavior added to them
  - Constructors and methods are supported
- **Inheritance**
  - Enumerations can implement interfaces, but cannot extend concrete classes
  - All enumerations are **Serializable** and **Comparable** by default
  - Enumerations cannot inherit from other enumerations
- **Polymorphism**
  - Enumerations can implement abstract methods either from interfaces they are implementing or abstract methods they define

Enumerations are a better class type than the Constant Interface Pattern and the Type Safe enum pattern shown earlier.

Enumerations can contain constructors, state and behavior, just like other classes. (with the exception that the constructor of an enum must be declared as private or default)

Enumerations can implement interfaces, but cannot extend other classes (because they implicitly extend **java.lang.Enum**). By default all enumeration already implement the **Serializable** and **Comparable** interfaces.

## When You Should Use Enumerations

- **Enumerations should be used when a fixed set of constants is needed**
  - Days of the week
  - Suits/values in a deck of cards
  - Command line options
  - Strings for use in menus

Enumeration should be used whenever a fixed set of constants is needed within the application.

## Using Enumeration in Switch Statements

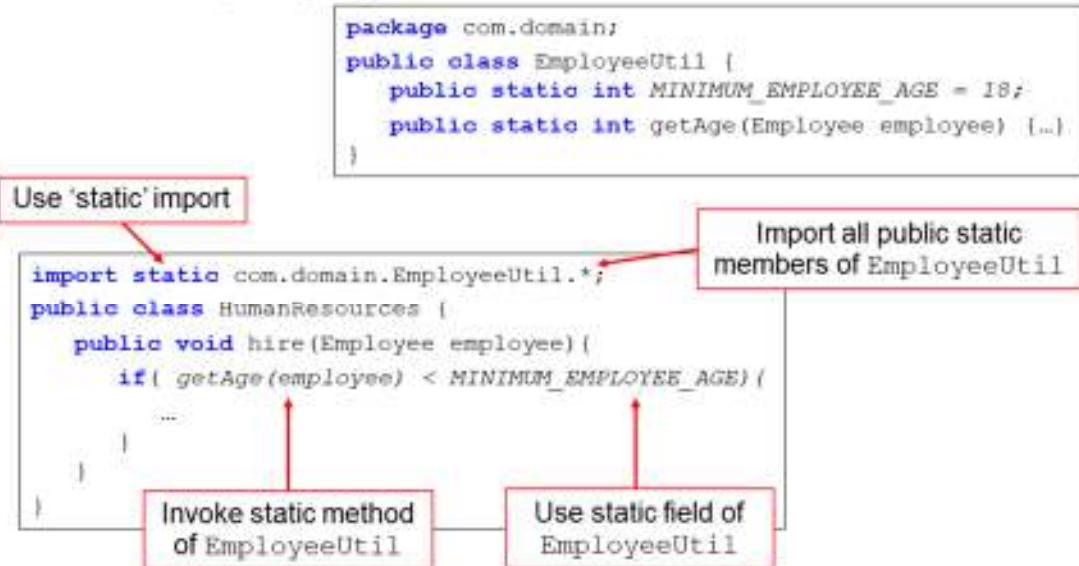
- Enumerations can be used in switch statements

```
CoffeeSize size = CoffeeSize.LARGE;
switch(size) {
    case SMALL:
    case MEDIUM:
        tryToSellLargeCoffee();
        break;
    case LARGE:
        serveCoffee();
        break;
    case EXTRALARGE:
        makeMoreCoffee();
}
```

Enumerations can also be used within switch statements. As the example shows, when a switch is defined on a variable of enum type, all case statements must be defined using one of the valid values of the enum type on which the switch takes place.

## Static Imports

- Static imports provide direct access to public static fields and methods of other classes.
  - Without specifying the class in which member is defined.



Until the introduction of static imports, referencing static fields or methods of another class could only be accomplished by referencing the field or method using the Classname of the class on which the field or method was defined.

In order to make code better readable, static imports were introduced in Java 5. Just like importing other classes in order to make sure that developers would not have to reference other classes using their fully qualified name (name including package-name), static imports can be used to allow developers to reference static fields and methods without having to define the fully qualified class name of the class on which the static field or method is defined.

Static fields or methods can be imported using '**import static**', followed by the fully qualified name of the class (on which the field or method is defined), followed by a \* (to import all static members of a class)

As the example shows, once a static import has been added, the static members of the other class can now be used without their 'prefix'

## Static Imports - Fields

- Individual public static fields can be imported
  - Reduces the chance of naming conflicts
  - Makes code easier to read

Import MINIMUM\_EMPLOYEE\_AGE  
field only

```
import static com.domain.EmployeeUtil.MINIMUM_EMPLOYEE_AGE;
public class HumanResources {
    public void hire(Employee employee) {
        if( getAge(employee) < MINIMUM_EMPLOYEE_AGE ) {
            ...
        }
    }
}
```

Method not recognized

Instead of importing all static members of a class (using the \* notation), developers can choose to import only one specific static field of a class.

In the example, only the static field is imported, as a result it is no longer possible to invoke the static **getAge(...)** method.

## Static Imports - Methods

- Individual public static methods can be imported

```
import static com.domain.EmployeeUtil.getAge;
public class HumanResources {
    public void hire(Employee employee) {
        if( getAge(employee) < MINIMUM_EMPLOYEE_AGE ) {
            ...
        }
    }
}
```

When only a static method is to be imported, the import should contain only the method name (without parenthesis).

In the example, only the static method is imported. As a result, the static field is no longer recognized

## When (Not) to Use Static Imports

- Using lots of static methods can make your code procedural
- Static imports make your code harder to read
  - Not immediately clear where method or field is defined
- Use static imports
  - Instead of making local copies of constants
  - To minimize repetition of the prefix for a small set of constants or static methods

```
import static java.util.Calendar.*;
import java.util.Calendar;
public class CalendarUtil {
    public void doSomething() {
        Calendar now = Calendar.getInstance();
        int day = now.get(DAY_OF_MONTH);
        int month = now.get(MONTH);
        int year = now.get(YEAR);
    }
}
```

In general, having a large number of static methods is an indication that the code is becoming more procedural instead of object oriented. So, having a lot of static imports is often a bad sign.

Also, code tends to become more difficult to read when methods and variables are referenced that are not declared in the enclosing class. You should therefore be very careful when using a lot of static imports in your code.

The code example shows the use of the `Calendar` class. The implementation of the `Calendar` class relies heavily on the static fields that are defined within the `Calendar` class. This is one example where using static imports of these fields might make your code easier to read.

## Lesson Review and Summary

- **What are the two Java patterns that enumerations were meant to address?**
- **What restrictions on the use of constructors and methods apply to enumerations?**
- **Are enumerations concrete classes or abstract classes?**
- **Can abstract methods be defined in an enumeration?**
- **What does the static values() method return?**
- **Can private data be defined in an enumeration?**
- **What interface types do enumerations implicitly implement?**
- **How do you define and use a static import?**

The two Java patterns that enumerations were meant to address are the Constant Interface Pattern (really and anti-pattern) and the Typesafe

Enumeration Pattern

Constructors and methods can be used within enumerations in the same way as a normal class except that the constructors cannot be declared public or protected

Enumerations are concrete classes

Abstract methods can be defined in an enumeration, but they must be implemented by all of the enumerated objects

The static values() method return an array of enums from the current enumeration type

Private data can be defined in an enumeration

Enumerations implicitly implement Serializable and Comparable

Static imports are defined using the static keyword after the import keyword in the import section outside of the class/interface definition

## Exercise 14: Enumerations

`~/StudentWork/code/Enumerations/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Introduction to Annotations

Utility Classes  
Enumerations and Static Imports  
**Introduction to Annotations**

## Objectives: Intro to Annotations

**This lesson covers how to use annotations in Java.  
Specifically, it covers:**

- **Discussing how annotations work in Java**
- **Understanding what is required to work with Java's annotations**
- **Using annotations**
- **Other technologies that are using annotations**

## Annotations Overview

- Provide metadata information about a class
  - Not part of the business logic
  - Have no (direct) effect on the code
- Provide information
  - To the compiler
    - ◆ Detect errors or suppress warnings
  - For compile-time and deployment-time processing
    - ◆ Tooling can process annotations to generate additional resources
  - Runtime processing
    - ◆ Frameworks (e.g. JPA) process annotations to add functionality to application

```
@Entity  
@Table(schema="DVD", name="MEMBERS")  
public class Member { ... }
```

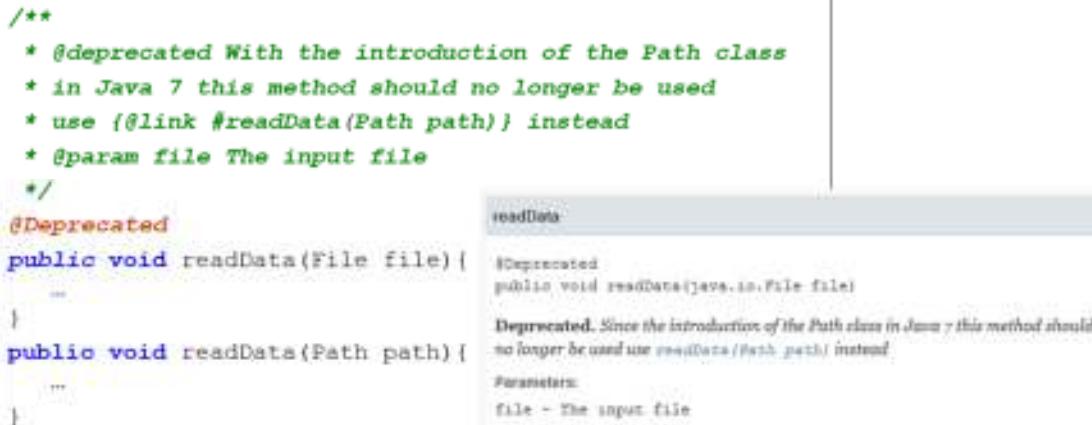
## Annotations Overview (cont'd)

- Consist of an at-sign (@) followed by the type of annotation
  - Optionally, a set of parentheses containing a comma-separated list of name/value pairs
- Java defines a small set of standard annotations
  - @Deprecated, @Override, @SuppressWarnings, ...
- API's might define technology-related annotations
  - JUnit, Servlets, JPA, JAXB, JAX-WS, JAX-RS, ...
- Custom annotations can be created
- By convention, annotations precede other modifier such as public, static or final

## @Deprecated

- **@Deprecated indicates that marked class, method or variable should not be used**
  - Compiler will generate warning when used anyway
- **Deprecated resources should be documented using Javadoc's @deprecated tag**

```
/**  
 * @deprecated With the introduction of the Path class  
 * in Java 7 this method should no longer be used  
 * use {@link #readData(Path path)} instead  
 * @param file The input file  
 */  
@Deprecated  
public void readData(File file){  
    ...  
}  
public void readData(Path path){  
    ...  
}
```



A screenshot of Java code showing a tooltip for a deprecated method. The code includes two methods: one taking a File parameter and another taking a Path parameter. The File-based method is annotated with @Deprecated. A tooltip for the File-based method is displayed, stating: 'Deprecated. Since the introduction of the Path class in Java 7 this method should no longer be used use readData(Path path) instead'. It also shows the parameter 'file - The input file'.

## @Override

- **@Override indicates to compiler that method is meant to override method in superclass**
  - Compiler will generate error when method does not override method in super class
- **Since Java 6, annotations can also be used to annotate implementation of methods defined by interface**

```
public class EmployeeDAOImpl implements EmployeeDAO{  
    @Override  
    public Employee getEmployee(int empID) {  
        ...  
    }  
}  
  
public interface EmployeeDAO {  
    Employee getEmployee(int empID);  
}
```

- **Use of annotation is optional**
  - Use is recommended to prevent errors

## @SuppressWarnings

- **@SuppressWarnings informs compiler to suppress warnings specific warnings**
- **Each compiler warning belongs to specific category**
  - Supported categories differs per compiler implementation
- **Category is defined as value of annotation**

```
@SuppressWarnings("deprecation")
public void process() {
    EmployeeService service = new EmployeeService();
    service.readData(new File(""));
}
```

- **Or as array of categories**

```
@SuppressWarnings({"deprecation", "rawtypes", "unused"})
public void process() {
    EmployeeService service = new EmployeeService();
    service.readData(new File(""));
    List l = new ArrayList<>();
}
```

“unchecked” is the only warning mentioned in the Java Language Specification. Each compiler has their own set of warnings, as do IDEs. In Oracle’s Java compiler, you can find out the list by using the command **javac -X**. The list also changes between versions of Java.

## @SuppressWarnings (cont'd)

- Or suppressing all warnings

```
@SuppressWarnings("all")
public void process() {
    EmployeeService service = new EmployeeService();
    service.readData(new File(""));
    List l = new ArrayList<>();
}
```

- Annotation can also be added to class

- Applies to all elements within class

```
@SuppressWarnings({"serial", "rawtypes"})
public class Employee implements Serializable{
    ...
}
```

## @FunctionalInterface (Java 8)

- **@FunctionalInterface indicates that the `interface` is intended to be functional interface**
- **Must follow rules defined by language specification**
  - Has exactly one abstract method
    - ◆ Not counting abstract methods that override methods defined in `Object` class

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {
        return t ->
            after.accept(t);
    }
}
```

- **Informational annotation**
  - All interfaces that meet definition of functional interface are implicitly functional interfaces
    - ◆ Can be created using lambda expressions

## Meta-annotations

- **Meta-annotations add information to other annotations**
- **@Retention defines scope of annotation**
  - **Value is defined by RetentionPolicy enum**
    - ◆ SOURCE : Only at source level (ignored by the compiler)
    - ◆ CLASS : Retained by the compiler (ignored by JVM)
    - ◆ RUNTIME : Can be used at runtime
- **@Target: Defines where the annotation may be used**
  - **Value is defined by ElementType enum**
    - ◆ ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE
- **@Documented**
  - **Indicates that annotation should be included in Javadoc**
    - ◆ By default, annotations are not included in Javadoc.

## Meta-annotations (cont'd)

- **@Inherited**
  - Indicates that annotation is inherited by subclasses
    - ◆ By default annotations are not inherited
- **@Repeatable (Java 8)**
  - Indicates that annotation can be applied more than once to the same declaration
    - ◆ By default an annotation can only be applied to an element once

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Sample {  
}
```

# Writing a Custom Annotation

- An annotation is defined using the `@interface` type

```
public @interface Development { }
```

- Using metadata annotations to define scope and usage

```
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Development { }
```

- Optionally, other 'configuration'

```
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Inherited  
@Documented  
public @interface Development { }
```

- Annotation can now be added to classes and methods

```
@Development  
public class EmployeeService {  
    @Development  
    public void readData(Path path) { ... }  
}
```

## Annotation Members

- Annotations might contain members
  - Defined as method definition within the annotation

```
public @interface Development {  
    String author();  
    double version();  
    Status status(); //enum type  
}
```

- Attributes of the annotation when used

```
@Development{author="Fred", version=1.0, status=Status.TEST}  
public class EmployeeService {  
    --  
}
```

- Only a restrictive set of member types can be used
  - Primitive types, String, Class, Enum, another annotation
  - An array of any of the these types

## Default Member Values

- Default values can be defined for annotation members

```
Public @interface Development {  
    String author();  
    double version() default 1.0;  
    Status status() default Status.TEST;  
}
```

- Members containing a default value do not have to be defined when annotation is used
  - All others are required

```
@Development(author="Fred")  
public class EmployeeService {  
    "  
}
```

## Java Reflection API

- **The Java Reflection API allows for runtime examination and modification of applications**
  - Obtaining information about all aspects of a class
    - ◆ Available methods
    - ◆ Method parameter types
    - ◆ Annotations defined within class
    - ◆ ...
- **Use reflection with caution**
  - Invoking methods using reflection is slower than direct method invocations
    - ◆ Compiler cannot optimize code
  - Security restrictions apply
    - ◆ Security manager might limit use of Reflection API
  - Internals of class are exposed
    - ◆ Private methods and fields can be accessed using reflection

## Obtaining Annotation Information

- The Reflection API can be used to obtain annotation information at runtime
  - Assuming annotation has @Retention set to RUNTIME
- Annotations can be obtained from Class class

```
EmployeeService service = ...
Class<?> clazz = service.getClass();
if(clazz.isAnnotationPresent(Development.class)) {
    Development ann = clazz.getDeclaredAnnotation(Development.class);
    ...
}
```

- The Method class

```
Method[] methods = clazz.getMethods();
for(Method method:methods) {
    method.getDeclaredAnnotation(Development.class);
    ...
}
```

- The Constructor, Field and Parameter classes

- ◆ Parameter class was introduced in Java 8

## Obtaining Annotation Information (cont'd)

- Member information can be obtained from annotation

```
Development ann = clazz.getDeclaredAnnotation(Development.class);
String author = ann.author();
double version = ann.version();
```

- Attributes not set by developer return the default value

## Annotations Applied

- The Java Persistence API is based on annotations
  - Just one of the many examples
- Annotations within class define mapping between instance and database
  - Java reflection is used to detect annotations and apply desired mappings
- Defines variety of aspects of JPA
  - Persistence declarations
  - Mapping
  - Relationships/Associations
  - Named Queries
  - EJB 3.0 persistence

```
@Entity(name="employee")
@Table(name="T001_EMPLOYEE")
public class Employee {
    @Id
    @GeneratedValue
    private int id;
    @Column(name="FULLNAME")
    private String name;
    private int salary;
    @Transient
    private String password;
    //getter and setter
    //equals, hashCode
}
```

## Annotations Applied (Java EE)

- Java EE relies on annotations
  - Instead of (or in addition to) deployment descriptors
- Defining runtime requirements of components
  - URL mapping to Java resources
  - Dependency injection
  - Transactional and security requirements
  - ...

```
@WebServlet("/AddEmployee")
public class EmployeeServlet extends HttpServlet {
    @Inject
    private EmployeeService service;

    protected void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        ...
    }
}
```

## Annotations Applied (JUnit)

- Third party libraries might rely on annotations for configuration
  - For example, JUnit

```
@RunWith(value = Parameterized.class)
public class ServiceTest {
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { "A", "B" }, { "A", "C" } };
        return Arrays.asList(data);
    }
    @AfterClass
    public static void tearDownAfterClass() throws Exception { ... }
    @Before
    public void setUp() throws Exception { ... }
    @Test
    public void test() { ... }
}
```

## Lesson Review and Summary

- 1) What is the purpose of an annotation?**
- 2) Name some of the standard annotations.**
- 3) Can information defined by annotations be read at runtime?**

- 1) Annotations provide metadata information about a class
- 2) Override, SuppressWarning, Deprecated
- 3) Only when RetentionType of the annotation is set to Runtime

## Exercise 15: Annotations

`~/StudentWork/code/Annotations/lab-code`

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## **Session: Collections and Generics**

### **Introduction to Generics Collections**

# **Lesson: Introduction to Generics**

## **Introduction to Generics**

Collections

## Lesson Agenda

- **Using generics**
- **Using generics as arguments in methods**
- **Using wildcards**

# Introduction to Generics

- **Introduced in Java 5**
  - Provides **compile-time** type checking
  - Remove risk of `ClassCastException`
- **Commonly used when working with collections of objects**
  - Java Collection framework (covered later) was re-written to use generics

Generics was introduced in Java 5 to provide developers with more compile time type information. It helps us to understand better how our code deals with types.

Throughout this lesson we will look at the use of generics when handling lists (collections) of object references. Java provides a full API for working with collections of objects (which will be covered in a later lesson), but for the moment we will look at example implementations of this API to explain how generics will provide the developer and the compiler with more type information.

It is important to try to keep in mind the distinction between what type information you have at compile-time (which is limited, even with generics) and what type information you have at runtime (which is the full type information of the instances).

By using generics, the compiler is capable of throwing exceptions which, without generics, would only appear at runtime.

## Writing a List

- When developing a class that can maintain a list of any object type
  - Methods would accept and return **Object** types

```
public class List{  
    public void add(Object o) {...}  
    public Object get(int index) {...}  
    public void remove(Object o) {...}  
    public int size() {...}  
}
```

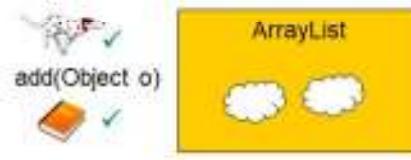
When you would need to develop a class that is capable of maintaining a list of object types, the methods you define within the class would need to accept and return **Object** references in order to make sure the class can be used for different object types.

(Keep in mind that in real projects you would (most likely) not implement this class yourself, but use one of the Collection classes instead)

## Using the List Class

- When creating an instance of **List** any object type can be added to the list

```
List list = new List();  
  
Dog dog = new Dog();  
Book book = new Book();  
list.add(dog);  
list.add(book);
```



- How often would you create a list of unrelated object types?
  - More likely you would make a list of just Dog types

When creating an instance of the **CustomList** class, you can now add any object type to this list. In the example above it is possible to add both **Dog** and **Book** instances to the **List**.

But how often would you do this? In most cases it doesn't make sense to add any type of object to a list. It's far more likely that you need to create a **List** just containing **Dog** references and a separate **List** of **Book** references.

## Using the List Class (cont'd)

- The get method returns Object references

- Can be any object

```
Object object = list.get(0);
Dog dog = (Dog) object;
Book book = (Book) list.get(1);
```



- Type would have to be checked

```
List list = new List();
list.add(new Book());
-
Object object = list.get(0);
if(object instanceof Dog){
    Dog dog = (Dog) object;
}
```

- Or ClassCastException will be thrown ( at runtime! )

```
Dog dog = (Dog) list.get(0); //ClassCastException
```

The **get** method was defined to return **Object** references. Since the **List** may contain any object type, the reference might point to any type of object. Only at runtime it is known what the actual type is of the object that is referenced. So every time you obtain a reference from the list, you will have to check its type, otherwise a **ClassCastException** might occur.

Even though in the examples above the content of the **List** is obvious (We add a book a few lines before, so we know what is within the list), you would have to cast the result and have detailed knowledge of what object type is at what position within the list.

In larger applications, the **List** might be populated in a completely different part of the application, as a result the developer who has to work with the content does not know what type of objects reside within the list. (besides them being **Object** types)

## Generic Class

- List as a generic class
- Angle brackets <> are used to specify type parameter
  - Generic type is defined by <T> placeholder

The diagram shows a code snippet for a generic class named `List`. The class has a single instance variable `private T[] objects;` and two methods: `public void add(T i)` and `public T get(int index)`. A red box highlights the entire class definition. Four red arrows point from the text labels to specific parts of the code:

- A red arrow points from the label "Type 'placeholder' definition" to the opening angle bracket `<T>` in the class header `List<T>`.
- A red arrow points from the label "Type 'placeholder' as variable type" to the type parameter `T` in the declaration `private T[] objects;`.
- A red arrow points from the label "Type 'placeholder' as parameter" to the type parameter `T` in the parameter list of the `add` method.
- A red arrow points from the label "Type 'placeholder' as return type" to the type parameter `T` in the return type of the `get` method.

```
public class List<T> {
    private T[] objects;
    public void add(T i) { ... }
    public T get(int index) { ... }
}
```

A real generic class defines a type using a 'placeholder' at class level. This placeholder can then be used throughout the class. (e.g. To define the type of an instance variable, parameter or return type)

# Using Generic Classes

- Type can now be specified when instance is created

```
List<Book> dept = new List<Book>();
```

■ ...or

```
List<Book> dept = new List<>();
```

- The get method now returns an instance of Book

■ Downcast is no longer required

```
Book book = ...;
List<Book> list= new List<>();
list.add(book);
Book item = list.get(1);
```

■ The add method only accepts instances of Book

```
Dog dog = new Dog();
Department<Book> dept = ...;
dept.add(dog);
```



When an instance of the generic class is created, the actual type is defined on the reference and on the assignment. To be exact, the type must be defined on the type of the reference and only has to be defined on the type argument of the constructor when the compiler cannot determine (infer) the type that has to be created. In most cases the compiler can figure out what type of object instance needs to be created by looking at the type that was defined by the reference to which the object reference will be assigned.

As a result of defining the type during the construction of the object, all placeholders that were defined in the class, will now be 'replaced' by the type you just specified. So the **add** method now only accepts instances of **Book** and the **get** method returns **Book** references (without the need for a cast).

## Type Parameter Naming Conventions

- **Naming conventions make it easier to understand code**
- **Type parameter names are (by convention) single uppercase letters**
  - E – Element in collection
  - K – Key
  - N – Number
  - T – Type
  - V – Value
  - S,U,W (etc.) used for additional parameter types

Several naming conventions are defined for the Java programming language. Naming conventions make code easier to read and understand (specially when this code was written by third parties)

Earlier we saw that the generic type is defined using a ‘placeholder’. By convention this ‘placeholder’ is a single uppercase letter. To be more exact, the coding convention defines specific letters, depending on the kind of parameter you define.

## Bounded Types

- The compile time type <T> can be any type

```
public class List<T> {  
    ...  
}
```

- Allows for any type of List to be created

```
List<String> dept1 = new List<>();  
List<Integer> dept2 = new List<>();  
List<Object> dept3 = new List<>();
```

- Type definition can even be omitted

- The get method now returns Object

```
List list = new List();  
Object object = list.get(1);
```

- The add method now accepts Object instances

So now we can define our classes in a very generic way, allowing developers to create object instances that are specialized for any particular type. But so far, we have defined just a placeholder, which can be substituted by any Java object.

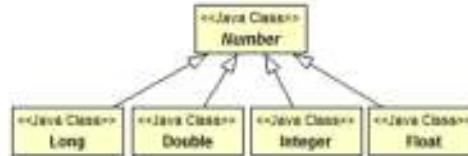
## Bounded Types (Upper Bound)

- Type can be bounded
  - Using the `< T extends ... >` notation

```
public class List<T extends Number> {  
    ...  
}
```

- Only (sub) types of Number are allowed

```
List<Number> list1 = new List<>(); //OK  
List<Long> list2 = new List<>(); //OK  
List<Integer> list3 = new List<>(); //OK  
  
List<Object> list4 = new List<>(); //Compile Error  
List<String> list5 = new List<>(); //Compile Error  
List<Dog> list6 = new List<>(); //Compile Error
```



You can further specify the type of the 'placeholder' by declaring the upper bound of the type. Basically you are specifying that the types for which this can be used can only be of the type specified. (This type or any subtype)

When you try to instantiate the **List** using a type which is not a **Number** type, the compiler will throw an error:

*Bound mismatch: The type Object is not a valid substitute for the bounded parameter <T extends Number> of the type List<T>*

# Bounded Types

- When type definition is omitted

- The get method returns Number

```
List list = new List();
Number number = list.get(1);
```

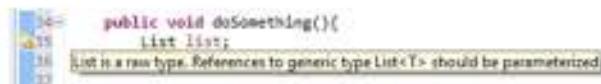
- The add method now accepts Number types

```
List list = new List();
list.add(new Integer(1)); //OK
list.add(new Double(0.5)); //OK
list.add(new String("...")); //Compile Error
```

Besides the fact that the class now can only be instantiated using a **Number** type, when you instantiate the class without declaring the generic type, the methods within the class (that use the generic ‘placeholder’) will use the upper bound of the generic type (in this example, **Number**).

## Raw Types

- When a generic type is used without any type parameter it is known as a raw type
  - Compiler will show warning



- Warning can be suppressed using **SuppressWarnings** annotation

```
@SuppressWarnings("rawtypes")  
public void doSomething() {  
    List list = new List();  
    -  
}
```

## Generic Methods

- Generic methods can also be created
  - Class does not have to be a generic class
  - Allows for types to be defined local to the method
- Type must be defined before return type



Until now we looked at defining type parameters at class level. Types that are declared at class level had to be specified when the class is instantiated and can be used throughout the class.

It is also possible to define generic methods. In this case you define a generic type that is local to the method. The class itself does not have to be a generic type.

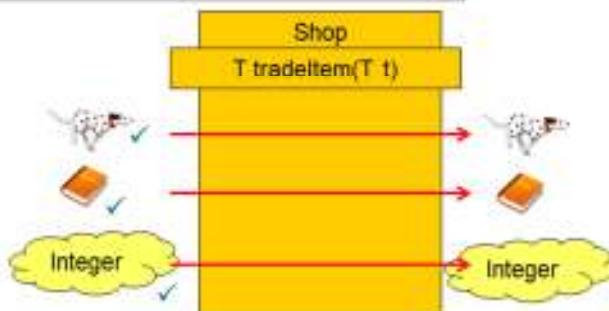
To do so, you define the type 'placeholder' before the return type of the method. The type can now be used as a method parameter, the return type and as method variables (within the method).

## Generic Methods (cont'd)

- Return type will now be the same as the parameter type

```
public <T> T tradeItem(T item) {  
    ...  
}
```

```
Shop shop = new Shop();  
Book book = new Book();  
Dog dog = new Dog();  
Book newBook = shop.tradeItem(book);  
Dog newDog = shop.tradeItem(dog);
```



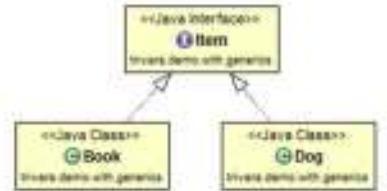
One example of using generic methods is shown above.

A generic type `T` has been specified that is used as the type of the parameter and the return type. So when the method is invoked using an instance of `Dog`, the method must return an instance of `Dog`. When the method is invoked using an instance of `Book`, the method will return an instance of `Book`.

## Generics and Subtypes

- Methods can be invoked with any (sub)type of the parameter specified

```
public void buy(Item item) {
    int price = item.getPrice();
    ...
}
Book book = new Book();
Dog dog = new Dog();
buy(book);
buy(dog);
```



- Is a `List<Dog>` a subtype of `List<Item>`?
  - Can the `print` method be invoked using a `List<Dog>` instance?

```
void print(List<Item> d) { ... }
```



As you know, when defining a method that accepts a parameter of a particular type, this method can be invoked using object instances of this exact type, but also using object instances of any subtype.

In our example, both **Book** and **Dog** are a subtype of **Item**, so they can both be used as a parameter of the `buy` method. Naturally, ‘within’ the **buy** methods you are referencing the object using an **Item** reference and can only use the methods of **Item**.

But now we have introduced generic classes. Is the ‘list of dogs’ a subtype of a ‘list of items’. In other words can I use an instance of the dog list when invoking the `print` method?

## Generics and Subtypes (cont'd)

- NO!

```
public void doSomething() {  
    List<Dog> dogList = new List<>();  
    print(dogList); //COMPILER ERROR  
}  
  
public void print(List<Item> list) {...}
```



- List<Dog> is not a subtype of List<Item>!



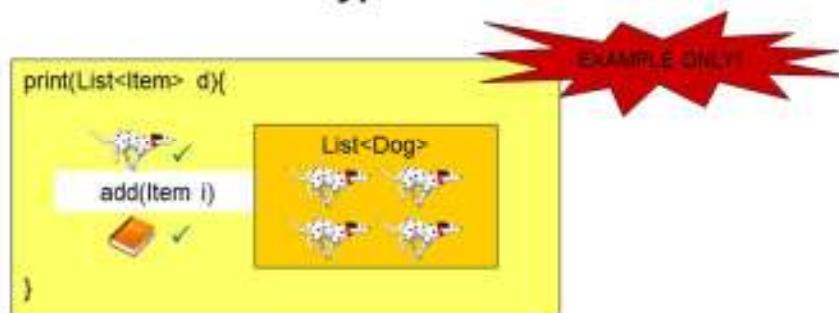
The answer is NO! You will not be able to invoke this method using a 'list of dogs'

## Generics and Subtypes (Example)

- **List<Dog> only accepts Dog instances**



- If it could be a subtype referencing it, using List<Item> would allow all item types to be added



So why is **List<Item>** not a super type of **List<Dog>**

You created an instance of a 'list of dogs' and as a result you could only add dogs to this list. Trying to add a book to it would be caught by the compiler.

Now let's assume for a moment that **List<Item>** would be super type of **List<Dog>**.

You would be able to reference your 'list of dogs' instance using a **List<Item>** reference. As a result the **add** method would now accept any **Item**. As a result (simply by adding another reference to your object) you would now be able to add books to your 'list of dogs'

# Wildcards

- To define a method that works with every List you have to specify a wildcard ( ? )

```
void print(List<?> d) { ... }
```

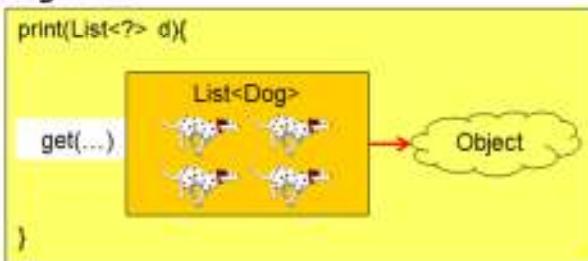
- Any type of List can now be used

```
List<Book> bookList = new List<>();
List<Dog> dogList = new List <>();
print(bookList);
print(dogList);
```

- List methods now return Object

- Real type is unknown

```
void print(List<?> d) {
    Object object = d.get(1);
}
```



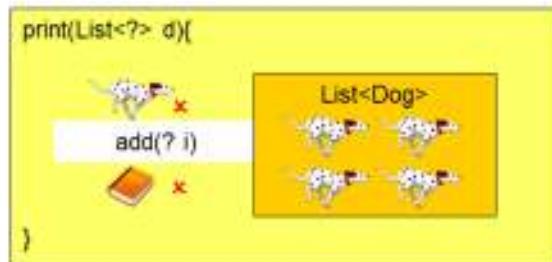
So when you need to define methods that accept generic classes, you will either have to accept the fact that the method only works on object of exactly the type provided OR use a wildcard.

What you are saying by using the wildcard is that you really don't care what type of object is used as parameter.

The drawback of this approach is that within the method, the generic type cannot be determined. So when invoking the `get` method of the `Item` class, the return type will be **Object**.

## Wildcards (cont'd)

- The add method now only accepts instances of type ?
  - Methods that potentially change state of the List can not be invoked



- Code assist in Editor:

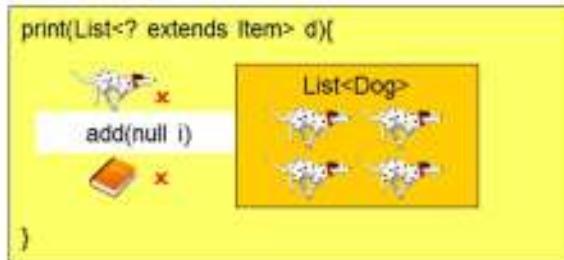
```
public void print(List<?> list) {  
    list.  
    ▾  
    • add(? o) : void - List  
    • equals(Object obj) : boolean - Object  
    • get(int index) : Object - List
```

Also, for methods of the class that use the generic type as parameter, the actual type can not be determined. So the compiler will expect a parameter type of '?', which is of course a type that does not exist.

As a result you will not be able to invoke these methods.

## Bounded Wildcards

- The add method now only accepts instances of type **null**
  - Methods that potentially change state of the List can **still** not be invoked



- Code assist in Editor:

For parameter types, the compiler can still not determine the exact type. However, in this case it will *NOT* accept any **Item** type (that would allow us once again to add books to the dog list). For parameters of the generic type it will only accept generic instances of type '**null**'. Since this type also does not exist, you will still not be able to invoke this method (and 'accidentally' pollute the state of your **List**).

## Lesson Review and Summary

- 1) What are generics?**
- 2) What are wildcards? What do they represent? What is their syntax?**
- 3) Are generics only useful for collections?**
- 4) How are generics defined as a parameter/return type to a method?**
- 5) What is a raw type?**

Generics define two new forms of types:

Parameterized types

Type variables

Wildcards represent unknown parameter types

They represent any parameter type

They are depicted using a question mark

Generics are useful for more than just collections, but that is where they are more commonly used

Generics are defined as a parameter/return type to a method in the following ways:

<T>

<T extends [Class/interface/type variable]>

<?>

<? Extends [class/interface/type variable]>

A raw type is a concrete generic type instantiated without a parameterized type

## **Tutorial: Setup JUnit 4 Project library in IntelliJ**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## Exercise 16: DynamicArray

**`~/StudentWork/code/DynamicArrays/lab-code`**

Please refer to the written lab exercise and follow the directions as provided by your instructor

# **Lesson: Collections**

**Introduction to Generics  
Collections**

## Lesson Agenda

- Using `ArrayList` to maintain a list of data elements
- Using `Iterator` to search elements of a collection
- Using methods of `List` to access sub-lists
- Sorting and shuffling elements in a list
- Creating read-only collections from standard
- Writing a Comparator to provide custom sorting
- Creating your own collection class

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

# The Collections Framework

- **Framework components**
  - Various types of pre-defined interfaces
    - ◆ General purpose
    - ◆ Legacy
    - ◆ Synchronized wrappers
    - ◆ Concurrent
  - Functionality to work with the collections
- **Benefits of the framework**
  - Simplifies application development
  - Improves the quality of application software
  - Provides interoperability across unrelated APIs
  - Interchangeable patterns makes framework easy to learn
  - Easy to extend and customize for application-specific tasks
- **Has full support for:**
  - Generics
  - The for-each loop
  - Autoboxing/Unboxing

## The Collections Framework

JDK 1.2 added a set of classes and interfaces to provide a more complete framework for representing data aggregation. While **Vector** and **Hashtable** proved to be extremely useful, there were a number of shortcomings with those classes.

In addition to providing the classes and interfaces that make up the framework, there are also implementation classes that utilize these more robust features.

The framework also easily supports the creation of application specific collections.

Probably the most notable feature of the Collections framework is that while there may be implementations of collection classes that provide specific behaviors, such as indexed access, all classes in the framework support a more generic set of APIs, allowing different collection implementations to be treated simply as a collection of data.

For example, there may be different implementations of a **Set**, such as hashed versus tree storage, yet application APIs can be written to utilize a **Set**, and not the underlying implementation type.

# Characterizing Collections

- **Type of collection**
  - There are fourteen types of collections grouped into four general categories that define the semantics for working with the data
    - ◆ Set – elements in collection are unique
    - ◆ List – elements in collection can be indexed
    - ◆ Map – set of key/value pairs
    - ◆ Queue – supports FIFO
- **Implementation access styles**
  - Hashed
  - Indexed
  - Linked list
  - Tree
- **Element ordering**
  - Sorted
  - Unsorted
  - Insertion order

## Implementation Styles

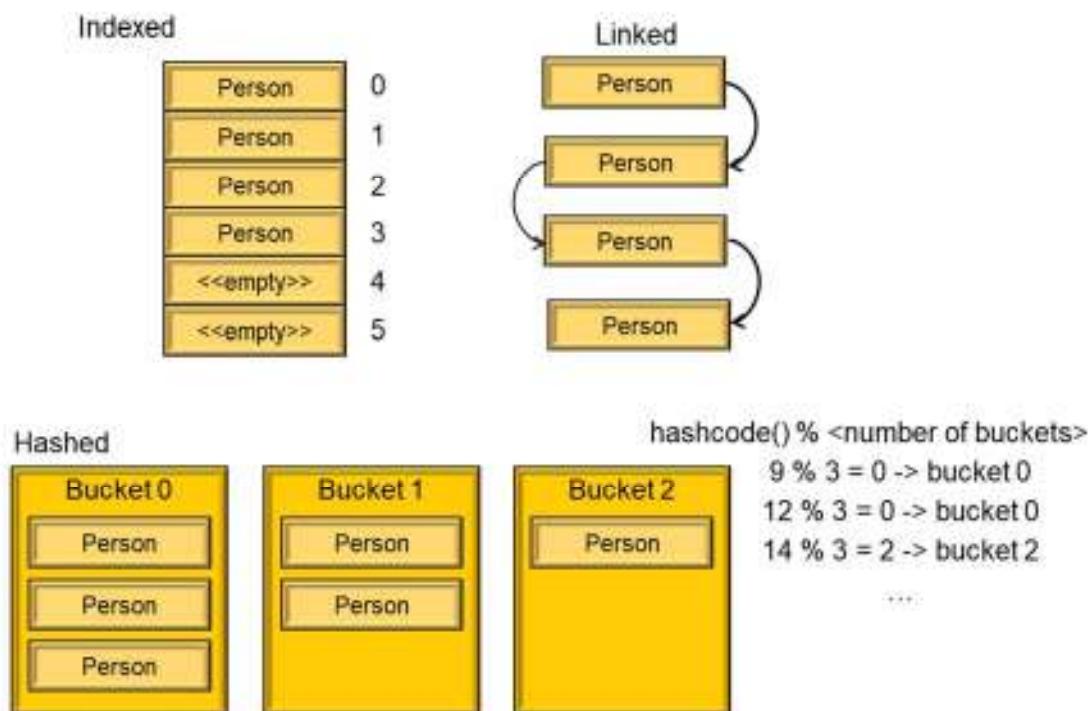
The framework provides implementations of the collection types with varying underlying implementations. Each implementation has its advantages regarding usage and performance, and gives the developer options when writing an application that uses collections.

## Element Ordering

Along the same line as above, there are also interfaces that stipulate that the data collection is sorted or unsorted. An implementation that keeps its collection sorted would implement this additional interface.

All of this will be expanded upon throughout the rest of this lesson.

## Indexed, Linked or Hashed Implementations



## Features of the Collection Implementation Classes

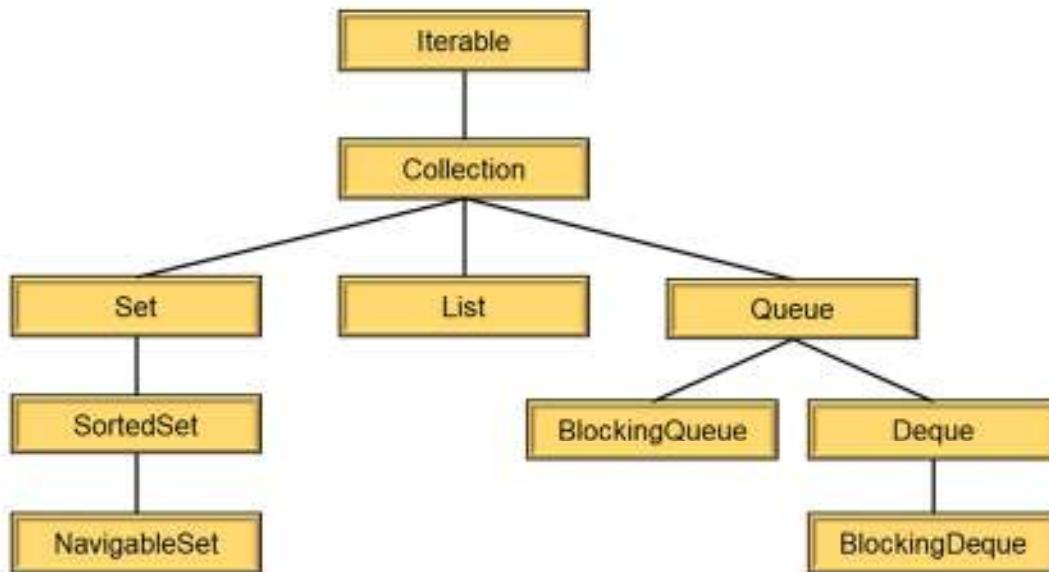
- **Support all of the methods**
- **Values placed into the collection can be null**
- **Access methods are unsynchronized for faster performance**
- **Supports fail-fast concurrent access for more predictable reliability**
- **Are implemented to support the following interfaces**
  - **Serializable**
  - **Cloneable**
- **Note: Code should be written to accept and return standard interface types, not implementations**

### Features Provided by the Standard Implementations

First note that these features are not required by the various Collection types. They are added features provided by the standard implementations that come with the JDK.

For more detailed information on the specific features of the implementation classes you may want to read their API documentation.

## Collection Interface Hierarchy



The **Iterable** interface was introduced in Java 5. Objects that implement this interface can be the source of a for-each loop.

## Top Level Collection Interfaces

- **public interface Collection**
  - Provides core set of collection methods
  - Not specific to lists, sets, maps, etc.
- **public interface Set extends Collection**
  - Provides no additional methods
  - Implies that elements within collection are unique
- **public interface List extends Collection**
  - Entries may not be unique
  - Adds methods to support indexed access
- **public interface Queue extends Collection**
  - Entries may not be unique
  - Adds methods to support insertion, removal, and inspection
  - Queue directly extends Collection, and is not a Set

### Collection

This is the high level interface from which other interfaces extend. It provides the basic methods to represent an object that maintains a data collection.

### Set

**Set** extends **Collection**, and adds no methods. An implementation class that implements this interface is stating that the elements in this collection are unique.

### List

**List** directly extends **Collection**, and is not a **Set**. Whereas elements in a **Set** are unique, elements in a **List** may not be. A **List** adds the methods to support indexed access, which allows for more advanced capabilities like sorting and shuffling.

## Collection<E> Interface

- Collection interface provides element handling
  - Relies heavily on generic types

```
boolean add(E e); //optional  
boolean addAll(Collection<? extends E> c); //optional  
void clear(); //optional  
boolean contains(Object o);  
boolean containsAll(Collection c);  
boolean isEmpty();  
Iterator<E> iterator();  
boolean remove(Object o); //optional  
boolean removeAll(Collection<?> c); //optional  
boolean retainAll(Collection<?> c); //optional  
int size();  
Object[] toArray();  
T[] toArray(T[] array);
```

### The Collection Interface

You'll notice that the methods in this interface are just enough to represent a class that maintains a group of elements. There are no methods that support indexed access, nor are there any methods that imply unique or non-unique elements.

The **Set** and **List** interfaces extend this interface, and the **List** interface adds the methods to support indexed access.

Historically, **Vector** and **Hashtable** supported the **Enumeration** interface to traverse the group of elements contained by that object. There is a minor improvement to **Enumeration** in a type called **Iterator**. This will be covered in more detail shortly.

You may also want to take note of the methods that may be implemented as optional in a **Collection** implementation.

As you will see throughout this lesson, the Collection API relies heavily on generic types for its implementation.

## Optional Methods

- Some methods in the various interfaces may be *optional*
  - Certain operations may not be valid for certain implementation types (e.g. Read-Only collections)
- Methods are declared to throw an exception in the interface
  - The implementation would only throw this exception if it were an *unsupported method*
  - Exception is a runtime exception
  - Developers are expected to read the documentation before attempting to use optional methods

```
public class UnsupportedOperationException extends RuntimeException {  
    ...  
}
```

### Optional Methods Make a More Powerful Framework

In the standard interfaces there are a number of methods that are defined as optional. This is indicated by the fact that in the interface the methods are declared as possibly throwing the **UnsupportedOperationException**.

While it may seem to be the lazy way of creating a framework for collection implementations, there is actually value added to the framework because of this feature.

For example, you may wish to create your own collection, yet do not want to provide the ability to remove elements once they are added. The remove method is optional, and you can write your implementation such that the remove method throws this exception when it is called.

The thing to remember is that this is a **RuntimeException**, and developers using an API with optional methods may wish to place such calls in a try/catch for this exception.

In practice, a developer would read the documentation for the implementation and realize which methods are optional, to determine whether that implementation is suitable for their application.

## Collections Class

- The Collections class contains several utility methods
  - All methods are declared as static

```
void shuffle(List<?> list);

<T extends Comparable<? super T>> void sort(List<T> list);
<T> void sort(List<T> list, Comparator<? super T> comp);
void reverse(List<?> list);

<T extends Object & Comparable<? super T>> T min(Collection<? extends T> c);
<T> T max(Collection<? extends T> coll, Comparator<? super T> c);
<T> void copy(List<? super T> list1, List<? extends T> list2);
<T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
<T> List<T> unmodifiableList(List<? extends T> list);
<K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m);
<T> Set<T> unmodifiableSet(Set<? extends T> set);
<T> List<T> synchronizedList(List<T> list);
<K,V> Map<K,V> synchronizedMap(Map<K,V> map);
```

### Global Utility Methods

This is a big part of the power of the Collections framework.

There is a class called **Collections** (separate from the interface called **Collection**) which provides a large number of static utility methods. These methods provide all sorts of useful operations on different types of collections.

For example, any collection of type **List** can be sorted and shuffled. In addition, any collection type can be wrapped inside an unmodifiable wrapper, or wrapped inside a collection that adds thread synchronization.

You may want to look at the full API of this class to get a better idea of the services available.

Note: There are many more static methods of the **Collections** class.

## java.util.Iterator Interface

- The Collection interface provides the iterator method
  - Can remove elements while within the iteration
  - Map does not have an iterator, but provides them for both keys and values
  - The Iterator<E> interface

```
boolean hasNext();
E next();
void remove(); // optional
```

```
List<Integer> integerList = new LinkedList<>();
integerList.add(new Integer(42));
integerList.add(new Integer(43));

Iterator<Integer> listIterator = integerList.iterator();
Integer item;

while (listIterator.hasNext()) {
    item = listIterator.next();
}
```

### Iterator

Each Collection implementation will provide an implementation of this interface to allow traversing the elements in the collection.

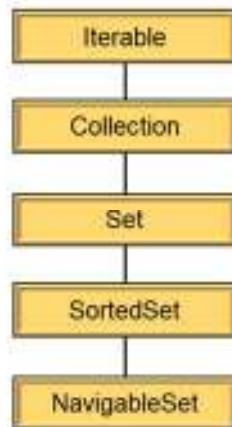
The primary advantage of **Iterator** over **Enumeration** is that **Iterator** may allow removal of elements during traversal. **Enumeration** does not support this method, and there is no reliable way (generally) of removing an element from a group upon completion of the enumeration.

Removing an element while traversing could also be much more efficient, especially in the case where the underlying storage is a linked list, and the element cannot be directly indexed.

Also note that the remove method is optional in this interface.

## The Set<E> Interface

- Provides no additional methods over the Collection interface
- Implementations using this interface imply that elements are unique (using their equals method)



### Set

The **Set** interface is a direct extension of **Collection**, and does not add any methods.

Any implementation class that implements the **Set** interface is guaranteeing that the elements in the collection are unique according to the equals methods of the elements themselves.

## SortedSet<E> Interface

- Extends Set interface
- There must be a comparison mechanism
  - Either the elements implement the Comparable interface
  - Or you provide a Comparator implementation
- Comparison mechanism should be consistent with equals and hashCode
- SortedSet adds the following methods:

```
Comparator<? super E> comparator();
E first();
SortedSet<E> headSet(E toElement);
E last();
SortedSet<E> subSet(E fromElement, E toElement);
SortedSet<E> tailSet(E fromIncl);
```

### SortedSet

A Set that maintains its elements in order would implement this interface to indicate that the elements are sorted, and to provide additional methods to access the elements in sorted order.

While sort order is primarily defined by the elements themselves, a **SortedSet** can bypass that mechanism using its own sorter.

Sorting will be covered shortly.

## Comparable<T> Interface

- Primitive wrappers and String implement the java.lang.Comparable interface

```
public int compareTo(T o);
```

```
public class Item implements Comparable<Item> {  
    @Override  
    public int compareTo(Item otherItem) {  
        int item2Price = otherItem.getPrice();  
        if(this.getPrice() > item2Price){  
            return 1;  
        else if(this.getPrice() < item2Price){  
            return -1;  
        else  
            return 0;  
    }  
    }  
  
    public int getPrice(){ ... }  
}
```

**■ < 0 – this object less than the other object**  
**■ 0 - this object equals the other object**  
**■ > 0 – this object greater than the other object**

### Comparable

Elements that will be maintained in sorted order will ideally implement this interface. It contains a single method which allows it to compare itself to another object (usually of the same type). Note that many standard classes, such as **String** and the primitive wrapper classes, implement this interface.

When elements of this type are placed in a **SortedSet**, the **SortedSet** can ask each element if it is greater than or less than the adjacent element.

The generic **Comparable** interface allows for the definition of a comparable implementation that is specific for an **Item** type. When the generic type is omitted, the type of the ‘other’ object needs to be checked and downcasted to the appropriate type before the actual code of comparing instance variables can be done.

## Comparator<T> Interface

- A `java.util.Comparator` implementation must be provided to a `SortedSet` to sort elements that do not implement `Comparable`

```
public int compare(T o1, T o2);
```

```
public class PriceComparator implements Comparator<Item> {
    @Override
    public int compare(Item item1, Item item2) {
        int item1Price = item1.getPrice();
        int item2Price = item2.getPrice();
        if(item1Price > item2Price) {
            return 1;
        } else if(item1Price < item2Price) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

■ < 0 – first object is less than the second object  
■ 0 - first object equals the second object  
■ > 0 – first object greater than the second object

### Comparator

A `SortedSet` may not wish to rely on the elements to determine their own sort order, or the elements may not implement the `Comparable` interface. In both cases the `SortedSet` will have to rely on an implementation of `Comparator` to perform these comparisons. Notice that the compare method takes two objects to compare.

Similar to the `Comparable` interface, the types of the two parameters need to be checked and downcasted when the generic type of the `Comparator` is not specified

```
class PriceComparator implements Comparator{
    public int compare(Object item1, Object item2) {
        //Downcast to Item objects
        int item1Price = ((Item)item1).getPrice();
        int item2Price = ((Item)item2).getPrice();

        if(item1Price > item2Price)
            return 1;
        else if(item1Price < item2Price)
            return -1;
        else
            return 0;
    }
}
```

## NavigableSet<E> Interface

- Extends SortedSet interface
- Provides methods for:
  - Finding closest match to a search target
    - ◆ lower, floor, ceiling and higher
  - Ascending operations on the set
  - Descending operations on the set
  - Subset between search targets
  - Subset before or subset after a search target

## The List<E> Interface

- Extends Collection interface
- Collection of elements that may not be unique
- In addition to Collection methods, adds methods to provide integer access

```
void add(int i, E o); //optional  
boolean addAll(int i, Collection<? extends E> c); //optional  
E get(int i);  
int indexOf(Object o);  
int lastIndexOf(Object o);  
ListIterator<E> listIterator();  
ListIterator<E> listIterator(int i);  
E remove(int i); //optional  
E set(int i, E o); //optional  
List<E> subList(int startIncl, int endExcl);
```

### List

The **List** interface is separate from **Set**, and directly extends the **Collection** interface. It adds the methods to support indexed access to elements in the collection.

A list may contain duplicate elements, although an implementation may choose not to allow this. There are no methods to provide sorting. However, a **List** may support sorting, and we'll see shortly that there is a helper method in the Collections framework that can perform sorting on any implementation of a **List**.

## ListIterator<E> Interface

- Extends Iterator, adds methods to go in reverse
- Indexing methods

```
public int nextIndex();  
public int previousIndex();
```

- Reverse iteration

```
public boolean hasPrevious();  
public E previous();
```

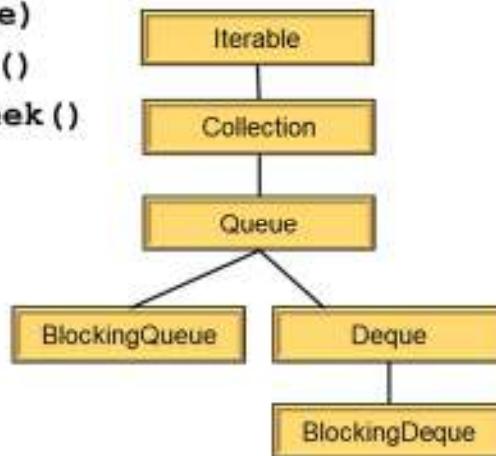
### A More Refined Iterator

Because a **List** implicitly maintains ordering of elements (at least, until you sort or shuffle them) there is a subtype of Iterator which is the **ListIterator**.

The **ListIterator** provides additional methods to back up in the iteration, and also to return the next and previous index value.

## Queue<E> Interface

- Queues provide insertion, extraction, and inspection operations for processing elements
- The order of insertion and removal is dependent on the type of implementation selected for the queue
- Insertion: `add(e)`, `offer(e)`
- Removal: `remove()`, `poll()`
- Inspection: `element()`, `peek()`



## Map Interfaces

- **Do not subclass from Collection**
- **public interface Map**
  - Somewhat similar to Collection interface
  - Same implication as Set (unique keys)
  - Methods designed to access elements as key/value pairs
- **public interface SortedMap extends Map**
  - Keys are ordered based on specified comparison
  - Allows access to first and last key
  - Similar in principle to extended features of SortedSet
- **Provide methods to return keys and values as collections**
- **Map has implementation specific to enums**
  - EnumMap

### Map

While presented as one of the three high level interfaces in the Collections framework, this interface actually does not extend **Collection**. As an interface that represents storage of key/value pairs, it really is not a **Collection**. In fact, a **Map** is really a pair of collections; a collection of keys and a collection of values.

This concept is supported by the fact that **Map** provides methods to access both the keys and the values as collections.

The **Hashtable** class is an implementation of this interface.

### SortedMap

The **SortedMap** extends **Map**, and guarantees that the keys are maintained in sorted order. The **Hashtable** class cannot implement this interface because the hashing mechanism is implicitly re-ordering the storage for faster access.

## Creating and Using a `HashMap<K, V>`

- **Create the `HashMap` using generics**

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

- **Place/get data into the map**

```
myMap.put("ID", "Idaho");
String s = myMap.get("ID");
```

- **Checking to see if something is already there**

- **For the key**

```
if (myMap.containsKey("ID")) {...}
```

- **For the value**

```
if (myMap.containsValue("Idaho")) {...}
```

**HashMap** stores only object references. That's why it's impossible to use primitive data types like **double** or **int**. Use wrapper class (like **Integer** or **Double**) instead.

For multi-threaded (synchronized) array class, use **Hashtable**.

## Feature Comparison

- **Set implementations – unique entries**
  - HashSet – hash algorithm for fast access, no ordering
  - TreeSet – tree structure, slower, preserves ordering
- **List implementations – non-unique, indexed**
  - ArrayList – fast indexing, slow insertion/deletion
  - LinkedList – slow indexing, fast insertion/deletion
- **Map implementations – key/value pairs**
  - HashMap – hash algorithm for fast access, no ordering
  - TreeMap – tree structure, slower, sorts by key

### Set

A **Set** guarantees that the elements are unique. However, a **Set** cannot be indexed nor can it be sorted or shuffled using the static utility methods. A **Set** may be sorted if it is actually the subtype **SortedSet**.

### List

The **List** type is probably the most flexible. While by definition it does not guarantee uniqueness, an implementation can stipulate that its entries are unique. It provides indexed access, which works well for array-based implementations, not so well for linked-list based implementations. **List** also has the greatest amount of support from the static utility methods, and can be shuffled, sorted, and reversed.

### Map

By definition a **Map** is not a collection. However, both its keys and its values can be accessed as a **Collection**. **Map** allows data to be stored as key/value pairs.

The **HashMap** implementation class hashes the keys, and so cannot guarantee ordering. The **TreeMap** will actually implicitly sort to maintain its internal tree.

## Using the Right Collection

- Collections or Collection interfaces provide several types
  - Set - Unique elements
  - List - Non-unique elements
  - Map - Key/value pairs, with unique keys
  - Queue - Ordered elements by insertion and removal
  - Easy to determine which of these you require
- What are the primary operations?
  - Value searching - Hash!
  - Indexed access - Array!
  - Insertions/deletions - Linked list!
  - Ordering - Queues
  - Sorting - Trees

## Using the Right Collection (cont'd)

- **Standard implementations**
  - Hash - HashSet, HashMap
  - Array - ArrayList
  - Linked list - LinkedList
  - Other - TreeSet, TreeMap
- **You may choose to use a combination**

Know how you will use a collection

Using the wrong kind of collection can cause mammoth performance degradation. First of all, consider that **Vector** and **HashTable** are synchronized. Do you need that? Second, are you searching or traversing? Third, is order important? Picking the right implementation can be critical.

This also illustrates the value of returning a **Collection** interface type, rather than a specific implementation to your user. This gives you the freedom to change the implementation without affecting the user.

## Collections and Multithreading

- Collection interfaces do not require implementations to be synchronized
  - Improves performance at the cost of thread safety
- Any Collection implementation can be wrapped in a standard synchronized wrapper
  - Using static methods of Collections class

```
<T> Collection<T> synchronizedCollection(Collection<T> c);
<T> List<T> synchronizedList(List<T> list);
<T> Set<T> synchronizedSet(Set<T> set);
<K,V> Map<K,V> synchronizedMap(Map<K,V> map);
<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> sset);
<K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> smap);
```

- These provide synchronized methods

### Making Collections Thread-Safe

If you were to write your own collection class, you could make the methods thread-safe. However, doing so would mean that in all cases the users of your class would have to pay for this overhead.

The convention is to create collection classes that are not thread-safe. In most cases a collection class will not be accessed by multiple threads simply by virtue of the application design. By keeping its methods unsynchronized, you are getting faster performance for the more common cases.

In the cases where thread safety is a concern, you can use the standard synchronization wrappers provided with the JDK. These wrappers work equally well with any implementation of a collection.

Simply use the static methods of the **Collections** class to create a synchronized wrapper around a given collection class, whether it be your own collection class, or one of the standard collection classes provided in the JDK.

## Optimizing Collection Constructors – Initial Size

- Some collection implementations use arrays for element storage
- Array must be copied to increase storage
- Use the constructor to provide an initial capacity large enough to hold the maximum required

```
// Instead of this...
List<Integer> testList = new ArrayList<>(); //default size!
// do this...
List<Integer> testList = new ArrayList<>(25);
```

- Internal code for ArrayList

```
private void ensureCapacityHelper(int minCapacity) {
    ...
    elementData = new Object[newCapacity];
    System.arraycopy(oldData, 0, elementData, 0, elementCount);
}
```

### Storage Allocation is Expensive

If you can, try to size the initial collection large enough to hold all of the elements. This will save reallocation later, and can be a very large performance improvement for long running programs.

## Lesson Review and Summary

- 1) What are the four collections styles?**
- 2) Must the application know the type of collection being referenced before it can be traversed?**
- 3) Are standard collections classes thread-safe?**

1. Collections framework defines these types
  - set, list, map, queue
2. No, all application method APIs should only reference interfaces
3. Most Collection implementations are not thread-safe

## **Exercise 17: Using Hashtable and HashMap**

**`~/StudentWork/code/HashMap/lab-code`**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## **Exercise 18: Collections Poker**

**`~/StudentWork/code/Collections/lab-code`**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

# **Session: Lambda Expressions; Collections and Streams**

**Introduction to Lambda Expressions  
Java 8 Collection Updates  
Streams  
Collectors**

# **Lesson: Introduction to Lambda Expressions**

## **Introduction to Lambda Expressions**

Java 8 Collection Updates

Streams

Collectors

## Lesson Agenda

- **Understanding the basic concept of functional programming**
- **Writing basic lambda expressions**
- **Understanding functional interfaces**
- **Understanding the difference between anonymous classes and lambda expressions**

## Functional vs OO Programming

- **Functional programming emphasizes functions that produce results**
  - Depend only on their inputs (not program state)
  - Functions deliver a result
  - Avoid internal state
  - Calling function with same parameters provides same result every time
- **Object Oriented Programming emphasis on state**
  - ‘Functions’ (methods) are used to control internal state
  - Not all methods provide a result (just change state)
  - Calling methods with same parameters might provide different result

With functional programming, the developer is focussed on creating functions that can be reused throughout the application. A function relies only on the input parameters it receives and avoids the use of state and mutable data. Calling the same function multiple times with the same parameters will therefore provide the same result every time.

Furthermore a function sole purpose is to produce some kind of result. This result can be internal within the function (e.g. print something on screen), but most often it will return a value.

Within object-oriented programming the functions (methods) are used to change the internal state of an object. Calling the same method twice might provide a different result because the internal state of the object has changed by the earlier method invocation. Also, in most OO applications, methods on objects do not even provide a result, invoking a setter method on an object only changes the internal state of the object.

# Functional Interfaces

- Java 8 defines the concept of Functional interfaces
  - Also called Single Abstract Method interfaces
- Functional interfaces
  - Contain at most one abstract method
  - May contain default methods
  - May contain static methods
- Instances of the interface can be created
  - Using Lambda Expressions
  - Using method references (covered later)
  - Using constructor references (covered later)
  - (Anonymous) inner classes

As we have seen in an earlier lesson, lambda expressions can be used instead of anonymous inner classes. However, where anonymous inner classes can also be used to implement interfaces that contain multiple methods, lambda expressions can only be used when the interface contains, at most, one abstract method.

Keep in mind that methods without a method body have to be declared as abstract. Since methods in an interface (until Java 8) never contain an implementation they are implicitly abstract.

Throughout the JDK you will find a number of interfaces that only contain a single method (e.g. **Runnable**, **Comparable**, **ActionListener**). These types of interfaces are often referred to as Single Abstract Method (SAM) interfaces, in Java 8 these are called “Functional Interfaces”.

A functional interface contains, at most, one abstract method. Starting Java 8, it has also become possible to write default and static methods in an interface!

Until Java 8, the only option you had to come up with an implementation of one of these SAM interfaces was to implement a class that implements this interface. From now on you can also use lambda expressions to accomplish exactly the same!

Additionally, you can also use a reference to a method of a constructor to create an instance of the functional interface. Method and constructor references will be covered later.

## The FunctionalInterface Annotation

- Indicates that the interface is intended as a Functional interface
  - Allowing the compiler to throw errors when interface does not comply to specification
- Informative annotation
  - All interfaces that comply to the specification rules are considered functional interfaces

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface()
```

The **FunctionalInterface** annotation, introduced in Java 8, is considered an informative annotation, allowing the compiler throw errors when the interface is not a valid functional interface.

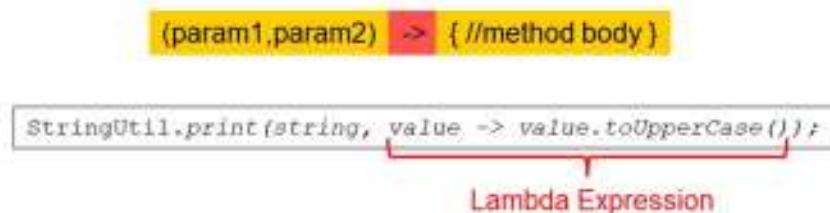
The use of the annotation is optional. Any interface that defines a single abstract method (not being one of the methods of **java.lang.Object**) is considered to be a functional interface. However, it is recommended that all functional interfaces are annotated using this annotation.

## Introduction to Lambda Expressions

- A lambda expression is an anonymous method
  - Methods without a name
  - May have parameters
  - Has a method body
- Method definition without
  - Name
  - Access modifier
  - Return type declaration
- A lambda expression is a powerful shorthand for defining interface implementations
  - Useful in places where a method is being used only once

# Lambda Expression Syntax

- Lambda parameters and body separated by ->



- Lambda expressions can only be used to 'implement' functional interfaces!

Lambda expressions can be seen as methods without a name (anonymous methods). Just like 'normal' methods in the Java programming language, lambda expressions have parameters and a method body, they just don't have a name (just like anonymous classes) and their syntax is slightly different from the methods you have written until now.

Every lambda expression consists of two parts:

- 1) The parameters that are used within the expression
- 2) The method body of the lambda expression.

The parameters of the lambda expression are 'variables', local the method body of the expression (similar to method parameters within regular methods).

The method body defines the function that is to be performed when this expression is interpreted.

Parameters and method body are separated by the '->' notation.

# Utility Methods

- Parameters of utility methods can consist of
  - Data that needs to be processed
  - Functionality to be performed on data

```
public class StringUtil {  
    /**  
     * @param string String to be printed  
     * @param function Function to be performed before printing  
     */  
    public static void print(String string, Function function) {  
        String result = function.apply(string);  
        System.out.println(result);  
    }  
}
```

- For example, ‘transform’ String before printing

```
public interface Function {  
    String apply(String value);  
}
```

# Implementing Functions

- **Different implementations of Function can be developed**

```
public class ToUpperCaseFunction implements Function<String, String> {
    public String apply(String value) {
        return value.toUpperCase();
    }
}
```

- ...and used when invoking utility method

```
public void displayText() {
    String string = "Some Text";
    ToUpperCaseFunction function = new ToUpperCaseFunction();
    StringUtil.print(string, function);
}
```

- ...or in a single line

```
public void displayText() {
    String string = "Some Text";
    StringUtil.print(string, new ToUpperCaseFunction());
}
```

## Inner Classes

- Java allows for the definition of classes within another class or method
  - Inner classes are only known to surrounding class or method

```
public void displayText() {  
    String string = "Some Text";  
    class ToUpperCaseFunction implements Function {  
        public String apply(String value) {  
            return value.toUpperCase();  
        }  
    }  
    StringUtil.print(string, new ToUpperCaseFunction());  
}
```

- Benefits of inner classes
  - Can access private data of the outer class
  - Can be hidden from other classes within the package

## Anonymous Inner Classes

- Anonymous inner classes can be defined in-line in your code
  - Has no name and can implement an interface

```
public void displayText() {  
    String string = "Some Text";  
    StringUtil.print(string,  
        new Function(){  
            public String apply(String value) {  
                return value.toUpperCase();  
            }  
        }  
    );  
}
```

- Used when functionality is only used once

## Transforming to Lambda

- Why do we have to specify the interface?
  - Compiler can infer interface type from parameter type of method we try to invoke!

```
StringUtil.print(string,
    new Function(){
        public String apply(String string) {
            return string.toUpperCase();
        }
    });

```

```
public class StringUtil {
    public static void print(String string, Function function){
        String result = function.apply(string);
        System.out.println(result);
    }
}
```

## Transforming to Lambda (cont'd)

- Why do we have to specify the method name?
  - Compiler can infer interface type from parameter type of method we try to invoke!
  - Functional interface only has a single method!

```
StringUtil.print(string, new Function() {
    public String apply(String string) {
        return string.toUpperCase();
    }
});
```

```
public interface Function {
    String apply(String value);
}
```

## Transforming to Lambda (cont'd)

- So everything that can be inferred by compiler can be removed
  - Lambda notation -> needs to be added in between parameters and method body

```
StringUtil.print(string, new Function<String, String>() {
    public String apply(String value) -> {
        return string.toUpperCase();
    }
});
```

Add lambda notation



- Results in a Lambda expression being used

```
StringUtil.print(string,
    (String value) -> {
        return value.toUpperCase();
});
};
```

# 'Optimizing' Lambda Expression

- Parameter type can also be inferred by compiler
  - But can be explicitly declared

```
StringUtil.print(string,
    (String value) -> {
        return value.toUpperCase();
    });
}

public interface Function {
    String apply(String value);
}
```



- When using a single parameter parentheses are optional

```
StringUtil.print(string,
    value ->{
        return value.toUpperCase();
    });
}
```

## 'Optimizing' Lambda Expression (cont'd)

- Curly brackets are optional when body contains single statement
- Return type of single statement expression is the return type of the body expression

```
StringUtil.print(string, value ->{  
    return value.toUpperCase();  
}  
);
```

- Lambda expression could 'simplified' to

```
StringUtil.print(string, value -> value.toUpperCase());
```

# Lambda Expression Parameters

- Lambda expressions can have zero or more parameters
  - Empty parentheses represent empty set of parameters

```
() -> System.out.println("No Argument");
(String string) -> System.out.println(string);
(String s, int i) -> System.out.println(s + "=" + i);
```

- Parameter types can be explicitly declared
  - Or inferred from context
- Parameters are enclosed in parentheses
- When using a single parameter parentheses are optional
  - Unless type is explicitly declared

```
string -> System.out.println(string);
```

Lambda expressions can define zero or more parameters. As you will see in a little bit, the amount of parameters depends on the interface that lies underneath.

Parentheses are used to enclose a comma-separated list of parameters. When no parameters are required by the lambda expression, empty parentheses () are used.

The types of the parameters can be explicitly declared in the expression, but since the types can most often be inferred from the context of the expression, the definition of parameter types is optional.

When the expression only accepts a single parameter and the type is **not** explicitly declared the use of parentheses is also optional.

## Lambda Expression Body

- **Body of lambda expressions contains zero or more statements**
- **Curly brackets are optional when body contains single statement**

```
(string) -> System.out.println(string);
```

- **Curly brackets are required when body contains multiple statements**

```
string -> {  
    System.out.println(string);  
    System.out.println(string.toLowerCase());  
};
```

- **(note the semicolon after the last curly bracket!)**

As you will see throughout this lesson, the body of the lambda expression may consist of zero or more lines. So it is possible to write a function that does nothing!

When the body of the expression consists of two or more lines, the method body must be enclosed by curly brackets {}. Note the semicolon after the closing curly bracket (similar to the implementation of an anonymous class).

However, when the expression consists of just a single statement, the use of curly brackets is optional.

## Functional Interfaces Which Return Data

- Return type of single statement expression is the return type of the body expression

```
Function function = value -> value.toUpperCase()
```

- The following will not compile

```
Function function = value -> value.length();
```



- Return type of multi statement expression is type of value returned

```
Function function = value -> {  
    int length = value.length();  
    if(length > 77) {  
        return value.substring(0, 77) + "...";  
    }  
    return value;  
};
```

```
public interface Function {  
    String apply(String value);  
}
```

- Or void...

When the body of a lambda expression consists of only a single line of code, the return type of this ‘anonymous method’, is determined by the result of the expression that makes up the body.

In the example, the **length** method returns a value of type **int** and can therefore not be assigned to a variable of type **Function**.

Just like with regular methods, it is possible to have multiple return statements within the method block. Naturally, all return values should be of the same type. When different types are used, the compiler will check the expected type by inspecting the type declared in the assignment and indicate which return statement returns the wrong type.

# Functional Interfaces and Generics

- Using generic functional interfaces allows for re-use of interface definition

```
public interface Function {  
    String apply(String value);  
}  
  
public interface Function<T, U> {  
    U apply(T value);  
}
```

```
public class DoubleUtil {  
    public static Integer round(Double d, Function<Double, Integer> f) {  
        return f.apply(d);  
    }  
}
```

```
public class StringUtil {  
    public static void print(String s, Function<String, String> f) {  
        String result = f.apply(s);  
        System.out.println(result);  
    }  
}
```

## Functional Interfaces and Generics (cont'd)

- Parameter and return type can still be inferred

```
public interface Function<T, U> {  
    U apply(T value);  
}
```

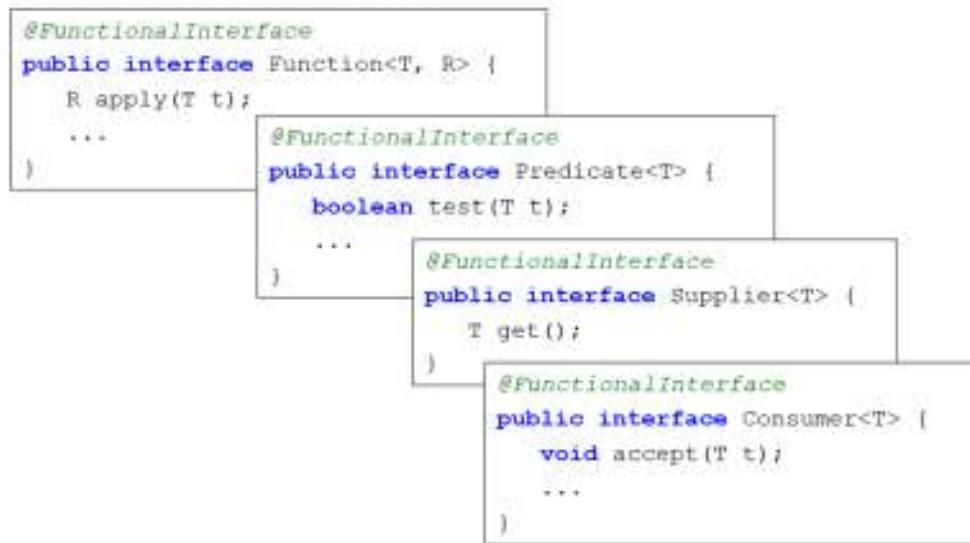
```
public class StringUtil {  
    public static void print(String s, Function<String, String> f) {  
        "  
    }  
}
```

```
String string = "Some Text";  
double d = 16.5;  
StringUtil.print(string, value -> value.toUpperCase());  
Integer round = DoubleUtil.round(d, dVal -> dVal.intValue());
```

```
public class DoubleUtil {  
    public static Integer round(Double d, Function<Double, Integer> f) {  
        "  
    }  
}
```

## java.util.function Package

- **java.util.function package contains commonly used functional interfaces**



The **java.util.function** package contains a wide variety of functional interfaces (function shapes). These interfaces contain definitions for the most commonly used operations to which lambda expressions can be applied.

The **Consumer**, **Functions**, **Predicate** and **Supplier** interfaces are considered to be interfaces which define the basic shapes. You will also find several derived function shapes including several function shapes that deal with primitives.

The **Consumer** interface accepts a single parameter and does not return any value. Unlike other functions, method body of the consumer relies on side-effects to perform its job.

## java.util.function Package (cont'd)

| Type                | Method             |                             |
|---------------------|--------------------|-----------------------------|
| Function<T, R>      | R apply(T)         |                             |
| BiFunction<T, U, R> | R apply(T, U)      |                             |
| UnaryOperator<T>    | T apply(T)         | extends Function<T, T>      |
| BinaryOperator<T>   | T apply(T, T)      | extends BiFunction<T, T, T> |
| Consumer<T>         | void accept(T)     |                             |
| BiConsumer<T, V>    | void accept(T, V)  |                             |
| Predicate<T>        | boolean test(T)    |                             |
| BiPredicate<T, V>   | boolean test(T, V) |                             |
| Supplier<T>         | T get()            |                             |

The most commonly used Functional interfaces (and their functional method) is shown above. Once you start using Lambda expressions, these are the interfaces you most often encounter.

## Default methods on Functional Interfaces

- Standard Functional Interfaces contain default methods
  - Can be used to create alternate interface implementations
- `java.util.function.Predicate`
  - `and(Predicate<? super T> other)`
  - `negate()`
  - `or(Predicate<? super T> other)`
- `java.util.function.Function`
  - `andThen(Function<? super R, ? extends V> after)`
  - `compose(Function<? super V, ? extends T> before)`
- ...and many more

You might have been looking at the JavaDoc of the Functional interfaces and noticed that besides the abstract method the ‘standard’ functional interfaces also contain several default methods that can be used to create instances.

The Predicate interface contains methods to combine two predicates using a logical AND/OR operation and also contains a method which creates a Predicate that is the negative version of the one provided.

The Function interface also contains methods that allow functions to be combined into a ‘larger’ function.

During the next exercise, make sure you take some time to browse through the JavaDoc looking for default methods that might make your life as a programmer a whole lot easier.

## Final Variables

- Until Java 8, variables used in inner class **had** to be declared as **final**

```
final String someValue = "button clicked";
JButton button = new JButton("OK");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        System.out.println(someValue);
    }
});
```

- Java 8 allows the use of variables that are not explicitly declared as final

```
String someValue = "button clicked";
JButton button = new JButton("OK");
button.addActionListener(event -> System.out.println(someValue));
```

- They must effectively still be final!

```
String someValue = "button clicked";
someValue = someValue.toUpperCase();
JButton button = new JButton("OK");
button.addActionListener(event -> System.out.println(someValue));
```



Until Java 8, variables used from within the body of an anonymous inner classes you could only reference variables that were implicitly defined as final.

Starting Java 8, the use of the **final** keyword has become optional. Variables that are not explicitly defined as **final** can now be used from within anonymous inner classes.

However, the referenced variable must still effectively be **final**, the second code snippet will not compile, since the variable is reassigned.

Also notice that instead of the bulky anonymous inner class, we used a lambda expression to define the event handler. Since the **ActionListener** contains only one abstract method, the interface is considered to be a functional interface. As a result, a lambda expression can be used to define the handler.

## Method References

- Lambda expressions define both the parameters and functionality
  - Corresponding to functional method of interface
- Existing method signatures might correspond to functional method signature

```
public class StringUtils {  
    public static boolean isUpperCase(String s) {  
        ...  
    }  
}  
  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

- Functional interface can also be implemented using method reference

```
Predicate<String> p = StringUtils::isUpperCase;
```

As we have seen, a lambda expression defines both the parameters and logic that, until now, you would have implemented in the method of an interface.

What if you have a method that has a signature that corresponds to the functional method. You could write a lambda expression that invokes that method, but you can also use a reference to this method directly.

## Method References (cont'd)

- Notice usage of double colon ( :: )
  - Followed by method name

```
Predicate<String> p = StringUtils::isUpperCase;
```
  - Method is referenced, **not** invoked (yet)
- Method references can point to
  - Static methods
  - Instance methods of objects
  - An instance method of one particular instance

A method reference is defined by using a double colon, followed by the method name (without parentheses). The next couple of slides will talk about the part before the double colon. For now, just remember that the method is not invoked at this point. A reference to the method is used, invoking this method occurs when the entire expression is executed

# Static method references

- Reference points to static method of a class

Classname :: Static method name

```
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}

public class StringUtils{
    public static boolean checkLength(String s, int length) {
        ...
    }
}

BiPredicate<String, Integer> bp = StringUtils::checkLength;
```

In this example we have a static method that takes two arguments and returns a boolean. This makes a perfect match to the BiPredicate functional interface, at least when the parameter types of the method match the generic types of the BiPredicate interface.

An instance of BiPredicate can now be created by using a method reference to the checkLength method of the StringUtils class.

# Instance method references

- Instance methods can be referenced

- As long as instance type and return value match method of functional method

```
public static String process(List<Passenger> passengers,
                           Function<Passenger, String> func) {
    for (Passenger passenger: passengers) {
        String apply = func.apply(passenger);
    }
}
```

process(passengers, Passenger::getName);

Classname :: Instance method name

```
public class Passenger {
    public String getName() { ... }
}

public interface Function<T, R> {
    R apply(T t);
}
```

Instead of referencing a static method you can also reference an instance method of a class.

The `process` method, shown above, accepts (besides a list of `Passenger` objects) a `Function` that accepts a `Passenger` and returns a `String`. In this example the `Passenger` class defines an instance method `getName`. By defining the function as a reference to the instance method, the list of passengers is now iterated through and the `getName` method is called on each object instance in the list.

The notation for referencing an instance method is identical to referencing a static method. The difference (of course) is that before the method can be invoked an instance of the class must be available

## Reference method of specific instance

- Use instance method of provided object instance



- `System.out` is an instance
  - `printXXX` methods can be referenced

```
Consumer<Passenger> c = System.out::println;
```

When the method you need to reference is implemented by a particular object instance, you can define a method reference to point to exactly that one method.

In order to reference a method on a particular object instance you define the variable name before the double colon.

For debugging purposes: keep in mind that `System.out` points to an object instance, so referencing any of the `printXXX` methods (to consume a value) is accomplished by the example shown above

# Constructor References

- **The Function interface**
  - Accepts one argument
  - Produces an object instance
- **Classes might define constructor with single parameter**
  - Also returns object instance
- **Functional interface can also be implemented using constructor reference**
- **Constructor parameters must match parameters of functional method**

```
public interface Function<T, R> {
    R apply(T t);
}
```

```
public class Passenger {
    public Passenger(String name) { ... }
}
```

```
Function<String, Passenger> f = Passenger::new;
```

Classname : new

```
public class Passenger {
    public Passenger(String name) { ... }
    public Passenger(String name, Integer id) { ... }
}
BiFunction<String, Integer, Passenger> bf = Passenger::new;
```

The function interface accepts one argument and produces an object reference. So, when you have a class that has a single argument constructor, you might want to reference this constructor. ‘Implementing’ a functional interface can be accomplished by referencing a constructor of a class.

To reference the constructor of a class you specify the class name followed by the double colon, followed by the new keyword.

Naturally, also in this situation the parameter type(s) must match the desired types of the functional method and the Class should be of the return type of the functional method.

## Lambda Expression vs Anonymous Class

- **Usage of the `this` keyword**
  - Within anonymous classes, references the anonymous class
  - Within lambda expressions, references enclosing class
- **Anonymous classes are compiled into separate class files**
- **Lambda expressions are compiled into private methods of enclosing class**

As stated throughout this lesson, lambda expressions can be used wherever anonymous inner classes are used as long as the interface to be implemented is a functional interface. However, there are a few minor differences between the two.

When the `this` keyword is used from within the body of an anonymous inner class (method), the `this` pointer references instance of the anonymous inner class.

When the `this` keyword is used from within the method body of a lambda expression, the `this` pointer references the instance of the enclosing class.

Not as important for you as a developer, but you should also notice that when a class containing an anonymous inner class is compiled, it results in two separate class files - one for the enclosing class and one for the anonymous inner class. The class name of inner class is 'generated' by adding `$x` (where `x` is a number) to the class name of the enclosing class.

When compiling a class containing a lambda expression, the expression is compiled into a private method of the enclosing class.

## Lesson Review and Summary

1. All lambda expressions require parameters to be defined
  - a. True
  - b. False
2. Lambda expression method body always return a value
  - a. True
  - b. False
3. What is a functional interface?
4. Is the use of the `FunctionalInterface` annotation mandatory for functional interfaces?
5. Name the differences (in use) between anonymous inner classes and lambda expressions

- 1) False: You can define a lambda expression without defining parameters, however for this you need to define a set of empty parentheses
- 2) False: For example, the `Consumer` interface performs the function within the method body of the expression
- 3) An interface that contains at most one abstract method
- 4) No, it is an informative interface, allowing the compiler to check if the interface complies to the specification of functional interfaces. All interfaces that contain only a single abstract method are considered functional interfaces
- 5) The use of the `this` keyword AND the fact that lambda expressions can only be used for functional interfaces, anonymous inner classes can also be used when the interface contains multiple abstract methods

# Lesson: Java 8 Collection Updates

Introduction to Lambda Expressions  
**Java 8 Collection Updates**  
Streams  
Collectors

## Lesson Agenda

- **Using lambda expressions to define operations on collection entries**
- **Using the Spliterator for parallel processing**
- **Becoming familiar the static methods of the Comparator interface**
- **Becoming familiar methods of the ConcurrentHashMap**

# Collections and Lambda Expressions

- Java 8 introduced new methods in the Collection API

- Utilizing lambda expressions and method references

```
List<String> destinations = ...  
destinations.forEach(System.out::println); //Consumer
```

- Allowing collections to be used as source of Stream

```
List<String> destinations = ...  
Stream<String> stream = destinations.stream();
```

- Simplify creation of Comparator instances

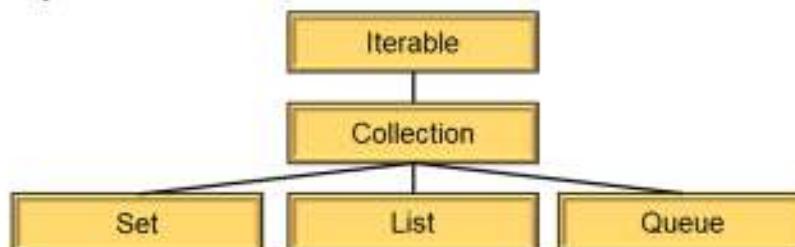
```
List<String> destinations = ...  
Comparator<String> comparator =  
Comparator.comparing(String::length);  
destinations.sort(comparator); //sort method also added in Java 8!
```

- ...

With the introduction of lambda expressions, method references and the new Stream API, the existing Collection API could now be updated to support these new ‘features’. Throughout this lesson, we will introduce some of the new methods that have been added to the Collection API.

## Iterable Interface

- Iterable objects can be source of a for-each loop
  - Introduced in Java 5
  - Super-interface of Collection interface



- Defines forEach method
  - Requires definition of Consumer

```
List<String> titles = new ArrayList<>();  
titles.forEach(System.out::println); //Consumer
```

In Java 5, the **Iterable** interface was added to the JDK. Classes that implement this interface can become a source for use in the for-each loop (also introduced in Java 5). As a result the **Iterable** interface became the super-interface of the **Collection** interface. Keep in mind that the **Iterable** interface is not part of the Collection API. Other interfaces, like **BeanContext**, **DirectoryStream** and **Path** (all not part of the Collection API) are also sub-interfaces.

With the introduction of the **forEach** method, a **Consumer** can be defined that will be applied to each element in the **Collection**. The example shown above, prints each element in the collection.

## Spliterator Interface

- **Spliterator can be obtained from Iterable**
  - Special iterator to traverse Collection
    - ◆ Can split the collection, allowing for parallel processing

```
List<Passenger> passengersForFlight = ...  
Spliterator<Passenger> spliterator = passengersForFlightspliterator();
```

- **Can be split into multiple Spliterator instances**
  - Returns second instance when data can be partitioned
    - ◆ ...or null when data cannot be split

```
Spliterator<String> spliterator = listspliterator();  
long estimateSize = spliterator.estimateSize(); //e.g. 5  
Spliterator<String> spliterator2 = spliteratortrySplit();  
long estimateSize2 = spliterator.estimateSize(); // 3  
long estimateSize3 = spliterator2.estimateSize(); // 2
```

- **Spliterator does not provide parallel processing behavior**

A **Spliterator** instance can be obtained from any **Iterable** instance. A **Spliterator** is a special kind of **Iterator**, not only allowing for the data to be traversed, but also provides methods to (optionally) split the collection of data into multiple parts, something that would be needed if the data is to be processed in parallel.

The **Spliterator** does not provide the parallel behavior, it assists in dividing the data into multiple parts, but also it provides information to determine if the data can be properly split into multiple chunks.

## Spliterator Interface (cont'd)

- Elements referenced by Spliterator can traversed
  - One at a time

```
Consumer<String> consumer = null;
while(spliterator.tryAdvance(consumer)) {
    System.out.println("Processing element");
}
```

- All remaining elements
  - ◆ Sequentially in current thread

```
Consumer<String> consumer = ...
spliterator2.forEachRemaining(consumer);
```

The **Spliterator** defines two methods for iterating over the elements. The **tryAdvance** method will take a consumer instance which processes the next available element (when available), returning true as long as elements are available.

The **forEachRemaining** method also accepts a **Consumer** instance, but invoking this method will process all remaining elements sequentially in the current thread.

## Spliterator Characteristics

- Provide information on the data (source) to be split
  - Reported as int value (logical OR of individual values)

```
List<String> list = new ArrayList<>();
Spliterator<String> split = listspliterator();
int characteristics = spliterator.characteristics();
boolean hasCharacteristics = split.hasCharacteristics(Spliterator.ORDERED);
```

- Not all sources are good candidates for parallel processing

| Characteristic | Description                                         |
|----------------|-----------------------------------------------------|
| CONCURRENT     | Source may be concurrently modified                 |
| DISTINCT       | Elements in source are unique                       |
| IMMUTABLE      | Element source cannot be modified                   |
| NONNULL        | Source does not hold null values                    |
| ORDERED        | Elements are encountered in predefined order        |
| SIZED          | Estimated size is known before split.               |
| SORTED         | Elements are encountered in a predefined sort order |
| SUBSIZED       | Obtained Spliterators will be SIZE and SUBSIZED     |

Once a Spliterator instance has been obtained, from the collection, the characteristics methods can be invoked to obtain information about the data-set that is to be processed. The characteristics value is represented as an int, being a logical OR value of several predefined values.

The hasCharacteristics method can be used to determine whether the Spliterator contains a specific characteristic. In the example shown above the source is an ArrayList, resulting in a Spliterator with the characteristics ORDERED, SIZED and SUBSIZED

Keep in mind that not all collections of data are good candidates for parallel processing. In order to process the elements in parallel, the data must be able to be split up in parts. Sources from which the number of elements can not be determined at the start are bad candidates, since splitting up the data into multiple (equal) chunks at the start of the process is next to impossible.

## Methods Added to Collection

- Methods have been added to create a Stream from the collection of elements
  - Java 8 introduced the Stream API

```
Collection<String> destinations = ...  
Stream<String> stream = destinations.stream();  
Stream<String> parallelStream = destinations.parallelStream();
```

- The removeIf method allows for removal of entries
  - ...if they satisfy given Predicate

```
Collection<FlightInformation> allFlights = ...  
LocalTime noon = LocalTime.of(12, 00);  
allFlights.removeIf(flight -> flight.getDepartureTime().isBefore(noon));
```

◆ ...or defined as method reference

```
allFlights.removeIf(FlightInformation::isCodeShare);
```

```
public class FlightInformation {  
    ...  
    public boolean isCodeShare()  
    {...}  
}
```

Two new methods were added to the **Collection** interface, allowing the creation of a **Stream** instance, based on the elements contained within the collection.

The new '**removeIf**' method allows for the definition of a **Predicate** (since Java 8, most likely defined using lambda expression or method reference). All **Collection** entries that match the given predicate will be removed from the collection.

## List Interface

- List interface adds **replaceAll** method
  - Requires **UnaryOperator** instance
  - ◆ Function where output type equals input type

```
passengersForFlight.replaceAll(passenger -> {  
    if (passenger.hasCheckedIn())  
        return passenger;  
    else  
        return passengerService.getNextStandbyPassenger(flightNo);  
});
```

- The **sort** method was ‘moved’ to List interface
  - Collections.sort now calls method on List

```
Comparator<Passenger> seatComparator = ...  
passengersForFlight.sort(seatComparator);
```

The **List** interface has also received new methods. The **sort** method has been moved from the **Collections** class into the **List** interface. Since Java 8, the **Collections.sort** method actually invokes the **List.sort** method in order to sort the elements.

With the introduction of lambda expressions, it now has also become easier to define a function that should be applied to elements within the list. As a result, a **replaceAll** method has been defined on the **List** interface. The **UnaryOperator** (A function, of which the parameter and the return type are the same) allows for replacement objects to be created.

## Comparator Static Methods

- **Comparator defines several utility methods for the creation of instances**

- **Function defines sort-key**

```
//sort by Destination code  
Comparator<Flight> comparator =  
    Comparator.comparing(Flight::getDestinationCode);
```

- **Combining sort-key with existing comparator**

◆ **String.CASE\_INSENSITIVE\_ORDER is Comparator**

```
Comparator<FlightInformation> comparator =  
    Comparator.comparing(Flight::getDestination, String.CASE_INSENSITIVE_ORDER);
```

- **Comparators optimized for primitives (int, long and double)**

```
Comparator<FlightInformation> comparator =  
    Comparator.comparingInt(FlightInformation::getFlightNumber);
```

Until Java 8, creating a **Comparator** required the definition of a new class that implements this interface and providing an implementation of the compare method. A lot of the implementations did a simple comparison of one (or maybe two) fields of the objects to be compared.

Java 8 introduced several static methods on the **Comparator** interface to simplify the creation of **Comparator** instances. The comparing method (for example) takes a **Function**. The **Function** acts as a sorting-key extractor and the resulting comparator uses the natural ordering of the provided key to determine the ordering.

Overloaded methods even allow for an external **Comparator** to be used to define the sorting order of the extracted keys. In the example shown above a **String** comparator (defined as a constant on the **String** class) is used.

When working with primitive types, specialized **Comparator** implementations can be obtained.

## Comparator Static Methods (cont'd)

### ■ Comparators based on natural ordering

#### ◆ Elements implementing Comparable interface

```
List<String> list = ...  
//sort in reverse natural order  
list.sort(Comparator.reverseOrder());  
...  
//sort in natural ordering again  
list.sort(Comparator.naturalOrder());
```

### ■ Null-safe comparators

```
Comparator<Flight> timeComparator =  
    Comparator.comparing(Flight::getDepartureTime);  
Comparator<Flight> nullsFirst = Comparator.nullsFirst(timeComparator);  
Comparator<Flight> nullsLast = Comparator.nullsLast(timeComparator);
```

As we have seen earlier, a sort method has been defined on the **List** class. To order the elements within the list using their natural ordering the 'Natural Ordering' comparator can be obtained, even allowing for the elements to be sorted in reversed natural ordering.

The **nullsFirst** and **nullsLast** methods can be used to wrap existing comparators into null-safe wrappers, defining the location of these null values in the result (at the beginning or at the end)

## Comparator Methods

- Several new methods were added to the Comparator interface
  - All implemented as default method
    - ◆ Existing implementations do not have to be updated
- Allows instances to be re-used or combined
  - Reversing the sorting order

```
Comparator<Flight> timeComparator = ...  
Comparator<Flight> reversedTimeComparator = timeComparator.reversed();
```

- Combining multiple comparators
  - ◆ When equal according to first comparator, second comparator will be applied

```
Comparator<Flight> destinationComparator = ...  
Comparator<Flight> timeComparator = ...  
Comparator<Flight> combined =  
    destinationComparator.thenComparing(timeComparator);
```

The **Comparator** interface also received several new methods. All of these methods have been defined as default methods in order to make sure that existing implementations of this interface do not have to be updated. For the most part, these default implementation contain implementations that can be used out of the box (instead of throwing **UnsupportedOperationException** instances as we see in other interfaces).

The **reversed** method can be used to create a **Comparator** based upon another **Comparator**, but this time resulting in a reversed sorting order.

The **thenComparing** method(s) can be used to define a ‘multi-level’ comparator, when the result of the first comparator is 0 (elements are equal), the second **Comparator** is applied.

## Comparator Methods (cont'd)

### ■ Defining 'multi-level' comparators using Function(s)

```
//Define Comparator using Function
Comparator<Flight> destComparator = Comparator.comparing(Flight::getDestinationCode);
//Define second-level comparator using Function
Comparator<Flight> combined = destComparator.thenComparing(Flight::getDepartureTime);
```

### ■ Allowing for 'intermediate' key-extractor to be defined

```
// sort by destination
Comparator<Flight> destComparator = Comparator.comparing(Flight::getDestination);
// sort by number of Passengers
Function<Flight, List<Passenger>> keyExtractor = flight -> flight.getPassengers();
Comparator<List<Passenger>> listSizeComparator = Comparator.comparing(List::size);
//Combine
Comparator<Flight> comb = destComparator.thenComparing(keyExtractor, listSizeComparator);
```

### ● Specialized methods are defined for use with primitives

- `thenComparingInt`, `thenComparingLong`,  
`thenComparingDouble`

We have seen that the static `comparing` method can be used to define a function (in the example shown above define using a method reference) to define the sorting order. A `thenComparing` method may also accept a **Function** to define the 'second-level' of the comparison. In the example shown above the **Flight** elements are sorted by destination. For a single destination the flights are sorted by departure time.

When defining multi-level comparators, the 'second-level' sorting key can also be defined using a **Function** and a **Comparator**. In the example shown above the flights are sorted by destination, for a single destination the elements are sorted by the number of passengers (the size of the list of passengers) on that flight.

# The Map Interface

- Several new methods have been added

| Method                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------|
| V getDefault(Object key, V defaultValue)                                                                                                   |
| Get value for given key, or default when no mapping is present                                                                             |
| V putIfAbsent(K key, V value)                                                                                                              |
| Put value when no mapping is present for key (or mapped to null). Returns null when no mapping is present, otherwise returns current value |
| boolean remove(Object key, Object value)                                                                                                   |
| Remove entry for given key only when current value equals given value                                                                      |
| boolean replace(K key, V oldValue, V newValue)                                                                                             |
| Replace entry for given key only when current value equals given oldValue                                                                  |
| V replace(K key, V value)                                                                                                                  |
| Replace entry for given key only when mapping already exists for key                                                                       |

```
Map<String, String> destinationMap = ...  
String destination = destinationMap.getOrDefault("XYZ", "Unknown Destination");
```

The **Map** interface has also been updated in Java 8. While the **getOrDefault** method allows for the definition of a value that is returned when there is no mapping for the given key, other methods allow for the map content to be updated only when the current key/value pair matched the values provided.

## The Map Interface (cont'd)

### ■ Allowing for 'functions' to be defined on Map entries

#### Method

V getOrDefault(Object key, V defaultValue)

Get value for given key, or default when no mapping is present

void forEach(BiConsumer<? super K,? super V> action)

Perform action for each key/value pair

void replaceAll(BiFunction<? super K,? super V,? extends V> func)

Replace each value with result of function

```
Map<String, Flight> flightMap = new HashMap<>();
flightMap.forEach((key, value) -> System.out.printf("Key: %s - Value %s\n", key, value));
```

Several new methods have become available to process key/value pairs contained within the Map

# The Map Interface...

## Method

V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)

When key is not in Map (or mapped to null), compute value and add to map (unless value is null). Returns value for key (current or computed)

V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)

When key is present, compute new value. Mapping is removed when function returns null. Returns new value

V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)

Compute new value. Mapping is removed when function returns null. Returns new value

V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)

Merge the given value with the value currently in the map for this key. Or add the value under the given key

```
destinationMap.put("AMS", "Amsterdam");
destinationMap.merge("AMS", " Schiphol Airport",
                    (entry, add) -> entry.concat(add));
//or using Method reference
destinationMap.merge("AMS", " Schiphol Airport", String::concat);
```

...or 'compute' a new map entry using the **Function** provided.

## ConcurrentHashMap Class

- Java 8 introduced 30 new methods
  - Methods for processing each mapping
    - ◆ `forEach, forEachKey, forEachValue, forEachEntry`
  - Methods for searching
    - ◆ `search, searchKeys, searchValue, searchEntry`
  - Methods for reducing
    - ◆ `reduce, reduceEntries, reduceKeys, reduce.ToDouble, ...`
  - **mappingCount (should be used instead of size)**
    - ◆ Map may contain more entries than can be represented by int
  - Method for getting value or a default
    - ◆ `getOrDefault`

```
ConcurrentHashMap<String, List<Passenger>> map = ...  
map.getOrDefault("KL603", Collections.EMPTY_LIST);
```

With the introduction of lambda expressions and functional interfaces, the **ConcurrentHashMap** has been expanded to allow for a more declarative way of defining operations that need to take place on its content.

Over 30 methods have been added to the class to allow for the processing, searching and reducing the mappings.

## ConcurrentHashMap forEach

- The **forEach** method can be used to process each entry in Map
  - BiConsumer processes both key and value

```
ConcurrentHashMap<String, List<Passenger>> passengersByFlightNo = ...  
passengersByFlightNo.forEach(  
    (flightNo, passengers) -> {  
        String format = String.format("%s passengers on flight %s",  
   passengers.size(), flightNo);  
    });  
public void forEach(BiConsumer<? super K, ? super V> action) {...}
```

- Overloaded version allows definition of parallel threshold
  - ◆ Number of elements needed for operation to be executed in parallel

```
passengersByFlightNo.forEach(threshold, (flightNo, passengers) -> {...});
```

Several **forEach** methods (different methods for dealing with just the keys, just the value or with the entire mapping) have been added. These methods require (Bi)Functions to be defined that define what should happen for each element that is about to be processed.

Overloaded versions of these methods exist, that allow the operation to be done in parallel streams. The first parameter define the estimated number of elements that is required before the process is to be run in parallel.

## ConcurrentHashMap forEach (cont'd)

- **forEach methods allowing definition of (bi)function**
  - Mappings can be transformed before being consumed
    - ◆ Will not be consumed when function returns null

```
passengersByFlightNo.forEachValue(threshold,
    passengers ->
        passengers.stream().map(Passenger::getName).collect(toList()),
    passengerNames ->
        passengerNames.forEach(System.out::println));
```

```
public <U> void forEachValue(long parallelismThreshold,
    Function<? super V, ? extends U> transformer,
    Consumer<? super U> action)
```

Other **forEach** methods allow for the definition of an additional transformer, before each element is consumed.

In the example shown above, each list (the map is defined as a list of Passengers by flightNumber) is transformed from a list of passenger object to a list of **String** objects (passenger names), the consumer then prints-out all passenger names.

## ConcurrentHashMap search

- Several search methods can be used to search Map
  - Searching for keys, values or entries

```
String name = ...  
ConcurrentHashMap<String, List<Passenger>> data = ...  
//Find out what flightNumber a given passenger is on  
String flightNumber = data.search(threshold,  
    {flightNo, passengers} -> {  
        for (Passenger p : passengers) {  
            if (name.equals(p.getName())) return flightNo;  
        }  
        return null;  
});
```

```
public <U> U search(long parallelismThreshold,  
    BiFunction<? super K, ? super V, ? extends U> searchFunction)
```

- (Bi)function returns non-null on success

Another great addition to the **ConcurrentHashMap** is the definition of several search methods. Just like the **forEach** methods, different search methods have been defined to allow for searching just the keys, just the values of the entire entry in the map.

A (Bi)Function is used to implement the search criteria. When an element meets the criteria a non-null value is returned. As soon as a single element meets the criteria, this value is returned (resulting in just one result).

In the example shown above all mappings are searched. For each mapping the value (the list of passengers) is inspected to determine if a passenger with the given name is present. When the passenger is found the flightNumber (the key under which the list was stored in the map) is returned. Otherwise the function returns null;

## ConcurrentHashMap Reducing

- Reducing results in an accumulated value
  - Second parameter defines how data is to be accumulated

```
ConcurrentHashMap<String, List<Passenger>> data = ...
List<Passenger> reduceValues = data.reduceValues(threshold,
  (list1, list2) ->
    combineLists(list1, list2) // invoking utility method
);
public V reduceValues(long parallelismThreshold,
  BiFunction<? super V, ? super V, ? extends V> reducer){...}
```

- Overloaded method allows for definition of intermediate transformer

- Additional methods exist for reducing to primitive type

```
int totalNumberOfPassengers = data.reduceValuesToInt(threshold,
  passengers -> passengers.size(),
  0,
  (count1, count2) -> count1 + count2);
public int reduceValuesToInt(long parallelismThreshold,
  ToIntFunction<? super V> transformer, int basis,
  IntBinaryOperator reducer) {...}
```

Reducing allows the developers to accumulate a single value, by using values from the **Map**. Again, several methods have been defined to accumulate just keys, just values or use the entire mapping.

In first example shown above, values of the **Map** (Lists of Passengers) are accumulated in a single **List**.

Just like the other new methods (**forEach** and **search**), overloaded versions are available to define an intermediate transformer, allowing the original data of the **Map** to be transformed into a different type before being accumulated.

Also notice that when the accumulated value must be a primitive type, special reduce methods have been made available (avoiding the overhead of (un)boxing)

The second example ‘transforms’ the list of passengers into an integer value (the size of the list), before the total number of passengers is accumulated by the **IntBinaryOperator**

## Lesson Review and Summary

- 1. What is the difference between the `Collections.sort` and the `List.sort` method?**
- 2. What is the purpose of the `Spliterator`?**
- 3. What is the purpose of the static method defined on the `Comparator` class?**

- 1) Nothing, `Collections.sort` forwards the call to the `List.sort` method
- 2) The `Spliterator` abstract the behavior involved in partitioning collection data, needed when the data is to be processed in parallel
- 3) Simplify the creation of `Comparator` instances by defining the keys on which sorting should take place using a `Function`

## Exercise 19: Functional Collections

`~/StudentWork/code/FunctionalCollections/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Streams

Introduction to Lambda Expressions  
Java 8 Collection Updates  
**Streams**  
Collectors

## Objectives: Streams

This lesson covers how to work with the Stream API.

Specifically, it covers:

- Understanding the problem with collections in Java
- Thinking of program solutions in a declarative way
  - What should the program do, not how should it be done
- Using the Stream API to process elements in collections of data
- Understanding the difference between intermediate and terminal stream operations
- Filtering elements from a Stream
- Finding element(s) within a Stream
- Collecting the elements from a Stream into a List

# Collections of Data

- **Collections of data are heavily used in Java**
  - Applications create and process data within collection
- **Data within collection is often**
  - Searched through for data that meets criteria
  - Grouped by category or type
  - Used to perform operations like average, max or min
- **Query languages like SQL provides a declarative language to accomplish such tasks**

SELECT MIN(DEPARTURE\_TIME) FROM FLIGHTS
- **Collection API does not provide such declarative language**
  - Data has to be iterated over (often multiple times) and logic has to be programmed explicitly

When developing Java applications, sooner or later you will be using collections of data on which operations need to be performed or to which data needs to be added. When we think of collections of data, we often think of databases and the SQL query language that allows us to perform operations on the set of data. Within other tiers of an application, developers use collections of data most of the time using one of the standard collection implementations that come with the JDK. (e.g. `java.util.List`)

Whether you look at collections of data within the context of a database or as instances of Java objects, the data within is what we are interested in. More precisely, the operations we can perform on this data.

When using SQL, you can select the rows of a table that meet certain criteria (only Flights to New York that leave after 2PM), or you select the row that has the earliest departure time. Even things like grouping rows of data by airline, destination or time of day are pretty straightforward when using SQL. Best of all, only the rows you select will be placed in memory and the remaining data will remain on the file-system.

When using the Collections API, a declarative query language like SQL does not exist. To get the earliest flight to New York, you as a developer have to obtain an iterator from the collection, iterate through the list one item at the time, inspect the destination of the flight and, when the destination is 'New York', you have to check the departure time, compare this to the departure time of a flight you found earlier in the collection and store the current object reference instead of the previous one, when you found a flight that leaves earlier.

## Declarative ‘Programming’

- **Program what you are trying to accomplish**
  - Not program how this is accomplished
- **For example: search for the earliest flight to LA**
  - The ‘old’ way
    - ◆ Iterate through the list of all available flights looking for flights to LA
    - ◆ Sort the list of flights found by departure time (asc)
    - ◆ Get the first entry in the list
- **What you really want to program is:**

```
SELECT MIN(DEPARTURE_TIME) FROM FLIGHTS WHERE DESTINATION='Los Angeles'
```

Declarative programming is a programming paradigm in which the focus lies on describing the structure and elements of an application in terms of the logic that needs to be implemented for the problem domain without describing the control flow. In other words, describe what the application should do, instead of describing how it should be done.

We just described the logic needed to find the earliest flight to New York, here we are looking for a flight to Los Angeles. In both cases the task that is to be performed is to find the earliest flight to a certain destination. The task remains the same, however the steps described here differ!

Now look at the SQL statement, that is a declarative way of defining what you want, the database will figure out how to come up with the answer (and will probably do so a lot faster than when we were using the ‘external’ iteration approach described above).

# 'Garbage Variables'

- The `laFlights` collection is a temporary container
  - Only created to support further processing

```
List<Flight> allFlights = FlightDAO.getAllFlights();  
  
List<Flight> laFlights = new ArrayList<>();  
for (Flight flight : allFlights) {  
    if ("Los Angeles".equals(flight.getDestination())) {  
        laFlights.add(flight);  
    }  
}  
  
Collections.sort(laFlights, (f1, f2) ->  
    f1.getDepartureTime().compareTo(f2.getDepartureTime()));  
Flight earliest = laFlights.get(0);
```

- Why create this *garbage* variable?
  - It has no meaning within the logic we are trying to implement

To make matters even worse, when browsing through collections in Java, developers often rely on extra variables and temporary collections to store information while iterating over the initial set of data.

Just take a moment and think of all the times you have written code similar to this, creating additional objects (and didn't clean up any cleanup) when iterating through a collection.

## Large Collections

- What if your collections contains thousands of entries?
  - Do we really have to iterate over the entire List before we can show results?

```
List<Flight> allFlights = ...  
  
List<Flight> laFlights = new ArrayList<>();  
for (Flight flight : allFlights) {  
    if ("Los Angeles".equals(flight.getDestination())) {  
        laFlights.add(flight);  
    }  
}
```

- Multithreaded implementations provide a solution
  - Result can be processed immediately in separate thread
- Concurrency issues have to be resolved by programmer!

When you are developing enterprise applications, specially when developing web applications, do you really want your customer to wait for the application to go through the entire collections of data before they are finally presented with a result?

Sure you could write a multithreaded implementation and make sure that every entry in the collection that meets the criteria is immediately handled by a separate thread to be displayed to the client, while the remaining part of the collections is iterated over. But, that would mean that you as a developer would have to deal with the concurrency issues that are involved.

## Java Stream API

- The Java Stream API allows for a declarative way of programming
  - Using pre-defined building blocks
  - Utilizing lambda expressions

```
List<Flight> laFlights = allFlights.stream()
    .filter( f -> "Los Angeles".equals(f.getDestination()) )
    .sorted(comparing(Flight::getDepartureTime))
    .collect(toList());
```

- Notice that the iteration is implemented within building blocks
  - Implemented to take advantage of multi-core architectures

The Stream API, introduced in Java 8, provides developers with the building blocks to define the logic of a task in a declarative way. Utilizing lambda expressions and functional interfaces, these building-blocks can be configured and chained to describe the operation that needs to take place on the collection of data.

The building blocks and expressions used to describe the operation(s) will be explained in more detail in following lessons, but you should notice that each block describes what needs to be done. How the collections of data is processed (iterated through) is defined within the building block. Not only does this mean that the iteration logic is now done internally (and no longer done by the developer), it also means that the implementations can take full advantage of the multi-core architectures on which applications are running these days.

# Collections and Streams

- Streams manipulate collections of data
  - Arrays
  - File(s)
  - Collection classes
  - ...
- The **stream()** method has been added to the **java.util.Collection** interface

```
List<Flight> allFlights = FlightDAO.getAllFlights();  
Stream<Flight> stream = allFlights.stream();
```

As we will see throughout this lesson, the Streams API allow us to manipulate collections of data using pre-defined building blocks and lambda expressions.

Even though a **Stream** can be created using a variety of sources, the examples used in this lesson will use streams that use an implementation of the **Collection** interface as the source of the data.

In order to provide support for streams, the **stream** method was added to the **Collection** interface.

## The Flight Example

- Throughout this lesson we will use a Stream of Flight objects
  - Departing from Boston's Logan International Airport

```
public class Flight {  
  
    private String destination;  
    private String destinationCode;  
    private String airlineName;  
    private String airlineCode;  
    private String flightNumber;  
    private LocalDate date;  
    private LocalTime departureTime;  
    private boolean international;  
  
    //Getter and Setter methods  
}
```

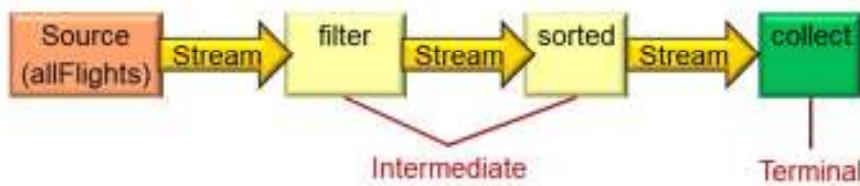
To introduce the various possibilities of using streams on collections of data, we will be using a **Collection of Flight** objects, each one resembling a flight departing from Boston's Logan International Airport. The collection contains all flights that leave this airport. Each instance containing information about the destination, the airline, the flight number and of course the departure time.

Over 400,000 flights leave this airport each year. Even though this might be a small set of data compared to the amount of data you might be working with on a daily basis, you would not want to iterate over this collection over and over again to find the flight that you are interested in.

## java.util.stream.Stream Interface

- Stream interface defines two categories of operations
  - Intermediate operations
  - Terminal operations

```
List<Flight> laFlights = allFlights.stream()
    .filter( f -> "Los Angeles".equals(f.getDestination()) )
    .sorted(comparing(Flight::getDepartureTime))
    .collect(toList());
```



When looking at the abstract methods defined by the **Stream** interface, you can categorize the methods into two groups – a group of methods that return an instance of **Stream**, and a group of methods to return non-stream instances.

The methods that return instances of **Stream** are referred to as ‘intermediate operations’. These methods can perform operations on a **Stream** and express the outcome in the form of another stream.

The other methods are called ‘terminal’ operations. As we will see later, they ‘start’ the operations on a **Stream** and collect the outcome of the stream into, for example, a **List**.

# Intermediate Operations

- **Intermediate operations**
  - Return a new Stream
  - Can be chained to define pipeline
- **Do nothing until terminal operation is invoked on stream**
  - (operations might be ‘merged’)

| Stream Interface        |
|-------------------------|
| Intermediate Operations |
| skip(long ...)          |
| distinct()              |
| peek(Consumer ...)      |
| map(Function ...)       |
| filter(Predicate ...)   |
| sorted()                |
| sorted(Comparator ...)  |
| flatMap(Function ...)   |

In short, intermediate operations return a new instance of **Stream**. As a result, these types of operations can be chained together to define multiple operations that need to be performed on the input stream.

By doing so, you are creating a pipeline through which the elements in the initial **Stream** will travel.

You should remember that invoking intermediate operations on a **Stream** does not produce an immediate result. Intermediate operations are processed once a terminal operation is invoked on the **Stream**. As a result, the implementation is capable of ‘merging’ operations on the **Stream** into a single iteration of the elements within the **Stream**.

## Terminal Operations

- Terminal operations execute operations on pipeline
  - Return non-stream value
  - Closes the pipeline once completed

| Stream Interface                 |
|----------------------------------|
| Terminal Operations              |
| R collect<Collector ...>         |
| long count()                     |
| void forEach(Consumer ...)       |
| Optional findAny()               |
| Optional findFirst()             |
| Optional max(Comparator ...)     |
| Optional min(Comparator ...)     |
| boolean noneMatch(Predicate ...) |
| boolean allMatch(Predicate ...)  |
| boolean anyMatch(Predicate ...)  |
| ...                              |

Every pipeline that is defined on a **Stream** should contain at least one terminal operation. Operations, defined by intermediate operations, are not executed until the terminal operation is invoked.

Besides invoking the intermediate operations and collecting the result of these operations, the terminal operation also closes the **Stream** once the operations have been completed. As a result a **Stream** instance can only be used once! An attempt to invoke another terminal operation on a closed **Stream** will result an **IllegalStateException** (“stream has already been operated upon or closed”).

## Collectors

- **Collector implementations accumulate stream elements**
  - Into a non-Stream type
- **Type of the Stream.collect method parameter**
- **The Collectors class contains factory methods for obtaining commonly used Collector implementations**
  - For example: Collect result of pipeline into List

```
allFlights.stream()
    .filter(Flight::isInternational)
    .collect(Collectors.toList());
```

- **For the moment we will only use Collectors.toList**
  - ...many more Collector implementations are available!

As explained before, once the intermediate operations on a **Stream** have been performed, a terminal operation will execute the operations in the pipeline and ‘assemble’ a result.

One terminal operation that will be often is the **Stream’s collect** method. This method takes a **Collector** instance as parameter. An instance of the **Collector** interface is responsible for accumulating the elements in a stream and place these in a non-stream result type. The **Collectors** class contains several factory methods for obtaining instances of the **Collector** interface. For the moment we will use the **Collectors.toList** method to obtain a collector which accumulates the elements within the **Stream** into a **List** object.

## Collectors Static Import

- Collectors methods often defined using static import

```
import static java.util.stream.Collectors.toList;  
  
allFlights.stream()  
    .filter(Flight::isInternational)  
    .collect( toList() );
```

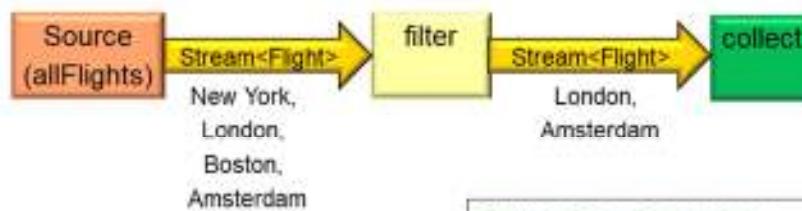
- Examples in this lesson use static import of toList

Since the factory methods in the **Collectors** class are defined as static, examples in this lesson will make use of a static import.

## Filtering

- The filter method accepts a Predicate
  - ‘Skips’ every element that does not comply to predicate

```
List<Flight> internationalFlights = allFlights.stream()
    .filter(Flight::isInternational)
    .collect(toList());
```



```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    ...
}
```

The intermediate filter method takes a **Predicate** as parameter. As a result all elements that do not comply to the criteria defined by the predicate will be skipped.

In the example shown above, the predicate uses a method reference to the **isInternational** method of the **Flight** object. As a result only those flight instances for which this method return true will pass through the filter.

# Truncating

- Limit the amount of elements in the Stream

```
allFlights.stream().filter(Flight::isInternational)
            .limit(1)
            .collect(toList());
```

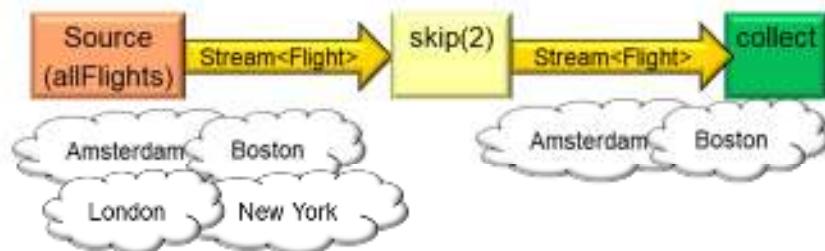


When you are only interested in a limited amount of elements from the stream, the intermediate limit operation can be used to restrict the amount of items in the pipeline

# Skiping Elements

- Skip a number of elements in the Stream

```
allFlights.stream().skip(2)  
    .collect(toList());
```

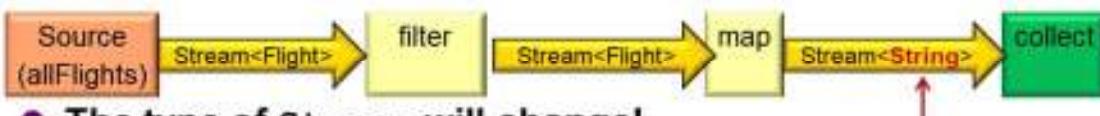


As the name already implies, the skip operation can be used to skip a certain amount of elements within the stream.

# Mapping

- The **map** method can be used to obtain a ‘subset’ of the information in the Stream
  - Accepts a **Function** parameter
  - e.g. You only want the destination, not the entire **Flight** object

```
allFlights.stream().filter(Flight::isInternational)
            .map(Flight::getDestination)
            .collect(toList());
```



- The type of Stream will change!

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    ...
}
```

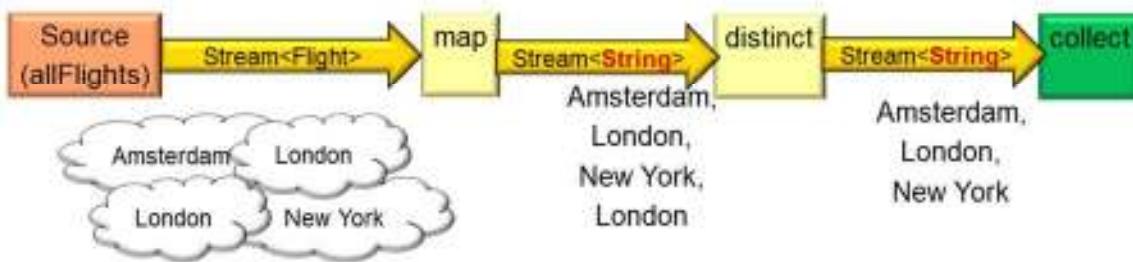
Often a **Stream** will contain elements (objects) that contain a large set of information, but we are only interested in a subset of the properties within these elements, similar to selecting only a few columns of a database table.

By using the **map** method, the elements in the ‘input’ **Stream** are mapped to a different type. In the example shown above, we are only interested in the destination property of the **Flight** object, so the map method will map the value (which is of type **String**) of the property to a new **Stream**. So the result of the map method will now no longer be a **Stream** of **Flight** objects, but a **Stream** of **String** objects.

## Filter Using `distinct`

- You can filter the stream to avoid duplicate values
  - Using `distinct()`

```
allFlights.stream().map(Flight::getDestination)
            .distinct()
            .collect(toList());
```



We have seen that the `filter` method can be used to filter the contents of a **Stream**. A different kind of filter is the `distinct` method. By adding this intermediate operation to the pipeline, duplicates will be filtered from the **Stream**.

# Matching

- Stream API provides several methods for checking contents of Stream
    - Checking if element matches certain criteria
    - allMatch, anyMatch, noneMatch

```
boolean b = allFlights.stream().anyMatch(Flight::isInternational);
```

  - Terminal operation
- Operations referred to as short-circuiting
    - Entire stream does **not** have to be processed to produce result

Until now, we have been ‘manipulating’ the contents of the stream. The Stream API also contains methods for checking the contents of a stream to see whether elements in the stream meet a criteria.

The **allMatch**, **anyMatch** and **noneMatch** methods return a **boolean** (terminal operations), but more importantly they are also short-circuiting operations. This means that the stream does not have to be processed completely before these methods can return. Just like the logical and (**&&**) and or (**||**), once part of the expression has been evaluated and does not meet the criteria, the remaining part of the expression is not evaluated.

## Finding Elements

- The **findFirst**, **findAny** methods return a single element from the Stream

```
allFlights.stream().filter(Flight::isInternational).findFirst();
```

- **findAny** returns an element from the stream

- ◆ Faster when using parallel operations

- Return value is an instance of **Optional**

- Element might not be present in stream

Instead of matching to check if the stream contains the element you are looking for, the **findFirst** and **findAny** methods return a single element from the stream.

However, these methods return an instance of **Optional**. When the stream does not contain an element, an empty instance of **Optional** is returned. When the stream does contain an element, this element will be ‘wrapped’ within the **Optional** object.

## Optional Overview

- **java.util.Optional acts as container for objects**
  - Might or might not contain value
- **Used instead of returning null**
- **Content can explicitly be checked before being used**

```
Optional<String> optional = ...  
  
if(optional.isPresent()) {...}  
  
optional.ifPresent( System.out::println ); //Consumer  
  
String string = optional.get(); //NoSuchElementException if empty  
  
String otherwise = optional.orElse("default");
```

- **More on Optional is covered in a separate lesson**

The **Optional** class will be covered in more detail in a separate lesson. For the moment you should remember that **Optional** is used as an alternative for returning a **null** pointer.

**Optional** acts as a container for objects and might, or might not, contain a value. The **findFirst** method of the Stream API returns an instance of **Optional**. So, instead of checking for **null** values to be returned when the stream does not contain elements, you use the methods on the **Optional** class to check if it contains a value.

# Stream ‘Sources’

- Streams can be obtained from different sources

- Arrays

```
String[] data = ...  
Stream<String> stream = Arrays.stream(data);
```

- Files

- ◆ Processing each line in a text file**

```
Stream<String> lines = Files.lines(path,  
Charset.forName("ISO-8859-1"));
```

- ‘Simple’ Values

```
#flight flight1 = new Flight("Paris", "Amsterdam", 62);  
Flight flight2 = new Flight("Amsterdam", "New York", 467);  
  
—  
Stream<Flight> flights = Stream.of(flight1, flight2, flight3, flight4);
```

- Empty Streams

```
Stream<String> stream = Stream.empty();
```

As mentioned at the beginning of the lesson, streams can be obtained for a variety of sources. Until now we only looked at **Collection** implementations as source of the **Stream**, but you can also create a stream based upon an array, or zero or more object references.

Another option is obtaining a stream from a **File**. In Java 8 (among other changes), the `lines` method has been added to the **Files** class. Streams can be created where each line in a text-file becomes an element in a **Stream**.

## Numeric Streams

- Stream API defines specialized streams for working with primitive types
  - IntStream, DoubleStream, LongStream
  - Avoid (un)boxing overhead
  - Add additional methods for working with numeric values

```
IntStream mapToInt =  
    flights.mapToInt(Flight::getFlightTime);  
  
int sum = mapToInt.sum();  
OptionalDouble average = mapToInt.average();
```

Until now, all streams that we have used deal with objects, so when the information in the stream is made up of numeric values, these values are represented by their **Object** representation (**Integer**, **Long**, **Double**). As a result, each primitive type that is obtained from an object needs to be boxed before it can be put into the **Stream** and unboxed when obtained from the stream.

The Stream API defines specialized streams for dealing with these primitive types. Besides the fact that these implementation do not require the values to be (un)boxed, they also provide some additional methods specialized for working with numeric primitive types. Methods like sum and average only make sense when working with numeric types. (The average Flight is not very interesting)

## Static Methods on Numeric Streams

- **Numeric stream interfaces contain static methods for obtaining instances**
  - **Of one or more values**

```
DoubleStream dStream = DoubleStream.of(1.2, 3.6, 7.5);
```
- **IntStream and LongStream can be created by defining ranges**

```
IntStream intStream = IntStream.range(1, 1024); // 1...1023
IntStream intStream2 = IntStream.rangeClosed(1, 1024); //1...1024
```

Just like with streams that deal with objects, numeric streams can be created using static methods from the interface, so a stream of double value can be constructed by using the 'of' method.

In addition, instances of **IntStream** and **LongStream** can also be constructed using the static '**range**' and '**rangeClosed**' methods. By supplying the start and end values of the range, an instance of the stream is created containing all the numeric values within that range.

## Reducing

- Reducing/Combining elements in a Stream into a single value
  - `average`, `min`, `max`, `count`, `sum`
  - `collect(...)`, `reduce(...)`
- `reduce(...)` accepts
  - Initial value (optional)
  - `BinaryOperator` to combine two values

```
flights.map(Flight::getFlightTime)
        .reduce(0, (time1, time2) -> time1 + time2);
```

- Same could be accomplished by using numeric stream
  - And the `sum` method

```
flights.mapToInt(Flight::getFlightTime).sum();
```

Reducing is the operation that ‘reduces’ all the elements in a stream into a single value. Besides existing operations like `min`, `max` (etc.) you can also define your own reducing operations by using the `reduce` method.

The `reduce` method accepts a **BinaryOperator** that should combine two elements into a single value, optionally providing an initial value.

## forEach Terminal Operation

- Instead of collecting results in a collection, `forEach` can be used to consume every element in Stream

```
allFlights.stream()
    .forEach(System.out::println);
    ^_____
    |       |
    |       Consumer
```

During the exercise you might want to display the content of a **Stream**, you might be tempted to create a **List** and write a for loop containing `System.out.println` statements. Well that's exactly what we are trying to avoid by using streams. No more external iteration over collections, let the stream building blocks take care of it internally.

```
allFlights.stream()
    .forEach(System.out::println);
```

## Lesson Review and Summary

- 1. Standard collections classes thread-safe**
  - a. True
  - b. False
- 2. What is declarative programming?**
- 3. Does the Stream API replace Collections?**
- 4. What are the differences between an intermediate and terminal operation?**
- 5. Which operations would you use to obtain a unique list of destinations from a large set of Flight objects. (destination being a property of Flight)**
- 6. Name (at least) three sources that can be used for a Stream**
- 7. What method would you use to locate a single element in a stream?**

- 1) false, the standard collections classes or not thread safe, however thread-safe versions can be created by using methods from the Collections class
- 2) Thinking of the program logic in terms of what needs to be done, instead of how it needs to be implemented
- 3) NO! Collections of data are a source of data that can be processed using the Stream API
- 4) Intermediate operations return a stream and are not executed immediately. Terminal operations return non-stream values and cause the intermediate operations in the pipeline to be executed
- 5) Stream, map, distinct, collect
- 6) Collections, Files, 'simple' values
- 7) find or findAny

## Exercise 20: Working with Streams

`~/StudentWork/code/Streams/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Collectors

Introduction to Lambda Expressions  
Java 8 Collection Updates  
Streams  
**Collectors**

## Lesson Agenda

- Using different ways to collect the items from a Stream
- Grouping elements within a stream
- Gathering statistics about numeric property of elements in a stream

## Introduction to Collectors

- Collectors ‘transform’ elements in a **Stream** into a mutable result
- Accumulating elements into a collection
- Grouping elements by element property
  - e.g. flights by destination
- Summarizing elements into a single value
  - Minimum/maximum value, average, count
- Partitioning elements
  - Split stream in parts depending on criteria

Intermediate operations can be defined on a stream of elements to define the operations that need to take place on the elements within the stream, creating a pipeline of operations.

A terminal operation can perform operations on a stream, but also creates a mutable result. The **collect** method of the **Stream** class accepts a parameter of type **Collector**. Different implementations of the **Collector** interface are capable of collecting elements in a **Stream** into different types of objects.

Often, the result of the intermediate operations on a **Stream** will be collected into a **Collection** implementation, but other operations that might be defined by the collector are grouping, summarizing and partitioning of elements, as will be explained in this lesson.

## Collectors and Collector

- The **Collectors** class contains extensive list of factory methods
  - Returning commonly used **Collector** implementations

| Factory methods of <b>Collectors</b> class |                        |                     |
|--------------------------------------------|------------------------|---------------------|
| averagingDouble(...)                       | averagingInt(...)      | averagingLong(...)  |
| collectingAndThen(...)                     | counting()             | groupingBy(...)     |
| groupingByConcurrent(...)                  | joining(...)           | mapping(...)        |
| maxBy(...)                                 | minBy(...)             | partitioningBy(...) |
| reducing(...)                              | summarizingDouble(...) | summarizingInt(...) |
| summingDouble(...)                         | summingInt(...)        | summingLong(...)    |
| toConcurrentMap(...)                       | toList()               | toMap(...)          |
| toSet()                                    |                        |                     |

- Custom collector implementations can be created

The **Collectors** class contains a list of factory methods that can be used to obtain implementations of commonly used **Collector** implementations. This lesson covers the use of some of these collectors. Do keep in mind that a lot of overloaded versions of these methods are available on the **Collectors** class. If you still can't find one that provides you with the logic you need, you can always write your custom collector by implementing the **Collector** interface yourself.

## Creating collection(s) from Stream

- The contents of a Stream can be collected into a collection implementation

- To a `java.util.List`

```
Stream<Flight> stream = ...
List<Flight> list = stream.collect(toList());
```

- To a `java.util.Set`

```
Set<Flight> set = stream.collect(toSet());
```

◆ removing duplicate elements!

- Or a specific collection implementation

```
TreeSet<Flight> treeSet = stream.collect(toCollection(TreeSet::new));
   ^_____
   Supplier
```

◆ Supplier must be defined

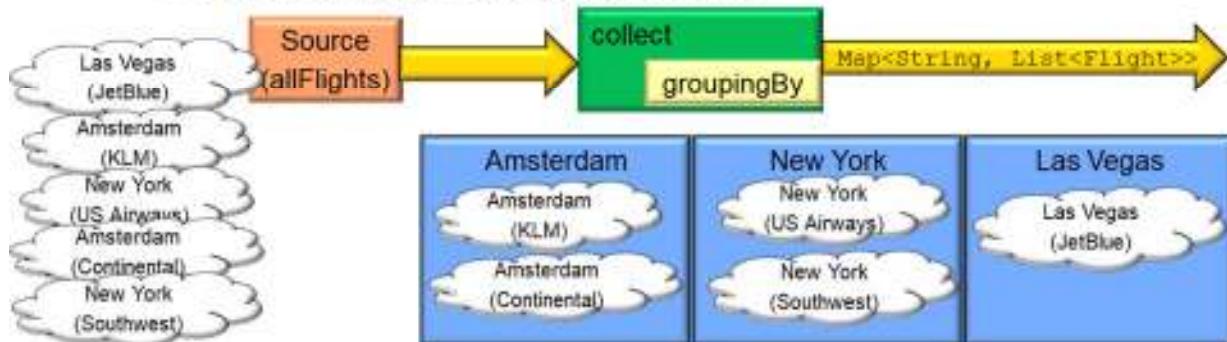
The contents of a stream can be collected into a collection implementation. Convenience factory methods for collecting into a `List` or `Set` implementation are available, but it is also possible to specify the specific collection implementation you need (using a constructor reference)

## Grouping with Streams

- The **groupingBy** collector groups elements within Stream

```
List<Flight> allFlights = FlightDAO.getAllFlights();  
Map<String, List<Flight>> flightsByDestination =  
    allFlights.stream()  
        .collect(groupingBy(Flight::getDestination));  
Function
```

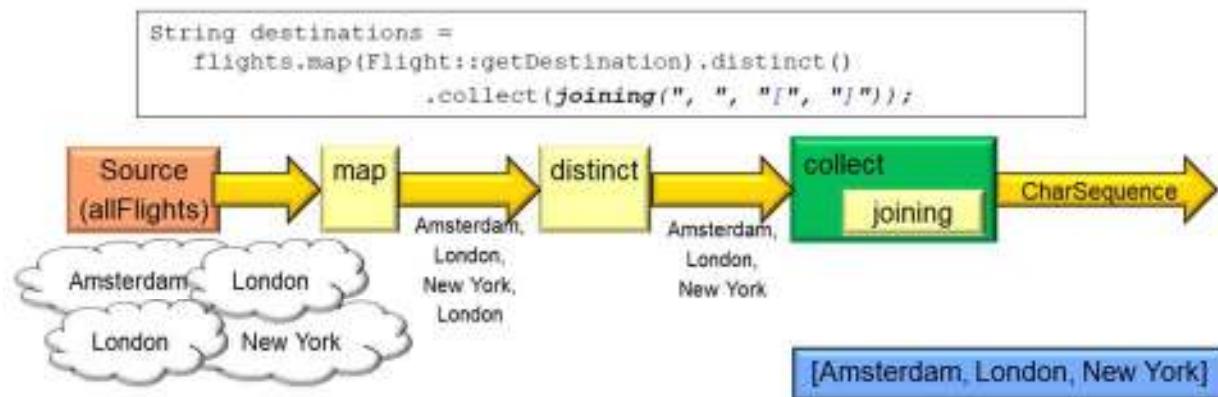
- Function defines grouping criteria



The **groupingBy** collector requires a **Function** that specifies the criteria on which the grouping should take place. In the example, the destination value of the flight object is read by using the method reference and elements are being grouped by the result of this method invocation.

## Joining Elements

- **Concatenate elements into a String (CharSequence)**
  - Optionally providing separator, prefix and suffix



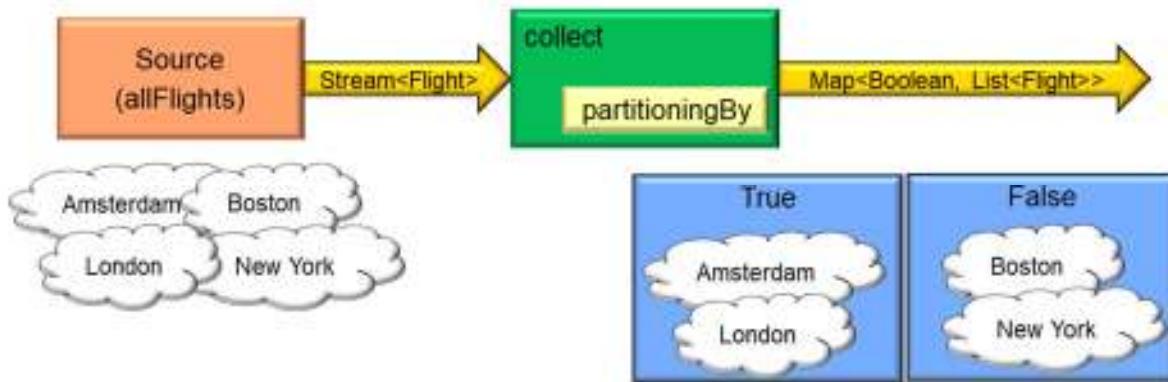
Joining allows for the joining of all the elements in the stream into a **String** value, specifying the separator that is placed in-between the elements and the prefix and suffix of the final result.

The example shown above creates a comma separated list of all the (distinct) destinations for which flights are scheduled.

# Partitioning

- Partition elements in Stream into two groups
  - According to given Predicate

```
Map<Boolean, List<Flight>> result =  
    stream.collect(partitioningBy(Flight::isInternational));
```



Partitioning is basically a group-by operation, but this time the result will be only two lists. One list of elements that conforms to the predicate of the operation and one list of elements that don't

# Counting and Summing

- Counting elements in a Stream

```
long count = flights.collect(counting());
```

- Sum (numeric) property of elements in Stream

```
Integer total= flights.collect(summingInt(Flight::getFlightTime));
```

- Collect statistics about (numeric) property of elements in Stream

```
IntSummaryStatistics stats = flights.collect(  
    summarizingInt(Flight::getFlightTime));  
  
double average = stats.getAverage();  
long count = stats.getCount();  
int max = stats.getMax();  
int min = stats.getMin();
```

We have seen the counting operation earlier. It counts the amount of elements available in the stream(s).

Collectors for summing numeric values within a stream are also available by default. When you are interested in more ‘statistics’ information about numeric value in the stream you might want to consider the **summarizingXXX** collectors. The result object not only contains the sum of the numeric value in the stream, but also the max, min and average.

## Lesson Review and Summary

- 1. What are collectors?**
- 2. What is the difference between groupingBy and partitioningBy?**

- 1) Implementations of the Collector interface, used to gather the elements in a stream into a mutable value
- 2) Grouping by divides the content into 1 or more groups, partitioningBy divides the element in 1 or 2 groups (depending on whether the element meets criteria)

## Exercise 21: Collecting

`~/StudentWork/code/Collecting/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

## **Session: Supplemental Materials (Time Permitting)**

**The new Date/Time API  
Formatting Strings  
Java Data Access JDBC API  
Unit Testing Fundamentals  
Jumpstart: JUnit 4.x  
Supplemental Exercises**

## **Lesson: The new Date/Time API**

**The new Date/Time API**

Formatting Strings

Java Data Access JDBC API

Unit Testing Fundamentals

Jumpstart: JUnit 4.x

## Objectives: Date and Time API

This lesson covers the Date/Time API that came out with Java 8. Specifically, it covers:

- The core Date/Time classes
- Formatting date and time values for presentation
- Modifying dates using the API
- Implementing applications that use Time-Zone information

## What's Wrong With `java.util.Date`?

- Most of the methods are deprecated
  - Since JDK 1.1
- Poor API design
  - Months start at 0?
- Resembles a moment on a timeline
- Classes are not thread-safe
  
- `java.util.Calendar` provided some relief

The **Date** class was introduced in Java 1.0, but already in Java 1.1 most of the methods were declared as deprecated. Right from the start it became clear that the **Date** class was poorly designed and difficult to use in applications. One common mistake made by developers was caused by the fact that the first month of the year was represented by the integer value 0 instead of 1.

Also, the name of the class implies that you are working with a particular date, where in fact you are working with a moment on a timeline, expression in both date and time.

On top of this, the **Date** class was not implemented to be thread safe. Developers had to write their own thread synchronization code in order to resolve concurrency issues.

The **Calendar** class was introduced in JDK 1.1 to resolve some of the common issues when working with date values in Java, but unfortunately, most of the issues remained.

## java.time

- A new date/time API is introduced in Java 8
  - Similar to the Joda Time API
- Classes are implemented to be thread-safe
  - Core-classes of API are immutable
- Domain-Driven design
  - API provides different classes for different use cases
- Fluent interface design
  - Most methods do not accept or return null values

Java 8 introduced the new Date/Time API. Modelled after the Joda Time API (the author of Joda-Time, Stephen Colebourne, worked closely with Oracle during the development of the API), the new Date/Time API was designed with all the problems of the **Date** class in mind.

All classes of the Date/Time API are designed to be thread-safe.

Classes in the API are immutable, meaning that operations on an object result in a new object instance to be created, instead of affecting the state of the current object. Also, most methods do not accept null-pointers as input parameters and do not return null pointers, making them perfect candidates for implementations that follow the fluent interface design.

Another big change, when working with date time values in Java, is the domain-driven design of the API. The API provides several implementations, each one focussed on a particular use-case.

Birthdays can now be defined using a type that only resembles a date (where **java.util.Date** would always also contain a time).

Developers can now choose between implementation classes, depending on whether they need timezone information in their application.

## Core Classes of Date/Time API

| Class or Enum  | Year | Month | Day | Hours | Minutes | Seconds | Zone Offset         |
|----------------|------|-------|-----|-------|---------|---------|---------------------|
| LocalDate      | ✓    | ✓     | ✓   |       |         |         |                     |
| LocalDateTime  | ✓    | ✓     | ✓   | ✓     | ✓       | ✓       |                     |
| LocalTime      |      |       |     | ✓     | ✓       | ✓       |                     |
| Year           | ✓    |       |     |       |         |         |                     |
| YearMonth      | ✓    | ✓     |     |       |         |         |                     |
| Month          |      | ✓     |     |       |         |         |                     |
| MonthDay       |      | ✓     | ✓   |       |         |         |                     |
| ZonedDateTime  | ✓    | ✓     | ✓   | ✓     | ✓       | ✓       | ✓<br>(incl zone ID) |
| OffsetDateTime | ✓    | ✓     | ✓   | ✓     | ✓       | ✓       | ✓                   |
| OffsetTime     |      |       |     | ✓     | ✓       | ✓       | ✓                   |

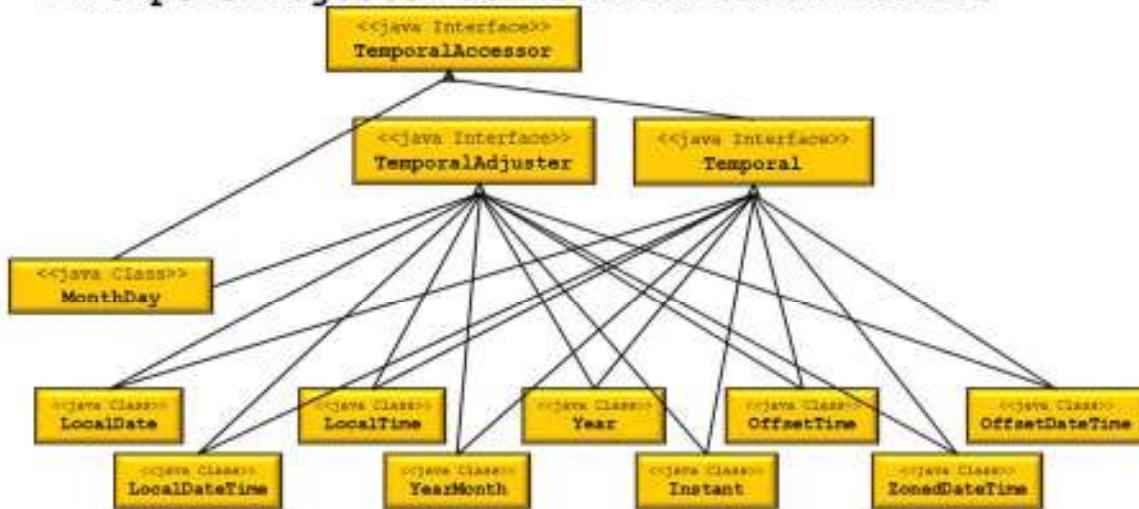
- All classes follow ISO-8601 calendar system

The overview above show the most used classes of the Date/Time API.

As you can see, you can choose the appropriate class depending on the date/time information you need for the use-case you are implementing.

## Date/Time Hierarchy

- **TemporalAccessor** defines read-only access
- **Temporal** defines read/write access
- **TemporalAdjuster** defines modification methods



The diagram above shows a class diagram of some of the most commonly used classes of the Date/Time API. As you will see throughout this lesson, many of the methods accept any of the defined interface types.

**TemporalAccessor:** Defines the read-only access of a temporal object

**Temporal:** Defines read/write access of a temporal object

**TemporalAdjuster:** Defines the methods for modifying temporal objects

## DayOfWeek Enum

- Provides (localized) textual representation of days

```
for (DayOfWeek day : DayOfWeek.values()) {  
    System.out.println(day.getDisplayName(TextStyle.FULL, Locale.US));  
}
```

Monday  
Tuesday  
Wednesday  
Thursday  
Friday  
Saturday  
Sunday

- Contains methods for day operations

```
DayOfWeek monday = DayOfWeek.MONDAY;  
DayOfWeek other = monday.plus(5);  
System.out.println(other);
```

- Also contains int value (ISO-8601: Monday = 1)

◆ ordinal() method should **not** be used!

```
DayOfWeek.MONDAY.ordinal(); //returns 0  
DayOfWeek.MONDAY.getValue(); //returns 1
```

In addition to the classes, the Date/Time API also defines enumerations for both **DayOfWeek** and **Month**.

The **DayOfWeek** enum contains (besides the seven days of the week) methods to display a localized String representation of the day and methods to perform simple operations.

According to the ISO-8601 standard, Monday is the first day of the week. The **DayOfWeek** enum contains an int value for each day of the week, according to the ISO-8601 standard. To obtain this int value the **getValue** method should be used. The **ordinal** value of the constant **DayOfWeek.Monday** will return 0!

## DayOfWeek Enum (cont'd)

- Provides (localized) textual representation of days

```
DayOfWeek day = DayOfWeek.MONDAY;
Locale l = Locale.US;
String fullText = day.getDisplayName(TextStyle.FULL, l); // Monday
String shortText = day.getDisplayName(TextStyle.SHORT, l); // Mon
String narrowText = day.getDisplayName(TextStyle.NARROW, l); // M

Locale italian = Locale.ITALIAN;
String fullTextIt = day.getDisplayName(TextStyle.FULL, italian); // lunedì
```

The **DayOfWeek** enum contains methods to obtain a localized textual representation of day of the week and perform simple day related operations.

## Month Enum

- Provides (localized) textual representation of months

```
Month month = Month.DECEMBER;
Locale l = Locale.US;
String fullText = month.getDisplayName(TextStyle.FULL, l); // December
String shortText = month.getDisplayName(TextStyle.SHORT, l); // Dec
String narrowText = month.getDisplayName(TextStyle.NARROW, l); // D

Locale italian = Locale.ITALIAN;
String fullTextIt = month.getDisplayName(TextStyle.FULL, italian); // dicembre
```

- Contains methods for common month operations

```
Month march = Month.MARCH;
int dayNumber = march.firstDayOfYear(false); // returns 60
Month february = Month.FEBRUARY;
int max = february.maxLength(); // returns 29
int min = february.minLength(); // returns 28
Month month = february.minus(3); // return NOVEMBER
```

- Also contains int value (ISO-8601: January = 1)

Just like the **DayOfWeek** enum, the **Month** enum contains methods to obtain a localized textual representation of the month of the year and perform simple month related operations.

Just like the **DayOfWeek** enum, the **Month** enum also contains an int value for each month of the year. In this case, January is resembled by the value 1.

To obtain the numeric value of a month, the **getValue** method should be used and **not** the **ordinal** method (which would return 0)

## The LocalDate Class

- Resembles only date
  - For example a birthday
- Contains factory methods to construct instances
  - Current date (just the date, no time!)

```
LocalDate now = LocalDate.now(); // current date
```

- of( ... ) methods to construct specific date

```
LocalDate birthday = LocalDate.of(1976, Month.MARCH, 9); // 1976-03-09
```

- parse(...) methods for parsing String representations

```
LocalDate parsedDate = LocalDate.parse("2014-12-06");
```

◆ DateTimeFormatter instances allow for different date formats

```
DateTimeFormatter dtf = DateTimeFormatter.BASIC_ISO_DATE; // Format yyyyMMdd  
LocalDate date = LocalDate.parse("20141231", dtf); // 2014-12-31
```

During this lesson we will not cover each and every class of this API. Once you become familiar with one or two of these classes, the rest will be pretty straight-forward (after all, most of them implement the same interfaces).

One of the classes you, most likely, will use very often is the **LocalDate** class. This class represents just a date (compared to **java.util.Date**, this class does not contain time information), and would therefore be a prime candidate to hold values like birthdays, creation dates, etc.

The class (and other temporal classes of the API) contains **of** methods to construct an instance using a specific moment in time and a **now** method to obtain the current value.

The temporal classes also contain parse methods that can be used to construct an instance using String representation of the value.

## The LocalDate Class (cont'd)

- Contains several methods for obtaining 'information' about the date

```
LocalDate birthday = LocalDate.of(1976, Month.MARCH, 9);
DayOfWeek dayOfWeek = birthday.getDayOfWeek(); // TUESDAY
boolean leapYear = birthday.isLeapYear(); //true
```

- ...and date manipulations

```
LocalDate parsedDate = LocalDate.parse("2014-12-06");
LocalDate withDayOfMonth = parsedDate.withDayOfMonth(1); // 2014-12-01
LocalDateTime atTime = withDayOfMonth.atTime(12, 30); // 2014-12-01T12:30
```

- Resulting in new instances! (immutable objects)

Once an instance of **LocalDate** has been created, several methods are available to obtain information about the date like which day of the week the instance defines or if the date is within a leap year.

Furthermore, the date can be manipulated, using the **withXXX** methods or can be converted into one of the other Date/Time classes. For example, adding time information to the **LocalDate**, results in a **LocalDateTime** instance to be created.

Keep in mind that the instances are immutable, as a result all operations performed on the instances will result in a new object instance to be created.

## TemporalField and ChronoField

- **TemporalField interface defines a single date-time field**
- **ChronoField enum implements this interface**
  - Defines a standard set of Date/Time fields
- **Used for field-based access of Date/Time properties**

```
int day = LocalDate.now().get(ChronoField.DAY_OF_MONTH);
```

- **Date/Time instances can be checked to see if they 'contain' required field**

```
public void print(Temporal temporal) {  
    if(ChronoField.YEAR.isSupportedBy(temporal)) {  
        int year = temporal.get(ChronoField.YEAR);  
        ...  
    }  
}
```

- **UnsupportedTemporalTypeException is thrown when unsupported field is accessed**

The **TemporalField** interface defines a single Date/Time field within one of the classes. The **ChronoField** enum provides an implementation of this interface. (Custom implementations can be created when alternative calendar implements are required).

The individual Date/Time classes contain methods for obtaining fields of the instance. **LocalDate** contains methods like **getYear**, **getMonth**, **getDayOfWeek**, etc. These methods are just methods provided on the class. All instances of **TemporalAccessor** contain a **get(TemporalField t)** method that can be used to obtain the value of a field within the instance.

To check an instance to see if the instance contains the requested field, a **isSupportedBy** method is available of the **ChronoField** enum. When a requested field is not implemented by the instance, an **UnsupportedTemporalTypeException** is thrown (as shown by the last example, where an attempt is made to obtain hour information from a **LocalDate** instance).

## Instant Class

- The Instant class is one of the core classes of the API
  - Models a single point on a time-line
  - Similar to java.util.Date
- Represents number of seconds since UNIX epoch
  - Midnight of January 1st 1970 UTC
- Supports nanosecond precision
- Intended for use by a machine, not human presentation
  - (e.g. does not provide operations to obtain day/month information)

```
i.get(ChronoField.MONTH_OF_YEAR);
```



The Date/Time API defines a clear distinction between classes that are to be used for human presentation (**LocalDate**, **LocalDateTime**, **Month**, etc) and classes that are to be used by machines. The **Instant** class is one of the classes that is meant to be used by machines. Similar to **java.util.Date**, it represents a single point on a timeline (which starts at midnight of January 1st of 1970).

An **Instant** contains a long value that represents the amount of seconds since January 1st, 1970, but also an int value that contains the nanosecond of the current second.

(JavaDoc indicates that the maximum measurable time is limited by the maximum value that can be stored in a long, which is greater than the current estimated age of the universe)

## Duration Class

- Duration class defines a **time-based amount of time**
  - Measured in (nano) seconds
- Can be used to calculate difference between Temporals
  - Object must support the seconds unit

```
LocalDateTime start = LocalDateTime.of(2014, 1, 30, 12, 15); //2014-01-30 12:15
LocalDateTime end = LocalDateTime.of(2014, 1, 31, 18, 30); //2014-01-31 18:30
Duration d = Duration.between(start, end);
long seconds = d.getSeconds(); //108900
```

- The Period class defines a **date-based amount of time**

```
LocalDate java1 = LocalDate.of(1996, 1, 23);
LocalDate java8 = LocalDate.of(2014, Month.MARCH, 18);
Period p = Period.between(start, end);
int years = p.getYears(); // 18
int months = p.getMonths(); // 1
int days = p.getDays(); // 23
//Period between Java 1 and Java 8 is: 18 years, 1 month and 23 days
```

In addition to classes that represent a moment in time, the Date/Time API also contains classes that can be used to obtain (or define) the duration between the **Temporal** instances (e.g. Difference in time)

However, the two temporal instances that are compared should be both of the same ‘type’. (either both the human readable version, or both machine times)

Where **Duration** is used to define or calculate duration of time, the **Period** class is used to calculate or define differences in dates.

For example, Java 1 was introduced on January 23rd, 1996 and Java 8 was introduced on March 18th, 2014. Using the **between** factory method of the **Period** class a **Period** instance is created which can be used to show that it took them 18 years, 1 month and 23 days to get from Java 1 to the Java 8 Date/API shown here...

# Manipulating Dates

- API provides several methods for date/time manipulations
  - Creating new (immutable) objects

```
LocalDate date = LocalDate.of(2012, 2, 29); // 2012-02-29 Leap Year!
LocalDate withDayOfMonth = date.withDayOfMonth(1); // 2012-02-01
LocalDate withMonth = date.withMonth(1); // 2012-01-29
LocalDate withYear = date.withYear(2013); // 2013-02-28 (!!!)
LocalDate plusYears = date.plusYears(1); // 2013-02-28 (!!!)
```

- Manipulations can be concatenated (Fluent API)

```
LocalDate date = LocalDate.of(2012, 2, 29); // 2012-02-29
LocalDate startOfNextYear = date.plusYears(1).withDayOfYear(1);
```

As mentioned earlier, classes of the Date/Time API contain methods allow for manipulation of the value. To be exact, it allows for new instances to be created containing the result of performing the ‘manipulation’ of the given date or time. Performing these manipulations not only result in new instances (instances are immutable) the result will also be a valid date. So adding one year to February 29, 2012 will result in February 28, 2013!

Remember the Fluent API design goal of the API? The manipulation methods are a prime example of this. Allowing multiple methods to be chained results in code that is more readable.

## TemporalAdjuster Interface

- **TemporalAdjuster implementations allow for more complex date operations**
  - e.g. Third Wednesday in March
  - **TemporalAdjusters class provides factory methods for most commonly used adjusters**
- **Custom implementations can be created**
- **TemporalAdjuster is a functional interface**
  - Allows for the use of Lambda Expressions

So far, the operations (manipulations) that can be performed of the temporal instances were pretty straightforward. When more complex date operations are required, instances of the **TemporalAdjuster** interface can be used.

The **TemporalAdjusters** class contains a variety of factory methods to obtain some of the most commonly used adjusters (as seen later), but custom implementations can also be created by implementing this interface.

Since the **TemporalAdjuster** only contains a single abstract method, this interface is a functional interface. As a result the adjuster logic can also be defined using a Lambda Expression

## Using the TemporalAdjusters Class

```
import static java.time.DayOfWeek.*;
import static java.time.temporal.TemporalAdjusters.*;
...

LocalDate date = LocalDate.of(2014, 11, 30);
date.getDayOfWeek(); // SUNDAY

LocalDate d1 = date.with( dayOfWeekInMonth(2, MONDAY) );
// Second Monday in November: 2014-11-10
LocalDate d2 = date.with( firstDayOfMonth() ); // 2014-11-01
LocalDate d3 = date.with( firstDayOfNextMonth() ); // 2014-12-01
LocalDate d4 = date.with( firstDayOfNextYear() ); // 2015-01-01
LocalDate d5 = date.with( firstDayOfYear() ); // 2014-01-01
LocalDate d6 = date.with( firstInMonth(MONDAY) ); // 2014-11-03
LocalDate d7 = date.with( lastDayOfMonth() ); // 2014-11-30
LocalDate d8 = date.with( lastDayOfYear() ); // 2014-12-31
LocalDate d9 = date.with( lastInMonth(FRIDAY) ); // 2014-11-28
LocalDate d10 = date.with( next(WEDNESDAY) ); // 2014-12-03
LocalDate d11 = date.with( nextOrSame(SUNDAY) ); // 2014-11-30
LocalDate d12 = date.with( previous(MONDAY) ); // 2014-11-24
LocalDate d13 = date.with( previousOrSame(WEDNESDAY) ); // 2014-11-26
```

The code snippet above shows methods of the **TemporalAdjusters** class. Notice the use of the static import statement.

As you can see the **TemporalAdjusters** class contains several methods to obtain date information in relation to an existing **Temporal** instance.

# Creating Custom Adjusters

- Custom adjuster classes can be created

```
public class LeapYearAdjuster implements TemporalAdjuster {  
  
    @Override  
    public Temporal adjustInto(Temporal temporal) {  
        Year year = Year.from(temporal).plusYears(1); // add at least 1 year  
        while (!year.isLeap()) {  
            year = year.plusYears(1); // Year is immutable!!!  
        }  
        return temporal.with(ChronoField.YEAR, year.getValue());  
    }  
}
```

- ...and used

```
LocalDate date = LocalDate.of(2012, 2, 29);  
LocalDate nextLeapYear = date.with(new LeapYearAdjuster()); //2016-02-29
```

When the **TemporalAdjusters** class does not provide an implementation that suits your needs, a custom implementation can be created by creating an implementation of the **TemporalAdjuster** interface.

## Formatting Date/Time

- **java.time.format.DateTimeFormatter can be used to format TemporalAccessor instances**
  - Thread-safe (**java.text.DateFormat** is not)
  - Provides factory methods for commonly used formats

```
DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
String s = formatter.format(LocalDateTime.of(2014, 12, 31, 23, 59));
//2014-12-31T23:59:00Z
```

- ..or can be configured using patterns

```
DateTimeFormatter formatter = DateTimeFormatter
        .ofPattern("dd MMMM uuuu hh:mm");
String s = formatter.format(LocalDateTime.of(2014, 12, 31, 23, 59));
//31 december 2014 11:59
```

- **The **DateTimeFormatterBuilder** class can be used to construct complex formatters**

Naturally, when working with human-readable Date/Time information, there will come a time that the value needs to be presented. The **DateTimeFormatter** class is used to format instances of **TemporalAccessor** into a string representation. Again, factory methods have been provided for the commonly used formatting patterns. When custom formats are required, these formats can be defined using the **ofPattern** method.

When the format to be used can not be defined using a 'simple' pattern, the **DateTimeFormatterBuilder** class can be used to construct all kinds for complex formatters.

## Working With Time Zones

- **java.util.ZoneId deals with time-zones**
  - Replaces `java.util.TimeZone`
  - Better support for Daylight Savings Time
  - Consists of a set of `ZoneRules` for given `TimeZone`
- **ZonedDateTime instances can be created**
  - By supplying `ZoneId` to `LocalDate`, `LocalDateTime` or `Instant`

```
ZoneId ams = ZoneId.of("Europe/Amsterdam");
ZoneId nyc = ZoneId.of("America/New_York");
LocalDateTime ldt = LocalDateTime.of(2014, JUNE, 30, 14, 00);

ZonedDateTime atZoneAms = ldt.atZone(ams);
//2014-06-30T14:00+02:00[Europe/Amsterdam]
ZonedDateTime atZoneNyc = atZoneAms.withZoneSameInstant(nyc);
//2014-06-30T08:00-04:00[America/New_York]
```

- ...or by using one of the factory methods

Writing applications that have to take time-zones into consideration is never easy. Luckily the Date/Time API provides some relief by defining **Temporal** implementations that deal specifically with timezones.

Defining timezone information can be done using the new **ZoneId** class (which replaces the existing **TimeZone** class). The new class promises better support for Daylight Savings Time, but more importantly, works closely with the **ZonedDateTime** and **OffsetDateTime** classes of the Date/Time API.

Instances that take time-zone information into account can easily be constructed from temporal classes that do not contain time-zone information, as shown in the code snippet above.

Once a time-zone aware instance has been created, methods are available to create instances of the same time, but for a different time-zone

## Working With Time Zones (cont'd)

- **ZoneOffset** expresses offset from UTC/Greenwich
  - Extends **ZoneId**

```
LocalDateTime ldt = LocalDateTime.of(2014, JUNE, 30, 14, 00);
ZoneOffset offsetAMS = ZoneOffset.ofHours(+2);
ZoneOffset offsetNYC = ZoneOffset.of("-04:00");
OffsetDateTime atZoneAms = OffsetDateTime.of(ldt, offsetAMS);
// 2014-06-30T14:00+02:00
OffsetDateTime atZoneNyc = atZoneAms.withOffsetSameInstant(offsetNYC);
//2014-06-30T08:00-04:00
```

Often time-zones are defined as an offset from the UTC/GreenWich time. The **ZoneOffset** class is a subclass of **ZoneId** and can be constructed by defining the offset.

## Legacy Date Code

- The **java.util.Date** and **java.sql.Date** classes have been updated to support conversion to and from Date/Time API

```
Date date = new Date();
Instant instant = date.toInstant();
Date convert = Date.from(instant);
```

- Instances of **Calendar** can also be converted into an **Instant**

```
Calendar calendar = Calendar.getInstance();
Instant instant = calendar.toInstant();
```

- **GregorianCalendar** instances can even be
  - Converted to **ZonedDateTime** instance

```
GregorianCalendar gCal = new GregorianCalendar();
ZonedDateTime zonedDateTime2 = gCal.toZonedDateTime();
```

- Constructed using **ZonedDateTime** instance

```
GregorianCalendar.from(zonedDateTime);
```

Naturally you will have to deal with a lot of existing code and libraries that still rely on the **java.util.Date**, **java.sql.Date** and **java.util.Calendar** classes and their concrete implementations.

In Java 8, methods have been added to these classes to convert these types into instances of classes defined by the Date/Time API. Keep in mind that both **Date** and **Calendar** contain date and time information as well as **TimeZone** information. As a result the conversion will initially result in the creation of an instance that also holds all this information (Instant or **ZonedDateTime**).

The **ZonedDateTime** class in turn, contains several methods to obtain instances that do not contain time-zone information (e.g. **LocalDate**)

## Lesson Review and Summary

### 1. What will be the output of the following code?

```
LocalDate date = LocalDate.of(2004, Month.FEBRUARY, 29);  
date.plusDays(1);  
System.out.println(date);
```

### 2. What is a TemporalAdjuster?

### 3. When would you use Instant instead of LocalDateTime

- 1) 2004-02-29 (LocalDate is immutable)
- 2) The interface that can be used to provide implementations to allow for modification of dates
- 3) Instant is intended for machine operation,  
LocalDateTime is intended for Human-presentation

## Exercise 22: Agenda

`~/StudentWork/code/Agenda/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Formatting Strings

The new Date/Time API  
**Formatting Strings**  
Java Data Access JDBC API  
Unit Testing Fundamentals  
Jumpstart: JUnit 4.x

## Objectives: String Formatting

This lesson covers how to format strings. Specifically, it covers

- Format a String using the formatter syntax
- Apply text formatting using the Formatter class
- Use String.format and System.out.printf

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

# java.util.StringJoiner

 Java 8

- Construct a sequence of characters
  - Separated by a delimiter
  - Starting with a prefix (optional)
  - Ending with a suffix (optional)

**Methods:**

```
StringJoiner add(CharSequence newElement)  
int length()  
StringJoiner merge(StringJoiner other)  
StringJoiner setEmptyValue(CharSequence emptyValue)  
String toString()
```

```
StringJoiner joiner = new StringJoiner(", ", "[", "]");  
joiner.add("a").add("b");  
System.out.println(joiner.toString()); // output: [a,b]
```

Java 8 introduced the **StringJoiner** class providing developers with an easy way to create a **String** that consists of a delimited list of character values.

## printf

- **Java 5 introduced printf functionality**
  - Inspired by C's String formatting options
- **Allows for formatted data to be send to output**
  - Taking care of layout, alignment, formatting and type conversion
- **Data is defined by 'format-string'**
  - Containing literal characters and format specifiers

```
String template = "Thank you %s for placing this order. Your order ID is: %d";
```

Inspired by C's String formatting options, Java 5 introduced a **printf** functionality, allowing for the creation of formatted text. Often we need to create text which consists of template text, combined with variable values.

The **printf** functionality allows us to do just that. The method takes a 'format string' which consists of a combination of literal text and format specifiers. Each format specifier acts as a placeholder for the 'live' value, defining not only the location of the variable but also how it should be formatted when inserted into the final text.

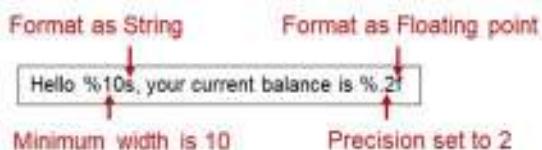
# Format Specifier Syntax

- Format-specifiers are defined using following syntax

**% [argument index \$] [flags] [width] [.precision] type suffix**

- Only the %-sign and type are mandatory

|                | Description                                                                  |
|----------------|------------------------------------------------------------------------------|
| %              | Special character indicating start of formatting instruction                 |
| argument index | The index of the argument value to insert. (defaults to order of definition) |
| flag           | Formatting instruction                                                       |
| width          | Minimum number of output characters                                          |
| precision      | Precision of floating point values                                           |
| type           | Type of object to be formatted                                               |
| suffix         | Additional conversion characters (Date/Time)                                 |



The format-specifiers that are to be placed in the template string consist of a %-sign followed by at least a type identifier (e.g. format as **String**, **decimal** or **date** type). Additional arguments allow for the definition of formatting/layout information (e.g. Padding left or right, minimum number of characters used or decimal digits to be displayed).

# Type Conversion Categories

- Valid conversions depend on type category
  - General : Can be applied to any type
  - Character : Any basic type representing Unicode char
    - ◆ Character, char, byte, short, ...
  - Integral : Any integral type
    - ◆ byte, short, int, long (and their wrapper classes)
    - ◆ BigInteger
  - Floating point: Any floating point type
    - ◆ float, double (and their wrapper classes)
    - ◆ BigDecimal
  - Date/Time: Types capable of encoding date or time
    - ◆ long, Calendar, Date and classes for the java.time package

The types that can be formatted are divided into several categories. The formatting types that can be applied depend on the category of the value to be formatted.

# Type Conversions

- The type defines conversion to be applied to the value
  - Conversions are Locale specific
  - Types defined in Uppercase result in Uppercase output

| Type | Description                    | Type | Description                                                  |
|------|--------------------------------|------|--------------------------------------------------------------|
| b B  | Boolean value                  | G    |                                                              |
| h H  | Hexadecimal String             | G    |                                                              |
| s S  | String representation of value | G    |                                                              |
| c C  | Unicode character              | C    |                                                              |
| d    | Decimal integer                | I    |                                                              |
| o    | Octal integer                  | I    |                                                              |
| x X  | Hexadecimal integer            | I    |                                                              |
|      |                                | e E  | Decimal number in computerized scientific notation           |
|      |                                | f    | Decimal number                                               |
|      |                                | g G  | Computerized scientific notation or decimal format           |
|      |                                | a A  | Hexadecimal floating-point number                            |
|      |                                | t T  | Date/Time conversion                                         |
|      |                                | %    | Literal '%'                                                  |
|      |                                | n    | Platform specific line separator, should use instead of '\n' |

- General types output 'null' when value is null
  - Type b and B will output 'false'

Depending on the category of the value to be formatted, one or more format-specifier letters can be selected to format the value. Keep in mind that all data is formatted according to a pre-defined locale. Unless a specific locale was defined, the system's default locale is being used. For example, when formatting floating point values, the decimal separator differs depending on the locale being used, while the internationalized names of months and days of the week might be used when formatting a value in the **Date** category.

Some type conversions are defined using both an lowercase and uppercase letter. When the uppercase letter is used, the result will also be generated in uppercase.

## The Formatter Class

- Interpreter for printf-style format strings
  - Implemented as fluent API
  - ◆ Invoking format returns reference to Formatter

```
Formatter fmt = new Formatter();
fmt.format("Hello %s", name)
    .format(" your current balance is %.2f\n", balance); // \n adds line-break
String result = fmt.toString();
```

- All formatting is Locale specific
  - Formatting locale can be defined
  - ◆ System locale is used by default

```
Formatter fmt = new Formatter(Locale.ITALIAN); //e.g. using decimal comma
```

The **Formatter** class acts as the interpreter for all printf-style strings. It allows for the template string to be interpreted and the format-specifiers to be replaced by their actual values. The **format** method of this class was implemented as a fluent API, returning a reference to the **Formatter** instance, allowing for multiple individual format operations to be defined, resulting in a single **String**.

When an instance of this class is created using the default constructor, the formatted text is appended to an internal **Appendable** instance (**StringBuilder**). An overloaded constructor allows for the definition of the **Locale** to be used for the formatting

## The Formatter Class (cont'd)

- Destination of output can be defined

- Output to Appendable instance

```
StringBuilder sb = new StringBuilder();
Formatter sbFormatter = new Formatter(sb);
```

- Defining File (or Filename) as output

```
try (Formatter formatter = new Formatter(new File("D:/bank.txt"));) {
    formatter.format("Hello %s, your current balance is %.2f", name, balance);
}
// or new Formatter("D:/bank.txt")
```

- Output to OutputStream

```
OutputStream os = ...
Formatter osFormatter = new Formatter(os);
```

- PrintStream

```
Formatter outFormatter = new Formatter(System.out);
```

- Overloaded constructors allow for definition of character set and Locale

Instead of using the default **Appendable** instance to 'hold' the result of the format operation, alternatives can be defined when creating an instance of the formatter class. Not only can an external **Appendable** (e.g. **StringBuilder**) be defined, it is also possible to have the instance write the result to a file, **OutputStream**, or **PrintStream**.

## System.out.printf / System.out.format

- PrintStream contains format and printf methods

- Both methods behave exactly the same way

```
float balance = 12.9f;  
System.out.printf("Current balance is %.2f",balance);  
// Current balance is 12.90
```

- Overloaded method allows definition of Locale

```
System.out.printf(Locale.ITALIAN, "Saldo corrente è %.2f", balance);  
// Saldo corrente è 12,90
```

- Methods do not add line-break!

- When needed should be added using %n

- ◆ Adds platform specific line separator

```
System.out.printf("Current balance is %.2f%n", balance);
```

The **printf** and **format** methods were added to the **PrintStream** class. Since the **System.out** references an instance of this class, it has now become possible to use the **printf** functionality when writing text to standard out.

The behavior of the **printf** and **format** methods is identical. In order to stay in sync with the other APIs (the method in the **Formatter** class and the other **print** methods of the **PrintStream**), both options are provided.

Keep in mind that the **printf** method does not add a line-break like the **println** method does. In order to add a line break, a **%n** character should be added to the text template. (**%n** adds a platform independent line-break)

## String.format

- **String class contains static format method**
  - Returns new formatting String

```
String fm = String.format("%s, your current balance is %.2f", name, balance);
```

- Optionally defining Locale

```
String fm = String.format(Locale.ITALIAN, "Saldo corrente è %.2f", balance);
```

A static ‘format’ method has been added to the **String** class. Invoking this method results in a new **String** object containing the output of the format operation

# Formatting a String

- **Formatting as String**

```
Person p = new Person();
//Lowercase - Uppercase - Person.toString
String format = String.format("%s %s", "abc", "def", p);
```

- **Defining the minimum width (padding on the left).**

```
String format = String.format("[%-7s]", "abc"); // [     abc]
```

- **Or with a flag (padding on the right)**

```
String format = String.format("%-7s", "abc"); // [abc     ]
```

- **Precision: Used to define maximum length of the string**

```
String format = String.format("%.3s", "abcdef"); // [abc]
```

The code samples above show some examples of how a string might be formatted. While %s formats the text ‘as-is’, %S formats the text in uppercase. When the variable to be formatted as a String is a non-String object, the **toString** method of this object is **null**. When the object reference is a null-pointer the text ‘null’ will be inserted into the **String**.

The minimum width of the string can also be defined as well as the padding that should be applied (left or right) when the provided values is shorter than the minimum width specified. By using the **Precision** notation (starting with a dot, followed by a numeric value), the maximum length of the string can be defined. When the length of the supplied value exceeds the maximum length, the value will be truncated.

# Formatting Numeric Types

## ● Formatting integer values

```
int integer = 6;
String fmt1 = String.format("%d", integer); // [6]
String fmt2 = String.format("%4d", integer); // [ 6] ...padding left
String fmt3 = String.format("%-4d", integer); // [6 ] ...padding right
String fmt4 = String.format("%04d", integer); // [0006] ... padded with zeros
```

## ● Formatting floating point values

### ■ Optionally defining number of decimal digits to be used

```
double floatVal = 19.5;
//Display as is
String fmt1 = String.format("%.2f", floatVal); // [19.50]
//Output at least 6 characters, 2 decimal digits
String fmt2 = String.format("%6.2f", floatVal); // [ 19.50] ...padding left
String fmt3 = String.format("%-6.2f", floatVal); // [19.50 ]...padding right
String fmt4 = String.format("%06.2f", floatVal); // [019.50] ...padded with zeros
floatVal = 3.141592;
String fmt5 = String.format("%.4f", floatVal); // [3.1416] ..rounding
```

Just like with **String** types, numeric types can also be defined using a minimum width, adding a minus sign to indicate padding on the right instead of the left. When the length is prefixed with a zero, the value is padded with zeros (on the left) when the size of the provided value is less than the minimum length.

For floating point values, the number of decimal digits can be defined by specifying the precision (starting with a dot, followed by a numeric value). When a **Precision** is defined for an integer value a **RuntimeException** will be thrown.

When the number of decimal digits provided exceeds the maximum number of digits to be displayed, the value will be rounded.

## Argument Index

- Arguments can be defined out of order
  - By default, arguments are applied in order of format specifiers
  - Often used to format same value multiple times

```
String format = String.format("%1$s - %1$d - %2$sh", "text", "hex")  
//text - TEXT - 1931b
```

- Argument index must be defined after % sign
  - Must be followed by \$ sign

By default, when defining the values of each format-specifier in the template, you will have to provide the values in the same order as the order in which the format-specifiers are defined within the template.

When the order of the values differs, or a single value should be added to the string multiple times (maybe being formatted in different ways), the format-specifier can be defined using an argument index. The argument index should be added directly after the %-sign and should be followed by a \$ (dollar) sign.

## Formatting Date/Time

- **Formatting of date/time is defined by t or T**
  - Followed by conversion suffix
- **Can be applied to Long, Calendar, java.util.Date**
  - ...and the date/time API introduced in Java 8

```
LocalDate date = LocalDate.of(2017, 1, 12);
//B outputs name of month
String fmt1 = String.format("%tB", date); // January
String fmt2 = String.format("%TB", date); // JANUARY
```

- **Often combined with argument index**
  - To output multiple fields of date/time

```
LocalDate date = LocalDate.of(2017, 1, 12);
String fmt1 = String.format("%tB %tY", date); // January 2017
```

In order to be able a date or time, it is not sufficient to only define a single letter, you must also define how the date or time should be formatted. The lowercase or uppercase letter t should be used to start the definition of a date/time format-specifier. The letter that follows this letter t defines how (or what) should be generated in the result.

In most cases you want to output multiple fields of a single date/time object (in the example shown above the month and the year) so you would have to use argument index notation to reference the same value multiple times.

# Formatting Times

- **Formatting suffixes can be applied to format time fields**

| Suffix | Description                                 |
|--------|---------------------------------------------|
| H      | Hour of day in 24 hour clock (2 digits)     |
| I      | Hour of day in 12 hour clock (2 digits)     |
| k      | Hour of day in 24 hour clock                |
| l      | Hour of day in 12 hour clock                |
| M      | Minute of hour (2 digits)                   |
| s      | Seconds of minute (2 digits)                |
| L      | Millisecond within the second (3 digits)    |
| N      | Nanosecond within the second (9 digits)     |
| p      | Locale-specific morning or afternoon marker |
| z      | Time zone offset from GMT                   |
| Z      | String representing the time zone           |
| s      | Seconds since epoch (1 January 1970)        |
| Q      | Seconds since epoch                         |

```
LocalTime time = LocalTime.of(13, 45);
String fmt1 = String.format("%1$tH:%1$tM", time); // 13:45
String fmt2 = String.format("%1$tI:%1$tM %1$tp", time); // 01:45 pm
```

Several letters have been defined to format the various fields of an object which represents time.

The formatting language is an extremely flexible API, as can be seen by the specification of the S (seconds) suffix, which is defined as the seconds within a minute, even accounting for leap seconds (valid values are 00 – 60)

# Formatting Dates

- **Formatting suffixes can be applied to format date fields**

| Suffix | Description                             |
|--------|-----------------------------------------|
| B      | Full name of Month                      |
| b      | Abbreviated name of month               |
| A      | Full name of day of the week            |
| a      | Abbreviated name of day of the week     |
| C      | 4 digit year, divided by 100 (2 digits) |
| Y      | Year (4 digits)                         |
| y      | Last two digits of the year             |
| J      | Day of the year (3 digits)              |
| m      | Month of the year (2 digits)            |
| d      | Day of the month (2 digits)             |
| *      | Day of the month                        |

```
LocalDate date = LocalDate.of(2016, 1, 12);
String fmt1 = String.format("%std-%istm-%isty", date); // 12-01-16
String fmt2 = String.format("%istb %ity", date); // Jan 2016
String fmt3 = String.format("%tj", date); // 012
```

Just like when formatting time-based values, individual letters have been defined to specify the various properties of a date-based value

# Formatting Date/Time

- Special conversion characters have been defined
  - For most commonly used date/time formats

| Suffix | Full format             | Description             |
|--------|-------------------------|-------------------------|
| R      | %tH:%tM                 | 24-hour clock           |
| T      | %tH:%tM:%tS             | 24-hour clock           |
| r      | %tI:%tM:%tS %Tp         | 12-hour clock           |
| D      | %tm/%td/%ty             | Date                    |
| F      | %tY-%tm-%td             | ISO 8601 formatted date |
| c      | %ta %tb %td %tT %tZ %tY | Date/Time formatted     |

```
LocalDateTime ldt = LocalDateTime.of(2016, DECEMBER, 31, 23, 59);
String fmt1 = String.format("%tR", ldt); // 23:59
String fmt2 = String.format("%tT", ldt); // 23:59:00
String fmt3 = String.format("%tr", ldt); // 11:59:00 PM
String fmt4 = String.format("%tD", ldt); // 12/31/16
String fmt5 = String.format("%tF", ldt); // 2016-12-31
String fmt6 = String.format("%tc", ldt.atZone(ZoneId.systemDefault())));
//Sat Dec 31 23:59:00 CET 2016
```

For the formatting of date/time based values several special conversion characters have been defined for the most commonly used date/time conversions.

## Lesson Review and Summary

- 1) What is the purpose of the StringJoiner?**
- 2) What is the method signature of the String.format method? (parameters and return types)**

- 1) Create a delimited sequence of characters (e.g. a comma-separated list)
- 2) It takes a String (the template) and a varags of Object references (String format, Object... args) and overloaded method takes a Locale as the first parameter (Locale l, String format, Object... args), in both cases the result is a String

## Lesson: Java Data Access JDBC API

The new Date/Time API

Formatting Strings

**Java Data Access JDBC API**

Unit Testing Fundamentals

Jumpstart: JUnit 4.x

## Lesson Agenda

- Connecting to a database using JDBC
- Executing a statement against a database that returns a ResultSet
- Setting up and working with PreparedStatements
- Extracting multiple rows of data from a ResultSet, where each column value is represented as a String
- Inserting, updating and deleting rows in a table

### Purpose of Performance Objectives

At the conclusion of this session you should be able to perform each of the items listed above.

Take a moment to rate yourself on each of these items, on a scale of 1 through 5. Write your values to the left of the bullet items above. At the conclusion of this session these objectives will be reviewed. You should rate yourself again to see how much benefit you received from this session.

## What is JDBC?

- **Java Database Connectivity**
  - Client APIs for communicating with a wide range of data sources
  - Data source is usually a DBMS
    - ◆ JDBC is NOT limited to RDBMS data sources
- **JDBC is a set of standard interfaces**
  - Representing commands to query and modify relational data
- **Database vendor provides a database driver that implements the standard JDBC interfaces**
  - Format of data streams
  - Conversions between database and Java types
  - Driver API's

Even though JDBC is often thought of as an acronym for Java Database Connectivity, but in reality, JDBC is a trademarked name all by itself.

The API's that are defined by JDBC can be used to access a wide variety of data sources. Even though most applications are only using JDBC to communicate with a DBMS, the API can be used to communicate with any compliant data source, from flat files to Relational Database management Systems (RDBMS).

The API defines full support for creating database connections, sending SQL statements, invoking stored procedures, handling transactions and (naturally) dealing with the data that might be returned as a result of a query against the data source.

Database vendors implement these interfaces in JDBC drivers specific to a particular database management system (DBMS) in order to become JDBC compliant.

## Structured Query Language (SQL)

- Application accesses tables within database using SQL commands

- **SELECT**

```
SELECT Name, Street, Town, Code FROM Customer  
WHERE Town = "London"
```

- **INSERT**

```
INSERT INTO Customer  
VALUES ("M. Bird", "12 kit St", "London", "SW4 4GL")
```

- **DELETE**

```
DELETE FROM Customer WHERE name = "D. Satos"
```

- **UPDATE**

```
UPDATE Customer SET name = "A. Smith" WHERE name = "J. Smith"
```

A relational database consists of a series of tables. There are no fixed relationships between tables; all relationships can be defined dynamically at run time. Thus, the database is ideal for situations where requirements are continually changing or where ad-hoc queries need to be supported. The usual method for accessing data is via an industry-standard language called SQL.

## Connecting to the Database

- Application requires (at least) two pieces of information
  - Database JDBC URL
  - Database JDBC driver
  - (Optionally) database specific properties
  - (Optionally) database authentication credentials
- Connections can be obtained
  - Using the DriverManager
  - Using a DataSource
    - ◆ A factory for connections
    - ◆ Recommended in Java EE applications

## Initializing `java.sql.DriverManager`

- The driver class must be known to the `DriverManager`
  - Using the System property `jdbc.drivers`
    - ◆ (e.g. specified as a D flag)

```
java -Djdbc.drivers=org.apache.derby.jdbc.ClientDriver
```

- Explicitly loaded using `Class.forName`

```
try {  
    Class.forName("org.apache.derby.jdbc.ClientDriver");  
}  
catch (ClassNotFoundException cnfe) { ... }
```

- Since JDBC 4.0 `DriverManager` supports Service Provider mechanism
  - Drivers must include `META-INF/services/java.sql.Driver` file
    - ◆ Specifying driver class

Explicitly adding the driver via either of these methods is not necessary if you are using a JDBC 4.x driver. Earlier drivers did need this.

## Connecting to the Database (Java SE)

- A connection is obtained from the **DriverManager**

```
String username = ...  
String password = ...  
String url = "jdbc:derby://localhost:50505/BertrandDB";  
try (Connection conn =  
    DriverManager.getConnection(url,username,password)) {  
  
    ...  
}  
} catch (SQLException sqle) {  
    ...  
}
```

- **DriverManager** searches for driver that supports URL
  - Format and syntax of the URL is specific for each driver

The application must provide information about both the type and the location of the database (the database URL). In most cases, databases require a user name and password in order to establish a connection.

When a connection is requested from the **DriverManager**, the **DriverManager** will search through all the available drivers in order to find a driver that supports the URL provided. Each driver defines a specific format and syntax of URL that is to be used.

## java.sql.Statement

- A Statement is created on the connection
- Important methods of Statement
  - executeQuery(String)
    - ◆ Used for SELECT statements , returns a ResultSet
  - executeUpdate(String)
    - ◆ Used for CUD operations, returns number of rows effected
  - getWarnings()
    - ◆ Returns warnings sent by the server due to last statement
  - close()
    - ◆ Closes all resources
- Implements AutoCloseable interface
  - Can be used in try-with-resources

```
try (Statement stmt = connection.createStatement();) {  
    ...  
    ... = statement.executeQuery("SELECT * FROM EMP");  
    ...}
```

A **Statement** object is used for executing a non-parameterized SQL statement and obtaining the results produced by it, in the form of a **ResultSet**.

## java.sql.PreparedStatement

- **PreparedStatement extends Statement**
  - Is compiled with a specific command
  - Overloaded `executeQuery` and `executeUpdate` methods do not take a SQL string as argument
  - Various `setXXX(int param, XXX)` to set the value of parameters in the query

```
String prep = new String("SELECT * FROM EMP WHERE Salary > ?");  
PreparedStatement prepStmt = conn.prepareStatement(prep);  
prepStmt.setInt(1, 50000); // sets first ? to 50000  
... = prepStmt.executeQuery();
```

```
void setString(int param, String str);  
void setInt(int param, int intValue);  
void setDate(int param, XXX);  
--
```

- **Unlike Java indexes, SQL column indices start at 1 !**

If the same SQL statement is executed many times, it is more efficient to use a **PreparedStatement**. A SQL statement with or without input parameters can be pre-compiled and stored in a **PreparedStatement** object. This object can then be used to efficiently execute this statement multiple times. Whether or not the database or its driver supports pre-compilation is largely transparent to programmers.

Rather than taking an SQL string as an execution argument, the **PreparedStatement** is constructed with the execution statement. Generally, dynamic SQL is used, meaning that question marks are embedded in the statement string where their values can be dynamically changed at runtime. This allows a database statement to be compiled once, and then be used repeatedly, changing only the parameters that control the result of its execution.

Note that unlike Java indexes, SQL column indices start at 1, not 0, so be careful!

## java.sql.ResultSet

### ● A sequence of row values

```
ResultSet rset = prepStatement.executeQuery();  
  
while (rset.next()) {  
    col1 = rset.getString("NAME");  
    col2 = rset.getInt("SALARY");  
    ...  
}
```

```
String getString(String columnName);  
String getString(int colIndex);  
int getInt(String name);  
int getInt(int colIndex);  
XXX getXXX(String columnName);  
XXX getXXX(int colIndex);  
boolean next();
```

### ● Methods of ResultSet

#### ■ Various getXXX methods

- ◆ Either take column index or column name as argument
- **next** tries to position cursor on the next row
  - ◆ Returns true or returns false depending on the success

### ● Connection must still be valid to use ResultSet

Only one **ResultSet** per **Statement** can be open at any point in time. Therefore, if the reading of one **ResultSet** is interleaved with the reading of another, each must have been generated by different **Statements**. All statement execute methods implicitly close a statement's current **ResultSet** if an open one exists.

There are two **getXXX()** methods for each type: one that takes a column index and one that takes a column name. For example, to get a **String** object from the first column (ename) of our employee table, we can use either **getString(1)** or **getString(ename)**. Using a column number is slightly more efficient, but makes the code less readable.

Connection must still be valid to use **ResultSet**.

## Executing Inserts, Updates and Deletes

- **Inserts, updates and deletes return row counts, not data**
- **Statement and PreparedStatement methods**

```
ResultSet executeQuery(...);  
int executeUpdate(...);
```

- **These database commands use executeUpdate**

```
int count = prepStatement.executeUpdate();  
System.out.println("Employee raise count: " + count);
```

The **Statement executeQuery()** method executes an SQL statement that returns a single **ResultSet**. Alternatively **executeUpdate()** can be used to execute an SQL INSERT, UPDATE or DELETE statement. In addition, SQL statements that return nothing such as SQL DDL statements can be executed this way.

# Controlling Transactions

- **Transaction control**

- By default, each statement is run in a new transaction and automatically committed upon return
- Automatic actions can be disabled, providing programmatic control of commits and roll-backs
  - ◆ Multiple statements can be bracketed to allow either all statements to be applied, or none

```
conn.setAutoCommit(false);
...
prepStmt.executeUpdate();
...
prepStmt.executeUpdate();
conn.commit();
// conn.rollback();
```

## Transaction Control

By default, the transaction mode for a connection is generally with auto-commit enabled. This means that each statement runs in a separate transaction, and that the transaction is implicitly committed prior to the call returning.

In many cases this behavior is not acceptable. This option can be turned off on the connection.

With auto-commit disabled, a new transaction is created when a statement is executed, but only if there is not a transaction already in place on the connection. Additional statements can be executed, and all will be performed in the same existing transaction.

Once the application decides that it is time to commit the work that has been performed across the multiple statements, it calls commit on the connection.

## Mapping SQL Types to Java Types

- JDBC provides mapping between most standard SQL data types and similarly named Java types
- Most SQL types will also map to String
- Vendor-specific data types are problematic
  - May provide custom Java classes to hold these types
  - Read the driver documentation for more details
  - Be sure to test the driver thoroughly
  - Examples: money, vargraphic, etc.
- Special types defined in `java.sql` to provide better mapping support than what would be possible by similar standard Java type
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.Timestamp`

### Mapping Data Types

JDBC provides reasonable type mappings from Java for the common SQL data types. There is also a corresponding type mapping from Java types to SQL types.

Not all databases may support all the SQL Types, e.g. BIGINT is not in the SQL 1992 standard.

**java.sql.Time** and **java.sql.Date** are thin wrappers for the **java.util.Date** class. The constructor of the **java.util.Date** class will not accept either a **java.sql.Time** or **java.sql.Date** object!

For more information on mapping SQL data types in Java, please refer to the JDBC specification section 8.

# Obtaining Database Metadata

- A Connection instance can be used to get database metadata
  - List of tables and columns in database
  - Database configuration parameters

```
Connection conn = ...  
DatabaseMetaData metaData = conn.getMetaData();  
String productName = metaData.getDatabaseProductName();  
String productVersion = metaData.getDatabaseProductVersion();  
int defaultTxIsolation = metaData.getDefaultTransactionIsolation();  
ResultSet tables = metaData.getTables(null, null, null, null);
```

- Getting columns of a table

```
ResultSet resultSet = metaData.getColumns(null, null, "EMPLOYEE", null);  
while (resultSet.next()) {  
    String columnName = resultSet.getString("COLUMN_NAME");  
    String columnType = resultSet.getString("TYPE_NAME");  
}
```

The **DatabaseMetaData** interface provides a standard way of getting information about a database itself. This information includes, for example, the tables in the database, the column names in each table, primary keys, foreign keys and stored procedures.

Columns in 'tables' ResultSet include

- The table's catalog (TABLE\_CAT)
- The table schema (TABLE\_SCHEM )
- The name of the table (TABLE\_NAME)
- The table type (TABLE\_TYPE )

e.g., TABLE, VIEW, SYSTEM TABLE, GLOBAL, TEMPORARY

A common use for database metadata is to present to the user a data entry form that represents all of the columns in a particular table. This would allow a standard application to support data entry on a table whose definition is not known at the time the data entry application was written.

## Obtaining ResultSet Metadata

- **ResultSetMetaData** can be obtained from **ResultSet**
  - Providing information on columns returned by query
    - ◆ Name, Type, ...

```
ResultSet rs = ...
ResultSetMetaData metaData = rs.getMetaData();
for(int i = 1; i <=metaData.getColumnCount(); i++) {
    String columnName = metaData.getColumnName(i);
    int columnType = metaData.getColumnType(i);
    -
}
```

The methods of the **ResultSetMetaData** interface provide a way to get the types and properties of the columns in a **ResultSet** object. This information includes the number of columns, the column labels and the column types.

## Stored Procedures Defined

- A stored procedure is a routine managed by the database
  - May return 0 or more result sets
- Represented by CallableStatement
  - Extends PreparedStatement
- Callable statements use a special syntax
  - Call is contained within "{" and "}" curly brackets
- The call itself is CALL pName (params)
  - pName is the name of the stored procedure
  - params are zero or more parameters passed to the procedure
  - You may supply parameters hard coded, or as ? placeholders

```
String sql = "{ call stProc( ?, ?, ?) } "
CallableStatement cstmt = conn.prepareCall(sql);
```

Stored procedures are used to provide server side business logic. Stored procedures can execute one or more SQL statements, and can define transactions.

The reason for the strange syntax is to flag an escape sequence to the driver's parser. This allows the driver to translate the call into the exact format supported by the database engine. As far as you are concerned, calls work very much like prepared statements.

## Stored Procedure Parameters

- Input parameters are set just like prepared statements
- You must register output parameters with their type
- INOUT parameters must be set first and then registered as an output type
- Parameter numbers start with 1

```
import java.sql.Types;

String sql = "{ call stProc( ?, ?, ?) }";
CallableStatement cstmt = con.prepareCall(sql);
cstmt.setInt(1, 1000); // IN
cstmt.registerOutParameter(2, double); // OUT
cstmt.setString(3, "O'Reilly"); // INOUT
cstmt.registerOutParameter(3, VARCHAR);
cstmt.execute();
```

### In's and Out's

Stored procedures support both **IN** and **OUT** parameters, which means that we need to define the **OUT** parameters. Parameters can also be **INOUT**, which means first we treat them as an **IN**, and then as an **OUT**. To handle **OUT** parameters, we issue a **registerOutParameter** method with the **java.sql.Types.TYPE** for the parameter.

## javax.sql.RowSet

- **RowSet implementations holds tabular data**
  - More flexible than a result set
- **RowSet is a sub-interface of the ResultSet interface**
  - Has all capabilities of ResultSet
- **Has a live connection to the database**
- **Not all databases support scrollable result sets**
  - RowSet can be used to 'add' this functionality
- **RowSet instances might throw events**
  - When the cursor is moved
  - When a row is updated, inserted or deleted
  - When the content of the RowSet changes

```
RowSet rowSet = null;
RowSetListener listener = new MyRowSetListener();
rowSet.addRowSetListener(listener);
```

## Connected vs Disconnected RowSets

- **Connected RowSet implementations maintain database connection throughout its life span**
  - `JdbcRowSet` is only 'standard' connected RowSet
- **Disconnected RowSet implementations only use connection to read or write data**
  - Connection to database is closed most the time
  - Works independently from result set
  - Are Serializable
- **JDBC contains several standard implementations**
  - `JdbcRowSet`: Thin wrapper around `ResultSet`
  - `CachedRowSet`: Connects when receiving or sending data
  - `FilteredRowSet`: Used to get subset of `ResultSet` data
  - `JoinRowSet`: Data from multiple RowSets as if from SQL JOIN
  - `WebRowSet`: Used to manipulate XML data

## javax.sql.rowset.JdbcRowSet

- **JdbcRowSet object requires connection to a database**
- **Contains properties used to connect to database**
  - ◆ **username:** Username to connect to database
  - ◆ **password:** User's database password
  - ◆ **url:** Database URL
  - ◆ **datasourceName:** JNDI name of a DataSource object
- **Can be constructed in different ways**
  - Wrapping an existing Connection or ResultSet
  - From RowSetFactory
  - Using default constructor
    - ◆ Providing connection properties through setter methods

```
JdbcRowSet jdbcRs = new JdbcRowSetImpl();
jdbcRs.setUsername( username );
jdbcRs.setPassword( password );
jdbcRs.setUrl( URL );
```

## Executing a Statement

- The command property contains the SQL statement to populate the RowSet

```
jdbcRowSet.setCommand("select * from EMP");
```

- The execute() method must be invoked to populate the RowSet

```
jdbcRowSet.execute();
```

- Established connection to database
  - ◆ Using connection properties of RowSet
- Executes 'command' query
- Populates RowSet with data from ResultSet

# Navigating Through RowSet

- RowSets can be iterated using `next()` method

```
jdbcRowSet.setCommand("SELECT * FROM EMPLOYEE");
jdbcRowSet.execute();
while(jdbcRowSet.next()) {
    String empID = jdbcRowSet.getString("EMPID");
    String firstName = jdbcRowSet.getString("FIRSTNAME");
    ...
}
```

- Scrollable RowSets can be navigated

- Using `absolute(int rowNum)` method

- ◆ Using negative rowNum select row relative to end of result set

```
jdbcRowSet.absolute(-1); //Get Last Row
jdbcRowSet.absolute(-2); //Get Next to Last Row
```

- Using `previous()` method

```
jdbcRowSet.absolute(-1); //Get Last Row
jdbcRowSet.previous(); //Get Next to Last Row
```

## JdbcRowSet – Retrieving Data

- **ResultSet methods are used to retrieve data**

```
String name = rowset.getString("name");
String id = rowset.getInt("id");
```

- **update<Type> methods can be used to update value**

- The **updateRow()** method causes change to be applied to database

```
jdbcRowSet.setCommand("SELECT * FROM EMPLOYEE");
jdbcRowSet.execute();
while (jdbcRowSet.next()) {
    jdbcRowSet.updateDouble("SALARY", 1235);
    jdbcRowSet.updateRow();
}
```

- **To insert, cursor must be moved to insert-row**

```
jdbcRowSet.moveToInsertRow();
jdbcRowSet.updateString("EMPID", "444");
//set values of all (not-null) columns
jdbcRowSet.insertRow();
```

- **deleteRow() causes current row to be deleted**

```
jdbcRowSet.deleteRow();
```

## **javax.sql.rowset.CachedRowSet**

- Disconnected container for rows of data
  - Caches its rows in memory
- Operates without always being connected to data source
  - Connected only when data is read from or propagated to the database
  - Disconnected all other times
    - ◆ Including when its data is being modified
- Is scrollable, updatable and **serializable**
- Data can be modified and propagated back to the database

## CachedRowSet Example

```
try (CachedRowSet rowSet = rowSetFactory.createCachedRowSet()) {
    rowSet.setUrl(URL);
    rowSet.setUsername(username);
    rowSet.setPassword(password);
    rowSet.setPageSize(3);
    rowSet.setCommand("SELECT * FROM EMPLOYEE");
    rowSet.execute();
    List<Employee> employees = new ArrayList<>();
    do {
        while (rowSet.next()) {
            String firstName = rowSet.getString("FIRSTNAME");
            String lastName = rowSet.getString("LASTNAME");
            double salary = rowSet.getDouble("SALARY");
            employees.add(new Employee(firstName, lastName, salary));
        }
        processEmployees(employees);
        employees.clear();
    } while (rowSet.nextPage());
}
```

## Lesson Review and Summary

- 1) What class is used to connect to the database?**
- 2) What information must be provided to enable a connection to succeed?**
- 3) From which class is a Statement created?**
- 4) Is it possible to list the columns of a specific table?**
- 5) Is it possible to see the list of columns returned in a ResultSet?**

1. A Connector class
2. A JDBC driver is used to connect to a database by providing the JDBC URL, a user name, and password
3. From the connection a statement is created, which can be used to issue commands
4. Metadata can be retrieved on both the database and a returned ResultSet which will provide detailed information regarding structure and content

## **Tutorial: Setup The Derby Database**

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## Exercise 23: Reading Table Data

`~/StudentWork/code/TableData/lab-code`

Please refer to the written lab exercise and follow the directions as provided by your instructor

## Exercise 24: Using JdbcRowSet

`~/StudentWork/code/RowSets/lab-code`

Please refer to the written lab exercise and follow  
the directions as provided by your instructor

## **Exercise 25: Executing within a Transaction**

**`~/StudentWork/code/Transactions/lab-code`**

Please refer to the written lab exercise and follow the directions as provided by your instructor

# Lesson: Unit Testing Fundamentals

The new Date/Time API  
Formatting Strings  
Java Data Access JDBC API  
**Unit Testing Fundamentals**  
Jumpstart: JUnit 4.x

## What is Unit Testing?

- A unit test examines the behavior of a distinct unit of work
- Unit of work is a task that is not directly dependent on the completion of any other task
- Unit tests should be fine grained, testing small numbers of closely-related methods and classes
- Unit tests focus on testing whether a method is following the terms of its API+ contract
  - API: Application Programming Interface - a formal agreement between the caller and the called

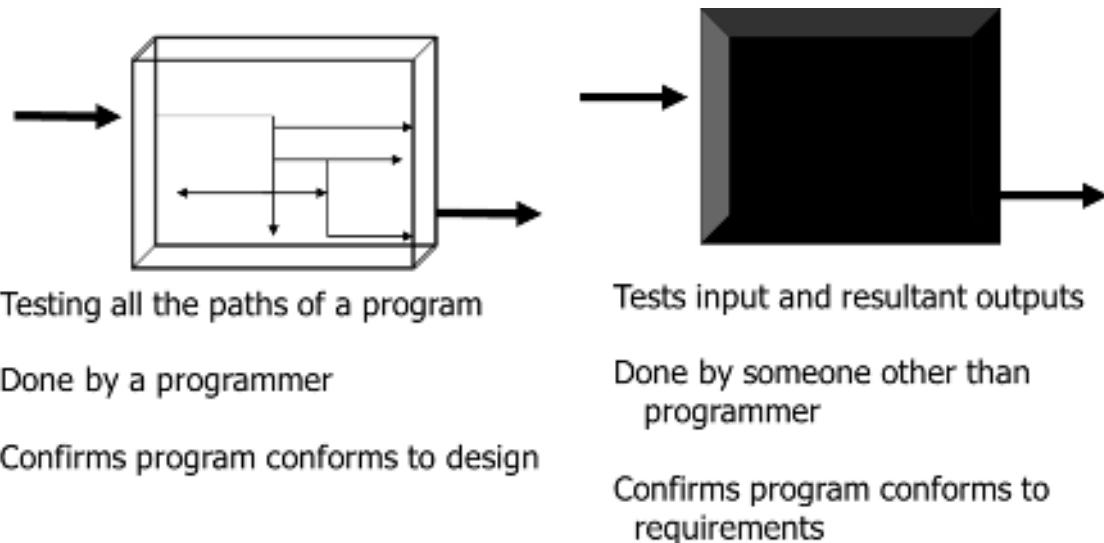
## What is Unit Testing? (cont'd)

- **A unit is the smallest testable software unit**
  - A unit typically is the work of one programmer (at least in principle)
  - As defined, it does not include any called sub-units (for procedural languages) or communicating units
- **In Unit Testing, units are replaced with stubs, simulators, or trusted units**
- **Calling units are replaced with drivers or trusted super-units**
- **The unit is tested in isolation**

Definitions from Boris Beizer

## What Is Unit Testing? (white-box/black-box)

- Typically consists of both white-box and/or black-box testing



# Purpose of Unit Testing

- Ensure the unit or program functions correctly
- Increase, to an acceptable level, the developer's confidence that the unit will behave correctly under all circumstances of interest
  
- Unit testing is the most cost effective method of execution testing for finding defects
  - More thorough code coverage
  - Easier to debug when an error is detected
  - Minimum disruption from fixes
  - Most cost effective method of validation

# Why Do We Write Unit Tests?

- **Main Goal:** To verify that the application works and try to catch bugs early
- **Unit tests are more powerful than functional testing because**
  - They allow greater test coverage
  - They enable teamwork
  - They prevent regression and limit the need for debugging
  - They give the courage to refactor
  - They improve the implementation design
  - They serve as developer's documentation

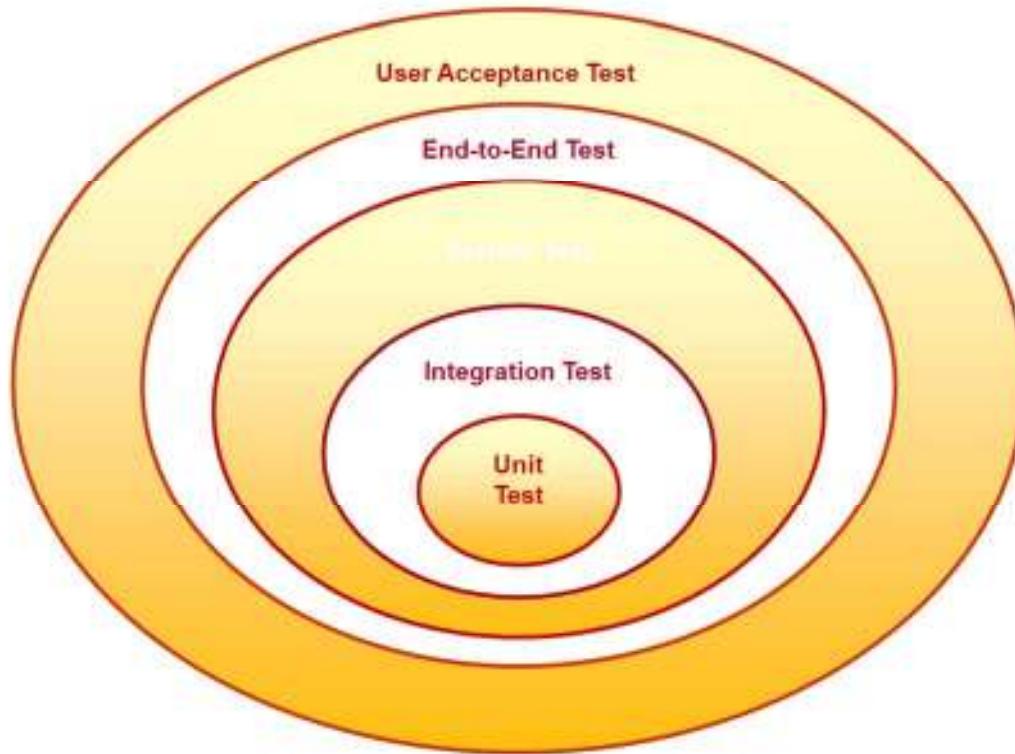
# Good Unit Tests

- **Good tests are standalone**
  - The test does not rely on previous tests
  - The test has well-understood and managed dependencies
  - The test has minimal dependencies such as
    - ◆ Time, Randomness and Concurrency
    - ◆ Infrastructure including networking
    - ◆ Pre-existing and persistent data
- **They should run fast**
  - Testing to make sure that code can handle a large data set is a different kind of test: scalability, load balancing, etc.
- **The test must be legible**
  - Create supporting methods with clear names
  - Use variables to clarify the meaning of the data being used

## Good Unit Tests (cont'd)

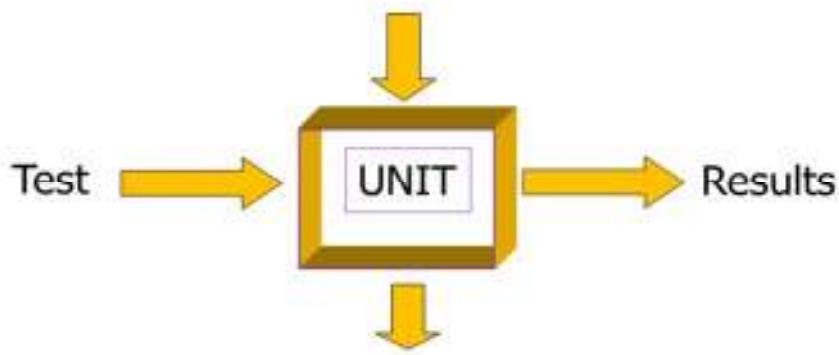
- **Good tests avoid smells and anti-patterns**
  - Anything that impacts ability to understand:
    - ◆ The intent of the test
    - ◆ What it is actually testing
  - Will be examining these in detail after completing our tour of JUnit
- **A good test should have only one reason to fail**
  - Variant of the OO design Single Responsibility Principle
  - Improves readability, understandability, and long-term viability

# Test Stages



## Unit Test Stage

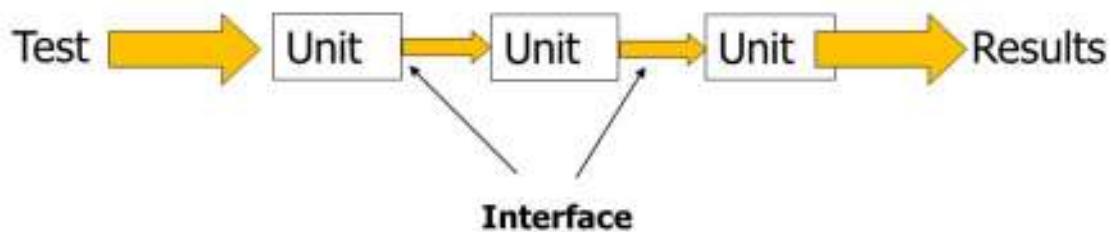
- **Testing of individual hardware or software units or groups of related units**
- **Usually performed by the developer**



Do not spend a lot of time on the next several slides – they are only introduction.

## Integration Test Stage

- Testing in which software units, hardware components, or both are combined and tested to evaluate the interaction between them
- Groups of units tested together within a **SINGLE APPLICATION**
  - The idea is: If units all work 100% correctly, problems can only be in the interfaces
  - Order of grouping the units becomes important



The assumption is that Unit Testing has been performed first. Reference: IEEE Std 610.12-1990

# Unit Testing vs Integration Testing

- **Integration testing is the extension of unit testing**
  - Some number of units are combined into components which are aggregated into larger modules
- If unit testing is being done in a complete and consistent fashion, errors that show up in the integration testing are related to the interfaces between the units
  - This controls the exposure, making it easier to determine the cause of the error

# Functional Testing

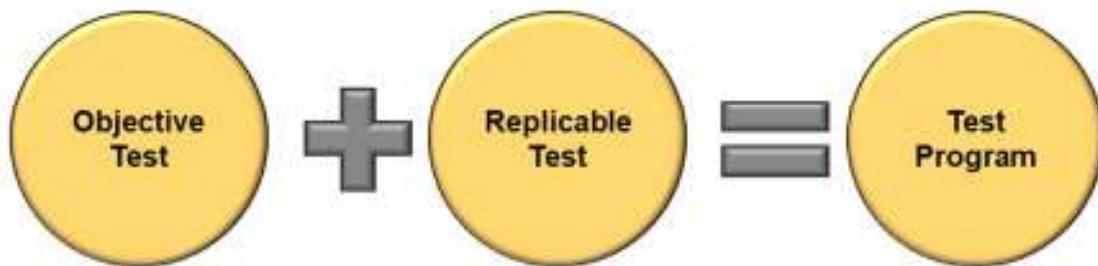
- **Testing that ignores the internal mechanism of a system or software unit and focuses solely on the outputs generated in response to selected inputs and execution conditions synonymous with black-box testing**
- **Testing conducted to evaluate the compliance of a system or software unit with specified functional requirements**

Ref: IEEE Std 610.12-1990 for all of the above

# Non-Functional Testing

- **Validate those requirements that are not functional**
  - Includes, but is not limited to:
    - ◆ Availability
    - ◆ Back Out
    - ◆ Compliance
    - ◆ Configuration
    - ◆ Conversion
    - ◆ Failover
    - ◆ Installation Testing
    - ◆ Recovery
    - ◆ Reliability
    - ◆ Security
    - ◆ Serviceability
    - ◆ Usability

# Understanding Unit Testing Frameworks



- All unit testing frameworks should observe:
  - Rule 1: Each unit test must run independently of all other unit tests
  - Rule 2: Errors must be detected and reported test by test
  - Rule 3: It must be easy to define which unit tests will run

## So What is a "Good" Test?

- **Definition of good is very subjective**
  - Some criteria change depending on the context
  - Judgment, bias, and experience blend to impact meaning
- **Some things are consistently true, especially when you consider that test code is part of your company's code base**

## Good: Readable Equates to Maintainable

- In any code base, unreadability equates to more energy to understand what is being done
  - Studies show that poor readability correlates to defect density
  - Less likely someone will take the time to analyze what is going on
  - More likely someone will misinterpret the code and make a mistake in that fashion
  - Less likely a given developer will even want to touch that test
- Do not want to make test code a maintenance problem

Raymond P.L. Buse, Westley R. Weimer, "Learning a Metric for Code Readability," IEEE Transactions on Software Engineering, 09 Nov. 2009. IEEE computer Society Digital Library. IEEE Computer Society, <http://doi.ieeecomputersociety.org/10.1109/TSE.2009.70>

# Proper Organization and Structure

- Large, monolithic tests have several negative impacts
  - Longer execution time increases delayed feedback
  - More challenging to find and analyze test failure points
  - More challenging to analyze prior to implementing a change
- Breaking code into consistent chunks of code does little to help the situation
  - Structure of the code needs to help developers locate the implementation of concepts and behavior
  - Test should be sized to show how associated production code should behave
- Need focused tests that are comprehensible, singular in function, and run in an efficient and rapid manner

This is something of an extension of readability - concept is similar...good structure leads to easier understanding

## Test the Right Thing

- When performing diagnostics and analysis on a code base, often run the unit tests to see what is succeeding and what is failing
- Two issues can negatively impact this approach
  - Tests need to be named to represent what is being tested
    - ◆ Should be able to trust the meaning of those names
  - Test the correct things the correct way
    - ◆ Test needs to target what name represents
    - ◆ Test for behavior and not to a specific implementation

# Run in Solitude

- **Trying to achieve both isolation and independence for tests**
  - The greater the dependencies, the more it takes to run the test and greater the likelihood of test failures not related to targeted functionality
  - Will be talking about managing dependencies later in course
- **Certain dependencies are difficult to manage**
  - Time and timing, randomness, concurrency, infrastructure and networking, pre-existing and persistent data
  - Legacy code that cannot be modularized
  - Each of these should be identified, avoided if possible, and isolated if not avoided

A couple of anti-pattern to look for in the code being tested:

Long methods typically imply more dependencies

Descriptive names for constructs such as variables, methods, and classes often imply better adherence to OO principles

# Reduce and Manage Dependencies

- **Use test doubles to substitute for library dependencies**
- **Co-locate test code and the resources they use**
  - For example, maintain in same package
- **Make sure test sets up entire context**
  - Do not depend on other tests being previously run
- **Use an in-memory database for any unit or integration testing that requires persistence**
  - Facilitates starting with clean data sets
  - Typically fast startup
- **Split threaded code into asynchronous and synchronous**
  - Test logic and data flows in synchronous units
  - Isolate concurrency to specific tests

Test doubles are a general term for artifacts such as stubs, fakes or mocks that are substituted in for real implementations.

# Reliability

- **Tests need to fail or they may not be testing anything**
  - Sometimes referred to as "happy" tests
  - Indicator is the use of try-catch's in test method bodies
    - ◆ Usually means test only fails if tested method throws an exception
- **Tests need to fail reliably**
  - Same input should consistently generate same results
  - Unrecognized and unmanaged dependencies on timing or randomness (random number generator for example) are usual culprit

## To Sum Up...

- A unit test case answers a single question about the code snippet it is testing
- Use Stubs and drivers to replace/represent interfaces
- There are a variety of white-box and black-box testing methods that can be used
- JUnit and NUnit are tools used to support the implementation and execution of unit tests

**It is more economical and easier to find the defect in Unit testing than in any Future Tests!**

## Keys to Success

- A thorough unit test case method, used properly will create a solid testing project
- Unit testing is a critical step to detecting defects on each project
- Unit testing will enhance future test cases during subsequent project testing
- Don't throw the code over the fence expecting end-users or testers to catch the defects
- If possible, write Unit Test cases before coding begins
- If there is a unit test time constraint, perform White-Box first, then Black-Box tests
- Code Reviews reduces dependence on
- Unit Testing results



## **Lesson: Jumpstart: JUnit 4.x**

The new Date/Time API  
Formatting Strings  
Java Data Access JDBC API  
Unit Testing Fundamentals  
**Jumpstart: JUnit 4.x**

## Lesson Agenda

- **Know the features of JUnit**
- **Write test programs using JUnit TestCase**
- **Write unit tests using @Test annotation**
- **Manage fixtures using @Before, @After, @BeforeClass and @AfterClass annotations**
- **Launch tests using @RunWith annotation**
- **Build test suite using Suite.class test runner**

## JUnit Overview

- JUnit is a simple open-source testing framework
  - Written by Kent Beck (Extreme Programming guru) and Erich Gamma (lead developer of the Eclipse JDT)
- JUnit home page: <http://www.junit.org>
- Many IDEs have direct support for JUnit

## JUnit Design Goals

- The framework must help to write useful tests
- The framework must help us lower the cost of writing tests by reusing code
- The framework must help us create tests that retain value over time

## JUnit Features

- **JUnit provides features to make tests easier to write, maintain and run:**
  - Standard resource initialization and reclamation methods
  - Separate classloaders for each unit test
    - ◆ To avoid side effects
  - Variety of assert methods to check the results of your tests
  - Alternate front-ends (or test runners) to display the result of tests
  - Integration with popular tools like Maven and popular IDEs like Eclipse, IntelliJ, NetBeans, etc.

## Reasons to Use JUnit

- **JUnit does a number of things:**
  - Makes tests repeatable
  - Provides initial client to use while developing code
  - Documents the usage of the class API under test
  - Makes it easy to run one or more tests either
    - ◆ From within IDE
    - ◆ As part of project build (e.g. Maven / Gradle)
    - ◆ From command line
- **The process: test, code, refactor, design, repeat**

Testing code can be a daunting task without a tool. All developers test; the question is at what point do they stop testing due to the onerous task of keeping their tests up-to-date due to a lack of standardization. JUnit presents that standardization. It is easy to learn and easy to use.

IDEs like Eclipse, IntelliJ and RAD all have explicit support for JUnit.

## How JUnit Works

- JUnit creates a new instance of your test class for every method that must be run
- When using JUnit 4, import these method annotations:

| org.junit.Test        | Designates method as a test         |
|-----------------------|-------------------------------------|
| org.junit.Before      | Designates per-test setup method    |
| org.junit.After       | Designates per-test cleanup method  |
| org.junit.BeforeClass | Designates per-class setup method   |
| org.junit.AfterClass  | Designates per-class cleanup method |

- Annotate various public, void, no-arg methods
  - Annotated method names do not matter
  - Containing class name does not matter
  - Containing class need not extend any base class
  - Containing class need not implement any interface

Once the TestRunner is given the name of the test class, and the classes to be run are in the classpath, all that is left is to examine the output to find where a test went wrong.

Always strive for the green bar!

## Test Case using JUnit

```
import static org.junit.Assert.*;
import org.junit.Test;
public class TaxCalculatorImplTest {
    @Test
    public void shouldUseLowestTaxRateForIncomeBelow38000() {
        TaxCalculatorImpl calc = new TaxCalculatorImpl();
        double expectedTax = 30000 * 0.195;
        double calculatedTax = calc.calculateIncomeTax(30000);

        assertEquals(
            "Tax below 38000 should be taxed at 19.5%",
            expectedTax, calculatedTax, 0);
    }
}
```

```
public class TaxCalculatorImpl implements TaxCalculator {
    public double calculateIncomeTax(double income) {
        ...
    }
}

public interface TaxCalculator {
    double calculateIncomeTax(double income);
}
```

The framework must help to write useful tests.

Any class can be a test case and all test methods should have @Test annotation

## Exploring JUnit



- **Tests:**
  - Any POJO class can be a test case
  - Contains one or more related tests
  - No special naming convention required
- **Runner:**
  - A launcher of tests
  - `@RunWith` annotation is used to indicate runner to be used
- **Result:**
  - Collects any errors or failures that occur during a test
  - Every Runner has a Result

## Writing the TestCase

- **Creating a test case with JUnit framework requires:**
  - The test class does not need to extend any particular class
  - Unit test methods to be marked by @Test annotation
  - All unit test methods to be public void and take no parameters
  - Test methods to make assert calls to validate the outcome

When assert methods are used, make sure the signature that takes String is used.

## Test Result Verification (Assertions)

- Do a static import on `org.junit.Assert.*`
- Class `org.junit.Assert` provides us with numerous overloaded static methods for testing

| Method                     | Description                                             |
|----------------------------|---------------------------------------------------------|
| <code>assertTrue</code>    | Asserts that a condition is true                        |
| <code>assertFalse</code>   | Asserts that a condition is false                       |
| <code>assertEquals</code>  | Asserts that two objects are equal                      |
| <code>assertNotNull</code> | Asserts that an object is not null                      |
| <code>assertNull</code>    | Asserts that an object is null                          |
| <code>assertSame</code>    | Asserts that two objects refer to the same object       |
| <code>assertNotSame</code> | Asserts that two objects don't refer to the same object |
| <code>fail</code>          | Fails a test with the given message                     |

## Launching Tests

- Test runners are designed to execute tests and provide you with statistics regarding the outcome
- When a class is annotated with `@RunWith`, JUnit will invoke the class it references to run the tests in that class
- `JUnitCore` is a facade for running tests
- To run tests from the command line, run:

```
java -cp junit.jar org.junit.runner.JUnitCore AllTests
```

## Failures vs. Errors

- **Failures**
  - Assert method fails if the API contract cannot be fulfilled
- **Errors**
  - These are unexpected conditions that are not expected by the test

## Introducing Class Message

- **Message is a simple class that has the following:**
  - A constructor that takes a string as an argument
  - A `getGreeting()` method that returns a String
- **Message represents a String that can be printed out to the console**

```
public class Message {  
    private String greeting = null;  
  
    public Message(String msg) {}  
  
    public String getGreeting() {  
        return null;  
    }  
}
```

```
package com.triveratech.samples;
```

## Creating Class MessageTest

- When using a testing framework like JUnit, you should write your test BEFORE you code the methods you want to test
  - This may seem strange at first, but it will help you write robust code
- The first method we'll test/write is the constructor for class Message
  - Ultimately, we want the constructor to assign the argument passed to it to the instance variable greet

# The First Test Implementation Steps

- In test, three steps are necessary to carry out test:
  - Declare an instance variable to hold reference to object under test
  - Use setup method to instantiate an instance of Message and assign it to the instance variable
    - ◆ Method is annotated using @Before
  - Define teardown method to get rid of Message instance
    - ◆ Method is annotated with @After

```
public class MessageTest {  
    private Message message;  
    @Before  
    public void setUp() throws Exception {  
        this.message = new Message("Hello, World!");  
    }  
    @After  
    public void tearDown() {  
        this.message = null;  
    }  
    ...  
}
```

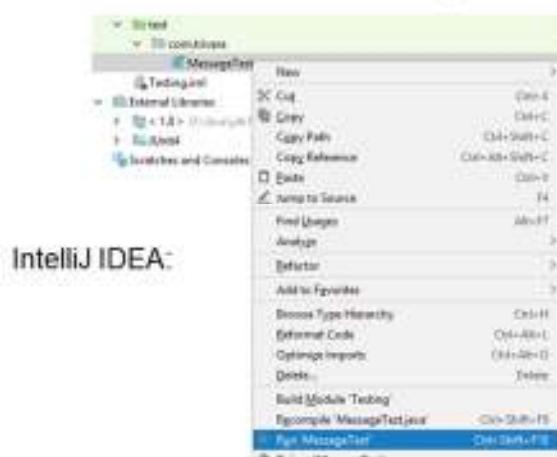
## Testing the Constructor

- Test for the constructor is defined in `MessageTest.testConstructor()` method
- The test is simple:
  - If the constructor is working correctly, test to see that the `getGreeting()` method returns the expected value
- The following code performs the test:

```
@testable
public void testConstructor() {
    String expected = "Hello, world!";
    String actual = message.getGreeting();
    assertEquals(expected, actual);
}
```

# Running a Test in an IDE

- **Running the JUnit TestRunner from within an IDE is a relatively trivial exercise**
  - The IDE has CLASSPATHs and other environmental information configured
  - Easily accessed via IDE's Preferences and Project properties



IntelliJ IDEA:

## Running a Test From the Command Line

- To run TestRunner from the command line:
  - Add junit.jar to the CLASSPATH or include it in the call to the JVM using the –classpath option (-cp)
  - Add the class to be tested and the testing class to the CLASSPATH or include them in the call to the JVM using the –classpath option
  - Execute TestRunner with the name of the test class

```
C:\>java -cp %classpath%;c:\junit4\junit-4.4.jar; - junit.textui.TestRunner MessageTest
```

## Seeing Results of a Test: JUnit View

- Since constructor in class Message has not been completed, test fails
  - No errors, but there was one failure that occurred in testMessage()
  - Message shows what problem was: “Hello, world!” was compared to null (the value returned by getGreeting()) and failed



You should always get a red bar first. This guarantees that your test code is being called properly and that the code you are developing does not make assumptions about how it should behave.

## Using the Results of a Test

- After viewing the results of the test, it is obvious that code needs to be added to constructor
  - Assign the argument passed to the instance variable

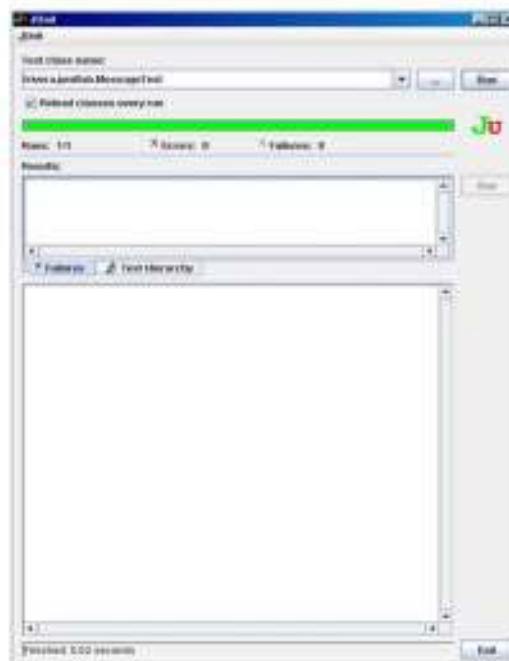
```
public Message(String msg) {  
    this.greeting = msg;  
}
```

- In addition, `getGreeting()` method needs to be implemented

```
public String getGreeting() {  
    return this.greeting;  
}
```

## Seeing Results of a Successful Test

- **Test remains the same!**
  - Original class is modified until it passes
- **Re-running the test after making the changes should result in a success**



So the process should be:

Write a test

The test should fail

Write the code that will cause the test to pass

The test should pass

Refactor the code to remove duplication

Repeat

## Test Suites

- A test suite allows us to combine a related group of test classes into a “suite” class
  - Alternative to running whole project “as JUnit Test”
  - Alternative to running a specific test class “as JUnit Test”
- The suite can then be invoked
  - Runs the test classes specifically bundled into the suite
- To create a test suite “placeholder” class is created
  - Class with an empty class body
    - ◆ Solely for the purpose of attaching annotations
- Add annotations
  - @RunWith – Specifies which TestRunner implementation to use (instead of TestClassRunner)
  - @Suite.SuiteClasses –Specifies which test classes to bundle into the suite

## Composing Tests Using Suite

- Default runner class scans the class for any methods that have @Test annotation
- Using Suite.class as a runner allows you to manually build a suite containing tests from many classes

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses ({TaxTest.class, BankAccountTest.class})
public class AllMyNewTests {

}
```

## JUnit Test Fixture

- **How ensure the results of a test are repeatable?**
  - There should be a well known and fixed environment in which tests are run so that results are repeatable
- **Examples:**
  - Loading a database with a specific, known set of data
  - Copying a specific known set of files
  - Preparation of input data and setup/creation of fake or mock objects
- **A test fixture is a fixed state of a set of objects used as a baseline for running tests**

## JUnit Method Lifecycle

- The sequence of events for a class containing JUnit-annotated methods is as follows:
  - Assuming two test methods within the class, the method sequence would be the method(s) (if any) annotated with...
    - ◆ @BeforeClass
    - ◆ @Before
    - ◆ @Test - (#1)
    - ◆ @After
    - ◆ @Before
    - ◆ @Test - (#2)
    - ◆ @After
    - ◆ @AfterClass

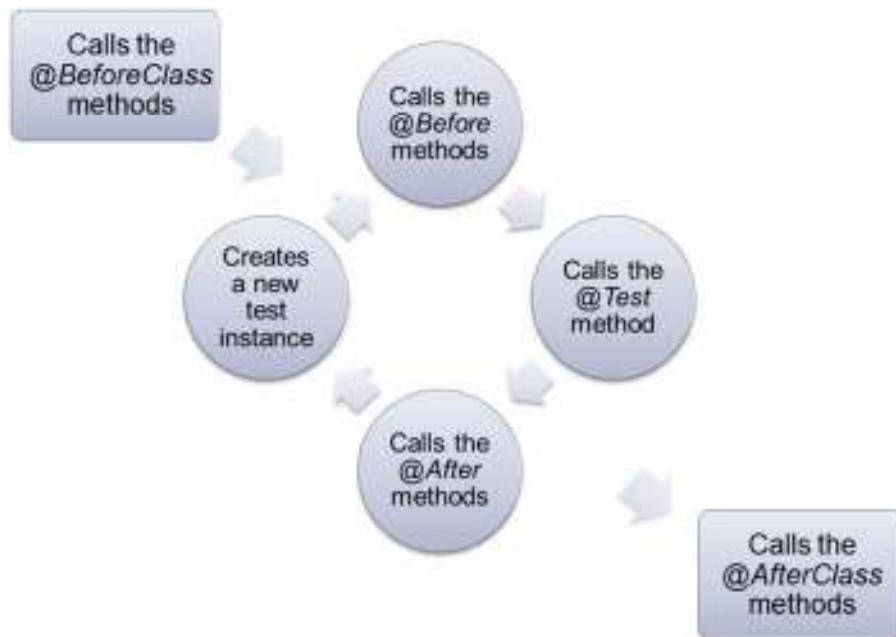
The `setUp()` and `tearDown()` methods are useful as they give the opportunity to single source initialization that would be common to all test methods. In a typical test class the `setUp()` method creates the object to be tested. As JUnit creates a fresh instance per test method it is not necessary to dispose of the object in `tearDown()`. Reserve `tearDown()` for the disposal of resources that were created for the test to run properly.

# Managing Resources with Fixtures

- Similar objects shared by several tests can be initialized and reclaimed using public void methods
  - @BeforeClass
    - ◆ Run before any test has been executed
  - @AfterClass
    - ◆ Run after all the tests have been executed
  - @Before
    - ◆ Run before each test
  - @After
    - ◆ Run after each test

```
public class TaxCalculatorImplTest {  
    private TaxCalculatorImpl calc = null;  
  
    @Before  
    public void prepareTaxCalculator() {  
        calc = new TaxCalculatorImpl();  
    }  
  
    @After  
    public void cleanupTaxCalculator() {  
        calc = null;  
    }  
  
    @Test  
    public void testIncomeBelow38000() {  
    }  
}
```

## Share Expensive Setups



The framework must help us lower the cost of writing tests by reusing code.

Each time you reuse the fixture, you decrease the initial investment made when the fixture was created

## Share Expensive Setups (cont'd)

- **@BeforeClass, @AfterClass annotated methods must be static**

```
public class TaxCalculatorImplTest {  
    private static TaxCalculatorImpl taxCalculator = null;  
    @BeforeClass  
    public static void initializeTaxCalculator() {  
        taxCalculator = new TaxCalculatorImpl();  
    }  
  
    @AfterClass  
    public static void releaseTaxCalculator() {  
        taxCalculator = null;  
    }  
  
    @Test  
    public void shouldUseLowestTaxRateForIncomeBelow38000() {  
        ...  
    }  
}
```

@BeforeClass, @AfterClass annotated methods must be static

# Review

- 1. How do you test protected methods?**
  - 2. How do you test private methods?**
  - 3. How do you test a method that doesn't return anything?**
  - 4. Under what conditions should you test get and set methods?**
  - 5. Why not just use the System.out.println method instead of the assert method?**
- **Challenge Questions:**
    - What are some advantages to using JUnit rather than your current testing techniques?
    - What are some disadvantages?

1. When a method is declared as "protected", it can only be accessed within the same package where the class is defined. In order to test a "protected" method of a target class, you need to define your test class in the same package as the target class.
2. When a method is declared as "private", it can only be accessed within the same class. So there is no way to test a "private" method of a target class from any test class. To resolve this problem, you have to perform unit testing manually. Or you have to change your method from "private" to "protected".
3. Often if a method doesn't return a value, it will have some side effect. There may be a way to verify that the side effect actually occurred as expected.
4. Tests should be designed to target areas that might break. set() and get() methods on simple data types are unlikely to break. So no need to test them. set() and get() methods on complex data types are likely to break. So you should test them.
5. In order to be able to run fully automated tests, the output of a test should be checked by the framework and should not require a human to check the output of the application

**Thank you for attending this course.**

**Please remember to turn in your completed  
course evaluations**

**Questions? Please contact  
[Training@triveratech.com](mailto:Training@triveratech.com)**



---

**Trivera Technologies LLC - Worldwide | Educate. Collaborate. Accelerate!**  
**Global Developer Education, Courseware & Consulting Services**

JAVA | JEE | OOAD | UML | XML | Web Services | SOA | Struts | JSF | Hibernate | Spring | Admin

IBM WebSphere | Rational | Oracle WebLogic | JBoss | TomCat | Apache | Linux | Perl

609.953.1515 direct | [Training@triveratech.com](mailto:Training@triveratech.com) | [www.triveratech.com](http://www.triveratech.com)