# Table of Contents – Fast Track to Java 7

## Fast Track to Java 7 and OO Development
- Course Overview 2
- Course Objectives 3
- Labs 5
- Typographic Conventions 6
- Course Outline 7

## Session 1 - A Simple Java Class and Running a Java Program
- Session Objectives 9
- A Simple Application Class 10
- The `HelloWorld` Program Broken Down 11
- Compiling `HelloWorld` 12
- Note on Comments 13
- **Lab 1.1 - HelloWorld** 14
- Session Review 15

## Session 2 - Java Overview
- Session Objectives 17
- Language and Platform Features 18
  - What is Java? 19
  - Java is Modern and Object-Oriented 20
  - Java is Portable and Safe 21
  - Java has Multiple Platforms 22
- Program Life Cycle 23
  - Java Source and Java Bytecode 24
  - Life Cycle of a Java Program 25
  - Java Programs Insulated From Environment 26
  - Java is Dynamic - The Runtime Process 27
- The Java SE Development Kit (JDK) 28
  - Java Development Kit (JDK) 29
  - The Java API 30
  - Downloading and Installing the JDK 31
  - **Lab 2.1 - The Development Environment** 32
- Session Review 33

## Session 3 - Class and Object Basics
- Session Objectives 35
- Object-Oriented Programming Overview 36
  - What is Object-Oriented Programming? 37
  - What is an Object? 38
  - Important Characteristics of Objects 39
  - About Object-Oriented Programming (OOP) 40
  - What’s a Type? 41
  - Types, Instances, and Property Values 42
  - Classes and Objects 43
  - **Lab 3.1 - Exploring Types** 44
  - Identity and Object References 45
  - **Lab 3.2 - Identity and Object References** 46
- Classes, References, and Instantiation 47
  - The Class in Java 48
<table>
<thead>
<tr>
<th>Class Definition</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Class Definition is a Blueprint</td>
<td>50</td>
</tr>
<tr>
<td>Creating and Referencing Objects</td>
<td>51</td>
</tr>
<tr>
<td>More About Identifiers</td>
<td>52</td>
</tr>
<tr>
<td><strong>Methods and Data in a Class</strong></td>
<td>53</td>
</tr>
<tr>
<td>Behavior and Methods</td>
<td>54</td>
</tr>
<tr>
<td>Invoking Methods</td>
<td>55</td>
</tr>
<tr>
<td>Storing Data in Objects</td>
<td>56</td>
</tr>
<tr>
<td>About Instance Variables</td>
<td>57</td>
</tr>
<tr>
<td>Data Access and Return Values in Methods</td>
<td>58</td>
</tr>
<tr>
<td>Accessing Data (Another Way)</td>
<td>60</td>
</tr>
<tr>
<td>Pretty Pictures</td>
<td>61</td>
</tr>
<tr>
<td>More About Variables</td>
<td>62</td>
</tr>
<tr>
<td>About Java Primitive Data Types</td>
<td>63</td>
</tr>
<tr>
<td>Numeric Literals</td>
<td>64</td>
</tr>
<tr>
<td>Non-Numeric Literals</td>
<td>65</td>
</tr>
<tr>
<td>Strings</td>
<td>66</td>
</tr>
<tr>
<td>Primitive Types are Value Types</td>
<td>67</td>
</tr>
<tr>
<td>Arithmetic Operations</td>
<td>68</td>
</tr>
<tr>
<td>Primitive Type Conversion and Casting</td>
<td>69</td>
</tr>
<tr>
<td>Lab 3.3 - Writing a Class Definition</td>
<td>70</td>
</tr>
<tr>
<td>Session Review</td>
<td>71</td>
</tr>
<tr>
<td><strong>Session 4 – More on Classes and Objects</strong></td>
<td>72</td>
</tr>
<tr>
<td>Session Objectives</td>
<td>73</td>
</tr>
<tr>
<td>Working With Methods and Data</td>
<td>74</td>
</tr>
<tr>
<td>Working Within Methods</td>
<td>75</td>
</tr>
<tr>
<td>Calling Methods</td>
<td>76</td>
</tr>
<tr>
<td>Local Variables</td>
<td>77</td>
</tr>
<tr>
<td>The this Variable and Instance Data</td>
<td>78</td>
</tr>
<tr>
<td>Pretty Pictures</td>
<td>79</td>
</tr>
<tr>
<td>Overloading Methods</td>
<td>80</td>
</tr>
<tr>
<td>Calling Overloaded Methods</td>
<td>81</td>
</tr>
<tr>
<td>The toString() Method</td>
<td>82</td>
</tr>
<tr>
<td><strong>Encapsulation and Access Control</strong></td>
<td>83</td>
</tr>
<tr>
<td>Encapsulation: Black Boxes</td>
<td>84</td>
</tr>
<tr>
<td>Key Advantages of Encapsulation</td>
<td>87</td>
</tr>
<tr>
<td>Program Correctness</td>
<td>88</td>
</tr>
<tr>
<td>Access Control</td>
<td>89</td>
</tr>
<tr>
<td>Access for Data Members and Methods</td>
<td>90</td>
</tr>
<tr>
<td>Private Access</td>
<td>91</td>
</tr>
<tr>
<td>Public Access</td>
<td>92</td>
</tr>
<tr>
<td>Lab 4.1 - Encapsulation</td>
<td>93</td>
</tr>
<tr>
<td><strong>Constructors</strong></td>
<td>94</td>
</tr>
<tr>
<td>Constructors</td>
<td>95</td>
</tr>
<tr>
<td>Using Constructors</td>
<td>96</td>
</tr>
<tr>
<td>Explicit Constructor Call</td>
<td>97</td>
</tr>
<tr>
<td>Lab 4.2 - Adding Constructors to a Class</td>
<td>98</td>
</tr>
<tr>
<td><strong>Static or Class Members</strong></td>
<td>99</td>
</tr>
<tr>
<td>Static Data Members and Methods</td>
<td>100</td>
</tr>
<tr>
<td>Declaring Static Members</td>
<td>101</td>
</tr>
<tr>
<td>Accessing Static Members</td>
<td>102</td>
</tr>
<tr>
<td>Accessing Data in Static Methods</td>
<td>103</td>
</tr>
</tbody>
</table>
### Session 5 - Flow of Control

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session Objectives</td>
<td>118</td>
</tr>
<tr>
<td>Branching Statements</td>
<td>120</td>
</tr>
<tr>
<td>Program Execution Sequence in Java</td>
<td>121</td>
</tr>
<tr>
<td>The Comparison Operators</td>
<td>122</td>
</tr>
<tr>
<td>The Logical Operators</td>
<td>123</td>
</tr>
<tr>
<td>if-else Statement</td>
<td>124</td>
</tr>
<tr>
<td>switch Statement</td>
<td>125</td>
</tr>
<tr>
<td>Iteration Statements</td>
<td>126</td>
</tr>
<tr>
<td>while Statement</td>
<td>127</td>
</tr>
<tr>
<td>do-while Statement</td>
<td>128</td>
</tr>
<tr>
<td>for Statement</td>
<td>129</td>
</tr>
<tr>
<td>break Statement</td>
<td>130</td>
</tr>
<tr>
<td>continue Statement</td>
<td>131</td>
</tr>
</tbody>
</table>

### Lab 5.1 - Data Validation

| Session Review                                                        | 132  |

### Session 6 - Strings and Arrays

| Session Objectives                                                   | 134  |
| String and StringBuffer/StringBuilder                                | 136  |
| Using Strings                                                        | 137  |
| Changing Strings                                                     | 138  |
| Classes StringBuffer and StringBuilder                               | 139  |
| StringBuffer and StringBuilder                                       | 140  |
| Regular Expressions                                                  | 141  |

| Arrays                                                                | 142  |
| Creating Arrays and Accessing Elements                               | 143  |
| Arrays of Object References                                          | 145  |
| Array of Strings                                                     | 146  |
| args Array                                                           | 147  |
| Iterating Over Arrays                                                | 148  |
| varargs                                                              | 149  |
| Lab 6.1 - Arrays                                                     | 150  |

### Session 7 - Packages

| Session Objectives                                                   | 153  |

---

*Note: The table is a simplified representation of the contents of the document.*
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages Overview</td>
<td>155</td>
</tr>
<tr>
<td>Dealing with Complexity</td>
<td>156</td>
</tr>
<tr>
<td>Packages</td>
<td>157</td>
</tr>
<tr>
<td>package Statement</td>
<td>158</td>
</tr>
<tr>
<td>The Default (or Unnamed) Package</td>
<td>159</td>
</tr>
<tr>
<td>import Statement</td>
<td>160</td>
</tr>
<tr>
<td>The import Statement</td>
<td>161</td>
</tr>
<tr>
<td>Importing a Complete Package</td>
<td>162</td>
</tr>
<tr>
<td>Importing a Single Package Member</td>
<td>163</td>
</tr>
<tr>
<td>Using the Fully Qualified Name</td>
<td>164</td>
</tr>
<tr>
<td>Standard Imports</td>
<td>165</td>
</tr>
<tr>
<td>Resolving Naming Conflicts</td>
<td>166</td>
</tr>
<tr>
<td>Creating Packages</td>
<td>167</td>
</tr>
<tr>
<td>Creating a Package</td>
<td>168</td>
</tr>
<tr>
<td>Access Control for Class Members</td>
<td>169</td>
</tr>
<tr>
<td>Access Control for Classes</td>
<td>170</td>
</tr>
<tr>
<td>Summary - Using Packages</td>
<td>171</td>
</tr>
<tr>
<td>Finding Classes</td>
<td>172</td>
</tr>
<tr>
<td>Tools Must Locate Class Definitions</td>
<td>173</td>
</tr>
<tr>
<td>Organizing Files and Packages</td>
<td>174</td>
</tr>
<tr>
<td>Class Path</td>
<td>175</td>
</tr>
<tr>
<td>Classpath Example</td>
<td>176</td>
</tr>
<tr>
<td>What is a JAR?</td>
<td>178</td>
</tr>
<tr>
<td>Classpath Example</td>
<td>179</td>
</tr>
<tr>
<td>Lab 7.1 - Packages</td>
<td>180</td>
</tr>
<tr>
<td>Session Review</td>
<td>181</td>
</tr>
<tr>
<td>Session 8 - Composition and Inheritance</td>
<td>182</td>
</tr>
<tr>
<td>Session Objectives</td>
<td>183</td>
</tr>
<tr>
<td>Composition</td>
<td>184</td>
</tr>
<tr>
<td>Dealing With Complexity and Composition</td>
<td>185</td>
</tr>
<tr>
<td>Composition</td>
<td>186</td>
</tr>
<tr>
<td>Delegation</td>
<td>187</td>
</tr>
<tr>
<td>Benefits of Composition</td>
<td>188</td>
</tr>
<tr>
<td>Issues with Composition</td>
<td>189</td>
</tr>
<tr>
<td>About Object Relationships</td>
<td>190</td>
</tr>
<tr>
<td>Other Kinds of Relationships</td>
<td>191</td>
</tr>
<tr>
<td>Lab 8.1 - Composition (Optional)</td>
<td>192</td>
</tr>
<tr>
<td>Inheritance</td>
<td>193</td>
</tr>
<tr>
<td>Inheritance and Dealing With Complexity</td>
<td>194</td>
</tr>
<tr>
<td>Inheritance Hierarchy</td>
<td>195</td>
</tr>
<tr>
<td>The extends Keyword</td>
<td>196</td>
</tr>
<tr>
<td>Inheriting from the Superclass</td>
<td>197</td>
</tr>
<tr>
<td>Inheritance and Superclass Data Members</td>
<td>198</td>
</tr>
<tr>
<td>A Subclass IS-A Superclass</td>
<td>199</td>
</tr>
<tr>
<td>Accessing Superclass Members</td>
<td>200</td>
</tr>
<tr>
<td>Constructors and Inheritance</td>
<td>201</td>
</tr>
<tr>
<td>Final Classes</td>
<td>202</td>
</tr>
<tr>
<td>Lab 8.2 - Inheritance</td>
<td>203</td>
</tr>
<tr>
<td>Overriding and Polymorphism</td>
<td>204</td>
</tr>
<tr>
<td>Changing Behavior with Method Overriding</td>
<td>205</td>
</tr>
<tr>
<td>OO Concepts - Polymorphism</td>
<td>206</td>
</tr>
</tbody>
</table>
Session Objectives

Overview
Java Collections Framework Overview
java.util Collection Interfaces
Collection Interface
Generics and Type-Safe Collections

List and ArrayList
List Interface
ArrayList
Using ArrayList - Example
The for-each Construct

Autoboxing and Collections of Object
Autoboxing
Using Autoboxing/Unboxing - Example
Summarizing Collection Features
Collections of Object
Issues with Collection of Object
Lab 11.1 - Using Collections

Other Collection Types
Set Interface
Using Sets
Map Interface
HashMap
Generic HashMaps
Creating and Using HashMap
Iterating Through a HashMap
Lab 11.2 - Using Sets

Iterator
Processing Items with an Iterator
Iterator Interface
Using Iterator - Example

[Optional] More About Generics
What Are Generics
Declaring a Generic Class
Summary - Basic Generics Usage
Using Generics - Example
Inheritance with Generic Types
Inheritance with Generic Types
Assignment with Generic Types
Wildcard Parameter Types
Generic Methods

[Optional] The Collections Class
Collections Class
Unmodifiable Wrappers
Unmodifiable Example
Checked Interface Example
Algorithms
Sort Example
Session Review

Session 12 - Database Access with JDBC and JPA
<table>
<thead>
<tr>
<th>Session Objectives</th>
<th>316</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JDBC Overview</strong></td>
<td>317</td>
</tr>
<tr>
<td>What is JDBC?</td>
<td>318</td>
</tr>
<tr>
<td>JDBC Architecture</td>
<td>319</td>
</tr>
<tr>
<td>The Fundamental JDBC API</td>
<td>320</td>
</tr>
<tr>
<td>Common JDBC Types</td>
<td>321</td>
</tr>
<tr>
<td>Naming Databases with URLs</td>
<td>322</td>
</tr>
<tr>
<td>The Item Database Table</td>
<td>323</td>
</tr>
<tr>
<td>Database Connection - Example</td>
<td>324</td>
</tr>
<tr>
<td>Using <strong>Statement</strong> - Example</td>
<td>325</td>
</tr>
<tr>
<td>Using <strong>PreparedStatement</strong> - Example</td>
<td>326</td>
</tr>
<tr>
<td>Summary</td>
<td>327</td>
</tr>
<tr>
<td><strong>JPA Overview</strong></td>
<td>328</td>
</tr>
<tr>
<td>Java Persistence API (JPA) Overview</td>
<td>329</td>
</tr>
<tr>
<td>JPA Architecture – High Level View</td>
<td>330</td>
</tr>
<tr>
<td>JPA Architecture – Programming View</td>
<td>331</td>
</tr>
<tr>
<td>Working with JPA</td>
<td>332</td>
</tr>
<tr>
<td>Entity Classes</td>
<td>333</td>
</tr>
<tr>
<td>MusicItem Entity Class</td>
<td>334</td>
</tr>
<tr>
<td>Annotation Overview</td>
<td>335</td>
</tr>
<tr>
<td>Additional MusicItem Annotations</td>
<td>336</td>
</tr>
<tr>
<td><strong>Lab 12.1 - Mapping an Entity Class</strong></td>
<td>337</td>
</tr>
<tr>
<td>The Persistence Unit</td>
<td>338</td>
</tr>
<tr>
<td>persistence.xml Structure</td>
<td>339</td>
</tr>
<tr>
<td>The EntityManager</td>
<td>340</td>
</tr>
<tr>
<td>JPA EM and EMF Example</td>
<td>341</td>
</tr>
<tr>
<td>Working with Transactions</td>
<td>342</td>
</tr>
<tr>
<td>Complete JPA Example</td>
<td>343</td>
</tr>
<tr>
<td>Summary</td>
<td>344</td>
</tr>
<tr>
<td><strong>Lab 12.2 - Using JPA</strong></td>
<td>345</td>
</tr>
<tr>
<td>Persisting a New Entity</td>
<td>346</td>
</tr>
<tr>
<td>Updating a Persistent Instance</td>
<td>347</td>
</tr>
<tr>
<td>Removing an Instance</td>
<td>348</td>
</tr>
<tr>
<td>Executing a Query</td>
<td>349</td>
</tr>
<tr>
<td><strong>Lab 12.3 - Insert/Query Demo</strong></td>
<td>350</td>
</tr>
<tr>
<td>Session Review</td>
<td>351</td>
</tr>
<tr>
<td><strong>Session 13 - Additional Language Features</strong></td>
<td>352</td>
</tr>
<tr>
<td>Session Objectives</td>
<td>353</td>
</tr>
<tr>
<td><strong>Assertions</strong></td>
<td>354</td>
</tr>
<tr>
<td>Assertions Defined</td>
<td>355</td>
</tr>
<tr>
<td>Assertion Uses</td>
<td>356</td>
</tr>
<tr>
<td>Assertion Non-Uses</td>
<td>357</td>
</tr>
<tr>
<td>Assertion Syntax</td>
<td>358</td>
</tr>
<tr>
<td>Using Assertions to Check State - Example</td>
<td>359</td>
</tr>
<tr>
<td>Using Assertions to Check Flow of Control</td>
<td>360</td>
</tr>
<tr>
<td>Enabling/Disabling Assertions at Runtime</td>
<td>361</td>
</tr>
<tr>
<td>Enabling/Disabling Assertions - Examples</td>
<td>362</td>
</tr>
<tr>
<td>What They Look Like at Runtime</td>
<td>363</td>
</tr>
<tr>
<td><strong>Type-Safe Enums</strong></td>
<td>364</td>
</tr>
<tr>
<td>Enumerated Types Defined</td>
<td>365</td>
</tr>
<tr>
<td>Problems with <code>int</code> Enumerated Types</td>
<td>366</td>
</tr>
<tr>
<td>The <code>enum</code> Keyword</td>
<td>367</td>
</tr>
</tbody>
</table>
Fast Track to Java 7 and OO Development

The Java Developer Education Series
Course Overview

- An introductory Java course that starts with basic principles
  - Provides a solid foundation in the concepts and practices for writing good object-oriented systems in Java
  - Provides knowledge needed to productively use core Java technology for programming
  - Including database access with JDBC/JPA

- Be prepared to work hard and learn a great deal!

- The course contains numerous hands-on labs
  - They exercise all the important concepts discussed
  - The lab solutions for the course are provided to you

- The course covers all core features of Java
  - It supports all recent versions of Java

Notes:

- The Java platform has evolved rapidly
  - However, many people are still using older versions
- The labs are designed to support those using Java 5 and any later versions

- Java 6 and 7 primarily introduced advanced features for running Java programs, and did not change the API, though it did add to it
  - However, those features are mostly beyond the scope of this course
  - Accordingly, this course can be used in any of these environments
Course Objectives

- Learn Java's architecture and uses
- Understand Java language basics
- Compile and execute Java programs with development tools such as javac and java
- Learn object-oriented (OO) programming and the object model
  - Understand the differences between traditional programming and object-oriented programming
  - Understand important OO principles such as composition, inheritance and polymorphism
- Use Java packages to organize code
- Understand interfaces, their importance, and their uses
Course Objectives

- Learn (and practice!) Java naming conventions and good Java coding style
- Create well-structured Java programs
- Use core Java API class libraries
- Understand how exceptions are used for error handling
- Understand the basics of database access with JDBC and JPA
- Learn the basics of the Collections Framework
- See some of the new/advanced Java language features
- Understand and use basic I/O streams (optional)
### Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - The detailed lab instructions are separate from the main student manual
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

```
Lab STOP
```

---

Notes:
Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:
  
  ```java
  JavaInstructor teacher = new JavaInstructor()
  ```

- Code fragments use the same font, e.g. `teacher.teach()`
- We **bold/color** text for emphasis
- Filenames are in italics, e.g. `JavaInstructor.java`
- We sometimes denote more info in the notes with a star *
- Longer code examples appear in a separate code box - e.g.

```java
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
    public void teach() {
        System.out.println("Java is way cool");
    }
}
```

Notes:
# Course Outline

- Session 1: A Simple Java Program and the JDK
- Session 2: Java Overview
- Session 3: Class and Object Basics
- Session 4: More on Classes and Objects
- Session 5: Flow of Control
- Session 6: Strings and Arrays
- Session 7: Packages and Access Protection
- Session 8: Composition and Inheritance
- Session 9: Interfaces
- Session 10: Exceptions
- Session 11: Collections and Generics
- Session 12: Database Access with JDBC and JPA
- Session 13: Additional Language Features
- Session 14: Java I/O (optional)
- Appendix: JDBC
Session 1 - A Simple Java Class and Running a Java Program
Session Objectives

- Look at a simple Java program, and how it is compiled and run

- Set your computer up for Java development
  - Setting paths, environment variables, etc.

- Use a text editor to type in a simple Java class

- Use the Java Development Kit to compile and run the code
A Simple Application Class

- To get started, we'll look at a very simple Java application, that displays the string "Hello World"
  - This is a simple non-graphic standalone application that runs on its own
- All programs in Java are just a class with a main method in it
  - The HelloWorld class needs to appear in a file HelloWorld.java
- The main method is a special method that is the starting point for every Java application
  - It must be declared as shown, and has to appear in a class
  - System.out.println how you print things to the console

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

Notes:

- Java classes can be deployed into several runtime execution environments, including:
  - Standalone applications that are run explicitly, like our Hello World application.
  - Servlets, which are run in a Web server.
  - Applets, which are downloaded from a Web server and run in a Web browser.
- We will concentrate on applications in this course.
- main is the special method which is the starting point for every standalone Java application.
  - Applets and servlets work a little differently
- The main method has to appear inside a class.
  - main takes an argument – an array of Strings.
  - This array holds arguments that are given on the command line when the application is run.
- System.out.println causes something to be printed to System.out, which is the standard output (the console) – more on this later.
The HelloWorld Program Broken Down

- We'll cover these details in more depth later

```java
// a simple application to display "Hello World"
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Class definition

Block

Method definition

Comment

Method call

Method argument

Semicolon
Compiling HelloWorld

- To use a Java class, you have to **compile** the Java code

  C:\StudentWork> javac HelloWorld.java

  - After compiling, a file called **HelloWorld.class** (containing Java bytecode) is created

- You use the Java Virtual Machine (**JVM**), to run a Java program
  
  - The JVM comes with Java as the **java** executable

  C:\StudentWork> java HelloWorld

  - This would produce the output "Hello World" printed in your command prompt window

**Notes:**

- HelloWorld.class is the bytecode for the program produced by the compiler

- We will talk more about all these details later in the course

- For now, we're giving you enough information to just run the program
**Note on Comments**

- Java has 3 kinds of comments
  - Single line comment: Start with `// ...`
  - Multiple lines using `/* ... */`
  - Javadoc comments (see note) using `/** ... */`

```java
/**
 * This class prints "Hello World" to standard output
 */
class HelloWorld {
   // this comment starts mid-line
   public static void main(String[] args) {
      System.out.println("Hello World");
      // we often comment out a line code this way
      // System.out.println("Bye");
   }
}
/** Note that this class still needs a lot of work. We need to add all sorts of interesting things, and show how cool Java has become. */
```

**Notes:**

- Comments are text that is ignored by the Java compiler
  - Comments can start anywhere on a line

- Javadoc comments are used in generating API documentation directly from the source code
  - These comments precede the items that they are documenting in the code

- A tool called **javadoc** reads these comments
  - Along with your source code
  - javadoc also understands a number of parameters that you can embed in these comments to enhance the generated documentation
  - We cover this tool in an appendix
Lab 1.1 - HelloWorld

In this lab, we will compile and run a very simple Java program.
Session Review

1. Why must you set your path include the `<java>\bin` directory?

2. What is the purpose of the `main` method?

3. What is the signature of the `main` method?

4. How do you print something to standard output (the console)?

5. What tool do you use to compile Java source code?

6. What tool do you use to run compiled Java code?

Notes:
Session 5 - Flow of Control

Branching Statements
Iteration Statements
Session Objectives

◆ Outline the comparison and boolean operators in Java
◆ Discuss branching statements and the operators used with them
  - if, if-else, switch
◆ Discuss iteration (looping) statements
  - while, do-while, for
  - break, continue
◆ Use flow of control logic to perform data validation in an object
Branching Statements

Iteration Statements
Program Execution Sequence in Java

- Unless directed otherwise, statements in Java are executed in sequence

- Java has a number of statements to change the flow of control
  - Branching/selection statements choose one of several flows of control:
    - *if*, *if-else*, and *switch*
  
    - Iteration statements specify looping
      - *while*, *do-while*, and *for*

  - Jump statements transfer control unconditionally
    - *break*, *continue*, *return*

Notes:

- All of these statements are modeled after their equivalent in the C programming language
The Comparison Operators

- Selection and iteration statements are based on the results of comparisons that return a true or false result.

- The comparison operators compare numerical values and produce boolean results.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

3 == 5 (== false)
3 != 5 (== true)
3 < 5 (== true)
3 > 5 (== false)
5 <= 5 (== true)
5 >= 5 (== true)
The Logical Operators

- The logical operators compare boolean values and produce boolean results.
- In the table below, assume we have:

```java
boolean t = true;
boolean f = false;
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>f &amp; t (== false, t evaluated)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>conditional AND</td>
<td>f &amp;&amp; t (== false, t not evaluated)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>exclusive OR</td>
<td>t ^ t (== false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>t ^ f or f ^ t (== true)</td>
</tr>
<tr>
<td>!</td>
<td>NOT</td>
<td>!t (== false)</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>f == f (== true)</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>t != t (== false)</td>
</tr>
</tbody>
</table>

Notes:
if-else Statement

- The if / if-else statements control whether or not a statement is executed, based on the value of an expression
  - The else is optional
    - It's executed only if the expression evaluates to false

```java
if ( Expression ) Statement else Statement
```

```java
int i = 1;
if (i == 0) {
    System.out.println("i equals 0");
} else {
    System.out.println("i is not 0");
}
```

Notes:

- The parentheses are required around the expression
  - The expression must have a boolean type.
- The statement may be a block of statements, enclosed in `{ }`.
- The expression is evaluated, and if it is true the statement is executed.
- RECOMMENDATION: always use the block form to avoid potential maintenance errors.
- You can chain them.
  ```java
  int k = 0;
  if (k == 0)
  {
      System.out.println("k equals 0");
  }
  else if (k == 1)
  {
      System.out.println("k equals 1");
  }
  else
  {
      System.out.println("k is neither 0 nor 1");
  }
  ```
**switch Statement**

- The **switch** statement extends the if statement, to allow testing for more than one value
  - Can only switch on byte, short, int, char, and enum
  - Java 7+ allows switch on string values
  - default case is optional and gets control when no case matches
- Note: If the break statement is **not present**, execution continues on to the code in the next case

```java
int i = 1; // This would usually be initialized elsewhere
switch ( i ) { // Execute a case based on value of i
    case 1:
        System.out.println("i is 1");
        break;
    case 2:
        System.out.println("i is 2");
        break;
    default:
        System.out.println("i is large");
        break;
}
```

**Notes:**

- The general form of the switch is:
  ```java
  switch ( variable ) {
      case value1:
          statement(s);
          break;
      case value2:
          statement(s);
          break;
      default:
          statement(s);
  }
  ```
- A switch statement may execute one or more of several alternatives. When executed:
  - The expression in the switch is evaluated and compared to the case constants.
  - If one of the case constants is equal to the value of the expression, execution continues at that statement (i.e., a jump to that point in the code occurs).
  - If none of the case constants match, the statements after the default label, if present, are executed.
Iteration Statements

Branching Statement
Iteration Statements
while Statement

- The while statement creates a loop – it has the form:

```
while ( Expression ) Statement
```

```java
int index = 10;
while (index > 0) // stop looping when index reaches 0
{
    // do some work
    index = index - 1;
}
```

```java
ResultSet rs = ...; // This is a JDBC ResultSet object
while (rs.next()) // rs.next() eventually returns false
{
    // process next row in result set
}
```

Notes:
- The expression must have a boolean type.
- The expression is evaluated repeatedly, and as long as it evaluates to true, the statement is executed.
- When the expression is no longer true, execution stops.
- The expression is evaluated before each execution of the statement. Thus the statement may be executed zero times.
**do-while Statement**

- The **do-while** statement creates a loop – it has the form:

```java
do Statement while ( Expression );
```

```java
int index = 10;
do {
   // do some work
   index = index - 1;
} while (index > 0); // stop looping when index reaches 0
```

- What is the difference between while and do-while?
  - What happens if the index is initially set to 0, before the do-while is executed?
  - The expression is evaluated **after** each execution of the statement.
    - Thus the statement in a do-while is executed **at least once**.
    - The statement in a while may never be executed

- The expression must have a **boolean** type.
- The statement is executed until the expression becomes **false**.
**for Statement**

- The for statement creates a loop – it has the form:

  \[
  \text{for ( Initialization}_{\text{opt}}; \text{ Expression}_{\text{opt}}; \text{ Increment}_{\text{opt}} \) \text{ Statement}
  \]

  // Print values from 0 to 3
  for (int i = 0; i <= 3; i++)
  {
    System.out.print(i + " ");
  }

- Java 5 introduced another version of the for loop – called for-each
  - We'll look at this later

**Notes:**

- The initialization is executed first and specifies initialization of the loop.
- If the expression is true, the statement is executed.
- The loop is exited when the expression becomes false.
- The increment expression is executed after each iteration, i.e., after the statement is executed.
- The for loop is often used to iterate through an array. We will discuss arrays later.
break Statement

- The `break` statement transfers control to the end of the enclosing loop (for, while, do-while) or switch statement
- It has the form:
  
  ```java
  break;
  ```

// this method scans values looking for a specific one
void findValue(int value) {
    for (int i = 0; ; i++) // No terminate expression
    {
        if (i == value)
        {
            System.out.println("got the value");
            break; // stop looping, found the value
        }
    }
    // control is here after the break
}

Notes:
- `break` statements can carry a label
- The form of the labeled `break` is:
  ```java
  break label;
  ```

- Here is an example

```java
outer: // 'outer' is not a reserved word, just a label
for (int i = 0; i < 100; i++) {
    for (int k = 0; k < 100; k++) {
        if (/* some condition */) {
            break outer; // exit outer loop
        }
    }
}
// control is here after the break
```
The `continue` statement exits the current iteration of the loop (for, while, do-while)
- It continues with the next iteration

It has the form:
```
continue;
```

```java
// here we use continue to print out even numbers only
for (int i = 1; i <= 10; i++)
{
    if ((i % 2) != 0) // if not divisible by 2 (not even)
        continue;        // exit this iteration of the loop
    System.out.println(i);
}
```

Unlike `break`, `continue` does not exit the loop entirely; it just exits the current iteration of the loop, and continues with the next iteration.

`continue` statements can carry a label, and the form of the labeled `continue` is:
```
continue label;
```

```java
outer:
    for (int i = 0; i < 100; i++) {
        for (int k = 0; k < 100; k++) {
            if (k == 17) {
                // exit this iteration of outer loop
                continue outer;
            }
        }
        System.out.println(k);  // Otherwise normal looping goes on
    }
```
Lab 5.1 - Data Validation

In this lab, we will add data validation to a class
Session Review

1. Name the two AND operators in Java. What is the difference between them?
2. True or false: if and if-else statements require the use of blocks
3. What data types can you use to control a switch statement?
4. What is the difference between while and do-while?
5. What is the difference between break and continue?
Session 9 - Interfaces
<table>
<thead>
<tr>
<th>Session Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>✷ Understand the similarities between interface types and class types</td>
</tr>
<tr>
<td>✷ Use interface types the same way that class types are used</td>
</tr>
<tr>
<td>✷ Explain the role that interfaces play in &quot;programming by contract&quot;</td>
</tr>
<tr>
<td>✷ Define and implement an interface</td>
</tr>
</tbody>
</table>

Notes:
What if All You Have to Share is an Idea

- It often happens that you know what a type will do
  - You know what it's behavior (methods) are
  - But you don't know how that type will do it
- Or there may be many related types that will implement some behavior differently
  - And you want to be able to treat all those varieties the same

- For example, you know a Timepiece needs to display the time
  - But there may be many variations of Timepiece that display time completely differently, and have no shared implementation
  - A clock, a sundial
  - What about a cell phone?
  - One strategy is to make Timepiece an interface
Interface Types

- Java interfaces allow you to specify a type that is totally separate from any implementation
  - It is an abstract type that can specify behavior
  - It embodies the idea of what a type is, but not its implementation
  - Interfaces are often used to define roles played by objects
  - Whereas a class can define how a type fulfills the role (via instance data and method implementation)

- An interface defines a type that is similar to a class
  - It can have method definitions, but all methods are abstract
  - Interfaces cannot be instantiated with the new keyword
  - It can also have properties, but all properties are static final constants

A Person can play roles such as Programmer or Instructor.
- If Programmer is an interface with code and test methods, and
- Instructor is an interface with teach and debugLabs methods,
- then Person can play the role of Programmer if it implements the code and test methods, and Person can also play the role of Instructor if it implements the teach and debugLabs methods.
- This means that you can use a Programmer or Instructor reference with this Person object, i.e., you can treat a Person as simply a Programmer or Instructor.

The use of interfaces is often called Programming by Contract
Interface Definitions

- An interface definition uses the `interface` keyword, in a way similar to the `class` keyword
  - We'll show examples of this using types for a shipping company

```java
// definition of interface Moveable
package com.mycompany.shipping;
public interface Moveable {
    // ...
}
```

- Interface methods are declared without a body
  - They are implicitly abstract, and have no implementation

```java
package com.mycompany.shipping;
public interface Moveable {
    public void moveTo(String dest);
}
```
### The implements Keyword

- Interfaces are used via the `implements` keyword in class definitions

- When you implement an interface you **must** provide implementations for every method of the interface
  - If you don't, the class will not compile

```java
import com.mycompany.shipping.Moveable;
public class PosterTube implements Moveable {
    // provides an implemented moveTo(String) method
    public void moveTo(String dest) {
        // PosterTubes's implementation
    }
}
```

**Note:**
- **NOTE:** when a class implements an interface, it can include concrete implementations or abstract implementations of the interface methods.
  - If some of those methods are abstract, then the class is abstract and cannot be instantiated.
  - Once a subclass provides concrete implementations of all the abstract methods it can be instantiated.

- An "empty" or no-op method is an implementation.
```
public void moveTo(String dest) {
}
```
Example of Using Interface Types

class MovingCompany {
    // Moveable is an interface type
    Moveable[] goods = null;

    MovingCompany(Moveable[] goodsIn) {
        goods = goodsIn;
    }

    void deliverAllGoods(String location) {
        for (Moveable m : goods) {
            // Iterate with for-each
            m.moveTo(location);
        }
    }
}

Notes:
Interface Types - Revisited

Keep in mind that an interface type is very similar to a class type

- A instance of a class that implements an interface might only be "viewed" by other objects as that interface type, and not by its actual class type

- For example, a moving company might not care about what exactly it is moving, just that the items are Moveable.
- Moveable is a type, but it is an interface types, not class type
Interface Types - Revisited

- Interface types can be used as **reference variable types**, but cannot be instantiated or used as object types

```java
Moveable m = new PosterTube();
```

- Interface types can be used as **parameters to methods**

```java
public void moveObject(Moveable m) {... }
```

- Interface types can be used as **return types** from methods

```java
public Moveable getMovedObject()
{
    return m;
}
```

Notes:

- You will often not know (or care) about what kind of type you are working with (class type or interface type).
  - You work with both of them the same way – basically, you call methods on them.
Extending Interfaces

- You can declare **subinterfaces** that extend other interfaces, using the **extends** keyword

  - This is the **IS-A** relationship again, this time with interfaces
  - Unlike class inheritance, you can extend multiple interfaces
  - A class that implements a subinterface must implement all the interfaces that the subinterface extends (**IS-A**)

```java
// A generic movable
public interface Moveable {
    public void moveTo(String dest);
}

// A moveable that will go onto a truck
public interface Carton extends Moveable {
    public float getSize();
}
```
Implementing Extended Interfaces

```java
// PosterTube implements Carton and thus Moveable, also
public class PosterTube implements Carton {
    // from interface Moveable
    public void moveTo(String dest) {
        // ...
    }
    // from interface Carton
    public float getSize() {
        // ...
    }
}
```
Example of Using Interface Types

```java
interface Moveable { /* ... */ }
interface Carton extends Moveable { /* ... */ }

class Car implements Moveable { /* ... */ }
class PosterTube implements Carton { /* ... */ }
class ShippingBox implements Carton { /* ... */ }
class WardrobeBox extends ShippingBox { /* ... */ }
```

This shows the relationships between the class and interface types used in this example.

In the example, a `PosterTube` object can be referenced by variables of type:
- `PosterTube`.
- `Object`.
- `Moveable`.
- `Carton`.
- That is, a `PosterTube` can be treated as a `PosterTube`, and of course it can be treated as a plain old `Object`. A `PosterTube` can also be treated as a `Moveable` or a `Carton`.
Example of Using Interface Types

class MovingCompany {
    // Carton and Moveable are interface types
    Carton[] cartons = null;
    Moveable[] goods = null;

    MovingCompany(Carton[] cartonsIn, Moveable[] goodsIn) {
        cartons = cartonsIn;
        goods = goodsIn;
    }

    void deliverAllGoods(String location) {
        float totalSize = 0.0F;
        for (int i = 0; i < cartons.length; i++) {
            totalSize += cartons[i].getSize();
            cartons[i].moveTo(location);
        }
        for (Moveable m : goods) {
            m.moveTo(location);
        }
    }
}

Notes:

- Notice in the example above that all the types involved are interface types.
  - That is, the MovingCompany doesn't really know (or care) what exactly it is moving, just that the objects are Cartons and Moveables.
  - This simplifies the point of view of the MovingCompany to just what's needed and no more. Furthermore, the MovingCompany class is flexible, in that the exact types that it's dealing with are not hardcoded.
Example of Using Interface Types

class GetMoving
{
    public static void main(String[] args)
    {
        Carton[] boxes = { new PosterTube(), new ShippingBox(),
        new WardrobeBox() };

        Moveable[] bigStuff = { new Car() };
        MovingCompany acme = new MovingCompany(boxes, bigStuff);
        acme.deliverAllGoods("San Francisco");
    }
}
Interfaces are Abstract

- Interfaces are implicitly abstract
  - You can also declare this explicitly, though you generally don't
  - The definition below is equivalent to one without using abstract

```java
// 'abstract' legal, but generally not included
public abstract interface Moveable {
    // ...
}
```

- abstract also legal on methods but generally not used
- The definition below is equivalent to one without using abstract

```java
public interface Moveable {
    // 'abstract' legal, but generally not included
    public abstract void moveTo(String dest);
}
```

Notes:
Data Members in Interfaces

- Interfaces can't declare instance data
  - If you need instance data, you need to use a class

- Interface data members are **implicitly** both **static** and **final**, and usually are declared as such, for clarity
  - It is legal to leave out final and static, but the compiler will just add them in for you

```java
package com.mycompany.shipping;
public interface Moveable
{
    // static and final are usually included
    // note the ALL_CAPS convention for class constants

    public static final String HOME_LOCATION = "HQ Office";
    public void moveTo(String dest);
}
```

Notes:

- Another example is the WindowConstants interface:
  - It contains constants for the possible actions that can be taken when a user requests a window be closed.
  - These constants are used with the `setDefaultCloseOperation` method when designing (GUIs) with Swing.

```java
package javax.swing;
public interface WindowConstants
{
    // static and final are usually included
    // note the ALL_CAPS convention for constants
    public static final int DISPOSE_ON_CLOSE = 2;
    public static final int DO_NOTHING_ON_CLOSE = 0;
    public static final int HIDE_ON_CLOSE = 1;
}
```
Implementing Multiple Interfaces

- A class can implement more than one interface

```java
public class Car implements Moveable, Serviceable {
   // from interface Moveable
   public void moveTo(String dest) {
       // ...
   }
   // from interface Serviceable
   public void serviceEngine() {
       // every 30,000 miles turn on "Check Engine" light
   }
}
```

- This is a useful property of interfaces
  - Has many of the advantages of multiple inheritance, which Java doesn't support
  - Is much less complex because the interfaces just declare a protocol, not an implementation

Notes:
- Interfaces are used in place of multiple inheritance. Consider the "Pegasus problem," in which we wish to define a flying horse.
  - If multiple inheritance was available, you might subclass both Bird and Horse. However, if there were any variables or methods defined in both Bird and Horse, you would have a duplicate/conflict situation and the compiler would have to resolve this somehow. Already it's getting a bit complicated.
  - If only single inheritance was available, would you extend Bird or Horse? Isn't Pegasus really a horse that has some of the characteristics of birds? Do all birds fly?
  - Using interfaces, we can define a role (a set of abstract methods) named FlyingAnimal, that can be used with any creature that can fly -- we can use it for Pegasus too.

```java
interface FlyingAnimal

class Pegasus
   extends Horse
   implements FlyingAnimal
```
Lab 9.1 - Interfaces

In this lab, we will work with interfaces - both creating and using them.
Session Review

1. How does "programming by contract" apply to interfaces?
2. What keyword is used for a class to "sign an interface contract?"
3. True or false: interfaces can be placed in packages.
4. Explain the difference between interfaces and abstract classes.
5. True or false: interfaces can exhibit inheritance characteristics similar to classes.
6. True or false: interfaces can only have a default or no-argument constructor.

Notes: