

# Java Track

## Getting Started

### Introduction

The Java Track will explain how to program in Java with specifics on the FSA environment. The curriculum has been designed to provide Java programming fundamentals, best practices, design patterns, and specific information on Java developing at FSA.

**This track is designed to be self-guided & completed at your own pace. The sections below should be completed in the following order:**

1. [Introduction to Java](#)
2. [Intermediate Java](#)
3. [Advanced Java](#)

It is recommended to complete each module (ex. Programming Foundations: Design Patterns) within each section (ex. Intermediate Java) before beginning the next module/section. You will find hands-on exercises and additional resources directly below the training sections. It is recommended that you attempt the exercises and review the additional resources to further supplement your learning.

Self-guided learning will be augmented with the availability of dedicated mentors ([FSADevMentors@fsa.usda.gov](mailto:FSADevMentors@fsa.usda.gov)) to assure each participant can get the most out of their learning. If you have questions please ask!

### Environment / Access

**Before starting the training tracks you will need complete and/or review the following:**

#### 1. Software & Tools

- Install FSA Eclipse Software (FES) 6.0. ([click here for FES 6 install instructions](#)) This setup needs to be completed all the way down to (but not including) 'Run Reference Project'.
  - **NOTE:** Some courses may demonstrate concepts using IDE's that are not currently approved for use at FSA. Please continue to use the approved IDE's. (If you have questions on how to translate something demonstrated in a different IDE to the approved IDE, please contact the mentor group.)

#### 2. FSA System Access

- "Application/Software Development" system access requested on FSA-13-A (check "Application/Software Development" in Box 18)
  - **NOTE:** If you are already a developer, you probably already have this access. If so, you do not need to submit a new 13-A.

#### 3. Curriculum Access

- [Lynda.com](#)
  - Federal employees will have access to [Lynda.com](#) courses through AgLearn. If you cannot access [Lynda.com](#) content via AgLearn, please follow the instructions on the Workforce Re-investment home page to access Lynda.com for free via your public library.
- AgLearn
  - Links are provided to the search page within AgLearn, where you can search for the courses by their AgLearn ID.
  - **You must visit the content via AgLearn in order to earn credit toward the curriculum.** If you complete any courses via the direct public links from the wiki, you must still visit them in AgLearn to certify that you have completed them.

## Lab Hours

### Scheduling Lab Time (KC only)

The Mentor Team has reserved **Room G-14** for lab hours from **9:30am-11:30am**, and **1:30pm-3:30pm**, **Monday-Friday**. ([Sign up for lab hours](#))

If you need help, have questions, or just need to get away from your desk to focus on the training material, please take advantage of the lab hours.

**We ask that you please [sign up](#) in advance, as space is limited.**

## Curriculum Material

[Download schedule planning spreadsheet \(Java Track\)](#)

### Introduction to Java (approx. 10 hrs)

[Learning Java with Kathryn Hodge](#) (2h 11m)

This course uses Java 9 and the IntelliJ IDE. It is fine to use Java 8. It doesn't have JShell but you can skip that section. We've included Eclipse specific files to use for the exercises instead of the IntelliJ ones from the video.

[AgLearn: Lynda\\_10316](#) | [Public Web](#)

1. Understanding Java basics: data types, strings, arrays, and more
2. Controlling flow with functions and loops
3. Debugging
4. Working with inheritance and interfaces
5. Learning lambda

Exercise Files for Eclipse instead of IntelliJ

### Intermediate Java (approx. 10.5 hrs)

#### Design Patterns

**Foundations of Programming: Design Patterns** (2h 19m)

[AgLearn: Lynda\\_1284](#) | [Public Web](#)

most important at FSA are in **bold**

1. Strategy Pattern
2. Observer Pattern
3. Decorator Pattern
4. **Singleton Pattern**
5. State Pattern
6. Collection pattern
7. **Factories**

Keyhole Software has created Java implementations of several design patterns - on GitHub:

<https://github.com/in-the-keyhole/khs-gof-design-patterns>

There are many more: The definitive book is often called the [Gang of Four or GoF book](#)

[GoF Design Patterns "cheat sheet"](#)

### Advanced Java (approx. 17 hours)

#### Automated Testing Mistakes

[AgLearn: FSA-WFR-1301](#)

[Mistakes that make automated testing more difficult](#)

#### Advanced Java Programming

**Advanced Java Programming** (3h 33m)

[AgLearn: Lynda\\_1465](#) | [Public Web](#)

1. Getting Started
2. Java 7 New Features
3. Using Advanced Class Structures
4. Using the Reflection API
5. More of the Collections Framework
6. Testing and Advanced Exception Handling
7. Managing Files and Directories in Java 7
8. Working with I/O Streams
9. Working with Multi-Threading

#### Mastering Microservices

Directions for using the above Eclipse files

Want more? see the [Intro to Data Structures & Algorithms](#)

---

## SOLID

AgLearn: FSA-WFR-1101

[What are the S.O.L.I.D. Principles?](#)

[From STUPID to SOLID](#)

---

## Testing

Test Pyramid - [AgLearn](#): FSA-WFR-1102 | [Wiki](#)

**Testing with JUnit** (Peggy Fisher) (1h 10m)

AgLearn: Lynda\_10015 | [Public Web](#)

1. JUnit Basics
  2. Getting Started with JUnit
  3. JUnit Options
- 

## Spring and Struts

AgLearn: FSA-WFR-1103

[Spring Best Practices](#)  
(this is just a suggestion)  
Lynda has a great training on Spring (see the Advanced Java path on this page)

[Struts Best Practices](#)  
provided because you may be maintaining a project based on struts - no new projects will be started in struts

---

## Debugging

AgLearn: FSA-WFR-1104

[Debugging basics](#)

[Debugging Tutorial](#)

---

## Exceptions

But there are [Anti-Patterns](#) you should know about!

---

## JDBC

**Java: Database Integration with JDBC** (2h 51m)

AgLearn: Lynda\_1466 | [Public Web](#)

1. Managing Database Resources
  2. Reading Data
  3. Managing Data
  4. Using Metadata
- 

## Transactions

AgLearn: FSA-WFR-1201

[Transaction Management Best Practices](#)

---

## Web Services

**Building Web Services with Java EE** (~~2h 25m~~) (1h 30m)

AgLearn: Lynda\_3141 | [Public Web](#)

1. About Web Services (28m 39s)
  2. Representational State Transfer (REST) (19m 2s)
  3. Programming a Web Service in Java EE (20m 52s)
  4. Improving Your EE Code (21m 56s)
- 

## Multi-Module Maven

**Multi-Module Build Automation with Maven** (1h)

AgLearn: Lynda\_9131 | [Public Web](#)

**Mastering Microservices with Java** (2h 24m)

AgLearn: Lynda\_9172 | [Public Web](#)

1. Installation and Setup
  2. Writing Sample Tests
  3. Wrapping Controls and Pages
  4. Complex Cases for Pages and Elements
- 

## Spring Framework in Depth

**Spring Framework In Depth** (2h 16m)

AgLearn: Lynda\_10307 | [Public Web](#)

1. Spring Overview
  2. Configuring the Application Context
  3. Annotation-Based Configuration
  4. XML-Based Configuration
  5. Bean Lifecycle
  6. Aspect Oriented Programming
- 

## AWS Security

**Cybersecurity Awareness: Breaking Down Cloud Security** (1h 48m)

AgLearn: Lynda\_10313 | [Public Web](#)

1. Brief Overview of Cloud Computing
  2. Cloud Security Considerations
  3. Security Best Practices for Clouds
  4. Other Cloud Security Considerations
- 

## AWS for DevOps

**AWS for DevOps: Monitoring, Metrics, and Logging** (2h 41m)

AgLearn: Lynda\_10308 | [Public Web](#)

AgLearn: FSA-WFR-1105

### Exceptions

---

### **FSA Builds, Deploys, CI /CD**

AgLearn: FSA-WFR-1106

### Intro to FSA Builds

### FSA Build Stack 1 Landing Page

### FSA Build Stack 2 Landing Page

### Continuous Integration

---

### **FSA Shared Services**

AgLearn: FSA-WFR-1107

### Shared Services

---

### **Refactoring**

If you question this in an "Intro to Java" setting, remember that refactoring is a great way to learn about the code... completely non-destructive since you do not HAVE to check it in if the refactoring goes awry... refactoring is a critical skill many don't pick up until later in their Java careers, but it is important to know how to do it well!

### **Programming Foundations: Refactoring Code (1h 44m)**

AgLearn: Lynda\_1289 | [Public Web](#)

1. Introduction to Refactoring
2. Getting Started: Method-Level Refactoring
3. Class- and Condition-Focused Refactoring
4. Data-Focused Refactoring
5. Communication and High-Level Refactoring

1. Create a Multi-Module Project
  2. Maven and IDEs - Eclipse
  3. Other Maven Features
- 

### **Generics**

### **Java: Generic Classes (1h 5m)**

AgLearn: Lynda\_10304 | [Public Web](#)

1. What Are Generics in Java
  2. Generics in Java
- 

### **Java Data Structures**

### **Java: Data Structures (5 8m 58s)**

AgLearn: Lynda\_10305 | [Public Web](#)

1. What Are Data Structures
2. Using Data Structures

Want more? see the [Intro to Data Structures & Algorithms](#)

### **Recursion**

### **Java: Recursion (55m 27s)**

AgLearn: Lynda\_10306 | [Public Web](#)

1. Recursion
2. Write recursive Functions

\*Project assigned by mentor team\*

1. Monitoring Approaches
  2. Monitoring Core AWS Services
  3. Core AWS Tools for Metrics and Logging
  4. Using AWS Services for Advanced Monitoring
  5. Advanced and Third-Party Tools
  6. Application Monitoring
- 

### **AWS Monitoring and Metrics**

### **Amazon Web Services: Monitoring and Metrics (2h 4m)**

AgLearn: Lynda\_10312 | [Public Web](#)

1. Exploring Monitoring Tools
  2. Using AWS Config
  3. Beyond Traditional Monitoring
  4. Taking Action
- 

### **Build and Deploy Spring Boot Microservices**

### **JHipster: Build and Deploy Spring Boot Microservices (2h 59m)**

AgLearn: Lynda\_10309 | [Public Web](#)

1. Setting Up Your Environment
2. Microservice Architecture
3. JHipster
4. JHipster API Gateway
5. Create a Microservice
6. Set Up the JHipster Registry
7. Implement REST Endpoint
8. Manage Multiple Microservices
9. Deployment with Docker
10. Deployment Cloud Solutions





\*Project assigned by mentor team\*

















































## Supplemental Material
























### Hands-On Exercises

#### Java Katas - practical exercises to improve your coding

▼ [Click here to expand...](#)

The code here is  beginner  intermediate  challenging  advanced... you may solve the problems differently depending on the level of experience you have... that is perfectly fine!

1. [Kata01: Supermarket Pricing](#) - no coding required - this is a thinker. Explore the possibilities     

2. [Kata02: Karate Chop](#) - Implement a binary search routine - there are several possibilities here - iterative, recursion, functional... see how many ways you can solve this.    
3. [Kata03: How Big? How Fast?](#) - tests your rough estimation abilities. The best way to improve estimation is to exercise it - do it often.    
4. [Kata04: Data Munging](#) - how easy is it to change or improve the code you have written? kata asks you to write code... then come back and "improve" it later... educate on design choices and ways to solve problems   
 
5. [Kata05: Bloom Filters](#) - Implement a Bloom filter based spell checker...uses bitmaps, hash functions... algorithm details are in the kata   
6. [Kata06: Anagrams](#) - find in a dictionary all the anagrams (words that use the exact same letters in different orders) for a particular word list   
7. [Kata07: How'd I Do?](#) - this is for intermediate or advanced coders - look back on code you have "forgotten" and review it with fresh eyes   
8. [Kata08: Conflicting Objectives](#) - you will write a program 3 times with a different goal each time...    
 
9. [Kata09: Back to the Checkout](#) - back to supermarket pricing - this time do implementation... the objective is to realize decoupling is critical and why...   
10. [Kata10: Hashes vs. Classes](#) - thought experiment, no coding... this focuses thoughts on design and flexibility    
11. [Kata11: Sorting It Out](#) - explore sorting with a contrived "lottery" balls puzzle     
  
12. [Kata12: Best Sellers](#) - no code necessary, although it might help to arrive at conclusions... 
13. [Kata13: Counting Code Lines](#) - seemingly simple...count the lines of code... but ignore blank lines and all comments, whether single line or multi-line   
14. [Kata14: Tom Swift Under the Milkwood](#) - generate sentences from "trigrams"   

15. [Kata15: A Diversion](#) - mathematical puzzle using binary numbers    
16. [Kata16: Business Rules](#) - how to design when business rules are complicated, and fairly arbitrary... what can be done to simplify them?   
17. [Kata17: More Business Rules](#) - complicated rules, how can we make it easy to modify after we go to production?   
18. [Kata18: Transitive Dependencies](#) - using dependencies to drive out coupling...   
19. [Kata19: Word Chains](#) - discover the chains between one word and another... change out one letter in each transition until you have all the words between the two    
20. [Kata20: Klondike](#) - use the card game to learn design and how to solve the problem   
21. [Kata21: Simple Lists](#) - write three implementations of the list: 1) a singly linked list (each node has a reference to the next node) 2) a doubly linked list (each node has a reference to both the next and previous nodes) 3) some other implementation of your choosing, except that it should use no references (pointers) to collect nodes together in the list (Obviously, we won't be using predefined library classes as our list implementations...)  
- 

## Additional Resources - Tips and Tricks

### Points of emphasis in FSA environment

- Transaction management - [Transaction Management Best Practices](#)
- 508 Compliance:
  - [Section 508 Automated Web Site and Application Testing](#)
  - [How-to Use the Section 508 Automated Testing Tools](#)
  - [Section 508 Testing](#)
- Caching strategies - info coming soon
- Shared Services / EJB - [Shared Services](#)
- External communication isolation - info coming soon
- Type safety, enums vs. Enum patterns - [Type safety, enums vs. Enum patterns](#)
- Business validation strategies - [Data validation](#)

### Control-M Developer information

[Control-M Landing Page](#)

[Control-M Developer Worksheet Instructions](#)

### Deeper Dives... extras that will help your understanding...

Introduction to Data Structures & Algorithms in Java (4h 56m)

AgLearn: Lynda\_10306 | [Public Web](#)

1. Introduction
2. Analysis of Algorithms
3. Basic Sorting and Search Algorithms
4. Linked Lists
5. Stacks and Queues
6. Recursion
7. Binary Search Trees
8. More Sorting Algorithms
9. Heaps
10. Hashtables

Java Design Patterns Reference and Examples

## ▼ GoF Creational Patterns...

### GoF Creational Patterns

---

#### Abstract Factory

Sets of methods to make various objects.

Abstract Factory is recognizable by the creational methods returning the factory itself which in turn can be used to create another abstract or interface type

Examples:

```
javax.xml.parsers.DocumentBuilderFactory#newInstance()  
javax.xml.transform.TransformerFactory#newInstance()  
javax.xml.xpath.XPathFactory#newInstance()
```

---

#### Builder

Make and return one object various ways.

Builder is recognizable by creational methods returning the instance itself

Examples:

```
java.lang.StringBuilder#append() (unsynchronized)  
java.lang.StringBuffer#append() (synchronized)  
java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)  
javax.swing.GroupLayout.Group#addComponent()  
All implementations of java.lang.Appendable
```

---

#### Factory Method

Methods to make and return components of one object various ways.

Factory Method is recognizable by creational methods returning an implementation of an abstract/interface type

Examples:

```
java.util.Calendar#getInstance()  
java.util.ResourceBundle#getBundle()  
java.text.NumberFormat#getInstance()  
java.nio.charset.Charset#forName()  
java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object per protocol)  
java.util.EnumSet#of()  
javax.xml.bind.JAXBContext#createMarshaller() and other similar methods
```

---

#### Prototype

Make new objects by cloning the objects which you set as prototypes.

Prototype is recognizable by creational methods returning a different instance of itself with the same properties

Example:

```
java.lang.Object#clone() (the class has to implement java.lang.Cloneable)
```

---

#### Singleton

A class distributes the only instance of itself.

Singleton is recognizable by creational methods returning the same instance (usually of itself) everytime

Examples:

```
java.lang.Runtime#getRuntime()  
java.awt.Desktop#getDesktop()  
java.lang.System#getSecurityManager()
```

## ▼ GoF Structural Patterns...

### GoF Structural Patterns

---

#### Adapter



A class extends another class, takes in an object, and makes the taken object behave like the extended class. Adapter is recognizable by creational methods taking an instance of different abstract/interface type and returning an implementation of own/another abstract/interface type which decorates/overrides the given instance)

Examples:

```
java.util.Arrays#asList()
java.util.Collections#list()
java.util.Collections#enumeration()
java.io.InputStreamReader(InputStream) (returns a Reader)
java.io.OutputStreamWriter(OutputStream) (returns a Writer)
javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()
```

---

## Bridge

An abstraction and implementation are in different class hierarchies.

Bridge is recognizable by creational methods taking an instance of different abstract/interface type and returning an implementation of own abstract/interface type which delegates/uses the given instance

Examples:

None comes to mind yet. A fictive example would be `new LinkedHashMap(LinkedHashSet<K>, List<V>)` which returns an unmodifiable linked map which doesn't clone the items, but uses them. The `java.util.Collections#newSetFromMap()` and `singletonXXX()` methods however comes close.

---

## Composite

Assemble groups of objects with the same signature.

Composite is recognizable by behavioral methods taking an instance of same abstract/interface type into a tree structure

Examples:

```
java.awt.Container#add(Component) (practically all over Swing thus)
javax.faces.component.UIComponent#getChildren() (practically all over JSF UI thus)
```

---

## Decorator

One class takes in another class, both of which extend the same abstract class, and adds functionality.

Decorator is recognizable by creational methods taking an instance of same abstract/interface type which adds additional behaviour

Examples:

All subclasses of [java.io](#).InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.

`java.util.Collections`, the `checkedXXX()`, `synchronizedXXX()` and `unmodifiableXXX()` methods.

`javax.servlet.http.HttpServletRequestWrapper` and `HttpServletResponseWrapper`

---

## Facade

One class has a method that performs a complex process calling several other classes.

Facade is recognizable by behavioral methods which internally uses instances of different independent abstract/interface types

Examples:

`javax.faces.context.FacesContext`, it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection).

`javax.faces.context.ExternalContext`, which internally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

---

## Flyweight

The reusable and variable parts of a class are broken into two classes to save resources.

Flyweight is recognizable by creational methods returning a cached instance, a bit the "multiton" idea

Example:

```
java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short, Long and BigDecimal)
```

---

## Proxy

One class controls the creation of and access to objects in another class.

Proxy is recognizable by creational methods which returns an implementation of given abstract/interface type which in turn delegates/uses a different implementation of given abstract/interface type

Examples:

`java.lang.reflect.Proxy`

`java.rmi.*`

`javax.ejb.EJB` (explanation here)

`javax.inject.Inject` (explanation here)

`javax.persistence.PersistenceContext`

## ▼ GoF Behavioral Patterns...

### GoF Behavioral Patterns

#### Chain of Responsibility

A method called in one class can move up a hierarchy to find an object that can properly execute the method.

Chain Of Responsibility is recognizable by behavioral methods which (indirectly) invokes the same method in another implementation of same abstract/interface type in a queue

Examples:

`java.util.logging.Logger#log()`

`javax.servlet.Filter#doFilter()`

---

#### Command

An object encapsulates everything needed to execute a method in another object.

Command is recognizable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been encapsulated by the command implementation during its creation

Examples:

All implementations of `java.lang.Runnable`

All implementations of `javax.swing.Action`

---

#### Interpreter

Define a macro language and syntax, parsing input into objects which perform the correct operations.

Interpreter is recognizable by behavioral methods returning a structurally different instance/type of the given instance /type; note that parsing/formatting is not part of the pattern, determining the pattern and how to apply it is

Examples:

`java.util.Pattern`

`java.text.Normalizer`

All subclasses of `java.text.Format`

All subclasses of `javax.el.ELResolver`

---

#### Iterator

One object can traverse the elements of another object.

Iterator is recognizable by behavioral methods sequentially returning instances of a different type from a queue

Examples:

All implementations of `java.util.Iterator` (thus among others also `java.util.Scanner!`).

All implementations of `java.util.Enumeration`

---

#### Mediator

An object distributes communication between two or more objects.

Mediator is recognizable by behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance

Examples:

`java.util.Timer` (all `scheduleXXX()` methods)

`java.util.concurrent.Executor#execute()`

`java.util.concurrent.ExecutorService` (the `invokeXXX()` and `submit()` methods)

`java.util.concurrent.ScheduledExecutorService` (all `scheduleXXX()` methods)

`java.lang.reflect.Method#invoke()`

---

## Memento

One object stores another objects state.

Memento is recognizable by behavioral methods which internally changes the state of the whole instance

Examples:

java.util.Date (the setter methods do that, Date is internally represented by a long value)

All implementations of [java.io.Serializable](#)

All implementations of javax.faces.component.StateHolder

---

## Observer

An object notifies other object(s) if it changes.

Observer (or Publish/Subscribe) is recognizable by behavioral methods which invokes a method on an instance of another abstract/interface type, depending on own state

Examples:

java.util.Observer/java.util.Observable (rarely used in real world though)

All implementations of java.util.EventListener (practically all over Swing thus)

javax.servlet.http.HttpSessionBindingListener

javax.servlet.http.HttpSessionAttributeListener

javax.faces.event.PhaseListener

---

## State

An object appears to change its` class when the class it passes calls through to switches itself for a related class.

State is recognizable by behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally

Example:

javax.faces.lifecycle.Lifecycle#execute() (controlled by FacesServlet, the behaviour is dependent on current phase (state) of JSF lifecycle)

---

## Strategy

An object controls which of a family of methods is called. Each method is in its` own class that extends a common base class.

Strategy is recognizable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been passed-in as method argument into the strategy implementation

Examples:

java.util.Comparator#compare(), executed by among others Collections#sort().

javax.servlet.http.HttpServlet, the service() and all doXXX() methods take HttpServletRequest and HttpServletResponse and the implementor has to process them (and not to get hold of them as instance variables!).

javax.servlet.Filter#doFilter()

---

## Template

An abstract class defines various methods, and has one non-overridden method which calls the various methods.

Template is recognizable by behavioral methods which already have a "default" behaviour defined by an abstract type

Examples:

All non-abstract methods of [java.io.InputStream](#), [java.io.OutputStream](#), [java.io.Reader](#) and [java.io.Writer](#).

All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

javax.servlet.http.HttpServlet, all the doXXX() methods by default sends a HTTP 405 "Method Not Allowed" error to the response. You're free to implement none or any of them.

---

## Visitor

One or more related classes have the same method, which calls a method specific for themselves in another class.

Visitor is recognizable by two different abstract/interface types which has methods defined which takes each the other abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it

Examples:

javax.lang.model.element.AnnotationValue and AnnotationValueVisitor

javax.lang.model.element.Element and ElementVisitor

javax.lang.model.type.TypeMirror and TypeVisitor

java.nio.file.FileVisitor and SimpleFileVisitor

javax.faces.component.visit.VisitContext and VisitCallback

