LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Control Flow

COURSE AGENDA

- What is an Object
- The Basics
- Operators
- Control Flow
- Implementation Hiding
- Reuse
- Interfaces
- Collections
- Functional Programming
- Stream
- Exceptions
- Generics
- Arrays

INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
  - DTCC
  - Penske
  - HSBC
  - Government Agencies

https://www.linkedin.com/in/hugoscavino/

TOPIC DESCRIPTION

Access control in Java defines the visibility and accessibility of classes, methods, variables, and constructors across different parts of an application. Java provides access modifiers to ensure encapsulation, maintain data integrity, and enforce boundaries on components' interactions.

- Discuss Packaging
- Discuss access control levels – public, protected, private & [default]
- Key Concepts – Housekeeping with constructors

REQUIRED SOFTWARE

- JDK™
  - JDK 8 or Newer (Open Source, Amazon, or Oracle)
  - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.

- IDE
  - IntelliJ Community Edition (Free) or Enterprise (Paid)
  - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind

RECOMMENDED TEXTS

- Java 5 Book
  - Thinking in Java – 4th Edition - Free – PDF - GitHub
  - Thinking in Java – 4th Edition – Hard Cover - Amazon

- Recommended Java 8 Book
  - Bruce Eckel on Java 8
    - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
  - Head First Java: A Brain-Friendly Guide 3rd Edition

ONLINE VIDEO TUTORIAL

- Learning Programming In Java
  - YouTube Tutorials
- Lab Exercises
  - http://www.learnprogramminginjava.com/

PACKAGES

- Packaging logically groups related classes, interfaces, and sub-packages into packages.
- Packages serve as namespaces to avoid naming conflicts and help organize code, making it easier to maintain and navigate.
- A package can contain multiple classes and interfaces and be part of a larger hierarchical structure through sub-packages.

WHY USE PACKAGES?

- **Avoid Name Conflicts:** Two classes can have the same name if they belong to different packages.
- **Code Organization:** Related classes and interfaces are grouped logically.
- **Encapsulation and Access Control:** Classes in a package can have package-private visibility, restricting access from other packages.
- **Reusability:** Frequently used classes can be grouped into reusable libraries or APIs.
- **Modular Development:** Packages promote a modular architecture, improving code management and maintenance.

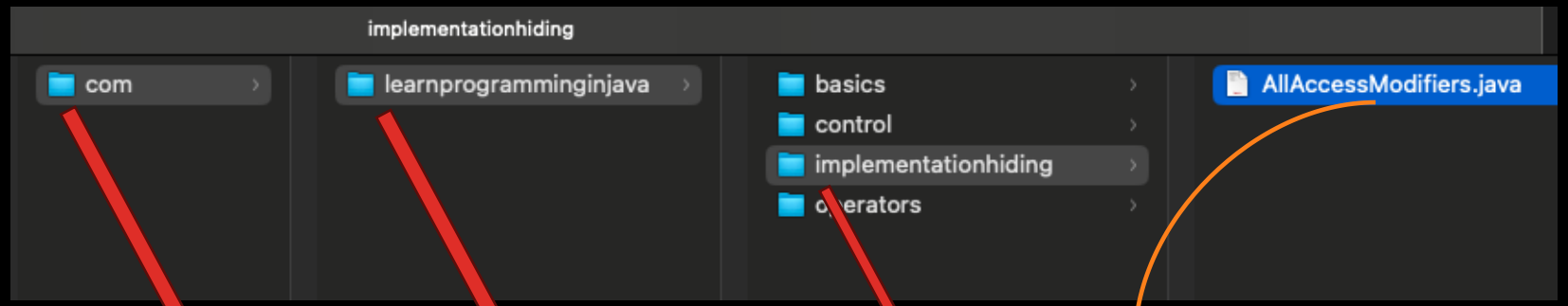
PACKAGES AND DIRECTORY STRUCTURE

- **User Defined:** These are the packages you create.

```
package com.learnprograminginjava.implementationhiding;


public class AllAccessModifiers {

```

- By convention, we reverse the user's domain name and use all lowercase letters.
- The `.` are translated to your OS's directory separator. "\", "/", etc

PACKAGES AND DIRECTORY STRUCTURE




```
package com.learnprogramminginjava.implementationhiding;

public class AllAccessModifiers {
```

TYPES OF PACKAGES

- **Built-In:** Provided by Java, for example, `java.lang, java.util`, etc.
- **User Defined:** These are the packages you create.
- **Declaring a package:** Use the `package` keyword as the FIRST line of the Java file.

```
package com.learnprograminginjava.implementationhiding;


public class AllAccessModifiers {
```

IMPORTING PACKAGES

- **Specific Class:** At the top of the file after the package

```
import java.util.ArrayList;
```

- **Wildcard (all classes in the package):**

```
import java.util.*;  // Imports all classes from java.util
```

- **Without an import:** Use the entire `package` name. This can become tedious.

```
java.util.ArrayList<String> list = new java.util.ArrayList<>();
```

JAVA SOLUTIONS

Four types of access control levels

Modifier	Class	Package	Subclass	Clients Other Classes
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
(default)*	✓	✓	✗	✗
private	✓	✗	✗	✗

**There is no default keyword, default is the access you get when you do not annotate the method or class*

PACKAGE ACCESS

- When a class, method, variable, or constructor is declared without any explicit access modifier (like public, private, or protected), the compiler gives it a default package access (also known as package-private access).
- The member or class is accessible only within the same package and is not visible outside the package.

Key Characteristics of Default Package Access

- Scope: Accessible to all classes within the same package.
- Inaccessible Outside the Package: Even if they import the package, other packages cannot access members with default access.
- Useful for Internal Components: Default access is typically used to limit the visibility of classes and members only meant to be used within the package.

PACKAGE ACCESS EXAMPLE

```java
package com.example;

class Person {  // No access modifier -> Default (package-private) access
    String name;  // Default access variable
    int age;      // Default access variable

    void displayInfo() {  // Default access method
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

- The class has no access modifier. Therefore, the class is package-private (aka default
- The same can be said for the method `displayInfo()`

PUBLIC PACKAGE ACCESS

- The `public` access modifier allows maximum visibility. A class, method, constructor, or variable declared with the public modifier is accessible from anywhere—both within and outside the package. They are typically used for components that need to be accessed openly, such as APIs or entry-point classes like public static void main().

Key Characteristics of the `public` Access Modifier

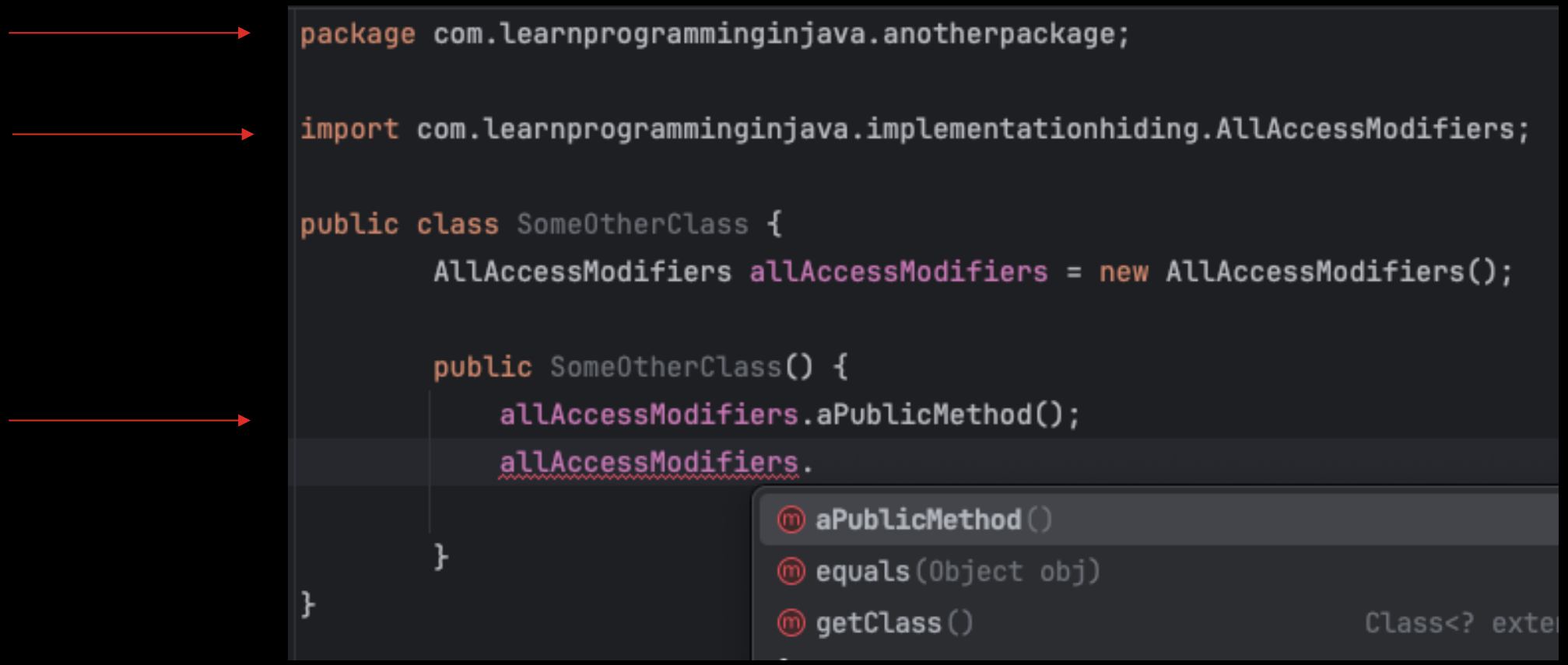
- Global Accessibility: Public members are accessible from any class, package, or module.
- Used for APIs and Core Components: Methods and classes intended to be reusable across different packages or modules are often declared public.
- Top-level Classes Can Be public: A top-level class (not nested within another class) can be public. This allows it to be referenced and used from other packages.
- Consistency Across Packages: Public access is appropriate if you need a method or class to be accessed universally (e.g., a utility class).

PUBLIC PACKAGE ACCESS EXAMPLE

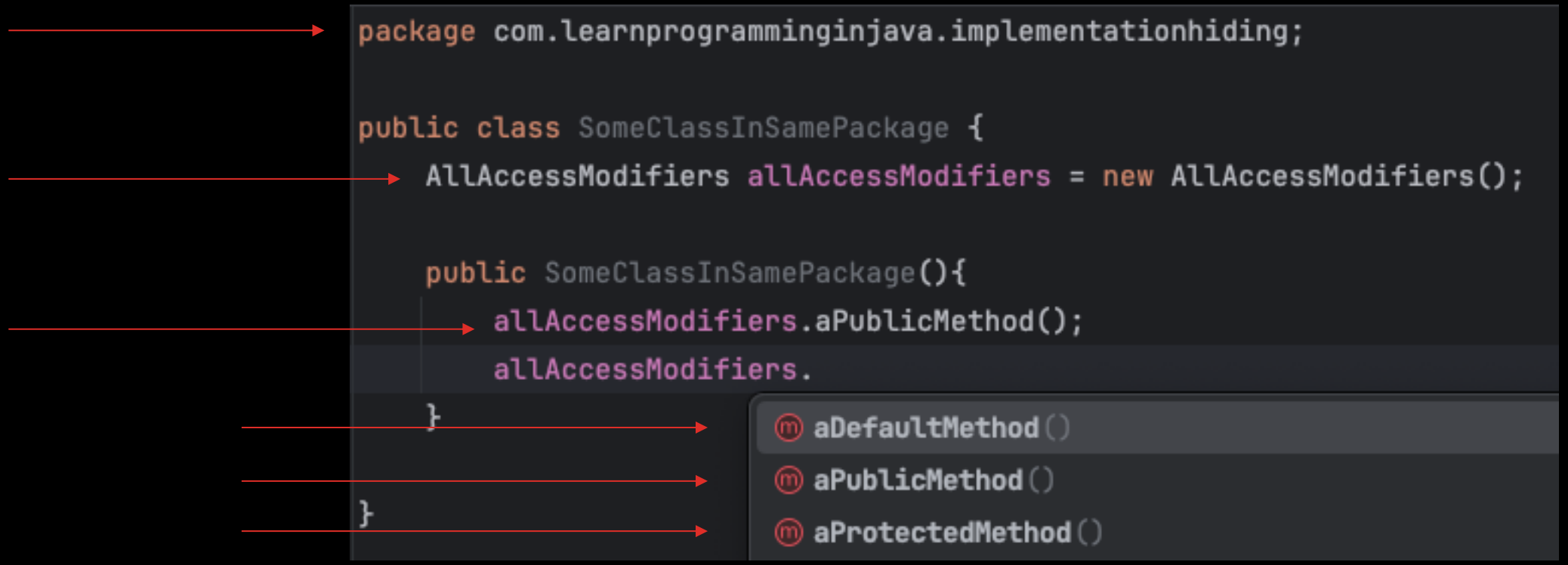
```java
package com.example;

public class Utility {  // Public class: accessible from any package
    public static void printMessage(String message) {  // Public method
        System.out.println(message);
    }
}
```

- The class is public because of the explicit use of the keyword `public`
- The `static` method is also `public`, because of the explicit use of the keyword `public`. `public` is NOT the default access modifier. The default is package.

VISIBILITY FROM SOME OTHER CLASS



VISIBILITY FROM CLASS IN SAME PACKAGE



PROTECTED ACCESS CONTROL

The protected access modifier allows a class member (method, field, or constructor) to be accessible:

1. Within the same package, and
2. In subclasses (child classes), even in different packages.

This access level balances encapsulation and inheritance, enabling some visibility while still limiting access to outside code.

Key Characteristics of `protected` Access Control

- Subclass access: Subclasses (in the same or different package) can access protected members through inheritance.
- Not globally accessible: Classes outside the package that don't extend the class cannot access protected members.

PROTECTED ACCESS CONTROL - EXAMPLE

```java
package com.example;

public class Animal {
    protected String type = "Mammal";  // Protected field

    protected void displayType() {  // Protected method
        System.out.println("Animal type: " + type);
    }
}
```

```java
package com.example;

public class Dog extends Animal {  // Same package

    public void showDetails() {
        displayType();  // Accessible because Dog is in the same package
    }
}
```

The Dog class accesses the Animal class and its `displayType()` method because it is a subclass of Animal

CONSTRUCTORS

A constructor is a unique method used to initialize an object when it is created. A constructor has the same name as the class and no return type. It is automatically called when an instance of a class is created, ensuring that the object is correctly initialized.

- Same name as the class: The constructor must have the same name as the class in which it is defined.
- No return type: A constructor cannot return any value, including void.
- Called automatically: It is invoked automatically when a new class object is instantiated.
- Overloading allowed: A class can have multiple constructors with different parameter lists (constructor overloading).
- Two types of constructors:
  - Default constructor: Provided by the compiler if no constructors are defined.
  - Parameterized constructor: Allows passing parameters during object creation.

DEFAULT CONSTRUCTORS

- Will be generated if not declared in the class
- The compiler will initialize the fields with the default values for classes, primitives, and arrays.
- Best practice is NOT to allow the compiler to define and set defaults for your fields.
  - You should take charge

DEFAULT CONSTRUCTORS

- This class has no declared constructor ; therefore, you get one like this

```
public class DefaultConstructor {
    private String name;
    private int age;

    // public DefaultConstructor(){
    //     System.out.println("name " + name + " age " + age);
    // }

    public static void main(String[] args) {
        DefaultConstructor dc = new DefaultConstructor();
        System.out.println(dc.name);
        System.out.println(dc.age);
    }
}
```

- The default value for a class is null
- The default value for an int is zero
- What will be printed out?
- Why can the class print out private fields?

DEFAULT CONSTRUCTOR WITH DEFAULTS

```java
public class DefaultConstructorWithDefaults {
    private final String name;
    private final int age;

    public DefaultConstructorWithDefaults() {
        name = "John Doe";
        age = 18;
    }
    public static void main(String[] args) {
        DefaultConstructorWithDefaults dc = new DefaultConstructorWithDefaults();
        System.out.println(dc.name);
        System.out.println(dc.age);
    }

}
```

- Setting the field values to non-default values
- Why can we use `final` when we declare the field

PARAMETERIZED CONSTRUCTORS

```java
public class DefaultConstructorWithParameters {
    private final String name;
    private final int age;

    /**
     * If you define a constructor, the default constructor generated
     * by the compiler is no longer generated.
     * @param name Name
     * @param age Age
     */
    public DefaultConstructorWithParameters(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

- The fields must be provided for the class to be created
- What happened to the “free” default constructor?
- Can I create the object without providing the fields?

PARAMETERIZED CONSTRUCTORS

```java
public boolean isAdult(){
    return age >= 18;
}

public static void main(String[] args) {
    // DefaultConstructorWithParameters defNoParams = new DefaultConstructorWithParameters();

    DefaultConstructorWithParameters def = new DefaultConstructorWithParameters( name: "Jane Doe",  age: 21);
    System.out.println(def.getName());
    System.out.println(def.getAge());
    System.out.println(def.isAdult());
}
```

- The “no-arg” constructor is gone!
- The class requires fields for a proper instance to be created

ARRAY INITIALIZATION

```
public class ArrayInitializing {  new *

    private static final String[] lastNames = {"Doe", "Smith", "Lee", "Mohamed", "Perez"};  5 usages
    private static final String[] userNames = new String[lastNames.length];  2 usages
```

- Sequence of objects or primitives under one same
- Uses the well known `[ ]` square bracket syntax (ala C and C++)
- You can set the size if you know the final length

ENUMS

- **enumeration** (or more commonalty an **enum**) is a special data type that enables a variable to be a set of predefined constants.
  - a type-safe way of defining a collection of constants under a single type name.
- Enums are particularly useful for representing data that does not change, such as days of the week, months, directions, etc.

Key Features of Enums

- **Type-Safety**: Enums provide type safety, meaning only predefined constants can be used, preventing the accidental assignment of invalid values.
- **Defined Values**: An enum specifies a list of constant values that are the only permissible values for variables of that enum type.

ENUM EXAMPLE

```
public enum WeekDay {  8 usages  new *
    Sun,  no usages
    Mon,  no usages
    Tue,  no usages
    Wed,  no usages
    Thu,  no usages
    Fri,  no usages
    Sat  no usages
}
```

- **Enum** has a .values() method which returns an array of the enum
- **Defined Values**: If the list of enums types changes the for-loop will not change.

```
static public void processAllWeekDay(){  1 usage  new *
    for (WeekDay weekDay : WeekDay.values()){ // .values() in the for loop
        System.out.print(weekDay);
        System.out.print(" : ");
        System.out.println(weekDay.ordinal()); // what is the ordinal in the declared enum
    }
}
```

```
static public WeekDay getWeekDay(String day){
    return WeekDay.valueOf(day);
}
```

ENUM EXAMPLE

```
public enum WeekDay {  8 usages  new *
    Sun,  no usages
    Mon,  no usages
    Tue,  no usages
    Wed,  no usages
    Thu,  no usages
    Fri,  no usages
    Sat  no usages
}
```

- **Type-Safety**: The method .processWeekDay(WeekDay weekday):void only takes an enum type. Not a problematic String or int. The WeekDay is a full-blown type! Compare that method to .getWeekDay() which is case sensitive and literal.

```
static public void processWeekDay(WeekDay weekDay){  1 usage  new *
    System.out.println(weekDay);
}
```

```
static public WeekDay getWeekDay(String day){
    return WeekDay.valueOf(day);
}
```

TOPIC SUMMARY

Access control in Java defines the visibility and accessibility of classes, methods, variables, and constructors across different parts of an application. Java provides access modifiers to ensure encapsulation, maintain data integrity, and enforce boundaries on components' interactions.

- Packaging
  - Declaring and importing
- access control levels
  - public, protected, private & [default]
- Housekeeping: Constructors
  - Default and parameterized
  - Arrays
  - Enums

ONLINE VIDEOS

- Learning Programming In Java
  - YouTube Tutorials
- Lab Exercises
  - http://www.learnprogramminginjava.com/