# CMSC 132: OBJECT-ORIENTED PROGRAMMING II

## Java Language Constructs I

Department of Computer Science
University of Maryland, College Park

## Enhanced Switch in Java

- **What is the Enhanced Switch?**
  - Introduced in Java 12 (preview) and standardized in Java 14.
  - Offers a concise, expressive, and safer way to use the switch statement.
  - Addresses common pitfalls of traditional switch, such as unintended fall-through.
- **Key Features**
- **Arrow Syntax (->)**:
  - Simplifies syntax and eliminates the need for break.
  - Default behavior prevents fall-through.
- **Multiple Labels per Case**:
  - Group related cases together for clarity and efficiency.
- **Expression Support**:
  - Allows switch to produce and return values directly.
- **Multi-Statement Cases**:
  - Use curly braces {} to include multiple actions within a single case.

  See: **`enhancedSwitch package`**

## Varargs (Variable Arguments) in Java

- **What is Varargs?**
  - **Varargs** allows a method to accept **zero or more arguments** of the same type.
  - Introduced in **Java 5** to make method calls more flexible and concise.
  - Declared by using an ellipsis (...) followed by the type, at the end of the parameter list.
- **Key Points:**
  - The parameter must be the **last parameter** in the method signature.
  - The **varargs parameter** is treated as an array inside the method.
  - Varargs is useful when you don't know the exact number of arguments a method will take, such as in utility methods or when dealing with variable-length lists of parameters.
- **Benefits:**
  - Simplifies method declarations when dealing with an unknown number of parameters.
  - Makes method calls cleaner and more readable without needing to pass an array.
  - Reduces the need for method overloading in cases of variable numbers of arguments.
- **Common Use Cases:**
  - Methods like System.out.printf() that accept a variable number of arguments for formatting.
  - Utility methods for joining or summing up values of any number of elements.
- **Considerations:**
  - Varargs parameters are **internally represented as an array**, so performance may be affected when dealing with a large number of arguments.
  - Overloading methods with varargs should be done carefully to avoid ambiguity with other parameter types.
  - See: **`varargsExample package`**

## Introduction to Java Enums

- **Definition:**Enums (short for Enumerations) are special data types in Java used to define collections of constants.
- **Purpose:**
  - Improve code readability.
  - Group related constants.
  - Prevent invalid values.
- **Key Characteristics:**
  - Enums are implicitly final and cannot be instantiated.
  - Inherit from java.lang.Enum.
    https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Enum.html
  - Provide type safety compared to traditional constants.
- **Common Use Cases:**
  - Representing states (e.g., Days of the Week, Directions).
  - Defining configuration options.

## Characteristics and Behavior of Enums

- **Static Constants:**
  Each value in an enum is implicitly public static final.
- **Singleton Nature:**
  Enum constants are unique instances.
- **Built-In Methods:**
  - values(): Returns an array of all enum constants.
  - ordinal(): Returns the position of the constant in the enum.
  - name(): Returns the name of the constant.
- **Interfaces & Methods:**
  - Enums can implement interfaces.
  - Can include fields, constructors, and methods.
- **Immutability:**
  Enum constants are immutable and thread-safe.

## Advantages of Using Enums

- **Improved Type Safety:**
  - Prevents assigning invalid values to variables.
- **Enhanced Readability:**
  - Easy-to-read and meaningful constants.
- **Encapsulation:**
  - Can group logic with related constants.
- **Seamless Integration:**
  - Works with Java Collections, switch statements, etc.
- **Memory Efficiency:**
  - Instances are created once and reused.
- **Built-in Methods & Overrides:**
  - Override toString(), define custom behaviors, and compare enum constants.

## Enum Best Practices

- **Clear Naming Conventions:**
  - Use all caps with underscores for constants (e.g., HIGH_PRIORITY).
- **Limit Enum Scope:**
  - Avoid bloating enums with unrelated methods or data.
- **Custom Methods:**
  - Add methods to extend functionality only when necessary.
- **Avoid Overuse:**
  - Use enums only for finite, fixed sets of constants.
- **Use in Switch Statements:**
  - Simplify control flow with enums.
  - See: `enumExamples package`

## Introduction to Java Annotations

- **What Are Annotations in Java?**
  - **Annotations** are a form of metadata that provide additional information about the program to the compiler or runtime environment.
  - Annotations do not alter the behavior of the program, but they can be used by tools and libraries to perform operations such as validation, documentation, or code generation.
  - Introduced in **Java 5**.
- **Why Use Annotations?**
  - Provide **metadata** to influence how code is processed.
  - Help with **code readability** and **maintainability**.
  - Enable **frameworks** to perform tasks like dependency injection, validation, etc.

## Types of Annotations

- **Built-in Annotations**
  - **@Override**: Indicates that a method is overriding a method from its superclass.
  - **@Deprecated**: Marks a method or class as outdated and suggests that developers avoid using it.
  - **@SuppressWarnings**: Tells the compiler to suppress specific warnings.
  - **@FunctionalInterface**: Marks an interface as functional (only one abstract method).
- **Custom Annotations**
  - Java allows the creation of custom annotations with specific metadata.
- **Annotation Target**
  - Annotations can be applied to classes, methods, fields, parameters, constructors, etc.
- Reference
  - http://docs.oracle.com/javase/tutorial/java/annotations/basics.html

  See: `annotation package`