Java User Group Philippines

Java User Group Philippines

Java User Group Philippines is a group for all Java advocates and enthusiasts. Members can talk about Java, JVM, and the latest frameworks within the Java ecosystem. Everyone is welcome to join their events whether one is a developer from a different tech stack, architect, or student.

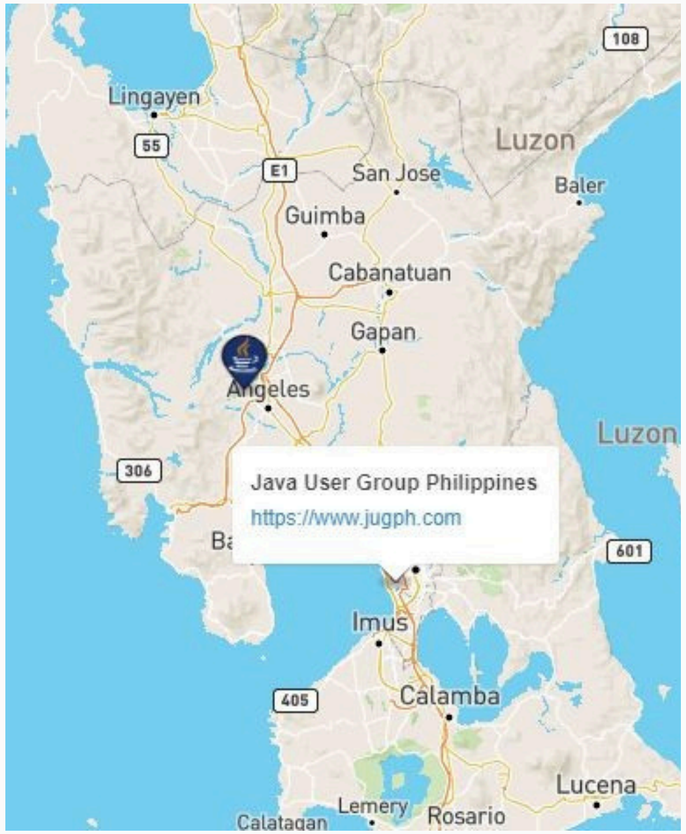
Revitalized JUG since April 2023.

2100+ Facebook Members
650+ Meetup.com Members

Recognized by Oracle as the only **ACTIVE** JUG* in the Philippines.

Member of the Java Community Process, a collaboration between the Java community and Oracle to develop and evolve the Java Platform.

*Java User Groups (JUGs) are volunteer organizations that strive to distribute the Java-related knowledg globally.



Java User Group

25th ANNIVERSARY 1998 2023 Java Community Process

foojay.io

JCP Java in Education Initiative

The JCP in Education initiative leverages the Java Community Process (JCP) program to promote Java technology in educational environments globally. Led by Java User Groups (JUGs) and supported by the JCP program, it aims to inspire junior developers and students to learn Java.

Key Activities:

- **Collaboration with Academics:** JUGs connect with student associations and academic staff to offer sessions led by industry professionals.

- **Curriculum Integration:** Promotes the inclusion of Java in academic curricula through guest lectures, hack days, and professional internship opportunities.

- **Secondary School Outreach:** Establishes relationships with computer courses and after-school programs, hosting events like "Meet a Programmer" days to engage students.

Learn more at https://www.jcp.org/java-in-education

Food Sponsor

The Java platform for the modern cloud enterprise

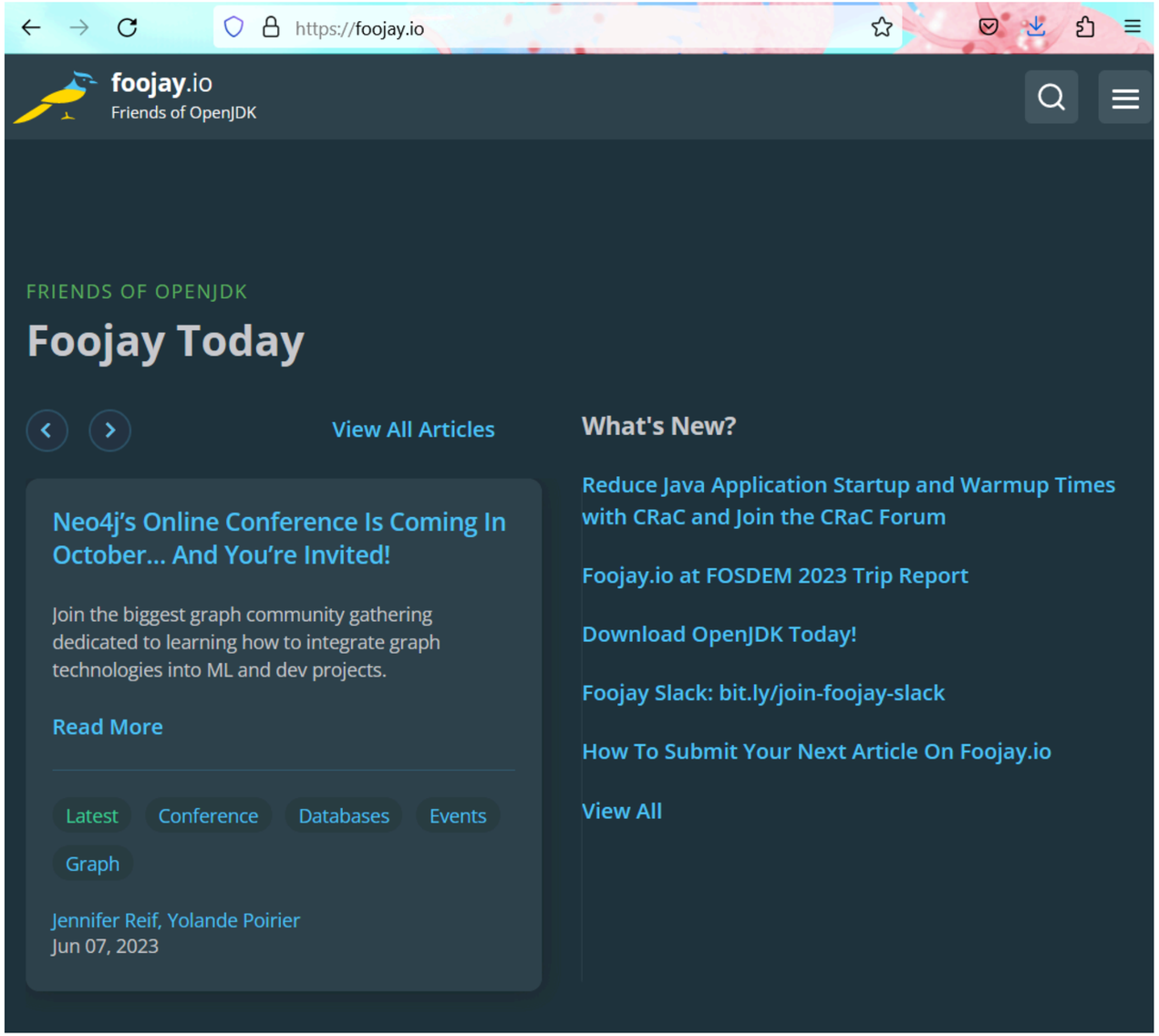
Master the Journey of OpenJDK Migration

The School of OpenJDK is your ultimate resource for mastering OpenJDK migration. Dive into our rich collection of webinars, where OpenJDK experts share their secrets. From foundational knowledge to advanced tactics, you'll leave the course with a blueprint to navigate the complexities of OpenJDK migration with confidence and ease.

Why join the School of OpenJDK Migration?

- **Expert Guidance: Learn directly from seasoned OpenJDK migration specialists who will share their insider tips and strategies**
- **Comprehensive Learning: Our curriculum includes a range of topics that prepare you for real-world challenges.**
- **Blueprint for Success: Each session equips you with actionable insights and a clear blueprint to implement OpenJDK migration effectively and efficiently.**

Join Friends of OpenJDK at Foojay.io



[How To Submit Your Next Article On Foojay.io]

[Foojay Slack: bit.ly/join-foojay-slack]

[Download OpenJDK Today!]

Java Basic Features

Syntax, OOP, Collections, Generics

Syntax

```java
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.println("************************************");
        for (int i = 0; i < 3; i++) {
            if (i == 1) {
                System.out.println("******Welcome to our training!******");
            } else {
                System.out.println("************************************");
            }
        }
        System.out.println("************************************");
        System.out.println("Hello, " + name + "! We're glad to have you here.");
        scanner.close();
    }
}
```

Set of rules that define the combinations of elements, or symbols that are considered to be correctly structured in Java.

Syntax

```java
//Package
package org.bootcamp.javaBasicFeatures.syntax;

//Comments
/*
  This file is property of Eric Marc Martin.
  Feel free to use, but I bet your examples will be spectacular!
*/

//Classes
public class Syntax {

    //Data types and variables
    int num = 5;
    char letter = 'E';
    boolean isOn = true;

    public static void main(String[] args) {
        System.out.println("Welcome to our Java Bootcamp!");
        //System.out.println("Welcome to our Hello World!");
    }

    //Methods
    private boolean isThisEasy() {
        //Operators and control flows
        if(1>2) return false;
        return true;
    }

}
```

Classes, Objects, and Methods

Blocks of code that perform tasks.

Comments

Single-line and multi-line

Data Types

int, byte, short, long, float, double, boolean

Syntax

```java
//Package
package org.bootcamp.javaBasicFeatures.syntax;

//Comments
/*
  This file is property of Eric Marc Martin.
  Feel free to use, but I bet your examples will be spectacular!
*/

//Classes
public class Syntax {

    //Data types and variables
    int num = 5;
    char letter = 'E';
    boolean isOn = true;

    public static void main(String[] args) {
        System.out.println("Welcome to our Java Bootcamp!");
        //System.out.println("Welcome to our Hello World!");
    }

    //Methods
    private boolean isThisEasy() {
        //Operators and control flows
        if(1>2) return false;
        return true;
    }

}
```

Variables

Containers for storing data values

Operators:

+, -, *, /, %, ==, !=, >, <, >=, <=, &&, ||, !

Control Flow Statements

if, if-else, switch, for, while, do-while

Syntax

```java
//Package
package org.bootcamp.javaBasicFeatures.syntax;

//Comments
/*
  This file is property of Eric Marc Martin.
  Feel free to use, but I bet your examples will be spectacular!
*/

//Classes
public class Syntax {

    //Data types and variables
    int num = 5;
    char letter = 'E';
    boolean isOn = true;

    public static void main(String[] args) {
        System.out.println("Welcome to our Java Bootcamp!");
        //System.out.println("Welcome to our Hello World!");
    }

    //Methods
    private boolean isThisEasy() {
        //Operators and control flows
        if(1>2) return false;
        return true;
    }

}
```

Packages

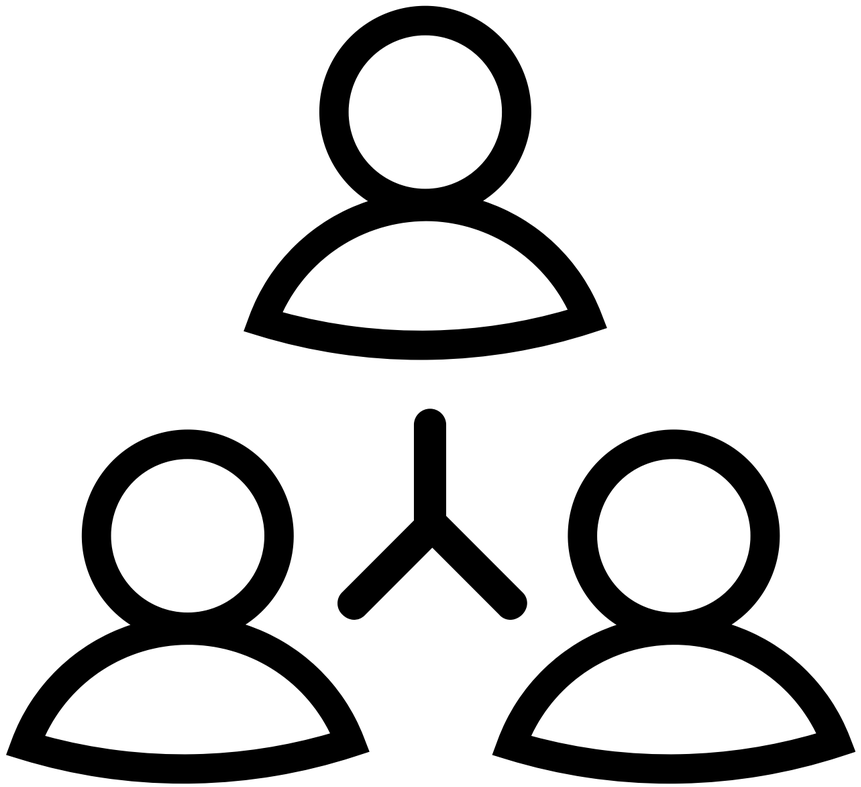
Used to group related classes

Object-Oriented Programming

Encapsulation

Practice of wrapping variables and methods that operate on the data into a single unit of class and restricting access to the internal state by exposing only select methods.

Object-Oriented Programming



Inheritance

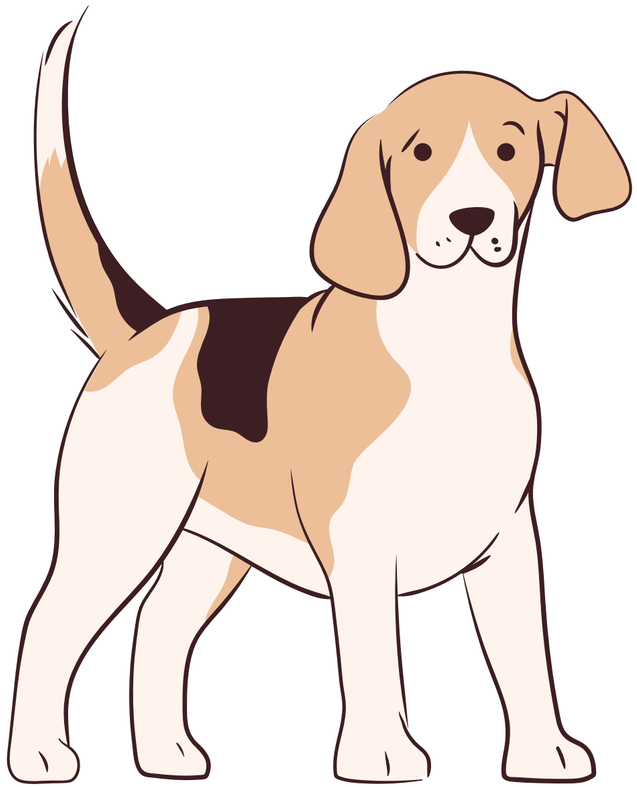
Mechanism where one subclass acquires the properties and behaviors (fields and methods) of superclass.

Object-Oriented Programming

Polymorphism

Ability of a method to operate in many ways which is achieved through method overriding and method overloading.

Object-Oriented Programming



Abstraction

Concept of hiding complex implementations details of a system and exposing only the needed parts through abstract classes or interfaces.

Collections

```java
package org.bootcamp.javaBasicFeatures.collections;

import java.util.List;

public class ArrayList {
    public static void main(String[] args) {
        // Creating an ArrayList
        List<String> names = new java.util.ArrayList<>();

        // Adding elements
        names.add("Drogon");
        names.add("Rhaegal");
        names.add("Viserion");

        // Iterating through elements
        for (String name : names) {
            System.out.println(name);
        }

        // Getting element by index
        System.out.println("First name: " + names.get(0));

        // Removing element
        names.remove( o: "Rhaegal");
        System.out.println("After removal: " + names);
    }
}
```

Data structure that allow you to store and manipulate groups of objects.

Collection Interfaces

- List
- Set
- Map
- Queue
- Duque

Generics

```java
package org.bootcamp.javaBasicFeatures.generics;

// Generic class Box
public class Box<T> {
    private T content;

    // Constructor
    public Box(T content) { this.content = content; }

    // Getter and setter for content
    public T getContent() { return content; }

    public void setContent(T content) { this.content = content; }

    // Example method using generics
    public void displayBoxContent() { System.out.println("Box contains: " + content); }

    public static void main(String[] args) {
        // Create a Box of Integer
        Box<Integer> integerBox = new Box<>( content: 123);
        integerBox.displayBoxContent();

        // Create a Box of String
        Box<String> stringBox = new Box<>( content: "Hello, Generics!");
        stringBox.displayBoxContent();
    }
}
```

Feature that allows you to write a code that can work with different types of objects while ensuring safety.

Java 8-12

- Lambdas and Functional Interfaces, Optional, Stream API
- Private methods in Interfaces, Immutable Factory methods for List, Set and Map
- Local Variable Type Inference: The var keyword

Java 8-12

- Files’ readString and writeString
- The Not Predicate method
- Old Switch Expression vs New Switch Expression

Lambdas and Functional Interfaces

```java
package org.bootcamp.oldfiles.java8.lambdasAndFunctionalInterface;

// Functional Interface
interface MathOperation {
    int operate(int a, int b);
}

public class LambdasAndFunctionalInterfaces {
    public static void main(String[] args) {
        // Lambda expression to implement addition
        MathOperation addition = new MathOperation() {
            @Override
            public int operate(int a, int b) { return a + b; }
        };

        // Lambda expression to implement subtraction
        MathOperation subtraction = (a, b) -> a - b;

        // Using the functional interfaces with lambdas
        System.out.println("10 + 5 = " + operate( a: 10,  b: 5, addition));
        System.out.println("10 - 5 = " + operate( a: 10,  b: 5, subtraction));
    }

    // Method that accepts a functional interface as a parameter
    public static int operate(int a, int b, MathOperation operation) { return operation.operate(a, b); }
}
```

Lambdas provide a brief way to implement functional interfaces, which are interfaces with a single abstract method, making the code more readable and expressive.

Optional

```java
public static void main(String[] args) {
    // Create a User with an email
    User userWithEmail = new User( name: "Daenerys", Optional.of( value: "Daenerys@example.com"));

    // Create a User without an email
    User userWithoutEmail = new User( name: "Jon",  email: Optional.empty());
```

Optional is a container object used to represent the presence or absence of a value, helping to avoid null checks and potential NullPointerException.

Stream API

```
23 @    private static List<String> getCarNamesWithFirstLetterT(List<String> carNames) {
24          return carNames.stream().filter(carName -> carName.startsWith("T"))
25                  .map(String::toUpperCase).toList();
26      }
27  }
```

Stream API provides a functional approach to process collections of objects, enabling operations like filtering, mapping, and reducing elements with concise and fluent syntax.

Private methods in Interfaces

```java
interface CollectionOperations {
    // Default method to perform operations on collections
    default void performOperations() {
        List<Integer> integerList = createImmutableList();
        Set<String> stringSet = createImmutableSet();
        Map<String, Integer> stringIntegerMap = createImmutableMap();

        printList(integerList);
        printSet(stringSet);
        printMap(stringIntegerMap);
    }

    // Private method to print items in a list
    private void printList(List<Integer> list) { list.forEach(this::printIt

    // Private method to print items in a set
    private void printSet(Set<String> set) { set.forEach(this::printItem); }

    // Factory method to create an immutable list
    default List<Integer> createImmutableList() { return List.of(1, 2, 3, 4, 5); }
```

Private methods in interfaces allow for encapsulating reusable code within the interface itself, accessible only to other default or static methods within the same interface.

Immutable Factory methods for List Set and Map

Immutable factory methods provide a way to create unmodifiable lists, ensuring that their content cannot be changed after creation, enhancing program reliability and predictability.

```
30      // Factory method to create an immutable list
31      default List<Integer> createImmutableList() { return List.of(1, 2, 3, 4, 5); }
34
35      // Factory method to create an immutable set
36      default Set<String> createImmutableSet() { return Set.of("A", "B", "C", "D"); }
39
40      // Factory method to create an immutable map
41      default Map<String, Integer> createImmutableMap() { return Map.of( k1: "one",  v1: 1,  k2: "two",  v2: 2,  k3: "three",  v3: 3); }
44
```

Local Variable Type Inference: Var Keyword

```java
// Without var
List<String> names = new ArrayList<>();

// With var
var names = new ArrayList<String>();
```

```java
// Original declaration
var names = new ArrayList<String>();

// Later change to LinkedList
var names = new LinkedList<String>();
```

```java
// Without var
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    // process entry
}

// With var
for (var entry : map.entrySet()) {
    // process entry
}
```

Var keyword introduces local variable type inference, allowing developers to declare variables without explicitly stating the type, as long as it can be inferred from the initializer.

Files’ readString and writeString

```
Files.writeString(Paths.get( first: "students.txt"), stringBuilder);

Files.readString(Paths.get( first: "students.txt")).split( regex: "\n");
```

Files.readString and Files.writeString are methods that allow reading the content of a file into a string and writing a string to a file, respectively, simplifying file handling operations.

The Not Predicate method

```
for (var student : readList) {
    if (!isGradeCOrLower(student)) {
        System.out.println("Student: " + student);
    }
}
```

Negate() method of the Predicate interface creates a new predicate that negates the result of the original predicate, effectively reversing the condition it evaluates.

Old Switch Expression vs New Switch Expression

```java
switch (day) {
    case "Monday":
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
    case "Friday":
        typeOfDay = "Weekday";
        break;
    case "Saturday":
    case "Sunday":
        typeOfDay = "Weekend";
        break;
    default:
        throw new IllegalArgumentException("Invalid day: " + day);
}
```

Old switch statement uses fall-through behavior for cases, while the new switch expression introduced in Java 12 uses a simplified syntax to return values directly from each case.

```java
String typeOfDay = switch (day) {
    case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" -> "Weekday";
    case "Saturday", "Sunday" -> "Weekend";
    default -> throw new IllegalArgumentException("Invalid day: " + day);
};
```

Activity Link

https://github.com/ericmarcmartin/devcon.bootcamp.git

WIFI Password:
BrokenWires@@2019

Afternoon Session

Java 14 to 21

Instance Main methods, Unnamed classes, Sealed Classes, Records, Switch Expressions, Pattern Matching, Sequenced Collections

Meet
JANSEN ANG

FREELANCE JAVA DEVELOPER

- Jansen Ang is a one of the leaders of the Java User Group Philippines.
- He is currently a freelance Java developer and has been developing Java applications with various companies throughout the years.
- He dedicates his time to contributing to the Java community and organizing JUG PH meetups.

Jansen Ang

https://www.linkedin.com/in/jansen-ang/

https://github.com/jmgang

About JEP 445

- JEP 445 aims to make Java easier to learn by allowing beginners to write their first programs without needing to understand advanced language features.

- Constructs like classes, access modifiers (public), and keywords (static) relate to "programming in the large" and should only be introduced when necessary.

Instance Main Methods

- Removes the need for *static*, *public*, or a *String[]* modifiers/parameter.
- Simplifies the entry point to a program
- If you have both the traditional *public static void main(String[] args)* and the instance *main()* the traditional version takes precedence.

Unnamed Classes

- Useful for standalone programs or as an entry point to a program.
- Are final and reside in the unnamed package.
- The *.class* file name on the hard disk is based on the source file name
- Are similar to normal classes but have only the default constructor provided by the compiler.
- Cannot be referred to by name, so instances cannot be constructed directly.
- Must have a *main()* method.

Sealed Classes

- Allow us to control the inheritance scope by specifying the permissible subclasses.

- Also applies to interfaces, where we can define the specific classes or interfaces that can implement and extend the interface, respectively.

Sealed Classes

- Direct subclasses of sealed classes must be defined as either final, sealed, or non-sealed.

Keyword	Purpose
sealed	Specifies that a class or interface can only be extended or implemented by certain designated classes or interfaces.
permits	Specifies that a class or interface can only be extended or implemented by certain designated classes or interfaces.
non-sealed	Indicates that a subclass of a sealed class can be further extended by any other classes, without restrictions.

Records

- Records are a unique type of class designed to reduce boilerplate code, often referred to as *data carriers*.
- Records are immutable and are inherently final.
- Records implicitly extend from the Record class.
- Records are declared using a record declaration, where the "components" of the record are specified.

Records

- Records can contain instance methods, static fields and static methods but not instance fields.

- Implicitly generated features include:
  - A canonical constructor.
  - A *toString()*, *equals()* and *hashCode()* methods.
  - Public accessor methods of declared components

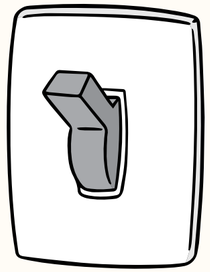
- Can override the generated methods/constructors.

Switch Expressions

- A more compact version of a switch statement that can return a value.

- The result of a switch expression can be assigned to a variable, requiring all branches to return a compatible data type.

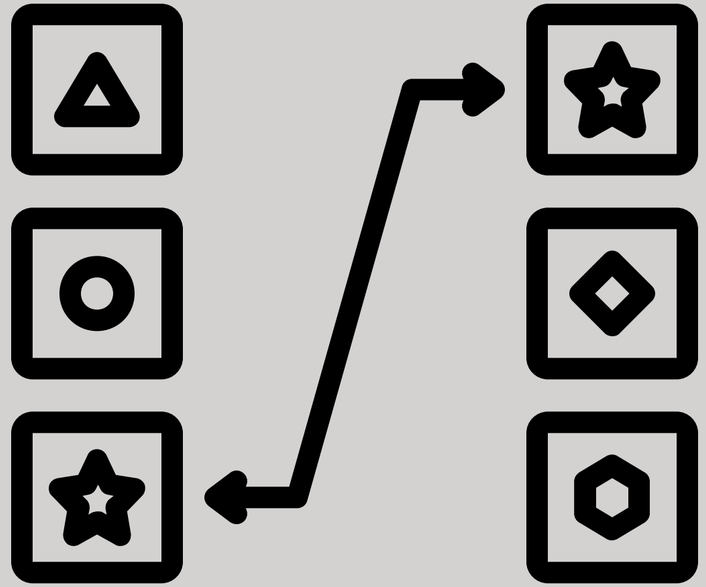
- Two types of branches: an *expression* and a *block*, each with specific syntax rules.
  - Blocks are enclosed in braces {} and use a *yield* statement to return a value if needed.



Trad. Switch Statements vs Switch Expressions

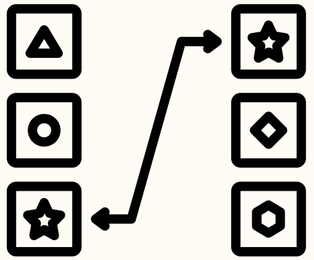
Feature	Traditional Switch Statement	Switch Expression
Syntax	Uses *switch* keyword, *case*, and *break*	Uses *switch* keyword with -> or block expressions with *yield*
Break Statement	Required to prevent fall-through	Not required; each case is independent
Return Values	Does not return values	Returns a value and can be used in assignments
Default Branch Requirement	Not required if no default case is needed	Required unless all cases are covered or no value is returned

Java Pattern Matching



Pattern Matching with Type Pattern

- Java 16 introduced pattern matching with *if* statements and the *instanceof* operator.
- Pattern matching controls program flow by executing code sections that meet specific criteria.
- It also helps reduce boilerplate code.
- Type patterns with the *instanceof* operator simplify type checking and casting by combining both into a single expression.

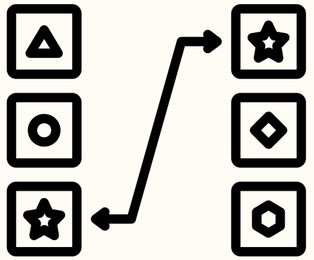


Record Patterns

- Includes a type, a list of component patterns (which can be empty), and an optional identifier.

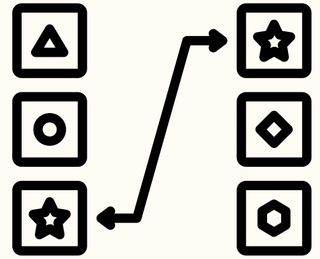
- Performs two actions:
  - They check if an object passes the *instanceof* test.
  - They break down the record instance into its individual components.

- Record patterns allow for nesting.



Pattern Matching for Switch

- Well-suited for pattern matching, eliminating the need for *instanceof* and casting.
- Case labels can now include patterns and *null*.
- Guarded Patterns enables us to separate case labels, patterns, and conditional logic from business logic (via the *when* keyword).
- Selector Expression Types broadened to Integral primitive types (excluding long) and any reference type.
- Label dominance ensures that more specific case labels are evaluated before more general ones.



Problem with Collections before Java 21

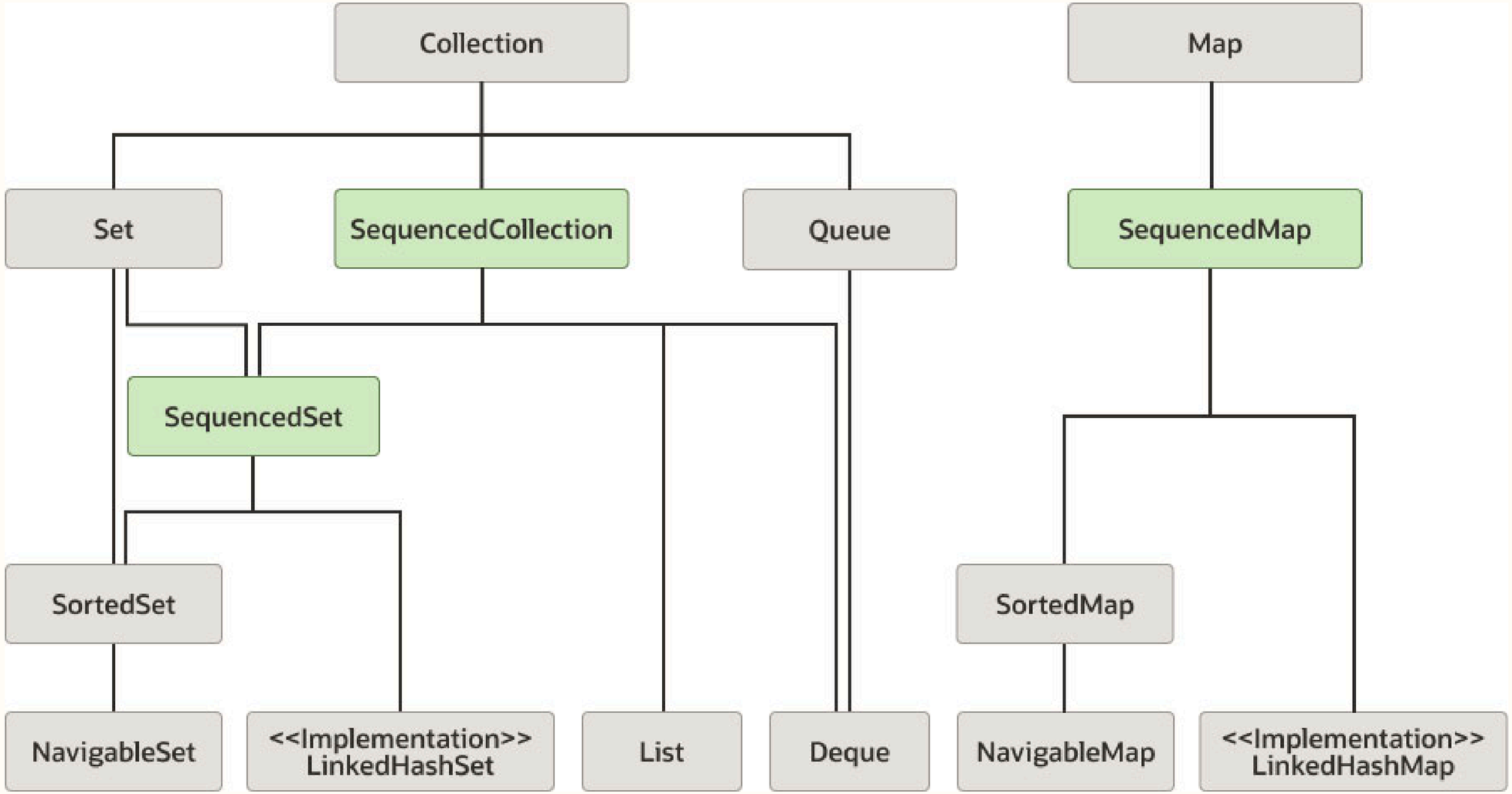
	Getting First Element	Getting Last Element
List	list.get(0)	list.get(list.size() - 1)
Deque	deque.getFirst()	deque.getLast()
HashSet	set.iterator().next()	None
LinkedHashMap	linkedHashMap.entrySet().iterator().next()	None

Many implementations support getting the first or last element, but each collection defines its own method.

Sequenced Collections

- JEP 431 establishes new interfaces for collections with a defined encounter order.
- Standardizes methods for accessing the first and last elements, and for processing elements in reverse order.
- Introduces sequenced collections, sequenced sets, and sequenced maps.
- Ensures consistent operations related to encounter order across different collections.
- All methods in these interfaces come with default implementations.

Sequenced Collections



Taken from https://docs.oracle.com/en/java/javase/21/core/creating-sequenced-collections-sets-and-maps.htm

We’d like to know your feedback!