Inventive Media

# Java Programming Essentials - Day 2

Jansen Ang

# What are we learning today?

- Type Casting
  - Implicit and Explicit Casting
- Modifiers
  - Access Modifiers
  - Non-Access Modifiers
- OOP II
  - Method Overloading
  - Constructor Overloading
  - this Keyword
  - Stack vs Heap
  - Primitive vs Reference Variables
  - Wrapper Classes
  - Autoboxing vs Unboxing
- ArrayList
  - Arrays vs ArrayList
  - Arraylist Operations

# What are we learning today?

- Separating Program Logic from Console UI
- Programming Paradigms and Algorithms
  - Paradigms
  - Searching and Sorting
    - Bubble Sort
    - Binary Search
  - Arrays.sort() and Arrays.binarySearch()

# Type Casting

# Type Casting in Java

Type casting in Java is the process of converting a variable from one data type to another. In Java, there are two types of casting:

1. **Widening Casting (Implicit)**: Automatically converting a smaller type to a larger type size.
    - Example: byte to short, short to int, int to long, long to float, float to double.
2. **Narrowing Casting (Explicit)**: Manually converting a larger type to a smaller size type.
    - Example: double to float, float to long, long to int, int to short, short to byte.

**Why is Type Casting Used?**

- **Compatibility**: Sometimes, it's necessary to convert data types to make them compatible with other data types in expressions or method calls.
- **Specific Operations**: Certain operations may require values to be in a specific type. For example, integer division vs. floating-point division.
- **API Requirements**: Some APIs or libraries may require data in a specific type.
- **Memory Management**: In some cases, casting to smaller types can save memory.

# Type Casting in Java

**Widening Casting (Implicit)**

byte to short and short to int

```java
byte byteVal = 42;
short shortVal = byteVal;
System.out.println("byteVal: " + byteVal); // Outputs 42
System.out.println("shortVal: " + shortVal); // Outputs 42

short shortVal = 1000;
int intVal = shortVal;
System.out.println("shortVal: " + shortVal); // Outputs 1000
System.out.println("intVal: " + intVal); // Outputs 1000
```

**Narrowing Casting (Explicit)**

double to float and long to int

```java
double doubleVal = 9.99;
float floatVal = (float) doubleVal;
System.out.println("doubleVal: " + doubleVal); // Outputs 9.99
System.out.println("floatVal: " + floatVal); // Outputs 9.99


long longVal = 100000L;
int intVal = (int) longVal;
System.out.println("longVal: " + longVal); // Outputs 100000
System.out.println("intVal: " + intVal); // Outputs 100000
```

# Modifiers

# Modifiers

Modifiers in Java are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following:

1. **Access Modifiers**: Control the access level.
2. **Non-Access Modifiers**: Provide other functionalities.

# Access Modifiers

Access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

**Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

```
class Car {
    private String model;

    private void display() {
        System.out.println("Model: " + model);
    }
}
```

**Default**: If no modifier is specified, Java uses a default access level. The default modifier is accessible within the same package.

```
class Vehicle {
    protected String brand = "Ford";

    protected void honk() {
        System.out.println("Tuut, tuut!");
    }
}
```

# Access Modifiers

**Protected:** The protected access level is accessible within the same package and also available to subclasses.

```
class Vehicle {
    protected String brand = "Ford";

    protected void honk() {
        System.out.println("Tuut, tuut!");
    }
}
```

**Public:** The public access level has the widest scope among all other modifiers. Classes, methods, or fields which are declared as public can be accessed from anywhere.

```
public class Program {
    public String name = "Program";

    public void run() {
        System.out.println("Program is running.");
    }
}
```

# Non-Access Modifiers

Java provides several non-access modifiers to achieve many other functionalities.

a. **Static**: The static keyword is used primarily for memory management. It can be applied to variables, methods, blocks, and nested classes.
   i. Static variables and methods belong to the class, not instances of the class.
   ii. Static methods can be accessed directly by the class name and don't need any object.
   iii. A common use is for constant values or utility methods.

```java
class Student {
    static int studentCount = 0; // Static variable

    Student() {
        studentCount++;
    }

    static void displayCount() { // Static method
        System.out.println("Number of students: " + studentCount);
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        new Student();
        new Student();
        Student.displayCount(); // Output: Number of students:
    }
}
```

# Non-Access Modifiers

**Final Modifier**

- The **final** keyword is used to restrict the user. It can be used with variables, methods, and classes.
- **Key Features**:
  - A final variable can be initialized only once.
  - A final method cannot be overridden by subclasses.
  - A final class cannot be subclassed.

```
final class Base { // Final class

    final void display() { // Final method
        System.out.println("This method cannot be overridden.");
    }
}

class Derived /* extends Base */ { // Error: Cannot extend a final class
    // Error: display() in Derived cannot override display() in Base
}
```

# Non-Access Modifiers

**Abstract Modifier**

- The abstract keyword is used to declare abstract classes or methods.
- **Key Features:**
  - Abstract classes cannot be instantiated, but they can have a subclass.
  - Abstract methods are declared without an implementation.

```java
interface Animal {
    void sound(); // Abstract method in interface
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Woof");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound(); // Output: Woof
    }
}
```

# Other Non-Access Modifiers

**Transient Modifier**

- Use: The transient keyword is used in serialization. It skips the serialization of the variables it is applied to.
- **Key Features:**
  - Transient variables are not part of the serialized object state.
  - Useful in hiding sensitive data during serialization.

**Synchronized Modifier**

- **Use**: The **synchronized** keyword is used in multithreading. It can be applied to methods or blocks.
- **Key Features**:
  - Synchronized code can only be executed by one thread at a time.
  - Helps in solving issues of thread interference and memory consistency errors.

**Volatile Modifier**

- **Use**: The **volatile** keyword is used in the context of multithreading.
- **Key Features**:
  - A volatile variable’s value will always be read from and written to the main memory.
  - It ensures the visibility of changes to variables across threads.

# Object-Oriented Programming II

# Method Overloading

a feature that allows you to have more than one method with the same name in a class, as long as each method has a different parameter list. This is a form of polymorphism that provides the flexibility to call a similar method for different types of data.

**Key points:**

- Overloaded methods **must** have a different number of parameters or parameters of different types.
- They can have different return types, but the return type alone is not sufficient for overloading.
- Overloading is determined at compile time.

# Method Overloading Example

Consider a class Calculator that performs addition. We can overload the add method to handle different data types or numbers of parameters.

```java
class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Invoke add method for two integers
        System.out.println(calc.add(10, 20));  // Output: 30

        // Invoke add method for three integers
        System.out.println(calc.add(10, 20, 30));  // Output: 60

        // Invoke add method for two double values
        System.out.println(calc.add(5.5, 4.5));  // Output: 10.0
    }
}
```

When you call the **add** method, Java determines which version of the method to use based on the number and type of arguments passed. This is an example of compile-time polymorphism.

# Method Overloading inside Java

Method Overloading in System class' println methods

```
    Terminates the current line by writing the line separator string. The line separator string is defined
    by the system property line.separator, and is not necessarily a single newline character ('\n').
public void println() { newLine(); }

    Prints a boolean and then terminate the line. This method behaves as though it invokes print
    (boolean) and then println().
    Params: x – The boolean to be printed
public void println(boolean x) {...}

    Prints a character and then terminate the line. This method behaves as though it invokes print
    (char) and then println().
    Params: x – The char to be printed.
public void println(char x) {...}
```

# Constructors

are special methods used to initialize objects. They are called when an instance of a class is created.

- **Purpose**: Constructors initialize the new object and its properties.
- **Naming**: A constructor must have the same name as the class and cannot have a return type.
- **Types**: Constructors can be overloaded, meaning a class can have more than one constructor with different parameters.

```java
class Dog {
    String breed;

    // Parameterized constructor
    Dog(String dogBreed) {
        breed = dogBreed;
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Labrador"); // Calls the parameterized constructor.
        System.out.println("The breed of my dog is: " + myDog.breed);
    }
}
```

# Default Constructor

If no constructor is defined in a class, Java automatically provides a default constructor.

```java
class Dog {
    // No constructor is defined here, so Java provides a default constructor.
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog(); // Calls the default constructor provided by Java.
        System.out.println("A new dog has been created!");
    }
}
```

When an object of Dog class is created in the main method, this default constructor is called

# Constructor Overloading

Constructor overloading in Java occurs when a class has more than one constructor with different parameters. It allows different ways to initialize an object, providing flexibility in object creation.

**Characteristics of Constructor Overloading:**

1. **Multiple Constructors**: A class will have multiple constructors, each with a unique signature.
2. **Different Parameter Lists**: Constructors differ in the number, type, or order of parameters.
3. **Flexibility in Object Creation**: Enables creating objects with different states based on available data.
4. **Same Class Name**: All overloaded constructors have the same name as the class.

**Benefits of Constructor Overloading:**

1. **Clarity and Simplicity**: It provides clear ways to create objects based on what data is available.
2. **Flexibility and Reusability**: Enhances class flexibility and encourages code reusability.
3. **Initialization Variability**: Allows different ways to initialize an object's properties.

# Constructor Overloading Example

Consider a class Calculator that performs addition. We can overload the add method to handle different data types or numbers of parameters.

```java
class Dog {
    String name;
    int age;

    // Constructor with no parameters
    Dog() {
        this.name = "Unknown";
        this.age = 0;
    }

    // Constructor with one parameter
    Dog(String name) {
        this.name = name;
        this.age = 0;
    }

    // Constructor with two parameters
    Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

```java
public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
        Dog dog2 = new Dog("Buddy");
        Dog dog3 = new Dog("Max", 5);

        dog1.displayInfo();
        dog2.displayInfo();
        dog3.displayInfo();
    }
}
```

# **this** Keyword

is a reference variable that refers to the current object. It's used within an instance method or a constructor to refer to the current object on which the method or constructor is being invoked.

**Uses of this Keyword:**

1. **Referencing Instance Variables**: Differentiates instance variables from parameters or local variables if they have the same names.
2. **Calling Other Constructors**: In constructor overloading, this can be used to call another constructor in the same class.
3. **Returning the Current Object:** Useful in method chaining where a method returns the current object.

# **this** Keyword Example

```java
class Dog {
    String name;
    String breed;
    int age;

    // Default constructor
    Dog() {
        this("Unknown", "Unknown", 0); // Calling three-argument construct
    }

    // Constructor with one parameter
    Dog(String name) {
        this(name, "Unknown", 0); // Calling three-argument constructor
    }

    // Constructor with two parameters
    Dog(String name, String breed) {
        this(name, breed, 0); // Calling three-argument constructor
    }

    // Constructor with three parameters
    Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    void bark() {
        System.out.println(name + " says Woof!");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog(); // Calls default constructor
        Dog dog2 = new Dog("Buddy"); // Calls one-parameter constr
        Dog dog3 = new Dog("Charlie", "Labrador"); // Calls two-pa
        Dog dog4 = new Dog("Max", "Beagle", 3); // Calls three-par

        dog1.bark();
        dog2.bark();
        dog3.bark();
        dog4.bark();
    }
}
```

# Stack vs Heap

It's important to understand where your program's data is stored: this is mainly in two places, the stack and the heap.

### The Stack

- **What is it?**
  - Imagine a stack of books. Each time you start a method, you place a new book on top. This book contains all the local variables and details needed for that method. When the method ends, you remove the book. That's your stack.
- **Characteristics:**
  - It's very orderly. Every new method call adds one "book" to the top, and when the method is done, that book is removed.
  - It's fast because you always know where the top book is.
  - If you run out of space in your stack (too many books), you get a "stack overflow" error.
- **Usage:**
  - Used for all the local variables and the details of each method call.
  - Each thread has its own stack.

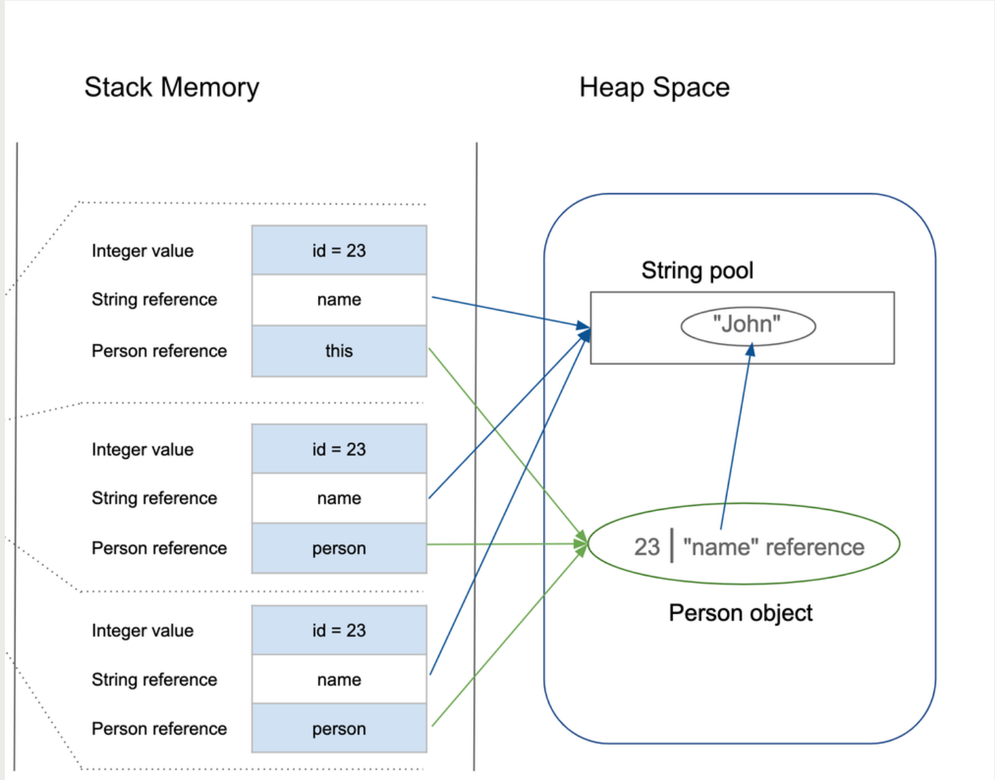
# Stack vs Heap

**The Heap**

- **What is it?**
  - Now, think of a heap as a big, messy pile of objects. When you create an object (using new), it's like tossing a new item onto this pile.
  - Objects can be big or small and can come and go at any time.
- **Characteristics:**
  - It's where all your objects (and their fields) live.
  - It's more complex than the stack because things aren't as neatly organized.
  - Objects stay in the heap until they're no longer needed, and then they're cleaned up by the garbage collector.
- **Usage:**
  - Used for all objects your program creates.

# Primitive vs Reference Variables

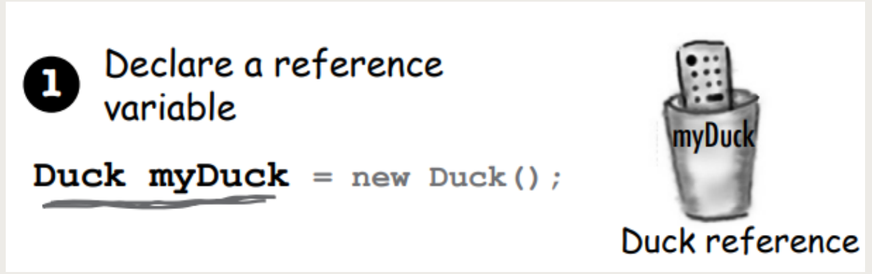
- **Primitive Variables**:
  - Stored on the **Stack**.
  - These include types like **int**, **double**, **char**, **boolean**, etc.
  - When you declare a primitive variable within a method, Java allocates memory for it on the stack. This memory is automatically freed up when the method call ends.
- **Reference Variables**:
  - The references themselves are stored on the **Stack**.
  - The objects they refer to are stored on the **Heap**.
  - When you create an object (e.g., **MyObject obj = new MyObject();**), the actual **MyObject** instance is placed in the heap, and the **obj** reference pointing to it is placed in the stack.

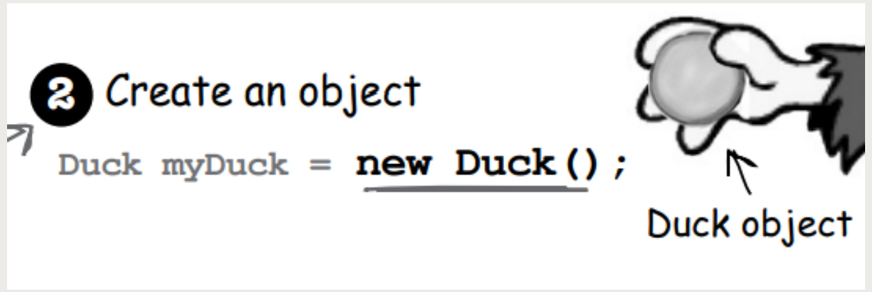


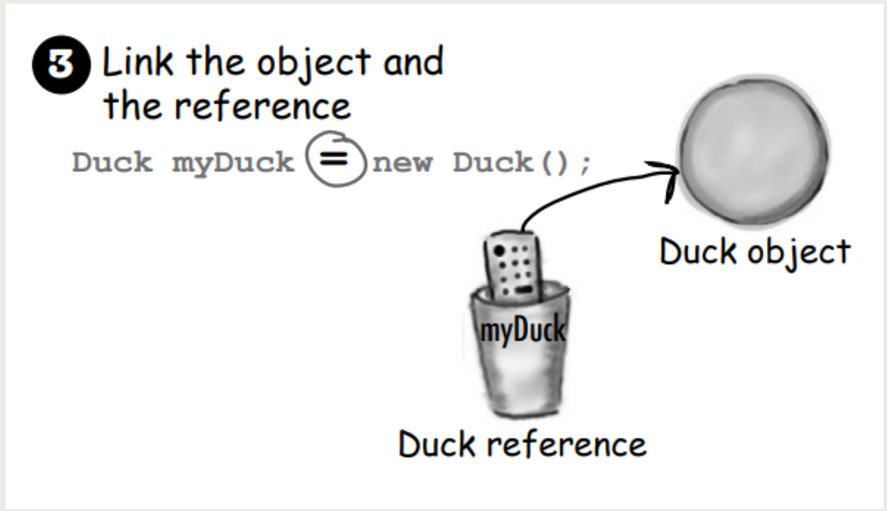
# Objects vs References

A **reference** is a variable that holds the memory address (location) of an object in the heap.

- It acts like a pointer or a remote control to the object, allowing you to access and manipulate the object's data and call its methods.
- References are typically stored in stack memory when they are local variables, or part of the heap if they are attributes of other objects.
- If a reference variable doesn't point to any object, it can be assigned the value null.




**Objects** are instances of classes. They are created using the new keyword and reside in heap memory.

- An object is the actual entity that contains the data (attributes) and the methods (behaviors) defined by the class.
- Objects have a lifespan that extends beyond the scope in which they were created, persisting in memory until they are no longer referenced and are garbage collected.

# Wrapper Classes

Wrapper classes in Java provide a way to use primitive data types (int, char, etc.) as objects.

Each primitive data type in Java has a corresponding wrapper class:

- int → **Integer**
- char → **Character**
- byte → **Byte**
- short → **Short**
- long → **Long**
- float → **Float**
- double → **Double**
- boolean → **Boolean**

**Purpose**

Wrapper classes serve several purposes:

1. **Object Creation**: They allow primitive values to be treated as objects. This is useful when you need to store primitive values in object collections like ArrayList or HashMap.
2. **Utility Methods**: These classes provide useful methods. For example, Integer has methods like parseInt, compare, and valueOf.
3. **Null Support**: Unlike primitives, wrapper objects can be null.

# Autoboxing and Unboxing

Java automatically converts between primitive and wrapper types:

- **Autoboxing**: Automatic conversion of primitive types to their corresponding wrapper class objects.
- **Unboxing**: Automatic conversion of wrapper class objects back to their corresponding primitive types.

**When to Use**

- When working with collections or frameworks that require objects, not primitives.
- When you need methods and utilities provided by wrapper classes.
- When you require to store **null** as a value, which isn't possible with primitives.

# Wrapper Classes Examples

## Creating Wrapper Class Objects

```
Integer myInteger = new Integer(10);  // Deprecated method
Integer anotherInteger = Integer.valueOf(20);  // Preferred method
Double myDouble = new Double(5.99);  // Deprecated method
Double anotherDouble = Double.valueOf(3.14);  // Preferred method
Character myChar = new Character('a');  // Deprecated method
Character anotherChar = Character.valueOf('b');  // Preferred method
```

## Autoboxing and Unboxing

```
// Autoboxing: Converting primitives to wrapper objects automatically
Integer autoBoxedInt = 50;  // Primitive int to Integer object
Double autoBoxedDouble = 4.5;  // Primitive double to Double object

// Unboxing: Converting wrapper objects to primitives automatically
int unboxedInt = autoBoxedInt;  // Integer object to primitive int
double unboxedDouble = autoBoxedDouble;  // Double object to primitive double
```

## Utility Methods of Wrapper Classes

```
int parsedInt = Integer.parseInt("123");
double parsedDouble = Double.parseDouble("45.67");

// Converting primitives to String
String intAsString = Integer.toString(10);
String doubleAsString = Double.toString(3.14159);

// Comparing two numbers
int comparisonResult = Integer.compare(10, 20);  // Returns negative if first < second
```

# ArrayList

# ArrayList

- **Specifics**: ArrayList is a resizable-array implementation of the List interface. It provides a way to work with dynamic arrays.
- **Advantages**: It's more flexible than a traditional array because it can resize itself automatically when elements are added or removed.
- **Usage**: Commonly used in scenarios where you need to frequently add and access elements, but not as often delete them.

**Why ArrayLists are Needed**

- **Dynamic Sizing**: Unlike arrays, they automatically adjust their size.
- **Ease of Operations**: Provide methods for easy manipulation of data, like add(), remove(), indexOf(), etc.
- **Flexibility**: Can hold any type of objects, including another collection.
- **Useful APIs**: Rich set of APIs for various operations like searching, sorting, and iterating.

# Arrays vs ArrayList

## Arrays

- **Fixed Size**: Once an array is created, its size cannot be changed.
- **Type-Specific**: Arrays can store primitive data types or objects, but the type must be declared and cannot be changed after creation.
- **Efficiency**: Faster access to elements, as arrays use indexed access and have a fixed memory layout.
- **Memory Allocation**: Allocated on the heap when created.
- **Syntax**: int[] numbers = new int[5];
- **Length Property**: Arrays have a length property to get the size.
- **Limitation**: Adding or removing elements requires manual resizing and copying to a new array.

## ArrayList

- **Dynamic Size**: ArrayList can grow and shrink in size dynamically.
- **Object Storage**: Can only store objects (or wrapper classes for primitives) and not primitive types directly.
- **Flexibility**: Provides methods for easy addition, removal, and manipulation of elements.
- **Memory Overhead**: More memory overhead than arrays due to the additional features and flexibility.
- **Syntax**: ArrayList<Integer> numbers = new ArrayList<>();
- **Size Method**: Uses the size() method to get the number of elements.
- **Convenience**: Provides built-in methods for common operations like searching, sorting, etc.

# ArrayList with an Object

## Lists with Wrapper Classes

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

public class WrapperClassListExample {
    public static void main(String[] args) {
        // Creating a List of Integer wrapper class
        List<Integer> integerList = new ArrayList<>();

        // Adding elements
        integerList.add(1);
        integerList.add(2);
        integerList.add(3);

        // Iterating over the list
        for (Integer number : integerList) {
            System.out.println(number);
        }
    }
}
```

## List with Custom Object

```java
public class Dog {
    String name;
    int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Dog{name='" + name + "', age=" + age + "}";
    }
}
```

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

public class CustomObjectListExample {
    public static void main(String[] args) {
        // Creating a List of Dog objects
        List<Dog> dogList = new ArrayList<>();

        // Adding Dog objects to the list
        dogList.add(new Dog("Buddy", 3));
        dogList.add(new Dog("Charlie", 5));
        dogList.add(new Dog("Max", 2));

        // Iterating over the list
        for (Dog dog : dogList) {
            System.out.println(dog);
        }
    }
}
```

# Object with an ArrayList

Consider a Bookshelf class, which contains an ArrayList of Book objects. Each Book has a title and an author.

```java
class Book {
    String title;
    String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}

class Bookshelf {
    ArrayList<Book> books;

    Bookshelf() {
        books = new ArrayList<>();
    }

    void addBook(Book book) {
        books.add(book);
    }

    void removeBook(Book book) {
        books.remove(book);
    }

    void displayBooks() {
        for (Book book : books) {
            System.out.println(book.title + " by " + book.author);
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Bookshelf shelf = new Bookshelf();
        shelf.addBook(new Book("1984", "George Orwell"));
        shelf.addBook(new Book("To Kill a Mockingbird", "Harper Lee"));

        shelf.displayBooks();
    }
}
```

# ArrayList Useful Methods

**add(element)**: Adds an element to the end of the list.

```
list.add("Hello");
```

**remove(index)** or **remove(object):** Removes the element at the specified position or the first occurrence of the specified element.

```
list.remove(0); // removes the first element
```

**get(index):** Returns the element at the specified position.

```
String element = list.get(1);
```

**size()**: Returns the number of elements in the list.

```
int size = list.size();
```

**contains(object):** Returns true if the list contains the specified element.

```
boolean exists = list.contains("Hello");
```

**set(index, element):** Replaces the element at the specified position with the specified element.

```
list.set(1, "World");
```

# Separating the user interface from program logic

# Separating UI from Program Logic

Separating UI (User Interface) from Program Logic is a design principle in software development where the code responsible for interacting with the user is kept distinct from the code that handles the core business logic or data processing. This principle is sometimes referred to as a "separation of concerns."

**Why is it Helpful?**

1. **Maintainability**: Changes in the business logic or UI can be made independently without affecting the other.
2. **Reusability**: The same business logic can be reused with different user interfaces (e.g., console, web, mobile).
3. **Testability**: Business logic can be tested separately from the UI, facilitating more straightforward and efficient testing processes.
4. **Clarity**: This separation makes the code easier to understand, organize, and manage.

# Separating UI from Program Logic

Calculator Program without Separating UI from Program Logic

```java
import java.util.Scanner;

public class SimpleCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number: ");
        int a = scanner.nextInt();
        System.out.print("Enter second number: ");
        int b = scanner.nextInt();

        int sum = a + b;
        System.out.println("The sum is: " + sum);
    }
}
```

# Separating UI from Program Logic

Calculator Program with Separating UI from Program Logic

```java
public class CalculatorLogic {
    public int add(int a, int b) {
        return a + b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }
}
```

```java
import java.util.Scanner;

public class ConsoleUI {
    private Scanner scanner = new Scanner(System.in);

    public int getNumberFromUser(String prompt) {
        System.out.print(prompt);
        return scanner.nextInt();
    }

    public void displayMessage(String message) {
        System.out.println(message);
    }
}
```

```java
public class MainApp {
    public static void main(String[] args) {
        ConsoleUI ui = new ConsoleUI();
        CalculatorLogic calculator = new CalculatorLogic();

        int a = ui.getNumberFromUser("Enter first number: ");
        int b = ui.getNumberFromUser("Enter second number: ");

        int sum = calculator.add(a, b);
        ui.displayMessage("The sum is: " + sum);
    }
}
```

# Programming Paradigms and Algorithms

# Introduction to Programming Paradigms

A programming paradigm is a fundamental style of computer programming. It's not about syntax or code; it's more about how you structure and approach problem-solving in programming.

**Importance**: Different paradigms can lead to different solutions for the same problem, and some paradigms are more suited to certain types of problems than others.

**Types of Programming Paradigms**

1. **Procedural Programming**: Focuses on a sequence of actions or commands. Programs are structured as a series of procedures, or routines, with clear steps.
   - Useful for: Straightforward, step-by-step processes.
   - Example: C programming language.
2. **Object-Oriented Programming (OOP)**: Based on the concept of "objects", which can contain data and code: data in the form of fields (attributes), and code, in the form of procedures (methods).
   - Useful for: Applications with complex data models, like GUI applications or games.
   - Example: Java, Python.
3. **Functional Programming (FP)**: Treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.
   - Useful for: Concurrent programming, programs with complex data processing.
   - Example: Haskell, Scala.

# Procedural Programming

In procedural programming, the focus is on writing step-by-step procedures or functions that operate on data.

```java
public class ProceduralExample {
    public static double calculateArea(double length, double width) {
        return length * width;
    }

    public static void main(String[] args) {
        double length = 5.0;
        double width = 4.0;
        double area = calculateArea(length, width);
        System.out.println("Area: " + area);
    }
}
```

# Object Oriented Programming

OOP revolves around creating objects that possess attributes and behaviors

```java
public class Rectangle {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return length * width;
    }
}

public class OOPExample {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5.0, 4.0);
        double area = rectangle.getArea();
        System.out.println("Area: " + area);
    }
}
```

# Functional Programming

Functional programming involves writing pure functions without side effects and emphasizes immutability. In Java, we can use lambda expressions and streams

```java
import java.util.function.Predicate;

public class FunctionalExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;

        System.out.println("Is 4 even? " + isEven.test(4));
        System.out.println("Is 7 even? " + isEven.test(7));
    }
}
```

# Understanding Algorithms

An algorithm is a set of instructions designed to perform a specific task. This can be as simple as adding two numbers or as complex as rendering a 3D image.

**Importance**: Algorithms are at the heart of computing and are crucial for solving complex problems in an efficient manner.

**Basic Algorithm Types**

1. **Sorting Algorithms**: Such as Bubble Sort, Merge Sort, and Quick Sort. They arrange data in a particular order.
2. **Search Algorithms:** Like Linear Search and Binary Search. They are used to find specific data within a dataset.
3. **Graph Algorithms**: Including Dijkstra's Algorithm for shortest path problems.
4. **Dynamic Programming**: Used for optimization problems by breaking them down into simpler sub-problems.

# Sorting Algorithms

Sorting algorithms are methods for rearranging a collection of items (such as numbers in an array) into a particular order, typically ascending or descending. These algorithms are fundamental in computer science and are widely used in various applications, from organizing data to searching and algorithm optimization.

**Bubble Sort** is one of the simplest sorting algorithms.
Algorithm:

1. **Compare Adjacent Elements**: It compares adjacent elements in the array.
2. **Swap If Necessary**: If an element is greater than the one next to it (for ascending order), they are swapped.
3. **Repeat**: This process is repeated for each pair of adjacent elements, starting from the beginning of the array.
4. **Iterate Through the Array**: The algorithm goes through the entire array multiple times until no more swaps are needed, indicating that the array is sorted.

# Bubble Sort Algorithm

```java
public class BubbleSortExample {
    public static void main(String[] args) {
        int[] numbers = { 5, 3, 8, 4, 2 };

        bubbleSort(numbers);

        for (int number : numbers) {
            System.out.print(number + " ");
        }
    }

    private static void bubbleSort(int[] array) {
        boolean swapped;
        for (int i = 0; i < array.length - 1; i++) {
            swapped = false;
            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    // Swap array[j] and array[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    swapped = true;
                }
            }
            // If no two elements were swapped in inner loop, then break
            if (!swapped)
                break;
        }
    }
}
```

# Searching Algorithms

Searching algorithms are methods used to find or retrieve an element from a dataset. They are crucial in computer science for efficiently finding data in large collections.

**Binary Search**

- **Basic Concept**: Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed the possible locations to just one.
- **Prerequisite**: The list must be sorted.

# Binary Search Algorithm

```java
public class BinarySearchExample {
    public static void main(String[] args) {
        int[] sortedArray = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
        int target = 11;

        int result = binarySearch(sortedArray, target);

        if (result == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element found at index: " + result);
    }

    private static int binarySearch(int[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Check if target is present at mid
            if (array[mid] == target)
                return mid;

            // If target greater, ignore left half
            if (array[mid] < target)
                left = mid + 1;

            // If target is smaller, ignore right half
            else
                right = mid - 1;
        }

        // Target not present
        return -1;
    }
}
```

# Java Built-in Sorting and Searching

**Arrays.sort()**

- **Functionality**: Arrays.sort() is a method used to sort arrays. It can sort arrays of primitives (like int[], double[], etc.) as well as arrays of objects (like String[], Object[], etc.).
- **Implementation**: For primitive types, it uses a variation of the quicksort algorithm. For objects, it uses a modified mergesort algorithm. These sorting algorithms are efficient and have good performance characteristics.

```java
int[] numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers)); // Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
```

**Arrays.binarySearch()**

- **Functionality**: **Arrays.binarySearch()** is used to perform a binary search on an array. It returns the index of the searched element if it is present in the array; otherwise, it returns a negative number.
- **Prerequisite**: The array must be sorted before calling this method. If the array is not sorted, the results are undefined.

```java
int[] sortedNumbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int index = Arrays.binarySearch(sortedNumbers, 6);
System.out.println(index); // Output: 5 (since 6 is at index 5)
```

# Any questions?

Jansen Ang