# Java Note Summary

# Java Notes Directory

## Lecture 1 – Classes and Objects

---

## Lecture 2 – Classes and Objects Features, Conditionals, Variables

---

## Lecture 3 – Collections and Iteration

    - Set Interface vs Implementations
5. For-each Loop
6. While Loop
7. Autoboxing
8. Choosing Between List and Set

---

## Lecture 4 – Class Invariants, Postconditions, Exceptions, Equality vs Identity

1. Class Invariants
2. Postconditions
3. Exceptions
4. Equality vs Identity
5. Object Class Methods

---

## Lecture 5 – Implementation vs Interface; Visibility; Class Members; Generics; Designing Classes

1. Implementation vs Interface
2. Visibility
3. Class Members
4. Constants
5. static Key Words
6. final Key Words
7. Generics
    - Generic Classes and Methods
    - Type Parameters and Bounds
    - **[NEW] Designing Generic Functions** *(Inserted here as it extends the existing Generics topic to include designing generic functions, such as the* indexof *_function required in Question 2)*
8. Designing Classes
9. Package

---

## Lecture 6 – Testing and Debugging

1. Testing
    - Manual Testing
    - Automated Testing (introduced)
2. Debugging
    - Steps of Debugging
    - Techniques
3. Common Errors

4. Best Practices in Testing and Debugging

---

## Lecture 7 – Interfaces and Function Overloading

1. Interface
   - Implementing an Interface
   - **[NEW] Default Methods in Interfaces** *(Inserted here as it relates to Question 4, which involves a default method in a Java interface)*
2. Function Overloading
3. Why Use Interfaces

---

## Lecture 8 – Inheritance, Abstract Classes, Inheritance-based Polymorphism

1. Inheritance
2. Overriding Methods
3. Abstract Data Type (ADT)
4. Abstract Classes
5. Inheritance-based Polymorphism

---

## Lecture 9 – Anonymous Classes, Functional Interfaces, Streams, Lambda Functions

1. Anonymous Classes
2. Functional Interfaces
   - **@FunctionalInterface Annotation** *(Inserted here as it directly relates to Question 3, which uses the* @FunctionalInterface *annotation)*
3. Lambda Expressions
   - **Lambda Expression Syntax in Java** *(Inserted here as it relates to Question 3, which requires providing a lambda expression for a Java functional interface)*
4. Function<A,B>
5. Streams
6. Combining Streams and Lambdas

---

> 1.Classes and Objects
>
> # 1. Object
>
> **Definition: An object is an instance of a class; it represents a real-world entity in a program.**

### *Features*

- Has **state** (stored in fields / attributes). State refer to the parameters such as `int` , `boolean` , `char` , `String` ...
- Has **behavior** (defined by methods).

### *Example*

- Classes: `Student` , `Book` , `Library`
- Objects: `Alice` (Student), `"Object First with Java"` (Book)

### *Explanation*

- An object maps to a real-world entity with **properties** (state) and **actions** (behavior).

---

## 2. Class

### **Definition**: A class is a **blueprint or template** for creating objects; it defines what states and behaviors its objects will have.

### *Example (Student Class)*

```java
public class Student {
    // Fields
    private String name;
    private int age;

    // Constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Methods
    public void introduce() {
        System.out.println("Hi, I'm " + name + ", age " + age);
    }
}
```

### *Remark*

- In this class, the particular student as an object has two fields: `name` and `age` , initialized via the constructor.

---

## 3. Fields

### **Definition:** Variables that belong to a class; they store the *state of an object*.

### *Example*

```
private String name;
private int age;
```

---

## 4. Constructor

### **Definition:** Special method to **initialize new objects**.

### *Features*

- Same name as the class.
- No return type (not even `void`).

### *Example*

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

### *Remark*

- Every class has at least one constructor. If not explicitly defined, Java provides a **default no-argument constructor**.

---

## 5. Methods

### **Definition:** Define the **behavior** of an object; what it can do.

### *Types of Methods*

- **Accessor (getter):** Returns information without modifying object state.
- **Mutator (setter):** Modifies object state.

### *Example*

```java
public String getName() {
    return name;
} // Accessor

public void setAge(int newAge) {
    age = newAge;
} // Mutator
```

### *Remark*

- Classes usually contain **fields, constructors, and methods**.

---

## 6. Creating an Object

### *Example*

```java
Student student1 = new Student("Alice", 20);
student1.introduce(); // Output: Hi, I'm Alice, age 20
```

### *Explanation*

- Here, `student1` is an object of `Student` class.
- The constructor `Student("Alice", 20)` initializes its fields with the provided values.

---

## 7.Main Method

### Definition:

**The main method is the entry point of a Java program. It is where the program starts running.**

### *Feature*

- Has a **fixed signature**.
- Must be `public static void main(String[] args)`.
- `args` stores **command-line arguments**.

### *Syntax*

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

### *Example*

- Running the program above prints:

```
Hello World
```

### *Explanation*

- `public` : Can be accessed anywhere.
- `static` : Runs without creating an object.
- `void` : Does not return a value.
- `String[] args` : Accepts command-line arguments.

### *Remark*

- Every standalone Java program needs a **main method** to run.
- In Java projects with multiple classes, only one class needs to have the main method as the **entry point**.

---

## 8. "this" and "new" Keywords

### Definition:

- **this:** Refers to the **current object** within its class.
- **new:** Used to **create a new object**.

### *Example*

```java
public void setName(String name) {
    this.name = name; // 'this.name' refers to the field, 'name' is the parameter
}
```

---

## 9.Data Types

### Definition

A **data type** specifies the kind of data a variable can hold and how it is stored.

### *Categories*

- **Primitive Data Types**: built-in types storing simple values.
- **Reference Data Types**: store references to objects.

---

## 9.1. Primitive Data Types

### **Definition**: Basic built-in types that store **simple values directly**.

### *Types in Java*

- `byte` : 8-bit integer ( `-128` to `127` )
- `short` : 16-bit integer ( `-32,768` to `32,767` )
- `int` : 32-bit integer ( `-2^31` to `2^31-1` )
- `long` : 64-bit integer ( `-2^63` to `2^63-1` )
- `float` : 32-bit floating point
- `double` : 64-bit floating point
- `char` : single 16-bit Unicode character
- `boolean` : true or false

### *Example*

```
int age = 20;
double score = 95.5;
char grade = 'A';
boolean passed = true;
```

### *Remark*

- Primitive types are **not objects**.
- They store actual values directly in memory.

---

## 9.2. Reference Data Types

### **Definition**: Types that **store references to objects**, not actual object data.

### *Examples*

- Class types ( `String` , `Scanner` , user-defined classes)
- Arrays
- Interfaces

### *Example*

```java
String name = "Alice";
Student s = new Student("Bob", 18);
```

### *Explanation*

- `name` stores a reference to a String object.
- `s` stores a reference to a Student object.

---

## 9.3. Differences Between Primitive and Reference Types

### *Primitive*

- Stores **actual value**.
- Size is fixed based on type.
- Not objects.

### *Reference*

- Stores **address/reference** to object.
- Size depends on object.
- Can be `null`.

## 9.4.Object Types VS. Reference Types

### *Short Answer*

**In Java, object types and reference types are often used interchangeably, but conceptually:**

- **All object types are reference types**, because variables of these types **store references to objects in memory**, not the actual object itself.
- **Reference type** is a **broader term** that includes:
  - **Class types** (e.g. `String`, `Scanner`, user-defined classes)
  - **Array types** (e.g. `int[]`, `String[]`)
  - **Interface types** (e.g. `Runnable`)

### *Long Answer*

| Term | Meaning | Example |
|---|---|---|
| **Object type** | Usually refers specifically to **class types** (types that are classes). | `Student s = new Student();``String name = "Alice";` |
| **Reference type** | Any type whose variable stores a **reference (address) to data** instead of the data itself. Includes class types, arrays, interfaces. | `String`, `int[]`, `Scanner`, `Runnable` |

### *Explanation*

- Primitive types (`int`, `double`, `char`, `boolean`, etc.) store **actual values directly**.
- Reference types store **references** (addresses/pointers) to objects on the heap.

Therefore:

- **Every object type is a reference type.**
- **Not every reference type is necessarily called an "object type"** in strict terminology, because **arrays and interfaces** are also reference types even though they are not classes (but their instances are objects in memory).

### *Summary*

- **Same in behavior** (both store references).
- **Object type** is a **subset of reference types**.

---

# 10. Object Type

## Definition

### `Object` is the **parent class of all classes** in Java. Every class **inherits from Object**.

## *Feature*

- Can store **any object type**.

## *Syntax*

```
Object variableName;
```

## *Example*

```
Object x = "Hello";
Object y = 10;
```

### *Explanation*

- `x` stores a **String** as an Object.
- `y` stores an **int** as an Object.
  (Because Java changes `int` to `Integer` automatically. I will introduce it in Autoboxing)

### *Remark*

- **Primitive types (int, char)** are not objects.
  But Java **converts them** to objects when needed (called **Autoboxing**).

---

# 11.Java Virtual Machine (JVM)

## Definition

### JVM is the part of Java system that runs Java programs by converting bytecode to machine code for the current platform.

### *Feature*

- Provides platform independence for Java programs.
- Performs Just-In-Time (JIT) compilation to improve performance.
- Manages memory automatically with garbage collection.

### *Syntax*

*No specific syntax; JVM runs the compiled `.class` files.*

### *Example*

- Compile a program:

```
javac HelloWorld.java
```

- Run with JVM:

```
java HelloWorld
```

### *Explanation*

- The Java compiler ( `javac` ) creates bytecode.
- JVM loads and executes this bytecode on the local machine.

- Because all JVMs understand bytecode, Java programs can run on any device with a JVM.

### *Remark*

- JVM is different from JDK (development kit) and JRE (runtime environment). JVM is part of JRE.

---

## Reference "Objects First with Java"

- **Chapter 1:** Objects and classes overview.
- **Chapter 2:** Class definitions, fields, methods, constructors.

---

2.Variables, Scope, and Preconditions

# 1. Class Features

## **Definition**: The main **components of a class** that define its structure and behavior.

### *Features*

- **Fields**: store object state.
- **Constructors**: initialize new objects.
- **Methods**: define object behavior.
- **Encapsulation**: restricting access using **access modifiers (as example)**.

### *Example of Access Modifier*

- `private` : Accessible within the class only.
- `public` : Accessible from any other class.
- `protected` : Accessible within the package and subclasses.

### *Explanation*

- Encapsulation hides internal data and exposes only necessary parts through methods.

---

# 2. Conditional Composition (if, if-else)

## **Definition**: Used to perform *different actions based on boolean conditions*.

### *Features*

- Uses boolean expressions that evaluate to `true` or `false`.
- Can be **nested** to check multiple conditions.

### *Example*

```java
if (age >= 18) {
    System.out.println("Adult");
} else {
    System.out.println("Minor");
}
```

### *Explanation*

- If `age >= 18` is true, prints "Adult"; otherwise prints "Minor".

### *Notation: "||", "&&"*

- "||" refer to "or"; "&&" refer to "and".

---

## 3. Variables

### **Definition:** Variables are **named storage locations** for data that a program uses.

### *Features*

- Must be **declared** with a type before use.
- Can be initialized with a value.

### *Example*

```java
int age = 20;
String name = "Alice";
```

### *Explanation*

- `int age` declares an integer variable.
- `String name` declares a string variable.

---

## 3.1. Instance Variables

### Definition:

Variables declared **inside a class but outside any method or constructor**. Also called **fields**.

### *Features*

- Each object (instance) has its **own separate copy**.
- Initialized to default values if not explicitly set.

### *Example*

```java
class Student {
    String name; // instance variable
    int age;     // instance variable

    void printInfo() {
        System.out.println(name + " is " + age + " years old.");
    }
}
```

### *Explanation*

- `name` and `age` are instance variables.
- Each `Student` object will have its own `name` and `age`.

### *Remark*

- Their lifetime is tied to the **object's lifetime**.

---

## 3.2. Class (Static) Variables

### Definition: Variables declared with the `static` keyword inside a class but outside any method or constructor.

### *Features*

- **Shared among all instances** of the class.
- Only **one copy** exists regardless of the number of objects created.

### *Example*

```java
class Student {
    static int count = 0; // static (class) variable

    Student() {
        count++;
    }
}
```

### *Explanation*

- `count` keeps track of the total number of `Student` objects created.
- All `Student` objects **share the same** `count` **variable**.
- **I will introduce the meaning of** `static` **key word in future.**

### *Remark*

- Accessed using `ClassName.variableName` syntax (e.g. `Student.count` ).

---

## 3.3. Local Variables

## **Definition:** Variables declared **inside methods, constructors, or blocks.**

### *Features*

- Only accessible **within the method or block** where declared.
- **Must be initialized** before use.

### *Example*

```java
void printSum() {
    int a = 5; // local variable
    int b = 10; // local variable
    System.out.println(a + b);
}
```

### *Explanation*

- `a` and `b` exist only while `printSum()` is executing.
- They are destroyed once the method finishes.

### *Remark*

- Local variables do **not have default values**; uninitialized use causes compilation error.

---

### *Summary Remark*

- **Instance variables**: per object (fields)
- **Static (class) variables**: per class (shared)
- **Local variables**: per method/block (temporary)

---

## 4. `double` Type

## Definition

**`double` is a primitive data type in Java used to store decimal numbers (floating-point numbers).**

### *Feature*

- Stores **numbers with decimals**.
- **Default type** for decimal numbers in Java.
- Uses **8 bytes** of memory.

### *Syntax*

```
double variableName = value;
```

### *Example*

```
double price = 19.99;
double pi = 3.14159;
```

### *Explanation*

- `price` stores **19.99** as a double.
- `pi` stores **3.14159** as a double.
- `double` is used when you need **more precision** than `int`.

### *Remark*

- For **whole numbers**, use `int`.
- For **decimal numbers**, use `double`.

---

## 5. Scope and Lifetime

### **Definition**: Scope determines where a variable can be **accessed**, while lifetime determines how long it **exists** in memory.

### *Types*

- **Class-level (fields)**: exist as long as the object exists (defined by fields).
- **Method-level (local variables)**: exist during method execution.
- **Block-level**: exist only within the block `{}` they are declared.

### *Example*

```
public void exampleMethod() {
    int x = 5; // x exists only within this method
    if (x > 0) {
        int y = 10; // y exists only within this if block
    }
}
```

### *Explanation*

- `x` is method-level, `y` is block-level.

---

## 6. Preconditions

### Definition

**Conditions that must be true before a method executes to ensure correct behavior.**

### *Features*

- Often implemented using `assert` statements.
- Protect methods from invalid inputs.

### *Example*

```
public void setAge(int age) {
    assert age >= 0;//If age < 0, the program will throw an assertion error.
    this.age = age;
}
```

### *Explanation*

- Ensures `age` is not negative before setting it.

---

## 7. Object Interaction

### Definition

**How objects communicate and interact with each other in a program.**

### *Features*

- Done by calling methods on objects.
- Parameters can be passed **by value or by reference**.

### *Example*

```java
Student student1 = new Student("Alice", 20);
student1.setAge(21);
```

### *Explanation*

- `student1.setAge(21)` calls the `setAge` method to update `student1` 's age.

---

## 8.Null vs .isEmpty()

### Definition

**`null` means a reference does not point to any object. `.isEmpty()` checks if an object like a String or collection has no content.**

### *Feature*

- `null` is not an object; calling methods on `null` causes `NullPointerException` .
- `.isEmpty()` is a method used on objects to check if they have no data (e.g., empty string or empty list).

### *Syntax*

```java
String s = null;
if (s == null) {
    System.out.println("s is null");
}
```

```java
String t = "";
if (t.isEmpty()) {
    System.out.println("t is empty");
}
```

### *Example*

- `null` example:

```java
String name = null;
if (name == null) {
    System.out.println("Name is not assigned");
}
```

- `.isEmpty()` example:

```java
String word = "";
if (word.isEmpty()) {
    System.out.println("Word has no characters");
}
```

### *Explanation*

- Use `== null` to check if a reference points to no object.
- Use `.isEmpty()` to check if an existing object has zero length or size.

### *Remark*

- Calling `.isEmpty()` on `null` will throw `NullPointerException`.
- Always check for `null` before calling `.isEmpty()` if a variable might be `null`.

---

## Reference "Objects First with Java"

- **Chapter 2:** Class features, conditionals, variables, scope, preconditions, object interaction.

---

3.Control flow -- Collection and Iteration

# 1.Java Collections

## Definition: A *Collection* is a **group of elements (objects)** treated as a single unit. It provides a way to **store, retrieve,**

**and manage multiple data items efficiently.**

### *Features*

- Stores **multiple elements** in one variable.
- Provides **methods** to add, remove, and check elements.
- Helps organize data logically, like a real-life collection (e.g. a list of books).

### *Example*

- Java provides different types of collections to suit different needs:
  - **List**: Ordered collection allowing duplicates.
  - **Set**: No duplicates, elements are unique.
  - **Map** (not technically a Collection, but part of the Collection Framework): Stores key-value pairs.

### *Note on Generics*:

You may see:

```
ArrayList<String> list = new ArrayList<>();
```

- The `<String>` part is called **Generics**, which specifies **what type of elements** the collection stores.
- You will learn Generics formally in Lecture 5. For now, understand it indicates **"this list stores String objects."**

### *Remark*

- Java Collections are designed to store **objects**, not primitive types (e.g. `int`, `char`).

```
Collection<Interger> hello = new ArrayList<>();//Correct
```

```
Collection<int> hello = new ArrayList<>();//Incorrect
```

- `Interger` is the wrapper class of `int`. I will introduce it later.

### *Explanation*

- Java collections are built on **Object references**.
- Primitive types are **not objects**, so they cannot be directly stored.

---

## 2. Array

## Definition:

### An **array** is a **fixed-size collection** that stores multiple values of the **same type** in order.

### *Feature*

- Stores **multiple values**.
- Each value has an **index** starting from **0**.
- **Size cannot change** after creation.

### *Syntax*

```
type[] arrayName = new type[size];
```

### *Example*

```
int[] numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

### *Explanation*

- `int[] numbers = new int[3];` creates an **array of 3 integers**.
- `numbers[0] = 10;` sets the **first element** to 10.
- You can access values using their **index**.

### *Remark*

- Arrays have **fixed size**.
- For resizable collections, use **ArrayList** instead.

---

## 2. Collections – List

### Definition: A **List** is an **ordered collection** that allows duplicate elements.

### *Features*

- Elements have **index positions**.
- Maintains **insertion order**.
- Allows **duplicate elements**.

### *Explanation*

- Lists model sequences where order matters and duplicates are allowed.

### *Remark*

- List is implemented by array.

---

## 2.1. ArrayList

## Definition: A resizable array implementation of List.

### *Features*

- Fast **access** using indices.
- Slower insertions/deletions in the middle (due to shifting elements).
- Uses **autoboxing (introduce later)** to store primitive types as wrapper objects.

### *Example*

```java
import java.util.ArrayList;

ArrayList<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
System.out.println(names.get(0)); // Output: Alice
```

```java
import java.util.ArrayList;

ArrayList list = new ArrayList();
list.add("apple");
list.add("banana");
list.add("cherry");

System.out.println(list); // Output: [apple, banana, cherry]
```

### *Major Methods*

**( `E` refer to `Element` , I would introduce it in Placeholder)**

- `.add(E e)` : add to end
- `.add(int index, E e)` : insert at index
- `.get(int index)` : get element
- `.set(int index, E e)` : replace element
- `.remove(int index)` : remove by index
- `.size()` : number of elements

- `.contains(Object o)` : check existence (Object Type)

### *Remark*

- Best when **frequent random access** is needed.

---

## 2.2. LinkedList

## Definition: **A doubly linked list implementation** of List.

### *Features*

- Faster insertions/deletions at ends or middle (no shifting needed).
- Slower random access (O(n), as traversal is needed).

### *Example*

```java
import java.util.LinkedList;

LinkedList<String> queue = new LinkedList<>();
queue.add("first");
queue.add("second");
queue.addFirst("zero");
System.out.println(queue.get(0)); // Output: zero
```

### *Major Methods*

- Same as ArrayList, because both implement List interface.
- **Additional methods**:
  - `.addFirst(E e)` : add to front
  - `.addLast(E e)` : add to end (E is a)
  - `.removeFirst()` : remove first
  - `.removeLast()` : remove last

### *Remark*

- Best for **frequent insertions/deletions** rather than random access.

---

## 2.3. List Interface vs Implementations

### *Remark*

- **List** is an interface.

- **ArrayList**: fast random access.
- **LinkedList**: fast insertion/deletion.
- Choose based on **use case**.

---

## 3. Collections – Set

**Definition**: A *Set* is a collection that stores *unique elements*, with no duplicates.

### *Features*

- No index positions.
- Unordered or ordered depending on implementation.

### *Explanation*

- Sets model collections of unique items where order is not important (unless using TreeSet).

---

## 3.1. HashSet

**Definition**: Implements Set using a **hash table**.

### *Features*

- No guaranteed order.
- Fast add, remove, contains operations (average O(1)).

### *Example*

```java
import java.util.HashSet;

HashSet<Integer> numbers = new HashSet<>();
numbers.add(1);
numbers.add(2);
numbers.add(1); // duplicate ignored
System.out.println(numbers); // Output: [1, 2]
```

### *Major Methods*

- `.add(E e)` : adds element if not present
- `.remove(Object o)` : removes element if present
- `.contains(Object o)` : checks membership
- `.size()` : number of elements

- `.clear()` : removes all elements

### *Remark*

- Best for **fast membership testing** and removing duplicates.

---

## 3.2. TreeSet

### **Definition**: Implements Set using a **tree structure** (red-black tree).

### *Features*

- Elements are sorted in **natural order** or by comparator.
- Operations are O(log n).

### *Example*

```java
import java.util.TreeSet;

TreeSet<Integer> sortedSet = new TreeSet<>();
sortedSet.add(3);
sortedSet.add(1);
sortedSet.add(2);
System.out.println(sortedSet); // Output: [1, 2, 3]
```

### *Major Methods*

- Same as HashSet.
- Additional ordering methods:
    - `.first()` : smallest element
    - `.last()` : largest element
    - `.higher(E e)` : least element greater than e
    - `.lower(E e)` : greatest element less than e

### *Remark*

- Best when **sorted set** is required.

---

## 3.3. Set Interface vs Implementations

### *Remark*

- **Set** is an interface.
- **HashSet**: fast, unordered.
- **TreeSet**: ordered, slower.
- Choose based on **ordering requirement**.

---

## 3.4. Choosing Between List and Set

### *Remarks*

- **List**
  - Ordered.
  - Allows duplicates.
  - Index-based access.
- **Set**
  - Unordered or ordered (TreeSet).
  - No duplicates.
  - No index access.

---

## 4. For-each Loop

### **Definition**: Simplified loop for **iterating over arrays or collections.**

### *Example*

```
for (String name : names) {
    System.out.println(name);
}
```

### *Explanation*

- Iterates through each element in `names` .

---

## 5. While Loop

### **Definition**: Loop that executes **while condition is true.**

### *Example*

```
int i = 0;
while (i < names.size()) {
    System.out.println(names.get(i));
    i++;
}
```

### *Explanation*

- Prints all elements in `names` using index `i`.

---

## 6. Wrapper Classes

### **Definition**: Classes that **wrap (encapsulate)** primitive data types as **objects.**

### *Syntax*

| Primitive Type | Wrapper Class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

### *Example*

```
int a = 5;
Integer b = new Integer(a); // explicit boxing
Integer c = a;              // autoboxing

int d = c.intValue();       // explicit unboxing
int e = c;                  // unboxing
```

### *Explanation*

- Wrapper classes provide:
  - **Object representation** of primitive types.
  - **Utility methods** (see *Object Class Methods*)
  - Use in **Collections** that require objects.
- For example, `Integer` wraps `int`, allowing it to be stored in `ArrayList<Integer>`.

### *Remark*

- Prefer **autoboxing/unboxing** for readability, but be aware of **null pointer exceptions** when unboxing null values.
- Using wrapper classes allows primitives to be treated as objects and stored in collections.

---

## 7. Autoboxing and Unboxing

### Definition:

- **Autoboxing**: Automatic conversion of **primitive types** (e.g. `int`) to their **wrapper class objects** (e.g. `Integer`).
- **Unboxing**: Automatic conversion of **wrapper class objects** back to **primitive types**.

### *Syntax*

```
Integer x = 5; // autoboxing: int 5 -> Integer object
int y = x;     // unboxing: Integer x -> int y
```

### *Example*

```
import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>();
list.add(10); // autoboxing: int 10 -> Integer object

int num = list.get(0); // unboxing: Integer -> int

System.out.println(num + 5); // Output: 15
```

### *Explanation*

- Java automatically converts between primitives and wrapper classes when needed.
- Useful when working with collections (e.g. `ArrayList`) that only store **objects**, not primitives.

### *Remark*

- **Autoboxing** and **unboxing** were introduced in **Java 5**.
- Frequent boxing/unboxing can impact **performance**, especially in loops.

---

## Reference "Objects First with Java"

- **Chapter 4**: Collections (List: ArrayList, LinkedList; Set: HashSet, TreeSet)
- **Chapter 5**: Iteration (for-each, while loops)

- **Appendix**: Java Collections Framework overview

---

4.Class Design -- Invariants, Postconditions, Exceptions

# 1.Class Invariants

### **Definition:** A **class invariant** is a condition that must always be true for all instances of a class after construction and between method calls.

### *Syntax*

```java
class Person {
    private int age; // invariant: age >= 0

    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}
```

### *Example*

- The invariant is that `age` should always be >= 0.

### *Explanation*

- Used to ensure **object consistency** throughout its lifetime.

### *Remark*

- Often documented or enforced via checks (assertions) but Java does not have built-in invariant syntax.

---

## 2. Postconditions

### **Definition:** A **postcondition** is a condition that must be true after a method finishes execution.

### *Syntax*

```java
public int increment(int x) {
    return x + 1; // postcondition: result == old x + 1
}
```

### *Example*

- After `increment` returns, the result is guaranteed to be input + 1.

### *Explanation*

- Ensures methods perform as specified.

### *Remark*

- Together with preconditions and invariants, postconditions are part of **Design by Contract**.

---

## 3. Exceptions

### **Definition**: An **exception** is an event that disrupts the normal flow of a program, used to signal errors or unusual conditions.

### *Syntax*

```java
// Option 1: Using try-catch to handle an exception
try {
    // code that may throw an exception
} catch (ExceptionType name) {
    // handling code
}
```

```java
// Option 2: Using throw to explicitly throw an exception
throw new ExceptionType("Error message");
```

### *Example*

- **Using try-catch** to handle division by zero:

```java
try {
    int result = 10 / 0;
    System.out.println(result);
} catch (ArithmeticException e) {
    System.out.println("Division by zero error");
}
//Output: Division by zero error
```

- **Using throw** to explicitly throw an exception:

```java
int number = -1;
if (number < 0) {
    throw new IllegalArgumentException("Number cannot be negative");
}
//Output: Exception in thread "main" java.lang.IllegalArgumentException: Number cannot
be negative at YourClassName.main(YourClassName.java:LineNumber)
```

### *Explanation*

- `try-catch` is used to **catch and handle** exceptions when they happen.
- `throw` is used to **explicitly create and throw an exception** if a certain condition is not acceptable.
- They are **two different ways** to deal with errors:
  - **try-catch** handles errors that occur in code.
  - **throw** creates and signals an error by yourself.

### *Remark*

- Always choose meaningful exception types (e.g. `IllegalArgumentException` for invalid arguments).
- Exceptions help make your programs **more robust and predictable**.

---

## 4. Equality vs Identity

### Definition:

- **Equality ( `equals` ) checks if two objects have the same content.**
- **Identity ( `==` ) checks if two references point to the same object in memory.**

### *Syntax*

```java
String a = new String("hello");
String b = new String("hello");

System.out.println(a == b); // false
System.out.println(a.equals(b)); // true
```

### *Example*

- `a == b` is false (different objects).
- `a.equals(b)` is true (same content).

### *Explanation*

- `==` : compares memory addresses (identity).
- `equals()` : compares values (equality), unless not overridden.

### *Remark*

- For objects, use `equals()` to compare content; use `==` for reference comparison.

---

## 5. Object Class Methods (Simplified)

### **Definition**: The **Object class** defines several general methods available to all Java objects. Two common methods are:

1. `toString()`
2. `equals(Object obj)`

### *Syntax*

```
object.toString();
object.equals(otherObject);
```

### *Example*

```
String a = "Java";
String b = "Java";

System.out.println(a.equals(b)); // prints true

System.out.println(a.toString()); // prints "Java"
```

### *Explanation*

- `equals(Object obj)`
  - Returns a **boolean value**: `true` or `false` .
  - It checks whether **the contents of two objects are the same**.

    - In this example, `a.equals(b)` is `true` because both `a` and `b` store the same string `"Java"`.
    - When we write:

    ```java
    System.out.println(a.equals(b));
    ```

    - `a.equals(b)` evaluates to `true`, then `System.out.println()` prints **true** to the console.
- `toString()`
    - Returns a **string representation** of the object.
    - For a `String` object, calling `toString()` simply returns itself.

### *Remark*

- You can think of `equals()` **as checking if two boxes have the same thing inside**.
- The output of `System.out.println(a.equals(b));` is `true` because `a.equals(b)` returns `true`, and `println` prints whatever is inside the parentheses.

---

5. Implementation, Interface, Visibility, Generics

# 1. Implementation vs Interface

## Definition:

### **Interface (in design)**: The **set of methods or functionality** exposed to the user (what it does).

### **Implementation**: The **code that actually performs the tasks** (how it does it).

### *Feature*

- Interface hides details and shows only what is necessary.
- Implementation can change **without affecting users**, as long as the interface stays the same.

### *Syntax*

*No specific Java syntax here; it is a **design concept**.*

### *Example*

```java
List<Integer> nums = new ArrayList<>();
```

- `List` is the **interface**.
- `ArrayList` is the **implementation**.

### *Explanation*

- You program to the **interface (`List`)** rather than the implementation (`ArrayList`), so you can later switch to another implementation (e.g. `LinkedList`) without changing other code.

### *Remark*

- In **Lecture 5**, interface refers to **software design interface**, **not Java's `interface` keyword**.

---

## 2. Visibility

### Definition

#### Controls where class members (fields, methods) can be accessed.

### *Types*

- **public**: Accessible from **anywhere**.
- **private**: Accessible **only within the same class**.
- **protected:** visible in class and subclasses.

### *Example*

```java
public class Student {
    private String name; // only within Student class
    public void setName(String newName) { // accessible anywhere
        name = newName;
    }
}
```

```java
public int x;     // visible to all
private int y;    // visible only within the class
protected int z; // visible in class and subclasses
int w;            // package-private, default
```

### *Explanation*

- `name` is **private**, so it can't be accessed directly outside Student.
- `setName` is **public**, allowing other classes to change name safely.

### *Remark*

- Default visibility (no modifier) means **package-private**.

---

## 3. Class Members

### **Definition**: Fields and methods that belong to a class.

### *Types*:

- **Fields**: Store data (state).
- **Methods**: Perform actions (behavior).

### *Feature*

- Includes **instance variables**, **class (static) variables**, **methods**, and **nested classes**.

### *Example*

```
public class Book {
    private String title; // field
    public void setTitle(String t) { // method
        title = t;
    }
}
```

---

## 4. Constants

### Definition:

#### A **constant** is a variable whose value **cannot be changed** once it is given a value.

### *Features*

- Declared using the `final` **keyword**.
- Usually combined with `static` if it is shared by all objects.
- Naming convention: use **ALL_CAPITAL_LETTERS** with underscores for clarity.

### *Syntax*

```
final int MAX_STUDENTS = 30;
```

- `final` makes it **unchangeable**.

**With static (class constant)**

```
static final int MAX_SCORE = 100;
```

- `static final` makes it a **constant shared by all objects** of the class.

### *Example*

```
class Classroom {
    static final int MAX_STUDENTS = 30;

    void printMax() {
        System.out.println("Max students allowed: " + MAX_STUDENTS);
    }
}
```

### *Explanation*

- `MAX_STUDENTS` is a **constant**.
- It is `static`, so **every Classroom uses the same value**.
- It is `final`, so **the value can never change**.

### *Remark*

- Constants make programs **clearer** because their values do not change.
- Always use **meaningful names** in **ALL_CAPITAL_LETTERS** to show they are constants.

---

## 5. `static` Keyword

### Definition:

**The `static` keyword makes a variable or method belong to the class itself rather than to any specific object (instance) of the class.**

### *Features*

- **Shared across all objects** of the class.
- Loaded into memory **once when the class is loaded**, not each time an object is created.
- Used for **class-level data** or **utility methods**.

### *Syntax*

```
class Counter {
    static int count = 0; // static variable

    static void showCount() { // static method
        System.out.println(count);
    }
}
```

## Example

```
Counter.count = 5; // Access static variable without object

Counter c1 = new Counter();
Counter c2 = new Counter();

c1.count++;
c2.count++;
System.out.println(Counter.count); // Output: 7
```

## Counterexample

```
class Student {
    String name;

    void sayHello() {
        System.out.println("Hello, my name is " + name);
    }
}
```

**`sayHello` is not a static method because**

- It **does not use the `static` keyword** in its declaration.
- Inside the method, it uses the instance variable `name`.

So, `sayHello` is a instance method.

**If you tried to make it static…**

```
static void sayHello() {
   System.out.println("Hello, my name is " + name); // Error!
}
```

**Compilation Error**: Cannot use non-static variable `name` in a static context.

## Explanation

- `count` is **shared by all Counter objects**.
- Incrementing `count` through any object affects the **same shared variable**.

## Static vs Normal (Instance)

| Aspect | Static Member | Normal (Instance) Member |
|---|---|---|
| **Belongs to** | Class itself | Specific object |
| **Memory** | Loaded once per class | Loaded separately for each object |
| **Access** | ClassName.member or object.member | Only through object |
| **Example** | `Counter.count` | `c1.name` |

### *Remark*

- Use **static** when a value or method is **common to all objects**, e.g. a counter tracking the number of created objects.
- Static methods **cannot directly use instance variables** because they don't belong to any object.

---

## 6. final Keyword

## Definition:

### The `final` keyword means "cannot be changed later."

### *Features*

- **For variables**: Once you give it a value, you **cannot change it**.
- **For methods**: No one can **rewrite** (override) it in child classes (as example).
- **For classes**: No one can **make a subclass** (as example) from it.

### 6.1. final variable

```java
final int MAX = 100;
MAX = 200; // ❌ Error: cannot change final variable
```

**Explanation**:
`MAX` is fixed at 100 forever.

---

### 6.2. final method

```java
class Parent {
    final void show() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    // void show() { } // Error: cannot override final method
}
```

**Explanation**:
Child class cannot change the `show()` method.

---

### 6.3. final class

```
final class Animal { }

// class Dog extends Animal { } // Error: cannot extend final class
```

**Explanation**:
No class can extend `Animal`.

---

### *Remark*

- **final + variable** → value **locked forever**.
- **final + method** → method **cannot be changed** in child classes.
- **final + class** → class **cannot be inherited**.

---

# 7. Generics

## Definition: A **Java feature** that allows classes, interfaces, and methods to operate on **types specified as parameters.**

### *Features*

- Enables **type-safe code** by checking types at compile time.
- Eliminates need for **explicit casting**.
- Uses **placeholders (type parameters)** such as `T`, `E`, `K`, `V`.
- Increases **code reusability** and readability.

### *Example*

```
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0); // No casting needed
```

Without generics:

```java
ArrayList list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0); // Requires casting
```

### *Major Syntax*

- **Generic class**:

  ```java
  class Box<T> {
      T value;
      void set(T value) { this.value = value; }
      T get() { return value; }
  }
  ```

- **Generic method**:

  ```java
  public <T> void printArray(T[] array) {
      for (T item : array) {
          System.out.println(item);
      }
  }
  ```

### *Limitations*

- Cannot use **primitive types** directly (e.g. `ArrayList<int>` not allowed, use `Integer`).
- **Type erasure**: type parameters are removed at runtime.

### *Remark*

- Generics make code **more robust** and **less error-prone**.
- Widely used in Java Collections Framework (`List<E>`, `Map<K,V>`, etc).

---

## 7.1. Placeholder (Generic Type Parameter)

### **Definition**: A **symbolic type name** used in generics to represent an unspecified type.

### *Features*

- Makes classes and methods **type-independent** and reusable.
- Common placeholders:
  - `T` : Type
  - `E` : Element
  - `K` : Key

  - `V` : Value
  - `N` : Number
- Replaced by **actual types** during compilation.

### *Example*

```
public class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

Box<Integer> intBox = new Box<>();
intBox.set(123);
System.out.println(intBox.get()); // Output: 123
```

### *Major Use Cases*

- **Generic classes**: e.g. `ArrayList<E>`, `HashMap<K,V>`.
- **Generic methods**: specify type parameter in method signature.

### *Remark*

- Improves **type safety** and reduces casting.
- Placeholder names have **no fixed meaning** but follow conventions.

---

## 7.2. Designing Generic Functions in Java

### Definition

**Designing generic functions in Java involves creating methods that operate on parameters of different types using generics, ensuring type safety at compile time while allowing flexibility across data types.**

### *Feature*

- Generics allow functions to work with any type specified at compile time.
- Type parameters (e.g., `<T>`) define placeholders for types.
- Constraints like extends can restrict types to specific classes or interfaces.

### Syntax

```java
public <T> int indexOf(T[] array, T element) {
    for (int i = 0; i < array.length; i++) {
        if (array[i].equals(element)) {
            return i;
        }
    }
    return -1;
}
```

### Example

```java
public class GenericSearch {
    public static <T> int indexOf(T[] array, T element) {
        for (int i = 0; i < array.length; i++) {
            if (array[i].equals(element)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Integer[] numbers = {10, 20, 30, 40};
        String[] words = {"apple", "banana", "cherry"};
        System.out.println(indexOf(numbers, 30)); // Outputs 2
        System.out.println(indexOf(words, "banana")); // Outputs 1
        System.out.println(indexOf(numbers, 50)); // Outputs -1
    }
}
```

### Explanation

- The `<T>` before the return type declares a generic type parameter.
- The function indexOf works with any array type T[] and element of type T, using equals for comparison.
- Compile-time type checking ensures that the array and element types match, preventing runtime errors.
- The method returns the first index where the element is found or -1 if absent.

### Remark

- Generics improve code reusability and type safety compared to using Object.
- Limitations include inability to use primitives directly (use wrapper classes like Integer).
- Use bounds (e.g., `<T extends Comparable<T>>`) for additional type constraints if needed.

# 8. Designing Classes

## Steps:

1. **Identify state** → fields.
2. **Identify behaviors** → methods.
3. Decide visibility (**private fields**, **public methods**).
4. Write constructor(s) to initialize fields.
5. Maintain **class invariants** (conditions that must always be true).

---

### *Example*

```java
public class BankAccount {
    private double balance; // state

    public BankAccount(double initialBalance) { // constructor
        balance = initialBalance;
    }

    public void deposit(double amount) { // behavior
        balance += amount;
    }

    public double getBalance() { // behavior
        return balance;
    }
}
```

### *Explanation*

- **Fields**: balance stores account state.
- **Constructor**: initializes balance.
- **Methods**: deposit and getBalance modify or return state.
- Fields are **private**, methods are **public** to control access.

---

# 9.Package

## Definition

### A package is a namespace that organizes classes and interfaces in Java.

### *Feature*

- Helps avoid name conflicts.
- Makes code easier to maintain by grouping related classes.

## *Syntax*

```java
package mypackage;

public class MyClass {
    // class code
}
```

- Using classes from a package:

```java
import mypackage.MyClass;
```

## *Example*

```java
package animals;

public class Dog {
    public void bark() {
        System.out.println("Woof");
    }
}
```

```java
import animals.Dog;

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.bark();
    }
}
```

## *Another Java Example*

```java
// File: vehicles/Car.java
package vehicles;

public class Car {
    public void drive() {
        System.out.println("Driving a car");
    }
}
```

```java
// File: vehicles/Bike.java
package vehicles;

public class Bike {
    public void ride() {
        System.out.println("Riding a bike");
    }
}
```

```java
// File: TestVehicles.java
import vehicles.Car;
import vehicles.Bike;

public class TestVehicles {
    public static void main(String[] args) {
        Car c = new Car();
        c.drive();

        Bike b = new Bike();
        b.ride();
    }
}
```

- Both `Car` and `Bike` are in the `vehicles` package.
- `TestVehicles` imports both classes and uses their methods.
- This shows how multiple related classes are organized in one package.

### *Explanation*

- The `package` statement must be the first line in the source file (except comments).
- `import` is used to access classes from other packages.

### *Remark*

- Packages can be stored in folders matching their names.

---

## Reference: "Objects First with Java"

- **Chapter 6**: Implementation vs interface, visibility, class members, constants, generics, designing classes.

---

6. Testing and Debugging

## 1. Testing

### Definition

**The process of checking whether your code works correctly by trying different inputs and verifying the output.**

### *Features*:

- Detects **bugs** (errors in program logic).
- Helps ensure **correctness** before using or submitting code.
- Should test **normal cases**, **edge cases**, and **invalid inputs** if applicable.

---

### *Example*

```java
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}

// Testing add method
Calculator c = new Calculator();
System.out.println(c.add(2, 3)); // Expected: 5
System.out.println(c.add(-1, 1)); // Expected: 0
System.out.println(c.add(0, 0)); // Expected: 0
```

### *Explanation*

- Tests different inputs to check if `add` returns correct results.

---

## 2. Types of Testing

## 2.1 Manual Testing

### Definition:

Testing your code by **running it yourself** and **looking at the results**.

### *Example*

- Use `System.out.println()` to print results and check if they are correct.

```java
public class Test {
    public static void main(String[] args) {
        int sum = add(2, 3);
        System.out.println(sum); // Should print 5
    }

    public static int add(int x, int y) {
        return x + y;
    }
}
```

---

## 2.2 Automated Testing

### Definition:

Writing **extra code** to test your program **automatically**.

### *Remark*

- You will learn this later when studying **JUnit**.
- For now, focus on **manual testing**.

---

## 2.3.Debugging

### Definition:

#### Debugging is **finding and fixing mistakes (bugs)** in your code.

### *Feature*

- Helps understand **why your code does not work**
- Makes your program **correct**

### *Example*

- Using print statements:

```java
public class DebugExample {
    public static void main(String[] args) {
        int x = 5;
        int y = 0;
        int z = x / y;
        System.out.println("z = " + z);
    }
}
```

- This causes an error (division by zero). To debug:

```java
public class DebugExample {
    public static void main(String[] args) {
        int x = 5;
        int y = 0;
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        // check before dividing
        if (y != 0) {
            int z = x / y;
            System.out.println("z = " + z);
        } else {
            System.out.println("Cannot divide by zero");
        }
    }
}
```

### *Explanation*

- Print statements help you see **variable values** to find what causes the error.

### *Remark*

- Always check **your variables** before using them in calculations.

---

## 3.Common Errors

## Definition:

### Common mistakes that cause bugs or program crashes.

### *Feature*

- Syntax Errors (wrong code grammar)
- Runtime Errors (crash while running)
- Logic Errors (wrong output without crash)

### *Syntax*

```
// Example of syntax error
int x = ; // missing value
```

### *Example*

- **Syntax Error**:

```
System.out.println("Hello") // missing semicolon
```

- **Runtime Error**:

```
int[] arr = new int[2];
System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
```

- **Logic Error**:

```
int sum = a - b; // should be a + b
```

### *Explanation*

- Syntax errors are caught by compiler.
- Runtime errors are caught when you run the code.
- Logic errors are hardest to find because program still runs.

### *Remark*

- Carefully read **compiler error messages** for syntax errors.
- Use **debugging techniques** to find runtime and logic errors.

---

## 6.4 Best Practices in Testing and Debugging

## Definition:

### Ways to test and debug your code **effectively**.

### *Feature*

- Test **small parts** of code first
- Use **simple print statements** to check values
- Check **edge cases** like 0 or negative numbers

### *Example*

- Testing method with different inputs:

```java
public static int add(int x, int y) {
    return x + y;
}
```

- Checking:

```java
System.out.println(add(2, 3)); // 5
System.out.println(add(0, 0)); // 0
System.out.println(add(-2, -3)); // -5
```

### *Explanation*

- Testing with **different inputs** makes sure your method works in all situations.

### *Remark*

- Always test **step by step** to find errors quickly.

## Reference: “Objects First with Java”

- **Chapter 7**: Testing and debugging programs to ensure correctness.

7. Interfaces and Overloading

# 1. Interface

## **Definition**: An **abstract type** in Java that specifies method signatures but **does not provide implementation.**

### ***Features***:

- Defines **what a class should do**, not **how to do it**.
- A class **implements** an interface to provide method bodies.
- Supports **multiple inheritance of type**, meaning a class can implement multiple interfaces.

### *Example*

```
public interface Animal {
    void makeSound(); // no body, only signature
}
```

### *Explanation*

- `Animal` interface defines a method `makeSound()` that any implementing class must write.

## 7.1. Implementing an Interface

### *Example*

```
interface Animal {
    void makeSound(); // interface
}

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

class Duck implements Animal {
    public void makeSound() { // implementation
```

```
            System.out.println("Ga");
        }
    }
```

### *Explanation*

- `Dog` and `Duck` class **implements** `Animal` interface and provides the body for `makeSound()`.

### *Remark*

- A class can **implement multiple interfaces** (e.g., `implements A, B`) while it can only **extend one class** (inheritance, discussed in Lecture 8).
- Interfaces help achieve **polymorphism**, allowing different classes to be treated the same way if they implement the same interface (polymorphism discussed in Lecture 8).

---

## 7.2. Default Methods in Java Interfaces

### Definition

**Default methods in Java interfaces provide a default implementation for methods, allowing interfaces to evolve without breaking existing implementations. They are defined using the default keyword.**

### *Feature*

- Provide fallback behavior for interface methods.
- Allow interface evolution without requiring all implementing classes to provide an implementation.
- Can be overridden by implementing classes if needed.

### *Syntax*

```
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default implementation");
    }
}
```

### *Example*

```
interface MyInterface {
    void abstractMethod();
    default void defaultMethod() {
        System.out.println("Default implementation");
    }
}
```

```java
class MyClass implements MyInterface {
    public void abstractMethod() {
        System.out.println("Abstract method implemented");
    }
}

public class DefaultMethodExample {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.abstractMethod(); // Outputs "Abstract method implemented"
        obj.defaultMethod(); // Outputs "Default implementation"
    }
}
```

### *Explanation*

- defaultMethod provides a default implementation in the interface.
- MyClass must implement the abstract method but inherits defaultMethod unless overridden.
- Default methods enable backward-compatible interface enhancements, especially in APIs like Java's Collections framework.

### *Remark*

- Default methods resolve the "diamond problem" in multiple inheritance by requiring explicit overrides if conflicts arise.
- They are useful for adding new functionality to existing interfaces without breaking compatibility.
- Use sparingly to avoid overly complex interface designs.

---

## 2. Function Overloading

## Definition:

### **Function Overloading**: Having **multiple methods with the same name** in a class but **different parameter lists** (different number or types of parameters).

### *Feature*

- Makes code easier to read.
- Same method name for similar actions with different inputs.

### *Example*

```java
class Print {
  void show(int a) {
    System.out.println(a);//prints an integer.
  }

  void show(String s) {
    System.out.println(s);//prints a string.
  }
}
```

```
public class Printer {
    public void print(String s) {
        System.out.println(s);
    }

    public void print(int n) {
        System.out.println(n);
    }

    public void print(String s, int n) {
        System.out.println(s + " " + n);
    }
}
```

- `print` is **overloaded** three times:
  - Takes a String.
  - Takes an int.
  - Takes a String and an int.

### *Explanation*

- Both methods are named `show` but have **different parameter types**.
- Java decides which one to call based on **argument type**.

### *Remark*

- Function overloading only depends on **method name and parameter list**, not return type.
- Overloading increases **readability** by using the same method name for similar actions.
- Overloading is resolved at **compile-time** based on method signatures.

---

## 3. Why Use Interfaces?

### *Reasons*:

- Define a **common set of methods** that different classes agree to implement.
- Enable writing code that works on **multiple unrelated classes** as long as they implement the same interface.

### *Example*

- A `Flyable` interface could be implemented by `Bird` and `Airplane` classes, even though they are unrelated in class hierarchy.

---

## Reference: “Objects First with Java”

- **Chapter 8**: Interfaces and function overloading.

---

8. Inheritance and Polymorphism

# 1.Inheritance

## Definition:

### **Inheritance** is a mechanism where a new class (**subclass/child class**) get **fields and methods** from an existing class (**superclass/parent class**).

### *Feature*

- Promotes **code reuse**.
- Subclass can:
    - Use methods and fields from superclass.
    - Add new fields and methods.
    - Override existing methods to change behavior.

### *Syntax*

```
class Subclass extends Superclass {
    // additional fields or methods
}
```

### *Example*

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
```

### *Explanation*

- `Dog` **inherits** `eat()` from `Animal`.
- `Dog` can also have its own method `bark()`.

### *Remark*

- The `extends` keyword shows inheritance.
- A subclass **inherits everything** from the superclass except constructors and private fields/methods.

---

## 2.Overriding Method

### Definition:

**Overriding Methods means a subclass can provide its own version of a method from its superclass.**

### *Feature*

- Changes the **behavior** of inherited methods.

### *Syntax*

```
@Override
void methodName() {
    // new implementation
}
```

### *Example*

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    @Override
    void eat() {
        System.out.println("Dog eats");
    }
}
```

### *Explanation*

- `Dog` overrides `eat()` from `Animal` with its own version

### *Remark*

- The `@Override` annotation is optional but recommended.
- `@Override` annotation tells the compiler you intend to override a method; helps catch mistakes if method signature doesn't match.

---

## 3.Abstract Data Type

### Definition

#### An **Abstract Data Type (ADT)** is a **concept** that defines a data type by its **behavior (operations)** rather than its implementation.

### *Feature*

- Defines **what operations** can be done.
- Hides **how** operations are implemented.
- Allows **flexibility** in choosing different implementations.

### *Syntax*

```
// ADT itself has no syntax in Java.
// It is implemented using interfaces or abstract classes.

public interface Stack<T> {
    void push(T item);
    T pop();
    boolean isEmpty();
}
```

### *Example*

- `List`, `Set`, `Map` are ADTs.
- You can implement `List` using `ArrayList` or `LinkedList`.

### *Explanation*

- **ADT focuses on behavior**, not implementation details.
- For example, Stack ADT can be implemented using **array or linked list**, but users only care about **push/pop/ isEmpty** operations.

### *Remark*

- ADT is a **design concept**, not a Java keyword.
- Helps in writing **flexible and reusable code**.

---

# 4.Abstract Method and Abstract Classes

## Definition

#### **Abstract methods** is the method without body.

#### **Abstract Classes** are classes that cannot create objects directly. They can have abstract methods.

### *Features*

- Abstract Classes are used when you want to define a **general class** but don't want to create its objects.
- Abstract Classes are used as a **base class** to be extended by subclasses.
- Abstract Class can have both
  - **Abstract methods** (no body).
  - **Concrete methods** (with body).

### *Syntax*

```java
abstract class ClassName {
    abstract void abstractMethodName();
}
```

### *Example*

```java
public abstract class Animal {
    public abstract void makeSound(); // abstract method

    public void sleep() { // concrete method
        System.out.println("This animal sleeps.");
    }
}

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

### *Explanation*

- `Animal` is abstract; `makeSound` has no body.
- `Cat` extends `Animal` and provides implementation for `makeSound`.

### *Remark*

- Subclasses of abstract classes **must implement** all abstract methods unless they are also abstract.
- Abstract classes can have both **abstract methods** and **normal methods**.

### *Discussion*

#### Does an abstract method always need the `abstract` keyword?*

**In classes: Yes!**

- If you declare an abstract method in a class, you must use the `abstract` keyword.

```
abstract class Animal {
    abstract void makeSound(); // Must use 'abstract' here
}
```

**In interfaces No!** In an **interface**, methods are **automatically abstract** (unless they are default or static methods).

```
public interface Printer {
    void print(String s); // This is abstract even without 'abstract' keyword
}
```

---

## 5.Inheritance-based Polymorphism

## Definition

#### **Inheritance-based Polymorphism** allows us to use a superclass reference to point to a subclass object.

### *Syntax*

```
Superclass obj = new Subclass();
```

### *Features*

- Enables writing **general code** that works on superclass type, but executes subclass behavior.

### *Example*

```
Animal myAnimal = new Dog();
myAnimal.eat(); // Calls Dog's eat() if overridden, otherwise Animal's eat()
```

### *Explanation*

- Although `myAnimal` is of type `Animal`, it runs `Dog`'s overridden `eat()` method.
- `myAnimal` is declared as type `Animal` but references a `Dog` object.
- This is called **polymorphism**: the method that runs depends on the actual object type, not the declared variable type.

### *Remark*

- Polymorphism works with **method overriding**.
- Allows flexible and reusable code design.
- Useful when writing methods that can work with many subclasses.

## Reference: "Objects First with Java"

- **Chapter 9**: Inheritance, abstract classes, overriding, polymorphism.

---

9. Advanced -- Functional Programming

# 1. Anonymous Classes

## Definition:

### A **class without a name**, created and used immediately, usually to **override methods** of a class or interface **for one-time use**.

### *Feature*

- Allows creating **small, quick classes** without writing a separate class file to do something immediately.
- Often used to **override abstract methods** quickly.

### *Syntax*

```
ClassOrInterface obj = new ClassOrInterface() {
    // override methods here
};
```

### Example

```java
Animal cat = new Animal() { // creates an anonymous subclass of Animal.
    @Override
    public void makeSound() {
        // providing a new implementation of makeSound.
        System.out.println("Meow!");
    }
}; // Remember ";"

cat.makeSound(); // Output: Meow!
```

### Explanation

- `new Animal() { ... }` creates an **anonymous subclass of Animal**.
- `makeSound()` is **overridden inside** the anonymous class.
- `cat` can use the new behavior immediately.

### Remark

- Useful for **quickly creating objects** with specific behavior without creating a named subclass.
- Anonymous classes are often used when **overriding methods of abstract classes or interfaces** without creating a full named class.

---

## 2. Functional Interfaces

### Definition:

#### An interface with **exactly one abstract method.**

### Feature

- Can have **default or static methods** (with body).
- Must have **exactly one abstract method**.

### Syntax

```java
@FunctionalInterface
public interface InterfaceName {
    ReturnType methodName(Parameters);
}
```

### Example

```java
@FunctionalInterface
public interface Printer {
    void print(String s);
}
```

### *Explanation*

- `Printer` has **one abstract method**: `print(String s)`.
- `@FunctionalInterface` annotation ensures the interface stays functional.

### *Remark*

- Examples: **Runnable** (method `run()`), **Comparator** (method `compare()`).

---

## 3. Lambda Expression

### Definition

**A short function without a name, used to easily implement the single abstract method of a functional interface.**

### *Feature*

- Makes code **shorter and cleaner**.
- Needs a **functional interface** to work.

### *Syntax*

```
(parameters) -> { body }
```

### *Example*

```java
@FunctionalInterface
interface Printer {
    void print(String s);
}

public class Main {
    public static void main(String[] args) {
        Printer p = (s) -> { System.out.println(s); };
        p.print("Hello!"); // Output: Hello!
    }
}
```

- An equivalent anonymous class implementation

```
Printer p = new Printer() {
    @Override
    public void print(String s) {
        System.out.println(s);
    }
};

p.print("Hello Lambda!");
```

## *More Example*

#### 1.Lambda with two parameters

```
interface Adder {
    int add(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Adder adder = (a, b) -> { return a + b; };
        System.out.println(adder.add(5, 3)); // Output: 8
    }
}
```

#### 2.No parameter lambda

```
interface Hello {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Hello h = () -> { System.out.println("Hello World!"); };
        h.sayHello(); // Output: Hello World!
    }
}
```

#### 3.Single statement lambda without braces

```
//Adder adder = (a, b) -> { return a + b; };
Adder adder = (a, b) -> a + b;
System.out.println(adder.add(10, 20)); // Output: 30
```

- If the lambda body only has **one line of code**, `{}` and `return` can be omitted (if it is a return statement, return directly).

**4.Multi-line code

```
interface AdderPrinter {
    int add(int a, int b);
```

```java
}

public class Main {
    public static void main(String[] args) {
        AdderPrinter ap = (a, b) -> {
            System.out.println("Adding numbers: " + a + " + " + b);
            return a + b;
        };

        int result = ap.add(5, 3);
        System.out.println("Result: " + result);
    }
}
```

- Multi-line code are written in curly braces `{}`, and if a return is needed, the `return` statement should be explicitly written.

### *Explanation*

- `(s) -> { System.out.println(s); }` defines a lambda function:
  - Takes input `s`.
  - Prints `s`.
- `p.print("Hello!")` calls the **`print` method** of the `Printer` interface, which is implemented by the lambda function here.

### *Discuss*

#### Why Lambda expression require functional interface.

- Because lambda expression is actually the only abstract method for implementing this interface.

### *Remark*

- Equivalent to writing an anonymous class, but **much shorter**.
- Commonly used with **streams**, **event handlers**, and for passing behavior as an argument.

---

## 4.Function<A,B>

### Definition

#### Function<A,B> is a **functional interface** that takes an input of type **A** and returns an output of type **B**.

### *Feature*

- Has a **single abstract method** `apply(A a)` returning **B**.
- Used to **transform data** from one type to another.

- Often used with **streams' map() method**.

### *Syntax*

```
Function<A, B> func = (A a) -> { return ...; };
```

### *Example*

```
import java.util.function.Function;

public class Test {
    public static void main(String[] args) {
        Function<String, Integer> lengthFunc = s -> s.length();

        System.out.println(lengthFunc.apply("hello")); // Output: 5
    }
}
```

### *Explanation*

- `Function<String, Integer>` means it **takes a String** and **returns an Integer**.
- `lengthFunc.apply("hello")` returns **5**, the length of "hello".

### *Remark*

- Other common function types:
  - `Predicate<T>` : returns boolean
  - `Consumer<T>` : takes input, no return
  - `Supplier<T>` : no input, returns value
- Located in `java.util.function` package.

---

## 4. Streams

### Definition:

#### A **sequence of elements** from a collection, allowing **easy and declarative processing** using operations like **filter**, **map**, and **forEach**.

### *Feature*

- **Does not store data** itself.
- Processes data **step by step** in a pipeline.

### *Syntax*

```
collection.stream().operation();
```

### *Example*

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");

names.stream().forEach(System.out::println);
```

### *Explanation*

- `names.stream()` turns the list into a **stream**.
- `forEach(System.out::println)` prints each element in the stream.

### *Remark*

- Streams help write **shorter and clearer code** for processing collections.

---

## 4.1.filter

### Definition

**Selects elements that satisfy a condition.**

### *Syntax*

```
stream.filter(condition)
```

### *Example*

```
names.stream().filter(s -> s.startsWith("A"))//In this case, s is String type.
```

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Anna", "Charlie");

        names.stream()
             .filter(s -> s.startsWith("A"))
```

```
               .forEach(s -> System.out.println(s));
    }
}
/*Output:
Alice
Anna
*/
```

### *Explanation*

- Keeps only elements starting with "A" (e.g. "Alice").

---

## 4.2.map

### Definition

**Transforms each element into a new form.**

### *Syntax*

```
stream.map(transformation)
```

### *Example*

```
names.stream().map(s -> s.toUpperCase())//In this case, s is String type
```

### *Explanation*

- Changes all names to **uppercase**.

---

## 4.3.forEach

### Definition

**Performs an action for each element (usually prints or uses the element).**

### *Syntax*

```
stream.forEach(action)
```

### *Example*

```
names.stream().forEach(System.out::println);
// "System.out::println" is equivalent to "s -> System.out.println(s)".
```

### *Explanation*

- Prints each element in the stream.

---

## 5. Combining Streams and Lambdas

### Definition:

#### Using **lambda functions with streams** to process collections in a concise and clear way.

### *Feature*

- Can chain multiple operations like **filter + map + forEach** together.

### *Syntax*

```
collection.stream().filter(condition).map(transformation).forEach(action);
```

### *Example*

```
names.stream()
     .filter(s -> s.startsWith("A"))
     .map(s -> s.toUpperCase())
     .forEach(System.out::println);
     //"System.out::println" is equivalent to "s -> System.out.println(s)"
```

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Anna");

        names.stream()
             .map(s -> s.toUpperCase())
             .forEach(s -> System.out.println(s));
    }
}
/* Output:
ALICE
BOB
```

```
ANNA
*/
```

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);

        nums.stream()
            .forEach(n -> System.out.println(n * n));//n is Interger type.
    }
}
/* Output
1
4
9
16
25
8
*/
```

### *Explanation*

1. `filter(s -> s.startsWith("A"))` : keeps only names starting with "A".
2. `map(s -> s.toUpperCase())` : changes them to uppercase.
3. `forEach(System.out::println)` : prints them.

### *Remark*

- Combining **streams** and **lambdas** makes code **more readable, powerful, and efficient** for data processing.

---

## Reference: “Objects First with Java”

- **Chapter 10**: Anonymous classes, functional interfaces, streams, lambda expressions.