Mila University Center
2nd Year – Bachelor's in Computer Science
Course: Object-Oriented Programming

Chapter III: Inheritance and Polymorphism

Lecturer : Dr. Sadek Benhammada
Email : s.benhammada@centre-univ-mila.dz

1. General

1.1. Definition

- Inheritance is a fundamental concept of object-oriented programming that defines a **hierarchical relationship** between a **general class** (also called base class or superclass) and a **specialized class** (subclass or derived class).
- The subclass:
  - **Inherits all attributes and methods** of the superclass,
  - **Can add new attributes and methods**, and
  - **Can override non-static methods** of the superclass to redefine their behavior.
- In Java:
  - A class may have **multiple subclasses**,
  - A class can extend **only one superclass** → Java does **not support multiple inheritance** directly.

2. Implementing inheritance

2.1. Declaring a Subclass

- In Java, a class inherits from another class using the keyword **extends**:

```
public class SubClass extends SuperClass {
   // attributes and methods
}
```

- **Example**

```
class Person {
   protected int id;
   protected String firstName;
   protected String lastName;

   public String getName() { return lastName; }
}
```

```
class Student extends Person {
   private String field;

   public String getField() { return field; }
   public void setField(String f) { field = f; }
}
```

An object of the `Student class` :

1. **Inherits the attributes and methods of the *`Person class`* :**
   - It must have a value for all attributes of the `Person class`
   - It can use all the methods of the Person class
2. **Has additional attributes and methods**
3. **Can override some non-static methods of the *`Person class`***

2.2. Access Modifiers and Inheritance

- Attributes and methods defined with the **`public access modifier`** are accessible by subclasses and all other classes.
- An attribute defined with the **`private modifier`** is inherited but is not directly accessible by subclasses.
- An attribute defined with the **`protected modifier`** is directly accessible in all girls' classes

Best Practice:

- Use **`protected`** for superclass attributes to allow direct access in subclasses.
- Use **`public`** for methods that should be accessible universally.

2.2. Access Modifiers and Inheritance

- **Example**

```
public class Person {
    protected int id;
    protected String firstName ;
    protected String lastName;
    public String getFirstName (){return this.firstName;}
//...
}
```

```
class Student extends Person{
    private String field ;
    public String getFile (){...}
    public void setFile (String f){...}
    //...
}
```

```
class Employee extends Person{
    private String jobTitle;
    public String get jobTitle (){...}
    public void setjobTitle (String j){...}
//...
}
```

2.3. Constructors in Subclasses

- A subclass constructor must explicitly call the superclass constructor using the **`super()`** keyword.
- This call must be the **first statement** in the subclass constructor.
- **Example :**

```
public class Person {
    protected String firstName;
    protected String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

```
public class Student extends Person {
    private String major; // Spécialité de l'étudiant

    public Student(String firstName, String lastName, String major) {
        super(firstName, lastName);
        this.major = major;
    }
}
```

3. Method Overriding

3. Method Overriding

- Method **overriding** is when a **subclass redefines** a method from its superclass **with the same name and parameters**.
  - The method name and parameters **must remain the same**.
  - If parameters differ**, it is not overriding but overloading.**

Example : Overriding toString() method

```
public class Person {
   protected String firstName;
   protected String lastName;

   public Person(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public String toString() {
      return "Name: " + firstName + " " + lastName;
   }
}
```

```
public class Student extends Person {
   private String field;

   public Student(String firstName, String lastName, String field) {
      super(firstName, lastName);
      this.field = field;    }

   @Override
   public String toString() {
      return return "Name: " + firstName + " " + lastName + ", Field: " + field;
   }

   public static void main(String[] args) {
      Student s = new Student("Ali", "Ahmed", "Computer Science");
      System.out.println(s.toString());
   }
}
```

3. Method Overriding

Using `super`

- **`super.attributeName`:** Accesses an attribute from the superclass.
- **`super.methodName()`:** Calls a method from the superclass.
- **`super`** is useful when extending functionality rather than completely replacing it.

Example: Calling the toString() method of the superclass **Person** within the toString() method of the subclass **Student**

```java
public class Person {
    protected String firstName;
    protected String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String toString() {
        return "Name: " + firstName + " " + lastName;
    }
}
```

```java
public class Student extends Person {
    private String field;

    public Student(String firstName, String lastName, String field) {
        super(firstName, lastName);
        this.field = field;
    }

    @Override
    public String toString() {
        return super.toString() + ", Field: " + field;
    }

    public static void main(String[] args) {
        Student s = new Student("Ali", "Ahmed", "Computer Science");
        System.out.println(s.toString());
    }
}
```

4. Abstraction

4. Abstraction

- **Definition:** Abstraction hides complex implementation details and only exposes necessary functionality:
  - Achieved through **abstract** classes and **interfaces**.
  - Defines what an object should do, not how it does it.
  - Reduces code complexity.

- **Abstraction applies to methods and classes:**
  - **An abstract method** is a method **without implementation** that must be implemented by subclasses.

```
abstract returnType methodName ( tpyparams );
```

  - Any class that has at least one abstract method must be declared **abstract** :

```
abstract class ClassName { ... }
```

- ***abstract*** class cannot be instantiated (Objects of an abstract class cannot be created).
- An abstract class can have subclasses that implement the abstract methods.
- if a subclass of an abstract class does not implement all of its abstract methods, then it too must be declared abstract.

4. Abstraction

- **Example:**

```
// Abstract base class
abstract class Shape {
   // Abstract method (must be implemented in subclasses)
   public abstract double getArea();

 // Concrete method that call the abstract method
   public boolean isLargerThan(Shape s) {
      return this.getArea() > s.getArea();
   }
}
```

```
// Concrete subclass representing a rectangle
public class Rectangle extends Shape {
   private double width, height;

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

   @Override
   public double getArea() {
      return width * height;
   }
}
```

```
// Concrete subclass representing a circle
public class Circle extends Shape {
   private double radius;

   public Circle(double radius) {
      this.radius = radius;
   }

   @Override
   public double getArea() {
      return Math.PI * radius * radius;
   }
}
```

4. Abstraction

Example (continued)

- The abstract class Shape defines the common interface for all geometric shapes by declaring an abstract method **`getArea()`**.
- The abstract method **getArea()** represents the conceptual operation of computing the surface area, but without specifying how the computation is performed.
- Subclasses such as **Rectangle** and **Circle** are required to provide concrete implementations of this method, in a function of their specific geometrical formulas.
- The method **isLargerThan(Shape s)** in the **Shape** class utilizes **getArea()** to compare the areas of two shapes.
- Since each subclass provides its own implementation of **getArea()**, the correct behavior is invoked at runtime based on the actual type of the object.
- Any subclass (Circle, Rectangle, etc.) can use **isLargerThan(Shape s) without rewriting it**. It just need to implement **getArea()** .

4. Abstraction

Example (continued)

```
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(3);         // Area ≈ 28.27
        Shape rectangle = new Rectangle(4, 5);  // Area = 20.0

        // Comparing two shapes using isLargerThan
        if (circle.isLargerThan(rectangle)) {
            System.out.println("The circle is larger than the rectangle.");
        } else {
            System.out.println("The rectangle is larger than or equal to the circle.");
        }
    }
}
```

5. Polymorphism

5. Polymorphism

5.1. Polymorphism

- **Polymorphism** is the ability of the same method call (or message) to result in different behaviors depending on the actual type of the object that receives it.
- In Java, polymorphism is achieved through **inheritance**, **interfaces**, **method overriding,** and **object cating**

```
Shape s = new Rectangle(5, 10);

Shape c = new Circle(10);

System.out.println(s.getArea()); // Executes Rectangle's getArea()

System.out.println(c.getArea()); // Executes Circle's getArea()
```

5. Polymorphism

5.2. Object casting

- **Object casting** refers to the process of converting one object reference type into another
- Object casting can be categorized into **upcasting** and **downcasting**.

5.2.1. Upcasting (implicit): Subclass → Superclass

- **Upcasting** refers to the conversion of a child class reference to a parent class type.
- It is **implicit** and does not require a cast operator.

5.2.2. Downcasting (explicit): Superclass → Subclass

- **Downcasting** refers to converting a parent class reference back to a child class type.
- It is **explicit** and requires the use of the cast operator **`(Subclass)`**.
- **Downcasting** is typically used to access methods or fields specific to the subclass, which are not available in the parent class.

5. Polymorphism

5.2. Object cast

- An object has two types:
  1. **Declared Type (Reference Type):**
     - This is the **type of the reference** variable used to refer to the object.
     - It is verified at **compile-tim**e by the **Java compiler (javac).**
  2. **Actual Type (Runtime Type):**
     - Determined by the **constructor** used during the object's creation.
     - It is checked at **runtime** by the **Java Virtual Machine (JVM)**.

Example 1

```
public class Person{
   private String name;
   //...
   private void setName (String n){ this.name =n;}
   }
```

```
public class Student extends Person{
   private String field ;
   //...
   private void setField (String f){ this.field =f;}
}
```

```
Person pers = new Student();
```

Declared type — *Actual type*

5. Polymorphism

5.2. Object casting

Example 1 (continued)

Code	Correct/Error	Explanation
Student e = new Student();	Correct	A Student reference is assigned to a Student object, which is valid.
Person p = new Student();	Correct	A Person reference is assigned to a Student object, valid due to **Upcasting**.
Student e = new Person();	Compilation Error	A Student reference cannot hold a Person object (as not all Person objects are Student objects)
Student e= new Student(); **Person p = e;**	Correct	A Person reference is assigned to a Student object. This is valid due to **upcasting**.
Student e= new Student(); **Object obj = e;**	Correct	A Student reference can be assigned to an **Object** reference, as **all classes in Java inherit from Object.**
Person p = new Student(); **Student e = p;**	Compilation Error	A Person reference cannot be directly assigned to a Student reference without ***explicit downcasting***.
Person p = new Student(); **Student e = (Student)p;**	Correct	**Upcasting**: Person p = new Student(); is valid as a Student object can be referred to by a Person reference. **Downcasting**: e = (Student)p; works since p refers to a Student object, making the explicit cast

5. Polymorphism

5.2. Object casting

Example 1 (continued)

Code	Correct/Error	Explanation
Person p = new Student(); **p.setName("Ahmed");**	Correct	setName () is a method of the declared type of p1 (Person)
Person p = new Student(); **p.setField("Math");**	Compilation Error	The setField() method is not part of the Person class, so it cannot be called using a Person reference without ***explicit downcasting***.
`Person p=new Student();` `((Student)p).setField("Math");`	Correct	Correct : ***Downcasting*** Superclass → Subclass, and the real type of **p** is Student.

5. Polymorphism

- **Example 2**

```
public class Person {
    protected String name;
    protected String firstName;
    protected int age;

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

    // Method to compare ages
    public boolean isOlderThan(Person p) {
        return this.age > p.age;
    }
}
```

```
public class Student extends Person {
    private String field;

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

    // Method specific to Student
    public String getField() {
        return field;
    }
```

5. Polymorphism

- **Example 2 (continued)**

```
public class Test {
    public static void main(String[] args) {
        // Create a Person object
        Person person = new Person("Idir", "Kamel", 20);

        // Create a Student object
        Student student = new Student("Mohammed", "Ali", 22, "Computer Science");

        // Use the isOlderThan method
        boolean result = person.isOlderThan(student);  // (UpCasting)

        System.out.println("Is Person older than Student? " + result);

        // Downcasting example: Access Student-specific method
        Person anotherPerson = new Student("Sara", "Ahmed", 21, "Mathematics"); // Upcasting
}
}
```

5. Polymorphism

5.3. The instanceof operator

- ***instanceof*** operator is used to test whether an object is an instance of a given type or one of that type's subclasses.

Example

```
public class Person{}
```

```
public class Student extends Person{}
```

```
public class Employee extends Person{}
```

```
public class Test {
public static void main(String arg []){
Person e = new Student();
System.out.println (e instanceof Student); // true
System.out.println (e instanceof Person); // true
System.out.println (e instanceof Employe ); // false
```

5. Polymorphism

5.3. Uses of Polymorphism

Example:

```java
// Superclass
public abstract class Shape {
    public abstract double getArea();
}
```

```java
// Subclass: Rectangle
public class Rectangle extends Shape {
    private double length;
    private double width;

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

    // Implement getArea
    @Override
    public double getArea() {
        return length * width;
     }
}
```

```java
// Subclass: Cirlce
public class Circle extends Shape {
    private double radius;


    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }

    // Implement getArea
    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

5. Polymorphism

5.3. Uses of Polymorphism

Example(continued):

```
public class TestPolymorphism {
  public static void main(String[] args) {
    // Create an array of Shape references
    Shape[] shapes = new Shape[4];

    // Populate the array with Rectangle and Circle objects
    shapes[0] = new Rectangle(5, 10); // Upcasting
    shapes[1] = new Circle(5);     // Upcasting
    shapes[2] = new Rectangle(10, 20); // Upcasting
    shapes[3] = new Circle(10);    // Upcasting

    // Display the list of shapes and their areas
    System.out.println("List of shapes and their areas:");
    for (int i = 0; i < shapes.length; i++) {
      if (shapes[i] instanceof Rectangle) {
        System.out.println("A rectangle of area: " + shapes[i].getArea());
      } else if (shapes[i] instanceof Circle) {
        System.out.println("A circle of area: " + shapes[i].getArea());
      }
    }
  }
}
```

```
Lists of shapes and their areas
A rectangle of area:50.0
A Circle of areas : 78.54
A rectangle of area: 200.0
A Circle of areas : 314.16
```

5. Polymorphism

5.3. Uses of Polymorphism :

1. Code Reusability: Write general code for a superclass that works for all its subclasses.

- **Example**: Using a **Shape** superclass with methods applicable to all shapes (Example : **getArea()**).

2. **Extensibility**: Add new subclasses without changing existing code that uses the superclass.

- **Example**: Adding a **Triangle** as a subclass of **Shape** without modifying the existing **Shape** superclass or other subclasses like **Rectangle** or **Circle**.

3. Dynamic Behavior: Method calls are resolved at runtime, allowing behavior to depend on the object's actual type.

- **Example**: A : **getArea()** method behaves differently for **`Circle`** and **`Rectangle`**.

4. Simplifies Maintenance: Reduces code duplication and then simplifies maintenance.

- **Example**: A **Shape** superclass with common methods like **`isLargerThan(Shape s)`** for all types of Shapes.

- ...

6. The *final* Keyword

6. The *final* Keyword

- `final` keyword applies to attributes, methods and classes.

Usage	Meaning
final attribute	The value of the attribute cannot be changed after it is initialized.
final method	The method cannot be overridden by subclasses, ensuring its implementation is preserved.
final class	The class cannot be extended, preventing any subclass from being created.

6. The *final* Keyword

6.1. Final variables

- A variable declared *final* can no longer have its value modified (a constant) .
- If it is an attribute, constants are also declared static, to save memory space (one copy for all objects).

Example

```
public class Circle extends Shape {
static final double PI= 3.141592653589793 ;
   private double beam;
public Circle(double r) {
radius=r;
}
   public double getSurface () {
return radius*radius*PI;
   }
}
```

6. The *final* Keyword

6.2. Final methods

- A final method in Java cannot be overridden by any subclass. Subclasses must use the inherited method as-is, ensuring its implementation remains unchanged across the inheritance hierarchy.
- **Example**

```
// Final class example
public final class A {
    public final void method() {
        System.out.println("This is a final method in a final class.");
    }
}
```

```
// Subclass attempting to override the final method
public class B extends A {
    // The following method will cause a compilation error
    @Override
    public void method() {
        System.out.println("Attempting to override the final method."); // Error
    }
}
```

6. The *final* Keyword

6.3. Final classes

- A final class cannot be extended.
- No subclass can inherit from a final class.
- The class's **attributes and methods of final class are locked** and cannot be altered via inheritance.

Example

```
public final class MyClass {
// class body
}
```

```
public class SubClass extends MyClass {
//  Error: Cannot inherit from final class
}
```

- Many classes in the Java library are **`final`** , including: **`java.lang.System`**, **`java.lang.String`** , and **`java.lang.Math`**

7. Interfaces

7. Interfaces

- With multiple inheritance, a class can inherit from multiple superclasses .
- **Multiple inheritance does not supported in Java.**
- **Interfaces** allow multiple inheritance to be replaced,
- An **interface** is a reference type that can contain:
  - ✓ **Abstract methods** (by default),
  - ✓ **Constants** (public static final),
  - ✓ **Default methods** (with a body, using **default**),

Declaring an interface:

```
[public] interface interfaceName [ extends Interface1, Interface2 ...
] {
// body of the interface
}
```

- The methods of an interface are public abstract: they are **implicitly** declared with the **`public modifier`** and **`abstract ;`**
- The attributes of an interface are public constants, they are **implicitly** declared with the modifiers **`public`** , **`static`** and **`final .`**
- **A default method** of an interface is a method defined by its signature and its implementation.

7. Interfaces

Implementing an interface

- A class implements an interface, inheriting the methods and constants of the interface.
- A class can implement multiple interfaces;

Implementing an interface

```
[Modifiers] class className [ extends superClass ]
[ implements interfacename1, interfacename 2, ...]
{
//class body
}
```

- The class must provide concrete implementations for all abstract methods otherwise it is declared abstract.
- If an interface has default methods, instances of concrete classes that implement that interface can call those methods.
- Classes can also override the default methods of the interfaces they implement.

7. Interfaces

Example

```
public interface Identifiable {
    String getLastName();
    String getFirstName();
}
```

```
public interface Describable {
    void describe();
}
```

```
public class Car implements Describable {

    @Override
    public void describe() {
        System.out.println("I am a car.");
    }
}
```

7. Interfaces

Example (continued)

```
public class Student implements Identifiable, Describable {
    private String lastName;
    private String firstName;

    public Student(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    @Override
    public String getLastName() {
        return lastName;
    }
    @Override
    public String getFirstName() {
        return firstName;
    }
    @Override
    public void describe() {
        System.out.println("I am a student named " + firstName + " " + lastName + ".");
    }
}
```

7. Interfaces

Example 2

```
public interface GeometricShape {
    double PI = 3.14;
    double calculateArea();
    double calculatePerimeter();
     default double square(double value) {
        return value * value;
    }
}
```

```
public class Circle implements GeometricShape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double calculateArea() {
        return square(radius) * PI;
    }
    @Override
    public double calculatePerimeter() {
        return 2 * PI * radius;
    }
}
```

8. Enumerations (Enums)

8. Enumerations (Enums)

- An **enumeration** (or **enum**) is a special data type that enables a variable to be a set of predefined constants.
- Enums are used when a variable (such as a day of the week, a direction, or a state) can only take one value out of a **finite and fixed set**.

- **Example**

```
public enum Field {
        COMPUTER_SCIENCE, MATHEMATICS, ECONOMICS, LITERATURE
}
```

8. Enumerations (Enums)

Example (continued)

```
public class Student extends Person {
    private Field field; // Academic field

    public Student(String firstName, String lastName, int age, Field field) {
        super(firstName, lastName, age);
        this.field = field;
    }

    public Field getField() {
        return field;
    }

    @Override
    public String toString() {
        return super.toString() + ", Field: " + field;
    }

    public static void main(String[] args) {
        Student student = new Student("Mohammed", "Ali", 20, Field.COMPUTER_SCIENCE);
        System.out.println(student);
    }
}
```

9. Arrays and Collections

9. Arrays and Collections

9.1. Arrays

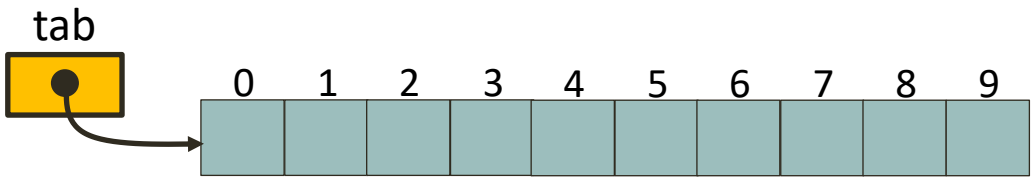
- An **array** is a data structure that allows to group **multiple values of the same type** under a **single identifier**.
- Arrays are used to store **fixed-size collections** of elements and provide indexed access to each element.
- **Syntax**

```
<type> <arrayName> [] =new <type> [ n ]
```

Or:

```
<type> [] <arrayName> = new <type> [ n ]
```

```
int tab[] = new int [10]; // declaration and creation of an array of 10
integers
```




```
char tabc []; // declaration
tabc = new char[10]; //creation
Person tabP =new Person[10] // declaration of an array of objects of the
Person class
```

9. Arrays and Collections

- Java does not directly support multi-dimensional arrays, the solution is to declare **an array of arrays** :

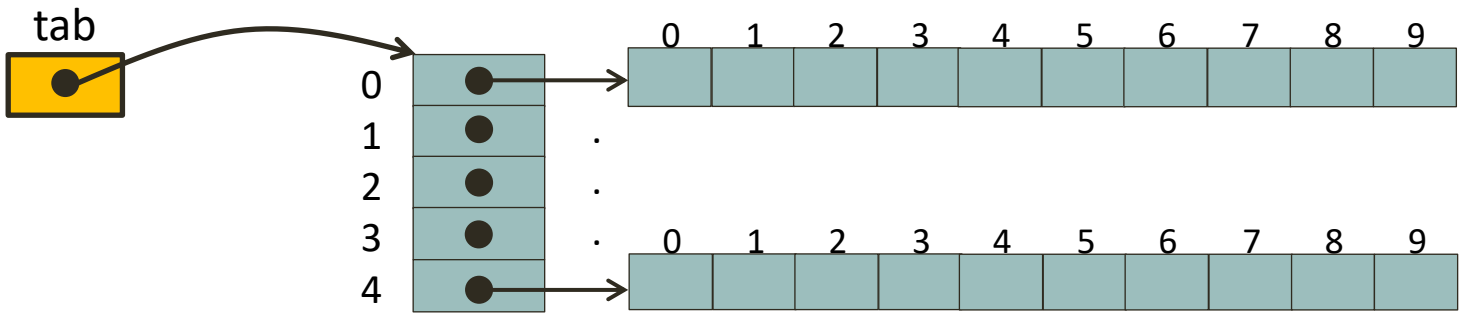
```
<type> <arrayname> [][] =new <arrayname> [n][p]
```

Or

```
<type> [][] < arrayname >=new < arrayname > [n][p]
```

Example

```
int tab [][]=new int [5][10]
```



9. Arrays and Collections

9.1.2. Initialization

- In Java, it is possible to **initialize an array at the time of its creation** or by assigning values manually.
- **Examples**

```
:
int tab1[] = {10,20,30,40,50};
int tab2[][] = {{5,1},{6,2},{7,3}};
```

```
int[] values = new int[3];
values[0] = 5;
values[1] = 10;
values[2] = 15;
```

9.1.3. Iterating through Arrays

- An array has a public constant **`length`** whose value is the size of the array. This constant can be used to iterate through it.

Example

```
int[] data = {3, 6, 9, 12};
for (int i = 0; i < data.length; i++) {
    System.out.println("Element at index " + i + ": " + data[i]);
}
```

9. Arrays and Collections

- In Java, **arrays have a fixed size**. Once an array is created, its length **cannot be changed**.
- For example, if you declare an array of 20 elements, you are limited to storing **exactly 20 elements**, no more, no less.
- To overcome this limitation, Java provides a flexible and powerful set of classes known as the **Java Collections Framework (JCF)**, available in the **`java.util package`**.
- **Key Features of Collections**
  - Collections are **resizable**: Their size can **grow or shrink dynamically**.
  - They are designed to **manage groups of objects** efficiently.
  - Collections provide **higher-level operations** like sorting, searching, filtering, and iteration.

- **Common Collection Types**

Type	Description	Example Class
List	Ordered collection, allows duplicates	ArrayList, LinkedList, Vector
Set	Unordered, no duplicates allowed	HashSet, TreeSet
Map	Stores key–value pairs	HashMap, TreeMap
Queue	First-In-First-Out (FIFO) structure	PriorityQueue, ArrayDeque

9. Arrays and Collections

9.2. Collections: `ArrayList` and `LinkedList`

- Among the most commonly used are **`ArrayList`** and **`LinkedList`**,
- **`ArrayList`** and **`LinkedList`** are part of the **Java Collections Framework** and implement the **`List` interface**.
- Some methods of the List interface:

Method	Description
add(E element)	Appends an element to the end of the list.
add(int index, E element)	Inserts an element at a specified index.
get(int index)	Retrieves the element at the specified index.
set(int index, E element)	Replaces the element at the specified index with a new element.
remove(int index)	Removes the element at the specified index.
remove(Object o)	Removes the first occurrence of the specified object.
size()	Returns the number of elements in the list.
isEmpty()	Returns true if the list contains no elements.
contains(Object o)	Returns true if the list contains the specified element.
clear()	Removes all elements from the list.
indexOf(Object o)	Returns the index of the first occurrence of the specified element.
lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element.

9. Arrays and Collections

9.2. Collections: `ArrayList` and `LinkedList`

Feature	`ArrayList`	`LinkedList`
Underlying structure	Dynamic array	Doubly linked list
Access speed	Fast random access (index-based)	Slower access (sequential traversal)
Insertion/removal	Slower insertion/removal	Faster insertion/removal

9. Arrays and Collections

- **9.2.1. `ArrayList`**
- An **`ArrayList`** uses a **dynamic array** to store a collection of objects. It can **automatically resize** when elements are added or removed.
- **Advantage**: Very fast element access (reading by index).
- **Disadvantage**: Insertion and deletion at arbitrary positions can be costly (due to shifting elements).
- To use `ArrayList`:

  - Import the **`ArrayList class :`**

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

  - Creating the list:

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

  - Once created, the list can be manipulated using methods such as: `add()`, `get()`, `remove()`, `set()`, `size()`, etc.

9. Arrays and Collections

9.2.2. LinkedList

- A LinkedList in Java is implemented as a **doubly linked list**, where each node contains references to both the **next** and the **previous** elements.
- This allows for efficient traversal in **both directions** and improves the performance of **insertions and deletions** at both ends of the list.
- .
- To use **LinkedList** :

  - Import the **LinkedList** `class` :

```
import java.util.LinkedList ;
```

  - Creating the list:

```
LinkedList <String> names = new LinkedList <>();
```

  - Once created, the list can be manipulated using methods such as: `add()`, `get()`, `remove()`, `set()`, `size()`, etc.

9. Arrays and Collections

9.2. Collections: `ArrayList` and `LinkedList`

ArrayList and LinkedList are generic and can be declared to hold heterogeneous types of objects:

- **Example:**

```
public class Person{...}
```

```
import java.util.ArrayList ;
public class TestArrayList {
public static void main(String arg []){
   ArrayList myList = new ArrayList ();
   myList.add ("Hello");
   myList.add (12);
   Person p=new Person("Ahmed", "Ali");
   myList.add (p);
   for( int i = 0; i < myList.size () ; i++)
      System.out.println (" Element "+i+" = "+ myList.get (i));
}
}
```

Note:
Values of primitive types (int, double, float, char, etc.), stored in ArrayList or LinkedList are automatically converted into objects of their corresponding **wrapper classes** (**Integer**, **Double**, **Character, etc**.).

9. Arrays and Collections

9.2. Collections: `ArrayList` and `LinkedList`

- It is possible to create a **homogeneous `ArrayList`** and **LinkedList** , in which the elements are limited to a specific type:

```
ArrayList < typeElements > myList = new ArrayList < typeElements > ();
```

- **Example**

```
 ArrayList <Person> myList =new ArrayList <Person> ();
 myList.add ("Hello"); // Compilation error
 myList.add (12); // Compilation error
Person p=new Person("Ahmed", "Ali");
 myList.add (p); //Correct
```

10. Core Java Packages

10. Core Java Packages

- Java provides a large set of built-in packages collectively known as the **Java Standard API**, which contains thousands of classes grouped by functionality.
- These packages are organized under the java.* and javax.* namespaces.
- You can consult all the API documentation on the site: *http://download.oracle.com/javase/1.4.2/docs/api/*

Key Core Packages

Package	Description
java.lang	Fundamental classes: Object, String, Math, System, wrappers (Integer, Double, etc.). Automatically imported.
java.util	Utility classes: data structures (ArrayList, HashMap, LinkedList), date/time, collections framework, random number generation.
java.io	Input and output: reading/writing files, streams, serialization (File, InputStream, BufferedReader, etc.).
java.nio	Non-blocking I/O: buffers, channels, advanced file and network handling.
java.net	Networking support: sockets, URLs, HTTP connections.
java.math	Mathematical operations beyond primitives: BigInteger, BigDecimal.
java.text	Classes for formatting text, dates, numbers, messages.
java.time	Modern date and time API (LocalDate, LocalTime, Duration, etc.).
java.sql	Classes and interfaces for accessing relational databases using JDBC.

10. Core Java Packages

10.1. The java.lang Package

- The java.lang package contains the **fundamental classes** that form the foundation of the Java programming language. It includes:
    - **Object** – the root superclass of all Java classes
    - **Class** – runtime representation of class metadata
    - **Math** – utility methods for mathematical operations
    - **System** – access to system resources and I/O streams
    - **String** – immutable text objects
    - **Thread** – support for multithreading
    - **Wrapper classes** for primitive types: **Integer**, **Double**, **Boolean**, **Byte**, etc.

Note : This package is **automatically imported** in every Java program : there is no need to explicitly import it.

10. Core Java Packages

10.1.1. The Object Class

Object is the **superclass of all Java classes**.

Every class implicitly extends Object and inherits its methods.

Common Methods of Object:

a) getClass(): Returns a Class object representing the runtime class of the current object.

Example

```
Person p= new Person("Mohammed","Ali",20);
String className = p.getClass ().getName();
System.out.println();  //Prson
```

// The displayed result is:
Person

10. Core Java Packages

b) toString(): Returns a String representing the object. By default, it prints the class name followed by @ and the hashcode (the object's **memory address)**.

```
Person p= new Person(" Mohammed","Ali ");
System.out.println( p.toString()) ;
```

```
// The displayed result can be:
Person@190d11
```

- This method is often **overridden** to provide more meaningful string representations.

c) equals(Object obj) : Checks whether two references refer to the **same object** (by default).

```
Person p1 = new Person("Mohammed", "Ali");
Person p2 = new Person("Mohammed", "Ali");

System.out.println(p1.equals(p2)); // false
p1 = p2;
System.out.println(p1.equals(p2)); // true
```

- You can **override equals()** to define logical equality, for example in the String class or custom classes.

10. Core Java Packages

10.1.2. The class java.lang.*String*

- In Java, a string is contained in an object of the `String class .`
- A String variable can be initialized without explicitly calling a constructor.

Example: The following two instructions are identical:

```
1  String s= "hello";
2  String s= new String("hello");
```

- The + operator allows the concatenation of character strings.
- Comparing two strings must be done using the **`equals () method`** which compares strings, and not the **`== operator`** which compares the references of these objects:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println (s1 == s2); // prints false
System.out.println (s1.equals(s2)); // print true
```

10. Core Java Packages

10.1.2. The class java.lang.*String*

The String class has many methods. Here are some of them:

Method	Description
length()	Returns the number of characters
charAt(int index)	Returns the character at a position
substring(int, int)	Returns a substring
toLowerCase()	Converts to lowercase
toUpperCase()	Converts to uppercase
indexOf(String)	Finds position of substring
equals(String)	Compares contents

10. Core Java Packages

Example :

```
String s = "COMPUTER";

System.out.println(s.length()); // Prints System.out.println(s.charAt(2));
// Prints M
System.out.println(s.startsWith("COM")); // Prints true
System.out.println(s.endsWith("TER")); // Prints true
System.out.println(s.concat(" AND MATHEMATICAL")); // Displays COMPUTER AND
MATHEMATICAL
System.out.println(s.substring(0,4)); // Prints COMP
System.out.println(s.toLowerCase()); // Displays computer
System.out.println(s.toUpperCase()); // Displays COMPUTER
```

10. Core Java Packages

10.1.3. The class java.lang.*System*

- The System class defines three static attributes that allow the use of input/output streams.

Attribute	Kind	Role
`in`	`InputStream`	Standard system input. By default, this is the keyboard.
`out`	`PrintStream`	Standard system output. By default, this is the monitor.
`err`	`PrintStream`	Standard output of system errors. By default, this is the monitor.

- **`InputStream`** and **`PrintStream`** are classes of the **`java.io package`**
- **`Java.io`** package defines a set of classes for handling input-output streams.
- ***`print`*** and ***`println`*** methods of the ***`PrintStream`*** class exist for the *int* , *long* , *float* , *double* , *boolean* , *char* , and *String types* .
- The **printf () method of the *`PrintStream`*** class uses the well-known operating mode in the C language.

10. Core Java Packages

10.1.3. The class java.lang.*System*

Example

```
System.out.println ( Math.PI );
System.out.printf ("%.2f",Math.PI);
```

Result displayed:

3.141592653589793

3.14

10. Core Java Packages

10.1.4. *The class java.lang .Math*

- Contains a series of mathematical methods and attributes.
- `Math` class is part of the java.lang package , it is automatically imported.
- All methods and attributes of the Math class are **static** .
- **The Math class attributes are:**

```
public static final double PI; //3.14159265358979323846
public static final double E; //2.7182818284590452354
```

- The Math class defines many mathematical methods:

Method	Description
`Math.abs(x)`	Absolute value
`Math.sqrt(x)`	Square root
`Math.pow(x, y)`	Exponentiation (x^y)
`Math.max(a, b)`	Maximum of two values
`Math.min(a, b)`	Minimum of two values
`Math.round(x)`	Rounds to nearest integer
`Math.random()`	Generates a random double in [0,1)

10. Core Java Packages

10.1.4. *The class java.lang .Math*

```
double radius = 5;
double area = Math.PI * Math.pow(radius, 2);
System.out.println("Area of circle: " + area);
```

10. Core Java Packages

10.2. The java.util Package

- The java.util package provides **utility classes** for everyday programming tasks, including:
- **Date and calendar management** (Date, Calendar, LocalDate)
- **Random number generation** (Random)
- **Collections** (ArrayList, LinkedList, HashMap, HashSet, etc.)
- **Scanner** for reading input

- Common Classes in java.util :

Class/Interface	Description
ArrayList	Resizable array (like a dynamic array)
LinkedList	Doubly linked list implementation
HashSet	Collection that doesn't allow duplicates
HashMap	Stores key-value pairs (like a dictionary)
Scanner	Reads input from the user (keyboard, file...)
Collections	Utility class for collection operations (e.g., sort, shuffle)
Arrays	Utility class for array operations
Random	Used to generate random numbers
Date / Calendar	Used for date/time handling (deprecated in favor of java.time)
Stack, Queue	Data structure interfaces and classes
Iterator / ListIterator	Used to loop through collections
Optional	Container to avoid null values (Java 8+)
Objects	Helper class with null-safe methods (e.g., Objects.equals())

10. Core Java Packages

10.2. The java.util Package

Scanner Class : The Scanner class is commonly used to read input from the keyboard:

```
import java.util.Scanner;

public class MyClass {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Please enter a word:");
        String str = sc.next();
        System.out.println("You entered: " + str);

        System.out.println("Please enter an integer:");
        int a = sc.nextInt();
        System.out.println("You entered: " + a);

        System.out.println("Please enter a double:");
        double x = sc.nextDouble();
        System.out.println("You entered: " + x);
    }
}
```