## Polymorphism in OOP

* Polymorphism allows objects of different classes to be treated as objects of a common superclass.
* Dynamic dispatch: The method called on an object is determined at runtime, based on the object’s actual type, not its declared type.
* Example: If S is a subclass of T, you can assign an S object to a T reference. When you call a method defined in both T and S, the version in S is executed if the object is of type S.

| class T {  void f() { System.out.println("T.f"); } } class S extends T {  @Override  void f() { System.out.println("S.f"); } } public class Main {  public static void main(String[] args) {  T v = new S(); // T reference, S object  v.f(); // Output: S.f  } } |
| --- |

This demonstrates dynamic dispatch: the method from the runtime type (S) is called

## Types of Polymorphism

1. Inheritance Polymorphism

* Achieved through class hierarchies.
* Superclass references can point to subclass objects.
* Methods are overridden in subclasses.

2. Structural Polymorphism

* Operations work on objects based on required capabilities, not just inheritance.
* Example: Reversing a list does not depend on the type of elements, just their position.
* NOT natively supported in Java.
* Behavior depends on what methods/operations are available, not class hierarchy.
* Find/Search: Requires an equality check (e.g., .equals() in Java).
* Sorting: Requires a comparison function (e.g., Comparable interface in Java)

Imagine two plugs that fit into the same socket. Even if they’re from different brands, they work **because they fit** — not because they belong to the same company.

### **In Programming:**

* We don’t care **what the object is**, only **what it can do** (what methods it has).
* Java doesn’t directly support this.
* But you can **simulate it** using:
  + Interfaces
  + Generics
  + Object type (less safe)

Code Example: Reversing an Array (Structural Polymorphism)

| void reverse(Object[] arr) {  int n = arr.length;  for (int i = 0; i < n / 2; i++) {  Object temp = arr[i];  arr[i] = arr[n - 1 - i];  arr[n - 1 - i] = temp;  } } |
| --- |

This works for any object array, regardless of the element type

## Equality and Comparison

* Equality Check: Needed for searching in a list. Java uses .equals() (default is reference equality, but can be overridden).
* Comparison: Needed for sorting. Java uses Comparable or Comparator interfaces.

Code Example: Finding an Element in an Array

| int find(Object[] arr, Object key) {  for (int i = 0; i < arr.length; i++) {  if (arr[i].equals(key)) return i;  }  return -1; } |
| --- |

This uses .equals() to check for equality

## Type Safety Issues with Object Arrays

* Problem: You can put any object into an Object[], but extracting it requires casting, which can cause runtime errors.
* Example: Copying a Date[] into an Employee[] is allowed at compile time but fails at runtime1.

Code Example: Unsafe Array Copy

| void arrayCopy(Object[] src, Object[] dest) {  int limit = Math.min(src.length, dest.length);  for (int i = 0; i < limit; i++) {  dest[i] = src[i]; // May throw ClassCastException at runtime  } } |
| --- |

This is unsafe if src and dest are of incompatible types.

## Generics in Java

* Purpose: Provide type safety and avoid casting.
* Generic Classes: Use type parameters to enforce that only compatible types are used.

Code Example: Generic List

| class List<T> {  private T[] data;//List<String> l=new List<>();  // Constructor, methods, etc.  T head() { return data[0]; }  void insert(T item) { /\* ... \*/ } } |
| --- |

Now, List<Date> and List<Employee> are distinct types, and type errors are caught at compile time.

Code Example: Generic Array Copy with Constraints

| <S extends T, T> void arrayCopy(S[] src, T[] dest) {  int limit = Math.min(src.length, dest.length);  for (int i = 0; i < limit; i++) {  dest[i] = src[i]; // Safe if S is subtype of T  } } |
| --- |

This ensures that src elements are assignable to dest elements

# What is Reflection?

Reflection is the ability of a programming language to examine and manipulate its own structure at runtime. In Java, this means you can inspect classes, interfaces, fields, and methods dynamically, and even invoke methods or access fields without knowing them at compile time.

* Introspection: Examining the structure and metadata of classes and objects.
* Intercession: Modifying or invoking behavior at runtime, such as calling methods or changing field values.
* Reification: Representing abstract concepts (like types) as concrete objects (like Class objects).
* Every class in Java has an internal representation called a Class object
* It stores the class's **name, methods, constructors, fields**, etc.
* Java provides three ways to get it:

**ClassName.class** Use this when the class is *known at compile time*.  
 Example: String.class //class java.lang.String  
//Class

**object.getClass()** Use this when you have an *instance* of the object and want its runtime class.  
 Example: myObject.getClass()

**Class.forName("fully.qualified.ClassName")** Use this when you only have the class *name as a String* (for example, read from a file, config, or user input).  
 Example: Class.forName("java.lang.String")//class java.lang.String

**//Class stores metadata using u can access its methods, attributes, constructors Dynamically**

| **Concept** | **Analogy** |
| --- | --- |
| .class | You know the phone model. |
| Class.forName() | You’re told the model name on a call and find it online. |

## 

| **Task** | **How to do it** |
| --- | --- |
| Get class of object | obj.getClass() |
| Create new instance | objClass.getDeclaredConstructor().newInstance() |
| Inspect constructors | objClass.getConstructors() |
| Inspect methods | objClass.getMethods() (public), getDeclaredMethods() (all declared in class) |
| Inspect fields | objClass.getFields() (public), getDeclaredFields() (all declared in class) |
| Set private field value | field.setAccessible(true); field.set(obj, value) |
| Print parameter types | constructor.getParameterTypes() |

## **Java Generics** let you write code that works with different types, while still keeping your code type-safe at compile time. For example, you can make a list that only holds integers or only holds strings, but the code for the list itself is written just once.

## **Type Erasure** is a feature of Java that makes sure generics don't slow down your program. At runtime, Java forgets what type you used in your generics. Instead, all generic types are treated as their upper bound (usually Object). This helps keep Java code compatible with older code that doesn't use generics.

* At compile time: Java checks if you use types correctly.
* At runtime: Java forgets the generic type and treats everything as Object (or the upper bound you specify).
* No runtime type info: You can't ask at runtime if something is a List<String> or List<Integer>—they look the same to Java.
* Primitives: You can't use primitive types (like int, double) directly with generics. Use wrapper classes (Integer, Double) instead.
* Type safety: Generics help prevent bugs by making sure you only use the right types in your code.
* method overloading is allowed with generic types — as long as the **method signatures are different after erasure**.

## Type Erasure in Action

What you write:

| List<String> strings = new ArrayList<>(); List<Integer> numbers = new ArrayList<>(); |
| --- |

What Java sees at runtime:

| List strings = new ArrayList(); // Treats as List<Object> List numbers = new ArrayList(); // Treats as List<Object> |
| --- |

This is why you can't use instanceof to check if a list is List<String>—Java doesn't know at runtime.

**Wrapper classes** in Java are used to convert primitive data types (like int, double, char, etc.) into objects. This is necessary because Java collections (such as ArrayList, HashMap) and generics can only work with objects, not with primitives. Each primitive type has a corresponding wrapper class:

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

Wrapper classes also provide useful methods for converting between strings and primitives, and for formatting or parsing data.

Key Points:

* Autoboxing: Java automatically converts primitives to wrapper objects when needed.
* Unboxing: Java automatically converts wrapper objects back to primitives when needed.
* Collections and Generics: Only wrapper objects can be used in collections and generics.
* Null Values: Wrapper objects can be set to null, whereas primitives cannot.

## 