# STATIC VS. DYNAMIC BINDING

## Binding in Java

- The process by which references are bound to specific classes.
- Used to resolve which methods and variables are used at run time.
  - **Static Binding** (Early Binding)
    - The compiler can resolve the binding at the **compile time**.
  - **Dynamic Binding** (Late Binding)
    - The compiler is not able to resolve the call and the binding is done at **runtime only**.
    - **Dynamic dispatch**

### Static vs. Dynamic Binding

- **Static Binding**
  - The compiler can resolve the binding at compile time.
  - For:
    - Static methods
    - Private methods
    - Final methods
    - Fields

- **Dynamic Binding**
  - The binding is done at runtime only.
  - Dynamic dispatch

### Static binding example – Static methods

```java
public class A {
    public static void m() {
        System.out.println("A");
    }
}

public class B extends A {
    public static void m() {
        System.out.println("B");
    }
}

public class StaticBindingTest {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.m();
        b.m();
    }
}
```

**Output:**
```
A
B
A
```

---

### Static binding (or early binding)

- **Static binding**: bind at **compilation time**
- **Performed if the compiler can resolve the binding at compile time**
- **Applied for**
  - Static methods
  - Private methods
  - Final methods
  - Fields
When to bind?

• void func (Account obj) {
    obj.deposit();
}

• What should the compiler do here?
  - The compiler doesn’t know which concrete object type is referenced by obj
  - the method to be called can only be known at run time (because of polymorphism and method overriding)
  - Run-time binding

Static binding example - Fields

```java
public class A {
    public String someString = "member of A";
}

public class B extends A {
    public String someString = "member of B";
}

public class StaticBindingTest {
    public static void main(String args[]) {
        A a = new A();
        A b = new B();
        B c = new B();
        System.out.println(a.someString);
        System.out.println(b.someString);
        System.out.println(c.someString);
    }
}
```

Output: member of A
member of A
member of B

When to bind?

• void func (Account obj) {
    obj.deposit();
}

• What should the compiler do here?
  - The compiler doesn’t know which concrete object type is referenced by obj
  - the method to be called can only be known at run time (because of polymorphism and method overriding)
  - Run-time binding

Dynamic Binding

```java
public class DynamicBindingTest {
    public static void main(String args[]) {
        Vehicle vehicle = new Car();
        //The reference type is Vehicle but run-time object will be Car
        vehicle.start();
        //Car’s start called because start() is overridden method
    }
}
```

class Vehicle {
    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
}

class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
}

Output: "Inside start method of Car"

NESTED CLASSES

מחלקות מקוננות

malıוקת מקוונת

(Nested Class)

מחלקה מקוונת היא מחלקה שהוגדרה בתוך מחלקה אחרת.

- סטטי (static member)
- לא סטטי (non-static member)
- אנונימי (anonymous)
- מקומתי (local)

class Outer {
    static class NestedButNotInner {
    }
    class Inner {
    }
}
### Static Member Class

- The class is defined within another class.
- Properties and methods of the inner class are visible to the outer class.
- Inner class methods are inherited by the outer class.
- Inner class methods can access the outer class fields.

```java
public class OuterClass {  
    public static class StaticNestedClass {  
        private static String someValue;  
    }  
}  
```

### Non-static Member Class

- All instances of the inner class are associated with an instance of the outer class.
- Association is established at the time of object creation and cannot be changed.
- Inner class objects contain a reference to the outer class object.

```java
public class OuterClass {  
    private class InnerClass {  
        private String innerValue;  
    }  
}  
```

### AbstractMap

- Abstract class representing a map.
- Implements the `java.util.Map` interface.
- Contains `Entry` class for map entries.

```java
public abstract class AbstractMap<K,V> implements Map<K,V> {  
    public static class SimpleEntry<K,V> implements Entry<K,V>, java.io.Serializable {  
        private final K key;  
        private V value;  
        ...  
    }  
}  
```

### Inner Classes

- Classes defined within another class.
- Inner classes can access fields and methods of the outer class.
- Inner classes are local to the outer class.

```java
public class House {  
    private String address;  
    private double height;  
    public class Room {  
        private double height;  
        public String toString() {  
            return "Room height: " + height + " House height: " + House.this.height;  
        }  
    }  
}  
```

### House Example

- Example of using inner classes.
- Shows how to access inner class fields and methods.

```java
public class House {  
    private String address;  
    private double height;  
    public class Room {  
        private double height;  
        public String toString() {  
            return "Room inside: " + address;  
        }  
    }  
}  
```
AbstractList

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    private class Itr implements Iterator<E> {
        ...
    }

    private class ListItr extends Itr implements ListIterator<E> {
        ...
    }
}

19

12

20

21

22