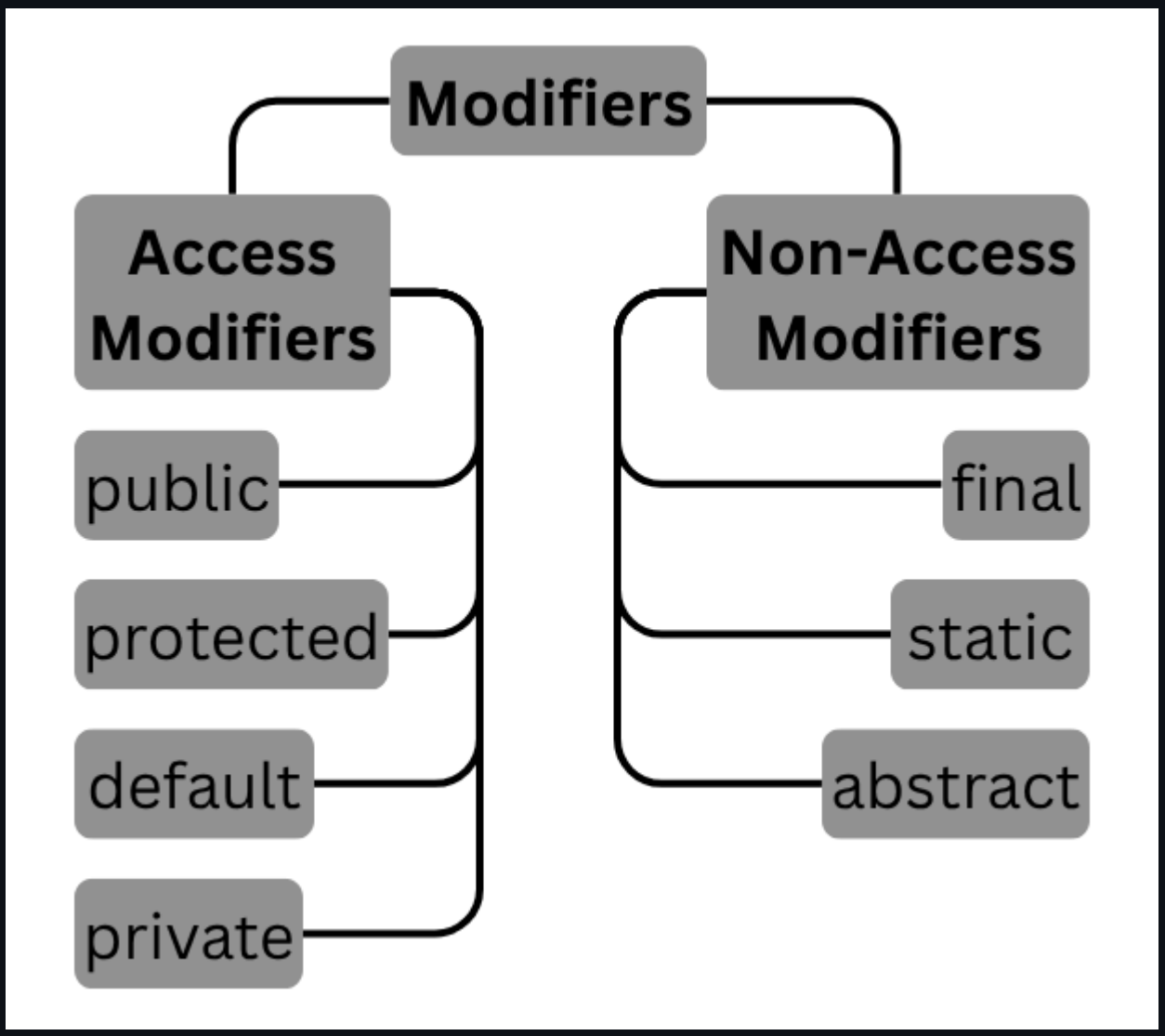
Modifiers

- These are *reserved* **keywords** in Java that determine functionality and accessibility of `class`, `method`, and `variable` declarations.
- There are **two** types of **modifiers:**
    - *Access* modifiers
    - *Non-access* modifiers



Package

- A **package** is a *collection* of several `classes` and `interfaces`.
- To enable communication between packages, use the `import` statement.
    - **Syntax:**

```
1  import packageName.ClassName;
```

Access Modifiers

- *Define* the accessibility scope of `class`, `method`, and `variable` declarations.

1. `public`

- A `class`, `method`, or `variable` declared as **public** can be accessed:
    - *Within* the same class
    - *Outside* the class (in the same package)
    - *Outside* the package (in other packages)
- When accessed from another package, the `class` must be **imported**.

2. `protected`

- A `method` or `variable` declared as **protected** can be accessed:
    - *Within* the same package
    - *Outside* the package, but only by a **subclass**.

Guidelines:

- To invoke *non-static protected* members from a subclass in a different package, use a **subclass object reference**.
- To invoke *static protected* members, use either a **parent** or **subclass object reference**.

3. Default (No Modifier)

- A `class`, `method`, or `variable` with **no explicit modifier** (default) is accessible *within* the same package **only**.

4. `private`

- A `method` or `variable` declared as **private** is accessible *within* the same class **only**.

Demonstration of Access Modifiers

```java
package com.example;

public class SampleClass {
    public int publicValue = 10;
    protected int protectedValue = 20;
    protected static int protectedStaticValue = 30;
    int defaultValue = 50;         // Default access
    private int privateValue = 40;

    public void displayPublic() {
        System.out.println("Public Method of SampleClass.");
    }

    protected void displayProtected() {
        System.out.println("Protected Method of SampleClass.");
    }

    void displayDefault() {
        System.out.println("Default Method of SampleClass.");
    }

    private void displayPrivate() {
        System.out.println("Private Method of SampleClass.");
    }

    public void showPrivateValue() {
        System.out.println("Private Field: " + privateValue);
        displayPrivate();
    }
}
```

```java
package com.example;

public class AccessTest {
    public static void main(String[] args) {
        SampleClass sample = new SampleClass();

        System.out.println(sample.publicValue);
        System.out.println(sample.protectedValue);
        System.out.println(SampleClass.protectedStaticValue);
        System.out.println(sample.defaultValue);
```

```
11
12          sample.displayPublic();
13          sample.displayProtected();
14          sample.displayDefault();
15          sample.showPrivateValue();
16      }
17  }
```

```
1   package com.example1;
2
3   import com.example.SampleClass;
4
5   public class SubClassExample extends SampleClass {
6       public static void main(String[] args) {
7           SampleClass parentReference = new SampleClass();
8           SubClassExample subclassInstance = new SubClassExample();
9
10          System.out.println(subclassInstance.publicValue);
11          System.out.println(subclassInstance.protectedValue);
12          System.out.println(SampleClass.protectedStaticValue);
13
14          subclassInstance.displayPublic();
15          subclassInstance.displayProtected();
16
17          System.out.println(parentReference.publicValue);
18          System.out.println(parentReference.protectedStaticValue);
19
20          parentReference.showPrivateValue();
21      }
22  }
```

Changing Access Modifier of Overridden Methods

- The *access modifier* of an **overridden method** in a subclass can differ from its parent method **only if** the accessibility scope remains the *same* or *increases*.

```
1   class ParentClass {
2       public void performAction() {
3           System.out.println("performAction of ParentClass");
4       }
5
```

```
    protected void performProtectedAction() {
        System.out.println("performProtectedAction of ParentClass");
    }
}

class ChildClass extends ParentClass {
    // Invalid: Reducing visibility from public to protected
    protected void performAction() {
        System.out.println("performAction of ChildClass");
    }

    // Valid: Increasing visibility from protected to public
    public void performProtectedAction() {
        System.out.println("performProtectedAction of ChildClass");
    }
}
```

Parent Modifier	Allowed Child Modifier
`public`	`public`
`protected`	`public`, `protected`
`default`	`public`, `protected`, `default`
`private`	**Cannot be overridden**

Non-access Modifiers

- These modifiers provide additional behavior or constraints to classes, methods, and variables.

1. `final`

- Applicable to `classes`, `methods`, and `variables`.
    - **Variables:** Once initialized, their values cannot be changed.
    - **Methods:** Cannot be overridden by subclasses.
    - **Classes:** Cannot be subclassed (inherited).

```
final int finalValue = 10;
```

```
finalValue = 20; // Compile-time Error

final class ImmutableClass {}
class AttemptInheritance extends ImmutableClass {} // Compile-time Error

class BaseClass {
    final public void execute() {
        System.out.println("execute of BaseClass");
    }
}

class DerivedClass extends BaseClass {
    public void execute() {} // Compile-time Error
}
```

- A final class may contain both final and non-final methods.

```
final class ImmutableClass {
    final public void finalMethod() {}
    public void regularMethod() {}
}
```

- A final class can extend a non-final class.

```
class ParentClass {}
final class ChildClass extends ParentClass {}
```

- Objects can be instantiated from final classes:

```
ImmutableClass instance = new ImmutableClass();
```

2. `static`

Method Hiding

- Redefining a static method in a subclass is known as **method hiding** (not overriding).

```java
class ParentClass {
    public static void staticAction() {
        System.out.println("staticAction of ParentClass");
    }
}

class ChildClass extends ParentClass {
    public static void staticAction() {
        System.out.println("staticAction of ChildClass");
    }
}

class Test {
    public static void main(String[] args) {
        ParentClass parentInstance = new ParentClass();
        ChildClass childInstance = new ChildClass();
        ParentClass parentReferenceToChild = new ChildClass();

        parentInstance.staticAction();           // Calls ParentClass.staticAction()
        childInstance.staticAction();            // Calls ChildClass.staticAction()
        parentReferenceToChild.staticAction();   // Calls ParentClass.staticAction(), based on reference type
    }
}
```

- **Method resolution** for static methods is determined by the **reference type**, not the actual object type.

3. `abstract`

- Applicable to `classes` and `methods`.

Abstract Methods

- Declared without an implementation in the *parent* class.

```java
abstract class AbstractBase {
    abstract public void performTask();
}

class ConcreteImplementation extends AbstractBase {
    public void performTask() {
        System.out.println("performTask of ConcreteImplementation");
    }
}
```

Rules:

- Abstract methods can only exist in abstract classes.
- Static methods cannot be declared abstract.
- All abstract methods in a parent class must be implemented by the first concrete subclass; otherwise, the subclass must also be declared abstract.

```java
abstract class AbstractBase {
    abstract public void performTask();
    abstract public void performSecondaryTask();
}

class FullImplementation extends AbstractBase {
    public void performTask() {}
    public void performSecondaryTask() {}
}

abstract class PartialImplementation extends AbstractBase {
    public void performTask() {}
}
```

Abstract Classes

- Cannot be instantiated directly.
- May contain both abstract and concrete (implemented) methods.

```
abstract class AbstractContainer {
    abstract void abstractMethod1();
    abstract void abstractMethod2();

    void concreteMethod1() {}
    void concreteMethod2() {}
}
```

- **Accessing non-static members** of an abstract class requires using a **subclass object reference**.