

UNIT 4

Inheritance & Packaging

(3 Hours) — *Object Oriented Programming in Java (CACS204)*

This unit introduces inheritance, one of the core pillars of object oriented programming, along with method overriding, dynamic method dispatch, the built-in Object class, abstract and final classes, packages for organizing code, and interfaces for defining shared behaviour across unrelated classes.

4.1 Inheritance

Inheritance is a mechanism by which one class acquires the fields and methods of another class. It models an is-a relationship between classes and allows code to be reused rather than rewritten. The class being inherited from is the superclass (or base/parent class); the class that inherits is the subclass (or derived/child class).

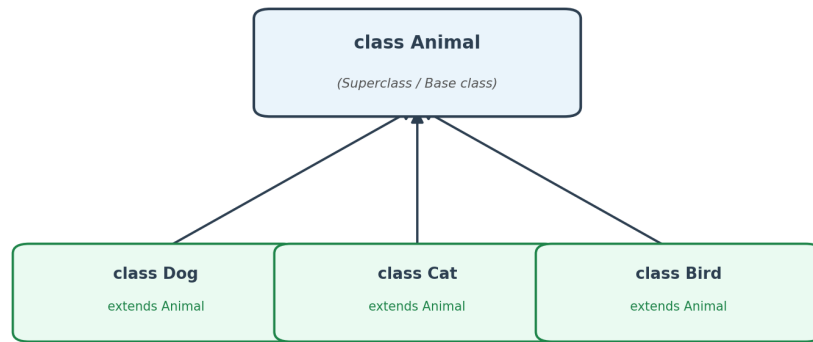
4.1.1 Using the 'extends' Keyword

A subclass is declared using the extends keyword, which establishes the inheritance relationship with its superclass.

```
class Animal {
    String name;
    void eat() {
        System.out.println(name + " is eating");
    }
}

class Dog extends Animal { // Dog inherits name and eat() from Animal
    void bark() {
        System.out.println(name + " is barking");
    }
}
```

An object of class Dog now has access to both its own method bark() and the inherited method eat() and field name, without needing to redefine them.



Subclasses inherit the fields and methods of the superclass,
and may add or override their own

Figure 4.1 — A superclass *Animal* with three subclasses that each extend it.

4.1.2 Subclasses and Superclasses

Java supports only single inheritance for classes — a class may extend exactly one direct superclass (multiple interface implementation, covered later, fills the gap left by not allowing multiple class inheritance). Every class in Java, if it does not explicitly extend another class, implicitly extends the built-in `Object` class (Section 4.5).

Term	Meaning
Superclass	The class being inherited from (parent / base class)
Subclass	The class that inherits (child / derived class)
extends	Keyword used by a subclass to inherit from a superclass

Note: A subclass does not inherit the superclass's constructors, but it can invoke them using `super(...)` (Section 4.2).

4.2 'super' Keyword Usage

`super` is a reference used inside a subclass to refer to its immediate superclass. It has three common uses:

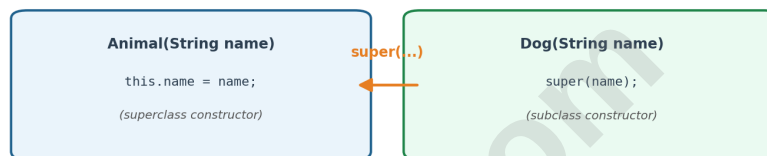
- `super(...)` — calls a constructor of the superclass; if used, it must be the very first statement in the subclass constructor.
- `super.method()` — calls a method of the superclass, typically used inside an overriding method when the subclass still wants to run the superclass's original logic.
- `super.field` — accesses a field of the superclass, useful when the subclass declares a field with the same name (shadowing).

```

class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name); // calls Animal's constructor first
    }
}

```



super(...) explicitly invokes the superclass's constructor before the subclass constructor body runs

super.method() similarly calls the superclass's version of an overridden method

Figure 4.2 — *super(...)* hands off construction to the superclass before the subclass constructor continues.

4.3 Overriding Methods

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass, using the exact same method name, return type, and parameter list. Unlike overloading (Unit 3), overriding is resolved at run time based on the actual object, not the reference type — this is what enables dynamic method dispatch (Section 4.4).

```

class Animal {
    void sound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() { // overrides Animal's sound()
        System.out.println("Woof");
    }
}

```

Note: The @Override annotation is optional but strongly recommended — the compiler will flag an error if the method does not actually override anything in the superclass, catching typos early.

Rule	Requirement for a valid override
Method name	Must match exactly
Parameter list	Must match exactly (same number, types, order)
Return type	Must be the same, or a covariant (narrower) subtype
Access level	Cannot be more restrictive than the superclass version

4.4 Dynamic Method Dispatch

Dynamic method dispatch (also called run-time polymorphism) is the process by which a call to an overridden method is resolved at run time rather than compile time. A superclass reference variable can hold a reference to any of its subclass objects; which overridden version of a method actually executes depends on the type of the object the reference points to at that moment — not the declared type of the variable.

```

Animal ref;
ref = new Dog();
ref.sound(); // Output: Woof (Dog's overridden version runs)
ref = new Cat();
ref.sound(); // Output: Meow (Cat's overridden version runs)

```

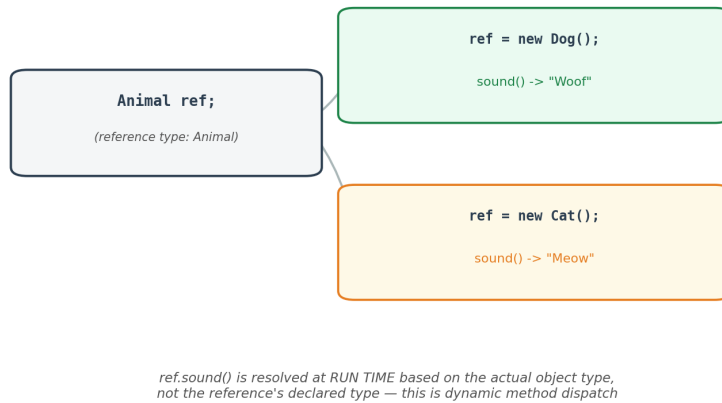


Figure 4.3 — The same reference variable and the same method call invoke different overridden methods, depending on the object currently assigned.

This is the mechanism that allows a single piece of code — for example, a method that accepts an `Animal` parameter — to work correctly with any current or future subclass of `Animal`, without being rewritten.

4.5 The Object Class

`Object` is the root of the Java class hierarchy — every class implicitly extends `Object`, directly or indirectly, whether or not extends is written explicitly. This means every object in Java automatically inherits a common set of methods.

Method	Purpose
<code>toString()</code>	Returns a <code>String</code> representation of the object
<code>equals(Object o)</code>	Tests whether two references are logically equal
<code>hashCode()</code>	Returns an integer hash code used by hash-based collections
<code>getClass()</code>	Returns the runtime class of the object

These methods are commonly overridden to provide meaningful, class-specific behaviour — for instance, overriding `toString()` so that printing an object shows readable field values instead of a default memory-address-like string.

```

class Student {
    String name;
    @Override
    public String toString() {
        return "Student: " + name;
    }
}

```

4.6 Abstract and Final Classes

4.6.1 Abstract Classes

An abstract class is declared with the `abstract` keyword and cannot be instantiated directly — it exists only to be subclassed. It may contain abstract methods (declared without a body, which subclasses are forced to implement) alongside ordinary concrete methods with full implementations.

```

abstract class Shape {
    abstract double area(); // no body — must be overridden
    void display() { // ordinary concrete method
        System.out.println("Area = " + area());
    }
}

class Circle extends Shape {
    double radius;
    double area() { return Math.PI * radius * radius; }
}

// Shape s = new Shape(); // ERROR: cannot instantiate an abstract class
Shape s = new Circle(); // OK — reference of abstract type, concrete object

```

4.6.2 Final Classes

A class declared with the `final` keyword cannot be subclassed at all — extending on a final class is a compile-time error. This is used when a class's implementation must not be altered or extended, often for security or design reasons (Java's own `String` class is final).

```
final class Constants {  
    static final double PI_APPROX = 3.14159;  
}  
  
// class MyConstants extends Constants { } // ERROR: cannot extend a final  
class
```

Note: final can also be applied to individual methods (preventing them from being overridden) and to variables (preventing reassignment after initialization) — a different, narrower use of the same keyword from what makes a class final.

4.7 Packages

A package is a namespace that groups related classes and interfaces together. Packages prevent naming conflicts between classes with the same name in different parts of a project, organize code logically, and let access control extend to the package level (Section 4.7.3).

4.7.1 Defining a Package

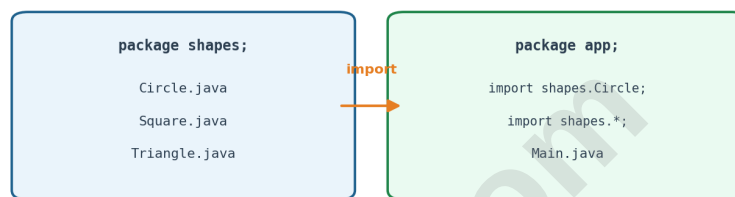
A source file declares which package its classes belong to using a package statement as the very first (non-comment) line in the file. The directory structure on disk must mirror the package's dotted name.

```
// File: shapes/Circle.java  
package shapes;  
  
public class Circle {  
    double radius;  
}
```

4.7.2 Importing a Package

Classes in one package use the import statement to access public classes defined in another package. A single class can be imported by name, or an entire package's public classes can be imported using a wildcard (*).

```
// File: app/Main.java
package app;
import shapes.Circle; // import a single class
// import shapes.*; // or import every public class in the package
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.radius = 5.0;
    }
}
```



Packages group related classes/interfaces into namespaces, avoid naming conflicts, and control access across folders

Directory structure mirrors the package name:
shapes/Circle.java <--> package shapes;

Figure 4.4 — The app package imports and uses a public class from the shapes package.

4.7.3 Access Control Across Packages

Package membership interacts directly with the access modifiers introduced in Unit 3. A class or member with default (no modifier) access is visible only within its own package; public members are visible from any package that imports them; protected members add visibility for subclasses in other packages as well.

Note: Only one public top-level class is allowed per .java file, and its name must match the file name exactly (e.g. Circle.java must contain public class Circle).

4.8 Interfaces

An interface defines a contract of behaviour — a set of method signatures that any implementing class agrees to provide — without dictating how those methods are implemented. Interfaces enable a form of multiple inheritance of type: unlike classes, a single Java class may implement any number of interfaces.

4.8.1 Defining an Interface

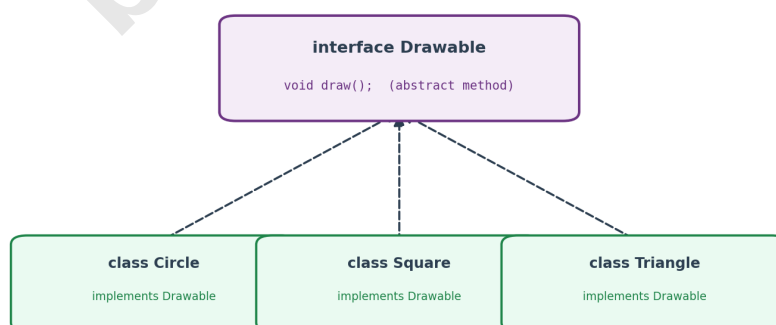
An interface is declared with the `interface` keyword. Its methods are implicitly public and abstract unless declared as default or static methods (a feature added in later Java versions).

```
interface Drawable {  
    void draw(); // implicitly public and abstract  
}
```

4.8.2 Implementing and Applying Interfaces

A class uses the `implements` keyword to declare that it fulfils an interface's contract, and must then provide a concrete body for every abstract method the interface declares — otherwise the class itself must be declared abstract.

```
class Circle implements Drawable {  
    double radius;  
    @Override  
    public void draw() {  
        System.out.println("Drawing a circle of radius " + radius);  
    }  
}  
  
Drawable d = new Circle(); // reference of interface type  
d.draw(); // Output: Drawing a circle of radius 0.0
```



*Each implementing class must supply its own body for `draw()`
(and may implement multiple interfaces, unlike single-class inheritance)*

Figure 4.5 — Multiple unrelated classes can each implement the same interface, each supplying its own version of the required method.

Example 4.1

A class Square also needs to be drawable, alongside Circle and Triangle from Figure 4.5. Show how Square would declare this.

Solution

```
class Square implements Drawable {  
    double side;  
    public void draw() {  
        System.out.println("Drawing a square of side " + side);  
    }  
}
```

Note: A class can implement several interfaces at once, separated by commas: class Circle implements Drawable, Resizable { ... }

Key Takeaways

- ✓ extends creates an is-a relationship: a subclass inherits its superclass's fields and methods
- ✓ super(...) calls the superclass constructor; super.method() calls its overridden method
- ✓ Overriding redefines an inherited method with the identical signature
- ✓ Dynamic method dispatch resolves overridden calls at run time, based on the actual object type
- ✓ Every class implicitly extends Object, gaining toString(), equals(), hashCode(), getClass()
- ✓ abstract classes cannot be instantiated and may hold unimplemented methods; final classes cannot be extended
- ✓ Packages group classes into namespaces and interact with access control (private/default/protected/public)
- ✓ Interfaces define a pure behavioural contract; a class may implement multiple interfaces

— End of Unit 4 —