

UNIT 3

Object Oriented Programming Concepts

(9 Hours) — Object Oriented Programming in Java (CACS204)

This unit builds the foundation of object oriented programming in Java. We begin with the fundamentals of defining a class and creating objects from it, then move through the core pillars of abstraction and encapsulation, the this keyword, constructors, how arguments are passed to methods, access control, returning values, polymorphism through method overloading, recursion, and finally nested and inner classes.

3.1 Fundamentals of Classes

A class is a user-defined blueprint or template from which objects are created. It bundles together data (fields / instance variables) and behaviour (methods) that operate on that data into a single logical unit. A class itself does not occupy memory for its data until an object (instance) is created from it.

3.1.1 A Simple Class

A class declaration in Java has the general form shown below. It consists of an optional access modifier, the keyword class, the class name, and a body enclosed in braces containing fields and methods.

```
class Student {  
    // instance variables (fields)  
    String name;  
    int rollNo;  
    double marks;  
}
```

Here, Student is the class name and name, rollNo, and marks are its instance variables. Each object created from this class will have its own copy of these variables.

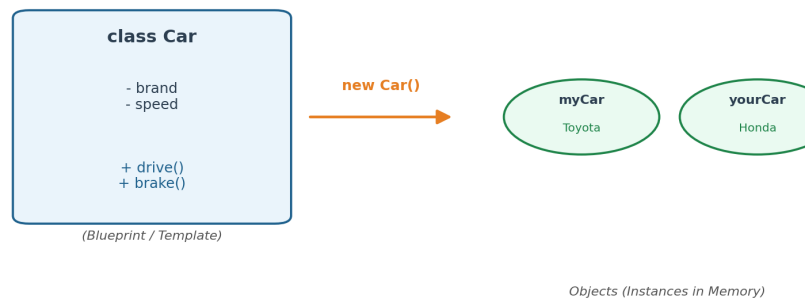


Figure 3.1 — A class acts as a blueprint; objects are concrete instances created from it.

3.1.2 Creating Class Instances

An object is created from a class using the `new` keyword, which allocates memory for the object on the heap and returns a reference to it. This reference is usually stored in a variable of the class's type.

```
Student s1 = new Student();    // s1 refers to a new Student object
s1.name = "Aayush";
s1.rollNo = 12;
s1.marks = 87.5;
```

Note: Each call to `new` creates a completely independent object. Changing `s1`'s fields has no effect on any other `Student` object.

3.1.3 Adding Methods to a Class

Methods define the behaviour of a class — what an object can do. A method has a return type, a name, an optional parameter list, and a body.

```
class Student {
    String name;
    double marks;
    void display() {
        System.out.println(name + " scored " + marks);
    }
}
```

3.1.4 Calling Functions / Methods

A method is invoked (called) on an object using the dot (.) operator. The method's code executes using the calling object's data.

```
Student s1 = new Student();  
s1.name = "Aayush";  
s1.marks = 87.5;  
s1.display(); // calls the display() method on s1  
// Output: Aayush scored 87.5
```

3.2 Abstraction

Abstraction is the process of hiding the internal implementation details of an object and exposing only the essential features or functionality to the user. It lets a programmer focus on what an object does rather than how it does it.

- A car's driver presses the accelerator without needing to know the internal combustion mechanism — that complexity is abstracted away.
- In Java, abstraction is achieved through abstract classes and interfaces, and more generally through any class that exposes a simple public method while hiding its internal logic in private helper code.

Example 3.1 — Abstraction via a simple public method

A Calculator class exposes a single method `add(a, b)` to the user. The user calls `add(5, 3)` and receives 8, without ever needing to know how the addition is carried out internally by the processor.

3.3 Encapsulation

Encapsulation is the mechanism of binding data (fields) and the methods that operate on that data into a single unit (a class), while restricting direct access to some of the object's components. It is achieved by declaring fields as private and providing controlled access through public getter and setter methods.

```

class Account {
    private double balance;    // hidden from outside
    public double getBalance() {    // getter
        return balance;
    }
    public void deposit(double amt) {    // setter-style method
        if (amt > 0) balance += amt;    // validation!
    }
}

```

Because `balance` is private, outside code cannot set it to an invalid value directly (e.g. a negative deposit). All access is filtered through the `deposit()` method, which can enforce rules. This protects the integrity of the object's data.

Why Encapsulation Matters

- ✓ Protects data from unintended or invalid modification
- ✓ Hides internal representation — it can change without breaking outside code
- ✓ Improves modularity, security, and maintainability

3.4 Using the 'this' Keyword

`this` is a reference variable that refers to the current object — the object whose method or constructor is currently executing. It is most commonly used to resolve naming conflicts between instance variables and parameters.

```

class Student {
    String name;
    Student(String name) {
        this.name = name;    // this.name = field, name = parameter
    }
}

```

- `this.name` refers to the instance variable of the current object.
- `name` (without `this`) refers to the constructor's parameter.
- `this` can also be used to call another constructor of the same class (`this(...)`), or to pass the current object as an argument to another method.

3.5 Constructors

A constructor is a special method that is automatically invoked when an object is created with new. It has the same name as the class, has no return type (not even void), and is used to initialize the object's fields.

3.5.1 Default Constructors

If a class defines no constructor at all, Java automatically supplies a default constructor that takes no arguments and initializes fields to their default values (0, null, false, etc.). A programmer may also explicitly write a no-argument constructor.

```
class Point {
    int x, y;
    Point() {                // explicit default (no-arg) constructor
        x = 0;
        y = 0;
    }
}
```

3.5.2 Parameterized Constructors

A parameterized constructor accepts arguments that are used to initialize the object's fields with specific values at the time of creation, avoiding the need to set each field individually afterwards.

```
class Point {
    int x, y;
    Point(int x, int y) {    // parameterized constructor
        this.x = x;
        this.y = y;
    }
}

Point p1 = new Point(3, 5); // x=3, y=5 set immediately
```

Note: A class can have multiple constructors with different parameter lists — this is constructor overloading, an application of the same rules used in method overloading (Section 3.9).

3.6 More on Methods

3.6.1 Passing by Value

Java is strictly pass by value. When a primitive type (int, double, char, boolean, etc.) is passed to a method, a copy of its value is made. Any modification made to the parameter inside the method affects only that local copy — the original variable in the caller remains unchanged.

```
void increment(int n) {  
    n = n + 1;    // modifies the local copy only  
}  
  
int x = 10;  
increment(x);  
System.out.println(x);    // still prints 10
```

3.6.2 Passing by Reference (Object References)

When an object is passed to a method, Java copies the reference (the memory address) to the object, not the object itself. Both the original variable and the method's parameter now point to the same object on the heap. So changes made to the object's fields through the parameter are visible after the method returns — though reassigning the parameter to a brand-new object does not affect the caller's reference.

```
void raiseSalary(Employee e) {  
    e.salary += 1000;    // modifies the SAME object  
}  
  
Employee emp = new Employee();  
emp.salary = 5000;  
raiseSalary(emp);  
System.out.println(emp.salary);    // prints 6000
```

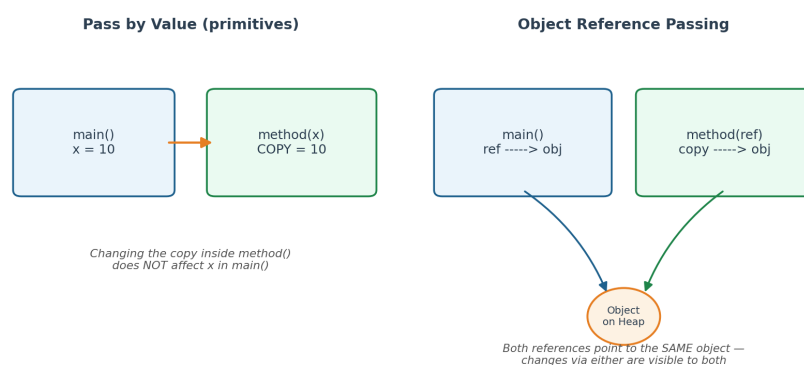


Figure 3.2 — Primitives are copied by value; object variables copy only the reference, so both point to the same underlying object.

3.6.3 Access Control

Java provides four access modifiers that control the visibility of classes, fields, methods, and constructors:

Modifier	Same Class	Same Package	Subclass (other pkg)	Other Package
private	Yes	No	No	No
default (no modifier)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Note: Good encapsulation practice is to keep fields private and expose behaviour through public methods, restricting direct outside access as much as possible.

3.6.4 Methods That Return Values

A method that is not declared void must return a value of the declared return type using the return statement. Execution of the method ends as soon as return is reached.

```
int square(int n) {  
    return n * n;    // returns an int value to the caller  
}  
  
int result = square(6);    // result = 36
```

Example 3.2

Write a method that returns true if a given number is even, and false otherwise.

Solution

```
boolean isEven(int n) {  
    return (n % 2 == 0);  
}
```

3.7 Polymorphism and Method Overloading

Polymorphism ('many forms') allows an entity — typically a method name — to behave differently depending on the context in which it is used. Java supports two kinds of polymorphism: compile-time (static) polymorphism, achieved through method overloading, and run-time (dynamic) polymorphism, achieved through method overriding (covered under inheritance in a later unit).

3.7.1 Method Overloading

Method overloading occurs when a class defines two or more methods with the same name but different parameter lists — differing in the number, type, or order of parameters. The Java compiler decides which overloaded version to invoke based on the arguments supplied at the call site (resolved at compile time).

```
class Calculator {  
    int add(int a, int b) {  
        return a + b;  
    }  
    double add(double a, double b) {  
        return a + b;  
    }  
    int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

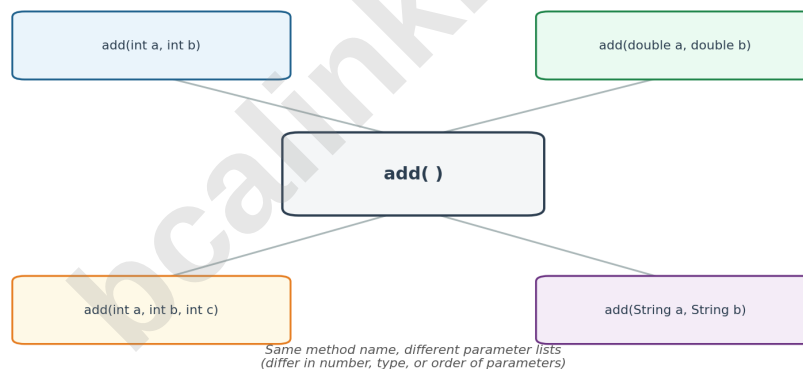


Figure 3.3 — Four overloaded versions of `add()` sharing the same name but distinguished by their parameter lists.

Note: Overloading cannot be based on return type alone — two methods with identical parameter lists but different return types will not compile.

3.8 Recursion

Recursion is a technique in which a method calls itself, either directly or indirectly, to solve a problem by breaking it down into smaller sub-problems of the same kind. Every correct recursive method needs two essential parts: a base case that stops the recursion, and a recursive case that reduces the problem and calls the method again.

```
int factorial(int n) {
    if (n == 0) {
        return 1;           // base case
    }
    return n * factorial(n - 1); // recursive case
}
```

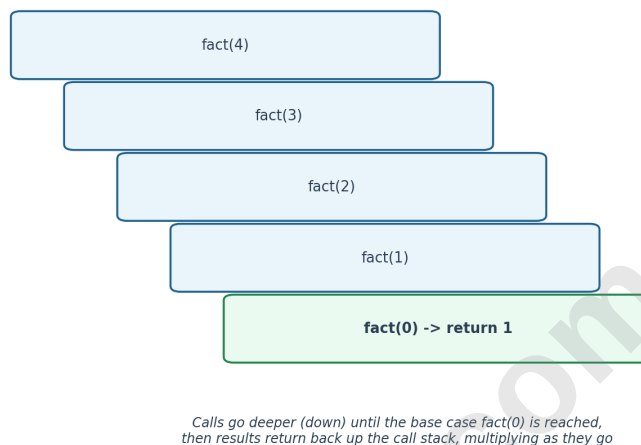


Figure 3.4 — Call stack for `factorial(4)`: calls descend to the base case, then results multiply their way back up.

Example 3.3

Trace `factorial(4)` and give the final result.

Solution

$$\begin{aligned}
 \text{factorial}(4) &= 4 * \text{factorial}(3) \\
 &= 4 * (3 * \text{factorial}(2)) \\
 &= 4 * (3 * (2 * \text{factorial}(1))) \\
 &= 4 * (3 * (2 * (1 * \text{factorial}(0)))) \\
 &= 4 * 3 * 2 * 1 * 1 = 24
 \end{aligned}$$

Note: Every recursive call adds a new frame to the call stack. A missing or unreachable base case causes infinite recursion, eventually leading to a `StackOverflowError`.

3.9 Nested and Inner Classes

A class declared entirely within the body of another class is called a nested class. Nested classes let related classes be grouped logically and increase encapsulation, since the nested class can be hidden from other classes in the same package. Java recognizes several categories:

- **Inner class (non-static)** — an instance member of the outer class; it holds an implicit reference to an instance of the outer class and can directly access the outer class's fields and methods.
- **Static nested class** — declared with the static modifier; it does not hold a reference to an outer instance and behaves much like a regular top-level class, just scoped inside another class's namespace.
- **Local class** — defined inside a method body; visible only within that method.
- **Anonymous class** — a local class without a name, declared and instantiated in a single expression, typically used to provide a one-time implementation of an interface or abstract class.

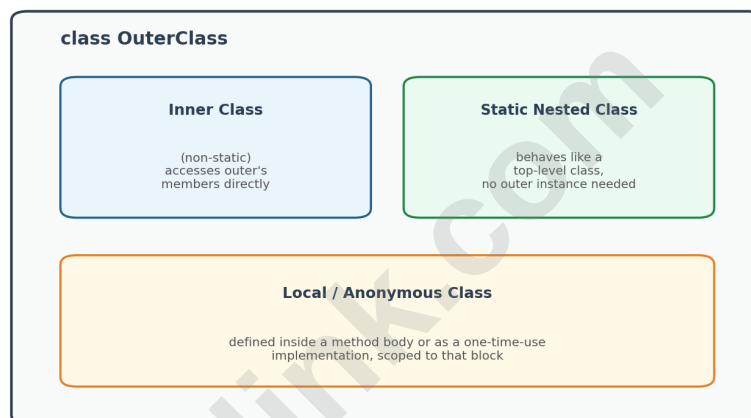


Figure 3.5 — Categories of nested classes in Java.

```

class Outer {
    private int data = 10;

    class Inner {                // non-static inner class
        void show() {
            System.out.println("Outer data: " + data);
        }
    }
}

Outer o = new Outer();
Outer.Inner in = o.new Inner(); // must create via an outer instance
in.show();                      // Output: Outer data: 10
  
```

Note: A static nested class is instantiated without needing an outer object: `Outer.Nested n = new Outer.Nested();`

Key Takeaways

- ✓ A class is a blueprint; objects are instances created from it using new
- ✓ Abstraction hides implementation detail; encapsulation binds data with behaviour and restricts access
- ✓ this refers to the current object and resolves naming conflicts between fields and parameters
- ✓ Constructors initialize objects — default (no-arg) or parameterized
- ✓ Primitives are passed by value (copied); objects pass a copy of the reference to the same instance
- ✓ Access modifiers (private, default, protected, public) control visibility
- ✓ Method overloading gives compile-time polymorphism through differing parameter lists
- ✓ Recursion needs a base case and a recursive case that shrinks the problem toward it
- ✓ Nested/inner classes group related classes together and can access the outer class's members

— End of Unit 3 —