ACHARYA

Dept of AIML

Module 2

Title: Methods and Classes

Subtitle: Overloading Methods, Objects as Parameters, Argument Passing, Returning Objects, Recursion, Access Control, Understanding static, Introducing final, Introducing Nested and Inner Classes.

By. Mohammed Tahir Mirji

Dept of AIML

Content

- Method Overloading
- Objects as Parameters
- Argument Passing
- Returning Objects
- Recursion
- Access Control
- static Keyword
- final Keyword
- Nested & Inner Classes

ACHARYA

Dept of AIML

Method Overloading

Method Overloading allows a class to have multiple methods with the same name but different parameters, enabling compile-time (static) polymorphism.

Rules:

- ✔ **Same method name**
- ✔ **Different number or type of parameters**
- ✖ **Return type alone cannot differentiate methods**

Dept of AIML

EXAMPLE:

```
void test()
{}

void test(int a)
{}

double test(double a)
{ return a * a; }
```

Why Use Method Overloading?

Dept of AIML

Improved Code Readability and Maintainability

- Avoids creating multiple method names for similar operations (e.g., `addInts`, `addDoubles`, `addThreeNumbers`).
- Enables use of a single, meaningful method name like `add`.
- Reduces cognitive load by minimizing the number of method names developers need to remember.

Flexibility and Convenience

- Allows calling the same method name with different parameter types or counts.
- Compiler automatically selects the correct method based on arguments.
- Makes client code simpler and more intuitive when interacting with the class.

ACHARYA

Dept of AIML

Achieving Compile-time (Static) Polymorphism

Overloaded methods enable polymorphic behavior determined at compile time.

- Method selection is based on the method signature (name + parameter list).
- Provides early binding, improving execution efficiency.

Handling Different Data Types

- Supports operations on various data types under a single method name.
- Enables specialized implementations for each parameter type.
- Maintains a consistent and uniform interface for users of the class.

ACHARYA

Dept of AIML

Benefits:

- Improves **code clarity**
- Allows **same behavior for different data inputs**
- Helps in **polymorphic programming**

Practical Example: `println()` is overloaded in Java.

Dept of AIML

Objects as Parameters

Objects can be passed to methods like primitive values.

- Java uses **pass-by-value**, but its effect differs for **primitive types** and **reference types**.
- When passing **primitive types**, the method receives a **copy of the value**, so changes inside the method do **not affect the original variable**.
- When passing **objects**, Java passes the **value of the reference** (not the actual object), creating behavior similar to **call-by-reference**.
- The parameter itself **cannot be replaced**, but the method can modify the **object's internal state** by calling its methods.

Dept of AIML

- Creating a class-type variable only creates a **reference** to an object, not the object itself.
- If this reference is passed to a method, both the original variable and the method parameter **refer to the same object**.
- As a result, **changes made to the object inside the method are visible** outside the method.
- Therefore, for objects, Java behaves **as if using call-by-reference**, despite technically being pass-by-value.

```
boolean equalTo(Test obj) {
   return (obj.a == a && obj.b == b);
}
```

Use Cases:

- Comparison of objects
- Copying object state in constructors
- Data transfer between methods

Argument Passing in Java

Let us assume that a function B() is called from another function A(). In this case A is called the "caller function" and B is called the "called function or callee function". Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Type	Passed As
Primitive Data Types	**Pass-by-Value**
Objects	**Value of object reference is passed**

Primitive Example:

```
void method1(int i, int j) { i*=2; j/=2; }
```

Object Example:

```
void meth(Test obj) { obj.a*=2; obj.b/=2; }
```

Dept of AIML

Returning Objects

```
Test incrByTen() {
   return new Test(a + 10);
}
```

Usage:

```
Test result = obj.incrByTen();
```

Dept of AIML

```
#include <bits/stdc++.h>
using namespace std;
class Example {
public:
   int a;

   // This function will take
   // an object as an argument
   void add(Example E)
   {
      a = a + E.a;
   }
};
```

Dept of AIML

```
// Driver Code
int main()
{

    // Create objects
    Example E1, E2;

    // Values are initialized for both objects
    E1.a = 50;
    E2.a = 100;

    cout << "Initial Values \n";
    cout << "Value of object 1: " << E1.a
        << "\n& object 2: " << E2.a
        << "\n\n";

    // Passing object as an argument
    // to function add()
    E2.add(E1);

    // Changed values after passing
    // object as argument
    cout << "New values \n";
    cout << "Value of object 1: " << E1.a
        << "\n& object 2: " << E2.a
        << "\n\n";

    return 0;
```

Dept of AIML

Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

- **A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the solution.**
- **Since called function may further call itself, this process might continue forever. So it is essential to provide a base case to terminate this recursion process.**

Dept of AIML

Recursion Example:

```
int fact(int n) {
   if(n == 1) return 1;
   return n * fact(n-1);
}
```

Access Control (Access Modifiers)

Dept of AIML

public int a;
private int c;

Modifier	Access Scope
public	Accessible from everywhere
private	Within same class only
default (no modifier)	Same package only
protected	Same package + subclasses

static Keyword

Dept of AIML

- Shared among all objects.
- Can be called using class name.

```
static int a;
ClassName.a;
```

final Keyword

Dept of AIML

- Used to create **constants**.

```
final int MAX = 100;
```

Nested & Inner Classes

Dept of AIML

- Class inside another class
- Inner class can access **outer class members**

```
class Outer {
   class Inner { ... }
}
```

Nested & Inner Classes

Dept of AIML

- Class inside another class
- Inner class can access **outer class members**

```
class Outer {
   class Inner { ... }
}
```

Thank you