Course: PARADIGMS AND COMPUTER PROGRAMMING FUNDAMENTALS (PCPF)

Course Instructor

Mrinmoyee Mukherjee B.E (Electronics), M.E (EXTC), PhD (Pursuing)
Assistant Professor
Department of Information Technology
St. Francis Institute of Technology
email: mrinmoyeemukherjee@sfit.ac.in

Academic Year: 2023-24 (Odd Semester)

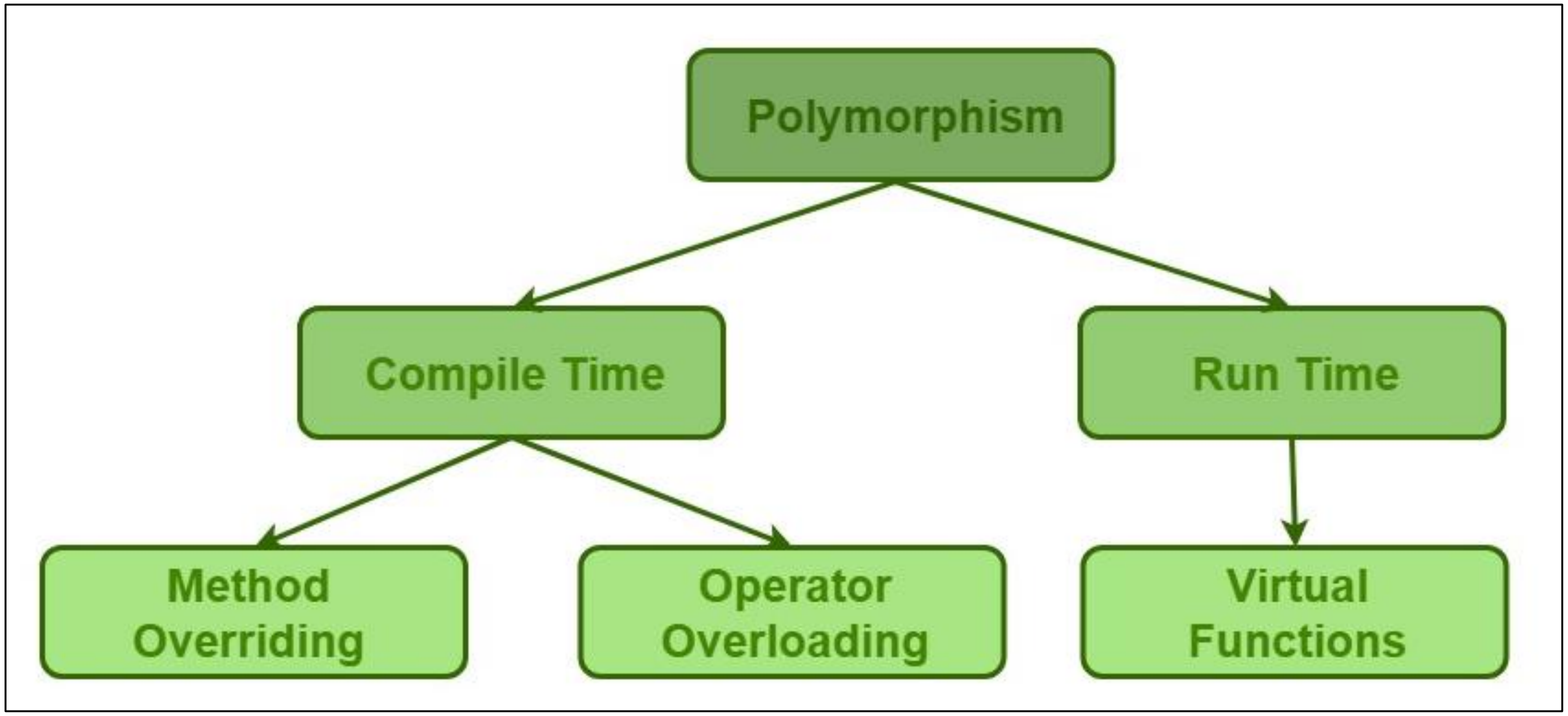
OUTLINE OF UNIT-2

Sub-Unit	Contents
2.1	Grouping of data and operations
2.2	Encapsulation
2.3	Overloading and polymorphism
2.4	Inheritance
2.5	Initialization and finalization
2.6	Dynamic Binding

POLYMORPHISM

- Important concept of object oriented programming
- Polymorphism , a Greek term means the ability to take more than one form
- The behaviour depends upon the types of data used in operation

Eg. *A person possess different behaviour- a father, husband, employee*



Compile Time Polymorphism	Run Time Polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as *overloading*, *early binding and static binding.*	It is also known as *overriding, Dynamic binding and late binding.*
More than one method is having the same name but with the **different** number of parameters or the type of the parameters.	More than one method is having **the same** name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Compile Time Polymorphism

- Compile-time polymorphism: A type of polymorphism which is achieved by function overloading or operator overloading.
- Function Overloading:
  - When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments. Method overloading increases the readability of the program.
  - There are two ways to overload the method in java
    - By changing number of arguments
    - By changing the data type
- Operator Overloading:
  - C++ also provides the option to overload operators. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.
  - Java does not support operator overloading, C++ does

Method Overloading Example

Method overloading: Only one operation, having same name but different parameters

Ex1:

```
class CTimeP {
  void display() {
    System.out.println("Display without
                                   parameter");
  }
  void display(String value) {
    System.out.println("In Display with
                         parameter" + value);
  }
}
# main class
public class Main {
  public static void main(String args[]) {
    CTimeP obj1 = new CTimeP();
    obj1.display();
    obj1.display("Polymorphism");
  }}
```

Ex2:

```
class Adder{
   static int add(int a, int b){
           return a+b;
           }
   static int add(int a, int b, int c){
           return a+b+c;
           }
   static int add(double a, double b){
           return a+b;
           }
}
# main class
class TestOverloading{
  public static void main(String[] args) {
    System.out.println(Adder.add(10,12));
    System.out.println(Adder.add(2, 3, 4));
    System.out.println(Adder.add(3.5, 6.4));
          }}
```

Operator Overloading Example

Ex1: Python

```
print(1 + 2)

# concatenate two
strings

print("Hi" + "All")

# Product two
numbers

print(3 * 4)

# Repeat the String
print("Hello" * 4)
```

Ex2: C++

```
#include<iostream>
using namespace std;
class Complex {
private:  int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}
   Complex operator+ (Complex const &obj) {
       Complex res;
       res.real = real + obj.real;
       res.imag = imag + obj.imag;
       return res;    }
void print() { cout << real << " + i" << imag << endl; }
};
int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print(); }
```

automatically called when '+' is used with/ betn two Complex objects

Output = 12 + i9

Run Time Polymorphism

- Run-time polymorphism:
- It is also called as Dynamic Method Dispatch
- Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism.
- It is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- Method overriding is an example of runtime polymorphism.
  - In method overriding, a subclass overrides a method with the same signature as that of in its superclass.
  - During compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.
- A virtual function is used to achieve Runtime polymorphism
  - A virtual function is a member function which is declared within a base class using virtual keyword and is re-defined (overridden) by a derived class
  - The resolving of function call is done at runtime.
  - Virtual functions ensure that the correct function is called for an object

Method Overriding Example

Ex1: Java

```java
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }}
class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }}

public class TestDog {
  public static void main(String args[]) {
    Animal a = new Animal();
    Animal b = new Dog();
    a.move(); // runs the method in Animal class
    b.move(); // runs the method in Dog class
  }
}
```

Ex2: C++

```cpp
#include <iostream>
using namespace std;
class Base {
  public:
   void print() {
      cout << "Base Function" << endl;   }
};
class Derived : public Base {
  public:
   void print() {
      cout << "Derived Function" << endl;   }
};

int main() {
   Derived derived1;
   derived1.print();
   return 0;
}
```

References-

1. Michael L Scott, " Programming Language Pragmatics", Third edition, Elsevier publication (Chapter-9, specifically 9.1 and 9.2)

2. Ravi Sethi, " Programming Languages-concepts and constructs", Pearson Education (Chapter-6)

3. NPTEL lecture series on Programming in Java, IIT Kharagpur
https://www.youtube.com/watch?v=K9gQwLeNXyw&list=PLbRMhDVUMngcx5xHChJ-f7ofxZI4JzuQR&index=8