Topic: Hiding and Overriding.

OOP concepts involved: Inheritance, Encapsulation, Abstraction, Polymorphism.

Programming generic concepts involved: Functions, Data Types, Variables, Access Modifiers.

➢ Theoric introduction

SIMPLE INHERITANCE: DERIVED CLASSES

Inheritance is an important tool we can use when programming in Java. When you want to create a new class and there is already a class that includes some of the code and behaviour that you want, you can just derive a new class from that existing class.

Inheritance is deeply integrated into Java itself. All classes in Java, including the ones that you create, all inherit from a common class called *Object.java*. Many classes inherit directly from that class, while other inherit from those subclasses and so on, forming a hierarchy of classes.

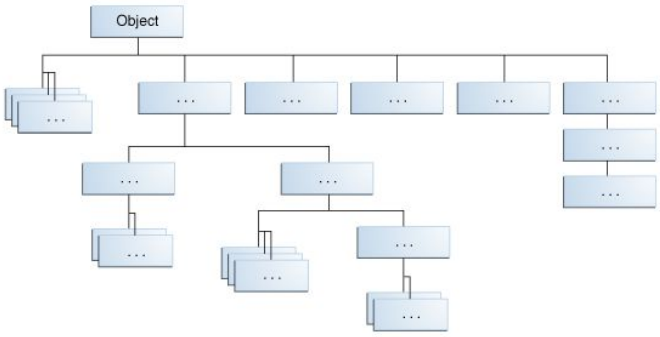


fig 1. Example of inheritance hierarchy tree

One thing to keep in mind is that Java does not support multiple Inheritance. What this means, is that a single class cannot extend two classes.

Inheritance is done with the keyword Extends using this syntax:

```
[Modifiers] class Derived_Class extends Super_Class{}
```

OVERRIDING AND HIDING

As we have learned before, a derived class will inherit all fields and methods from the parent class, but sometimes the behavior of the parent class is just 'close enough' to what we actually need. For cases like this, we have the option to override or hide the fields and methods of that parent class to fit the behavior needed for the derived class.

Overriding is done when we create a new method in the derived class with the same name, number, and type of parameters, and return type as a method in the parent class. Notice how this is different from overloading, as everything must be the same between the two.

```
public class Parent {
	public int testMethod(int a);
}

public class Child extends Parent {
	public int testMethod(int a);	//This is overriding the parent class's
method
	public int testMethod();	//This is overloading the parent class's
method
}
```

When overriding a method, you might want to use the *@Override* annotation that instructs the compiler that you intend to override a method in the superclass. If for some reason, the compiler detects that the method does not exist in one of the superclasses, it will generate an error.

It is still possible to access an overridden method by using the keyword super learned before.

```
public int testMethod(int a){ //This is overriding the parent class's
testMethod(int a)
	super.testMethod(a);	//invoking the overridden testMethod(int a)
	//Rest of body
}
```

Hiding methods is done when we create a new method in the derived class with the same name, number, and type of parameters, and return type as a **static** method in the parent class. The distinction between hiding and overriding is the following:

- The version of the method that gets invoked when overriding is always the one in the subclass
- The version of the method that gets invoked when hiding depends on whether is it invoked from the superclass or the subclass

Hiding fields is done when we create a new field in the subclass with the same name as one in the parent class, even if their types are different. In the subclass, the field cannot be accessed by its simple name. Instead, the field must be accessed through the keyword ***super***. **Generally, hiding fields is not recommended as it makes code difficult to read.**

SUPER KEYWORD

The super keyword is a reference variable which is used to refer to the immediate parent class object. Whenever you create the instance of a subclass, an instance of the parent class is created Implicitly which is referred by super reference variable.

This keyword is primarily used in this three contexts, to allow us to access members of the parent class:

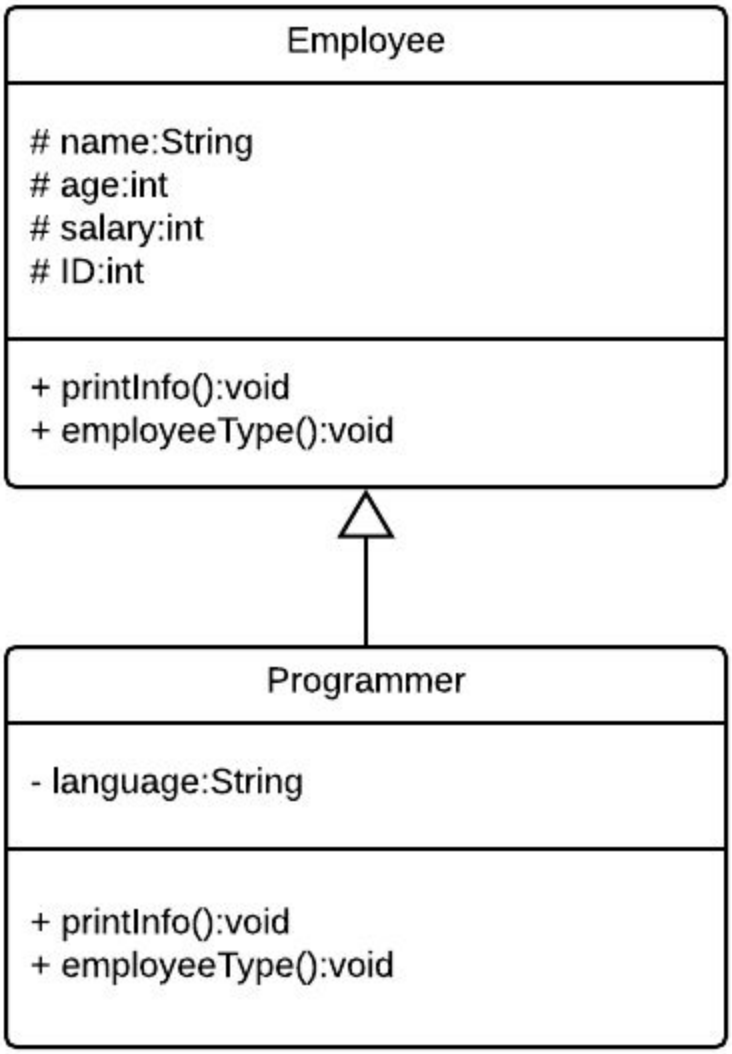
- ➢ **Use of super with fields:** This is done when a field in the parent class is hidden by a field in the subclass. The keyword super allows us to access the hidden fields of the parent class from the subclass.
- ➢ **Use of super with methods:** In this case, the super keyword allows us to access overridden methods of the parent class from the subclass.
- ➢ **Use of super with constructors:** The keyword super() also allow us to invoke the constructor of the parent class. It is important to note, that we can invoke a parametric or non-parametric constructor depending on the situation.

If a constructor does not explicitly invoke the superclass constructor, the Java compiler will automatically insert a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error.

➢ **Statement**

Using basic Inheritance, create an Employee class that includes an employee's information and create a class named Programmer that extends it adding more information. To end it all, store multiple employees in an ArrayList and print their information to simulate a database.

➢ **Class design (UML)**



➢ Program Code

Employee.java

```
public class Employee {
	protected String name;
	protected int age;
	protected int salary;
	protected int ID;

	public Employee(String name, int age, int salary, int ID) {
		this.name = name;
		this.age = age;
		this.salary = salary;
		this.ID = ID;
	}

	public void printInfo() {
		System.out.println("\nID:       " + this.ID);
		this.employeeType();
		System.out.println("Name:     " + this.name);
		System.out.println("Age:      " + this.age);
		System.out.println("Salary:   " + this.salary + " pesos");
	}

	public static void employeeType(){
		System.out.println("Type:     General Employee");
	}

}
```

Programmer.java

```
public class Programmer extends Employee{
	private String language;

	public Programmer(String name, int age, int salary, int ID, String language) {
		super(name,age,salary,ID); //Invokes the parent class's constructor
		this.language =  language;
	}

	@Override
	public void printInfo(){  //Overrides the parent class's method
		super.printInfo();      //Invokes the overriden method
		System.out.println("Language: " + this.language);
	}

	public static void employeeType(){
```

```
                System.out.println("Type:     Programmer");
        }

        public String getLanguage() {
                return this.language;
        }

        public void setLanguage( String language ) {
                this.language = language;
        }
}
```

Data.java

```
import java.util.ArrayList;
import java.util.Scanner;

public class Data {

        public static void main(String[] args) {
                ArrayList<Employee> employee_list = new ArrayList<Employee>();
                int id = 0;

                employee_list.add(new Employee("Jose",29,10000,1));
                employee_list.add(new Programmer("Andre",21,8000,2,"java"));
                employee_list.add(new Programmer("Omar",20,2,3,"C++"));
                employee_list.add(new Employee("Zahid",21,90000,4));

                for ( Employee e : employee_list ) {
                        e.printInfo(); //Calls the printInfo() method of every Employee stored
                }
        }
}
```

➢ Program execution

In this example, we can see how inheritance allows us to organize connected classes and treat them as a same object while overriding and hiding help us modify the behavior in each of these classes even if they are treated as the same object.
In this case, both Programmer and Employee are treated as a same object Employee, but each has its own PrintInfo() method thanks to Overriding.

```
ID:       1
Type:     General Employee
Name:     Jose
Age:      29
Salary:   10000 pesos

ID:       2
Type:     General Employee
Name:     Andre
Age:      21
Salary:   8000 pesos
Language: java

ID:       3
Type:     General Employee
Name:     Omar
Age:      20
Salary:   2 pesos
Language: C++

ID:       4
Type:     General Employee
Name:     Zahid
Age:      21
Salary:   90000 pesos
```

➢ Conclusions

Overriding and Hiding are important aspects of Inheritance as they allow us to “mold” the parent class to better fit the behavior wanted in the derived class. These concepts are what give inheritance its power, and without them, inheritance wouldn't have the same flexibility it has right now. While inheritance helps us save time and resources, overriding and inheritance allow us to apply inheritance in places where we couldn't do otherwise.