Topic: Hiding and Overriding.

OOP concepts involved: Inheritance, Encapsulation, Abstraction and 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.

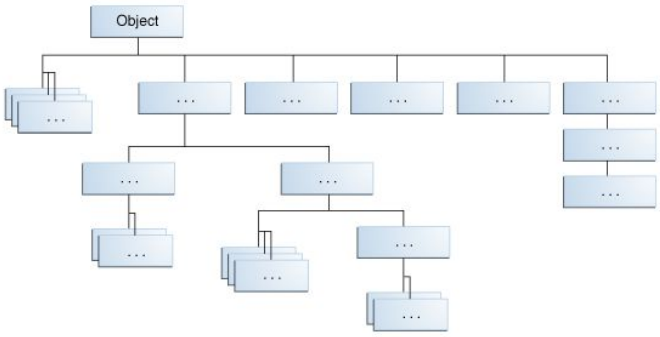


fig 1. Example of inheritance hierarchy tree

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:

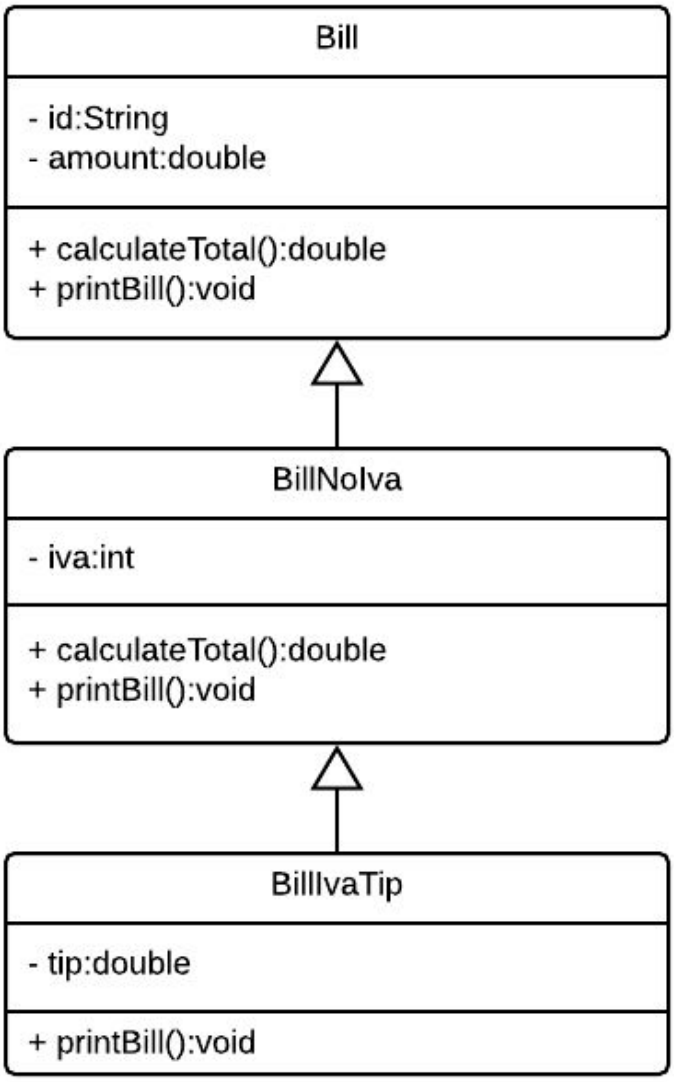
- **Use of super with fields:** This is done when a field in the parent class is hidden by a field in the subclass. The keyword super allows us to access the hidden fields of the parent class from the subclass.
- **Use of super with methods:** In this case, the super keyword allows us to access overridden 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**

Create a Bill class that includes the basic behavior of a generic Billing. Using inheritance, extend this Bill class into two other classes BillIva and BillIvaTip. Override the needed methods and fields to fit the behavior wanted for both of those classes.

➢ **Class design (UML)**



➢ Program Code

Bill.java

```
public abstract class Bill {
	private String id;
	private double amount;

	public Bill(String id, double amount){
		this.id = id;
		this.amount = amount;
	}

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public double getAmount() {
		return amount;
	}

	public void setAmount(double amount) {
		this.amount = amount;
	}

	public abstract double calculateTotal();

	public void printBill() {
		System.out.print("\n**********************************");
		System.out.println("\nBILL ID : $" + this.getId());
		System.out.println("AMOUNT  : $" + this.getAmount());
	}

}
```

BillIva.java

```
public class BillIva extends Bill {
	private int iva;

	public BillIva(String id, double amount, int iva){
		super(id,amount);
		this.iva = iva;
```

```
	}

	@Override
	public double calculateTotal() {
		return this.getAmount()*1.21;
	}

	@Override
	public void printBill() {
		super.printBill();
		System.out.println("IVA     : $" + (calculateTotal() - this.getAmount()) + "
(" + this.getIva() + "%)");
		System.out.println("TOTAL   : $" + calculateTotal());
	}

	public void setIva(int iva) {
		this.iva = iva;
	}

	public int getIva() {
		return this.iva;
	}

}
```

BillIvaTip.java

```
public class BillIvaTip extends BillIva{
	private double tip;

	public BillIvaTip(String id, double amount, int iva, double tip){
		super(id,amount,iva);
		this.tip = tip;
	}

	@Override
	public void printBill() {
		super.printBill();
		System.out.println("\nTIP     : $" + this.getTip());
		System.out.println("TOTAL W/TIP: $" + (this.calculateTotal()+this.getTip()));
	}

	public void setTip(double tip) {
		this.tip = tip;
	}

	public double getTip() {
		return this.tip;
```

```
	}
}
```

Test.java

```
public class Test {
	public static void main(String[] args) {
		BillIva b1 = new BillIva("0001",123,11);
		BillIvaTip b2 = new BillIvaTip("0002",234,11,23);

		b1.printBill();
		b2.printBill();
	}

}
```

➢ Program execution

During execution, we can create different bills each with one of two available behaviors. These derived classes overwrite the parent class's calculateTotal() method to change the way it calculates the final amount to be paid.

```
**********************************
BILL ID : $0001
AMOUNT  : $123.0
IVA     : $25.829999999999984 (11%)
TOTAL   : $148.82999999999998

**********************************
BILL ID : $0002
AMOUNT  : $234.0
IVA     : $49.139999999999986 (11%)
TOTAL   : $283.14

TIP     : $23.0
TOTAL W/TIP: $306.14
```

➢ 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.