Topic: Simple Inheritance.

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 behavior 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.

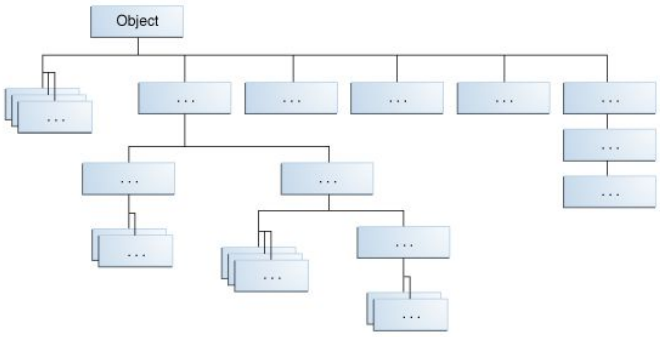


fig 1. Example of inheritance hierarchy tree

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{}
```

A subclass will inherit all *public* and *protected* members (fields, methods, and nested classes) from its superclass no matter what package the subclass is in. If the subclass is in the same package as its parent, it will also inherit all package-private members of the parent. One important observation is that Constructors are not members, so they are not inherited by subclasses, but they can still be invoked from the subclass. It's also important to note that extending an already derived class will also inherit all members from that class' superclass too.

What about private members?

When inheriting a class, we don't inherit the private members of its parent class. However, if this superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

Another way to get access to those private members is by inheriting a nested class. Since nested classes have access to all private members of its enclosing class, by inheriting a nested class, we get indirect access to all of the private members of the superclass.

What can you do with those inherited members?

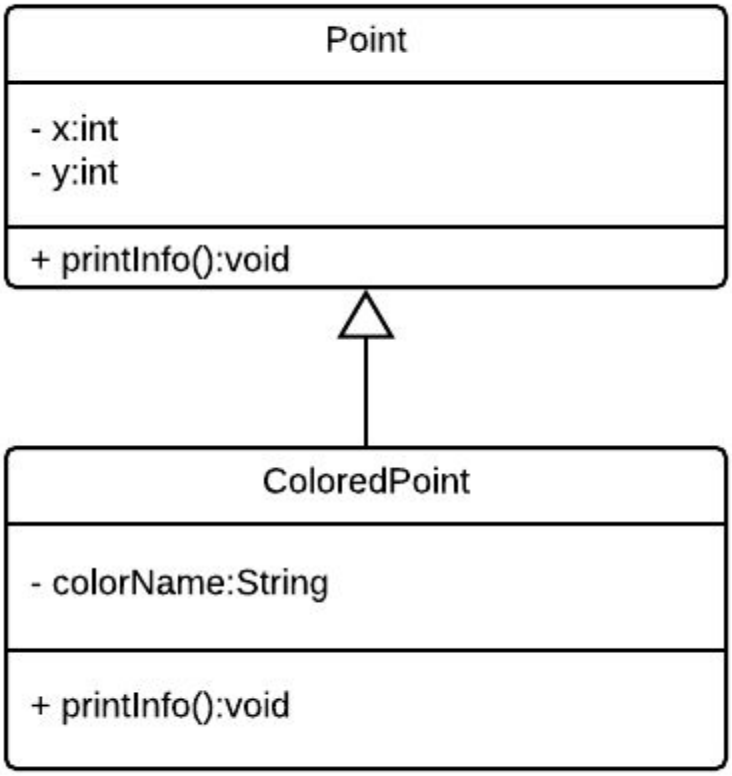
Subclasses allow us to do a variety of things to inherited members to mold them to our specific needs.

- Inherited fields can be used directly, just like any other fields
- You can declare new fields in the subclass with the same name as an already existing field in the parent class, thus *hiding* it. *(This is not recommended)*
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as one in the superclass, thus *overriding* it.
- You can write a new static method in the subclass that has the same name signature as one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword **super**.

➢ Statement

Create a Point class with its X and Y coordinate. Using what you learned about Simple Inheritance, create a derived class called ColoredPoint that adds the colorName field and prints their information.

➢ Class design (UML)



➢ Program Code

Point.java

```
public class Point {
	// fields marking X and Y position of the point
	private int x;
```

```
	private int y;

	public Point(int x, int y) {
		this.x = x;
		this.y = y;
	}

	public void printInfo(){
		System.out.println("with coordinates x=" + this.x + " y=" + this.y);;
	}

	// getter and setter methods
	public int getX() {
		return x;
	}
	public void setX(int x) {
		this.x = x;
	}
	public int getY() {
		return y;
	}
	public void setY(int y) {
		this.y = y;
	}

}
```

ColoredPoint.java

```
import java.util.Scanner;

public class ColoredPoint extends Point {

	// new field added to store the color name
	private String colorName;

	public ColoredPoint(int x, int y, String colorName) {
		super(x, y);
		this.colorName = colorName;
	}

	public void printInfo() {
		System.out.println("the color of the point is: " + this.colorName);
		super.printInfo();
	}

	public String getColorName() {
		return colorName;
	}
```

```
	public void setColorName(String colorName) {
		this.colorName = colorName;
	}
}
```

Test.java

```
import java.util.Scanner;

public class Test {
	public static void main(String[] args) {
		Point point;
		int op;
		int x;
		int y;
		Scanner sc = new Scanner(System.in);

		System.out.println("Regular point or Colored Point?");
		System.out.println("[1] Regular Point");
		System.out.println("[2] Colored Point");
		System.out.println("Selection: ");
		op = Integer.parseInt(sc.nextLine());
		System.out.print("X: ");
		x = Integer.parseInt(sc.nextLine());
		System.out.print("y: ");
		y = Integer.parseInt(sc.nextLine());
		if (op == 1) {
			point = new Point(x,y);
		} else {
			System.out.println("Color?:");
			point = new ColoredPoint(x,y,sc.nextLine());
		}
		point.printInfo();
	}
}
```

➢ **Program execution**

In this example, we can create 2 different kinds of points using simple inheritance. This allows us to extend the simple point and just code the added behavior for the ColoredPoint. While this example may not be that complicated, this same principle applies to more complex examples.

```
Regular point or Colored Point?
[1] Regular Point
[2] Colored Point
Selection:
2
X: 12
y: 23
Color?:
Red
the color of the point is: Red
with coordinates x=12 y=23
```

➢ **Conclusions**

Inheritance is an extremely useful tool that helps to reduce code duplication. With the code written in the parent class, we no longer have to worry about rewriting it on multiple child classes that have the same or a similar behavior. In this way, Inheritance saves us both time and resources when coding.