Inheritance

In real life you can observe inheritance almost everywhere. A child for example takes on the characteristics of parents. It is so in OOP also. This is one of the most powerful feature of OOP.

Inheritance is the concept of a child class (subclass) automatically inheriting the variables and methods defined in its parent class (superclass). It is the mechanism through which a class can be defined in terms of an existing class.

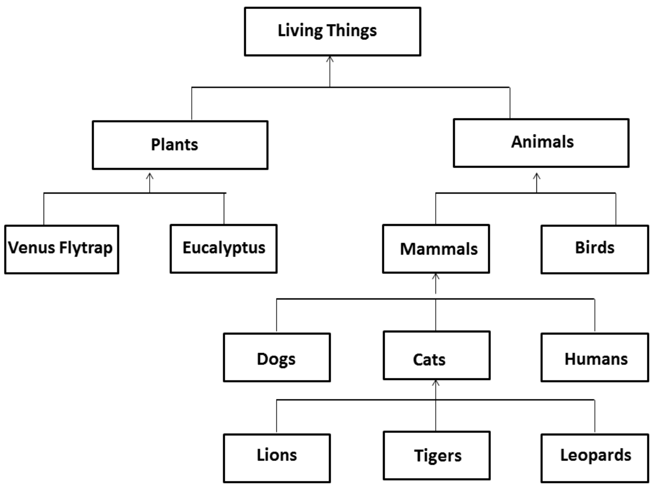


Figure 1: Inheritance Example

Inheritance enables the refinement or specialization of an existing class. Through inheritance a child can acquire characteristics of its parents and can also have its own features.

In the above diagram plant 'Venus Flytrap' not only inherits the characteristics of plants and also has features by which it traps and eats insects.

Importance of inheritance

- The benefit of inheritance in OOP is reusability. Once a behavior (method) is defined in a superclass, that behavior is automatically inherited by all subclasses.
- Once a set of properties (fields) are defined in a superclass, the same set of properties are inherited by all subclasses. A class and its children share common set of properties.
- A subclass only needs to implement the differences between itself and the parent.

The class that inherits a set of attributes is called as **subclass** or **derived class** and the class from which it inherits is called as **superclass** or **parent class**. We can define inheritance is an OOP term, in which the non-private features and attributes of a given superclass are made available to its subclass(es).

A subclass acquires the properties from superclass using a keyword **extends**.

Deriving a subclass

- To derive a child class, you use the **extends** keyword.
- Suppose you have a parent class called Person.

```
public class Person {
    protected String name;
    protected String address;
    /**
     * Default constructor
    */
    public Person(){
        System.out.println ("Inside Person:Constructor");
        name = "";
        address = "";
    }
    // methods
}
```

- Now, create another class named Student.
- Since a student is also a person, just extend the class ‘Person’, so that ‘Student’ class can inherit all the properties and methods of the existing class Person.
- To do this, use *extends* keyword and write as below,

```
public class Student extends Person {
    public Student(){
        System.out.println ("Inside Student:Constructor");
    }
    // methods
}
```

Inheritance defines **is a** relationship between the classes.

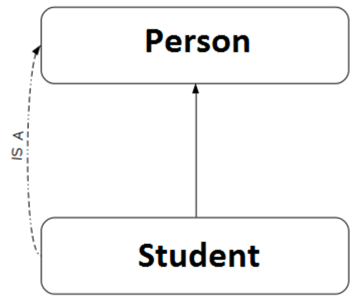


Figure 2: Single Inheritance

From the above example, we can say *Student* **is a** *Person*.

What a subclass can do

- A subclass inherits all of the public and protected members (fields or methods) of its parent, no matter what package the subclass is in.
- If the subclass is in the same package as its parent, then it also inherits the package-private members (fields or methods) of the parent.

What a sub-class can do regarding fields

- The inherited fields can be used directly, just like any other fields.
- Can declare new fields in the subclass that are not in the superclass.
- Can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be applied by the subclass.

What a subclass can do regarding methods

- The inherited methods can be used directly as they are.

- Can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- Can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- Can declare new methods in the subclass that are not in the superclass.

Types of inheritance

Java can implement inheritance in the following types.

Single/Simple Inheritance

Single inheritance enables a derived class to inherit properties and behavior from a single base class.

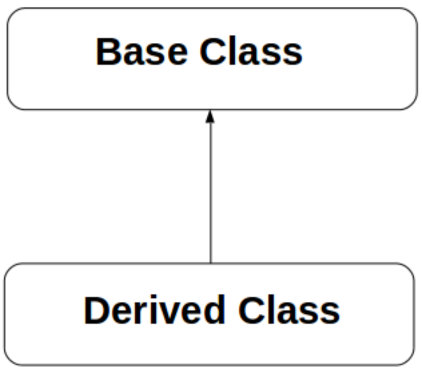


Figure 3: Single Inheritance

As it allows a derived class to inherit the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code. This makes the code much more elegant and less repetitive.

Multilevel Inheritance

Multilevel inheritance enables a derived class to inherit properties and behavior from another derived class, which is inherited from a base class.

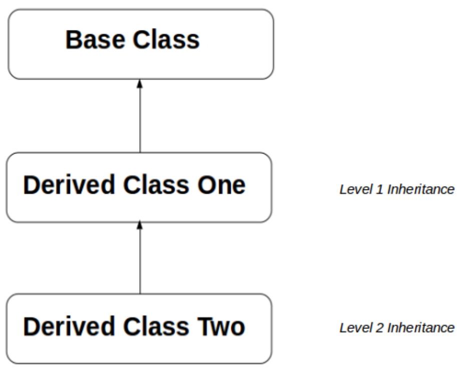


Figure 4: Multilevel Inheritance

In this process inheritance builds different levels, thats why this type of inheritance is called as *multilevel inheritance*.

Hierarchical Inheritance

Hierarchical inheritance enables a base class to inherit its properties and behavior to multiple derived classes.

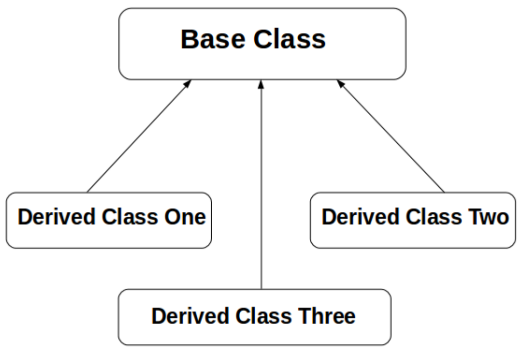
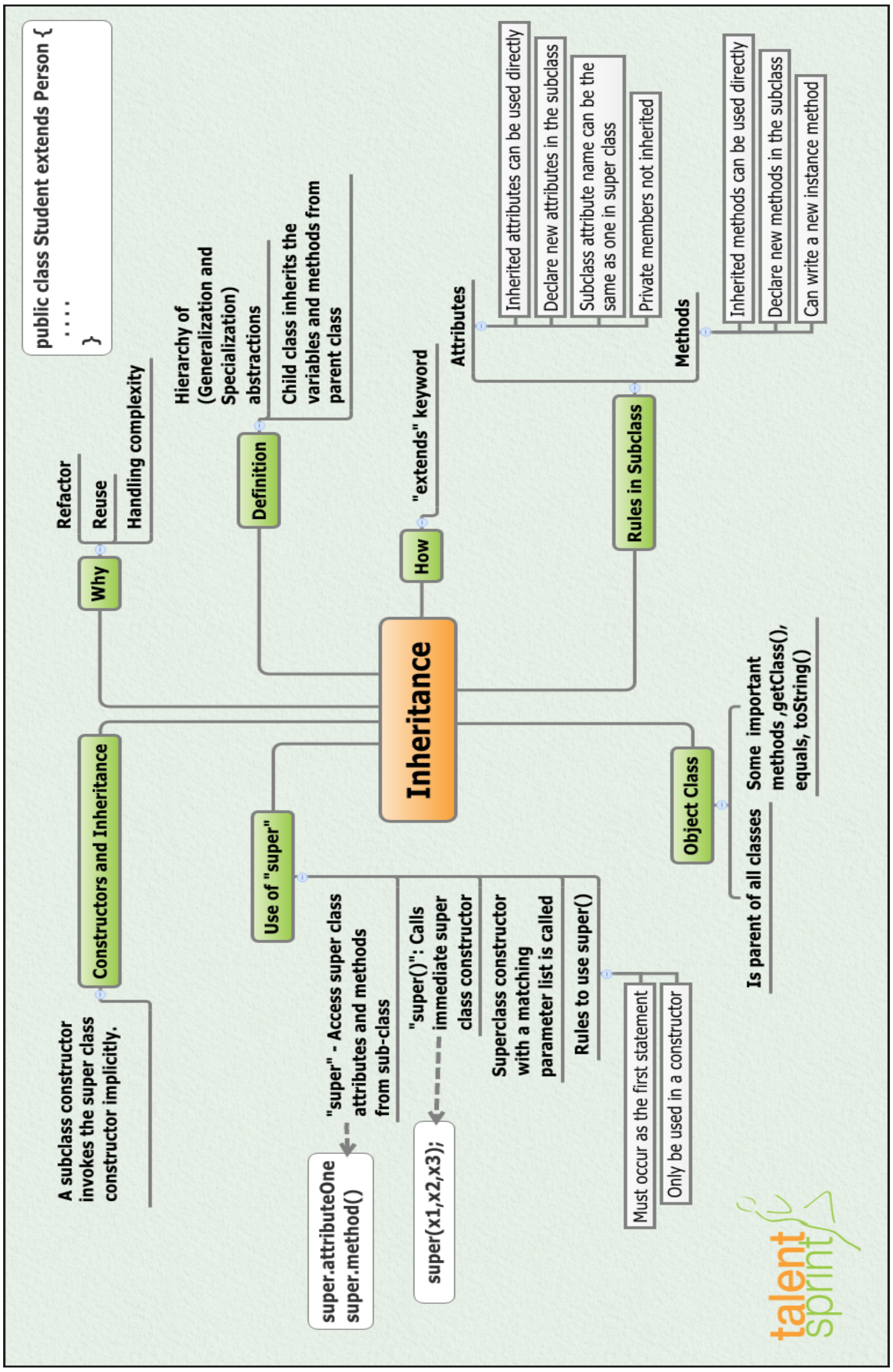


Figure 5: Hierarchical Inheritance Example


Inheritance
Why
Refactor
Reuse
Handling complexity
public class Student extends Person {
. . . .
}
Definition
Hierarchy of (Generalization and Specialization) abstractions
Child class inherits the variables and methods from parent class
How
"extends" keyword
Rules in Subclass
Attributes
Inherited attributes can be used directly
Declare new attributes in the subclass
Subclass attribute name can be the same as one in super class
Private members not inherited
Methods
Inherited methods can be used directly
Declare new methods in the subclass
Can write a new instance method
Constructors and Inheritance
A subclass constructor invokes the super class constructor implicitly.
Use of "super"
"super" - Access super class attributes and methods from sub-class
super.attributeOne
super.method()
"super()": Calls immediate super class constructor
super(x1,x2,x3);
Superclass constructor with a matching parameter list is called
Rules to use super()
Must occur as the first statement
Only be used in a constructor
Object Class
Is parent of all classes
Some important methods ,getClass(), equals, toString()

‘super’ keyword

In base class members are overriden by its derived class, hence base class members are hidden on accessing.

To refer the overriden members of base class, in Java **super** keyword is used.

Assume the below class is base class.

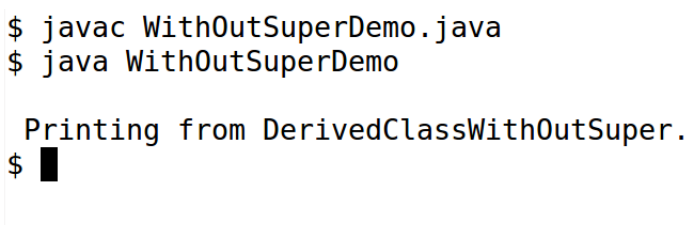
```
class BaseClass {
    public void display () {
        System.out.println("\nPrinting from BaseClass.");
    }
}
```

Compile the above program

```
$ javac BaseClass.java
```

Example without using super

```
class DerivedClassWithOutSuper extends BaseClass {
    public void display () {
      System.out.println(" Printing from DerivedClassWithOutSuper.");
    }
}
public class WithOutSuperDemo {
    public static void main(String[] args) {
        /* Creating an object for DerivedClassWithOutSuper. */
        DerivedClassWithOutSuper derivedClass
                                   = new DerivedClassWithOutSuper();
        // calling display method.
        derivedClass.display();
    }
}
```

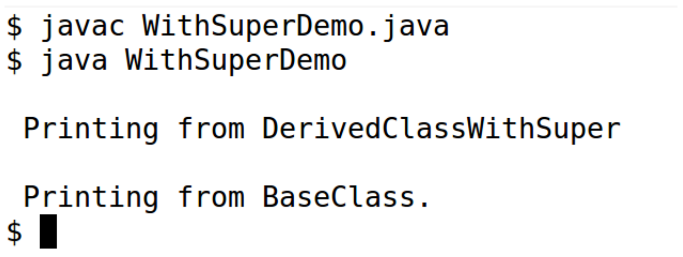


```
$ javac WithOutSuperDemo.java
$ java WithOutSuperDemo

 Printing from DerivedClassWithOutSuper.
$ █
```

Output: WithOut Super

Example using super

```
class DerivedClassWithSuper extends BaseClass {
   public void display () {
     System.out.println ("Printing from DerivedClassWithSuper");
     super.display ();
   }
}

public class WithSuperDemo {
    public static void main (String [] args) {
        /* Creating an object for DerivedClassWithSuper. */
        DerivedClassWithSuper derivedClass
                                  = new DerivedClassWithSuper ();
        // calling display method.
        //Displays from DerivedClass
        derivedClass . display ();
    }
}
```



```
$ javac WithSuperDemo.java
$ java WithSuperDemo

 Printing from DerivedClassWithSuper

 Printing from BaseClass.
$ 
```

Output: With Super

super as a base class constructor

super keyword is also used to invoke base class constructor from derived class.

Example for super used as a constructor.

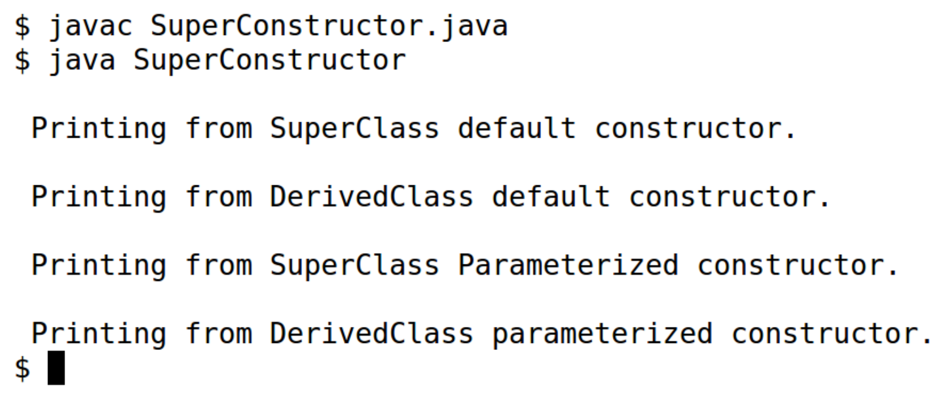
superclass definition

```
class SuperClass {
    private String name = null;
    // Default constructor
    public SuperClass () {
        System.out. println ('' Printing from SuperClass '' +
                             ''default constructor'');
    }

   // Parameterized constructor.
    public SuperClass (String name) {
        System.out. println ('' Printing from SuperClass'' +
                                ''Parameterized constructor'');
    }
}
```

derived class definition

```
class DerivedClass extends SuperClass {

  public DerivedClass () {
      super ();
```

```
        System.out.println("Printing from DerivedClass" +
                           "default constructor");
    }

    public DerivedClass (String name) {
        super (name);
        System.out.println("Printing from DerivedClass" +
                           " parameterized constructor");
    }
}

public class SuperConstructor {

    public static void main (String[] args) {
        /* Creating an object with default constructor. */
        DerivedClass derivedClassOne
                     = new DerivedClass ();
        /* Creating an object with parameterized constructor. */
        DerivedClass derivedClassTwo
                     = new DerivedClass ("TalentSprint");
    }
}
```



```
$ javac SuperConstructor.java
$ java SuperConstructor

 Printing from SuperClass default constructor.

 Printing from DerivedClass default constructor.

 Printing from SuperClass Parameterized constructor.

 Printing from DerivedClass parameterized constructor.
$ 
```

Output: SuperConstructor

Polymorphism

The dictionary definition of **polymorphism** refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming.

Java supports two types of polymorphism

Static Polymorphism

In Java, static polymorphism is achieved through **method overloading**.
A Java class can define more than one methods with same name, that approach is called as **method overloading** and also known as **Static / Compile-Time Polymorphism**.

Method overloading

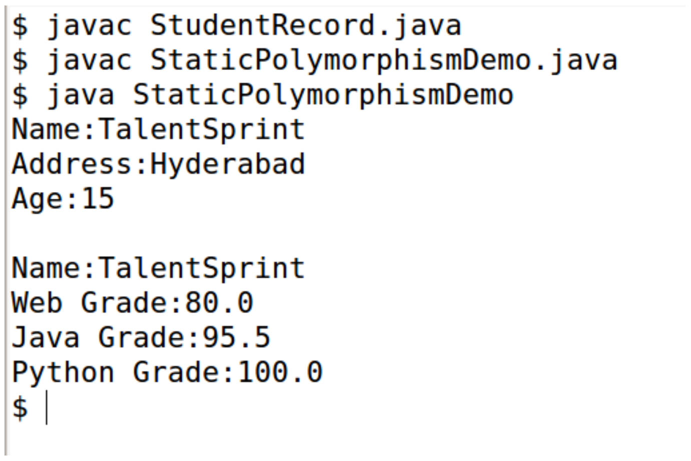
- Allows a method with the same name but different parameters, to have different implementations and return values of different types
- Can be applied when the same operation has different implementations
- Always remember that overloaded methods have the following properties:
  1. The same method name
  2. Different parameters or different number of parameters
  3. Return types can be different or same

Assume a class '*StudentRecord*' defines overloaded methods

```
public class StudentRecord {
    String name;
    String address;
    int age;
    double pythonGrade;
    double javaGrade;
    double webGrade;

    \\ setter methods here
    \\ getter methods here

    public void display ( String temp ){
        System.out.println ("Name:" + name);
        System.out.println ("Address:" + address);
        System.out.println ("Age:" + age);
    }
    public void display (double pGrade, double jGrade,
                                double wGrade) {
        System.out.println ("\nName:" + name);
        System.out.println ("Web Grade:" + wGrade);
        System.out.println ("Java Grade:" + jGrade);
        System.out.println ("Python Grade:" + pGrade);
    }
}
```

Set the values and run '*StudentRecord*' class by creating instance.

```
public class StaticPolymorphismDemo {
    public static void main( String[] args ) {
        StudentRecord tsRecord = new StudentRecord();
        tsRecord.setName("TalentSprint");
        tsRecord.setAddress ("Hyderabad");
        tsRecord.setAge(15);
        tsRecord.setWebGrade(80);
        tsRecord.setJavaGrade(95.5);
        tsRecord.setPythonGrade(100);

        //overloaded methods
        tsRecord.display( tsRecord.getName() );
        tsRecord.display( tsRecord.getPythonGrade(),
            tsRecord.getJavaGrade(), tsRecord.getWebGrade());
    }
}
```



```
$ javac StudentRecord.java
$ javac StaticPolymorphismDemo.java
$ java StaticPolymorphismDemo
Name:TalentSprint
Address:Hyderabad
Age:15

Name:TalentSprint
Web Grade:80.0
Java Grade:95.5
Python Grade:100.0
$ |
```

Dynamic Polymorphism

In Java subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

In Java, dynamic polymorphism is achieved through method overriding.

Having the same method of base class in its derived class is called as **Method Overriding**.

Assume that 'Animal' is a super class that contains a method 'speak()'.

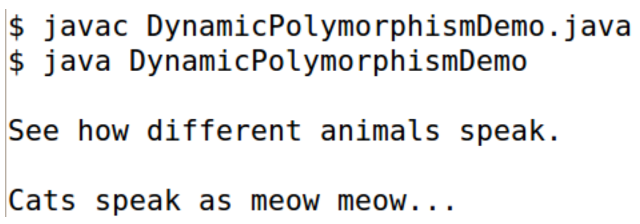
```
class Animal {
    @Override
    public void speak () {
        System.out. println (" See how different animals speak.");
    }
}
```

A derived class 'Cat' inherits 'Animal' class and overrides 'speak()' method.

```
class Cat extends Animal {
    @Override
    public void speak () {
        System.out. println (" Cats speak as meow meow...");
    }
}
```

The below code snippet shows how dynamic polymorphism works.

```
public class DynamicPolymorphismDemo {
    public static void main (String [] args) {
        /* Creating a super class reference variable , animal. */
        Animal animal;
        animal = new Animal();
        animal.speak();
            // Creates Cat object
        Cat cat = new Cat();
        // Cat object is assigned to animal
        animal = cat;
        // animal calls speak method of cat.
        animal.speak();
    }
}
```

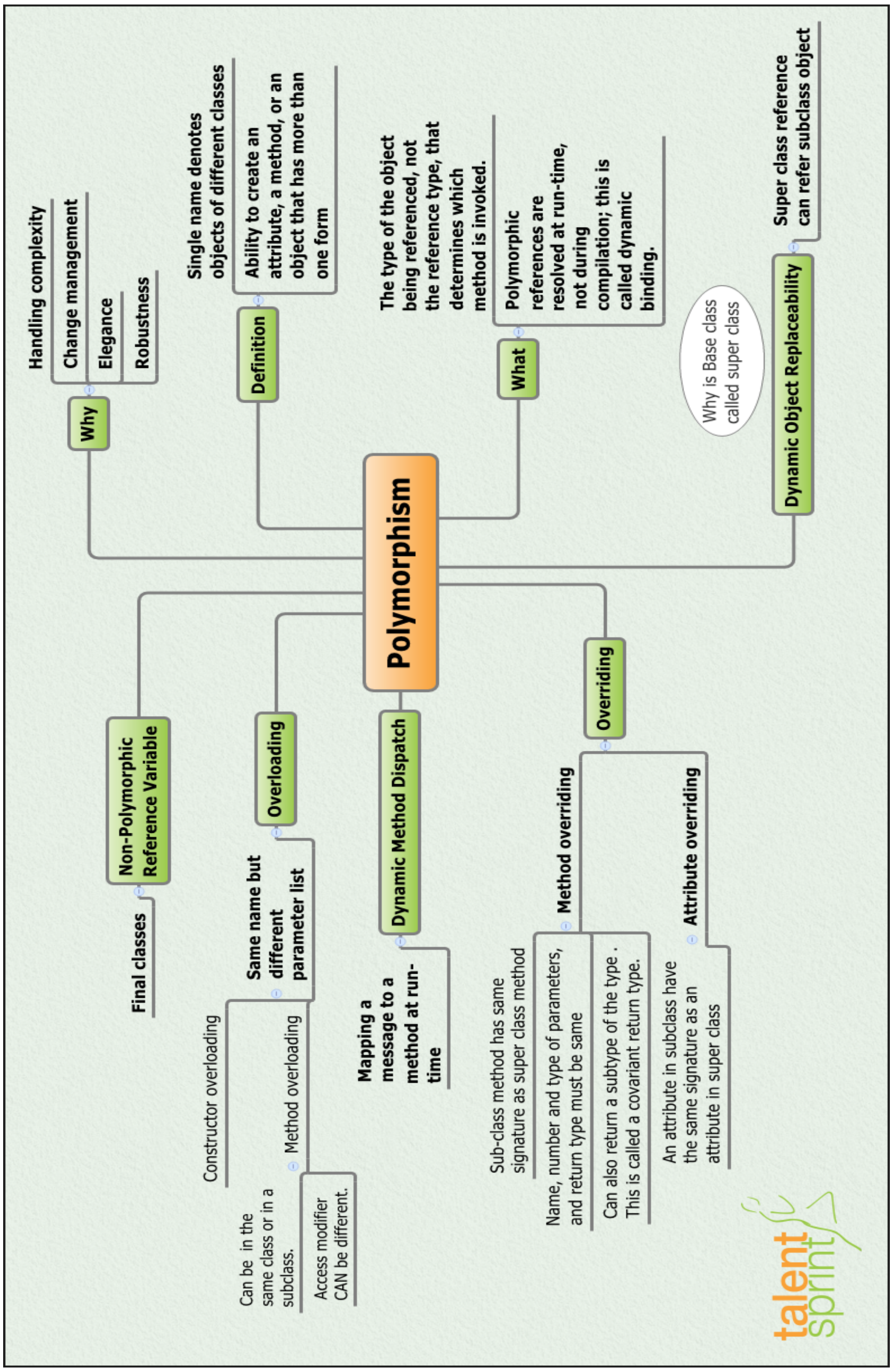


```
$ javac DynamicPolymorphismDemo.java
$ java DynamicPolymorphismDemo

See how different animals speak.

Cats speak as meow meow...
```

The above defined polymorphism is also known as **Dynamic/Runtime Polymorphism**. Both the classes (Animal and Cat) have a common method *speak()*.

Difference between Overloading and Overriding

Overloading	Overriding
Signature has to be different. Just a difference in return type is not enough.	Signature has to be the same. (including the return type)
Accessibility may vary freely.	Overriding methods cannot be more private than the overridden methods.
Just the name is reused. Methods are independent methods. Resolved at compile-time based on method signature.	Related directly to sub-classing. Overrides the parent class method. Resolved at run-time based on type of the object.
Can call each other by providing appropriate argument list.	Overriding method can call overridden method by super.methodName() , this can be used only to access the immediate method of superclass. super.super will not work. Also, a class outside the inheritance hierarchy cannot use this technique.
Methods can be static or non-static. If two methods have the same signature, declaring one as static and another as non-static does not provide a valid overload. Its a compile time error.	static methods do not participate in overriding, since they are resolved at compile time based on the type of reference variable. A static method in a sub-class cannot use super. A static method cannot be overridden to be non-static and vice-versa.
There is no limit on number of overloaded methods a class can have.	Each parent class method may be overridden at most once in any sub-class.


Polymorphism
Why
Handling complexity
Change management
Elegance
Robustness
Definition
Single name denotes objects of different classes
Ability to create an attribute, a method, or an object that has more than one form
What
The type of the object being referenced, not the reference type, that determines which method is invoked.
Polymorphic references are resolved at run-time, not during compilation; this is called dynamic binding.
Why is Base class called super class
Dynamic Object Replaceability
Super class reference can refer subclass object
Non-Polymorphic Reference Variable
Final classes
Overloading
Same name but different parameter list
Constructor overloading
Method overloading
Can be in the same class or in a subclass.
Access modifier CAN be different.
Dynamic Method Dispatch
Mapping a message to a method at run-time
Overriding
Method overriding
Sub-class method has same signature as super class method
Name, number and type of parameters, and return type must be same
Can also return a subtype of the type . This is called a covariant return type.
Attribute overriding
An attribute in subclass have the same signature as an attribute in super class