Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Inheritance

(IT069IU)

Nguyen Trung Ky

ntky@hcmiu.edu.vn

it.hcmiu.edu.vn/user/ntky

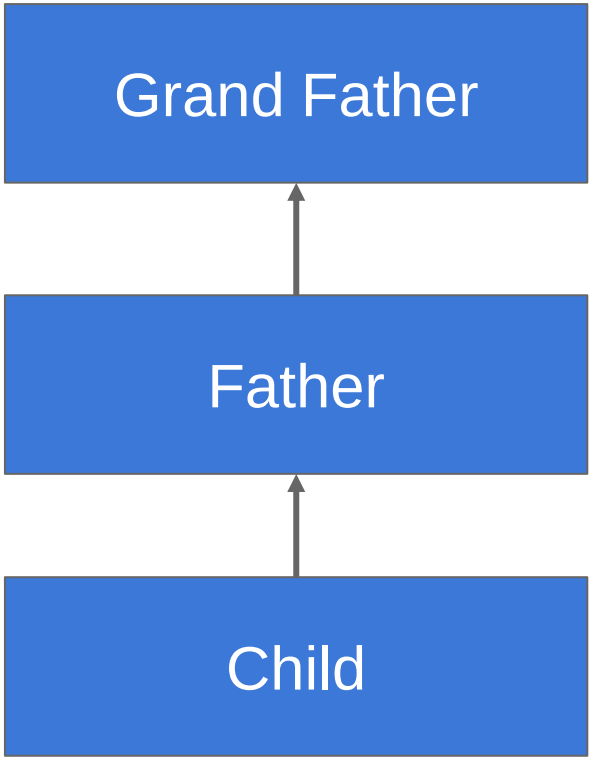
Previously

- **Control flow statements**:
  - Decision making statements:
    - **If**
    - **If...else**
    - **Switch**
  - Loop statements:
    - **While**
    - **Do while**
    - **For**
  - Jump statements:
    - **Break** statement
    - **Continue** statement
- **Array:**
  - Declare and Create Array
  - Loop through Array
  - Pass Arrays to Methods
    - Pass by Value vs Pass by Reference
  - Class Arrays for helper methods

Agenda today

- **Inheritance**
  - Definition and Examples
  - Types of Inheritance
  - UML Diagram
  - Animal Inheritance Example
    - Without Inheritance
    - With Inheritance
    - Method Overriding
  - Constructors in Subclasses
    - Keyword Super
  - Method Overriding
    - Keyword Super
  - Access Modifier (Protected)
- **Overloading**
  - Method Overloading
  - Constructor Overloading
- **Final** Keyword
  - Constant Variable
- **Static** Keyword
  - Static Variable
  - Static Method

Inheritance



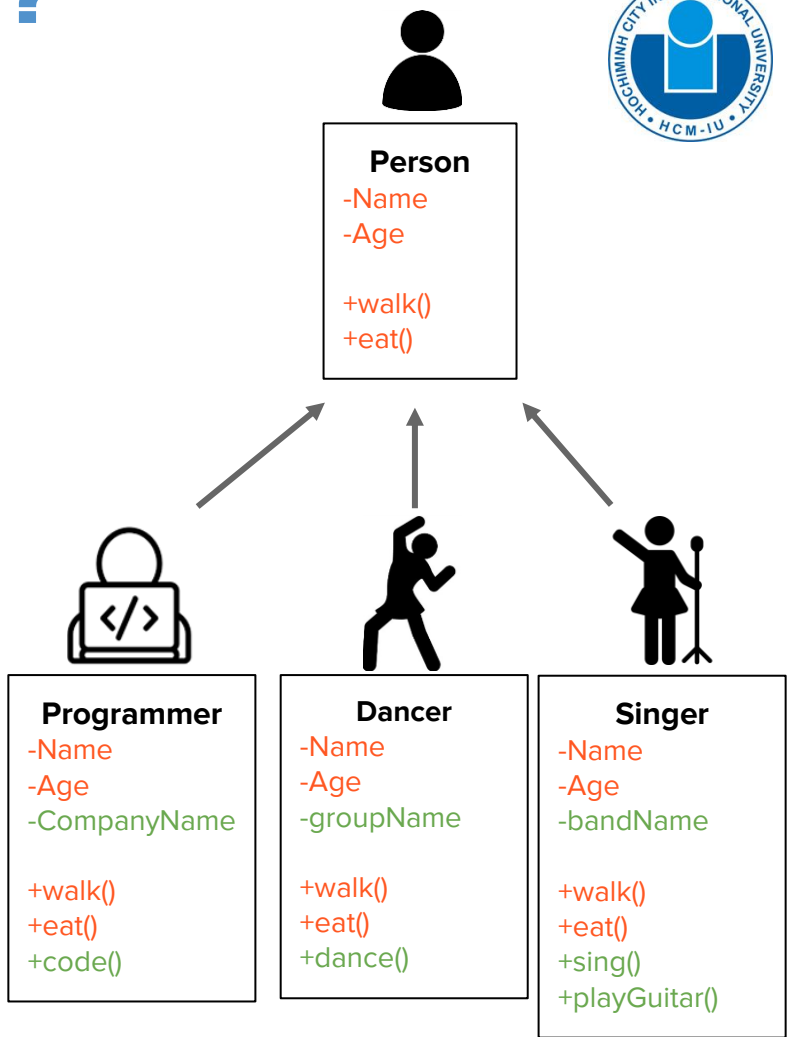
What is Inheritance in OOP?

- Inheritance is **the procedure in which one class inherits the attributes and methods of another class**. The class whose properties and methods are inherited is known as the Parent class (**superclass**).

- Inheritance is a way to **reuse classes by expanding them into more specific types of classes.**

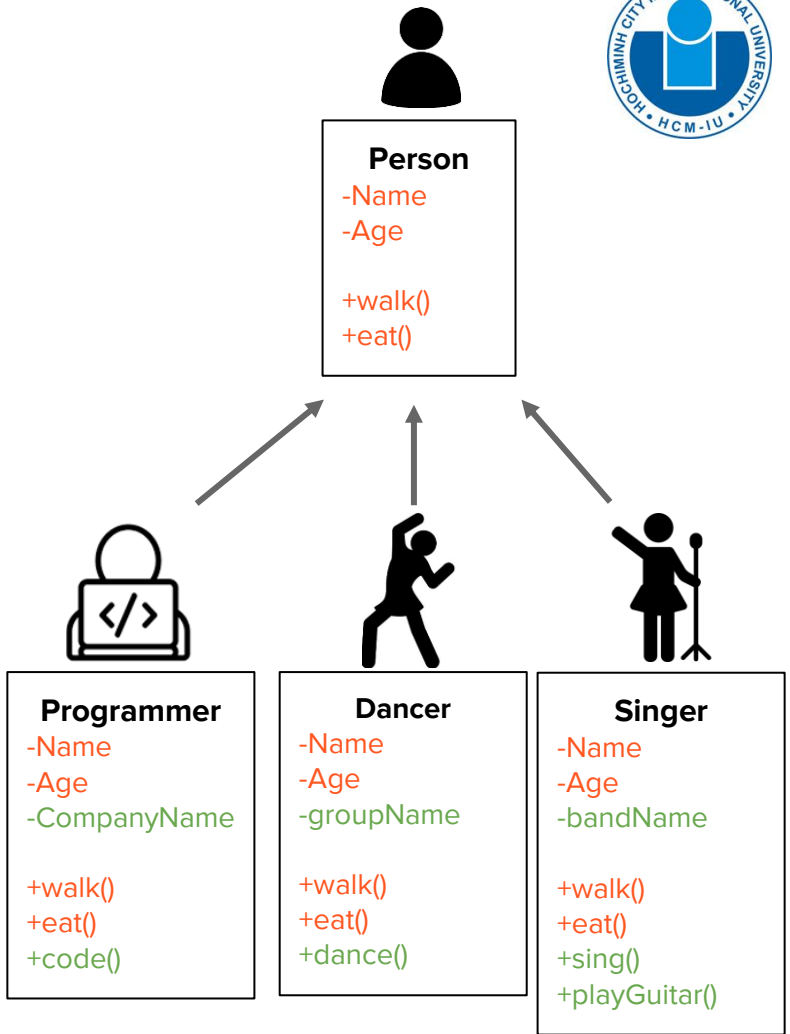
- Inheritance allows a **child class** (**subclass**) to **inherit the attributes and the methods** of a **parent class** (**superclass**). So **a child class can do anything that the parent class can do**!
- A **child class**
  - can have **its own attributes and methods.**
  - A child class can customize methods that it inherits from its parent class (**method overriding**)

- Inheritance represents **IS-A relationship** between parent and child objects.



Why we need Inheritance?

- Inheritance promotes the idea of **code reuse** to **reduces code repetition** as classes can share similar common logic, structure, attributes and methods.

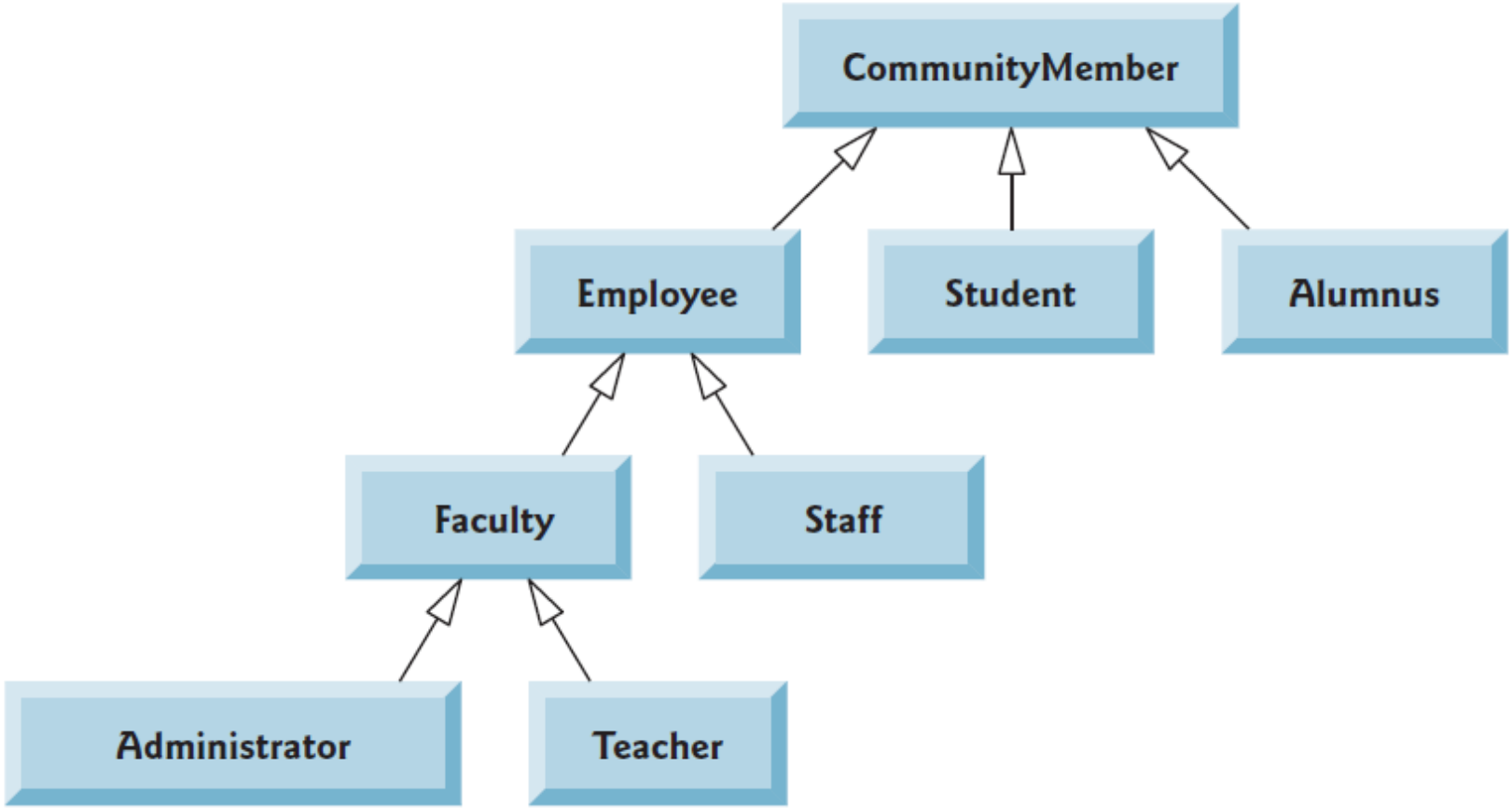


Examples of Superclass, Subclasses

- **Superclass** tend to be “more general” and **Subclass** is more “more specific.”

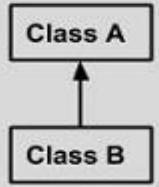
Parent Class (Superclass)	Child Class (Subclasses)
Vehicles	Car, Truck, Boat, Bicycle
Shape	Circle, Triangle, Rectangle, Cube
UniversityStaff	Lecturer, TeachingAssistant
Animal	Dog, Cat, Spider, Duck

UML Class Diagram for CommunityMembers



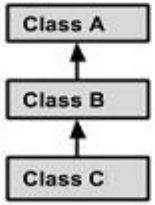
Types of Inheritance & Syntax

Single Inheritance



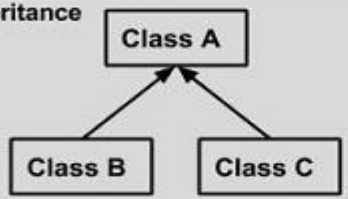

```
public class A {
   .....
}
public class B extends A {
   .....
}
```

Multi Level Inheritance



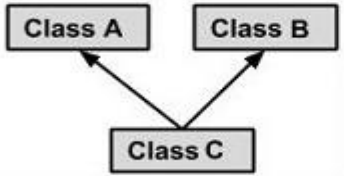

```
public class A { ..................}
public class B extends A {.....................}
public class C extends  B {..................... }
```

Hierarchical Inheritance




```
public class A { ..................}
public class B extends A {.....................}
public class C extends A {..................... }
```

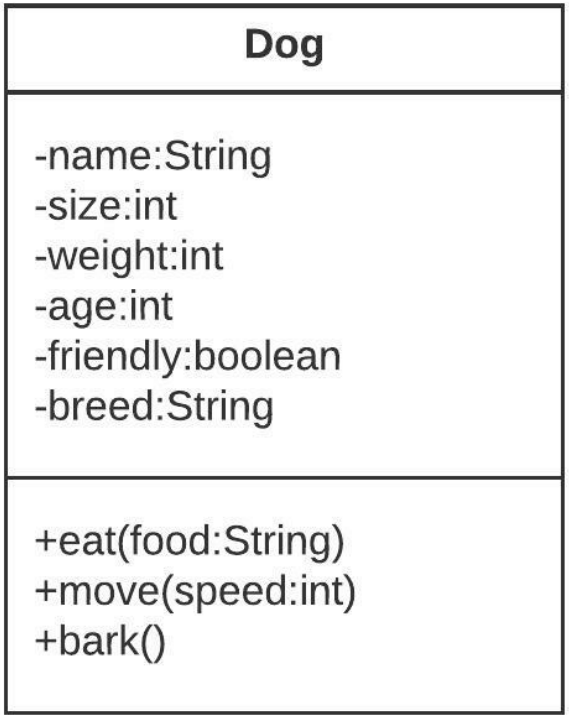
Multiple Inheritance

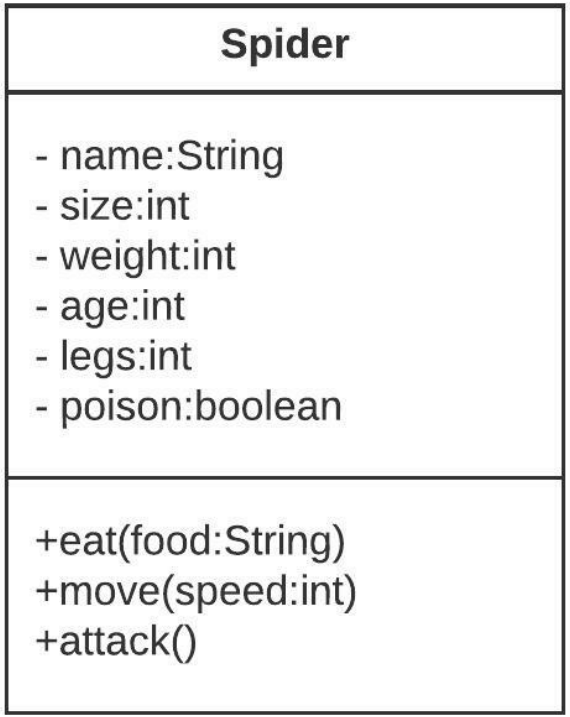



```
public class A { ..................}
public class B {.....................}
public class C extends  A,B {
   .....................
} // Java does not support mutiple Inheritance
```

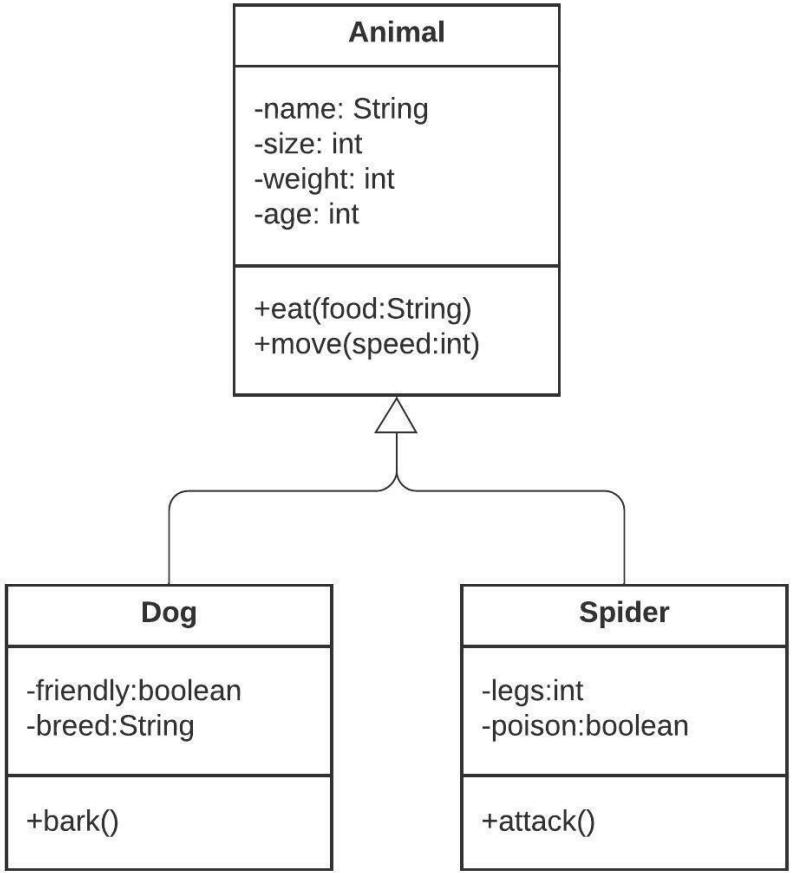
Java does not support multiple inheritance because the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Without Inheritance





With Inheritance



Animal Inheritance Example

Let's live code in Java!

Animal Class

[Info] Did you notice the superclass Animal looks like the normal class with attributes and method.

```java
public class Animal {
    private String name;
    private int size;
    private int weight;
    private int age;

    public Animal(String name, int size, int weight, int age) {
        this.name = name;
        this.size = size;
        this.weight = weight;
        this.age = age;
    }

    public void eat(String food){
        System.out.printf("The %s is eating %s!\n", getName(), food);
    }

    public void move(int velocity){
        System.out.printf("The %s is moving %d km/h!\n", getName(), velocity);
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSize() {
        return size;
    }
    public void setSize(int size) {
        this.size = size;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

Dog Class

[Question] Can you guess what does the keyword super do in the constructor of Dog Class?

```
public class Dog extends Animal {
    private boolean friendly;
    private String breed;

    public Dog(String name, int size,
               int weight, int age,
               boolean friendly, String breed) {
        super(name, size, weight, age);
        this.friendly = friendly;
        this.breed = breed;
    }

    public void bark(){
        System.out.println("The dog is barking!");
    }

    public boolean isFriendly() {
        return friendly;
    }

    public void setFriendly(boolean friendly) {
        this.friendly = friendly;
    }

    public String getBreed() {
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }
}
```

Spider Class

[Question] Can you guess what does the keyword super do in the constructor of Spider Class?

```
public class Spider extends Animal {
    private int legs;
    private boolean poison;

    public Spider(String name, int size,
                  int weight, int age,
                  int legs, boolean poison) {
        super(name, size, weight, age);
        this.legs = legs;
        this.poison = poison;
    }

    public void attack(){
        System.out.println("The spider is attacking!");
    }

    public int getLegs() {
        return legs;
    }

    public void setLegs(int legs) {
        this.legs = legs;
    }

    public boolean isPoison() {
        return poison;
    }

    public void setPoison(boolean poison) {
        this.poison = poison;
    }
}
```

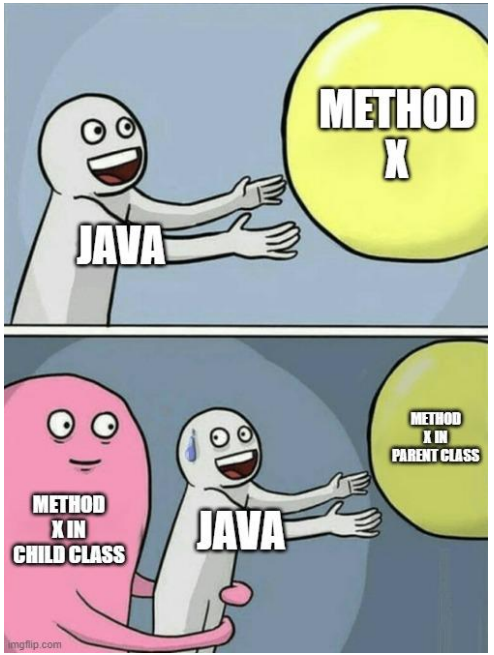
Main Class for Testing

```
public class Zoo {

    public static void main(String[] args) {
        Dog myDog = new Dog("Kiki", 200, 10, 5, true, "Corgi");
        Spider mySpider = new Spider("Spider-man", 2, 3, 20, 20, false);

        myDog.eat("sauces");
        myDog.move(20);
        myDog.bark();
        System.out.println(myDog.getAge());

        mySpider.eat("sauces");
        mySpider.move(20);
        mySpider.setName("Venom");
        System.out.println(mySpider.getName());
    }
}
```

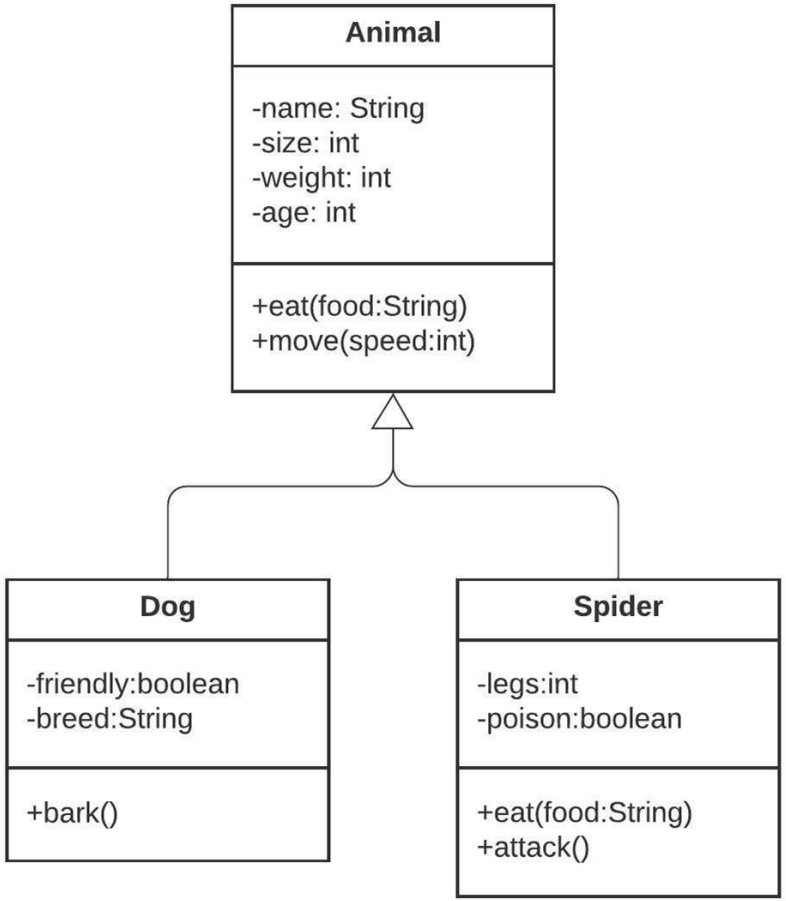
```
The Kiki is eating sauces!
The Kiki is moving 20 km/h!
The dog is barking!
5
The Spider-man is eating insects!
The Spider-man is moving 3 km/h!
Venom
```

Method Overriding

- If **subclass** (child class) **has the same method as declared in the superclass** (parent class), it is known as **method overriding** in Java.
- Method overriding is used to **provide a different implementation of a method which is already provided by its superclass.**
- **Rules** for method overriding:
  - Method must have the **same name** as in the parent class.
  - Method must have **same parameter** as in the parent class.
  - Method overriding must **happens in IS-A relationship (inheritance).**



Method Overriding Example



[Question] Can you find out which method of which class is overridden?

Method Overriding in Spider Class

Spider.java

```java
public class Spider extends Animal {
    private int legs;
    private boolean poison;

    public Spider(String name, int size,
                  int weight, int age,
                  int legs, boolean poison) {...}

    @Override
    public void eat(String food){
        System.out.printf("Spider is capturing the %s first before eating it!", food);
    }
```

[Question] @Override is optional, but why does we want it when we override any class?

Zoo.java

```java
public class Zoo {
    public static void main(String[] args) {
        Spider mySpider = new Spider("Spider-man", 2, 3, 20, 20, false);
        mySpider.eat("insects");
    }
}
```

```
Spider is capturing the insects first before eating it!
```

Keyword super(): definition and usage

- The super keyword refers to superclass (parent) objects.
- It is used to call **superclass methods**, and to access the **superclass constructor**.
- The most common use of the super keyword is to eliminate the confusion between **superclasses** and **subclasses** that have methods with the same name.

Keyword super() in a constructor in child class

- A child class, all properties, instance variables, and methods are inherited, except for constructors, so **you need to define constructors for child class**.
- The **keyword super in a constructor in a child class can be used to call a constructor in the parent class**. It's a way to delegate the responsibility to the parent class.

```
public class Animal {
    private String name;
    private int size;
    private int weight;
    private int age;

    public Animal(String name, int size, int weight, int age) {
        this.name = name;
        this.size = size;
        this.weight = weight;
        this.age = age;
    }

}
```

```
public class Spider extends Animal {
    private int legs;
    private boolean poison;

    public Spider(String name, int size,
                  int weight, int age,
                  int legs, boolean poison) {
        super(name, size, weight, age);
        this.legs = legs;
        this.poison = poison;
    }
```

Keyword super() in a method in child class

- Also, the keyword super can be used in a method of child class to call a method of a parent class even if that method is overridden.

Animal.java

```java
public class Animal {
    private String name;
    private int size;
    private int weight;
    private int age;

    public Animal(String name, int size, int weight, int age) {...}

    public void eat(String food) {
        System.out.printf("The Animal is eating %s!\n", food);
    }
}
```

Spider.java

```java
public class Spider extends Animal {
    private int legs;
    private boolean poison;

    public Spider(String name, int size,
                  int weight, int age,
                  int legs, boolean poison) {...}

    @Override
    public void eat(String food) {
        System.out.printf("Spider is capturing the %s!\n", food);
        super.eat(food);
    }
}
```

Zoo.java

```java
public class Zoo {
    public static void main(String[] args) {
        Spider mySpider = new Spider("Spider-man", 2, 3, 20, 20, false);
        mySpider.eat("human");
    }
}
```

Output:

```
Spider is capturing the human!
The Animal is eating human!
```

Protected Members

- Remember, apart from public and private, we have “protected” to be an access modifier
- “Protected” access offers an intermediate level of access between public and private.
- **A superclass’s protected members** can be accessed by **members of that superclass**, by **members of its subclasses** and by **members of other classes** in the same package.
- In short, the “protected” access modifier means that **anything within the class** can use it, as well as **anything in any subclass**.

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

The 'protected' Access Modifier

In the past, we discussed the private and public access modifiers. To review, remember that **public meant anyone could get access to the member** (variable, method, property, etc.), while **private means that you only have access to it from inside of the class that it belongs to**.

With inheritance we add **another option: protected**. If a member of the class uses the protected accessibility level, then **anything inside of the class can use it, as well as any subclass**. It's a little broader than private, but still more restrictive than public.

Issue Without Protected Access

Animal.java

```java
public class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

[Info] This will give **an error** as **the instance variable "name" is private in the parent class** because remember that private instance variables can be access within the parent class but not in the child class.

Dog.java

```java
public class Dog extends Animal {
    public Dog(String name, boolean friendly) {
        super(name);
        this.friendly = friendly;
    }
    // This will give an error that name has a private access in Animal class
    public void bark(){
        System.out.printf("The Dog %s is barking!", name);
    }
}
```

Protected Access Solution

Animal.java

```java
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

Dog.java

```java
public class Dog extends Animal {
    private boolean friendly;

    public Dog(String name, boolean friendly) {
        super(name);
        this.friendly = friendly;
    }
    // This will give an error that name has a private access in Animal class
    public void bark(){
        System.out.printf("The Dog %s is barking!", name);
    }
}
```

Zoo.java

```java
public class Zoo {
    public static void main(String[] args) {
        Dog myDog = new Dog("Kiki", true);
        myDog.bark();
    }
}
```

With protected access modifier, the **subclass** Dog **can access** the **instance variable "name"** that it inherited from Animal Class.

Output:

```
The Dog Kiki is barking!
```

Getter Methods for Private Attributes

```java
public class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```java
public class Dog extends Animal {
    private boolean friendly;

    public Dog(String name, boolean friendly) {
        super(name);
        this.friendly = friendly;
    }
    // This will give an error that name has a private access in Animal class
    public void bark(){
        System.out.printf("The Dog %s is barking!", getName());
    }
}
```

```java
public class Zoo {
    public static void main(String[] args) {
        Dog myDog = new Dog("Kiki", true);
        myDog.bark();
    }
}
```

Output:

```
The Dog Kiki is barking!
```

- **This is a better way** to **keep your instance variables to be private** and **still have ways to access or modify their values using getter and setter methods.**

Class Object

- Secretly, any class/objects are inherited from the **default Object** class that is the **base class of everything**.
- If you go up the inheritance hierarchy, everything always gets back to the object class eventually.
- You can say that **Class Object is the mother of all objects in Java**.

You can imagine every classes or objects would be an child to the default class Object of Java.

```
public class MyClass extends Object {


}
```

Default Methods of the Class Object

Method	Description
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method (which include also overriding method `hashCode`), refer to the method's documentation at `docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)`. The default `equals` implementation uses operator `==` to determine whether two references *refer to the same object* in memory. Section 14.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with `==` and with `equals`.
`hashCode`	Hashcodes are `int` values used for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (see Section 16.11). This method is also called as part of `Object`'s default `toString` method implementation.

- Since **all objects are inherited from the class Object** then **all objects can use those methods.**
- All classes in Java inherit directly or indirectly from class Object (package java.lang), so its 11 methods (some are overloaded) are inherited by all other classes.

Method	Description
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class typically followed by a hexadecimal representation of the value returned by the object's `hashCode` method.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5 and 12.5) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`).
`finalize`	This `protected` method is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall from Section 8.10 that it's unclear whether, or when, `finalize` will be called. For this reason, most programmers should avoid method `finalize`.
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called shallow copy—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference-type instance variable. *Implementing `clone` correctly is difficult. For this reason, its use is discouraged.* Some industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 15. Recall from Chapter 7 that arrays are objects. As a result, like all other objects, arrays inherit the members of class `Object`. Every array has an overridden `clone` method that copies the array. However, if the array stores references to objects, the objects are not copied—a shallow copy is performed.

The default implementation of toString() method of Object Class

```
public class Zoo {
    public static void main(String[] args) {
        Dog myDog = new Dog("Kiki", false);
        System.out.println(myDog);
    }
}
```

Output:

```
Dog@1b28cdfa
```

- When you print out any object, it will automatically call toString() method of the class Object.
- By default, toString() method will print out the Class Name with the location of the object in the memory.
- This is not very useful to us! So let's override it!

Override toString() method

```java
public class Dog extends Animal {
    private boolean friendly;

    public Dog(String name, boolean friendly) {
        super(name);
        this.friendly = friendly;
    }

    // Overriding toString() method of Object Class
    @Override
    public String toString() {
        return String.format("This Dog %s is %s friendly", getName(),
                friendly==false?"not":"");
    }
}
```

```java
public class Zoo {
    public static void main(String[] args) {
        Dog myDog = new Dog("Kiki", false);
        System.out.println(myDog);
    }
}
```

Now, our new overridden toString() method is customized to print out the string that is interesting and useful to our defined classes by call print() method.

Output:

```
This Dog Kiki is not friendly
```

(Overloading)

Method Overloading & Constructor Overloading

Method Overloading

- **Methods** of the **same name** can be **declared** in the **same class**, as long as they have **different sets of parameters** (i.e. number of arguments, data type of arguments)

```
public return_type method_name () {
...
}

public return_type method_name (parameter_1) {
...
}

public return_type method_name (parameter_1, parameter_2) {
...
}
```

Method Overloading Example

SumCalculator.java

```
public class SumCalculator {

    private int sum(int a, int b) {
        return a + b;
    }
    private int sum(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SumCalculator s = new SumCalculator();
        System.out.println("Calling the first method:" + s.sum(4, 7));
        System.out.println("Calling the first method:" + s.sum(1, 2, 3));
    }

}
```

[Question]
how can Java knows which method to call when they have the same name?

Output:

```
Calling first sum method: 11
Calling second sum method: 6
```

MethorOverload.java

```java
// Fig. 6.10: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public static void main(String[] args)
   {
      System.out.printf("Square of integer 7 is %d%n", square(7));
      System.out.printf("Square of double 7.5 is %f%n", square(7.5));
   }

   // square method with int argument
   public static int square(int intValue)
   {
      System.out.printf("%nCalled square with int argument: %d%n",
         intValue);
      return intValue * intValue;
   }

   // square method with double argument
   public static double square(double doubleValue)
   {
      System.out.printf("%nCalled square with double argument: %f%n",
         doubleValue);
      return doubleValue * doubleValue;
   }
} // end class MethodOverload
```

Output:

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

[Question]
In this example, the number of parameters of the square method is the same, then how can Java know which method should be used?

Constructors Overloading

- As we know, **a constructor to specify how objects of a class** should be **initialized**.
- A class with several **overloaded constructors** that enable objects of that class to be **initialized in different ways**.
- To **overload constructors**, simply provide multiple constructor declarations with **different parameters**.
- **Three overloaded constructors** having **different parameters lists**:

```
public class School {
        public School() {
                ...
        }
        public School(String name) {
                ...
        }
        public School(String name, int year) {
                ...
        }

}
```

Zero parameter Constructor

One parameter Constructor

Two parameter Constructor

Overloaded Constructors Example

Student.java

```
class Student{
     private int rollno;
     private String name;
     static String college="ITS";

     Student(){
          rollno = -1;
          name = "No name";
     }
     Student(String studentName){
          rollno = -1;
          name = studentName;
     }
     Student(int studentNo){
          rollno = studentNo;
          name = "No name";
     }
     Student(int r, String n){
            rollno = r;
            name = n;
     }
     public int getStudentNumber() {
          return rollno;
     }
     public String getStudentName() {
          return name;
     }
     //static method to change the value of static variable
     static void change(){
          college = "BBDIT";
     }
     //method to display the values
     void display(){
          System.out.println(rollno +" "+ name +" "+ college);
     }
}
```

Overloaded Constructors Example

StudentFactory.java

```
public class StudentFactory {
    public static void main(String args[]){
        Student.change();
        Student s1 = new Student();
        Student s2 = new Student("Quang");
        Student s3 = new Student(2);
        Student s4 = new Student(3,"Thang");
        s1.display();
        s2.display();
        s3.display();
        s4.display();
    }
}
```

Output:

```
-1 No name BBDIT
-1 Quang BBDIT
2 No name BBDIT
3 Thang BBDIT
```

Reuse constructors with "this"

this keyword in Java is a reference variable that refers to the current object of a method or a constructor. The main purpose of using **this** keyword in Java is to remove the confusion between class attributes and parameters that have same names

Before

```
class Student{
    private int rollno;
    private String name;
    static String college="ITS";

    Student(){
        rollno = -1;
        name = "No name";
    }
    Student(String studentName){
        rollno = -1;
        name = studentName;
    }
    Student(int studentNo){
        rollno = studentNo;
        name = "No name";
    }
    Student(int r, String n){
          rollno = r;
          name = n;
    }
    public int getStudentNumber() {
        return rollno;
    }
    public String getStudentName() {
        return name;
    }
    //static method to change the value of static variable
    static void change(){
        college = "BBDIT";
    }
    //method to display the values
    void display(){
        System.out.println(rollno +" "+ name +" "+ college);
    }
}
```

After

```
class Student{
    private int rollno;
    private String name;
    static String college="ITS";

    Student(){
        this.rollno = -1;
        this.name = "No name";
    }
    Student(String name){
        this.rollno = -1;
        this.name = name;
    }
    Student(int rollno){
        this.rollno = rollno;
        this.name = "No name";
    }
    Student(int rollno, String name){
          this.rollno = rollno;
          this.name = name;
    }
    public int getStudentNumber() {
        return this.rollno;
    }
    public String getStudentName() {
        return this.name;
    }
    //static method to change the value of static variable
    static void change(){
        college = "BBDIT";
    }
    //method to display the values
    void display(){
        System.out.println(rollno +" "+ name +" "+ college);
    }
}
```

Final Keyword

"This cannot be changed!"

Final Keyword

- "Final" indicates **"This cannot be changed"**.
- The final modifier for finalizing the implementations of **classes**, **methods**, and **variables.**
- The main purpose of using a class being declared as final is to prevent the class from being sub-classed. If a class is marked as final then no class can inherit any feature from the final class.

Example

```
final class Super {
        private int data = 30;
}

public class Sub extends Super{
        public static void main(String args[ ]){
        }
}
```

Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
    at newJavaExamples.Sub.main(Sub.java:9)
```

Constants (keyword final)

- A constant is a variable which **cannot have its value** changed after declaration.
- It uses the '**final**' keyword.
- Syntax:

Global constant:

accessModifier **final** dataType variableName = value;

Class constant:

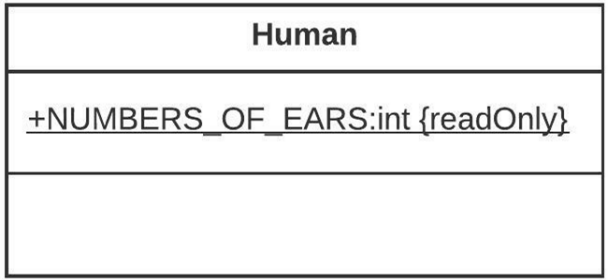
accessModifier **static final** dataType variableName = value;

Constant Examples

```
//global constant, outside of a class
public final double PI = 3.14;
```

```
//class constant within a class
public class Human {
   public static final int NUMBER_OF_EARS = 2;
}

//accessing a class constant by class name
int ears = Human.NUMBER_OF_EARS;
```

UML for Human Class



Static Keyword

Definition: The **static** keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

Static Variables (Class Variables)

- The static variable can be used to refer to the **common property** of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.
- For example, Class Math have two static constants, **Math.PI** (3.14159) and **Math.E** (2.71828)
- Making these fields **static** allows them to be accessed via the class name Math and a dot (.)
- **Advantage of static variable**:
  - Make your program memory efficient (e.g., it saves memory)

Variable without Static Example

Student.java without static variable

```
class Student{
     int rollno;
     String name;
     String college="ITS";

     Student(int r, String n){
           rollno = r;
           name = n;
     }
     //method to display the values
     void display(){
         System.out.println(rollno +" "+ name +" "+ college);
     }
}
```

StudentFactory.java

```
public class StudentFactory {
     public static void main(String args[]){
           Student s1 = new Student(111,"Quang");
           Student s2 = new Student(222,"Thang");
           s1.display();
           s2.display();
     }
}
```

Output:

```
111 Quang ITS
222 Thang ITS
```

[Question] Why do we use static variable in Student class?

Static Variable Example

Student.java with static variable

```
class Student{
     int rollno;
     String name;
     static String college="ITS";

     Student(int r, String n){
           rollno = r;
           name = n;
     }
     //method to display the values
     void display(){
         System.out.println(rollno +" "+ name +" "+ college);
     }
}
```

StudentFactory.java

```
public class StudentFactory {
     public static void main(String args[]){
           Student s1 = new Student(111,"Quang");
           Student s2 = new Student(222,"Thang");
           Student.college = "IU";
           s1.display();
           s2.display();
     }
}
```

Output:

```
111 Quang IU
222 Thang IU
```

Static Methods (Class Method)

- **Some classes** also provide **methods** that **perform common tasks** and **do not require** you to **create objects of those classes**.
- A static class method is a method that belongs to the class itself, not the instance object of a class.
- That means we don't need an object to call a static class method. We call them directly from the class itself.
- A static method can access **static data member** and can **change the value of it**.
- To mark a method as **static**, we simply add the static keyword between the **access modifier** and the **return type**.

```
access_modifier static return_type method_name() {
    // body
}
```

Static Methods (Class Method)

- Instead, we call the method from an object of a class:

```
ClassObjects.methodName(arguments)
```

- We can call class method right from a class:

```
ClassName.methodName(arguments)
```

- For example, you can calculate the square root of 900.0 with the static method call “sqrt” of the Math class:

```
Math.sqrt(900.0)
```

Static Method Example

Student.java

```
class Student{
     int rollno;
     String name;
     static String college="ITS";

     Student(int r, String n){
           rollno = r;
           name = n;
     }
     //static method to change the value of static variable
     static void change(){
         college = "BBDIT";
     }
     //method to display the values
     void display(){
         System.out.println(rollno +" "+ name +" "+ college);
     }
}
```

StdudentFactory.java

```
public class StudentFactory {
    public static void main(String args[]){
          Student.change();
          Student s1 = new Student(111,"Quang");
          Student s2 = new Student(222,"Thang");
          s1.display();
          s2.display();
    }
}
```

[Question] What is the output?

Why Method main is static?

- Remember, we have:

```
public static void main(String args[])
```

- When you execute the Java Virtual Machine (**JVM**) with the java command, the JVM attempts to **invoke the main method** of the class you specify—**at this point no objects of the class have been created**. **Declaring main as static allows the JVM to invoke main without creating an instance of the class.**
- Remember, when we execute an application, we need to specific class name as an argument to java command:

```
java ClassName argument1 argument2 ...
```

- By having main method to be static, the JVM loads the class specified by ClassName and uses that class name to invoke method main.

Static Block

- A static block, or static initialization block, is code that is run once for each time a class is loaded into memory. It is **useful for setting up static variables or logging, which would then apply to every instance of the class.**

Static Block

StaticBlock.java

```
public class StaticBlock {
    static{
        System.out.println("static block is invoked");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
         System.out.println("Hello main");
    }

}
```

Output:

```
static block is invoked
Hello main
```

Static Nested Class

- A class can be made **static** only if it is a nested class.
- We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes.
- Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

Static Nested Class Example

OuterClass.java

```
public class OuterClass {

    private static String str = "Hello World";
    String str1 = "Cannot use";

    // Static class
    static class MyNestedClass {

        // non-static method
        public void disp(){
          System.out.println(str);
          //System.out.println(str1);
        }
    }

    public static void main(String args[])
    {
        OuterClass.MyNestedClass obj= new OuterClass.MyNestedClass();
        obj.disp();
    }
}
```

Output:

```
Hello World
```

Thank you for your listening!

“One who never asks
Either knows everything or nothing”

Malcolm S. Forbes