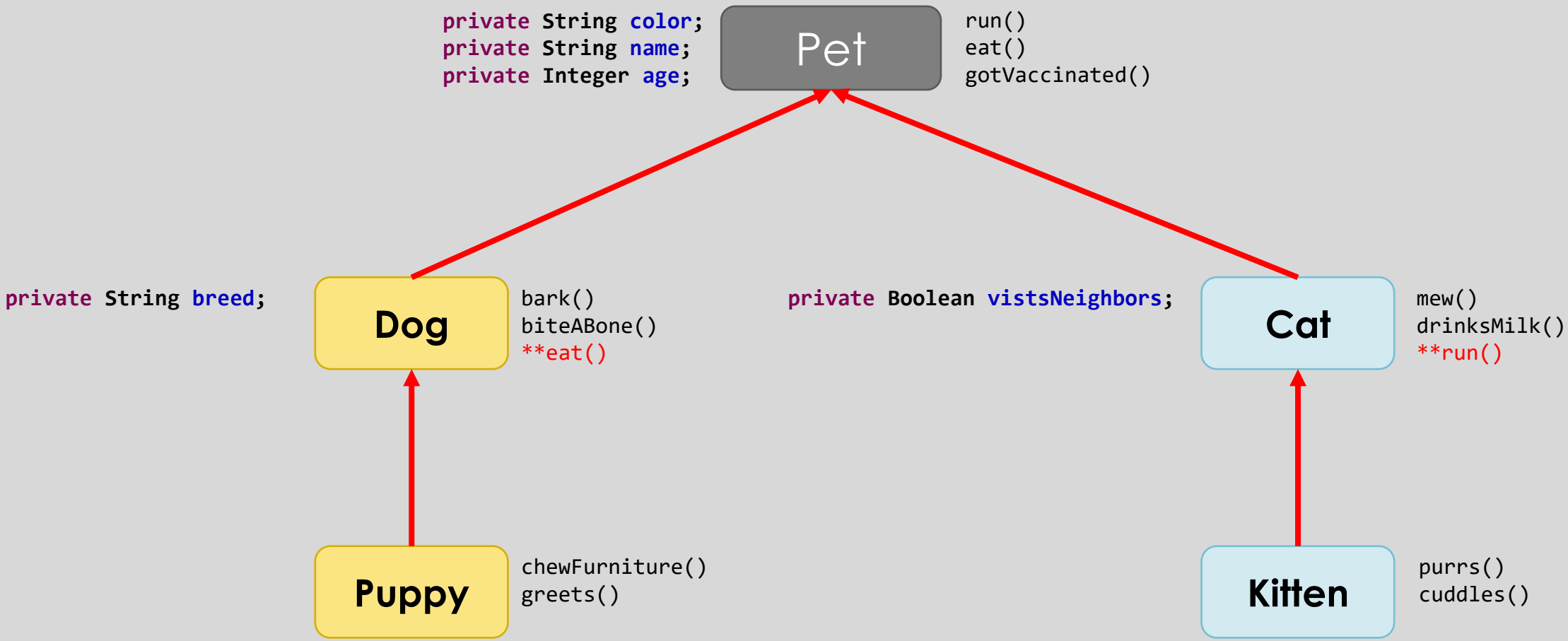

LETS LEARN JAVA
INHERITANCE

Training Plan - Intermediate

Topics to be covered
Inheritance and Aggregation Example of Inheritance Constructors invocation during Inheritance Using super to invoke super class constructors Overriding methods Types of inheritance

Inheritance – Let us examine below



Observations

There are attributes and behaviour common to all Pet

Requirement	Syntax
Dog is a Pet	`Dog extends Pet`
Cat is a Pet	`Cat extends Pet`
Puppy is a Dog and hence is a Pet	`Puppy extends Dog`
Kitten is a Cat and hence is a Pet	`Kitten extends Cat`

When a class extends another class, it has access to all its non private attributes and methods

```
Dog extends Pet
```

Dog is the sub class, child class or inherited class
Pet is the super class, parent class or the base class

Inheritance and Aggregation

```java
public class Pet {
    // Generic
    private String color;
    private String name;
    private Integer age;
```

```java
public class Dog extends Pet {

    // Specific
    private String breed;
```

Inheritance

IS-A relationship

Dog is a Pet
Cat is a Pet

Reusability, and basis of good and extensible design

```java
public class Person {
    String name;
    Character gender;
    Address address;
    List<Pet> pets = new ArrayList<>();
```

Aggregation

HAS-A relationship

Person has an address
Person has pets

Nothing unique. Just like other attributes and data type

Default Constructors across Hierarchies

```java
public class Pet {
    // Generic
    private String color;
    private String name;
    private Integer age;

    public Pet() {
        System.out.println("Pet is getting created with the default constructor");
    }
```

```java
public class Dog extends Pet {

    // Specific
    private String breed;

    // Default constructor
    public Dog() {
        System.out.println("Dog is getting created with the default constructor");
    }
```

```java
Dog d = new Dog();
```

```
Pet is getting created with the default constructor
Dog is getting created with the default constructor
```

Even if not explicitly invoked, the **super class default constructor is invoked when an object of a sub class is being created**.

This also means, **sub class cannot exist without a super class**.

Parameterized Constructors and **super**

```java
public class Pet {
    // Generic
    private String color;
    private String name;
    private Integer age;

    public Pet(String color, String name, Integer age) {
        this.color = color;
        this.name = name;
        this.age = age;
        System.out.println("Pet is getting created with the parameterized constructor");
    }
```

```java
public class Dog extends Pet {

    private String breed;

    public Dog(String color, String name, Integer age, String breed) {
        super(color, name, age);
        this.breed = breed;
        System.out.println("Dog is getting created with the parameterized constructor");
    }
```

```java
Dog bruno = new Dog("Brown", "Bruno", 3, "Pug");
```

Parameterized constructors of super class are invoked by making a call using **super**

Super class constructor call **has to be the first statement in a sub class constructor** if it needs to be involved.

Method overriding

```java
public class Pet {

    public void run() {
        System.out.println("Pet is running");
    }

    public void eat() {
        System.out.println("Pet is eating a very delicious and nutritious meal");
    }
```

```java
public class Cat extends Pet {

    @Override
    public void run() {
        System.out.println("Cat does not want to run :(");
    }
```

```java
public class Dog extends Pet {

    @Override
    public void eat() {
        if (breed.equalsIgnoreCase("German Shepherd")) {
            System.out.println("Dog is eating very fast");
        } else {
            System.out.println("Dog is eating and enjoying its food");
        }
    }
```

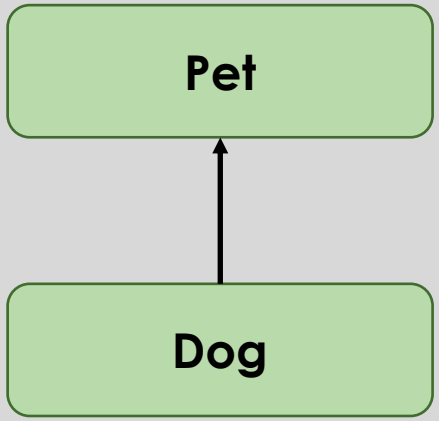
Methods of super class can be overridden in the sub class.

For methods to be overridden, they must

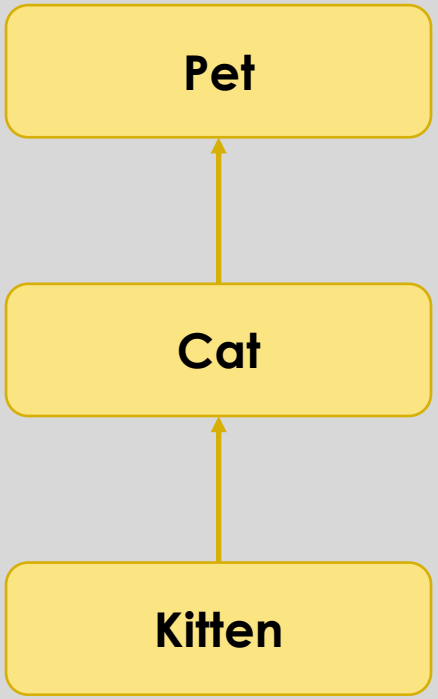
- Have same name
- Same return type and
- Same arguments

An `@Override` annotation may be used to indicate that this is an overridden method in a sub class; However such annotation is not mandatory

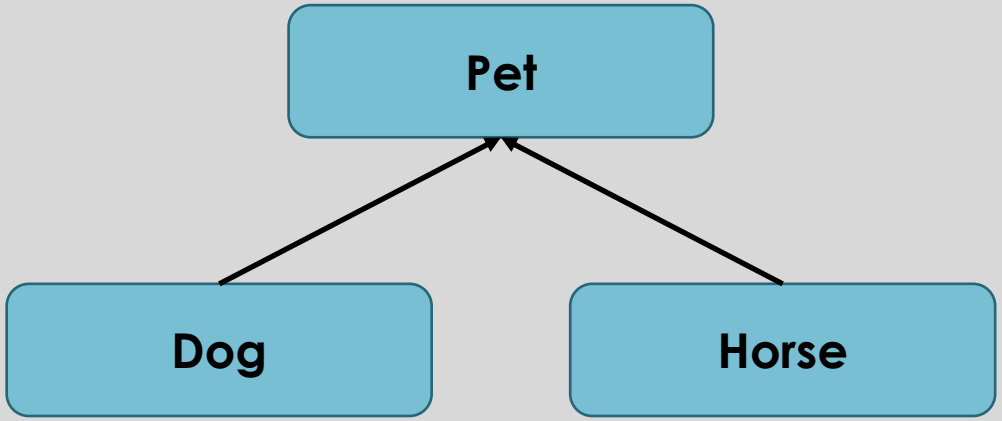
Types of inheritance



Single

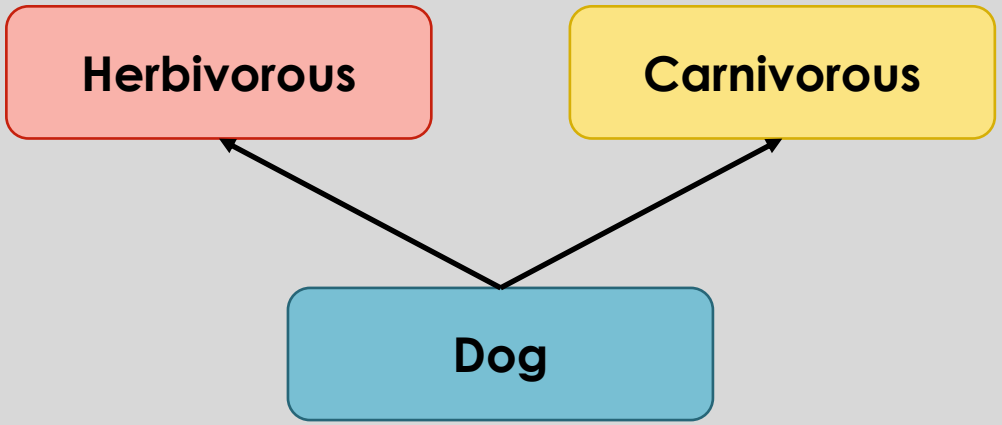


Multi Level

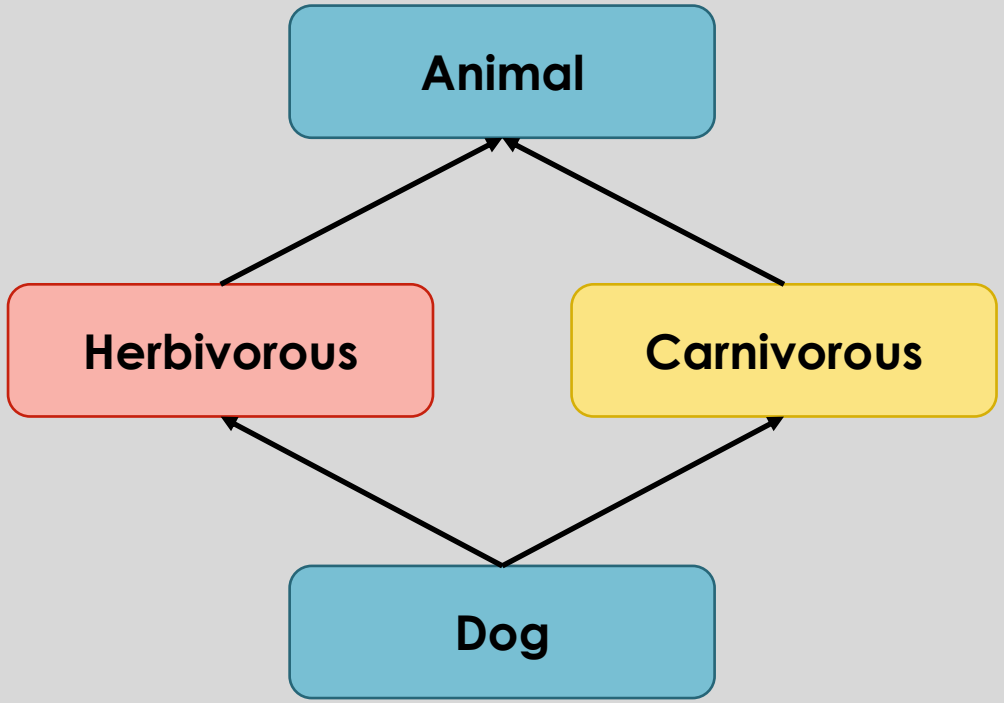


Hierarchical

Types of inheritance – **not allowed in Java**



Multiple



Hybrid

This limitation is overcome using interfaces. We will learn about interfaces in a subsequent module