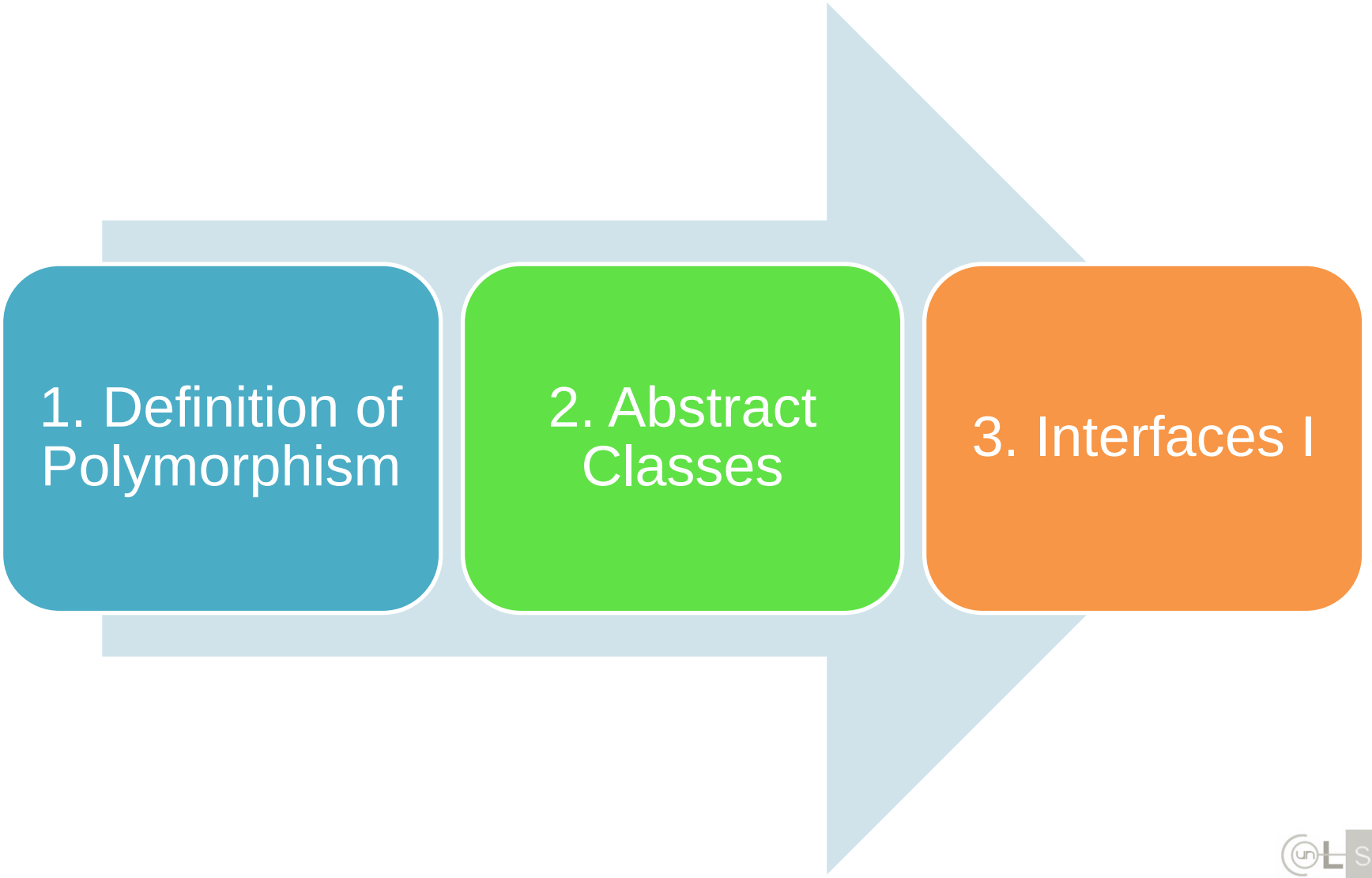
Polymorphism

Christian Rodríguez Bustos
Edited by Juan Mendivelso
Object Oriented Programming

Agenda


1. Definition of Polymorphism
2. Abstract Classes
3. Interfaces I

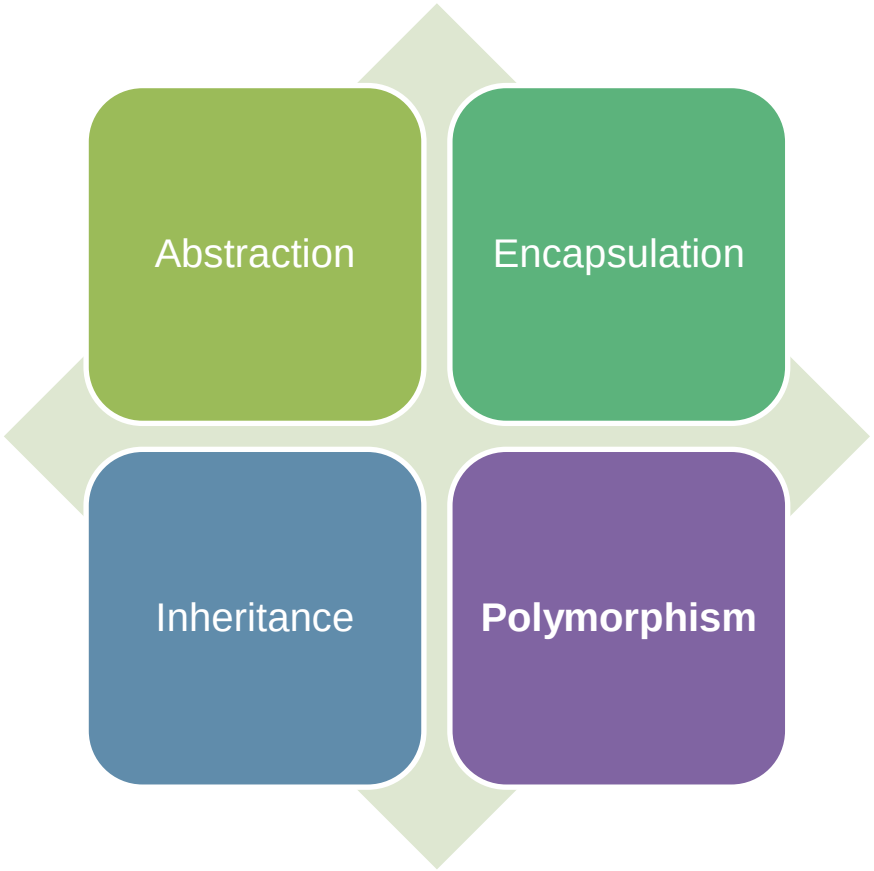
1. Polymorphism

1.1 Polymorphism Simplifies Code Maintenance

1.2 Polymorphism in Code

1.1 Polymorphism Simplifies Code Maintenance

Polymorphism is one of the four principles of OOP



Is the ability of two or more objects belonging to **different classes** to respond to exactly the same message (method call) in **different class-specific ways**.

Polymorphism example

Different **Person** objects can perform the same **kick** behavior in different ways

It is possible to say that they respond to exactly the same message in **different class-specific ways**.

Polymorphism Simplifies Code Maintenance

In the **Pet Store** application you have to print the information available from pets depending of his type

Polymorphism Simplifies Code Maintenance

```
public static void printInfo(Pet pet) {

    char animalType = pet.getType();

    switch (animalType) {
        case 'D': {
            pet.printDogInfo();
        }
        break;
        case 'B': {
            pet.printBirdInfo();
        }
        break;
        case 'C': {
            pet.printCatInfo();
        }
        break;
        default: {
            System.out.println("Bad Type");
        }
    }
}
```

Without polymorphism !!!

We have to **evaluate every possible answer or situation** depending of the animal type

Polymorphism Simplifies Code Maintenance

New pet arrives!!!!

We have to **modify our switch case** in order to support the new pet type

Polymorphism Simplifies Code Maintenance

```
public static void printInfo(Pet pet) {

    char animalType = pet.getType();

    switch (animalType) {
        case 'D': {
            pet.printDogInfo();
        }
        break;
        case 'B': {
            pet.printBirdInfo();
        }
        break;
        case 'C': {
            pet.printCatInfo();
        }
        case 'S': {
            pet.printSerpentInfo();
        }
        break;
        default: {
            System.out.println("Bad Type");
        }
    }
}
```

The new Code for new pet type

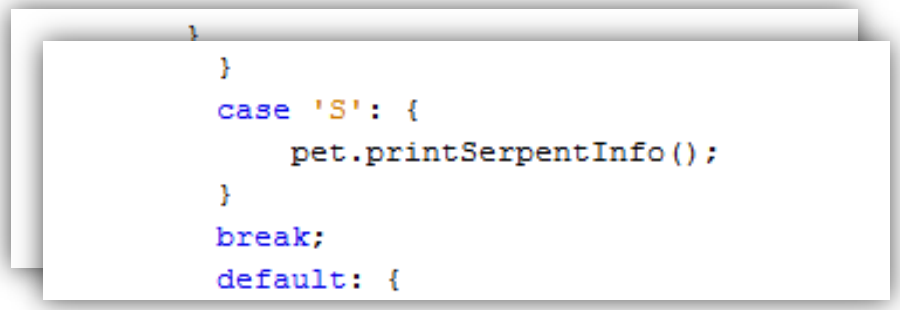
Polymorphism Simplifies Code Maintenance

How many lines of code do you need to add for every new pet type?

```
        }
        case 'S': {
            pet.printSerpentInfo();
        }
        break;
        default: {
```

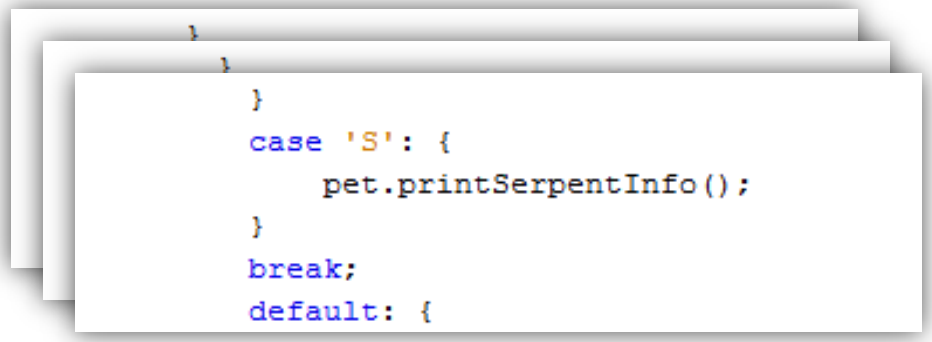
Polymorphism Simplifies Code Maintenance

How many lines of code do you need to add for every new pet type?



Polymorphism Simplifies Code Maintenance

How many lines of code do you need to add for every new pet type?



Polymorphism Simplifies Code Maintenance

How many lines of code do you need to add for every new pet type?

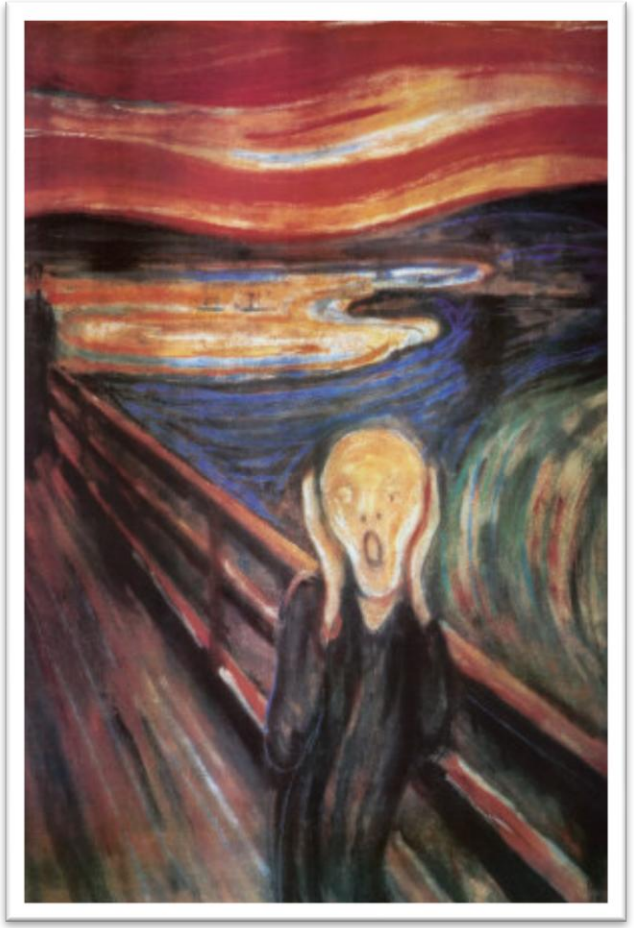
Polymorphism Simplifies Code Maintenance

Polymorphism Simplifies Code Maintenance

What happens if the pet type now is stored in a **String** variable instead of a **char** variable?

Polymorphism Simplifies Code Maintenance

Maintenance of non polymorphic applications quickly becomes a **nightmare!!!**



Polymorphism Simplifies Code Maintenance

```
public void printInfo(Pet pet) {

    pet.printInfo(pet);

}
```

With polymorphism !!!

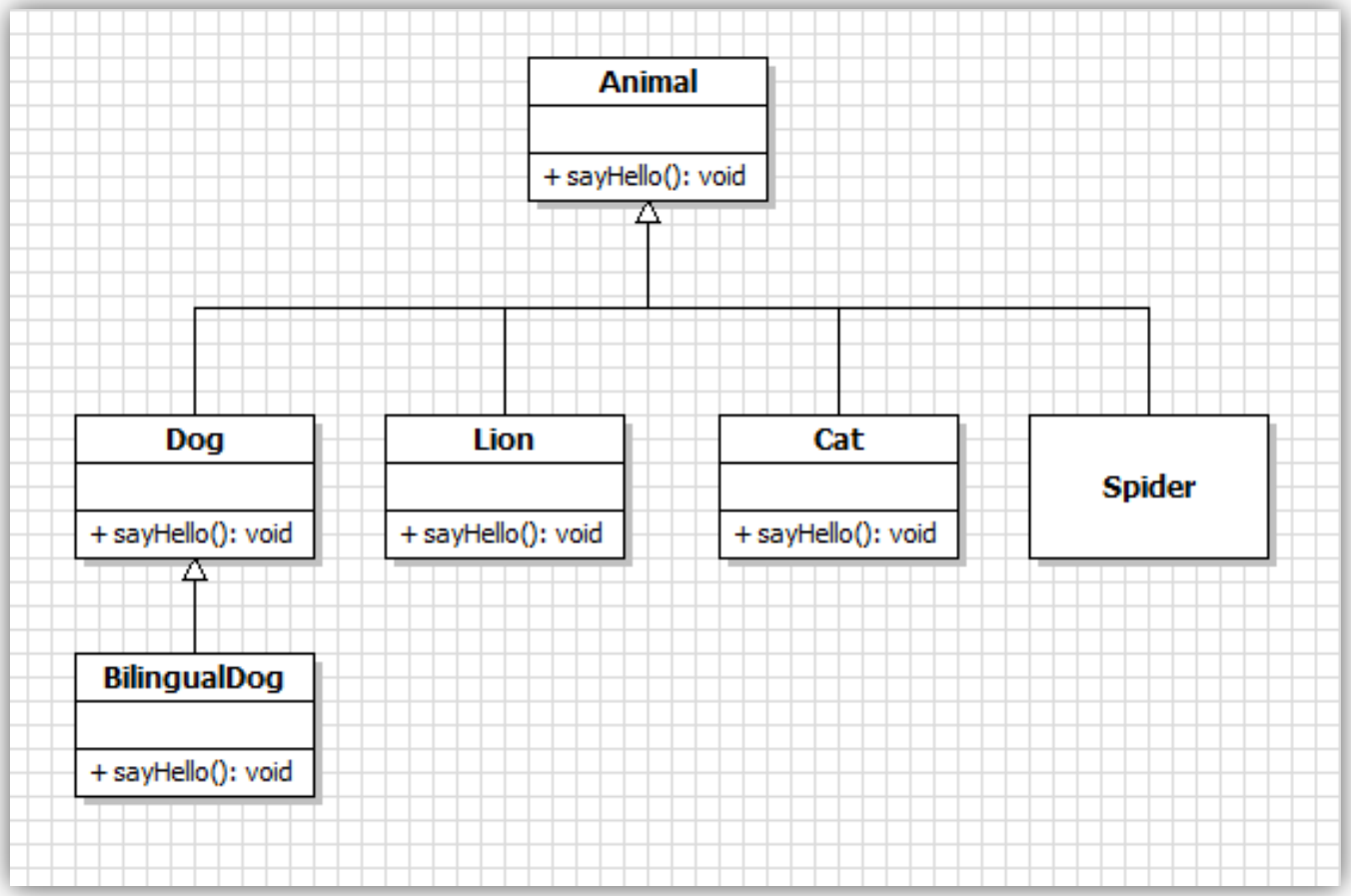
We only have to **override and inherit** the print method and the JVM will execute the corresponding behavior.

1.2 Polymorphism in Code

Polymorphism example

Do you remember our old friends?

Polymorphism example



This is Polymorphism

```java
public abstract class Animal {

    public void sayHello() {
        System.out.println("I have nothing to say");
    }
}
```

```java
public class Dog extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GUAU!!!");
    }
}
```

```java
public class Lion extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GRRRR!!!");
    }
}
```

```java
public class Cat extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say MEOW!!!");
    }
}
```

```java
public class Spider extends Animal {
}
```

This is Polymorphism

```java
public abstract class Animal {

    public void sayHello() {
        System.out.println("I have nothing to say");
    }
}
```

```java
public class Dog extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GUAU!!!");
    }
}
```

```java
public class Lion extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say GRRRR!!!");
    }
}
```

```java
public class Cat extends Animal {

    @Override
    public void sayHello() {
        System.out.println("I say MEOW!!!");
    }
}
```

These three animals answer the **sayHello** calling in **different ways**.

This is Polymorphism in code

Inheritance combined with **overriding** facilitates **polymorphism**

```
public class Dog extends Animal {

    @Override
     public void sayHello() {
          System.out.println("I say GUAU!!!");
     }
}
```

2. Abstract keyword

2.1 Abstract Classes

2.2 Abstract Methods

2.1 Abstract Classes

Abstract classes

In abstraction processes, it is common to find **abstract classes and abstract behaviors**

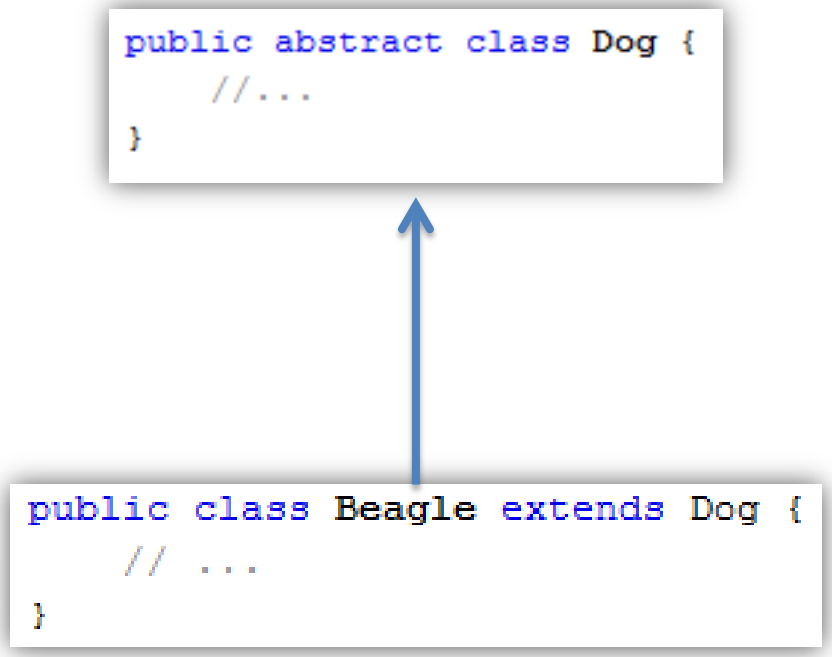
We know that we can buy **Dogs**

But we need to choose a **Breed** at the moment of the buying because we cannot buy only **Dogs**.

Abstract classes

We usually buy **Beagle Dogs**, **Akita Dogs**, **Cocker Spaniel Dogs**, etc.

Abstract classes cannot be instantiated (or bought)

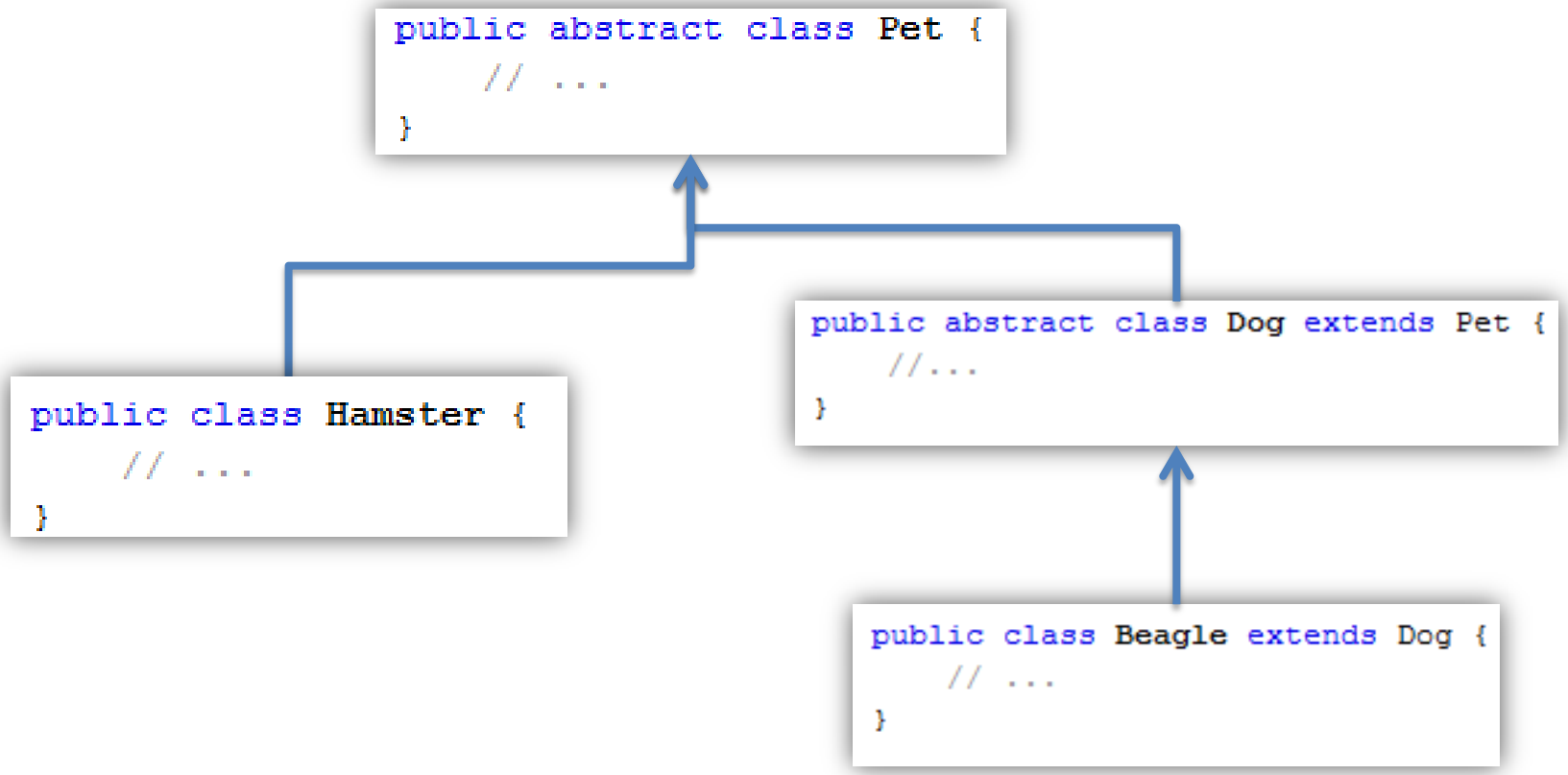


Abstract classes example

In the same way we do not buy a **Pet**

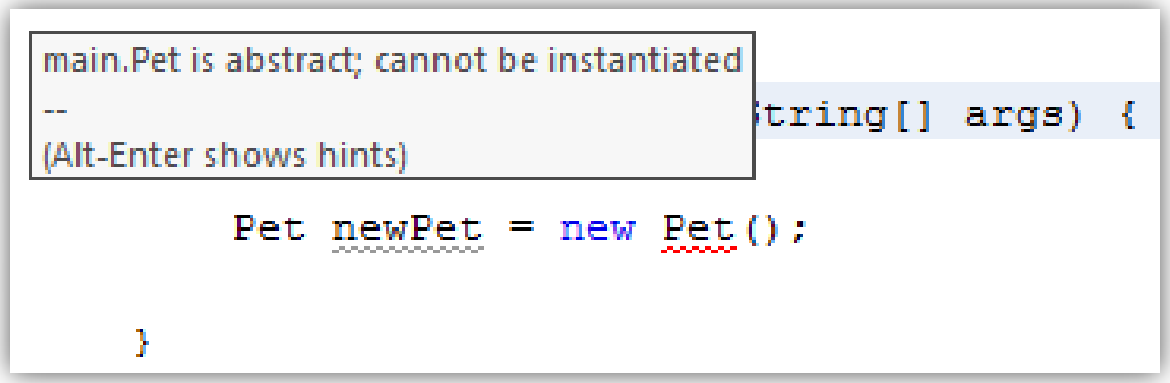
Usually, we buy a **Hamster**, a **Persian Cat**, a **Parrot Bird**, a **Beagle Dog**, etc.

Abstract classes example



Abstract Classes and Instantiation

Do not forget that **abstract classes cannot be instantiated**



2.2 Abstract Methods

Abstract methods

We know that all **pets walk in the same way**

```
public abstract class Pet {

    public void walk(long distance) {
        goForward(distance);
    }
```

All pets walk going forward, so this is a **general implementation for all pets** because we know how pets walk

Abstract methods

But we know that **pets talk in different ways**

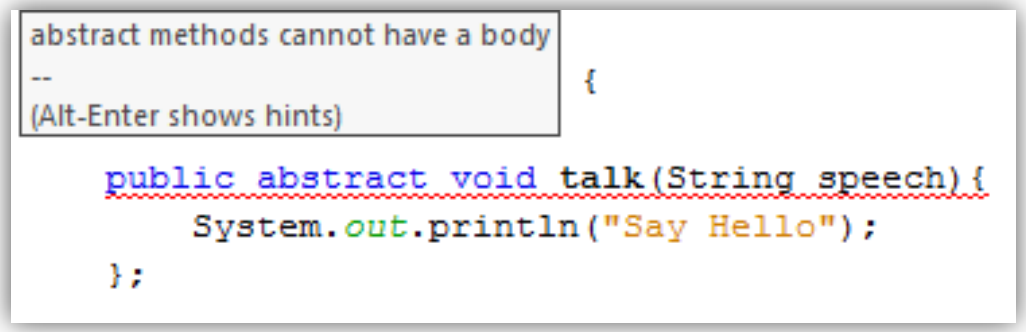
```
public abstract class Pet {

    public abstract void talk(String speech);

}
```

We cannot generalize the talking process, so we define the behavior as **abstract**

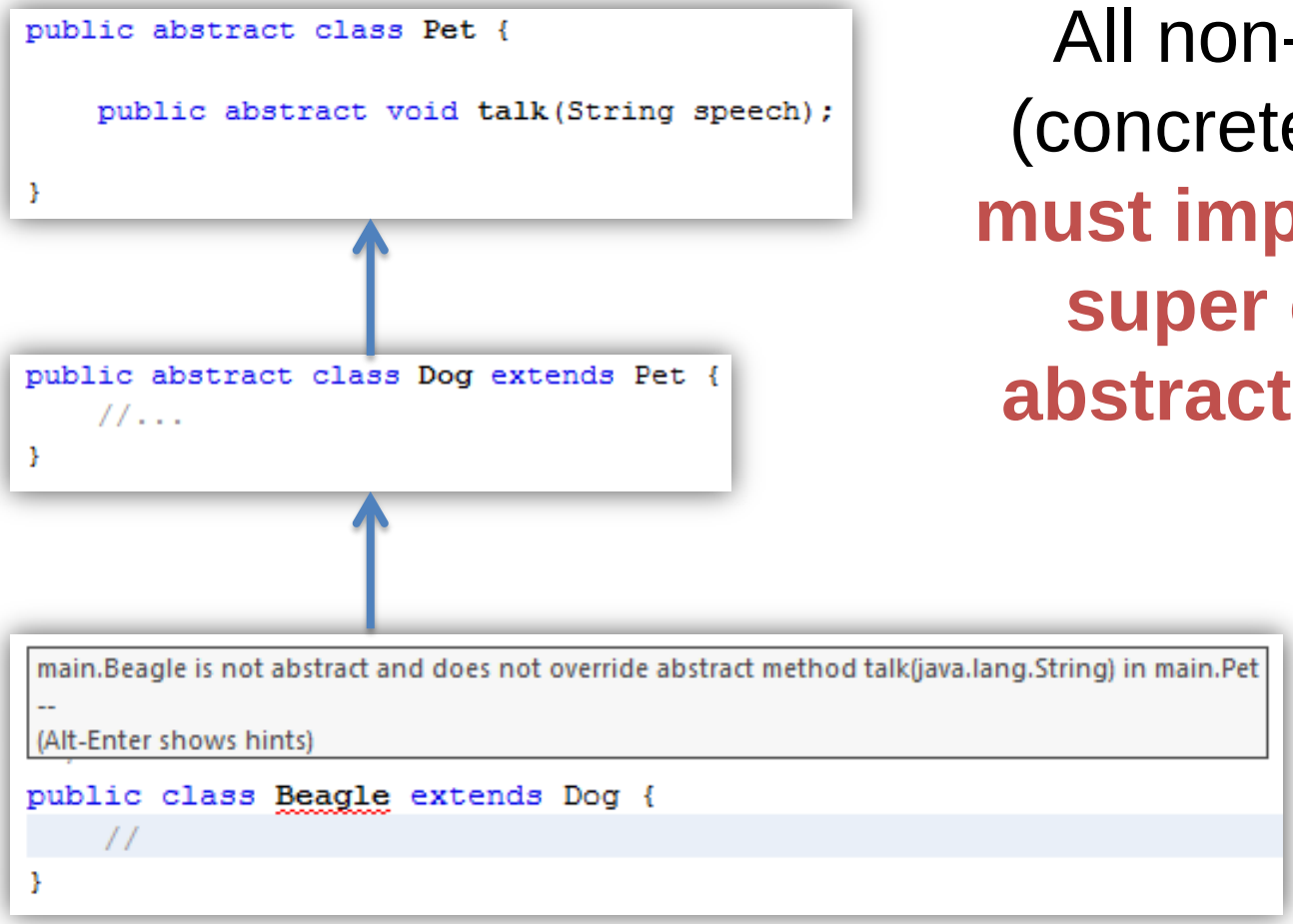
Abstract methods

```
public abstract class Pet {

    public abstract void talk(String speech);

}
```

Abstract methods definition does not include the body of the method we only define the method's header



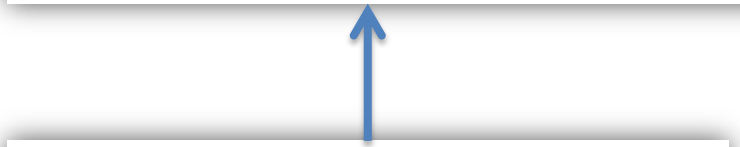
Abstract methods



All non-abstract (concrete) classes **must implement all super classes abstract methods**

Abstract methods

```java
public abstract class Pet {

    public abstract void talk(String speech);

}
```

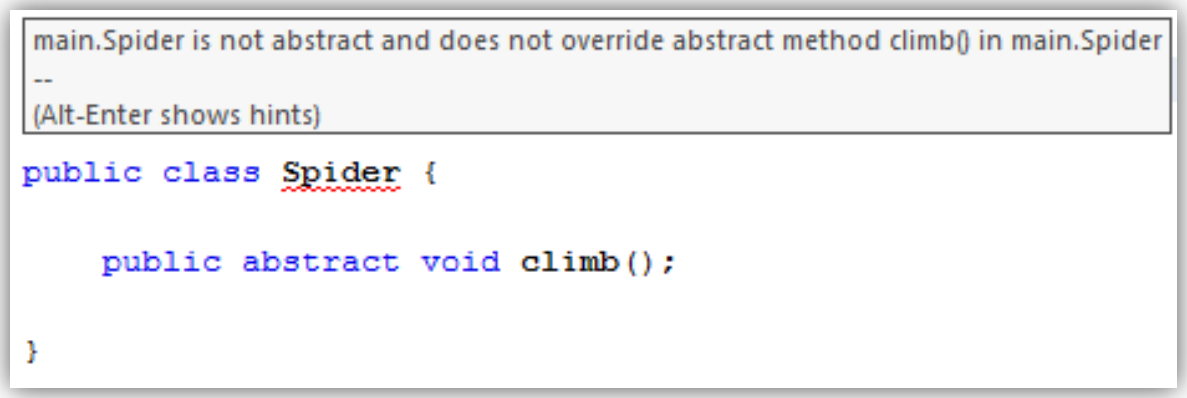


```java
public abstract class Dog extends Pet {
    //...
}
```

```java
public class Beagle extends Dog {

    @Override
    public void talk(String speech) {
        System.out.println("Say Guau");
    }
}
```

All non-abstract (concrete) classes **must implement all super classes abstract methods**

Abstract methods



We can define abstract methods only if the class is abstract

Abstract methods

```
public abstract class Pet {

    public abstract void talk(String speech);

    public void walk(long distance) {
        goForward(distance);
    }

    // ...
```

Abstract method

Concrete method

We can define abstract methods and concrete methods in the same class

3. Interfaces I

3.1 What is an Interface?

3.2 Interfaces in Java

3.1 What is an Interface?

Interfaces are contracts between classes

Using abstract classes we can **omit some implementation details** using abstract methods

```
public abstract class Pet {

    public abstract void talk(String speech);

    public void walk(long distance) {
        goForward(distance);
    }

    // ...
```

Implementation details are omitted here

Interfaces are contracts between classes

An **interface provides a definition** of business functionality of a system (through methods).

A **class** that **implements** an interface needs to implement the actual business functionality (the methods).

What is an interface?

Interfaces are contracts defines a set of methods.

A class that implements this interface must implement such methods.

Interface methods must be abstract

- Where we are using interfaces, we **have to omit** all method implementation details.

- However, we do not need to put the word ‘abstract’ in the methods as it is assumed.

```
public interface Pet {

    public void walk(long distance);

    public void talk(String speech);
}
```

Interface methods must be abstract

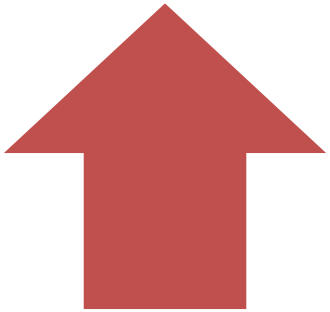
Where we are using interfaces we **have to omit** all methods implementation details

```
public interface Pet {
        interface methods cannot have body
        --
        (Alt-Enter shows hints)
    p                                    tance);

    public void talk(String speech){
        System.out.println("I am talking");
    };
}
```

```
public interface Pet {

    public void walk(long distance);

    public void talk(String speech);
}
```

3.2 Interfaces in Java

Interfaces in Java are implemented by subclasses



Interfaces **are not extended** by subclasses

```
public class Dog extends Pet {

}
```

Interfaces **are implemented** by subclasses

```
public class Dog implements Pet {

    // ...
```

Interfaces in Java are implemented by subclasses

Subclasses have to implement all interface methods

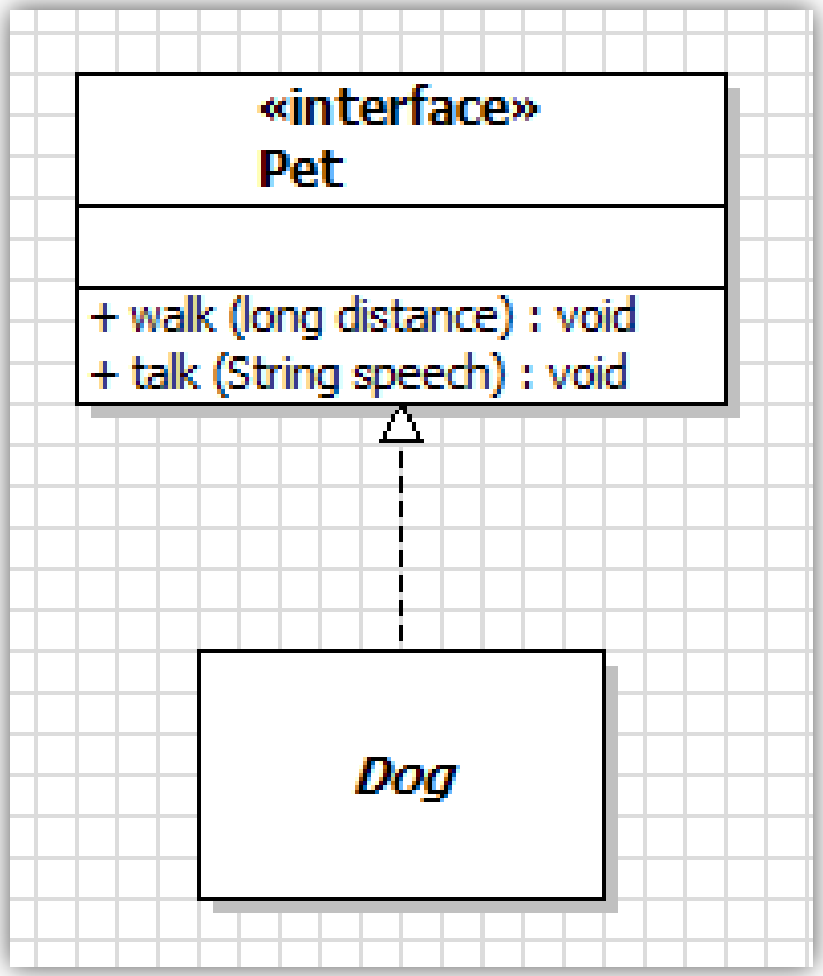
```
main.Dog is not abstract and does not override abstract method talk(java.lang.String) in main.Pet
--
(Alt-Enter shows hints)
public class Dog implements Pet {

    public void walk(long distance) {
        // code code code
    }

}
```

Interfaces in Java are implemented by subclasses

Subclasses have to implement all interface methods

```
public class Dog implements Pet {

    public void walk(long distance) {
        // code code code
    }

    public void talk(String speech) {
        // code code code
    }
}
```

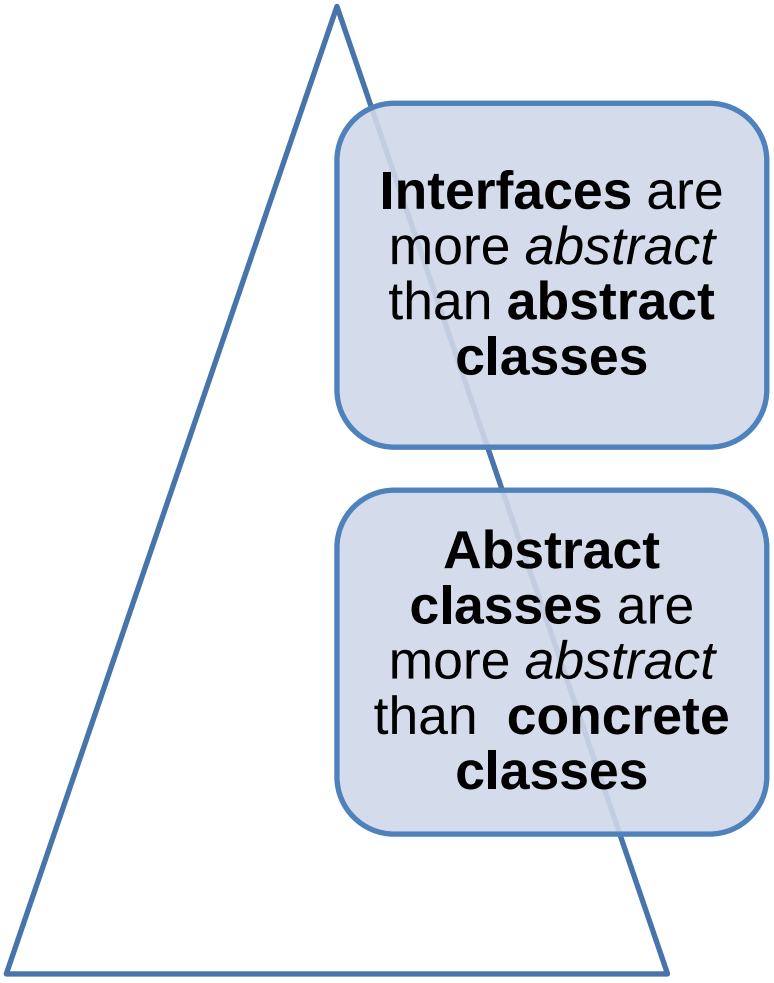
Interfaces and UML



Interface methods do **not appear** in the subclasses

Interfaces Review

Interfaces review

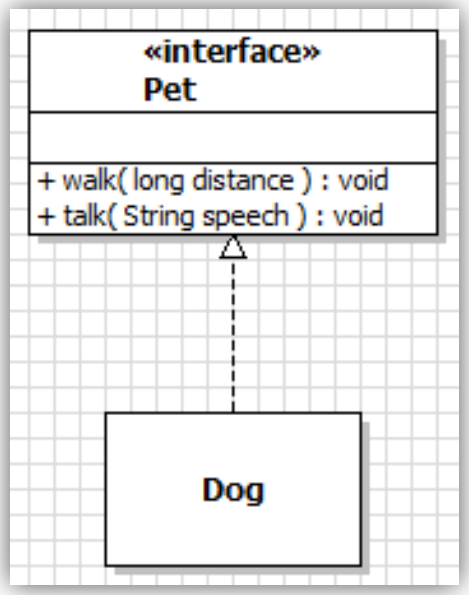


Interfaces leave implementation details to the subclasses

```
public interface Pet {

    public void walk(long distance);

    public void talk(String speech);
}
```

Interfaces are implemented

```
public class Dog implements Pet {

    public void walk(long distance) {
        // code code code
    }

    public void talk(String speech) {
        // code code code
    }
}
```



Abstract classes & Interfaces

```
public abstract class Pet {

    private String name;

    public abstract void walk (long distance);

    public void talk (){
        System.out.println("Say Hello");
    }
}
```

```
public interface Pet {

    void walk (long distance);

    void talk ();

}
```

Abstract classes

- **May** prescribe data structure (attributes).
- Abstract methods **are defined using** "abstract" keyword
- **May** declare concrete methods

Interfaces

- **May NOT** prescribe data structure (attributes are declared as `final`).
- We **need not use** the "public" or “abstract” keyword to define methods
- **May NOT** declare concrete methods.

Interfaces details

May NOT prescribe data structure (attributes).

```
public interface Pet {

    public String NAME;

}
```

Interface attributes are defined **as final** by default

```
modifier private not allowed here

= expected
--
(Alt-Enter shows hints)
```

Interface attributes work as **public constants**

Final attributes are public and must be instantiated at the declaration moment

Interfaces details

May NOT prescribe data structure (attributes).

Interface attributes are defined **as final** by default

Interface attributes work as **public constants**

```
public interface Pet {

    public String NAME = "NONAME";

}
```

Final attributes are public and must be instantiated at the declaration moment

Interfaces details

```
public interface Pet {

    private void walk(long distance);

    void talk();
}
```

```
modifier private not allowed here
--
(Alt-Enter shows hints)
```

We do not **need to use** the "public" or “abstract” keyword to define methods

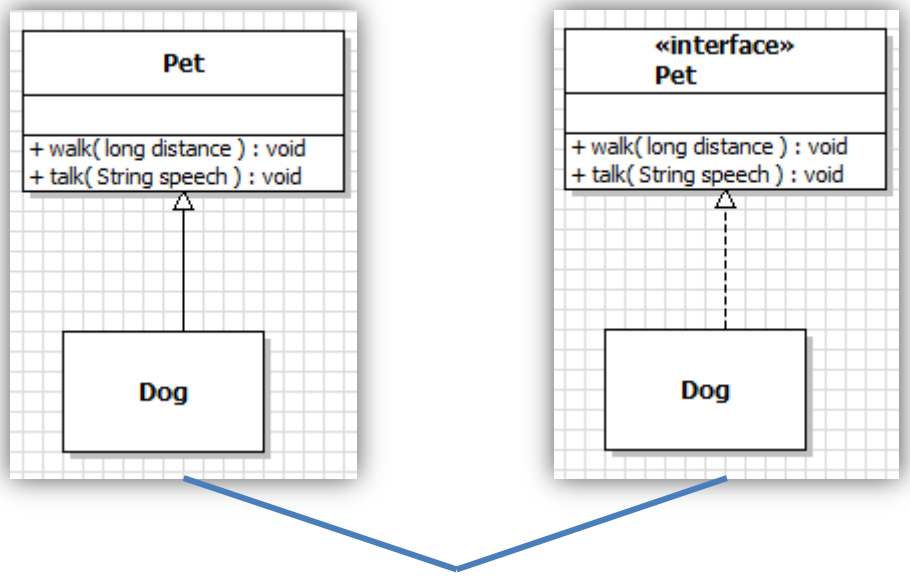
Interfaces details

```
public interface Pet {

    void walk(long distance);

    public void talk();
}
```

We do not **need to use** the "public" or “abstract” keyword to define methods

Interfaces details

Interface implementation and **superclass extension** represent the relation "**is a**"

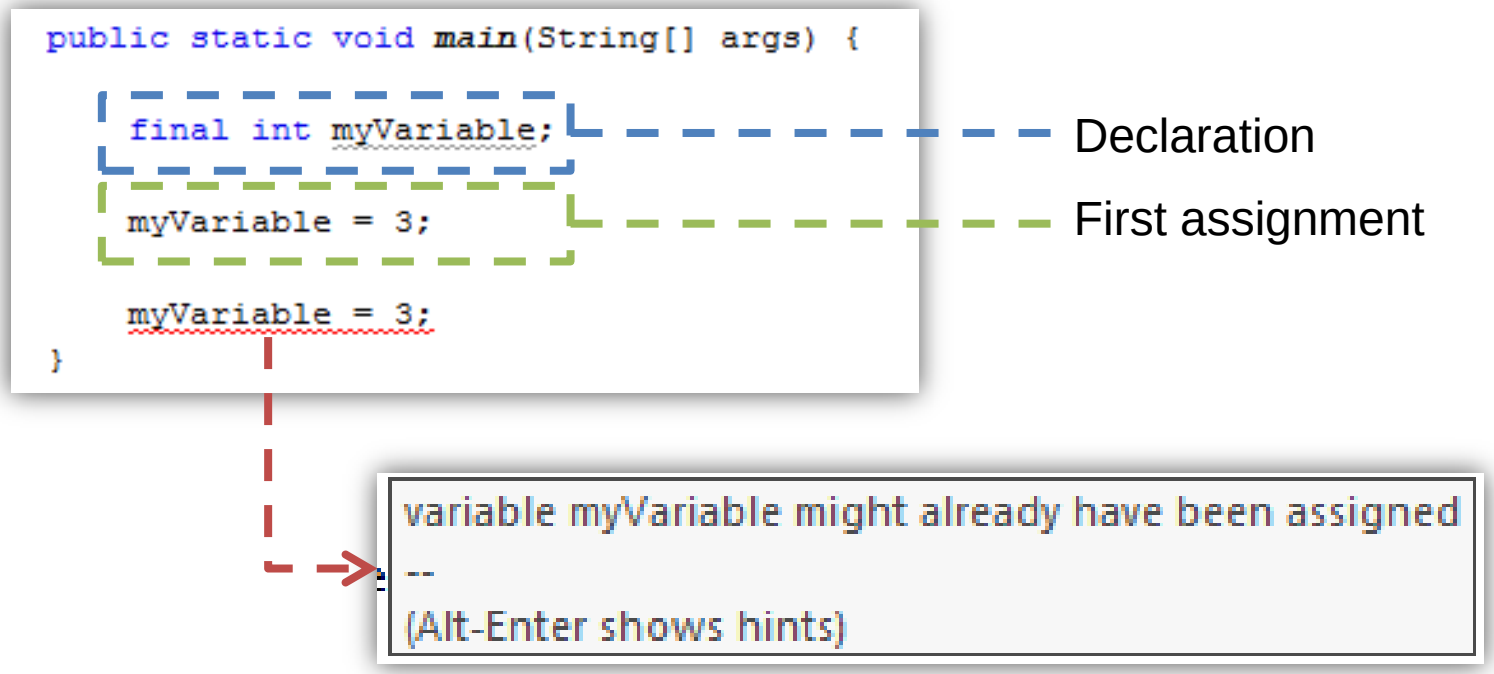


Dog **is a** Pet

4. The final Keyword

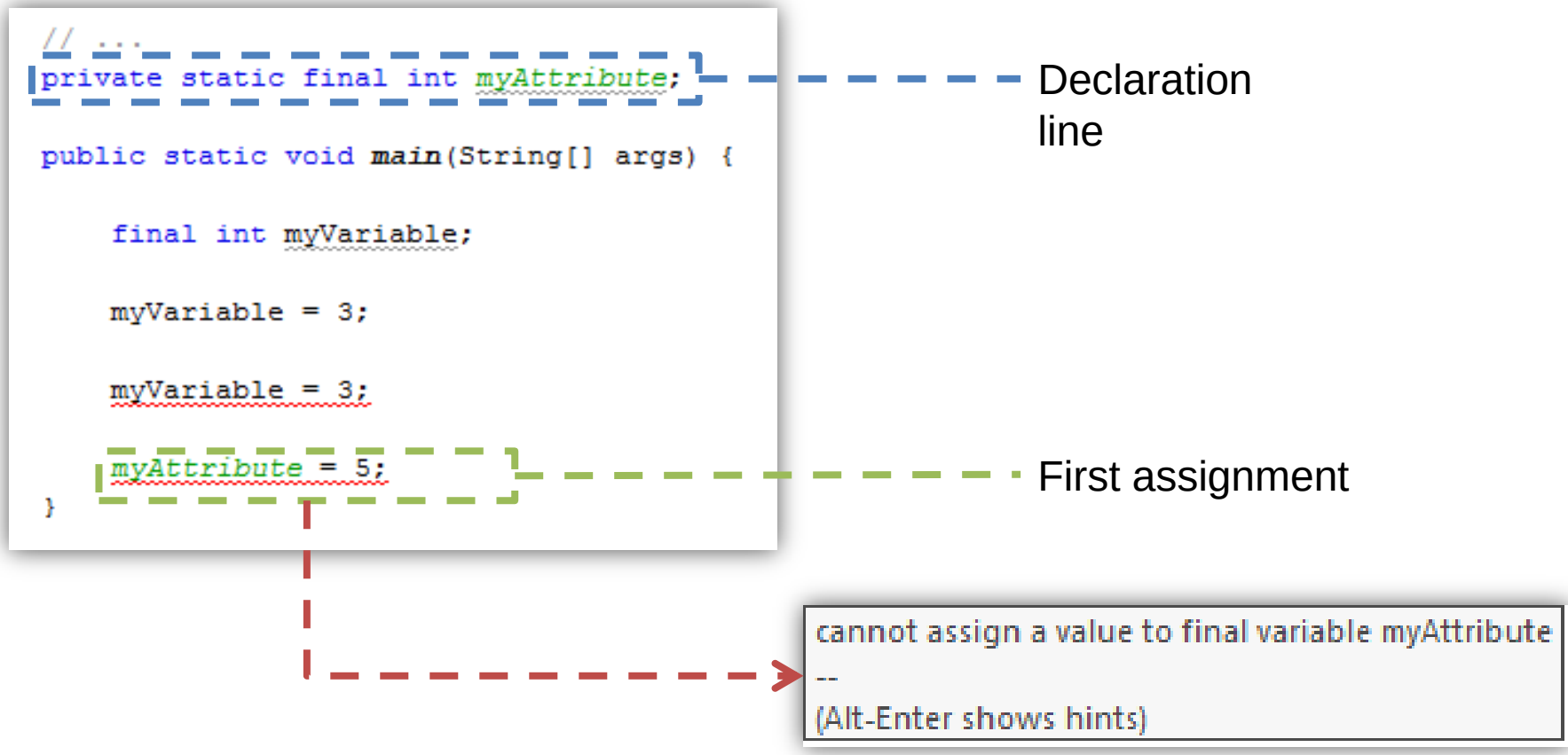
final variable

Can be assigned a value only once in a program



final attribute

Can be assigned a value only in the declaration line



final method

Final methods cannot be overridden in a subclass

```
public abstract class Pet {

    public final void talk() {
        System.out.println("Say Hello");
    }
}
```

```
public class Dog extends Pet {

    @Override
    public void talk() {
        System.out.println("Say Hello");
    }
}
```

```
talk() in animals.Dog cannot override talk() in animals.Pet
  overridden method is final
--
(Alt-Enter shows hints)
```

final class

Final classes cannot be subclassed

```
public final class Pet {

    public final void talk() {
        System.out.println("Say Hello");
    }
}
```

```
public class Dog extends Pet {

}
```

```
cannot inherit from final animals.Pet
--
(Alt-Enter shows hints)
```

5. Threads basics

5.1 How to instantiate a thread

5.2 How to simulate a clock

5.1 How to instantiate a thread

Java Threads

Java provides built-in support for *multithreaded programming*.

A multithreaded program contains **two or more parts that can run concurrently**.

Java defines two ways to define a Thread

```
public class SuperTimer implements Runnable {

    @Override
    public void run() {...}
}
```

We can implement the **Runnable** interface.

```
public class SuperTimer extends Thread {


}
```

We can extend the **Thread** class.

5.2 How to simulate a clock

```java
public class SuperTimer implements Runnable {

    private int timeLimit;

    public SuperTimer(int timeLimit) {
        this.timeLimit = timeLimit;
    }

    public void stop() {
        timeLimit = -1;
    }

    @Override
    public void run() {

        int counter = 0;

        while (counter < this.timeLimit) {
            try {
                counter++;
                Thread.sleep(1000);
//                System.out.println(counter);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

    }
}
```

This method will be executed at thread's start()

```java
public class SuperTimer implements Runnable {

    private int timeLimit;

    public SuperTimer(int timeLimit) {
        this.timeLimit = timeLimit;
    }

    public void stop() {
        timeLimit = -1;
    }

    @Override
    public void run() {

        int counter = 0;

        while (counter < this.timeLimit) {
            try {
                counter++;
                Thread.sleep(1000);
//                System.out.println(counter);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }

    }
}
```

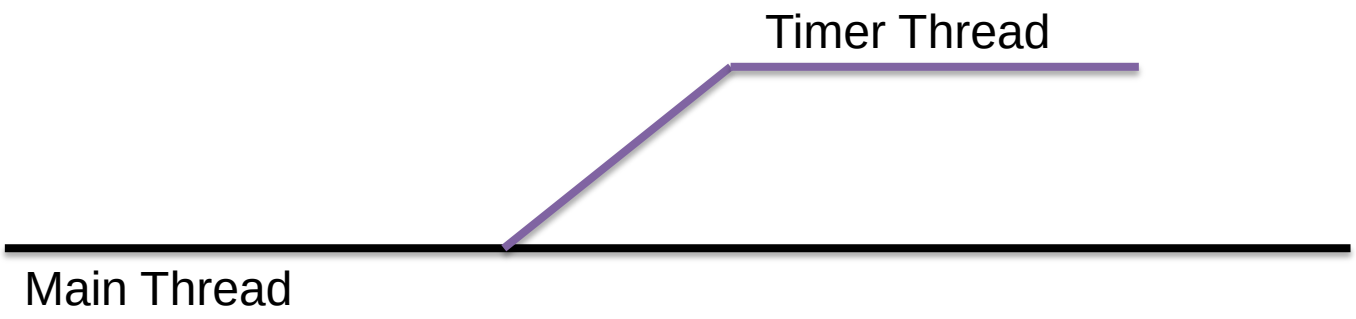
This method pause the Thread 1 second

Creation of a Thread in 3 steps

1. Runnable object instantiation
2. Thread instantiation
3. Thread start

```
public class ThreadTest {

    public static void main(String args[]) {

        SuperTimer timer = new SuperTimer(5);

        Thread timerThread = new Thread(timer);

        timerThread.start();

        play(generateSecretNumber(), timer, timerThread);

    }
    // ...
```

Threads in time



Main thread is the responsible for starting the timer thread, read and print data to the user

Timer Thread is responsible for controlling the game guessing time

References

[Barker] J. Barker, *Beginning Java Objects: From Concepts To Code*, Second Edition, Apress, 2005.