LETS LEARN JAVA

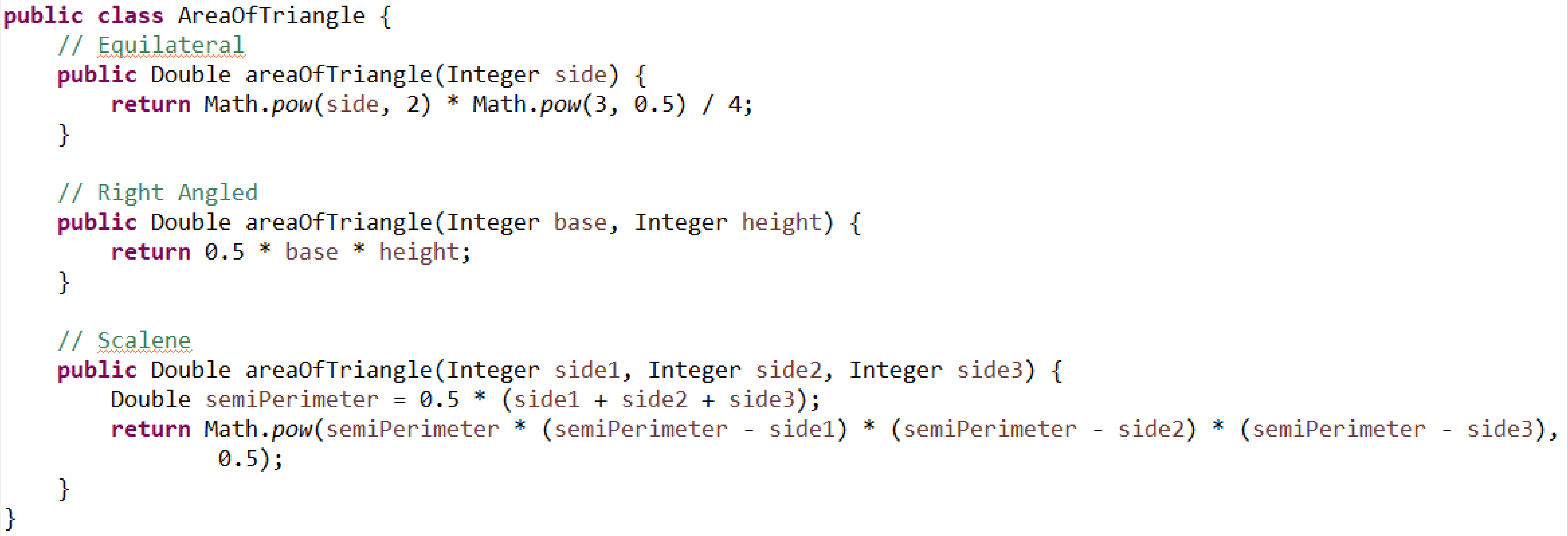
POLYMORPHISM

Training Plan - Basics

Topics to be covered
What is Polymorphism Overloading vs. Overriding Overloading - Variable number of Arguments Using Inheritance to implement Polymorphism Covariant Return type Run Time Polymorphism instanceOf

Poly-morph-ism – overloading (compile time)

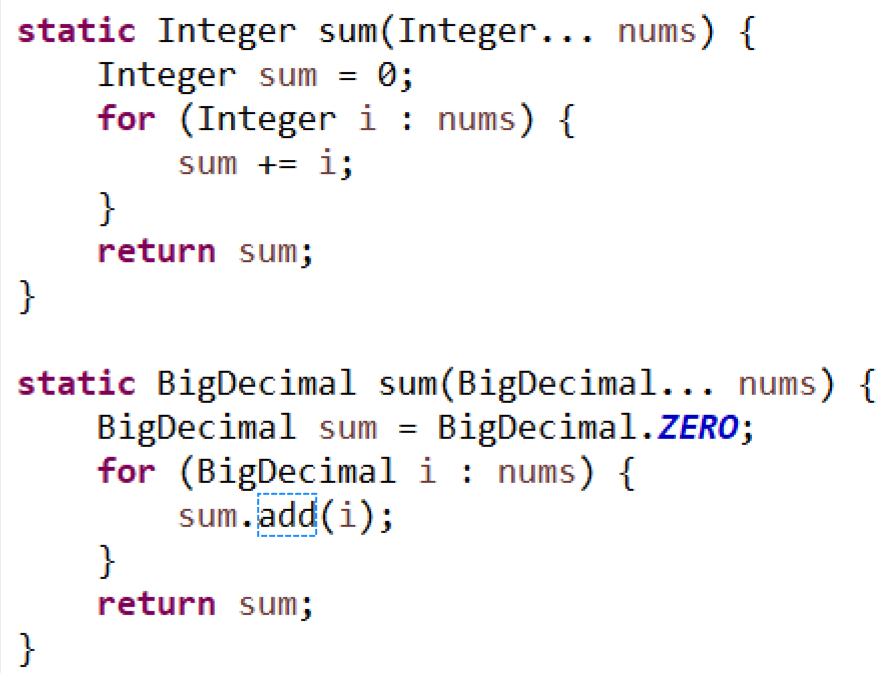
Polymorphism simply means ability to take multiple forms. Consider the following methods. All are used to calculate area, and hence has same name, but different parameters being passed



```
public class AreaOfTriangle {
    // Equilateral
    public Double areaOfTriangle(Integer side) {
        return Math.pow(side, 2) * Math.pow(3, 0.5) / 4;
    }

    // Right Angled
    public Double areaOfTriangle(Integer base, Integer height) {
        return 0.5 * base * height;
    }

    // Scalene
    public Double areaOfTriangle(Integer side1, Integer side2, Integer side3) {
        Double semiPerimeter = 0.5 * (side1 + side2 + side3);
        return Math.pow(semiPerimeter * (semiPerimeter - side1) * (semiPerimeter - side2) * (semiPerimeter - side3),
                0.5);
    }
}
```

Poly-morph-ism - Overloading

Another example which uses the var-arg concept



```java
static Integer sum(Integer... nums) {
    Integer sum = 0;
    for (Integer i : nums) {
        sum += i;
    }
    return sum;
}

static BigDecimal sum(BigDecimal... nums) {
    BigDecimal sum = BigDecimal.ZERO;
    for (BigDecimal i : nums) {
        sum.add(i);
    }
    return sum;
}
```

… operator is used to indicate multiple values but unknown number

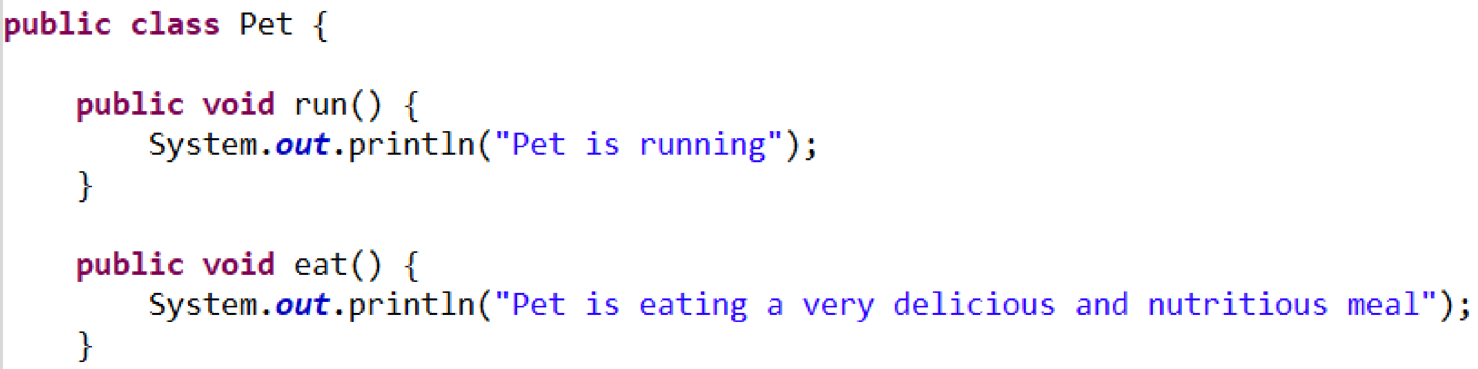
```java
Integer sum(Integer... nums)
```

Will satisfy all the below calls

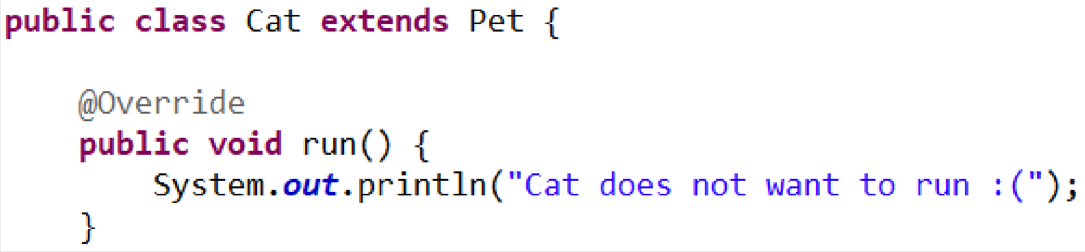
```java
sum(1);
sum(10, 20, 30);
sum(10, 20, 30, 40);
sum(new Integer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });
```

… operator can be used **only once** in a parameter list and has to be the **last parameter** in a method definition

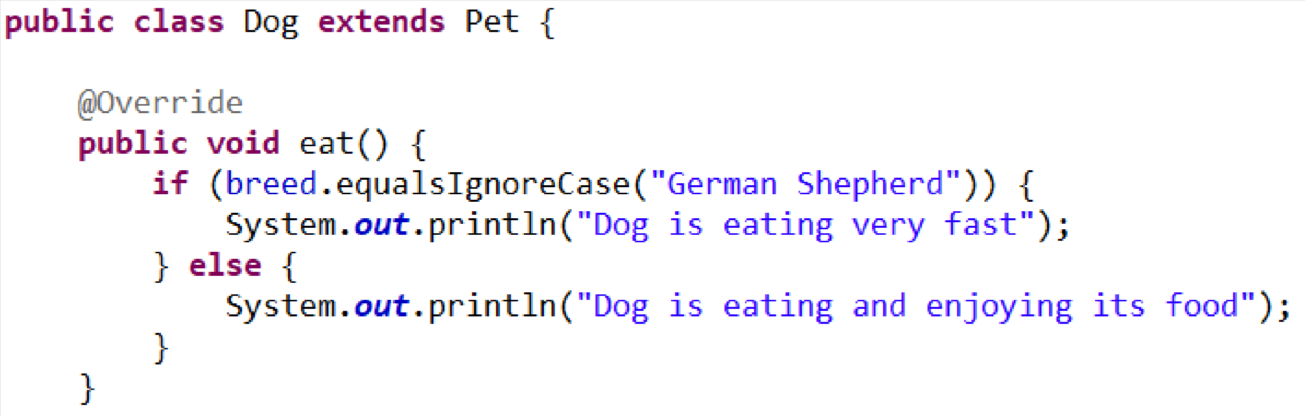
Poly-morph-ism - Overriding



```
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");
    }
```



```
public class Cat extends Pet {

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



```
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");
        }
    }
```

We have already seen how overriding works across inheritance hierarchies

Overloading vs. Overriding

Overloading	**Overriding**
Happens in same class (typically)	Occurs across inheritance hierarchies
Method name has to be same	Method name has to be same
Needs to have different types or number of arguments	Needs to have exactly same arguments
Return type cannot be different	Return type of overriding method can be the subclass of the return type of the original method.

Covariant Types

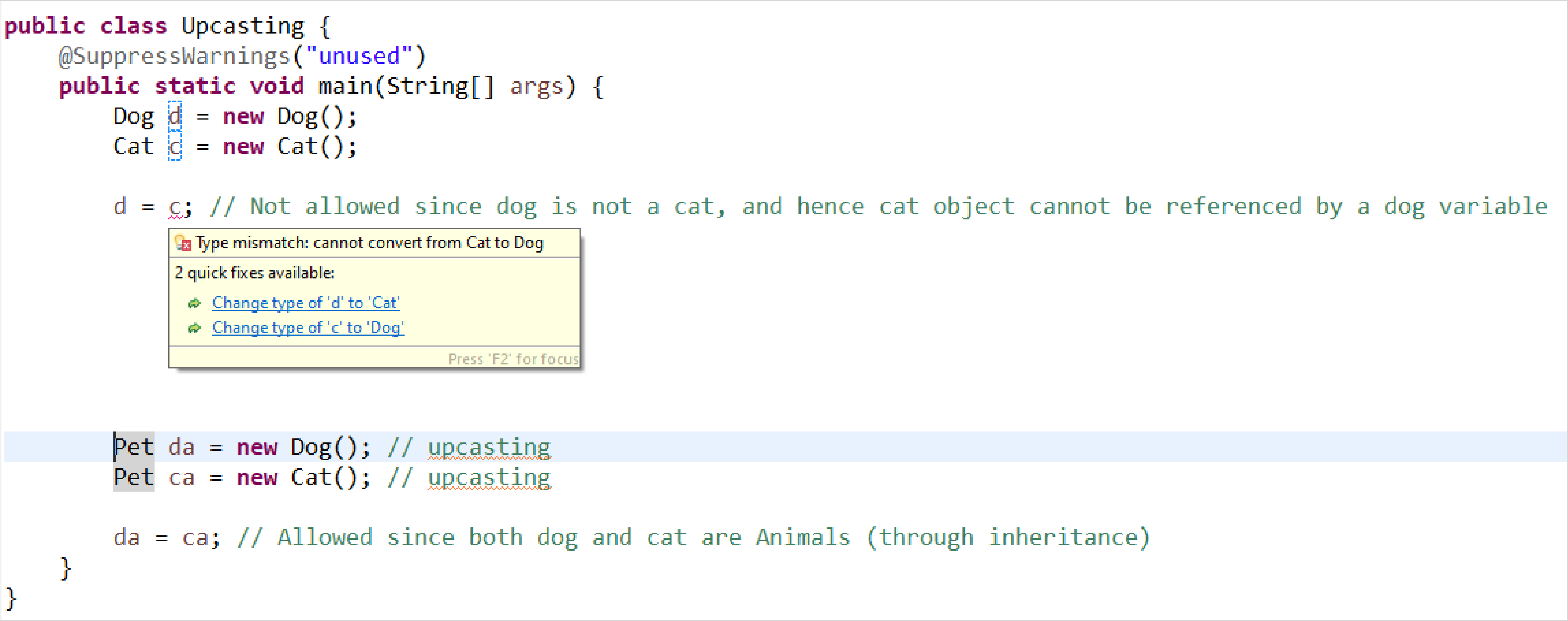
```java
public class Pet {

    public Pet getNew() {
        return new Pet();
    }
}
```

```java
public class Dog extends Pet {

    @Override
    public Dog getNew() {
        return new Dog();
    }
}
```

In the above example, the getNew() method in Dog class is considered to be an overridden method of the same method in Pet class.

This is because Dog is a subclass of Pet. If Dog did not extend Pet, then while method can stay valid, it will no longer be an overridden method and the `@Override` annotation would need to be removed

Runtime Polymorphism



```
public class Upcasting {
    @SuppressWarnings("unused")
    public static void main(String[] args) {
        Dog d = new Dog();
        Cat c = new Cat();

        d = c; // Not allowed since dog is not a cat, and hence cat object cannot be referenced by a dog variable




        Pet da = new Dog(); // upcasting
        Pet ca = new Cat(); // upcasting

        da = ca; // Allowed since both dog and cat are Animals (through inheritance)
    }
}
```

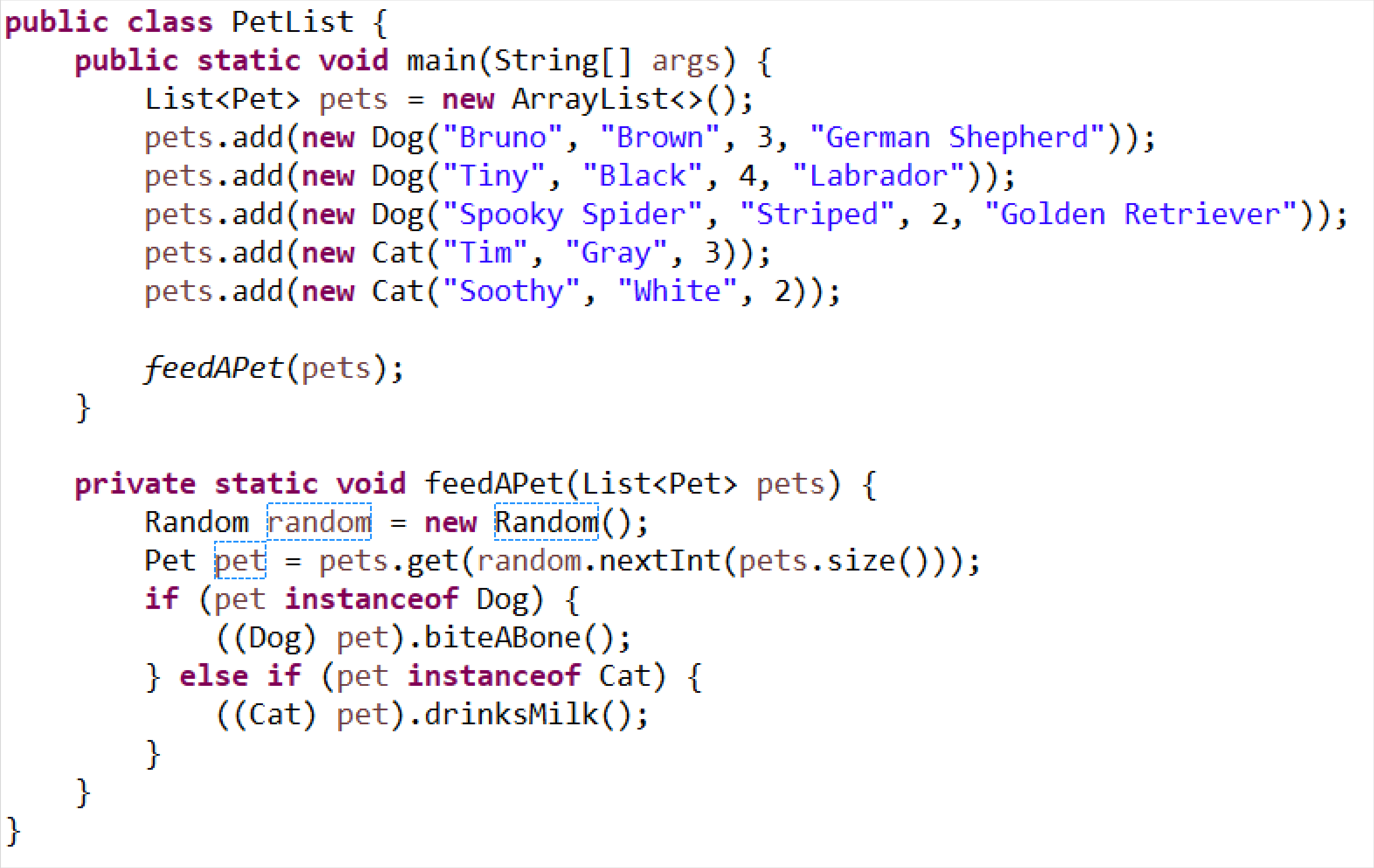
Type mismatch: cannot convert from Cat to Dog

2 quick fixes available:

- Change type of 'd' to 'Cat'
- Change type of 'c' to 'Dog'

Press 'F2' for focus

Runtime Polymorphism - Usage



```java
public class PetList {
    public static void main(String[] args) {
        List<Pet> pets = new ArrayList<>();
        pets.add(new Dog("Bruno", "Brown", 3, "German Shepherd"));
        pets.add(new Dog("Tiny", "Black", 4, "Labrador"));
        pets.add(new Dog("Spooky Spider", "Striped", 2, "Golden Retriever"));
        pets.add(new Cat("Tim", "Gray", 3));
        pets.add(new Cat("Soothy", "White", 2));

        feedAPet(pets);
    }

    private static void feedAPet(List<Pet> pets) {
        Random random = new Random();
        Pet pet = pets.get(random.nextInt(pets.size()));
        if (pet instanceof Dog) {
            ((Dog) pet).biteABone();
        } else if (pet instanceof Cat) {
            ((Cat) pet).drinksMilk();
        }
    }
}
```

Even if list is created of `Pet`, since both `Dog` and `Cat` are `Pet`, both can be accommodated in the same `List<Pet>`

During run time, we can use `instanceOf` to find which sub class method to invoke.

If we override the right methods, then we do not even need `instanceOf` handling.