TECHIN

10 - OBJEKTINIS PROGRAMAVIMAS (III)

Jaroslav Grablevski

Turinys

- casting, instanceof
- Polimorfizmas
- Abstrakčios klasės
- Interfeisai

Casting

Compiles but fails later:

```
class Animal {}
class Dog extends Animal {}
class DogTest {
   public static void main(String[] args) {
        Animal animal = new Animal();
        Dog d = (Dog) animal; // compiles but fails later
   }
}
```

Will NOT compile:

```
Animal animal = new Animal();
String s = (String) animal; //animal can't EVER be a String
```

instanceof

- The instanceof operator tests if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

```
class Animal {}
class Dog extends Animal {}
```

```
Dog dog = new Dog();
Animal animal = new Animal();
System.out.println(dog instanceof Dog);    //true
System.out.println(dog instanceof Animal); //true
System.out.println(dog instanceof Object); //true

System.out.println(animal instanceof Dog); //false
dog = null;
System.out.println(dog instanceof Dog);    //false
```

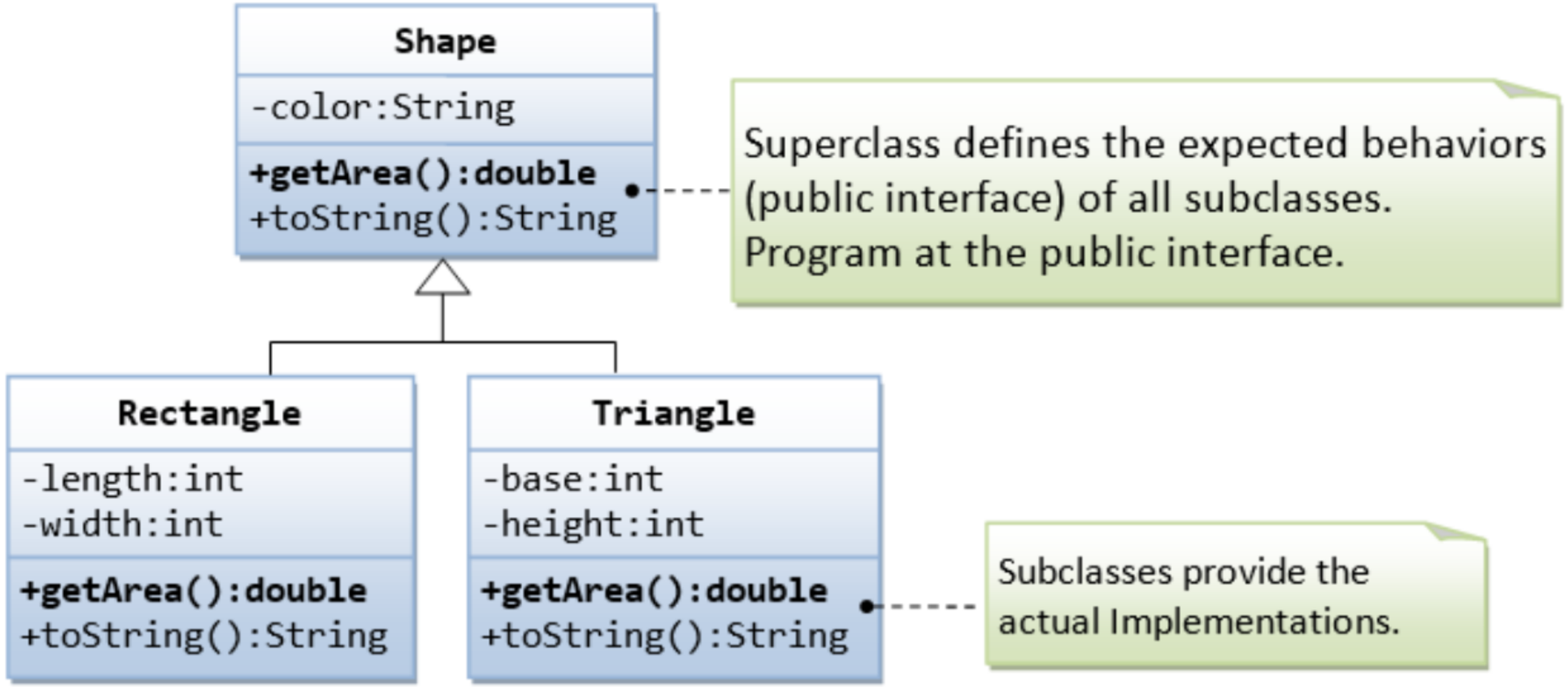
Casting

```
class Animal {
    void makeNoise() {
        System.out.println("generic noise");
    }}
class Dog extends Animal {
    void makeNoise() {
        System.out.println("bark");
    }
    void playDead() {
        System.out.println("roll over");
    }}
class CastTest2 {
    public static void main(String[] args) {
        Animal[] a = { new Animal(), new Dog(), new Animal() };
        for (Animal animal : a) {
            animal.makeNoise();
            if (animal instanceof Dog) {
                animal.playDead(); // <- won't compile
                Dog d = (Dog) animal; // downcast
                d.playDead();
    }}}}
```

Polimorfizmas

- objektiniame programavime naudojama sąvoka, kai metodas gali būti vykdomas skirtingai, priklausomai nuo konkrečios klasės realizacijos, metodo kvietėjui nieko nežinant apie tokius skirtumus.
- “moku dirbti su vienu tipu, moku dirbti su visais tipais, kurie įgyvendina tą patį kontraktą”
  - Grįstas paveldėjimu ir įgyvendinimu*
  - Jei Y extends X, Y galime naudoti visur kur tikimes gauti X

Polimorfizmas



Polimorfizmas pvz.

```
class MythicalAnimal {
    public void hello() {
        System.out.println("Hello, I'm an unknown animal");
    }
}
class Chimera extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Hello! Hello!");
    }
}
class Dragon extends MythicalAnimal {
    @Override
    public void hello() {
        System.out.println("Rrrr...");
    }
}
```

Polimorfizmas pvz.

```
Chimera chimera = new Chimera();
Dragon dragon = new Dragon();
MythicalAnimal animal = new MythicalAnimal();
```

```
MythicalAnimal chimera = new Chimera();
MythicalAnimal dragon = new Dragon();
MythicalAnimal animal = new MythicalAnimal();
```

```
chimera.hello(); // Hello! Hello!
dragon.hello(); // Rrrr...
animal.hello(); // Hello, i'm an unknown animal
```

Polimorfizmas pvz.

```
public class Shape {
    // All shapes must have a method called getArea().
    public double getArea() {
        System.err.println("Shape unknown! Cannot compute area!");
        return 0; // We need to return some value to compile the program.
    }
}
```

```
public class Rectangle extends Shape {
    private int length, width;
    public Rectangle(int length, int width) {..
    @Override // Override to provide the proper implementation
    public double getArea() {
        return length * width;
    }
}
```

```
public class Triangle extends Shape {
    //...
    public double getArea() {
        return 0.5 * base * height;
    }
    //..
```

Polimorfizmas pvz.

```
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle(4, 5); // Upcast
        System.out.println("Area is " + s1.getArea()); // Runs Rectangle's getArea()

        Shape s2 = new Triangle(4, 5); // Upcast
        System.out.println("Area is " + s2.getArea()); // Runs Triangle's getArea()

    }
}
```

```
Area is 20.0
Area is 10.0
```

Polimorfizmas pvz. 2

```
public class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}
```

```
public class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay ");
    }

    public void eat(String s) {
        System.out.println("Horse eating " + s);
    }
}
```

Polimorfizmas pvz. 2

Method Invocation Code	Result
`Animal a = new Animal();` `a.eat();`	`Generic Animal Eating Generically`
`Horse h = new Horse();` `h.eat();`	`Horse eating hay`
`Animal ah = new Horse();` `ah.eat();`	`Horse eating hay` Polymorphism works—the actual object type (`Horse`), not the reference type (`Animal`), is used to determine which `eat()` is called.
`Horse he = new Horse();` `he.eat("Apples");`	`Horse eating Apples` The overloaded `eat(String s)` method is invoked.
`Animal a2 = new Animal();` `a2.eat("treats");`	Compiler error! Compiler sees that the `Animal` class doesn't have an `eat()` method that takes a `String`.
`Animal ah2 = new Horse();` `ah2.eat("Carrots");`	Compiler error! Compiler still looks only at the reference and sees that `Animal` doesn't have an `eat()` method that takes a `String`. Compiler doesn't care that the actual object might be a `Horse` at runtime.

Dynamic binding

```
public class DynamicBinding {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
```

```
class GraduateStudent extends Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person {
    public String toString() {
        return "Person";
    }
}
```

```
Student
Student
Person
java.lang.Object@7d6f77cc
```

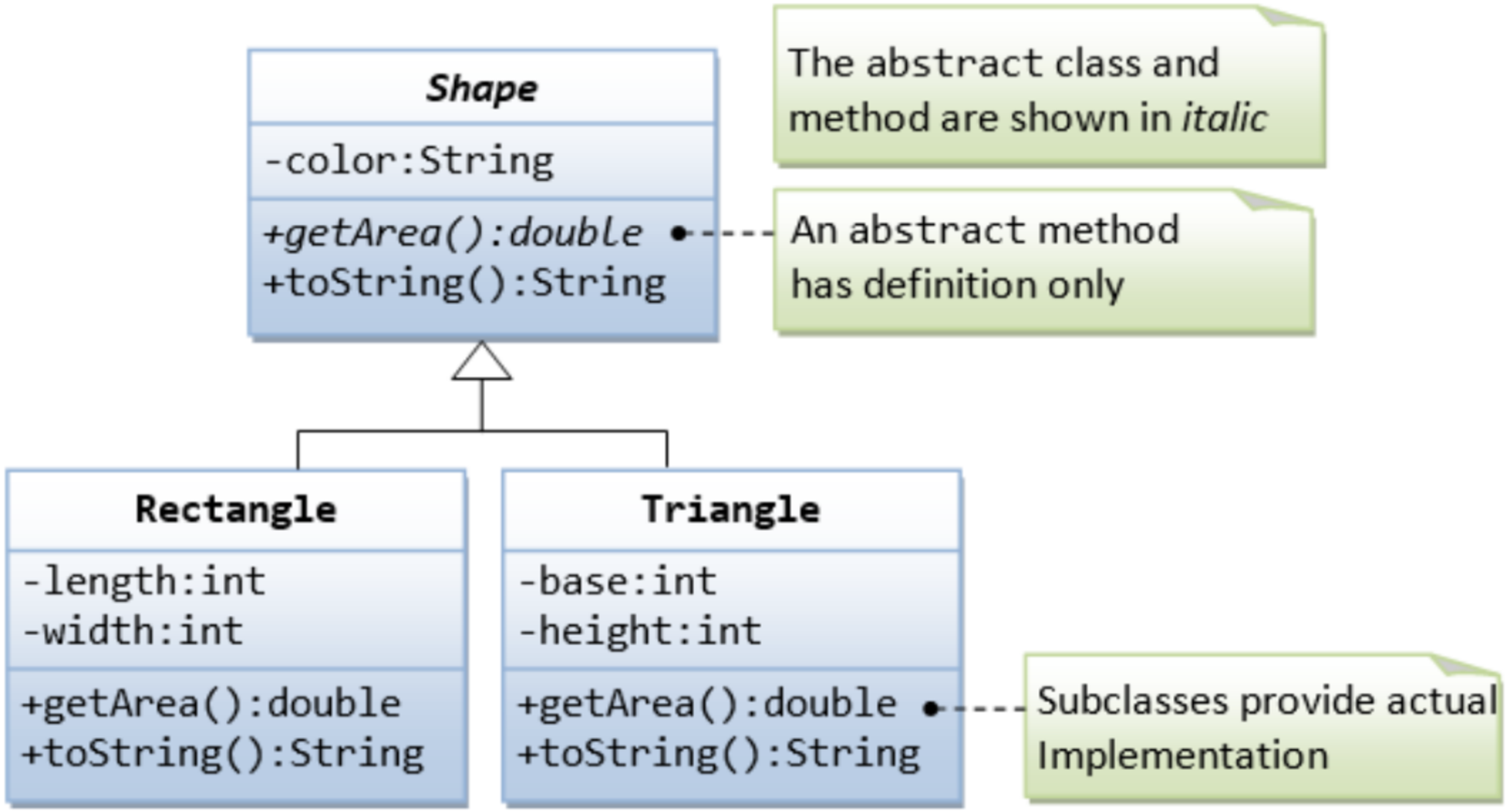
Abstrakti klasė

- Abstrakčios klasės turi būti pažymėtos raktažodžiu *abstract*;
- Negalima sukurti objektų iš abstrakčių klasių;
- Gali turėti nerealizuotų/abstrakčių metodų;
- Abstrakti klasė skirta paveldėjimui - paveldinti klasė turi realizuoti visus abstrakčius metodus arba pati būti abstrakčia;

Abstrakti klasė

- Iš abstrakčios klasės objektai negali būti sukurti
- Abstrakčios klasės negali būti pažymėtos private bei final raktažodžiais
- Abstrakčios klasės gali deklaruoti nuo 0 ir daugiau abstrakčių metodų
  - Abstraktūs metodai gali būti deklaruoti tik abstrakčiose klasėse
  - Abstraktūs metodai negali būti pažymėti raktažodžiais private bei final
  - Abstraktūs metodai negali turėti metodo kūno (realizacijos)
- Jei abstrakti klasė išplečia kitą abstrakčią klasę, tuomet ji paveldi visus tėvinės klasės abstrakčius metodus
- Pirma ne abstrakti klasė, kuri išplečia abstrakčią klasę, privalo realizuoti visus abstrakčius metodus.

Abstrakti klasė



Abstrakti klasė

```
abstract public class Shape {
    // Private member variable
    private String color;

    // All Shape subclasses must implement a method called getArea()
    abstract public double getArea();

    // Constructor
    public Shape(String color) {
        this.color = color;
    }

    @Override
    public String toString() {
        return "Shape [color=" + color + "]";
    }
}
```

```
Shape s3 = new Shape("green");    // Compilation Error!!
```

Abstrakti klasė

```
public abstract class Pet {

    protected String name;
    protected int age;

    protected Pet(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public abstract void say(); // an abstract method
}
```

Abstrakti klasė

```java
class Dog extends Pet {

// It can have additional fields as well

    public Dog(String name, int age) {
        super(name, age);
    }

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

```java
class Cat extends Pet {

//It can have additional fields as well

    public Cat(String name, int age) {
        super(name, age);
    }

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

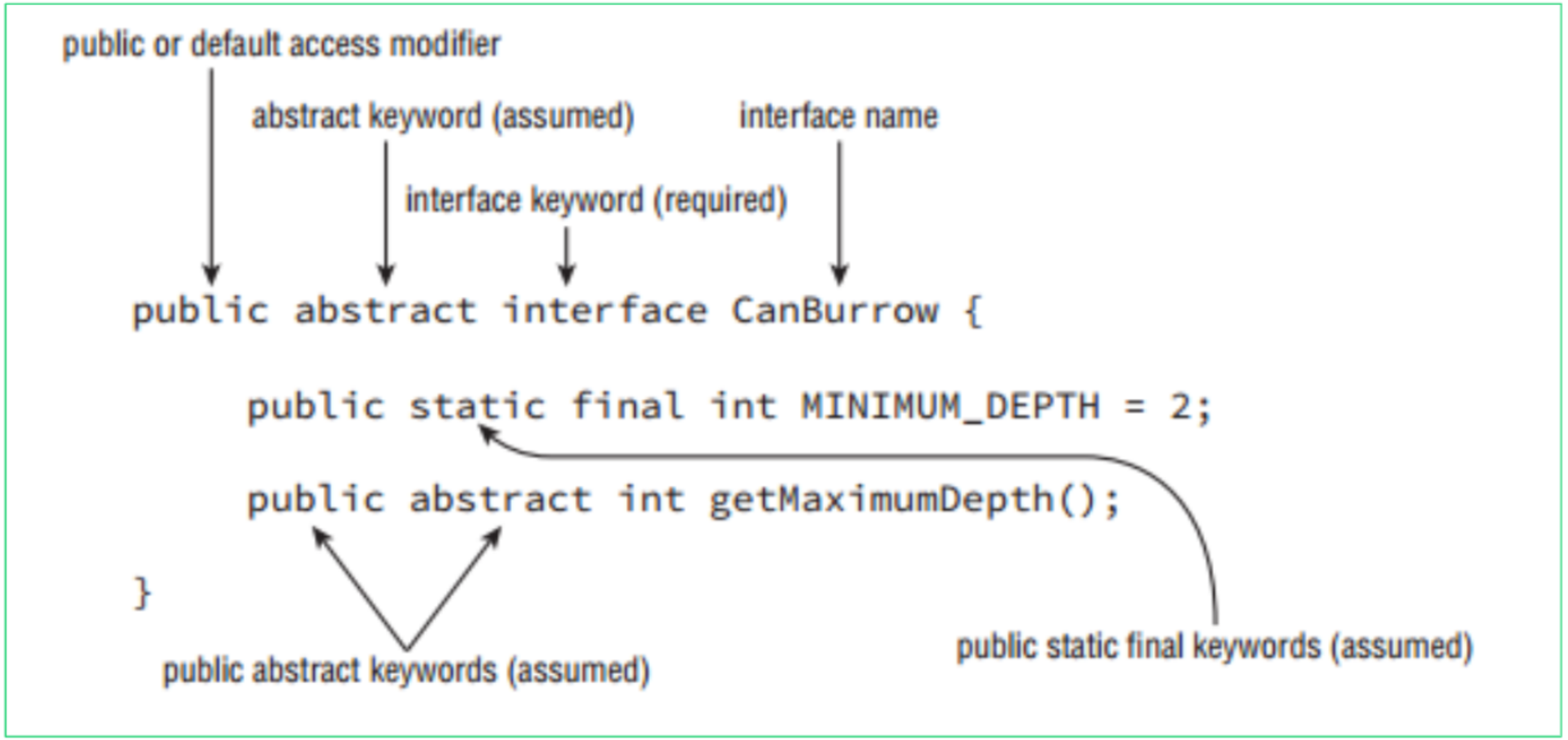
Interfeisas

- Interfeisas - tai abstraktus duomenų tipas, kuris apibrėžia aibę viešų metodų, kuriuos visos klasės, realizuojančios šį interfeisą, privalo taip pat realizuoti.
- Naudojama nusakyti kontraktą
  - “Aš esu …”
  - “Aš moku …”

```
public interface Flyable {
    void fly();
}

// Klasė Bird realizuoja interfeisą Flyable
class Bird implements Flyable {
    public void fly() {
        System.out.println("Bird is flying");
    }
}
```

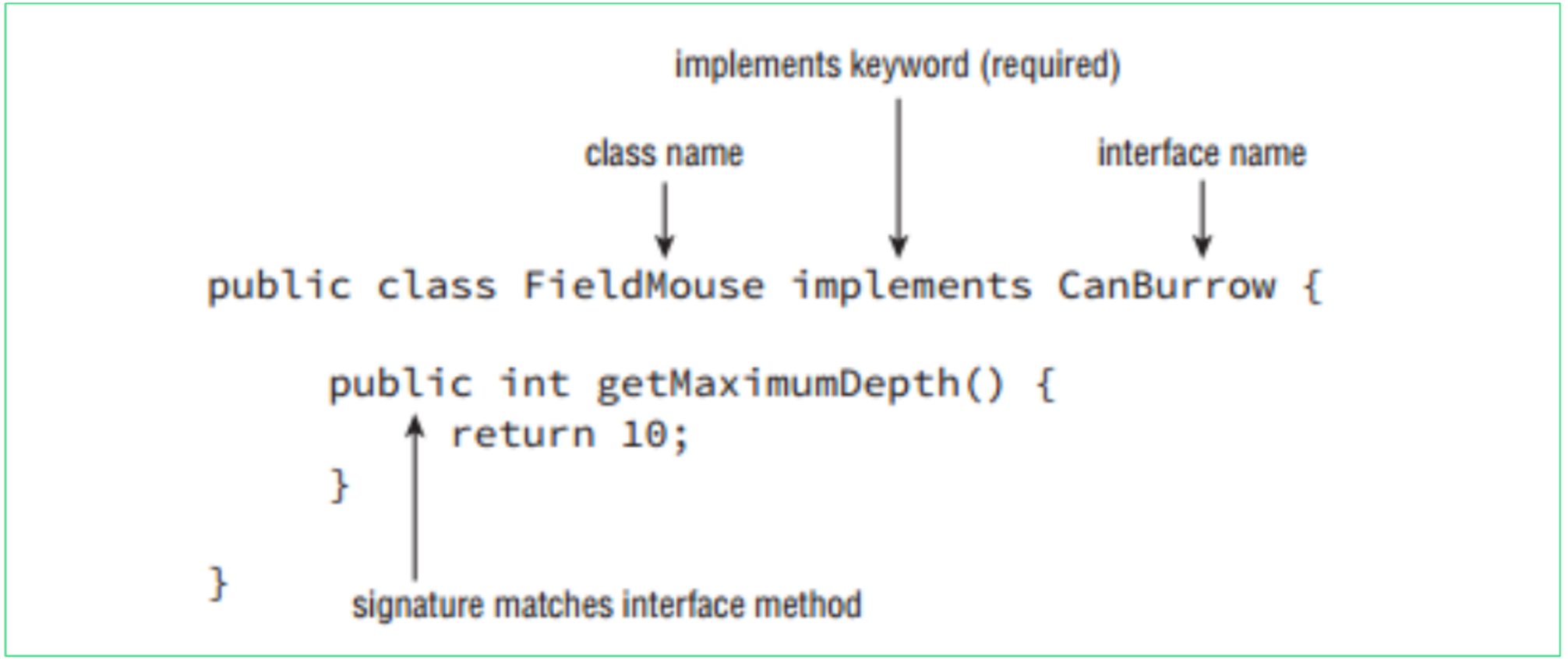
Interfeisas

- Defining an interface

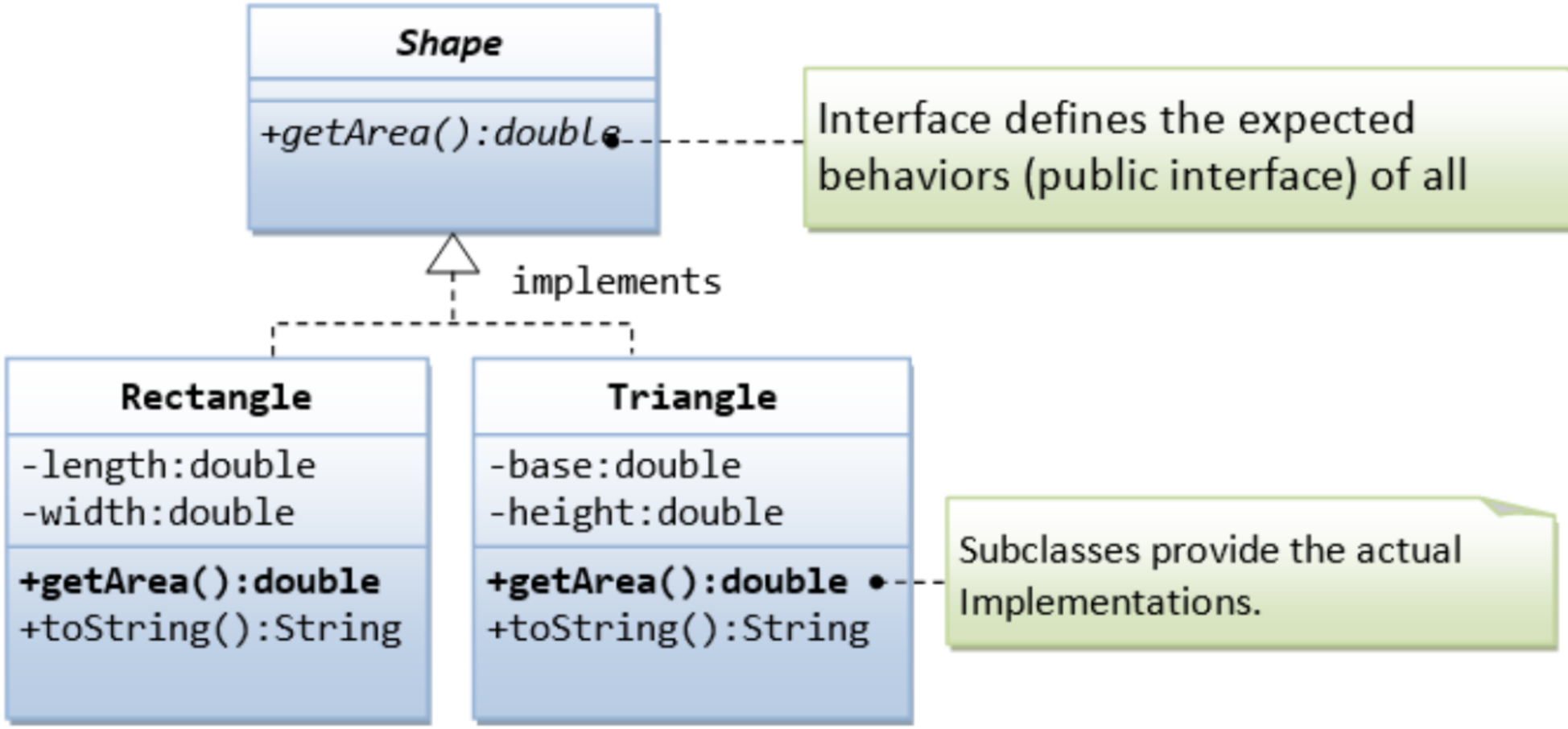


Interfeisas

- Implementing an interface



Interfeisas



Interfeisas

- Klasė gali realizuoti kelis interfeisus. Tokiu atveju ji privalo realizuoti visų interfeisų abstrakčius metodus.

```
interface Flyable {
    void fly();
}

interface Walkable {
    void walk();
}
```

```
class Bird implements Flyable, Walkable {
    public void fly() {
    System.out.println("I’m flying"); }

    public void walk() {
    System.out.println("I’m walking");
    }
}
```

Interfeisas

- Visi interfeise deklaruoti kintamieji - konstantos. T.y. jie automatiškai yra *public static final*. Todėl jų nėra būtina pažymėti minėtais raktažodžiais.
- Kadangi kintamieji yra konstantos, todėl jiems privaloma iškarto priskirti konkrečias reikšmes.

```
interface Runner {
    // Tas pats kas deklaruotume tokiu būdu:
    // public static final int MAX_SPEED = 30;
    int MAX_SPEED = 30;
}
```

Interfeisas > metodai

- Interfeise galime apibrėžti abstrakčius (*abstract*) metodus.
  - Šie metodai neturi realizacijos ir yra automatiškai vieši.
  - Juos privalo realizuoti klasės, kurios realizuoja šį interfeisą.
- Interfeise galima apibrėžti numatytuosius (*default*) metodus.
  - Tai tokie metodai, kurie turi realizaciją
  - Numatytieji metodai privalo būti deklaruoti su raktažodžiu *default*.
  - Numatytieji metodai nėra static, final arba abstract.
  - Gali (bet neprivalo) būti perrašyti konkretų interfeisą realizuojančiose klasėse.
- Interfeise galima apibrėžti statinius metodus
  - Statiniai metodai interfeisuose nėra paveldimi juos realizuojančiose klasėse
- Nuo 9 java versiojos galima apibrėžti privačius (*private*) metodus

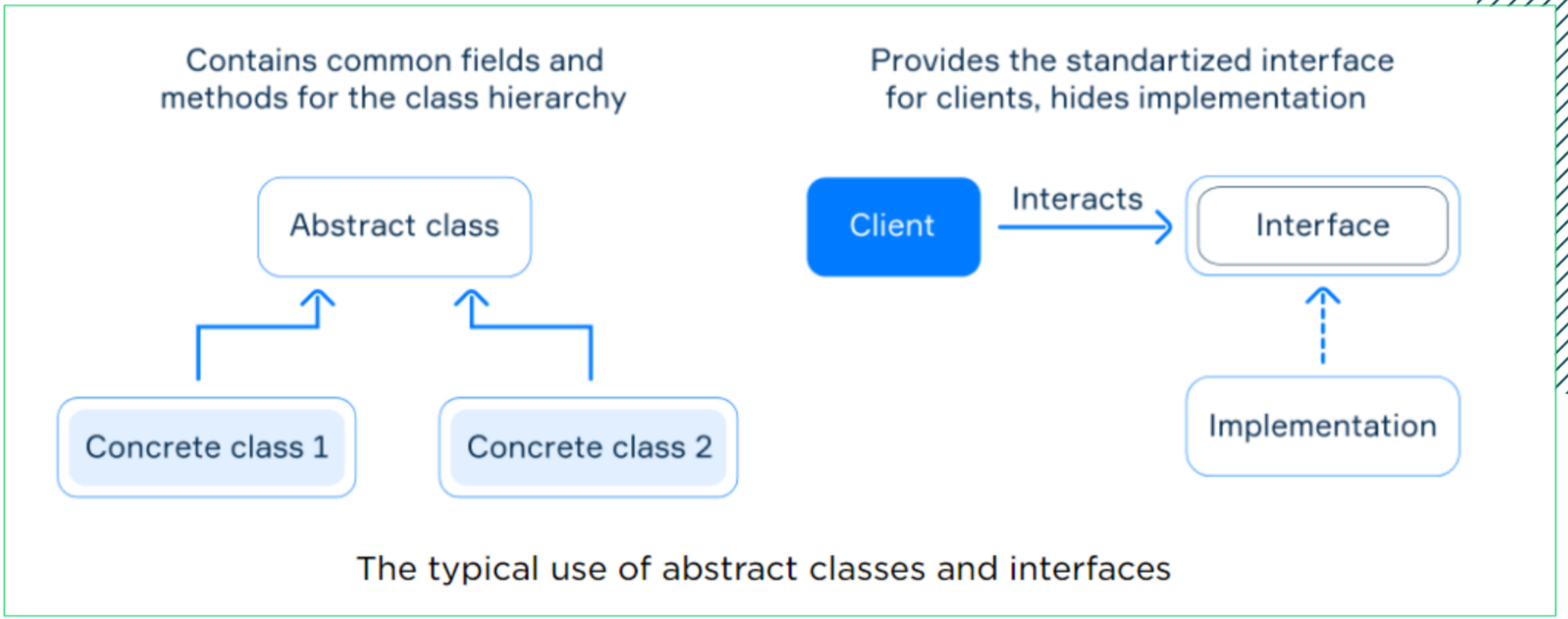
Interfeisas > metodai

```
public interface Interface {

    int INT_CONSTANT = 0; // public static final

    void instanceMethod1();// public abstract

    void instanceMethod2();

    static void staticMethod() {
        System.out.println("Interface: static method");
    }

    default void defaultMethod() {
        System.out.println("Interface: default method."
                + " It can be overridden");
    }
}
```

Interface vs Abstract class

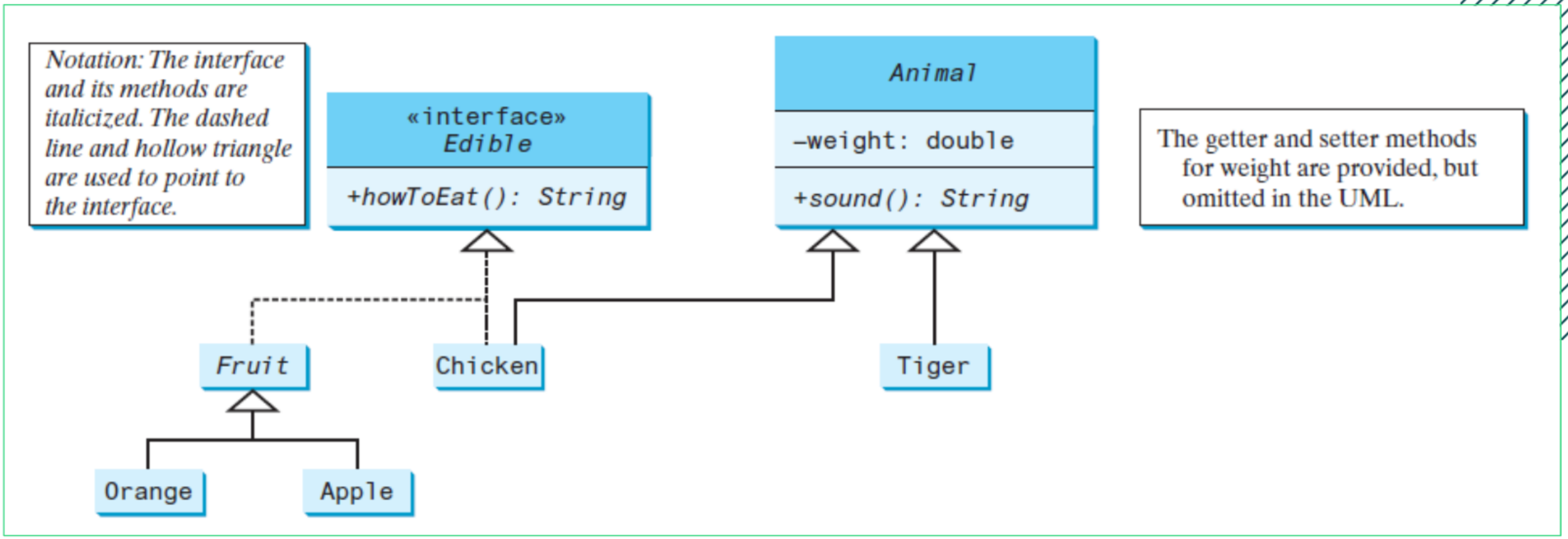
	Java 7 and Earlier	Java 8 and Later
Abstract Classes	• Can have concrete methods and abstract methods • Can have static methods • Can have instance variables • Class can directly extend one	(Same as Java 7)
Interfaces	• Can only have abstract methods – no concrete methods • Cannot have static methods • Cannot have mutable instance variables • Class can implement any number	• Can have concrete (default) methods and abstract methods • Can have static methods • Cannot have mutable instance variables • Class can implement any number

Interface vs Abstract class



The typical use of abstract classes and interfaces

Interface vs Abstract class



Interface hierarhy

- An interface can extend another interface
  - Interfaces can form hierarchy in the same way as classes.
  - A concrete class must override all abstract methods that it has inherited, regardless of where they were defined.