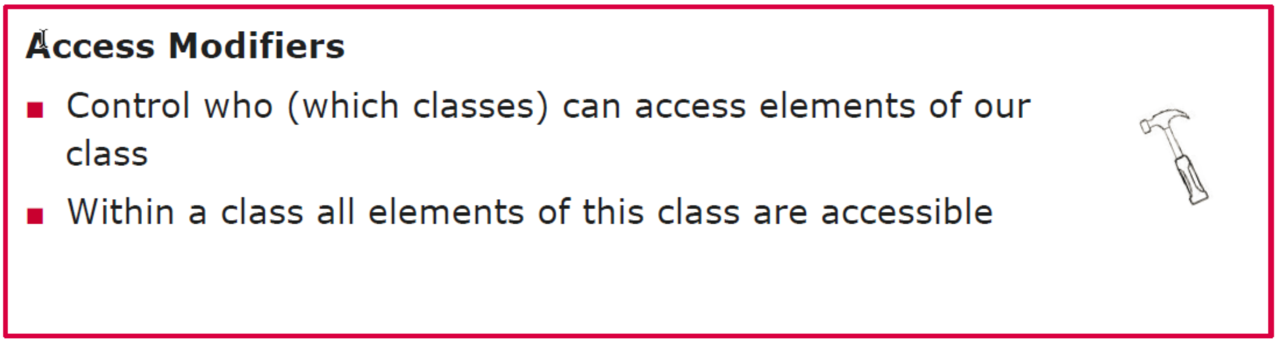
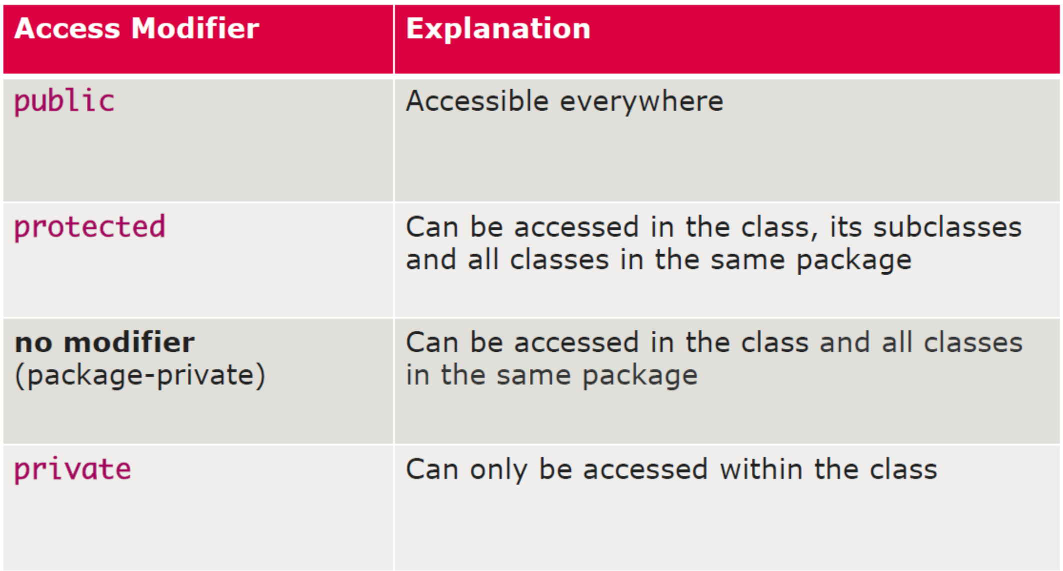
Summary Week 3

Access Control

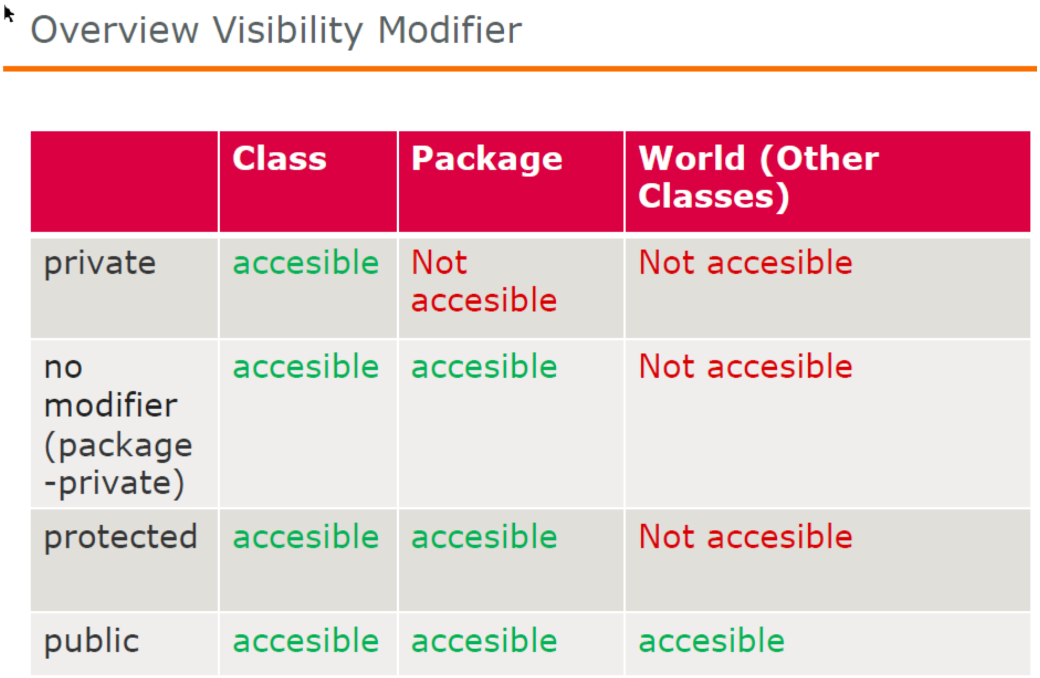


Access Modifiers

- Control who (which classes) can access elements of our class
- Within a class all elements of this class are accessible

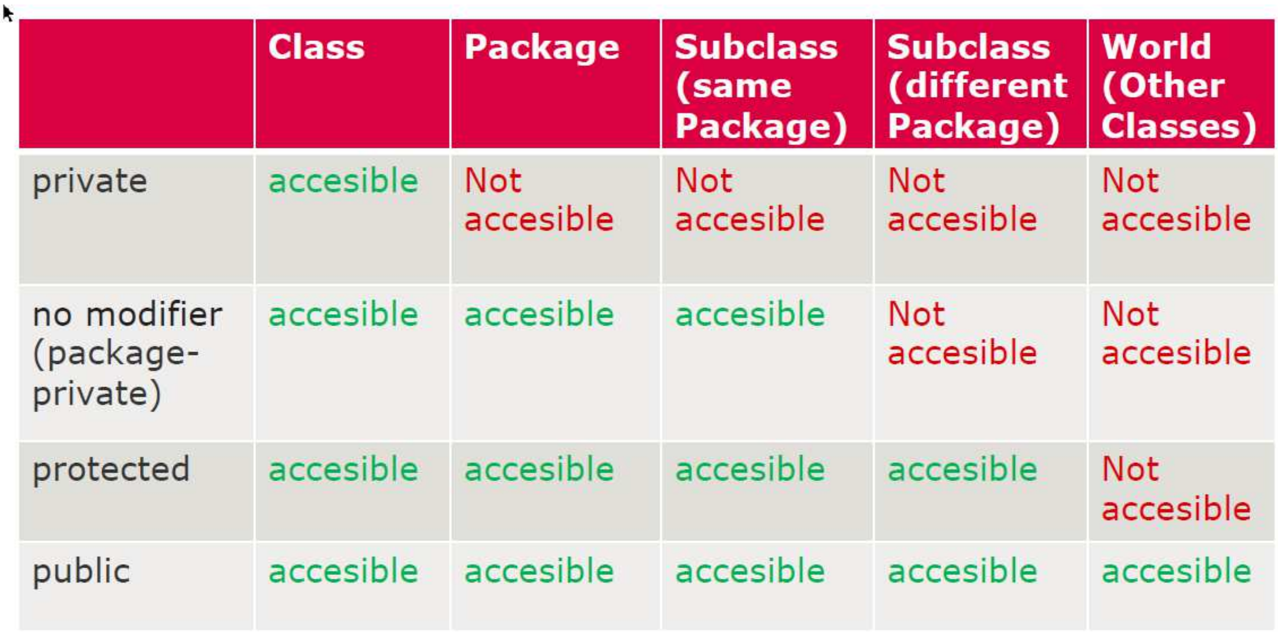


Access Modifier	Explanation
public	Accessible everywhere
protected	Can be accessed in the class, its subclasses and all classes in the same package
no modifier (package-private)	Can be accessed in the class and all classes in the same package
private	Can only be accessed within the class

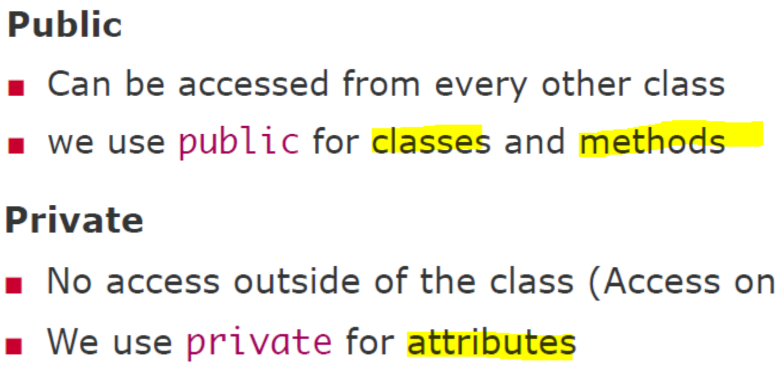


Overview Visibility Modifier

	Class	Package	World (Other Classes)
private	accesible	Not accesible	Not accesible
no modifier (package-private)	accesible	accesible	Not accesible
protected	accesible	accesible	Not accesible
public	accesible	accesible	accesible



	Class	Package	Subclass (same Package)	Subclass (different Package)	World (Other Classes)
private	accesible	Not accesible	Not accesible	Not accesible	Not accesible
no modifier (package-private)	accesible	accesible	accesible	Not accesible	Not accesible
protected	accesible	accesible	accesible	accesible	Not accesible
public	accesible	accesible	accesible	accesible	accesible

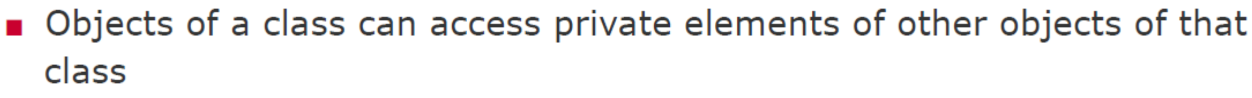


Public

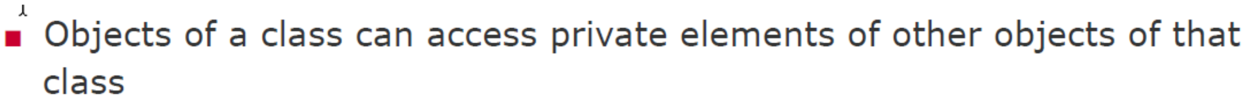
- Can be accessed from every other class
- we use public for classes and methods

Private

- No access outside of the class (Access only within the same class)
- We use private for attributes

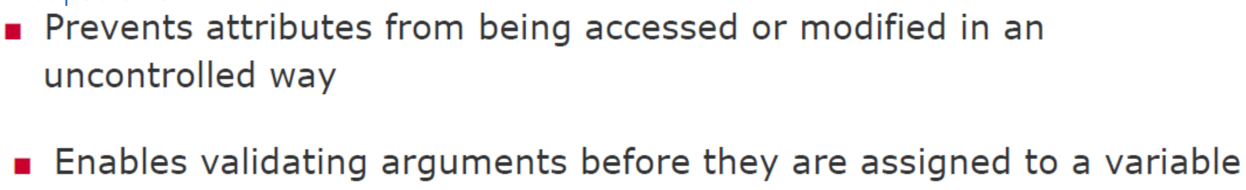


- Objects of a class can access private elements of other objects of that class

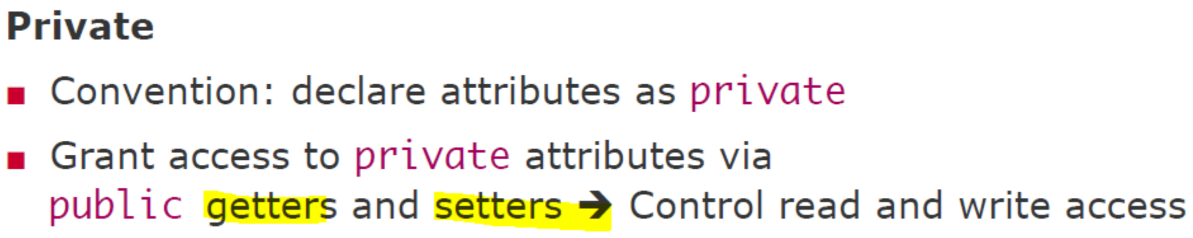


- Objects of a class can access private elements of other objects of that class

Encapsulation

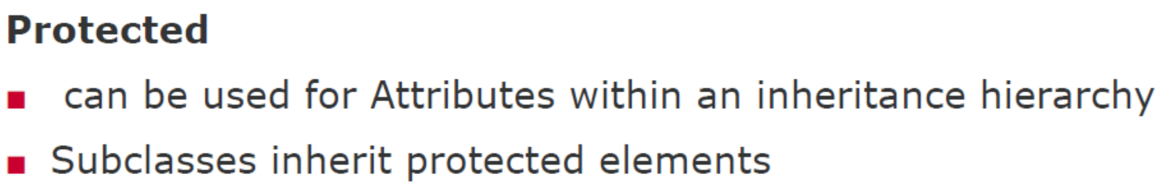


- Prevents attributes from being accessed or modified in an uncontrolled way
- Enables validating arguments before they are assigned to a variable



Private

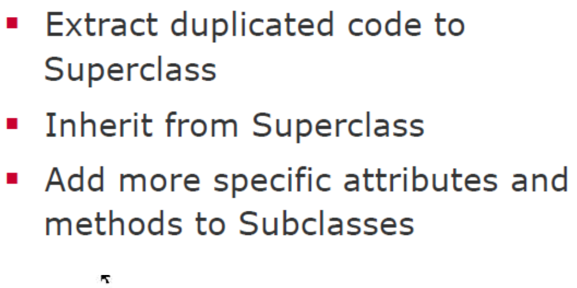
- Convention: declare attributes as private
- Grant access to private attributes via public getters and setters ➔ Control read and write access



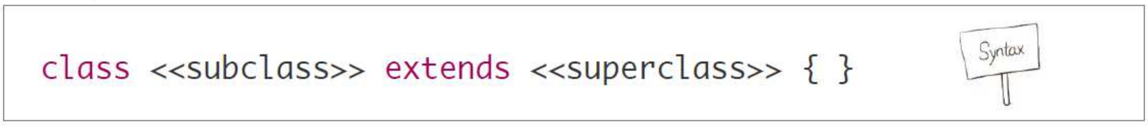
Protected

- can be used for Attributes within an inheritance hierarchy
- Subclasses inherit protected elements

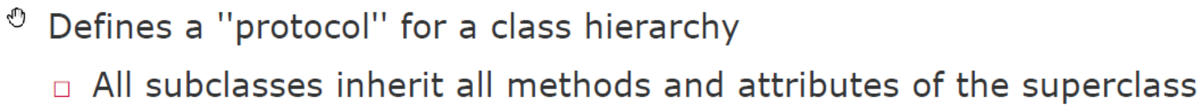
Inheritance



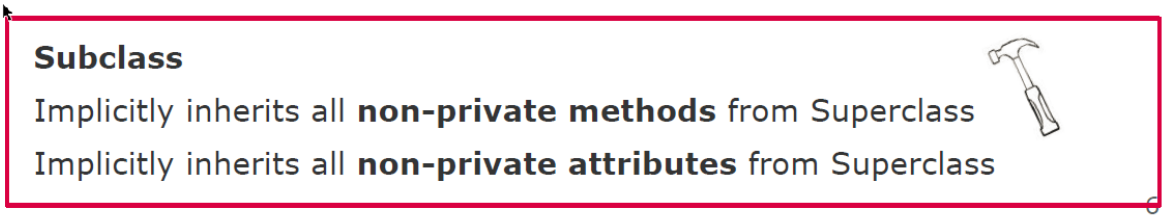
- Extract duplicated code to Superclass
- Inherit from Superclass
- Add more specific attributes and methods to Subclasses



```
class <<subclass>> extends <<superclass>> { }
```



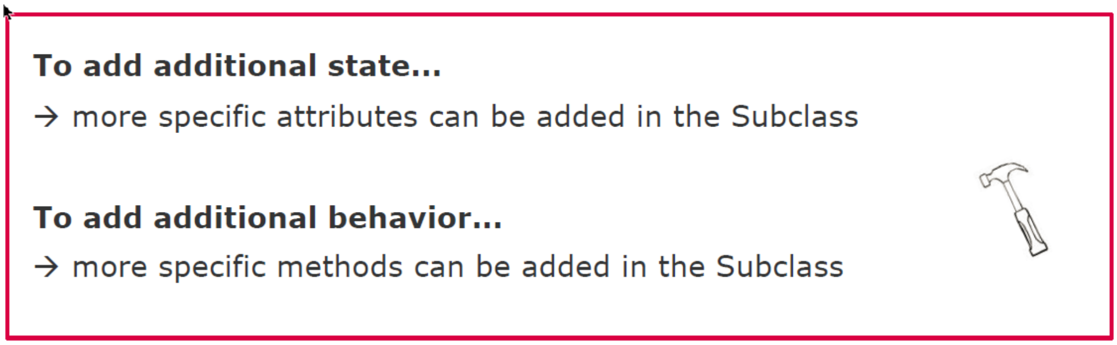
- Defines a "protocol" for a class hierarchy
  - □ All subclasses inherit all methods and attributes of the superclass



Subclass

Implicitly inherits all **non-private methods** from Superclass

Implicitly inherits all **non-private attributes** from Superclass

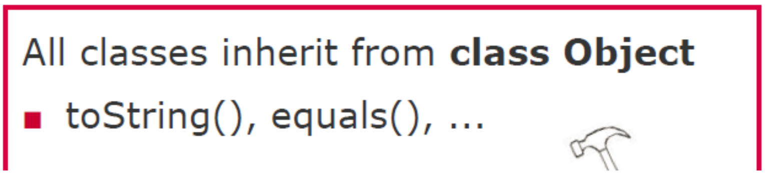


To add additional state…

→ more specific attributes can be added in the Subclass

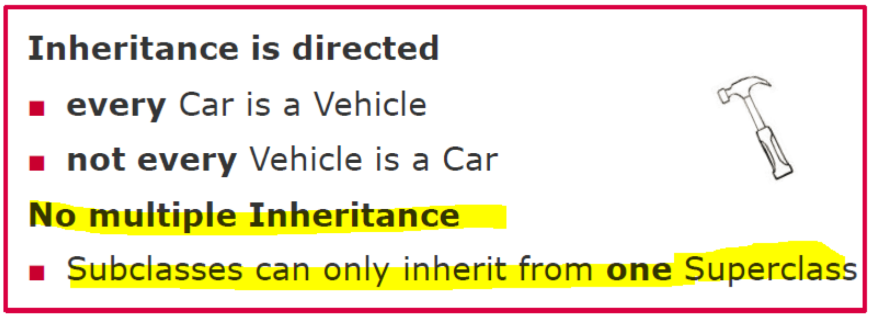
To add additional behavior…

→ more specific methods can be added in the Subclass



All classes inherit from **class Object**

- toString(), equals(), …

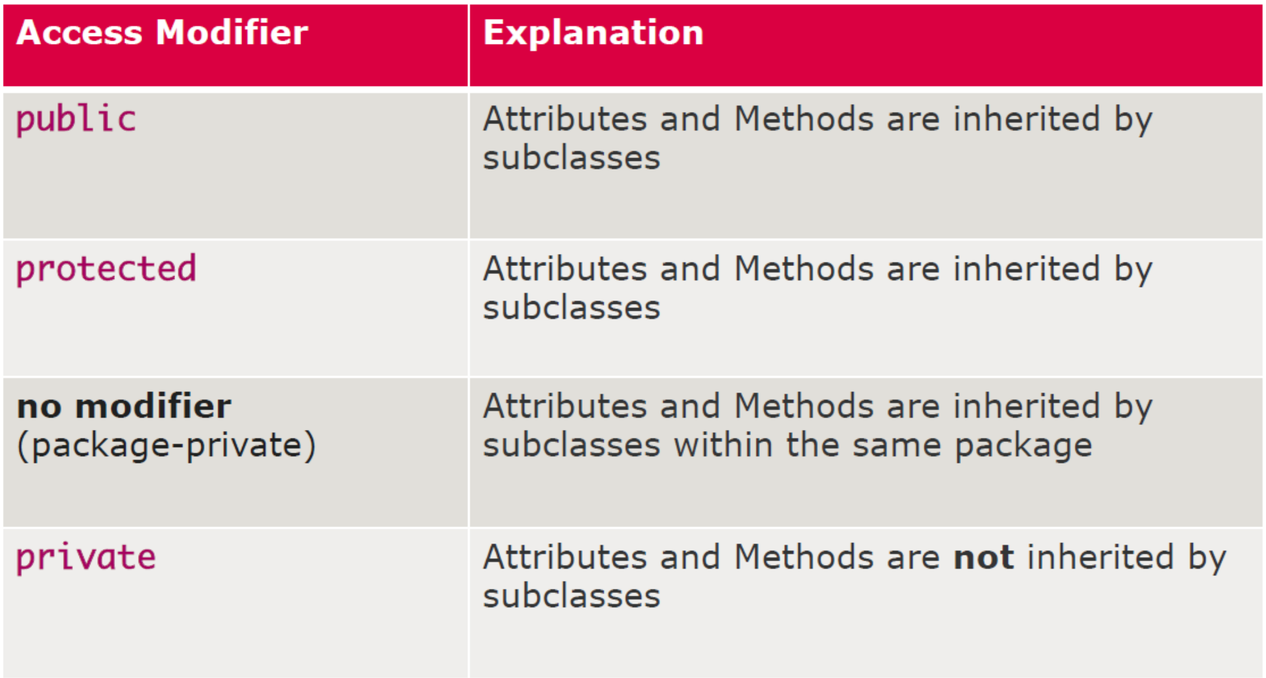


Inheritance is directed

- **every** Car is a Vehicle
- **not every** Vehicle is a Car

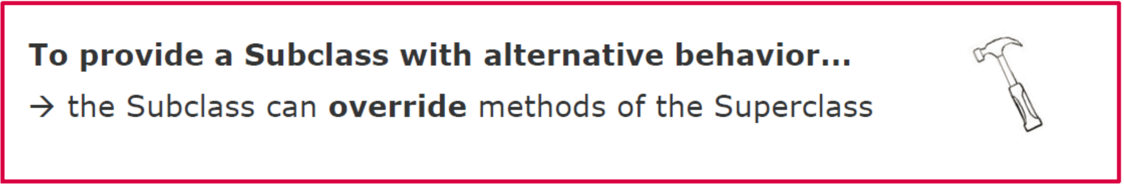
No multiple Inheritance

- Subclasses can only inherit from **one** Superclass

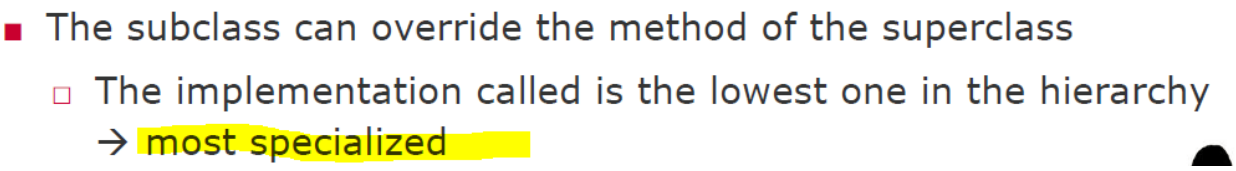


Access Modifier	Explanation
`public`	Attributes and Methods are inherited by subclasses
`protected`	Attributes and Methods are inherited by subclasses
no modifier (package-private)	Attributes and Methods are inherited by subclasses within the same package
`private`	Attributes and Methods are **not** inherited by subclasses

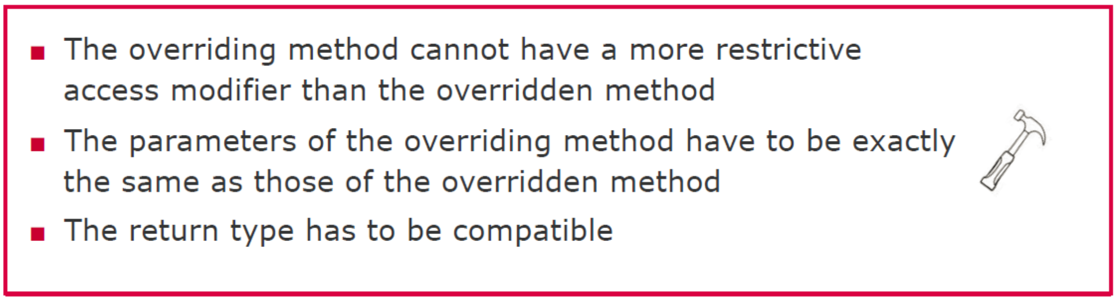
Override



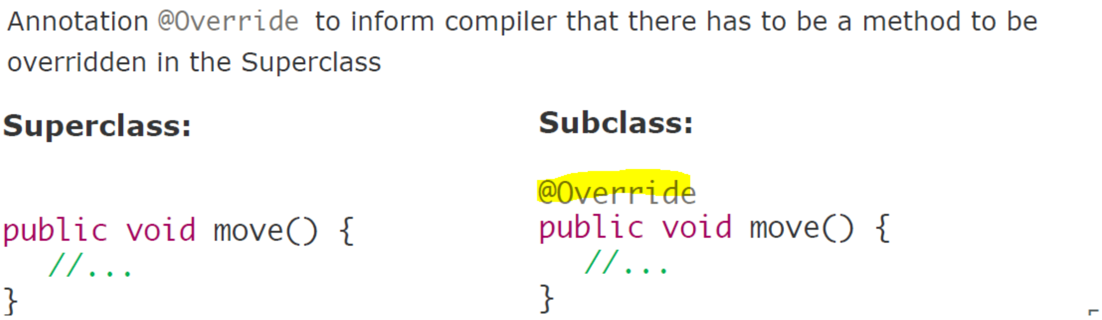
To provide a Subclass with alternative behavior…
→ the Subclass can **override** methods of the Superclass



- The subclass can override the method of the superclass
  - □ The implementation called is the lowest one in the hierarchy → most specialized



- The overriding method cannot have a more restrictive access modifier than the overridden method
- The parameters of the overriding method have to be exactly the same as those of the overridden method
- The return type has to be compatible



Annotation @Override to inform compiler that there has to be a method to be overridden in the Superclass

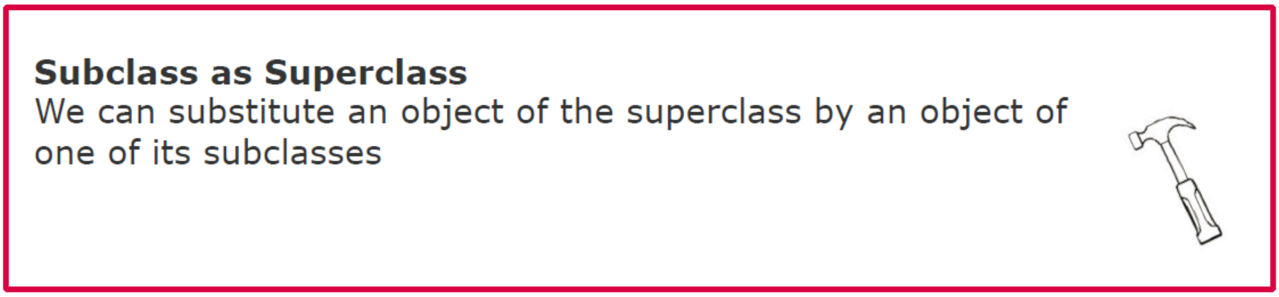
Superclass:

```
public void move() {
   //...
}
```

Subclass:

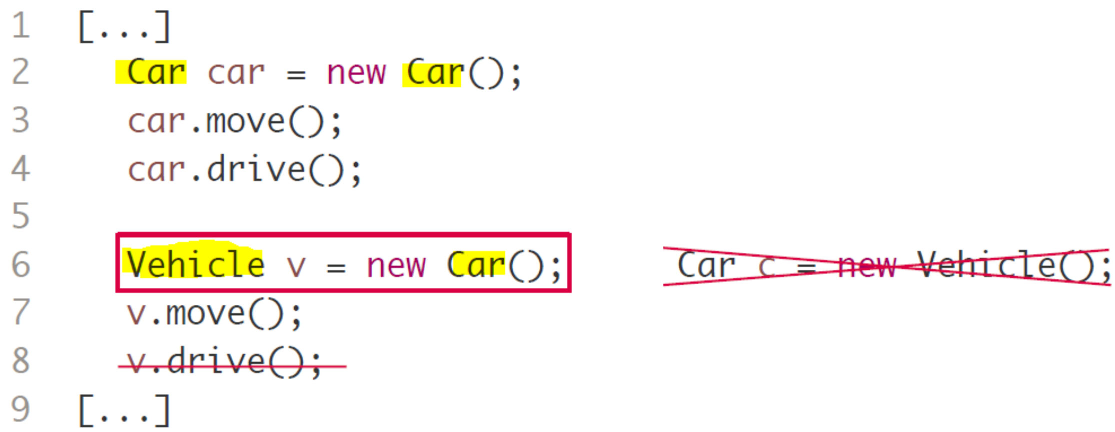
```
@Override
public void move() {
   //...
}
```

Polymorphism



Subclass as Superclass

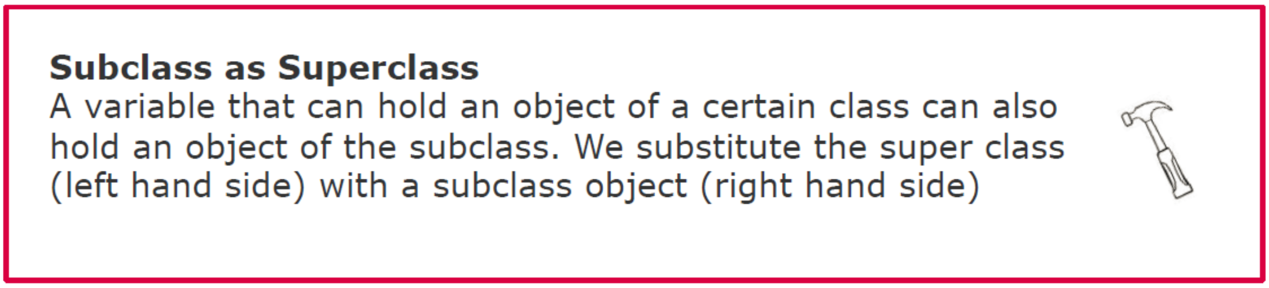
We can substitute an object of the superclass by an object of one of its subclasses



```
[...]
   Car car = new Car();
   car.move();
   car.drive();

   Vehicle v = new Car();
   v.move();
   ~~v.drive();~~
[...]
```

~~Car c = new Vehicle();~~



Subclass as Superclass

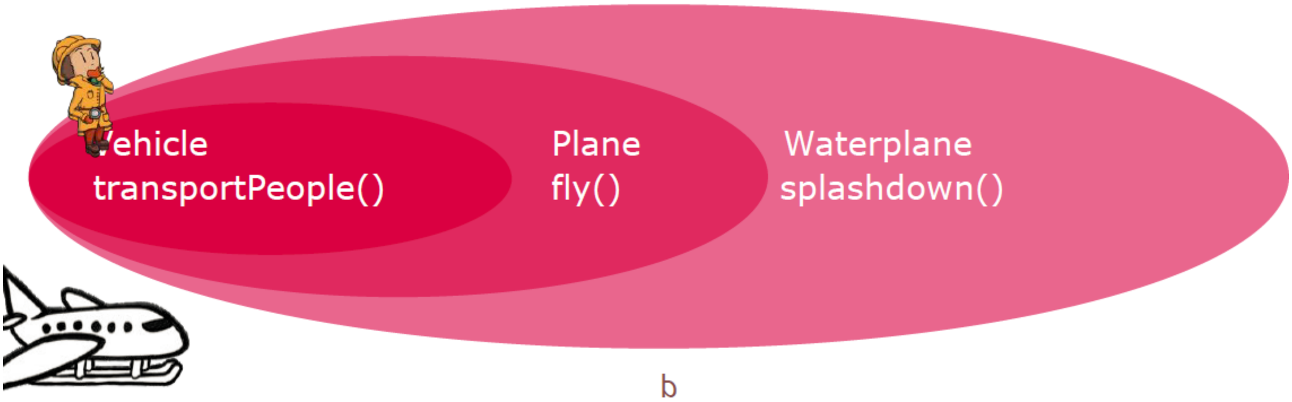
A variable that can hold an object of a certain class can also hold an object of the subclass. We substitute the super class (left hand side) with a subclass object (right hand side)

- Substitution works in every context
  - We can create an array of Vehicles
  - Pass Vehicles as parameters
  - Call methods on Vehicles
  - …
- The compiler determines whether a method can be called on the basis of the data type of the variable
- A variable can hold objects of its own type as well as of all its subtypes
- They comply to our protocol: per definition we can use a subclass in any place for a superclass
- The most specialized method is called

Polymorphism
If we call a method of a Superclass, we execute the most specialized (most appropriate) behavior for every (subclass) object.

- □ New subclasses do not need modifications on the method code

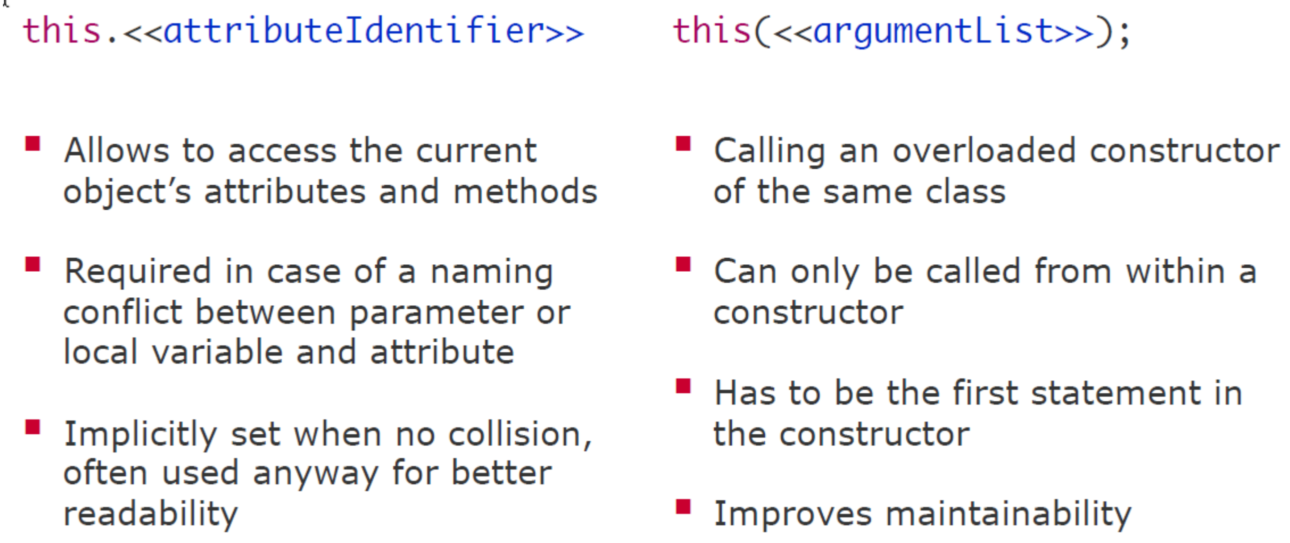
  → improved maintainability



b

- Vehicle Waterplane(); .tP(); ~~p.fly();~~ ~~p.splashdown();~~
- Plane p = new Waterplane(); p.tP(); p.fly(); ~~p.splashdown();~~
- Waterplane p = new Waterplane(); p.tP(); p.fly(); p.splashdown();

This is super

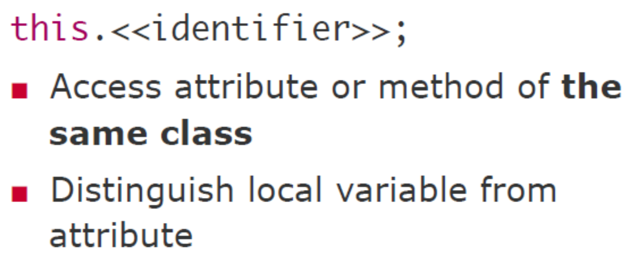


`this.<<attributeIdentifier>>`

- Allows to access the current object’s attributes and methods
- Required in case of a naming conflict between parameter or local variable and attribute
- Implicitly set when no collision, often used anyway for better readability

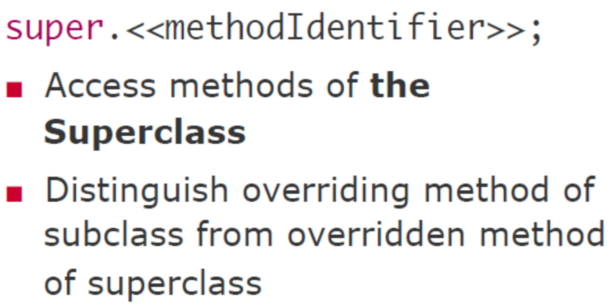
`this(<<argumentList>>);`

- Calling an overloaded constructor of the same class
- Can only be called from within a constructor
- Has to be the first statement in the constructor
- Improves maintainability



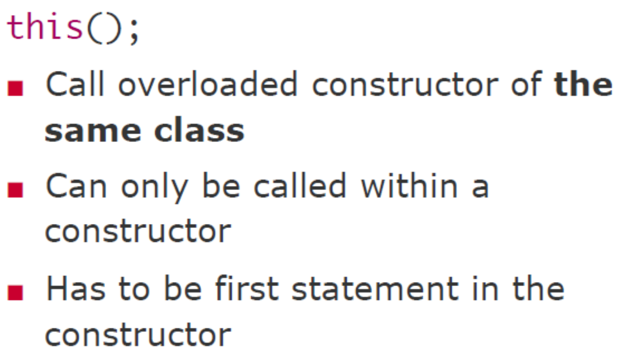
`this.<<identifier>>;`

- Access attribute or method of **the same class**
- Distinguish local variable from attribute



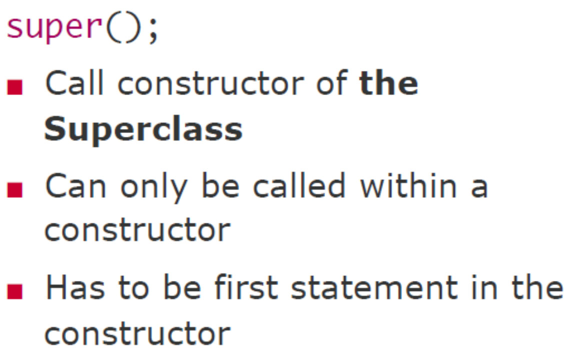
`super.<<methodIdentifier>>;`

- Access methods of **the Superclass**
- Distinguish overriding method of subclass from overridden method of superclass



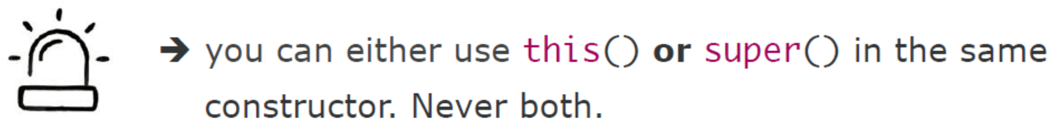
`this();`

- Call overloaded constructor of **the same class**
- Can only be called within a constructor
- Has to be first statement in the constructor



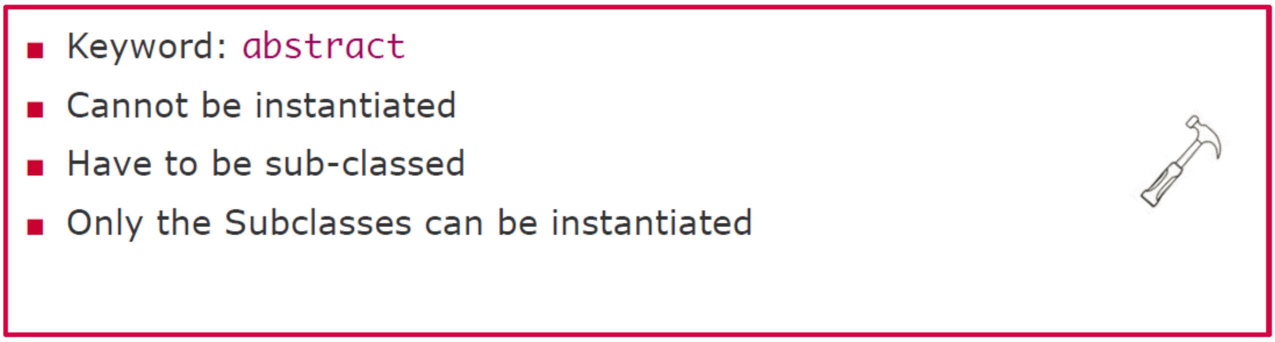
`super();`

- Call constructor of **the Superclass**
- Can only be called within a constructor
- Has to be first statement in the constructor

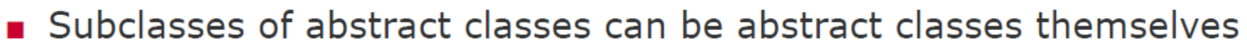


➔ you can either use this() **or** super() in the same constructor. Never both.

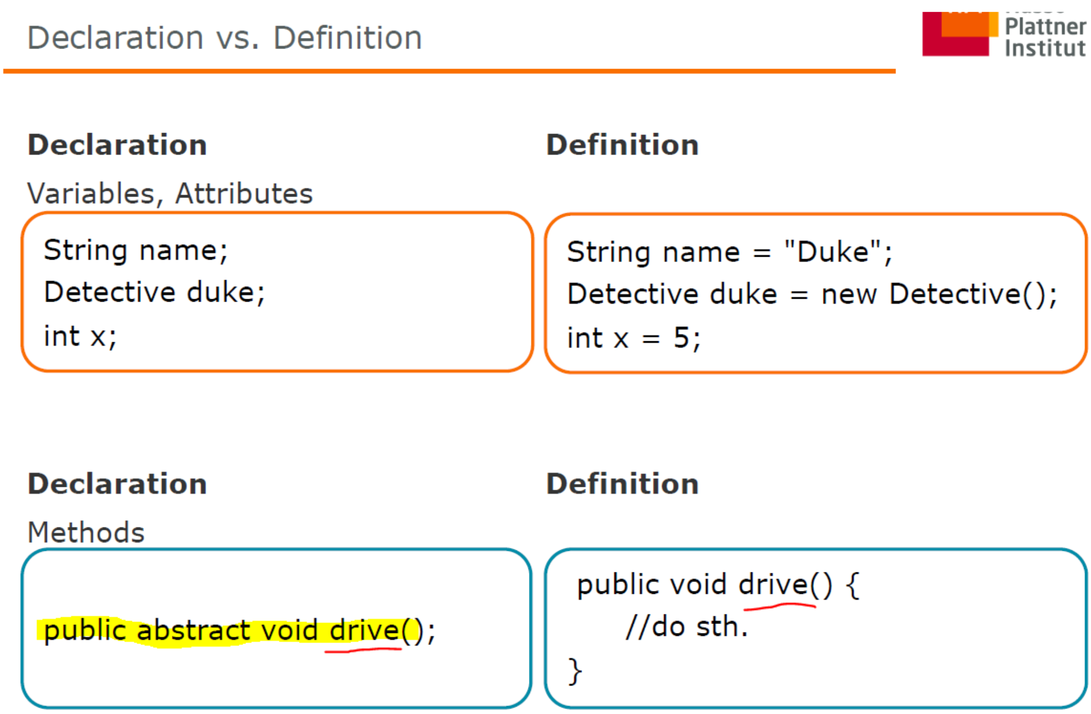
Abstract Classes



- Keyword: abstract
- Cannot be instantiated
- Have to be sub-classed
- Only the Subclasses can be instantiated



- Subclasses of abstract classes can be abstract classes themselves



Declaration vs. Definition

Plattner Institut

Declaration

Variables, Attributes

```
String name;
Detective duke;
int x;
```

Definition

```
String name = "Duke";
Detective duke = new Detective();
int x = 5;
```

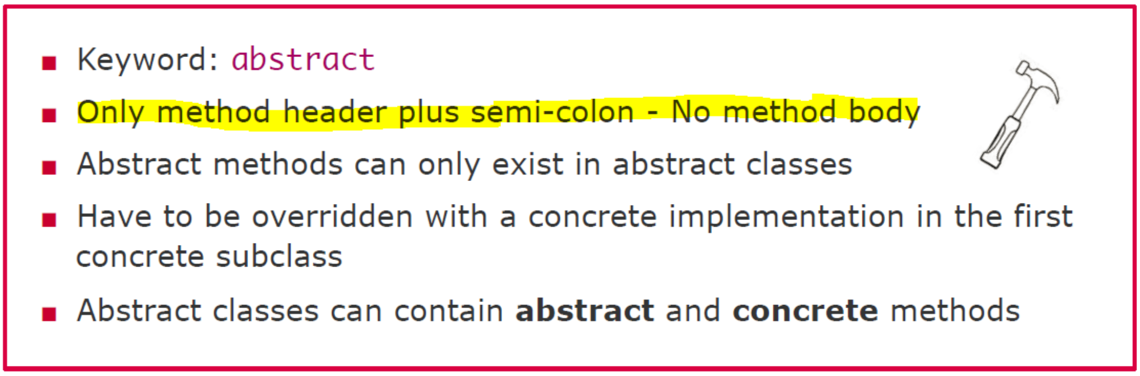
Declaration

Methods

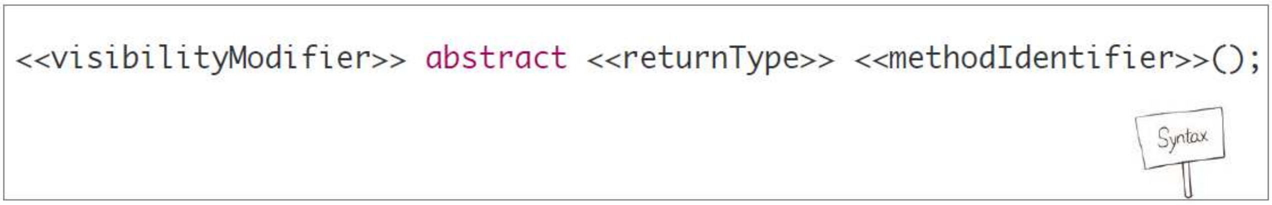
```
public abstract void drive();
```

Definition

```
public void drive() {
    //do sth.
}
```



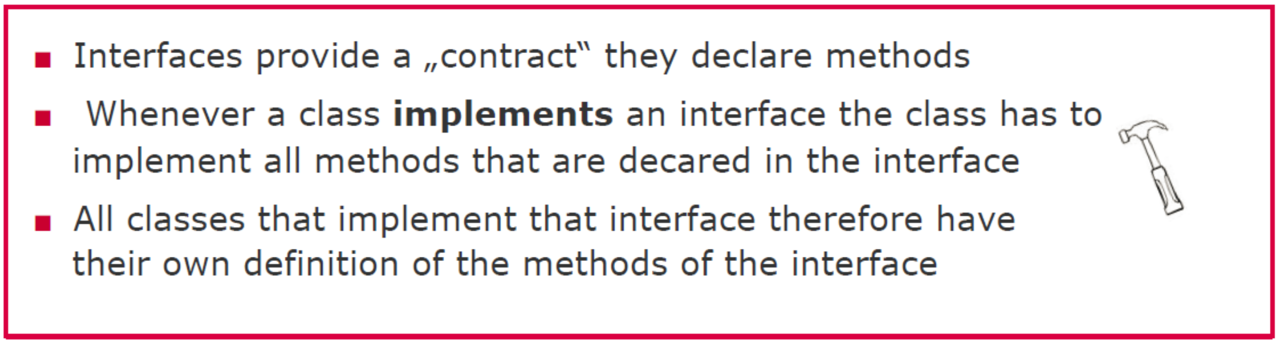
- Keyword: abstract
- Only method header plus semi-colon - No method body
- Abstract methods can only exist in abstract classes
- Have to be overridden with a concrete implementation in the first concrete subclass
- Abstract classes can contain **abstract** and **concrete** methods



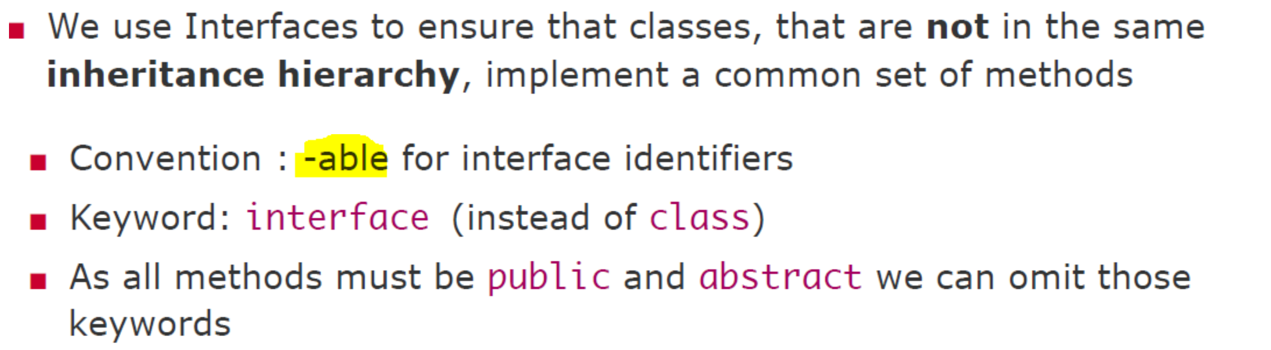
```
<<visibilityModifier>> abstract <<returnType>> <<methodIdentifier>>();
```

Interfaces

Java does not support multiple inheritance

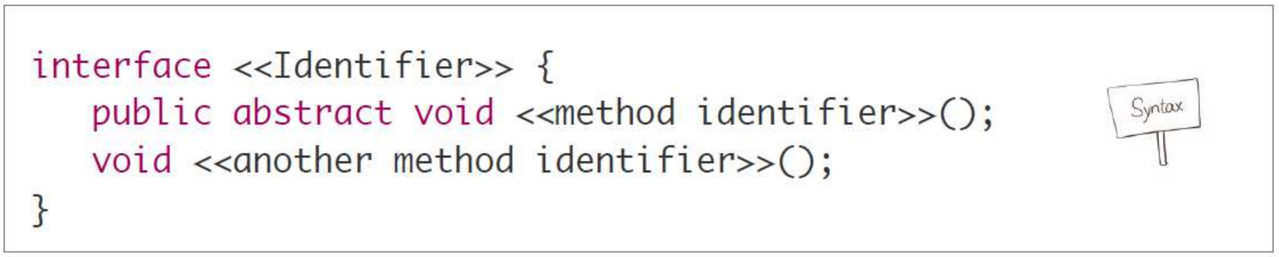


- Interfaces provide a „contract" they declare methods
- Whenever a class **implements** an interface the class has to implement all methods that are decared in the interface
- All classes that implement that interface therefore have their own definition of the methods of the interface

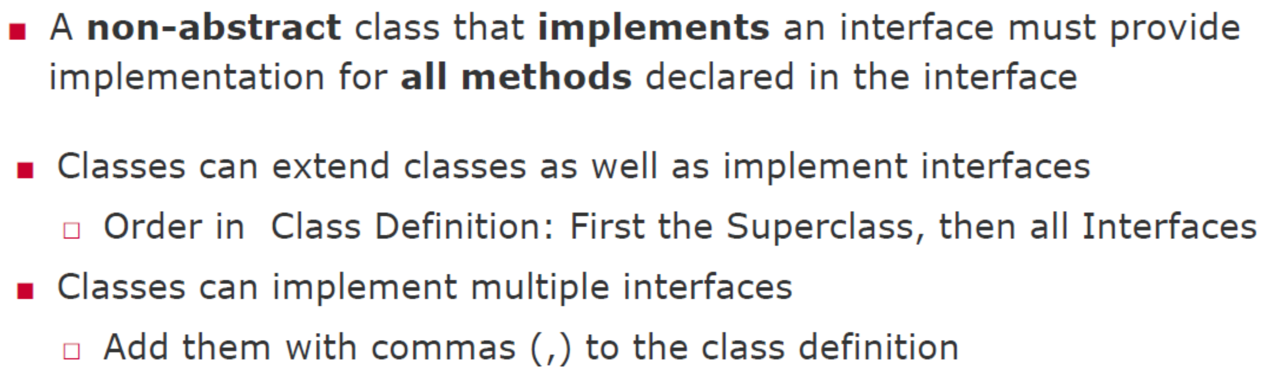


- We use Interfaces to ensure that classes, that are **not** in the same **inheritance hierarchy**, implement a common set of methods

- Convention : -able for interface identifiers
- Keyword: interface (instead of class)
- As all methods must be public and abstract we can omit those keywords



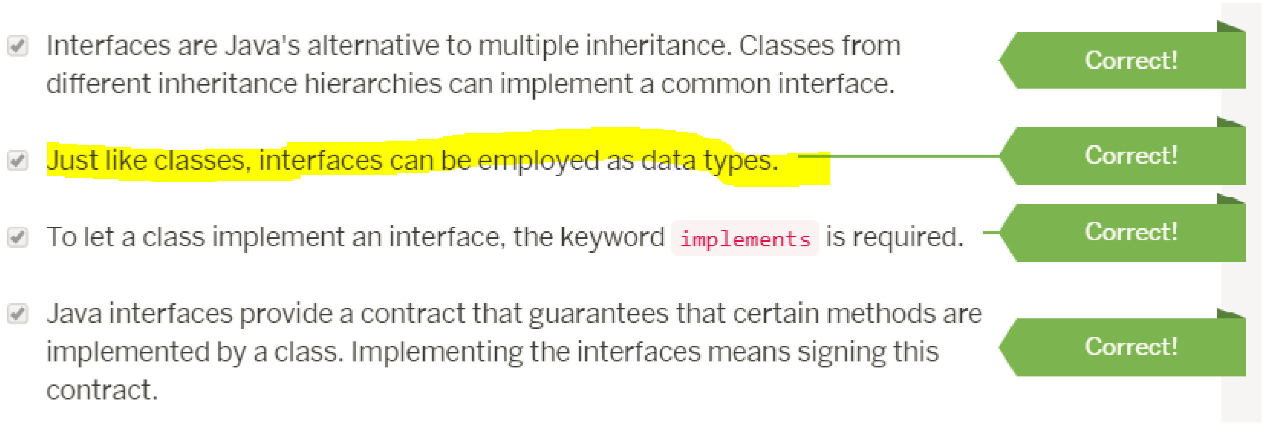
```
interface <<Identifier>> {
   public abstract void <<method identifier>>();
   void <<another method identifier>>();
}
```



- A **non-abstract** class that **implements** an interface must provide implementation for **all methods** declared in the interface

- Classes can extend classes as well as implement interfaces
  - □ Order in Class Definition: First the Superclass, then all Interfaces
- Classes can implement multiple interfaces
  - □ Add them with commas (,) to the class definition

Interfaces DON'T have STATE → **DON'T have ATTRIBUTES**

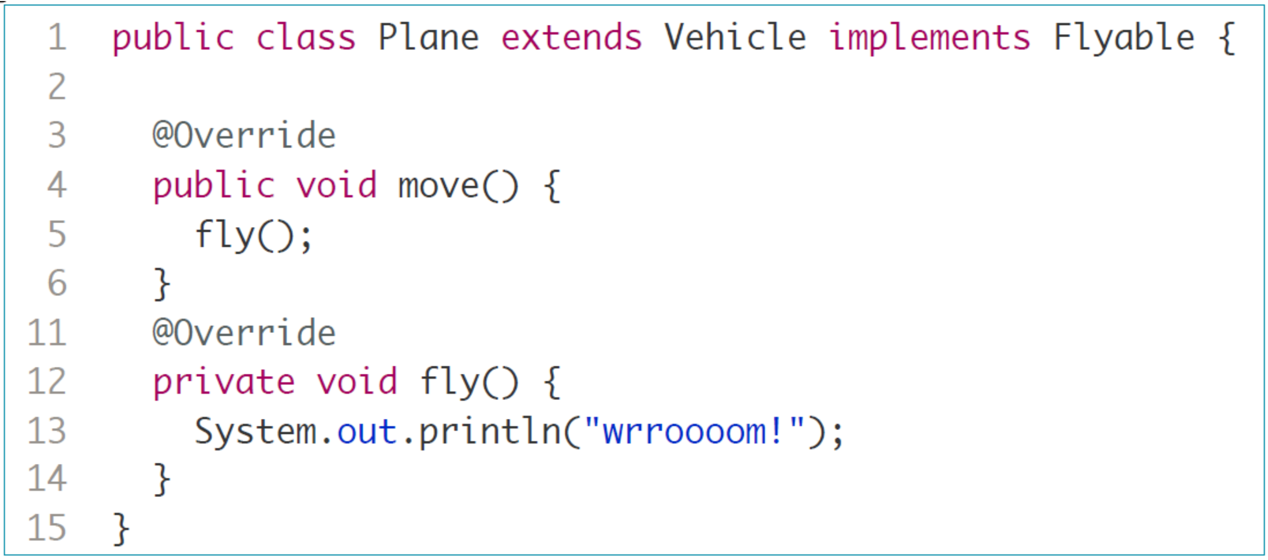


☑ Interfaces are Java's alternative to multiple inheritance. Classes from different inheritance hierarchies can implement a common interface.

☑ Just like classes, interfaces can be employed as data types.

☑ To let a class implement an interface, the keyword `implements` is required.

☑ Java interfaces provide a contract that guarantees that certain methods are implemented by a class. Implementing the interfaces means signing this contract.



```
public class Plane extends Vehicle implements Flyable {

  @Override
  public void move() {
    fly();
  }
  @Override
  private void fly() {
    System.out.println("wrroooom!");
  }
}
```

Class vs. Subclass

Class

- Define behavior and state for a set of objects
- None of the already existing classes match your use case

Subclass

- Need a more specific version of an existing class
- You want to override methods or add new behavior

Abstract Class vs. Interface

Abstract Class

- Want to define a template for a set of subclasses (fora inheritance hierarchy)
- Provide some implementation that all subclasses could use
- Want to ensure that this class is never instantiated

Interface

- Provide common functionality for a set of classes, regardless of their inheritance hierarchy.
- Realization of the functionality is provided in the implementing classes