# Recapitulare:

- modificatori de acces

- array

- singleton

* static final instance of itself as a member of the class

**private static final** OnlineShopService ***onlineShopService*** = **new** OnlineShopService();

* private constructor

**private** OnlineShopService() {  
}

* getter that retrieves the static final instance

**public static** OnlineShopService getInstance(){  
 **return *onlineShopService***;  
}

# Inheritance

Inheritanceis the capability of a class to use the properties and methods of another class while adding its own functionality.

A class uses the keyword **extends** to inherit from another class (process also referred as subclassing).

Exercise with animals:

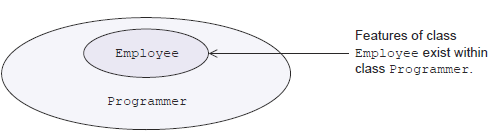
* Create animal – bird – mammal
* Add behavior (eat – fly)

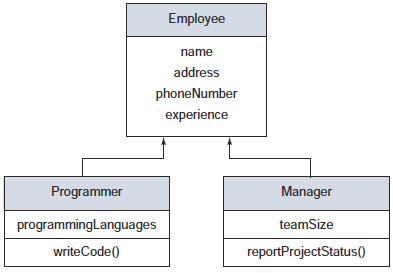
Access modifiers: **private**

Exercise:

* **Add private access modifier to the eat method in Animal. Try to call eat() method in Bird or Mammal instances**
  + *Private stuff are not visible in subclass*

In the following example, the classes Programmer and Manager inherit the non private variables and methods (also considering access modifiers) of Employee and can use them directly, as if they were defined in their own classes.





Constructors:

Exercise:

* **Create constructor in subclass that receives parameters to initialize attributes**
* **Create constructor in parent class to receive parameters to initialize attributes**
  + *The call to super() with no attributes is implicit, for constructors with attributes has to be explicit*
  + *The first step in a constructor is to call super()*

When you refer to an object**, the type or reference variable** can be different from **the object being referred** to. Objects of derived classes can be referred to using a reference variable of any of the types:

* its own type
* Its superclass
* Implemented interfaces

You will have access only to members of the class that’s used as the reference variable type.

**class** A {  
 **int i** = 10;  
  
 **public void** a(){  
 System.***out***.println(**"a"**);  
 }  
}  
  
**class** B **extends** A {  
 **int i** = 20;  
  
 **public void** b(){  
 System.***out***.println(**"b"**);  
 }  
}

A a = **new** B();  
System.***out***.println(a.**i**);  
  
*//a.a();  
//a.b();*

# Polymorphism

= many forms

The ability to request that the same operations be performed by a wide range of different types of things. This means that you can declare several methods with the same name until they are different in certain characteristics

## Method overloading and method overriding

* Static: resolved at compile time – overloading
* Dynamic: resolved at runtime – overriding

## Overriding

Properties:

1) the name of **the overridden method in the parent class** and the name of the **overriding method in the child class** must be the same

Exercise:

* **Change access modifiers for overridden method in subclass(Programmer). Public – private.**
* **Add new class Person in different package and make Employee extend it. Add method in Person speak() with default access modifier. Try to override it from Employee. Change access modifier to Protected. Try to call method in Employee (or any of the subclasses)**
  + *2) access modifiers for an overriding method can be the same or less restrictive than the ones for the method in the parent class. Protected methods can be overridden by methods in other packages through inheritance*
* **Make overridden method final in parent class**
  + *3) a child class can override only non final methods*

4) the argument list must be the same (or of compatible types - subclasses)

5) the return type of the overriding method can be the same or a subclass of the return type of the method in the parent class

6) overriding methods should not throw new or broader checked exceptions

## Overloading

Properties:

Exercise:

* **Write communicate method returns String; has String param message**
* **Write communicate method returns String: has 2 String params message1, message2**

1. *Method must have the same name*
2. *Must have method parameters different from one another*

* **Write communicate method with different return type and no params**
* **Write communicate method with different return type but same no of params- does it compile?**

1. *Method may or may not define a different return type*

* **Write method with different access modifiers**

1. *Method may or may not define different access levels*

# Object class

Java.lang.class

* Root of all class hierarchies
* Every class has Object as a superclass
* All objects, including arrays, implement the methods of this class
* Object methods
  + Equals()
  + hashCode()
  + contract between equals() and hashCode()

# Aggregation

We call aggregation those relationships whose objects have an independent lifecycle, but there is ownership and child objects cannot belong to another parent object

ex: cell phone and a cell phone battery. A single battery can belong to a phone, but if the phone stops working, and we delete it from our database, the phone battery will not be deleted because it may still be functional. So in aggregation, while there is ownership, objects have their own lifecycle.

# Composition

We use the term composition to refer to relationships whose objects don’t have an independent lifecycle, and if the parent object is deleted, all child objects will also be deleted, or putting from a different perspective, the lifetime of the objects are the same.

ex: Human class is a composition of several body parts including Hand, Leg and Heart. When human object dies, all it's body part ceased to exist meaningfully.