Langages de programmation
Notes de cours

Maria-Iuliana Dascalu, PhD

mariaiulianadascalu.com

mariaiuliana.dascalu@gmail.com

Université Polytechnique de Bucarest

Faculté d'ingénierie en Langues étrangères

Rappel

- fonctions récursives
- tableaux dynamiques et hétérogenes (ArrayList)

```
ArrayList a = new ArrayList();
a.add("Ana");//string
a.add(2);//number
a.add('c');//character
for (int i=0;i<a.size();i++)
{
    System.out.println(a.get(i));
}
for (Object i:a)
{
    System.out.println(i);
}
System.out.println (Arrays.toString(a.toArray()));
```

Aujourd'hui

.Structures de données

C'est l'organisation efficace d'un ensemble de données, sous la forme de tableaux, de listes, de piles etc. Cette efficacité réside dans la quantité mémoire utilisée pour stocker les données, et le temps nécessaire pour réaliser des opérations sur ces données**.**

.Algorithmes

Sont utilisés pour traiter les éléments d'un ensemble de données. Ils définissent une procédure informatique, par exemple: tris, recherche etc.

.Collections

Une collection permet à un groupe d'objets à traiter comme une seule unité. Les objets peuvent être stockés, récupérés et manipulés comme des éléments d'une collection. Le “Java Collections Framework” (en **java.util paquet**) fournit un ensemble de classes d'utilitaires standard pour la gestion des différents types de collections**.**

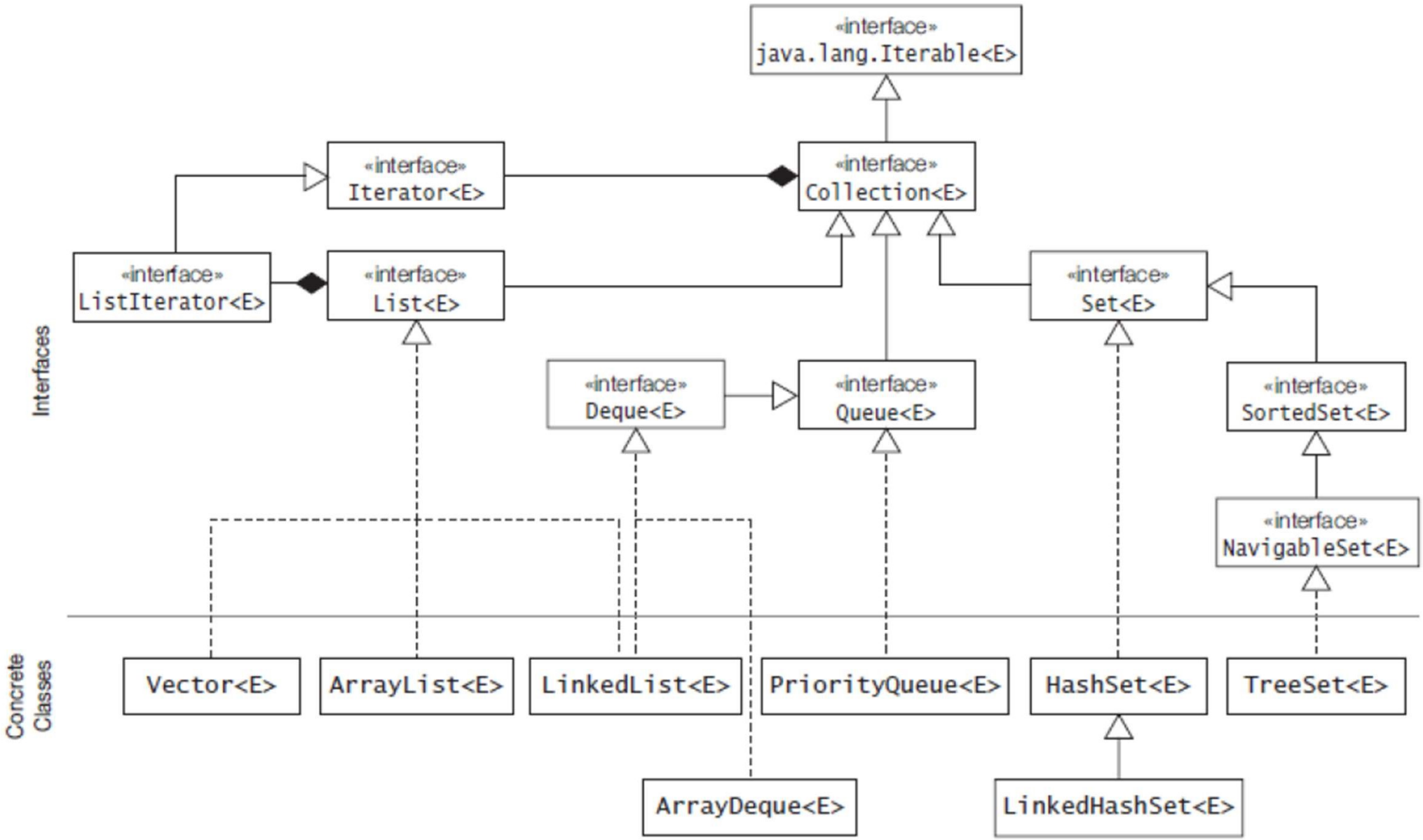
.Introduction à POO

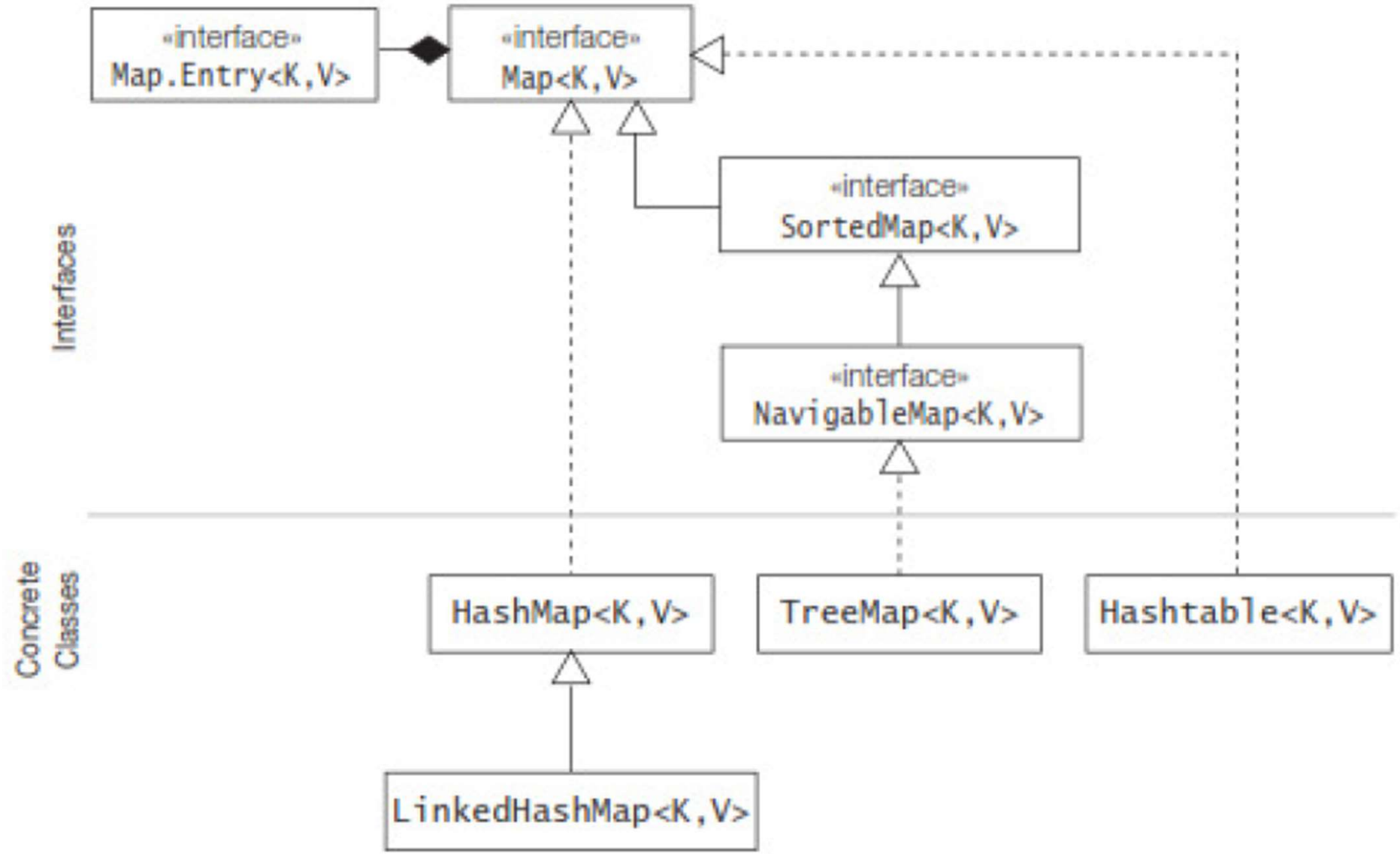
Java Collections Framework

- **Collection**: un groupe d'objets où la duplication peut-être autorisée.
- **Set**: est ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées.

Par exemple l'ensemble A = {1,2,4,8}. Set hérite donc de **Collection**, mais n'autorise pas la duplication. SortedSet est un Set trié.

- **List**: hérite aussi de collection, mais autorise la duplication. Dans cette interface, un système d'indexation a été introduit pour permettre l'accès (rapide) aux éléments de la liste.
- **Map**: est un groupe de paires contenant une clé et une valeur associée à cette clé. Cette interface n'hérite ni de Set ni de Collection. La raison est que Collection traite des données simples alors que Map des données composées (clé,valeur). SortedMap est un Map trié.


«interface»
java.lang.Iterable<E>
«interface»
Iterator<E>
«interface»
Collection<E>
«interface»
ListIterator<E>
«interface»
List<E>
«interface»
Set<E>
«interface»
Deque<E>
«interface»
Queue<E>
«interface»
SortedSet<E>
«interface»
NavigableSet<E>
Interfaces
Concrete Classes
Vector<E>
ArrayList<E>
LinkedList<E>
PriorityQueue<E>
HashSet<E>
TreeSet<E>
ArrayDeque<E>
LinkedHashSet<E>


«interface»
Map.Entry<K,V>
«interface»
Map<K,V>
«interface»
SortedMap<K,V>
«interface»
NavigableMap<K,V>
Interfaces
Concrete Classes
HashMap<K,V>
TreeMap<K,V>
Hashtable<K,V>
LinkedHashMap<K,V>

Exemples

```
import java.util.*;

public class JavaApplication1 {

    public static void main(String[] args) {
        HashSet <Integer> v = new HashSet();
        v.add(4);
        v.add(5);
        v.add(4);
        System.out.println(Arrays.toString(v.toArray()));

        Stack <String> s = new Stack();
        s.push("Ana");
        s.push("are");
        s.push("mari");
        System.out.println(s.peek());
        s.pop();
        System.out.println(s.peek());

    }

}
```

Auto-Boxing et Auto-UnBoxing

L'auto-boxing n'est pas une notion spécifique des collections, mais il nous facilite la vie.
Dans la version 1.4 de java, nous étions obligés d'ajouter un *new Integer(int)* :

```
Hashtable monHashtable = new Hashtable() ;
monHashtable.put(new Integer(1),"Janvier");
monHashtable.put(new Integer(2),"Fevrier");
```

Cette syntaxe devient vite très lourde lors de manipulations fréquentes de types primitifs (int, float, boolean, double, ...).
L'auto-boxing nous affranchit de cette syntaxe rébarbative et nous permet de mettre directement le type primitif :

```
Hashtable monHashtable = new Hashtable() ;
monHashtable.put(1,"Janvier");
monHashtable.put(2,"Fevrier");
```

Bien évidemment, l'opposé est possible. Il est donc possible de récupérer un type primitif à partir de l'objet associé comme ici :

```
Integer i = 3; // Integer i = new Integer(3); -> Auto-Boxing
int j=i; // int j=i.intValue(); -> Auto-UnBoxing
```

Voici un tableau présentant l'association entre types primitifs et objets associés :

Type primitif	Objet associé
int	Integer
char	Character
float	Float
double	Double
long	Long
boolean	Boolean
byte	Byte
short	Short

Programmation orientée objet (POO)

- Paradigme de programmation
- Programme vu comme un ensemble d'objets(composants) qui communiquent (via leur interface respective)

Qu’est ce qu’est un Objet?

- Entité/concept caractérisée par
  - des **données** (champs/attributs) qui décrivent l'état de l'objet
  - des **méthodes** permettant d’agir sur ces données
  - une **identité**
- Exemple: une figure géométrique
  - données: coordonnées, dimension, couleur
  - méthodes: translation, rotation, explosion...
  - identité: figure1, figure2,...

Notions connexes

- Objet: instance d'une classe
- Classe: modèle à partir de laquelle les objets sont faits
- Instanciation: processus de création d'objets à partir d'une classe
- Valeur de référence: retourné quand un objet soit créé
- Variable de référence (référence d'objet): une variable qui peut stocker une valeur de référence

Les étapes de la création de objets

Exemple: *Dog myDog= new Dog ();*

- Déclarez une variable de référence d'une classe:

  Dog myDog= new Dog ();

- Créez un objet:

  Dog myDog= new Dog ();

- Affectez l'objet à la référence:

- *Dog myDog= new Dog ();*

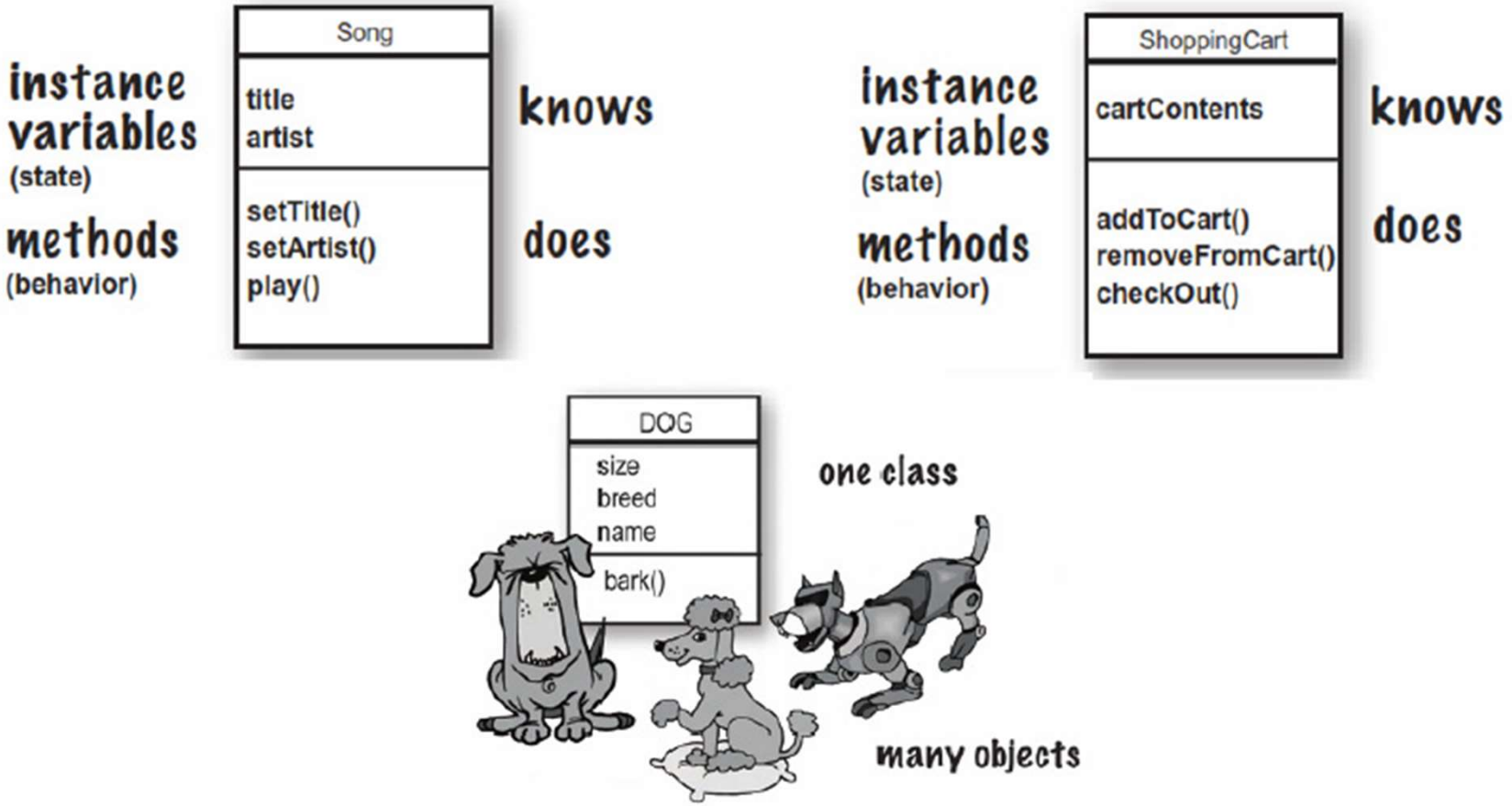
Caractéristiques des objets

- Comportement/behaviour (les choses quel objet fait/does): quelles sont les méthodes que vous pouvez appliquer/ ce que vous pouvez faire avec l'objet?

- État/state (les choses quel objet sais/knows): comment l'objet réagit lorsque vous appliquez ces méthodes?

- Identité: comment est l'objet distingue des autres ayant le même comportement et de l'état?

Exemples


instance variables
(state)
methods
(behavior)
Song
title
artist
setTitle()
setArtist()
play()
knows
does
instance variables
(state)
methods
(behavior)
ShoppingCart
cartContents
addToCart()
removeFromCart()
checkOut()
knows
does
DOG
size
breed
name
bark()
one class
many objects

La classe Dog

```
7   class Dog
8   {
9       // instance variables
10      private int size;
11      private String breed;
12      private String name;
13
14      // methods
15      public void bark()
16      {
17          System.out.println("The dog "+this.name+" barked");
18      }
19  }
20
21  public class TestDog
22  {
23      public static void main(String[] args)
24      {
25          // instantiation
26          Dog myDog = new Dog();
27          myDog.bark();
28      }
29  }
```

Le Constructeur

- Il fonctionne avant que l'objet peut être affecté à une référence
- A le même nom que la classe
- Il ressemble à une méthode, mais n'a pas de type de retour
- Il est utilisé pour initialiser l'état d'un objet

```
public class Dog()
{
    public Dog ()
        {
            size = 13;
            Breed = "cocker";
            name = "Micky";
        }
}
```

Constructeurs surchargés (overloaded)

- Si vous avez plus d'un constructeurs dans un classe, ils doivent avoir différentes listes d'arguments!
- Faites attention: le type de variables et l' ordre de variable sont importantes!
- Par défaut, le constructeur sans arguments est mis en œuvre (il existe, sans l'écrire), mais...
- Si vous écrivez un constructeur qui prend arguments, et vous voulez encore un no-arg constructeur, vous aurez à construire le no-arg constructeur par vous-même!

Constructeurs surchargés

```java
// constructors
public Dog()
{
    this.size= 13;
    this.breed = "cocker";
    this.name = "Micky";
}

public Dog(int size, String breed, String name)
{
    this.size= size;
    this.breed = breed;
    this.name = name;
}
```

```java
public class TestDog
{
    public static void main(String[] args)
    {
        // instantiation
        Dog myDog = new Dog();
        myDog.bark();
        Dog yellowDog = new Dog(15, "Golden Retriever", "Goldy");
        yellowDog.bark();
    }
}
```

Accessibilité

```
class Dog
{
    // instance variables
    private int size;
    String breed;
    private String name;
    // methods
    public void bark() {System.out.println("The dog "+this.name+" barked");}
    // constructors
    public Dog(int size, String breed, String name)
    {
        this.size= size;
        this.breed = breed;
        this.name = name;
    }
}
public class TestDog
{
    public static void main(String[] args)
    {
        Dog yellowDog = new Dog(15, "Golden Retriever", "Goldy");
        System.out.println("The dog "+yellowDog.name+" is a "+yellowDog.breed);
    }
}
```

public

RuntimeException: Uncompilable source code - name has private access in testdog.Dog

Has access, as "breed"is public

Les membres statiques vs. les membres d'instance

- Les membres statiques appartiennent à une classe, pas à des objets individuels.
- Les membres statiques peuvent être accessibles à la fois par le nom de la classe et via des références d'objet.
- Les membres d'instance ne sont accessibles que par des références d'objets.

Exemple

```
class Dog
{
    String name;
    int id;
    static int nextId = 1;
    public Dog(String name)
    {
        this.name = name;
        this.id = nextId;
        nextId++;
    }
}
public class TestDog
{
    public static void main(String[] args)
    {
        Dog yellowDog = new Dog("Goldy"); Dog blackDog = new Dog("Blacky");
        System.out.println("The dog "+yellowDog.id+" has the name "+yellowDog.name);
        System.out.println("The dog "+blackDog.id+" has the name "+blackDog.name);
        int dogsNo = Dog.nextId-1;
        System.out.println("The total numbers of dogs is: "+dogsNo);
    }
}
```

yellowDog.nextId or blackDog.nextId returns the same result!

Identité des objects

- Deux objets sont identiques si elles ont le même endroit dans la mémoire (égalité de référence") ": un changement à l'un affecte l'autre.
- It is implemented with “==“ operator.

```
public class JavaApplication1 {

    public static void main(String[] args) {
        String a = new String ("Hello");
        String b = new String ("Hello");
        System.out.println(a==b);
        System.out.println(a.equals(b));
    }

}
```

```
vaapplication1.JavaApplication1 > main >
t - JavaApplication1 (run) ×
run:
false
true
BUILD SUCCESSFUL (total time: 0 seconds)
```

Mémoire en Java

- 2 zones de la mémoire
  - Stack (où les invocations de méthode et variable locale vivent)
  - Heap (où les objets vivent)
- Variable d'instance sont déclarées dans une classe, pas à l'intérieur d'une méthode; ils vivent à l'intérieur de l'objet auquel ils appartiennent.
- L'environnement d'exécution Java supprime les objets quand il détermine qu'ils ne sont plus utilisés (garbage collection).