GRAU D'ENGINYERIA INFORMÀTICA

PROGRAMACIÓ II

Bloc 2:

Programació Orientada a Objectes (4)

Sergio Sayago (basat en material de Laura Igual)
Departament de Matemàtiques i Informàtica
Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

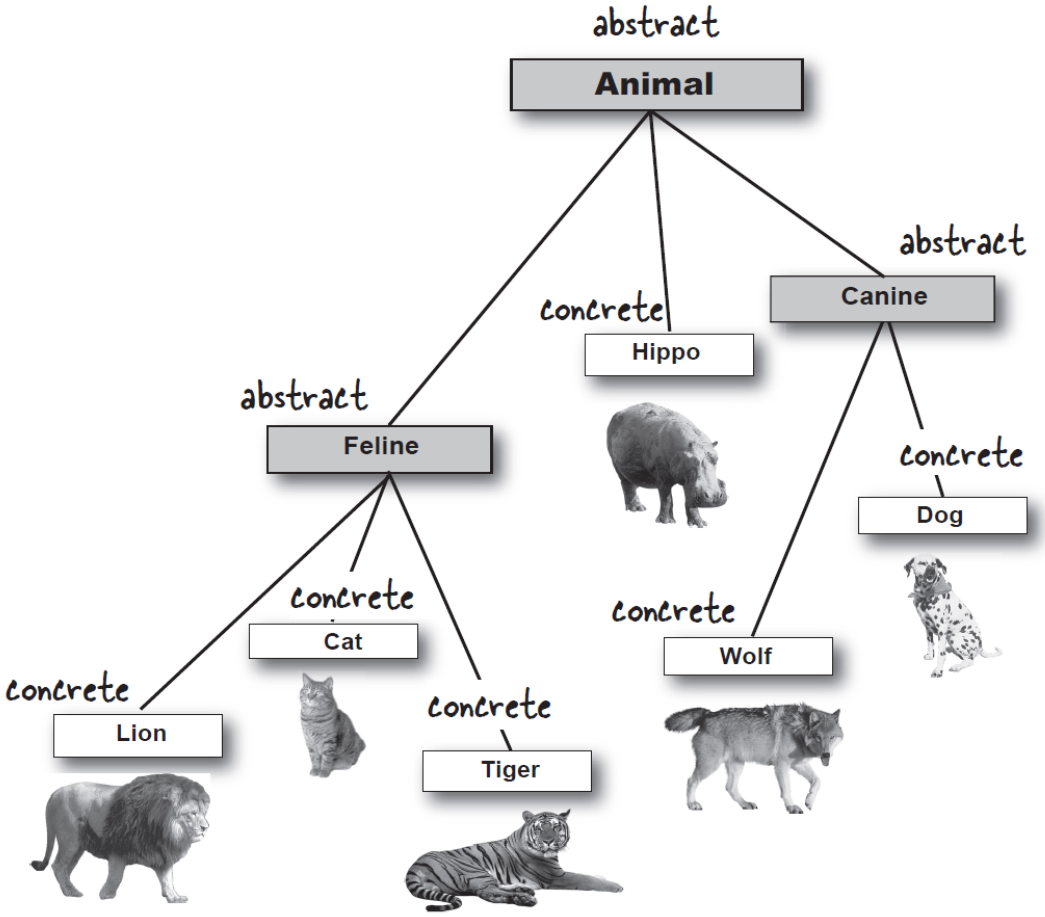
Bloc 2:
Programació orientada a objectes

- Abstracció en el desenvolupament del *software*
- Conceptes fonamentals: classes i objectes
- Característiques de l'orientació a objectes
- Ús de classes i objectes
- Constructors i destructors
- Encapsulació
- Herència i jerarquia de classes
- Polimorfisme
- Lligadures
- **Interfícies**
- col·leccions

Introducció

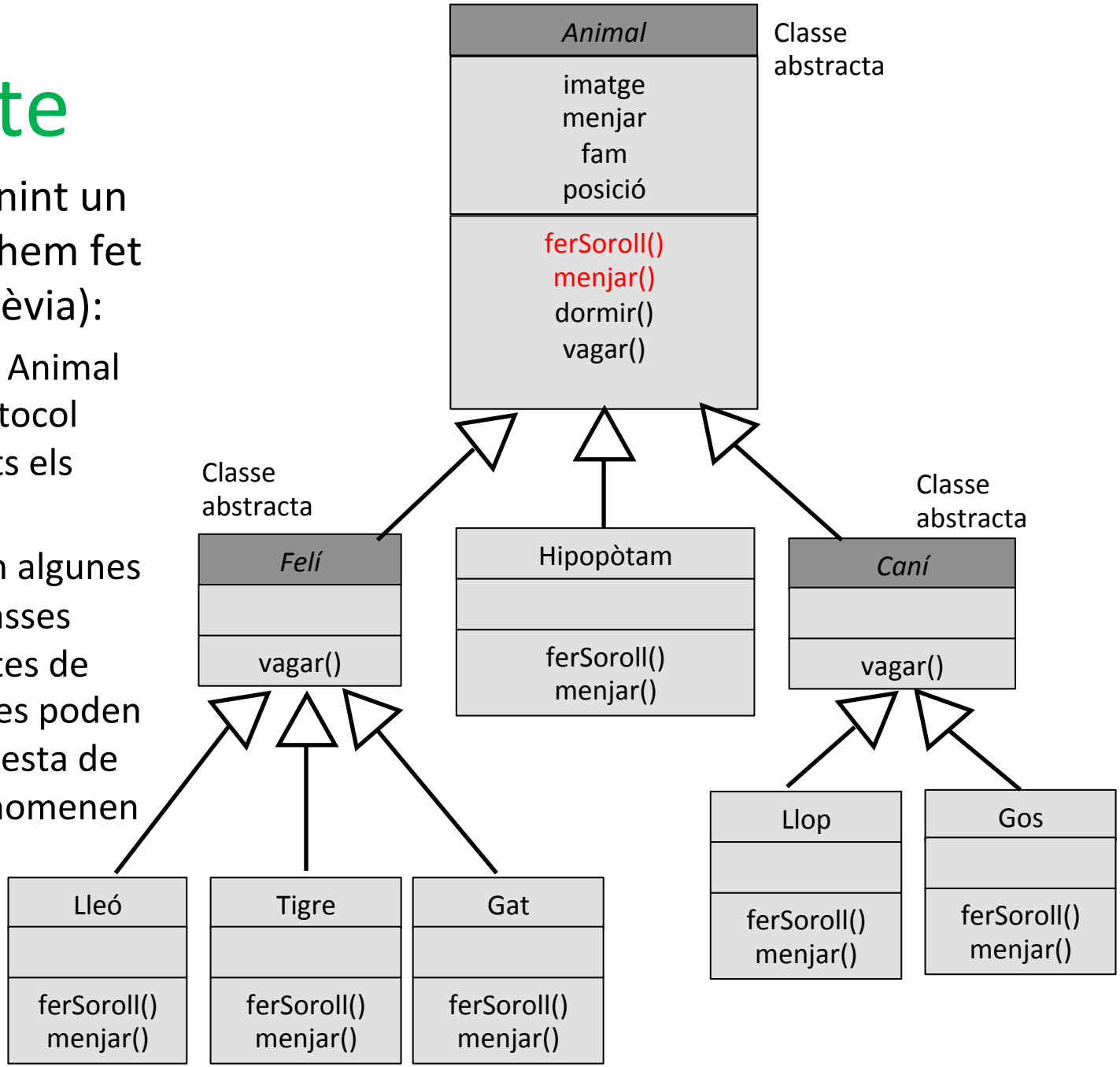
- Introducció d'interfícies amb un exemple:
  - La jerarquia d'herències de la classe Animal.

Contracte


abstract
Animal
abstract
Canine
concrete
Hippo
abstract
Feline
concrete
Dog
concrete
Cat
concrete
Wolf
concrete
Lion
concrete
Tiger

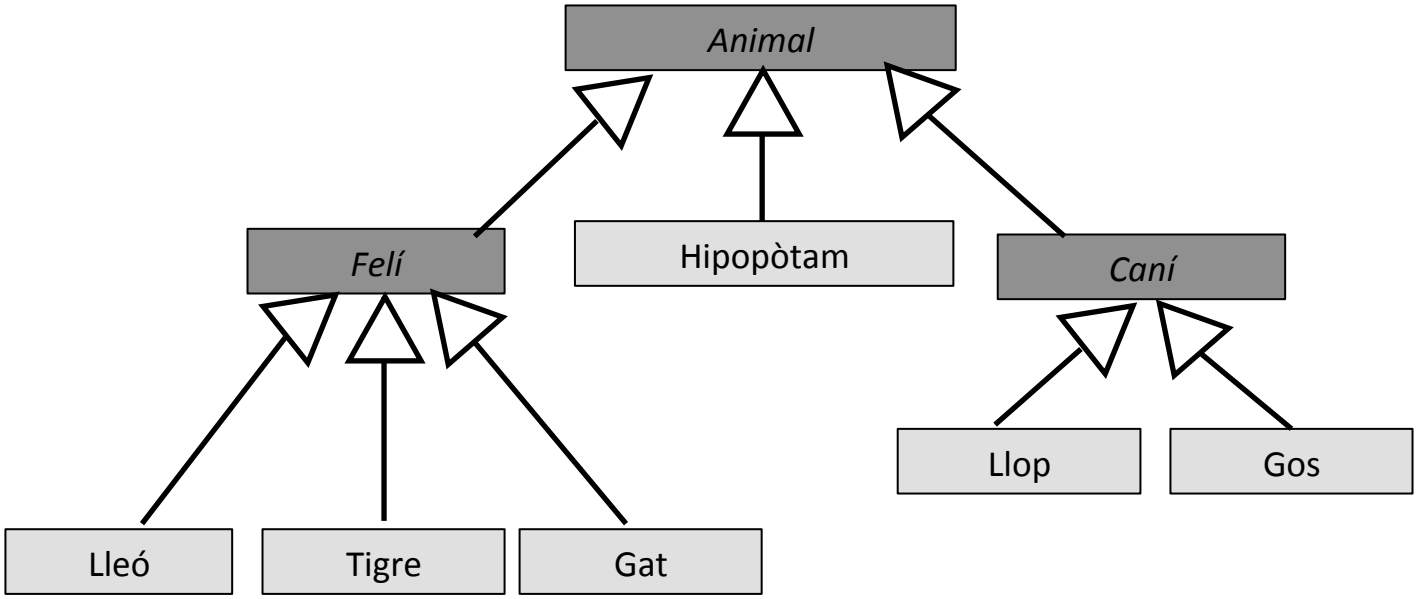
Contracte

- Comencem definint un contracte (com hem fet en una classe prèvia):
  - La superclasse Animal defineix el protocol comú per a tots els animals.
  - A més, definim algunes de les superclasses com a abstractes de forma que no es poden instanciar. La resta de les classes s'anomenen concretes.



Volem afegir els comportaments de les mascotes. Possibles dissenys?

- Veiem diferents opcions de disseny per reutilitzar algunes de les classes existents en un programa d'una tenda de **mascotes**.



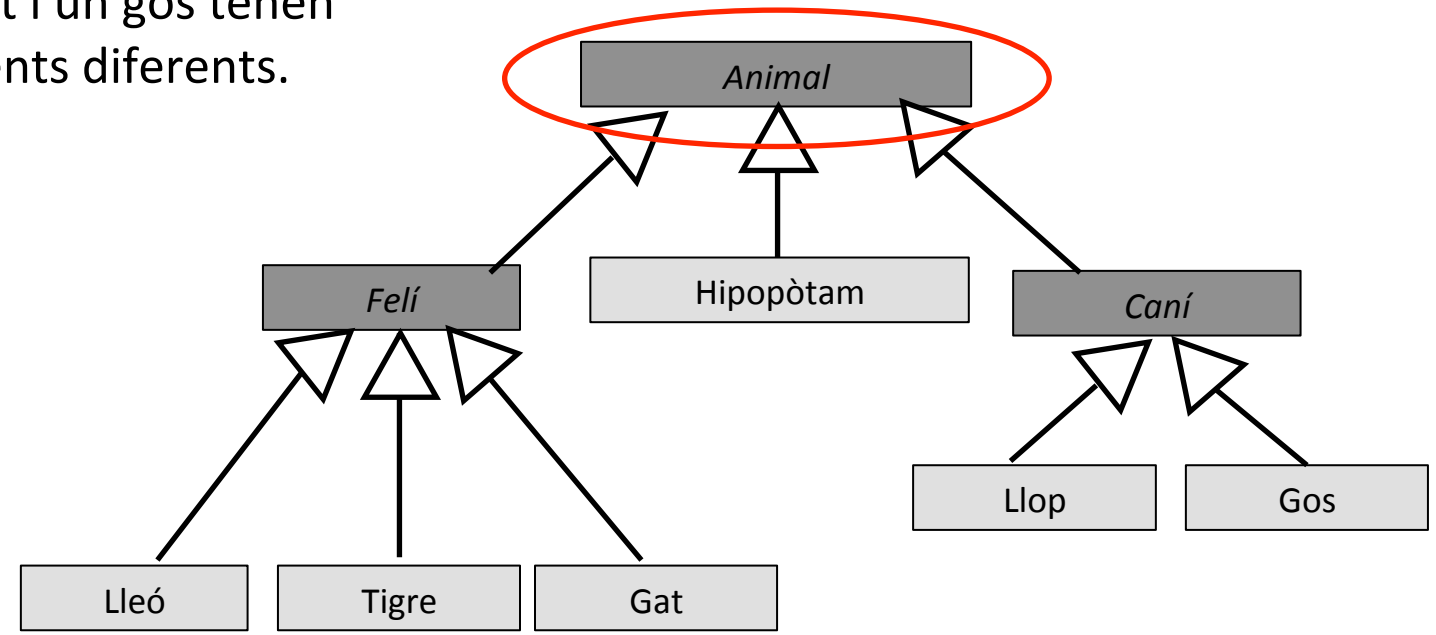
Opció 1

- Posem els mètodes de mascota en la classe Animal.

  Pros: No modifiquem les classes existents i les noves classes que afegim heretaran aquests mètodes.

  Contres: Un Hipopòtam no és una mascota!

  A més, un gat i un gos tenen comportaments diferents.

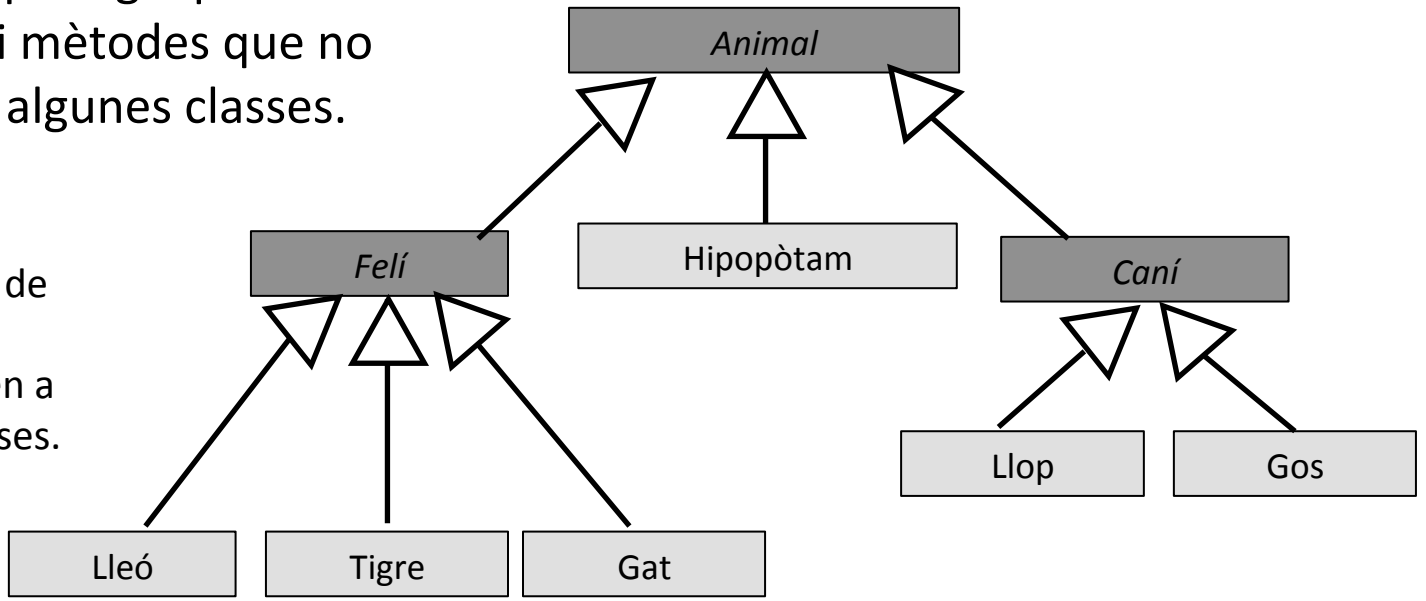


Opció 2

- Posem els mètodes de mascota en la classe Animal, però fem els **mètodes abstractes** forçant les subclasses d'Animal a sobreescrire'ls.

Pros: Els mateixos que l'opció 1, però a més podem definir no-mascotes.
Com? Fent que les implementacions no facin res.
Contres: S'han d'implementar tots els mètodes abstractes de la classe Animal encara que sigui per no fer res → molt treball i mètodes que no tenen sentit en algunes classes.

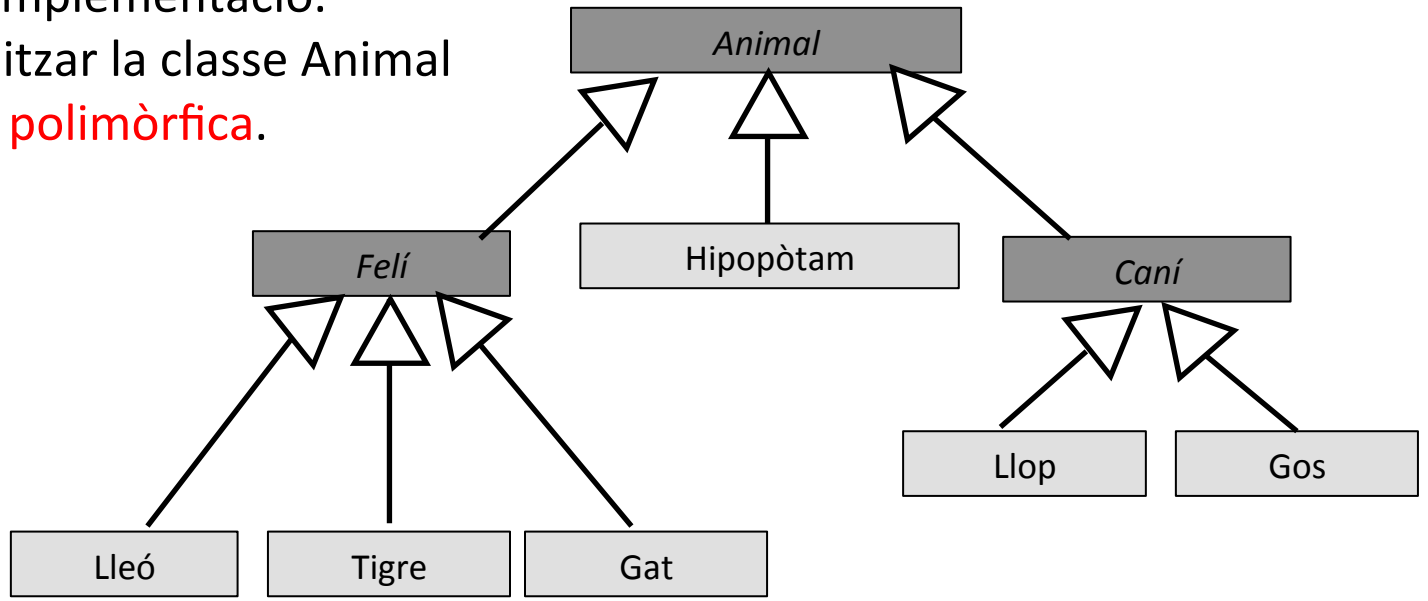
En aquest cas, només hauríem de posar dins de la classe Animal, els mètodes que s'apliquen a totes les seves subclasses.



Opció 3

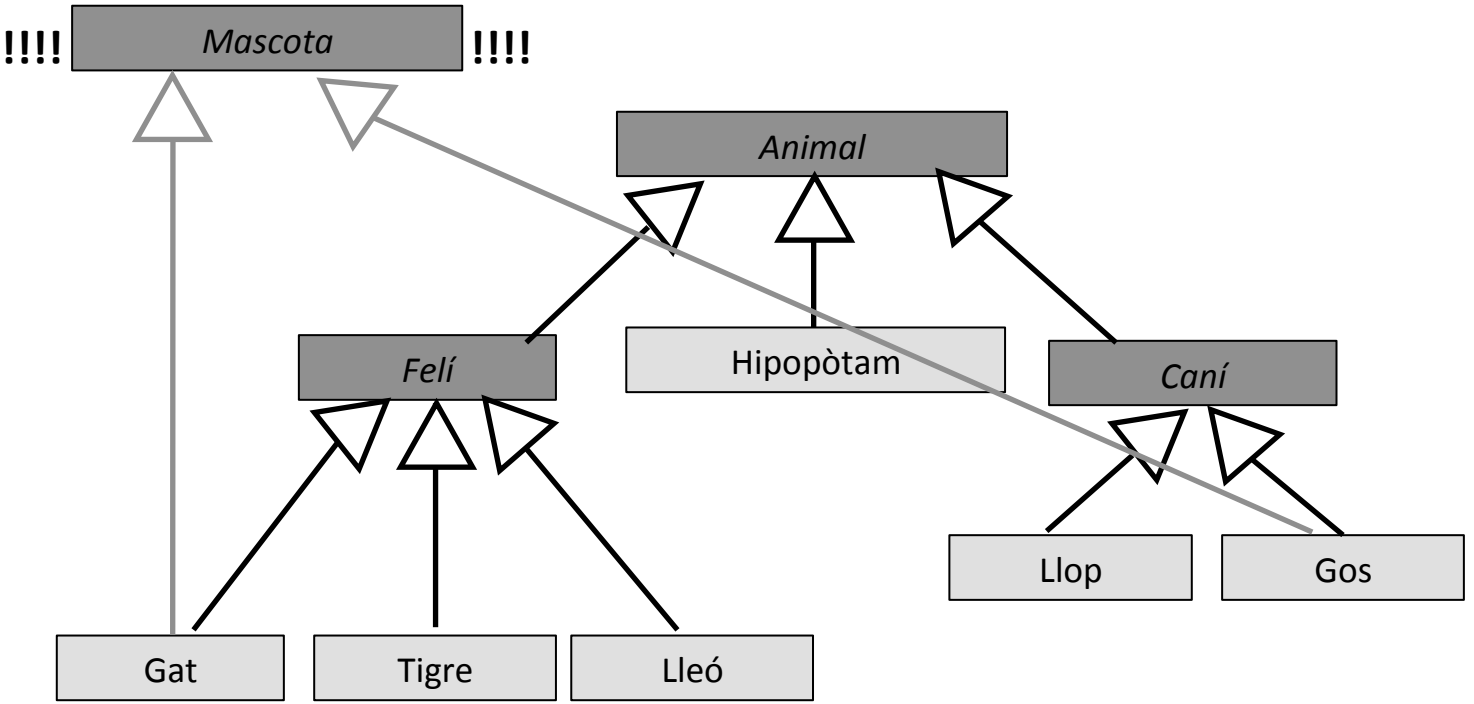
- Posem els mètodes de mascota només en les classes que ho són.

Pros: Desapareixen els hipopòtams com a mascotes i els mètodes estan on toca.
Contres: Tots els programadors hauran de conèixer el protocol. No hi ha contracte que obliga el compilador a verificar la implementació.
No es pot utilitzar la classe Animal com la classe polimòrfica.



Necessitem dues superclasses

→ Herència múltiple



→En lloc de classes abstràctes, utilitzarem **interfícies**.

Interfícies

- Una interfície és un conjunt de **declaracions de mètodes** (sense definició)
- Una interfície també pot definir **constants** que són implícitament *public*, *static* i *final*, i sempre s'han d'inicialitzar en la declaració
- Totes les classes que implementen una interfície estan obligades a proporcionar una definició als mètodes de la interfície
- Una interfície defineix el protocol d'implementació d'una classe

Interfícies

- Una classe pot implementar més d'una interfície

→representa una alternativa a l'herència múltiple en Java.

- La paraula clau és:

implements + el nom de la interfície

interface *nom_interficie* {
  tipus_retorn nom_metode (*llista_arg*) ;
  . . .
}

class *nom_classe* **implements** *nom_interficie* {
  tipus_retorn nom_metode (*llista_arg*) {
    <codi>
  }
}

Interfícies

- A C++, les interfícies s'implementen mitjançant classes abstractes
- Les interfícies defineixen tipus
  - Son una classe
- Les interfícies no es poden instanciar (és a dir, no podem fer **new**)
  - De què es faria un new?

Implementació

```
public interface Mascota {
   public void serAmigable();
   public void jugar();
}
```

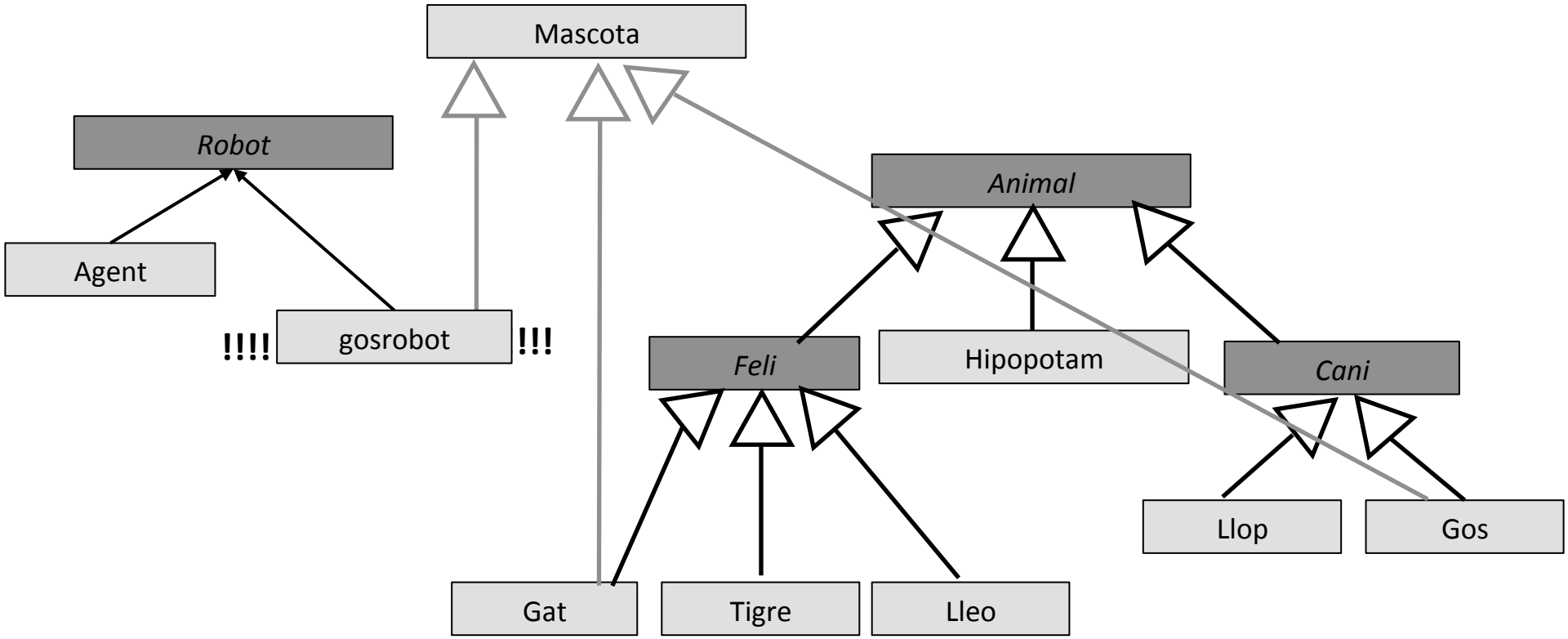
Mascota.java

Exemple

```
public class Gos extends Cani implements Mascota{
   public void ferSoroll(){
      System.out.println("guau");
   }
   public void menjar(){
     System.out.println(”menjo molt");
   }
   public void serAmigable() {
      System.out.println("fa gràcies");
   }
   public void jugar() {
     System.out.println("juga");
   }
}
```

Gos.java

Classes de diferents arbres d'herència poden implementar la mateixa interfície



Diferents classes poden implementar la mateixa interfície

```
public interface VideoClip {
  // comença la reproducció del video
  void play();
  // reprodueix el clip en un bucle
  void bucle();
  // para la reproducció
  void stop();
}

//I una classe que implementa la interfície:
class LaClasse implements VideoClip {
 void play() { <codi> }
 void bucle(){ <codi> }
 void stop() { <codi> }
}
```

...implementant interfícies

```
public interface VideoClip {
  // comença la reproducció del video
  void play();
  // reprodueix el clip en un bucle
  void bucle();
  // para la reproducció
  void stop();
}
```

//I una altra classe que també implementa la interfície:

```
Class LaAltraClasse implements VideoClip {
      void play()  { <codi nou> }
      void bucle() { <codi nou > }
      void stop()  { <codi nou > }
}
```

Interfície

Quan utilitzar una interfície en lloc d'una classe abstracta?

- Per la seva senzillesa es recomana utilitzar interfícies sempre que sigui possible.
- Si la classe ha d'incorporar atributs, o resulta interessant la implementació d'alguna de les seves operacions, llavors declarar-la com a classe abstracta.
- Dins la biblioteca de classes de **Java** es fa un ús intensiu de les interfícies per a caracteritzar les classes.

- Alguns exemples:
  - Per a que un objecte pugui ser guardat en un fitxer, la seva classe ha d'implementar la interfície *Serializable,*
  - Per a que un objecte sigui duplicable, la seva classe ha d'implementar la interfície *Cloneable,*
  - Per a que un objecte sigui ordenable, la seva classe ha d'implementar la interfície *Comparable.*

Extensió d'interfícies

- Les interfícies poden extendre altres interfícies
- La sintaxis es:

interface *nom_NovaInterficie* **extends** *nom_interficie* , . . . {
    tipus_retorn nom_metode (*llista_arguments*) ;
    . . .
}

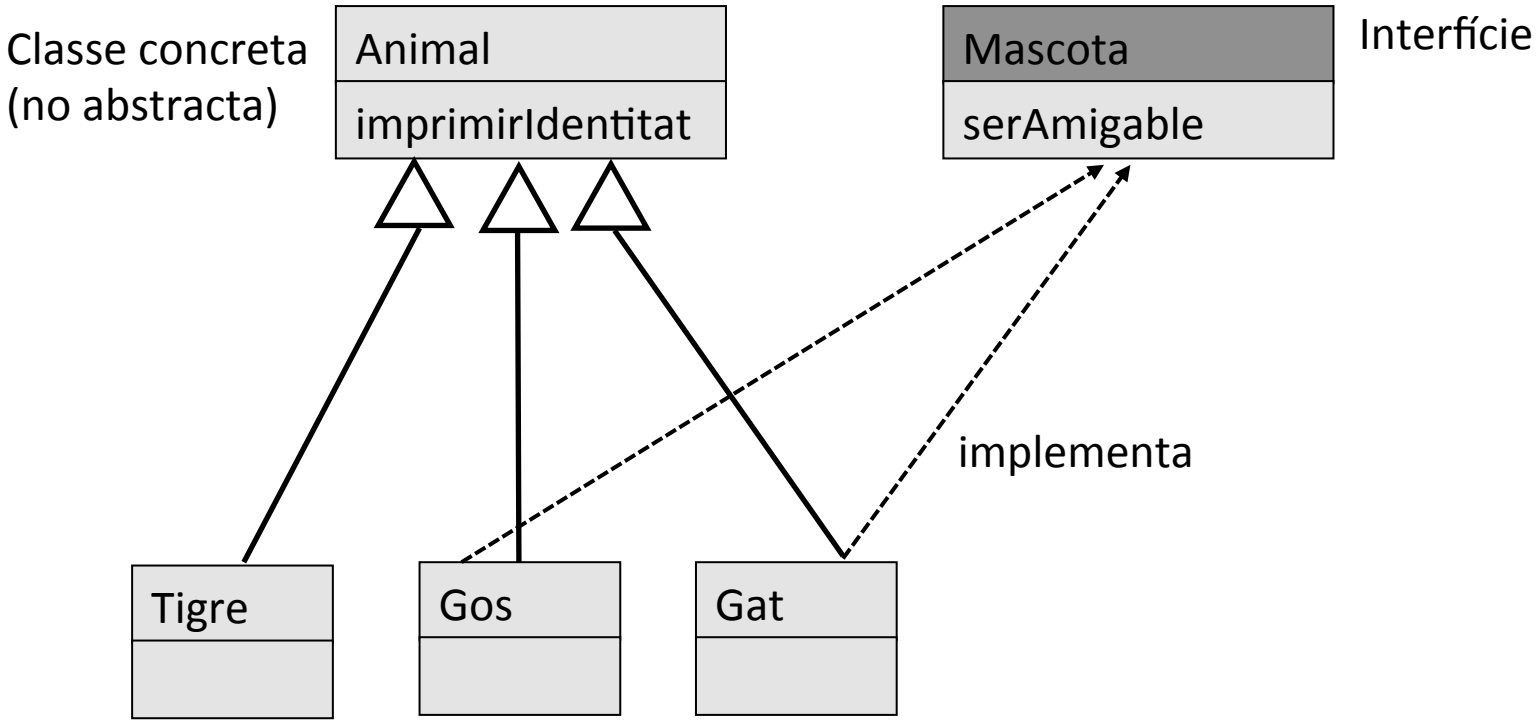
Interfícies per herència múltiple

- Si necessitem fer herència múltiple, a Java ho fem mitjançant interfícies

- Una mateixa classe pot implementar diferents interfícies

Exercici

Implementa les següents classes i interfícies.



Solució

```
public class Animal {
  public void imprimirIdentitat(){
      System.out.println("Soc un animal");
  }
}
```

```
public interface Mascota {
    public void serAmigable();
}
```

Solució

```
public class Tigre extends Animal{
}
```

```
public class Gos extends Animal implements Mascota{
    public void serAmigable() {
        System.out.println("fa gràcies, guau");
    }
    public void imprimirIdentitat(){
      System.out.println("Soc un gos");
     }
}
```

```
public class Gat  extends Animal implements Mascota{
    public void serAmigable() {
          System.out.println(“es passeja, miau");
    }
     public void imprimirIdentitat(){
      System.out.println("Soc un gat");
     }
}
```

Exercici

Quina serà la sortida per pantalla?

```
public class TestAnimals {
        public static void main(String[] args){
          Animal[] animals = new Animal[10];
           …
          animals[0] = new Gos();
          animals[1] = new Tigre();
          animals[2] = new Tigre();
          animals[3] = new Gat();
           …
         for (int i=0; i<4; i++){
              if(animals[i] instanceof Mascota){
                   Mascota masc = (Mascota) animals[i];
                   masc.serAmigable();
               }
              animals[i].imprimirIdentitat(); }
           }
        }
}
```

→ fa gràcies, guau
Soc un gos
Soc un animal
Soc un animal
Es passeja, miau
Soc un gat

Algunes preguntes més…

- El següent codi, és correcte?

```
public interface SomethingIsWrong {
   void aMethod(int aValue) {
      System.out.println(”Hello World");
   }
}
```

Algunes preguntes més…

- El següent codi, és correcte?

```
public interface SomethingIsWrong {
   void aMethod(int aValue) {
      System.out.println(”Hello World");
   }
}
```

- No és correcte. Té un métode amb cos

Algunes preguntes més…

- El següent codi, és correcte?

```
public interface Serializable {
}
```

Algunes preguntes més…

- El següent codi, és correcte?

```
public interface Serializable {
}
```

- Sí és correcte. Les interfícies es poden utilitzar per definir tipus i no tenir cap mètode. Un exemple és la interfície Serializable a Java – que utilitzareu a la P2

Referències

- Bertrand Meyer, "**Construcción de software orientado a objetos", Prentice Hall, 1998.**

- "Software Architecture and UML" de Grady Booch (Rational Software). Presentació P. Letelier.

- Bert Bates, Kathy Sierra. **Head First Java**. O'Reilly Media, 2005.

Bloc 2:
Programació orientada a objectes

- Abstracció en el desenvolupament del *software*
- Conceptes fonamentals: classes i objectes
- Característiques de l'orientació a objectes
- Ús de classes i objectes
- Constructors i destructors
- Encapsulació
- Herència i jerarquia de classes
- Polimorfisme
- Lligadures
- Interfícies
- **Col·leccions**

Framework Col·leccions

- Una **col·lecció** és un objecte que agrupa múltiples elements en una única unitat.
- Normalment representen elements d'informació dins d'un grup natural, com
  - una bústia de correu (una col·lecció de correus),
  - un directori (una col·lecció de fitxers),
  - una guia telefònica (una associació entre noms i números de telèfon).
- La llibreria standard de Java ens ofereix classes i interfícies que ens permeten manegar col·leccions d'objectes
- **Piles**, **Cues**, **Llistes**, **Conjunts** són casos particulars de col·leccions d'objectes

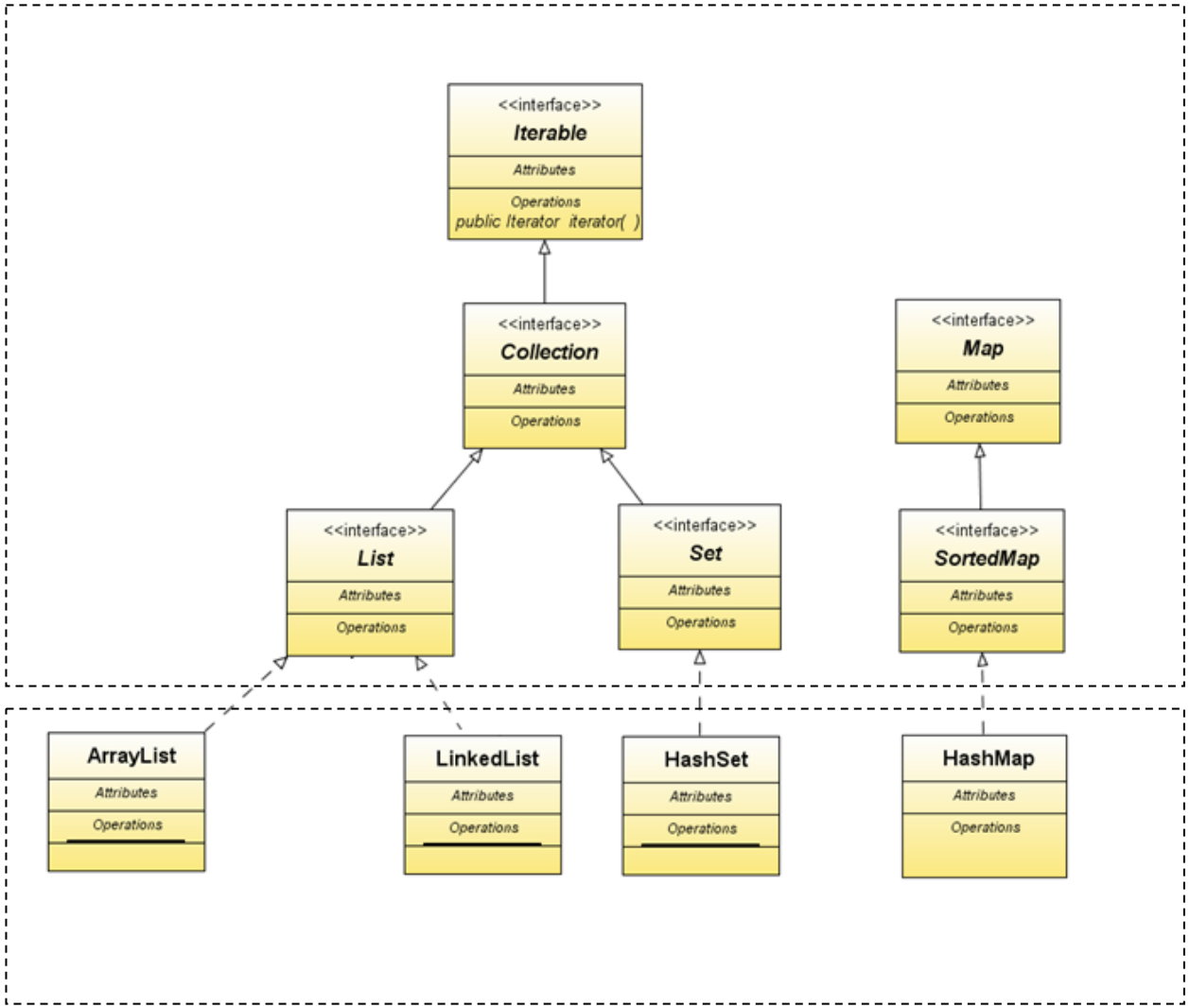
Col·leccions

- Encara que ArrayList és la que més utilitzem a les pràctiques, hi ha altres col·leccions útils:
  - LinkedList – llista enllaçada
  - HashMap – mapa hash

Diagrama de classes simplificat

Interfaces

Implementations



ArrayList vs. LinkedList

- LinkedList: una altra implementació d'una llista.
- Una qüestió d'implementació:

- Quan necessiteu accedir de forma seqüencial i teniu un nombre poc variable d'elements → ArrayList.
- Quan necessiteu esborrar o inserir al davant o al mig moltes vegades el contingut de la llista → LinkedList.

Creació d'una ***LinkedList***
LinkedList list= new LinkedList ();

Col·leccions i iteradors

- Un iterador és un objecte que proveeix una forma de processar una col·lecció d'objectes, un a un, seguint una seqüència.
- Un iterador ens permet recorre els elements d'una col·lecció d'objectes

- Un iterador es crea formalment implementant la interfície `Iterator<E>`, que conté 3 mètodes:
  - **`hasNext`** → retorna un resultat booleà que és cert si a la col·lecció queden objectes per processar
  - **`next`** → retorna el següent objecte a processar
  - **`remove`** → elimina l'últim objecte (el més recent) retornat pel mètode `next`

Col·leccions i iteradors

```
public interface Iterator<E>
{
     E next();
     Boolean hasNext();
     void remove();  //opcional
}
```

- Alguna cosa és iterable si es pot iterar sobre ell. Per poder iterar usem un iterador. Una classe és iterable si és capaç de retornar-nos un iterador

```
public interface Iterable<E> {
   public Iterator<E> iterator();
 }
```

- Implementant la interfície Iterator una classe formalment estableix que els objectes d'aquesta classe són iteradors
- El programador ha de decidir com implementar les funcions d'iteració
- Un iterador, per tant, caracteritza una seqüència

Col.leccions: Exemples d'Ús

- **Creació d'una col·lecció d'objectes**

```
Collection c = new ArrayList();
c.add("Hello");
c.add("World");
```

- **Recorregut** d'una col·lecció amb un **iterador**

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
        String s = (String)i.next();
        System.out.println(s);
}
```

- **Recorregut** d'una col·lecció amb un ***for .. each***

```
for (Object item : c) {
        System.out.println(item.toString());
}
```

Exemples d'Ús

- **Exemple** 1: Definició de mètodes que treballen contra la interface Collection.

  Col·leccions de tipus heterogeni

  CreaColeccio.java

Col·leccions i iteradors: Exemple 1

```
import java.util.*;

public class CreaColeccio {
  public static void main(String[] args) {
    Collection myCollection1 = new ArrayList();
    Collection myCollection2 = new HashSet();

    fillCollection(myCollection1);
    fillCollection(myCollection2);
    showCollection(myCollection1);
    showCollection(myCollection2);
    treuMaria(myCollection1);
    treuMaria(myCollection1);
    diguesSiEstaMaria(myCollection1);
    diguesSiEstaMaria(myCollection2);
  }
```

Col·leccions i iteradors: Exemple 1

```
public static void fillCollection(Collection c) {
   c.add(34);
   c.add("Pepe");
   c.add(new Gato("Sasha"));
 }

public static void showCollection(Collection c) {
   if (c.isEmpty()) { System.out.println("La col·lecció esta buida");
   } else {
       System.out.println("La col·lecció conté " + c.size() + " elements:");
       System.out.println(c);
   }
 }
```

Col·leccions i iteradors: Exemple 1

```
 public static void treuMaria(Collection c) {
    c.remove(“Maria");
  }

public static void diguesSiEstaMaria (Collection c) {
    if (c.contains(“Maria")) {
      System.out.println(“Maria està dins de la col·lecció");
    } else {
      System.out.println(“Maria no està a la col·lecció");
    }
  }
}
```

Referències

- Bertrand Meyer, "**Construcción de software orientado a objetos", Prentice Hall, 1998.**

- "**Thinking in Java**" Bruce Eckel.

- Bert Bates, Kathy Sierra. **Head First Java**. O'Reilly Media, 2005.