

Programació

UT9.2 Classes genèriques

Introducció

- Ja coneixes una classe genèrica com és `ArrayList` i la interfície genèrica `Comparable`.
- Les classes genèriques **permeten parametritzar tipus de dades**.
- Això ens obri la possibilitat de definir una interfície, una classe o un mètode amb tipus genèrics que el compilador pot substituir per tipus concrets.

Exemple

- Per exemple, `ArrayList` en la seua API es definia com una classe genèrica capaç de treballar amb tipus genèrics, on posteriorment al utilitzar el constructor indicavem el tipus concret que voliem emmagatzemar.
- En este capítol anem a aprendre a definir les nostres pròpies classes, interfícies i mètodes genèrics i demostrar com estos poden millorar l'eficiència i la llegibilitat del nostre codi.

Motivacions y beneficis

Recordem la interfície Comparable vista en unitats anteriors:

```
package java.lang;  
  
public interface Comparable<T> {  
    public int compareTo(T o)  
}
```

Les classes que implementen esta interfície estan obligades a implementar el mètode compareTo que determina la comparació entre dos elements.

Tipus genèric

- `<T>` representa el tipus de dada genèric, que pot ser substituït per un tipus de dada concret.
- A este reemplaçament l'anomenem "**instanciació genèrica**"

*Per convenció, els noms dels paràmetres de tipus són lletres simples i majúscules.
Els noms de paràmetres de tipus més utilitzats són:*

E - Element (utilitzat àmpliament pel Java Collections Framework)

K - Clau

N - Número

T - Tipus

V - Valor

S,U,V, etc. - 2n, 3r, 4t tipus

Filtrat d'errors en temps de compilació

Gràcies a les classes genèriques podem filtrar errors quan s'intente fer ús d'un element que no siga del tipus definit en temps d'execució.

```
Comparable<Date> c = new Date();  
System.out.println(c.compareTo("red"));
```

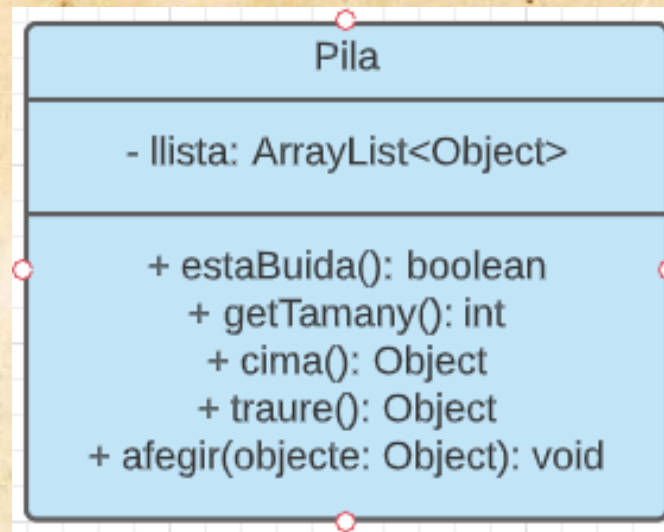
Error

```
ArrayList<String> list = new ArrayList<>();  
list.add("Red");  
list.add(new Integer(1));
```

Error

Definir classes genèriques

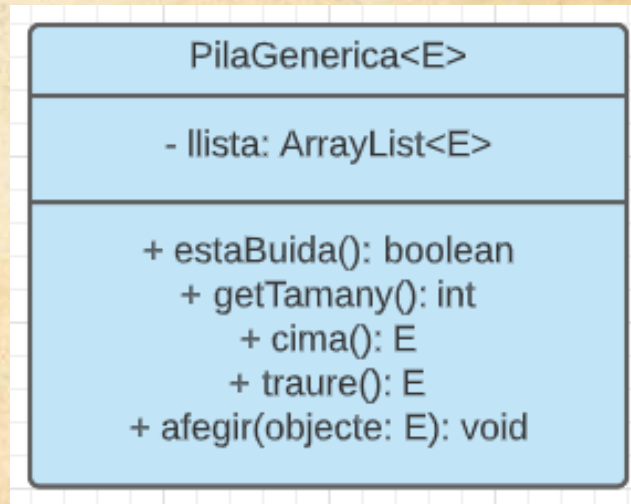
- Suposant la classe Pila següent:



- L'estructura ens permetria emmagatzemar elements i gestionar-los en forma de pila, però estariem controlant en temps de compilació que els objectes que s'introdueixen són tots del mateix tipus?

Definir classes genèriques

- La resposta és NO i per tant estaríem atemptant contra el principi de tot array (tots els elements deuen ser del mateix tipus)
- Les classes genèriques ens permeten controlar aquesta situació. La classe Pila quedaria de la següent manera:



Definir classes genèriques

```
public class PilaGenerica<E> {  
  
    private ArrayList<E> llista;  
  
    public PilaGenerica() {  
        llista = new ArrayList<>();  
    }  
  
    public int getTamany() {  
        return llista.size();  
    }  
  
    private int ultimIndex() {  
        return getTamany() - 1;  
    }  
}
```

```
    public E cima() {  
        return llista.get(ultimIndex());  
    }  
  
    public void afegir(E objecte) {  
        llista.add(objecte);  
    }  
  
    public E traure() {  
        E objecte = this.cima();  
        llista.remove(objecte);  
        return objecte;  
    }  
}
```


Mètodes genèrics

- També és possible utilitzar tipus genèrics a la definició de mètodes estàtics.
- Exemple:

tipus
genèric

```
1 public class GenericMethodDemo {
2     public static void main(String[] args) {
3         Integer[] integers = {1, 2, 3, 4, 5};
4         String[] strings = {"London", "Paris", "New York", "Austin"};
5
6         GenericMethodDemo.<Integer>print(integers);    O bé print(integers);
7         GenericMethodDemo.<String>print(strings);      bé print(strings);
8     }
9
10    public static <E> void print(E[] list) {
11        for (int i = 0; i < list.length; i++)
12            System.out.print(list[i] + " ");
13        System.out.println();
14    }
15 }
```

Implementa un mètode "imprimir" que imprimisca el primer paràmetre, siga del tipus que siga.

Tipus genèrics limitats

- Un tipus genèric també pot ser **especificat com un subtipus** d'un altre tipus.
- Es fa ús de la paraula reservada **extends**

<E> és el mateix que
<E extends Object>


El tipus genèric pot ser només derivat de GeometricObject

```
public class BoundedTypeDemo {  
    public static void main(String[] args) {  
        Rectangle rectangle = new Rectangle(2, 2);  
        Circle circle = new Circle(2);  
  
        System.out.println("Same area? " +  
            equalArea(rectangle, circle));  
    }  
  
    public static <E extends GeometricObject> boolean equalArea(  
        E object1, E object2) {  
        return object1.getArea() == object2.getArea();  
    }  
}
```

Amb això podem fer ús de mètodes com "getArea()" als quals no tindriem accés des d'un tipus genèric que no estiguera limitat.

Pregunta

- Ja sabem que al següent codi es produeix una **pèrdua d'identitat** en el moment que invoquem a “tindreNouFill”



```
public static Persona tindreNouFill(Persona conjugel, Persona conjugue2){  
    // codi  
    return null;  
}
```

```
Home home = new Home();  
Dona dona = new Dona();
```

```
Persona fill = Persona.tindreNouFill(home, dona);
```

- Tot i la pèrdua d'identitat el programa funciona per polimorfisme. Però que passarà en la situació següent?

```
public static Persona tindreNouFill(ArrayList<Persona> parella){  
    // codi  
    return null;  
}
```

```
Home home1 = new Home(), home2 = new Home();  
ArrayList<Home> parella = new ArrayList<>();  
parella.add(home1); parella.add(home2);
```

```
// Persona fill = Persona.tindreNouFill(parella);
```


Resposta

Apareix un error de compilació, ja que encara Home és una classe derivada de Persona, en este cas **ArrayList<Home>** no és una classe derivada de **ArrayList<Persona>**.

Per donar solució a este problema es va desenvolupar els tipus **wildcard** (comodí)

Tipus de wildcard

- ? (similar a ? extends Object)
- ? **extends** T (representa a T o a una classe derivada de T)
- ? **super** T (representa a T o a una classe superior a T)

```
public static Persona tindreNouFill(ArrayList<? extends Persona> parella){  
    // codi  
    return null;  
}
```

Ara el mètode acceptarà un ArrayList que continga objectes de tipus "Persona" o bé dels seus subtipus.

I en este cas?

```
class Fruita{};
class Pera extends Fruita{};
class Taronja extends Fruita{};

public class Exemple {

    public static void rebbeFruita(Fruita fruita){
        System.out.println("Rebuda: " + fruita);
    }

    public static void rebbeCistellaFruita(List<Fruita> fruites){
        System.out.println("S'ha rebut una cistella de fruites");
    }

    public static void main(String[] args) {
        Pera pera = new Pera();
        Taronja taronja = new Taronja();
        /*
        rebreFruita(pera); // Funciona?
        rebreFruita(taronja); // Funciona?

        rebreCistellaFruita(new ArrayList<Fruita>()); // Funciona?
        rebreCistellaFruita(new ArrayList<Pera>()); // Funciona?
        rebreCistellaFruita(new ArrayList<Taronja>()); // Funciona?
        */
    }
}
```


Tipus de wildcard

- Les dos últimes instruccions no funcionaran.
- ArrayList<Pera> o ArrayList<Taronja> NO son classes derivades de la classe ArrayList<Fruita>
- Solució:

```
public static void rebreCistellaFruita(List<? extends Fruita> fruites){  
    System.out.println("S'ha rebut una cistella de fruites");  
}
```


I en este cas?

```
class Fruita{};
class Pera extends Fruita{};
class Taronja extends Fruita{};

public class Exemple {

    public static <T> void copiarCistella(List<T> orige, List<T> desti){
        System.out.println("Copiant la llista...");
        for (int i = 0; i < orige.size(); i++) {
            desti.add(orige.get(i));
        }
    }

    public static void main(String[] args) {

        List<Fruita> fruites1 = new ArrayList<>();
        List<Fruita> fruites2 = new ArrayList<>();
        List<Taronja> taronjes1 = new ArrayList<>();
        List<Taronja> taronjes2 = new ArrayList<>();
        /*
        copiarCistella(fruitess1, fruitess2); // Funciona?
        copiarCistella(taronjes1, fruites1); // Funciona?
        copiarCistella(fruitess1, taronjes2); // Funciona?
        */
    }
}
```


Tipus de wildcard

- Les dos últimes instruccions no funcionaran.
- T no coincideix en l'orige i el destí (al mètode copiarCistella)
- Solució:

```
public static <T> void copiarCistella(List<T> orige, List<? super T> desti){  
    System.out.println("Copiant la llista...");  
    for (int i = 0; i < orige.size(); i++) {  
        desti.add(orige.get(i));  
    }  
}
```

- *COMPTE! Ara podrem copiar la cistella de taronjes dins d'una cistella de fruites (lògic). Però no podrem copiar una cistella de fruites dins d'una cistella de taronjes.*

Què passa realment al compilar?

- Els tipus genèrics són admesos pel compilador sense problemes.
- Al compilar, no és possible saber realment eixe tipus genèric a quin tipus concret es traduirà fins que el programa no és executat. **(TYPE ERASURE)**
- El compilador realitza una conversió interna (a Object o al límit superior).

```
ArrayList<String> list = new ArrayList<>();  
list.add("Oklahoma");  
String state = list.get(0);
```

(a)



```
ArrayList list = new ArrayList();  
list.add("Oklahoma");  
String state = (String)(list.get(0));
```

(b)

Altres exemples de conversió interna

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

(a)

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

(b)

```
public static <E extends GeometricObject>  
    boolean equalArea(  
        E object1,  
        E object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

(a)

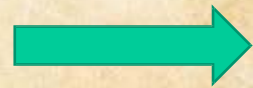
```
public static  
    boolean equalArea(  
        GeometricObject object1,  
        GeometricObject object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

(b)

Restriccions dels tipus genèrics

Suposant E com un tipus genèric,
no es pot utilitzar:

1. `new E()`
2. No es pot utilitzar
`new E[tamany]`
3. Excepcions amb tipus
genèrics



Es pot realitzar
`(E[])new Object[tamany]`