Pràctiques de Llenguatges, Tecnologies i Paradigmes de programació. Curs 2016-17

Part I
Programació en Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Pràctica II

Genericitat en Java

Índex

1 Objectiu

L'objectiu d'aquesta pràctica d'una sessió consisteix en introduir l'aplicació dels coneixements sobre *genericitat* en el llenguatge Java. Es presentarà la sintaxi necessària per a utilitzar aquest concepte en les versions actuals de Java i es proposaran exercicis per a realitzar.

2 Classes genèriques

Una classe genèrica és una classe normal, llevat de que la seva declaració es parametritza amb *variables de tipus* també anomenades *tipus genèrics* o *variables genèriques* en contrapartida als *tipus purs*.

Els noms de les variables genèriques s'escriuen darrere del nom de la classe i entre `"<"`i `">",`separats per comes. Els tipus que representen aquestes variables no són *tipus bàsics* sinó classes (*tipus referència*).

`class/interface NomClasse` <llista-de-tipus-genèrics> `{ ... }`

Per conveni, aquestes variables s'escriuen amb una lletra majúscula.

Vegem una introducció a l'ús de les classes genèriques definint i instanciant la classe `G1`, i més endavant veurem l'ús d'una classe genèrica predefinida en el llenguatge.

```
1 public class G1<T> {
2   private T a;
3   public G1(T x) { a = x; }
4   public String toString(){ return ""+a; }
5 }
```

on `T` és el nom d'una variable que no representa un valor sinó un tipus, però s'escriu dins de la classe com si es tractés d'un tipus pur, com passa a la línia 2 definint l'atribut `a` de tipus `T`. Fixa't, que en la definició del constructor no s'usa la notació `<...>` entre el nom de la classe i els paràmetres. No obstant això, la creació d'objectes segueix la següent sintaxi:

`new NomClase` <llista-de-tipus-purs> `(`llista-paràmetres-formals`)`

La substitució de les variables genèriques per un tipus pur es realitza durant l'anàlisi estàtic. Com els objectes es creen dinàmicament, només es poden crear objectes de tipus purs. És a dir, existeixen classes genèriques però no objectes genèrics. En la llista-de-tipus-purs no es poden nomenar tipus bàsics, i en el seu lloc s'usen les corresponents *classes embolcall* que s'estudiaran un poc més endavant en el punt següent.

Per exemple, podem crear un objecte de la classe `G1` per a que continga una `String` en el seu atribut `a` invocant al constructor de la classe amb `new`

`G1 <String>("hola món")`. En crear aquest objecte, la variable genèrica `T` ja ha estat canviada pel compilador al tipus de la classe `String`. A la línia 2 s'assigna aquest tipus a l'atribut `a` i en la línia 3 al paràmetre del constructor. És com si haguéssim escrit la classe `G1`:

```
private String a;
public G1(String x) {a = x;} ...
```

3 Classes embolcall

Freqüentment serà útil poder tractar les dades primitius (`int`, `double`, `boolean`, etc.) com a objectes podent així, a més, utilitzar-los genèricament. Per exemple, els contenidors definits per l'API al package `java.util` (Arrays dinàmics, llistes enllaçades, col·leccions, conjunts, etc.) estan tots definits en termes genèrics amb variables de tipus.

Aquests contenidors poden emmagatzemar, per tant, qualsevol tipus d'objectes. Però els dades primitius no són objectes, per lo que en principi queden exclosos d'aquestes possibilitats.

Per resoldre aquesta situació l'API de Java incorpora les classes embolcall (*wrapper class*), que consisteixen en dotar les dades primitius amb un embolcall que permeta tractar-los com objectes. Per exemple podríem definir una classe embolcall per als enters, de forma prou senzilla, amb:

```
public class Entero {
  private int valor;
  public Entero(int valor) { this.valor = valor; }
  public int intValue() { return this.valor; }
}
```

La API de Java fa innecessari aquesta tasca en proporcionar un conjunt complet de classes embolcall per a tots els tipus primitius. Addicionalment a la funcionalitat bàsica que es mostra en l'exemple, les classes embolcall proporcionen mètodes d'utilitat per a la manipulació de dades primitives (conversions de / cap a dades primitives, conversions a String, etc.)

Les classes embolcall existents són: `Byte` per a `byte`; `Short` per a `short`; `Integer` per a `int`: `Long` per a `long`; `Boolean` per a `boolean`; `Float` per a `float`; `Double` per a `double` i `Character` per a `char`.

Quan creem una instància d'una classe genèrica, no es poden usar tipus bàsics com `int` com a instància del tipus genèric, i aleshores cal emprar les *classes embolcall* corresponents. Per exemple, podem crear un objecte d'una classe genèrica `G1<T>` perquè continga un `integer` en el seu atribut invocant al constructor de la classe amb `new G1<Integer>(...)`. En crear aquest objecte, la *variable de tipus* `T` ja ha sigut canviada pel compilador al tipus de la classe embolcall `Integer`.

Exercici 1 *Escriu un programa de prova en el mètode* `main` *d'una classe* `UsandoClasesEnvoltorio` *en el qual es definisquen variables per a cada tipus bàsic. Assigna a cada variable un objecte de la seua corresponent classe embolcall. Escriu el contingut de les variables en l'eixida estàndard. En el mateix* `main`*, fes el mateix en sentit invers: defineix variables de tipus embolcall i assigna'ls el seu corresponent valor de tipus bàsic.*

4 Classes genèriques predefinides en Java

Existeixen moltes classes genèriques predefinides en Java. Una d'elles és la classe `ArrayList`, que implementa un array redimensionable. La Figura 1, extreta de les APIs de Java, representa la jerarquia de classes de la qual descendeix.

Aquesta classe està continguda en el paquet `java.util`. Deriva de la classe abstracta i genèrica `AbstractList<E>`, la qual és una classe derivada de la classe `AbstractCollection<E>` (també abstracta i genèrica), que al seu torn descendeix d'`Object` que es troba al paquet `java.lang`, el qual conté el nucli del llenguatge i sempre s'importa per defecte.

També implementa sis interfícies (tres d'elles genèriques) entre les quals es troba la interfície `List<E>`, la qual, mirant la seua API, també és implementada per la classe `AbstractList<E>`.

La quantitat de mètodes definits en la classe `ArrayList<E>` és menor que els que especifica la interfície `List<E>`. Açò s'explica perquè la classe pare d'`ArrayList<E>` implementa mètodes de la mateixa interfície.

Moltes d'aquestes classes estan en el paquet `java.util`, però no totes. Les tres classes predefinides en el llenguatge que estenen d'`ArrayList<E>` estan en diferents paquets, igual que tres de les interfícies que implementa.

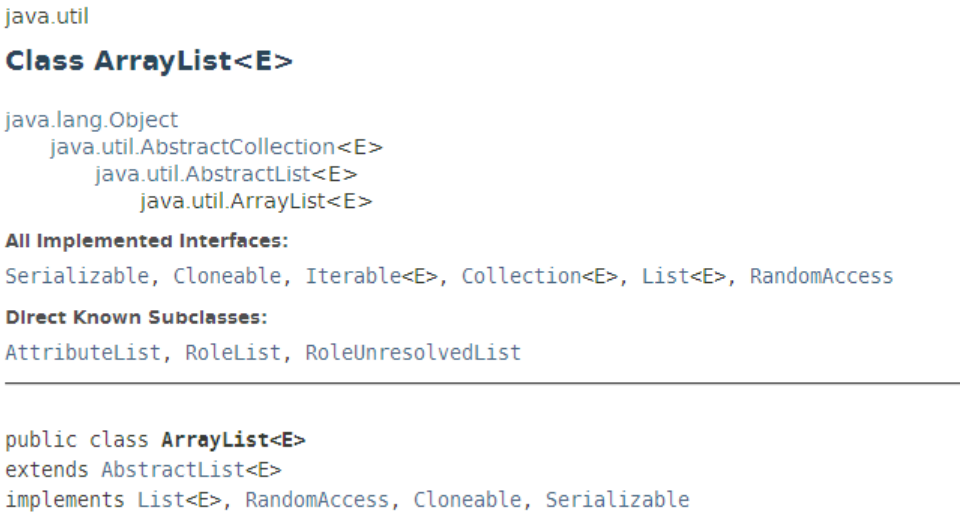

Figura 1: Jerarquia d'`ArrayList<E>`

Exercici 2 *Implementa una classe que llegeixca línies d'un fitxer i les mostre ordenades alfabèticament. Per a açò crea el fitxer* `UsoDeArrayList.java` *i importa els paquets* `java.io.` *per a instanciar el tipus* `File` *i el* `java.util.` *per a les altres classes. Defineix la classe* `UsoDeArrayList` *i en el seu mètode* `main` *realitza els següents passos:*

- *Crea un objecte de la classe* `File` *amb el nom del fitxer com paràmetre i referència'l amb la variable del mateix tipus* `fd`*. Per a la lectura del fitxer, defineix la variable* `fitxer` *de tipus* `Scanner` *i crea una instància d'aquesta classe passant la variable* `fd` *com a argument al constructor. Crea una instància de la classe* `ArrayList<E>` *amb el tipus pur* `String` *i referència-la amb la variable* `llista` *del mateix tipus.*

- *Pots realitzar la lectura amb un bucle* `while` *que comprove en la seua guarda que no s'ha arribat al final del fitxer aplicant* `hasNext()` *a la variable* `fitxer`*. En el cos del bucle llig una línia del text aplicant el mètode* `nextLine()` *a la mateixa variable. Per a afegir la linea a l'objecte de tipus* `ArrayList<String>`*, passa-la com a argument al mètode* `add(E e)` *aplicat a* `llista`*.*

- *Ordena les línies de la llista amb el mètode estàtic* `sort(List<T> list)` *de la classe* `java.util.Collections`*. Aquest mètode rep com paràmetre objectes la classe dels quals implementa l'interfície* `List<E>`*. Entre aquestes classes es troba la classe* `ArrayList<E>`*.*

- *Escriu les cadenes de caràcters guardades en* `llista` *invocant el mètode* `toString` *que per defecte està definit en la classe* `ArrayList`*.*

Pots trobar més informació sobre l'ús dels mètodes en l'API.

5 Genericitat i herència

La màquina virtual no fa servir objectes de tipus genèric, pel que en temps d'execució es perd la informació sobre la variable de tipus utilitzada per parametritzar la classe genèrica. Tota variable de tipus es transforma a un tipus pur en la fase de anàlisi estàtica. Una conseqüència d'això és, per exemple, que amb `instanceof` només podem comprovar si un objecte és de tipus pur. El següent exemple mostra que només podem usar la variable de tipus `T` si també està com a paràmetre en la definició de la classe on s'utilitza

```
class ClaseX<T> {
...
   if (variable instanceof ClaseA<T>)
...
}
```

A més de la possible definició de mètodes d'instància genèrics, com els vistos en els exemples anteriors, hi ha la possibilitat de definir mètodes estàtics genèrics en qualsevol classe, encara que aquesta no siga genèrica. Per a això es precedeix el tipus retornat pel mètode amb les variables de tipus que s'usen en la definició del mètode. En el següent exemple es defineixen dues classes i a la classe `Test` es realitzen crides al mètode estàtic genèric `metodo`.

```
import java.util.*;
class Estaticos {
  public static <T> void metodo(ArrayList<T> p) {
    System.out.println(p); }
}

public class Test {
  public static void main (String a[]) {
    ArrayList<Figura> lF = new ArrayList<Figura>();
    lF.add(new Circulo(1,2,3));
    Estaticos.metodo(lF);
  }
}
```

Com saps, donades dues classes (o interfícies) pures `ClaseB` i `ClaseA`, on la primera deriva de la segona, es diu que la primera és *compatible* amb la segona. És a dir, a una variable de tipus `ClaseA` se li pot assignar un objecte de tipus `ClaseB`.

Prenent un exemple de la pràctica anterior, a una variable de tipus `Figura` se li pot assignar qualsevol objecte amb un tipus `Figura` o hereu de la mateixa. Fins i tot, a una variable de tipus `Volumen` se li pot assignar un objecte de qualsevol classe (`Volumen v=new Cilindro(1,2,3,4);`) que implemente aquesta interfície.

No obstant això, aquesta compatibilitat de tipus es pot perdre amb la genericitat. Donada una classe abstracta `ClaseX<T>`, si es particularitza amb les dues classes pures `ClaseX<ClaseA>` i `ClaseX<ClaseB>`, la segona no és una subclasse de la primera. Així, si intentes compilar el codi següent:

```
1  class ClaseB extends ClaseA {}
2  class ClaseA {}
3  class ClaseX<T> {}
4
5  class TestGenericidadYHerencia {
6    public static void main () {
7        ClaseX<ClaseA> cXA1 = new ClaseX<ClaseA>();
8        ClaseX<ClaseA> cXA2 = new ClaseX<ClaseB>();
9    }
10 }
```

pots comprovar que es produeix un error en la línia 8 per falta de compatibilitat en l'assignació, tot i ser `ClaseB` heretera de la `ClaseA`.

Fent un ús conjugat de genericitat i herència podem plantejar diversos casos. Nota que en els exemples usats a continuació es deriven classes, però també es poden aplicar, idénticament, a la implementació i extensió d'interfícies:

- No propagar la genericitat i definir una classe pura (`ListaString`) a partir d'una genèrica amb tipus pur (`ArrayList<String>`).

  ```
  class ListaString extends ArrayList<String>{...}
  ```

- Mantenir la genericitat de la classe pare (`ArrayList`) mantenint la llista de variables de tipus. Per exemple

  ```
  class ListaOrdenada<T> extends ArrayList<T>{...}
  ```

- Restringir la genericitat d'una variable de tipus escrivint darrere d'ella la paraula reservada `extends` seguida del tipus al qual es vol restringir. Com pots veure en el següent exemple, creant una llista d'objectes tals que el seu tipus necessàriament ha d'estendre `Figura`

  ```
  class ListaFiguras<T extends Figura> extends ArrayList<T>{...}
  ```

  podem crear una llista de `Circulo`:

  ```
  ListaFiguras<Circulo> l = new ListaFiguras<Circulo>();
  ```

  També es pot restringir amb classes genèriques com pots veure en els dos exemples següents en els quals es defineix, en tots dos, una llista de piles genèriques:

  ```
  cclass ListPilas1<T extends Pila<K>,K> extends ArrayList<T>{...}
  class ListPilas2<K> extends ArrayList<Pila<K>>{...}
  ```

- Augmentar la generalitat afegint variables de tipus a les què s'hereten de la classe pare. Com per exemple:

  ```
  class ListaMas<T,G> extends ArrayList <T>{...}
  ```

6 Implementació del tipus genèric `Cola<T>`

Com saps, els tipus de dades lineals són aquells on els elements estan organitzats per linealitats o seqüències a les quals se'ls pot aplicar operacions de modificació i consulta.

Una cua és una estructura lineal FIFO (*First In, First Out*) en la qual el primer element que entra és el primer que ix. L'especificació del tipus `Cola<T>` es descriu en la Figura 2.

En Poliformat pots trobar el directori comprimit de l'aplicació `lineales`. Aquest directori té un subdirectori (paquet Java) `librerias`, el qual conté tres subdirectoris: corresponenents als tres paquets:

librerias.modelos

Interface Cola<T>

`public interface Cola<T>`

Define el TAD de una cola

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
T	**desencolar**() Consulta y extrae el primer elemento, solo si la cola no esta vacia
void	**encolar**(T e) inserta el Elemento en una Cola situandolo al final
boolean	**esVacia**() Comprueba si una Cola esta vacia
T	**primero**() Solo si la cola no esta vacia, consulta el primer elemento en cabeza, (el primero en orden de insercion)
int	**talla**() Devuelve la cantidad de elementos de la cola

Figura 2: Interfície `Cola<T>`

- `librerias.modelos`, que conté una interfície que especifica les operacions de les cues en la interfície `Cola<T>`.
- `librerias.implementacionesDeModelos`, que conté una implementació parcial de la interfície.
- `librerias.aplicaciones`, que conté programes que fan servir el tipus `Cola<T>`.

Exercici 3 *En poliformat pots trobar el fitxer* `ColaAC.java` *parcialment implementat. Aquest fitxer conté alguns atributs i mètodes ja implementats i uns altres que has de completar.*

La implementació de la classe `ColaAC<T>` *ha de ser una implementació de les cues emprant arrays circulars tal com s'il·lustra en la Figura 3. Per tant, l'estructura interna d'un objecte de tipus* `Cola<T>` *ha de tindre almenys:*

- *un array de tipus genèric* `T` *per a guardar els elements de la cua*
- *els atributs* `primer` *i* `últim` *de tipus enter que mantenen una referència als índexs on estan situats el primer i l'últim element de la cua.*
- *un atribut* `talla` *per a representar la quantitat d'elements de la cua.*

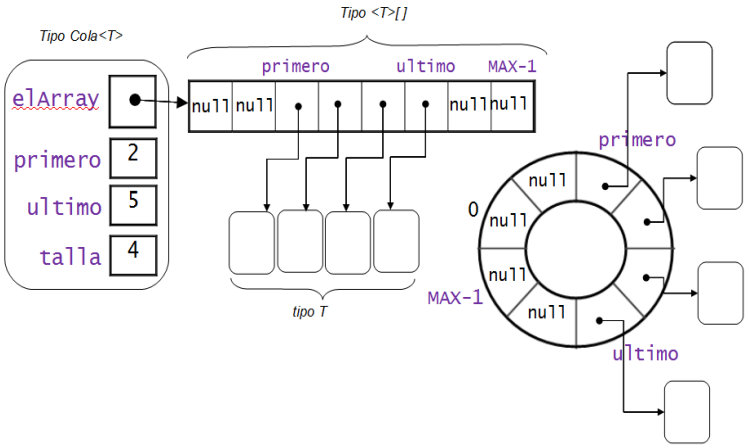

Figura 3: Estructura de dades de la classe `ColaAC<T>`

El mètode privat `int incrementar (int i)` *s'encarrega de retornar la primera posició buida de l'array.*

Una vegada completada la classe, comprova el teu codi executant la classe `AppCola` *en la llibreria* `aplicaciones`*.*

Exercici 4 *Es desitja implementar una cua redimensionable. Per a açò, defineix una nova implementació de l'interfície* `Cola<T>`*, que s'anomene* `ColaAL<T>`*, canviant l'estructura de dades interna suport de la cua a una instància de la classe* `ArrayList<T>`*. Per a açò usa les operacions que es troben especificades en les APIs, entre les quals pots trobar com afegir i eliminar elements en la llista, consultar la grandària de la llista, etc.*

Una vegada completada la nova classe, comprova el teu codi modificant primer (per a fer servir la nova implementació) i executant després, la classe `AppCola` *de la llibreria* `aplicaciones`*.*

7 Avaluació

L'assistència a les sessions de pràctiques és obligatòria per a aprovar l'assignatura. L'avaluació de la primera part de pràctiques de Java es realitzarà mitjançant un examen individual en el laboratori.