Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2020-21

Parte I: Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 2 - Material de lectura previa 2

Genericidad en Java

Índice

1. Clases genéricas

Una clase genérica es una clase normal, salvo que su declaración se parametriza con *variables de tipo* también llamadas *tipos genéricos* o *variables genéricas* en contrapartida a los *tipos puros*.

Los nombres de las variables genéricas se escriben detrás del nombre de la clase y entre `<` y `>`, separados por comas. Los tipos que representan estas variables no son *tipos básicos* sino clases (*tipos referencia*).

`class/interface NomClase` <lista-de-tipos-genéricos> `{ ... }`

Por convenio, estas variables se escriben con una letra mayúscula.

Veamos una introducción al uso de las clases genéricas definiendo e instanciando la clase `G1`, y más adelante veremos el uso de una clase genérica predefinida en el lenguaje.

```
public class G1<T> {
  private T a;
  public G1(T x) { a = x; }
  public String toString() { return `` '' + a; }
}
```

donde `T` es el nombre de una variable que no representa un valor sino un tipo, sin embargo se escribe dentro de la clase como si se tratara de un tipo puro; como ocurre en la segunda línea definiendo el atributo `a` de tipo `T`. Se aprecia que en la definición del constructor no se usa la notación `<...>` entre el nombre de la clase y los parámetros. Sin embargo, la creación de objetos sigue la siguiente sintaxis:

`new NomClase` <lista-de-tipos-puros> `(`lista-parámetros-formales`)`

La sustitución de las variables genéricas por un tipo puro se realiza durante el análisis estático. Como los objetos se crean dinámicamente, solo se pueden crear objetos de tipos puros. Es decir, existen clases genéricas pero no objetos genéricos. En la lista-de-tipos-puros no se pueden nombrar tipos básicos, y en su lugar se usan sus correspondientes *clases envoltorio* que se usarán en el primer ejercicio de esta práctica.

Por ejemplo, podemos crear un objeto de la clase `G1` para que contenga una `String` en su atributo `a` invocando al constructor de la clase con:

```
new G1<String>("hola mundo");
```

Al crear este objeto, la variable genérica `T` ya ha sido cambiada por el compilador al tipo de la clase `String`. En la línea 2 se asigna este tipo al atributo `a` y en la línea 3 al parámetro del constructor. Es como si hubiéramos escrito la clase `G1`:

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

Además, Java proprociona clases genéricas ya predefinidas. Una de ellas, que se usará en el segundo ejercicio de esta práctica, es la clase `ArrayList`, que implementa un array redimensionable cuyos datos son del tipo genérico `<T>` que se sustituirá por algún tipo puro al instanciar la lista, por ejemplo:

```
new ArrayList<String>();
```

2. Genericidad y herencia

La máquina virtual no maneja objetos de tipo genérico, por lo que en tiempo de ejecución se pierde la información sobre la variable de tipo utilizada para parametrizar la clase genérica. Toda variable de tipo se transforma a un tipo puro en la fase de análisis estático. Una consecuencia de esto es, por ejemplo, que con `instanceof` solo podemos averiguar si un objeto es de tipo puro. El siguiente ejemplo muestra que solo podemos usar la variable de tipo `T` si también está como parámetro en la definición de la clase donde se utiliza

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

Además de la posible definición de clases genéricas, como las vistas en los ejemplos anteriores, existe la posibilidad de definir métodos de instancia genéricos y métodos estáticos genéricos en cualquier clase, aunque no sea genérica. Para ello se precede el tipo devuelto por el método con las variables de tipo que se usan en la definición del método. En el siguiente ejemplo se definen dos clases y en la clase `Test` se realizan llamadas al método estático genérico `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[] args) {
    ArrayList<Figure> lf = new ArrayList<Figure>();
    lf.add(new Circle(1, 2, 3));
    Estaticos.metodo(lf);
  }
}
```

Como sabes, dadas dos clases (o interfaces) puras `ClaseB` y `ClaseA`, donde la primera deriva de la segunda, se dice que la primera es *compatible* con la segunda. Es decir, a una variable de tipo `ClaseA` se le puede asignar un objeto de tipo `ClaseB`.

Tomando un ejemplo de la práctica 1, a una variable de tipo `Figure` se le puede asignar cualquier objeto con un tipo `Figure` o heredero de la misma. Incluso

a una variable de tipo `Printable` se le puede asignar un objeto de cualquier clase (`Printable p = new Rectangle(1, 2, 3, 4);`) que implemente esa interfaz.

Sin embargo, esta compatibilidad de tipos se puede perder con la genericidad. Dada una clase genérica `ClaseX<T>`, si se particulariza con las dos clases puras `ClaseX<ClaseA>` y `ClaseX<ClaseB>`, la segunda no es una subclase de la primera. Así, si intentas compilar el código siguiente:

```
1 class ClaseB extends ClaseA {}
2 class ClaseA {}
3 class ClaseX<T> {}

4 class TestGenericidadYHerencia {
5   public static void main(String[] args) {
6     ClaseX<ClaseA> cXA1 = new ClaseX<ClaseA>();
7     ClaseX<ClaseA> cXA2 = new ClaseX<ClaseB>();
8   }
9 }
```

puedes comprobar que se produce un error en la línea 7 por falta de compatibilidad en la asignación, aún siendo `ClaseB` heredera de `ClaseA`.

Haciendo un uso conjugado de genericidad y herencia podemos plantearnos varios casos. Nota que en los ejemplos usados a continuación se derivan clases, aunque también se pueden aplicar a la implementación y extensión de interfaces:

- No propagar la genericidad y definir una clase pura (`StringList`) a partir de una genérica con tipo puro (`ArrayList<String>`).

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

- Mantener la genericidad de la clase padre (`ArrayList`) manteniendo la lista de variables de tipo. Por ejemplo:

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

- Restringir la genericidad de una variable de tipo escribiendo detrás de ella la palabra reservada `extends` seguida del tipo al que se quiere restringir. Como puedes ver en el siguiente ejemplo, creando una lista de objetos cuyo tipo necesariamente debe extender `Figure`:

  ```
  class FiguresList<T extends Figure> extends ArrayList<T> {...}
  ```

  podemos crear una lista de círculos:

  ```
  FiguresList<Circle> l = new FiguresList<Circle>();
  ```

  También se puede restringir con clases genéricas, como puedes ver en los dos ejemplos siguientes en los que se define, en ambos, una lista de pilas genéricas:

  ```
  class StackList1<T extends Stack<K>, K> extends ArrayList<T> {...}
  class StackList2<K> extends ArrayList< Stack<K> > {...}
  ```

- Aumentar la generalidad añadiendo variables de tipo a las que se heredan de la clase padre. Como por ejemplo:

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