4. Genéricos

Date	Empty
Status	Not started
Type	Empty
Units	7. Uso avanzado de clases

Genéricos

- Interfaces genéricas
- Clases genéricas
- Clases genéricas especializas

Genéricos

Los genéricos son aquellos elementos se adaptan de forma automática a una misma funcionalidad pero con diferentes tipos de datos ya que en la creacion no se especa el tipo del mismo. Un ejemplo de genérico con el que ya se ha trabajado son los ArrayList y los HashTables. Si recordáis, cuando se creaba un objeto de tipo ArrayList o HashTable existían dos posibilidades:

```
// sin tipar, por lo que se podía añadir cualquier elemento
// en este caso se utiliza un tipo genérico ArrayList listaEl
```

```
ementosTotales = new ArrayList(); // tipado, por lo que solo
se pueden añadir elementos de tipo String ArrayList<String> l
istaElementosString = new ArrayList(); // sin tipar, por lo q
ue se podía añadir cualquier elemento y cualquier clave // en
este caso se utiliza un tipo genérico Hashtable tablaElemento
sTotales = new Hashtable(); // tipado, por lo que solo se pue
den añadir claves de tipo String y elementos de tipo Object H
ashtable<String, Object> tablaElementosString = new Hashtable
();
```

Si pensáis un poco en su uso, una alternativa al uso de genéricos sería el uso de Object (solución que utilizábamos con los arrays) ya que también admite cualquier tipo de objeto (pero sin restricciones). Sin embargo el uso de genéricos tiene bastantes más ventajas:

- garantizar la seguridad
- evita la necesidad de hacer casteos
- flexibilizar la creación de clases que admiten varios tipos de parámetros
- favorece la creación de métodos que son independientes al tipo de dato

Imaginad que se quiere crear una clase que admite parámetros de tipo String, otra con la misma funcionalidad que admite parámetros de tupo Integer y por último otra que admite parámetros de tipo Boolean. La necesidad de crear tres clases se elimina si se crea una clase que admite objetos de tipo genérico, el cual se define en el momento de creación del objeto.

Si pensáis un poco en su uso, una alternativa al uso de genéricos sería el uso de Object ya que también admite cualquier tipo de objeto (pero sin restricciones)

Cuando se utilizan genéricos se dividen en: interfaces y clases

Interfaces genéricas

Se trata del mismo concepto de interfaz que se ha visto a lo largo del curso, con la diferencia que no se indica alguno de los tipos que admiten los métodos.

Si recordáis una interfaz es un conjunto de métodos abstractos (con la posibilidad de tener algún método escrito - default) y constantes de un tipo concreto. Estos métodos no definían la funcionalidad, sino que marcaban un punto de comunicación entre dos clases que ya tenían

marcaban un punto de comunicación entre dos clases que ya tenían configurada la herencia y no tienen nada que ver entre ellas. Cuando se realizan interfaces genéricas, no se indican los tipos de los métodos que serán utilizados tanto por los argumentos como por los tipos de retorno. Para ello se utilizan las letras T,S,Z de forma genérica (se podrán utilizar tantas letras como sean nacesarias).

Por ejemplo, imaginad que se tiene una ClaseA que tiene una colección de palabras y una ClaseB que tiene una colección de letras

```
public class ClasePalabras { ArrayList<String> listaPalabras;
public ClasePalabras() { this.listaPalabras = new ArrayList
(); } }
```

```
public class ClaseNumeros { ArrayList<Integer> listaNumeros;
public ClaseNumeros() { this.listaNumeros = new ArrayList();
} }
```

En ambas clases lo que se quiere es guardar una colación de datos. Adicionalmente lo que se quiere hacer que tener una interfaz para poder agregar y eliminar datos. Para ello se puede crear una interfaz que tenga la firma de los métodos necesarios. Una solución sería la siguiente:

```
public interface InterfazOperaciones { void agregarElementoPa
labra(String elemento); void elimnarElementoPalabra(String el
emento); void agregarElementoNumero(Integer elemento); void e
limnarElementoNumero(Integer elemento); void mostrarDatos();
}
```

En esta interfaz se crean los métodos que son necesarios en las clases comentadas más arriba y se implementa en ambas:

```
public class ClasePalabras implements InterfazOperaciones{ Ar
rayList<String> listaPalabras; public ClasePalabras() { this.
listaPalabras = new ArrayList(); } @Override public void agre
garElementoPalabra(String elemento) { } @Override public void
elimnarElementoPalabra(String elemento) { } @Override public
void agregarElementoNumero(Integer elemento) { } @Override pu
```

```
blic void elimnarElementoNumero(Integer elemento) { } @Overri
de public void mostrarDatos() { } }
```

Lo malo que tiene esto es que la ClasePalabras tiene métodos que no son necesarios, aunque hagan cosas muy parecidas. La solución a esto es modificar la interfaz y convertirla en genérica. Para esto lo que se hace es definir que el tipo de los métodos no es especificado directamente, utilizando la letra T

```
public interface InterfazOperaciones<T> { void agregarElement
o(T elemento); void eliminarElemento(T elemento); void mostra
rDatos(); }
```

De esta forma esta interfaz admite cualquier tipo de dato. Con esto ahora si se puede implementar de forma correcta en ambas clases, identificando cual es el tipo de cada interfaz. De forma automática se convertirán todos los parámetros de tipo T en el tipo indicado

```
public class ClasePalabras implements InterfazOperaciones<Str
ing>{ ArrayList<String> listaPalabras; public ClasePalabras()
{ this.listaPalabras = new ArrayList(); } @Override public vo
id agregarElemento(String elemento) { this.listaPalabras.add
(elemento); } @Override public void elimnarElemento(String el
emento) { this.listaPalabras.remove(elemento); } @Override pu
blic void mostrarDatos() { for (String palabra: listaPalabras
) { System.out.println(palabra); } } }
```

```
public class ClaseNumeros implements InterfazOperaciones<Inte
ger>{ ArrayList<Integer> listaNumeros; public ClaseNumeros()
{ this.listaNumeros = new ArrayList(); } @Override public voi
d agregarElemento(Integer elemento) { this.listaNumeros.add(e
lemento); } @Override public void elimnarElemento(Integer ele
mento) { this.listaNumeros.remove(elemento); } @Override publ
ic void mostrarDatos() { for (Integer numero: listaNumeros) {
System.out.println(numero); } } }
```

En la llamada de estas clases desde la clase main el trabajo es normal, ya que se crea un objeto de la clase y es la propia clase la que trabaja

con los métodos implementados.

```
public class Entrada { public static void main(String[] args)
{ ClaseNumeros claseNumeros = new ClaseNumeros(); claseNumero
s.agregarElemento(1); claseNumeros.agregarElemento(2); claseN
umeros.agregarElemento(3); claseNumeros.agregarElemento(4); c
laseNumeros.mostrarDatos(); ClasePalabras clasePalabras = new
ClasePalabras(); clasePalabras.agregarElemento("Uno"); claseP
alabras.agregarElemento("Dos"); clasePalabras.agregarElemento
("Tres"); clasePalabras.agregarElemento("Cuatro"); clasePalab
ras.agregarElemento("Uno"); clasePalabras.mostrarDatos(); } }
```

Y la salida

```
1 2 3 4 Uno Dos Tres Cuatro Uno
```

Clases genéricas

Al igual que con las interfaces genéricas, la forma de trabajar con las clases genéricas es idéntica a las clases normales con la única diferencia que no se identifica el tipo de los argumentos, retornos, etc... hasta que el objeto de la clase es creado. Siguiente con el ejemplo anterior, se han creado dos clases que manejan elementos de tipo String e Integre respectivamente. Sin embargo se podría haber indicado a la clase que se gestionan elementos de tipo genérico, pudiendo pasar este genérico hasta la interfaz. Para ello se va a crear una clase nueva con el nombre ClaseElemento que tiene una lista de tipos genéricos

```
public class ClaseElementos<T> { ArrayList<T> listaElementos;
}
```

Al no identificar el tipo de los elementos que se guardarán en el ArrayList existe la posibilidad de guardar cualquier tipo de objeto, siendo este definido en la creación del objeto

```
public class Entrada { public static void main(String[] args)
{ ClaseElementos<Integer> claseNumeros = new ClaseElementos
```

```
(); ClaseElementos<String> clasePalabras = new ClaseElementos
(); } }
```

Así se acaba de crear dos objetos donde uno se ha tipado con String y otro se ha tipado con Integer (sustituyendo al T de la clase). De esta forma la clase Elemento tendrá el tipo correspondiente. Si continuamos con el ejemplo anterior, la clase ClaseElemento implementaría la interfaz creada

```
public class ClaseElementos<T> implements InterfazOperaciones
<T> { ArrayList<T> listaElementos; public ClaseElementos() {
listaElementos = new ArrayList(); } @Override public void agr
egarElemento(T elemento) { listaElementos.add(elemento); } @O
verride public void elimnarElemento(T elemento) { listaElemen
tos.remove(elemento); } @Override public void mostrarDatos()
{ for (T elemento: listaElementos ) { System.out.println(elem
ento); } } }
```

Al no poder identificarle un tipo de dato a la interfaz se utiliza el mismo dato genérico que se ha utilizado en la clase, pasándolo directamente en la definición del objeto. Con esto se podría crear un objeto tipado con String o con Integer de forma completa (con toda la funcionalidad de los métodos) ahorrándonos la creación de una clase

```
public class Entrada { public static void main(String[] args)
{ ClaseElementos<Integer> claseNumeros = new ClaseElementos
(); // Error // claseNumeros.agregarElemento("asd"); claseNum
eros.agregarElemento(1); claseNumeros.agregarElemento(2); cla
seNumeros.agregarElemento(3); claseNumeros.agregarElemento
(4); claseNumeros.mostrarDatos(); ClaseElementos<String> clas
ePalabras = new ClaseElementos(); // Error // clasePalabras.a
gregarElemento(1); clasePalabras.agregarElemento("Uno"); clas
ePalabras.agregarElemento("Dos"); clasePalabras.agregarElemen
to("Tres"); clasePalabras.agregarElemento("Cuatro"); clasePal
abras.mostrarDatos(); } }
```

Clases genéricas especializas

Al igual que en los puntos anteriores se han creado clases genéricas que admitían cualquier tipo de dato, mediante el mecanismo de la especialización se puede crear clases que admitan solo determinado

tipos de genéricos. Imaginad el caso tener la necesidad de crear una clase que admita cualquier tipo de dato numérico, bien sea Integer, Double, Float, Big, etc... Con la forma de crear clases genéricas que hemos visto antes esto es imposible:

```
public class ClaseNumeros<T>{ ArrayList<T> listaNumeros; publ
ic void agregarNumero(T numero){ listaNumeros.add(numero) } p
ublic void mostrarNumeros(){ for(T numero: listaNumeros){ Sys
tem.out.println(numero) } } }
```

Es imposible ya que de la forma en la que se ha programado, cualquier tipo de dato es admitido:

```
public class Entrada{ public static void main(String [] arg
s){ // admite cualquier tipo ListaNumeros numeros1 = new List
aNumeros() // admite solo numero Integer ListaNumeros<Integer
> numeros2 = new ListaNumeros() // admite solo numero Double
ListaNumeros<Double> numeros3 = new ListaNumeros() // admite
solo palabras ListaNumeros<String> numeros4 = new ListaNumero
s() } }
```

Al tipar de forma genérica cualquier tipo puede ser utilizado. En los ejemplo anteriores se puede ver que sin poner genérico entra cualquier dato y tipándolo entra el tipo indicado incluso String, cosa que en este caso no es lo buscado. Para poder restringir un poco el tipo de genéricos que se indica se utiliza la extensión de genéricos. Para eso en la creación de la clase donde se define el genérico se pone que éste debe ser de un grupo concreto

```
public class ClaseNumeros<T extends Number>{ ArrayList<T> lis
taNumeros; public void agregarNumero(T numero){ listaNumeros.
add(numero) } public void mostrarNumeros(){ for(T numero: lis
taNumeros){ System.out.println(numero) } } }
```

Con la definición del genérico :

```
<T extends Number>
```

Se indica que cuando se implemente la clase ClaseNumeros tan solo se podrá tipar con clases que haya extendido de Number. En este caso tan solo serán admitidos los tipos que representan números. Una vez hecho esto si se tipa con un dato diferente se produce un error.

```
public class Entrada{ public static void main(String [] arg
s){ // admite cualquier tipo ListaNumeros numeros1 = new List
aNumeros() // admite solo numero Integer ListaNumeros<Integer
> numeros2 = new ListaNumeros() // admite solo numero Double
ListaNumeros<Double> numeros3 = new ListaNumeros() // admite
solo palabras produciendo un error ListaNumeros<Float> numero
s4 = new ListaNumeros() // admite solo palabras produciendo u
n error ListaNumeros<String> numeros5 = new ListaNumeros() }
}
```