

Paradigmas de Programación

3.3. Genericidad

Departamento de Informática
Universidad de Valladolid

Curso 2020-21

Grado en Ingeniería Informática
Grado en Ingeniería Informática de Sistemas





Genericidad

El término **Genericidad** se refiere a una serie de **técnicas** que permitan escribir algoritmos o definir contenedores de forma que puedan aplicarse a un amplio rango de tipos de datos.

- Haciendo **abstracción** del tipo de datos que contienen o al que son aplicados.
- **Parametrizando** el tipo o tipos de datos que intervienen.

Con el objetivo adicional de mantener la **seguridad** del sistema de tipado.



Casos de uso

- Operaciones aplicables a cualquier dato, independientemente de su tipo (swap).
- Contenedores cuyas operaciones no dependen del tipo de datos almacenado (colas, pilas, listas)
- Acceso uniforme a contenedores secuenciales (arrays, ficheros, etc.)
- Algoritmos y contenedores aplicables a tipos de datos para los que esté definida una determinada operación (ordenación):
Genericidad restringida (*constrained genericity*)



Ejemplo – Búsqueda secuencial

```
type
  TVector = array[1..100] of char;
function Existe(const V: TVector; x: char): boolean;
var i : integer;
begin
  i := 1;
  while (i <= 100) and not (V[i] = x) do i := i+1;
  Existe := i <= 100
end;
```

- Sólo aplicable a arrays de tipo TVector
- El esquema de algoritmo es el mismo para a cualquier contenedor recorrible en secuencia
- Requisitos: Que almacene datos del mismo tipo (o subtipos) que el dato buscado y los datos buscados deben poderse comparar mediante igualdad



Técnicas para Genericidad

- **Tipado dinámico:** Desventaja: La comprobación de tipos se realiza en tiempo de ejecución.
- **Sobrecarga de subrutinas, métodos, operadores:** Desventaja: El código debe reescribirse para cada tipo. Interfiere con el polimorfismo, complicando la resolución de métodos.
- **Templates (C++):** Desventajas: No incluye genericidad restringida, no O.O. Otras de tipo técnico.
- **O.O. – Clase Universal:** Desventaja: Necesita conversiones explícitas (casting).
- **O.O. – Clases Parametrizadas:** (Java, C#, Eiffel)
Problemas de consistencia teórica (covarianza – contravarianza)
- **Prog. Funcional – Tipado Algebraico.**



Otras Tecnicas

- **Variantes**: Los tipos variantes representan la exclusión mutua de varios tipos distintos (típicamente los primitivos, strings y punteros representando arrays). Una variable de tipo variante puede almacenar un valor de cualquiera de estos tipos. Puede existir un selector que marque el tipo almacenado.
- **Metatipos, Metaclases**: Una variable cuyo tipo sea un *meta-tipo* puede almacenar referencias a tipos de datos. Una *meta-clase* es una clase cuyas instancias son referencias a clases. Con una meta-clase es posible crear instancias y aplicar *castings* explícitos de los objetos cuya clase referencia.
 - Delphi: Definición **class of *ClaseBase***
 - Java: Clase **Class**. Método **getClass()** de la clase **Object**.



Tipado dinámico

```
def existe(v,x):  
    for e in v:  
        if e == x: return True  
    return False
```

...

```
> existe(["Perez","Sanchez","Rodriguez"],"Sanchez")  
> True  
> existe([12,34,56],34)  
> True  
> existe([12,34,56],"Sanchez")  
> False
```



Sobrecarga

- Se pueden definir métodos con el mismo nombre siempre que el tipo de los parámetros difiera.

```
static boolean existe(String[] vec, String x) {  
    for(String e : vec) { if(e.equals(x)) return(true); }  
    return(false);  
}  
  
static boolean existe(Integer[] vec, Integer x) {  
    for(Integer e : vec) { if(e.equals(x)) return(true); }  
    return(false);  
}  
  
...  
String[] v = {"Perez", "Sanchez", "Rodriguez"};  
Integer[] w = {12, 34, 56};  
System.out.println(existe(v, "Sanchez"));  
System.out.println(existe(w, 34));  
System.out.println(existe(w, "Sanchez")); // Da error
```




Clase Universal

- En Java toda clase hereda de Object
- Object define el método equals
- En Java los arrays son covariantes: B[] deriva de A[] si B deriva de A.

```
static boolean existe(Object[] vec, Object x) {  
    for(Object e : vec) {  
        if( e.equals(x) ) { return(true); }  
    }  
    return(false);  
}  
...  
String[] v = {"Perez","Sanchez","Rodriguez"};  
Integer[] w = {12,34,56};  
System.out.println(existe(v,"Sanchez"));  
System.out.println(existe(w,34));  
System.out.println(existe(w,"Sanchez")); // No da error
```



Clases y métodos genéricos

- Se pueden definir tipos (clases) parametrizadas con la sintaxis *<Identificador de Tipo,...>*

```
static <T> boolean existe(List<T> vec, T x) {  
    for(T e : vec) {  
        if( e.equals(x) ) { return true; }  
    }  
    return false;  
}  
...  
List<String> v = Arrays.AsList("Perez","Sanchez","Rodriguez");  
List<Integer> w = Arrays.AsList(12,34,56);  
System.out.println(existe(v,"Sanchez"));  
System.out.println(existe(w,34));  
System.out.println(existe(w,"Sanchez")); // Da error
```



Templates en C++

```
#include <iostream>
using namespace std;

template <class T>
bool existe(T* v, int n, T e) {
    for(int i = 0; i < n; i++) {
        if(v[i] == e) { return(true); }
    }
    return(false);
}

int main () {
    int vec[] = {12,34,56};
    cout << existe<int>(vec,3,34) ? "Si" : "No";
    return 0;
}
```

Contenedores (clases parametrizadas)



- Un contenedor almacena elementos de un determinado tipo.
- Cada contenedor define una determinada semántica de acceso, inserción y borrado de los elementos, así como las relaciones existentes entre ellos: precedencia, jerarquía (árbol), vecindad (grafo)
- Algunos contenedores (mapa, diccionario) almacenan pares clave-valor.
- La representación más conveniente es mediante una clase parametrizada por el tipo de datos de los elementos.



Ejemplo: Contenedor Pila

```
class NodoString {
    private String val; // Valor almacenado
    private NodoString sig; // Enlace al nodo siguiente

    public NodoString(String val, NodoString sig) {
        this.val = val; this.sig = sig;
    }

    public String getVal() { return val; }
    public NodoString getSig() { return sig; }
}

class PilaString {
    NodoString lis = null; // Lista de elementos

    public void push(String x) { lis = new NodoString(x,lis); }

    public String pop() {
        String p = lis.getVal(); lis = lis.getSig();
        return p;
    }
}
```



Generalización con Object (I)

```
class Nodo {
    private Object val; // Valor almacenado
    private Nodo sig;   // Enlace al nodo siguiente

    public Nodo(Object val, Nodo sig) {
        this.val = val; this.sig = sig;
    }

    public Object getVal() { return val; }
    public Nodo getSig() { return sig; }
}

class Pila {
    Nodo lis = null; // Lista de elementos

    public void push(Object x) { lis = new Nodo(x,lis); }

    public Object pop() {
        Object p = lis.getVal(); lis = lis.getSig();
        return p;
    }
}
```



Clase Parametrizada (I)

```
class Nodo<T> {
    private T val; // Valor almacenado
    private Nodo<T> sig; // Enlace al nodo siguiente

    public Nodo(T val, Nodo<T> sig) {
        this.val = val; this.sig = sig;
    }

    public T getVal() { return val; }
    public Nodo<T> getSig() { return sig; }
}

class Pila<E> {
    Nodo<E> lis = null; // Lista de elementos

    public void push(E x) { lis = new Nodo<E>(x,lis); }

    public E pop() {
        E p = lis.getVal(); lis = lis.getSig();
        return p;
    }
}
```



Generalización con Object (II)

```
Pila pilaStr = new Pila(); // Pila de Strings
Pila pilaInt = new Pila(); // Pila de Integers

// Toda clase deriva de Object
pilaStr.push("Pedro");
pilaStr.push("Marcos");
pilaInt.push(42);

// Al obtener datos es necesario casting explícito
String cad = (String) pilaStr.pop();
int n = (Integer) pilaInt.pop();

// Es posible mezclar datos
pilaStr.push(23);
// La siguiente instrucción produce error en ejecución
cad = (String) pilaStr.pop();
```




Clase Parametrizada (II)

```
Pila<String> pilaStr = new Pila(); // Pila de Strings
Pila<Integer> pilaInt = new Pila(); // Pila de Integers

// Existe comprobación estática de tipos
pilaStr.push("Pedro");
pilaStr.push("Marcos");
pilaInt.push(42);

// No se requiere casting explícito
String cad = pilaStr.pop();
int n = pilaInt.pop();

// No es posible mezclar datos - Esta sentencia produce
// error en compilación
pilaStr.push(23);
```



Genericidad restringida

- Existen algoritmos que no se pueden aplicar a cualquier tipo de datos. Ejemplo: La ordenación sólo se puede aplicar a datos para los que tenga sentido la operación de comparación.
- En Java (y C#, Eiffel, ...) se puede restringir el alcance de un tipo parametrizado:
- **<T extends Clase>** : Sólo permite clases que sean o deriven de *Clase*.
- **<?>** : Representa una clase desconocida
- **<? extends Clase>** : Representa una clase desconocida pero que es o deriva de *Clase*.
- **<? super Clase>** : Representa una clase desconocida pero que es una clase base de *Clase*.



Ejemplo 1 – Búsqueda Sec.

- Extender el algoritmo de búsqueda secuencial no solo a arrays sino a cualquier contenedor que se pueda recorrer.
- En Java es posible usar un bucle for-each a cualquier contenedor que implemente la interfaz `Iterable<E>` (contiene elementos de tipo E)

```
static <T extends Iterable<E>,E> boolean existe(T vec, E x)
{
    for(E e : vec) {
        if( e.equals(x) ) { return(true); }
    }
    return(false);
}
```



Ejemplo 2 – Ordenación

- Algoritmo para ordenar cualquier lista de elementos de tipo (clase) B: (definición en clase [Collections](#))

```
public static <T extends Comparable<? super T>> void  
    sort(List<T> list)
```

- En esta versión se requiere que los elementos implementen la interfaz [Comparable<A>](#) (que define la operación de comparación entre A's). En una lista de B's, la clase B pueden no implementar directamente la interfaz [Comparable](#), sino basarse en la implementación de alguna clase base suya (A).
- Otra definición, usando una función externa (encapsulada en objeto con interfaz [Comparator<A>](#)):

```
public static <T> void  
    sort(List<T> list, Comparator<? super T> c)
```



Covarianza y Contravarianza

- Dadas las clases (o tipos de datos) **A** y **B**, se dice que **B es compatible con A** si se puede asignar un valor de tipo **B** a una variable de tipo **A**.
- En orientación a objeto, el **polimorfismo** nos dice que si **B deriva de A**, entonces **B es compatible con A**.
- Cuando tratamos con **clases parametrizadas**, las cosas se complican: ¿Bajo que circunstancias son compatibles las clases **List<A>** y **List**?
 - **Covarianza**: **List** es compatible con **List<A>** si **B** es compatible con **A**
 - **Contravarianza**: **List<A>** es compatible con **List** si **B** es compatible con **A**
 - **Incompatibilidad**: **List** y **List<A>** son incompatibles, independientemente de la compatibilidad entre **B** y **A** (salvo que sean la misma la clase, claro).



Herencia y Genericidad (I)

- ¿Una **Lista de Alumnos** es una **Lista de Personas**? (Suponemos que todo **Alumno** es una **Persona**)
- Si la respuesta es si, entonces **Lista<Alumno>** se podría considerar derivado de **Lista<Persona>** (todo contenedor sería **covariante** respecto a su tipo parametrizado)

```
Lista<Alumno> lisA = new Lista();  
... Se introducen alumnos en lisA ...  
// Correcto si son covariantes  
Lista<Persona> lisP = lisA;  
// Correcto, se obtiene un Alumno que es una Persona  
Persona p = lisP.elemento(0);  
// Incorrecto: Se inserta una Persona en una lista  
// de alumnos (el compilador no lo detectaría)  
lisP.incluir( new Persona(...) );
```



Herencia y Genericidad (II)

- Un contenedor parametrizado por B debería ser **covariante** respecto a los accesos (se puede acceder a datos considerando que es un contenedor de A's, donde B deriva de A)
- Sin embargo debería ser **contravariante** respecto a las modificaciones (se pueden insertar datos de clase C , derivada de B) y por tanto se puede considerar que es un contenedor de C's.
- Para manejar estas situaciones, los lenguajes adoptan distintos enfoques:
 - En Java los arrays son **covariantes**.
 - Pero las clases normales no: Son **incompatibles** respecto al tipo parametrizado. Se incorpora el tipo desconocido (*wilcard*, **?**) con modificadores **super** y **extends**.
 - En C# se puede indicar para cada método de una clase si va a ser covariante, contravariante o incompatible.



Ejemplo de uso de sort (I)

```
class Persona { public String nombre; }  
  
// Clase para ordenar Personas por nombre  
class CompNombre implements Comparator<Persona> {  
    public int compare(Persona o1, Persona o2) {  
        return o1.nombre.compareTo(o2.nombre);  
    }  
}  
  
class Alumno extends Persona { public double nota; }  
  
// Clase para ordenar Alumnos por nota  
class CompNota implements Comparator<Alumno> {  
    public int compare(Alumno o1, Alumno o2) {  
        if(o1.nota == o2.nota) { return 0; }  
        else { return (o1.nota < o2.nota ? -1 : +1); }  
    }  
}
```




Ejemplo de uso de sort (II)

- Supongamos que queremos ordenar una lista de alumnos primero por nota y luego por nombre, usando objetos de clase **CompNota** y **CompNombre** para indicar el método de comparación:

```
List<Alumno> lis = new List();  
// Se introducen datos en lis y se ordena por nota  
Collections.sort(lis, new CompNota()); // (1)  
// Se ordena por nombre  
Collections.sort(lis, new CompNombre()); // (2)
```

- Si la declaración de sort hubiera sido:

```
static <T> void sort(List<T> list, Comparator<T> c)
```

- Entonces el primer ordenamiento sería válido (**List<Alumno>** y **Comparator<Alumno>**) pero no el segundo (**List<Alumno>** y **Comparator<Persona>**)



Ejemplo de uso de sort (III)

- Ya que todo **Alumno** es una **Persona**, sería deseable poder ordenar colecciones de alumnos usando comparadores definidos no solo para alumnos sino tambien para personas (y cualquier clase base de persona)
- Para ello no se debe indicar que el tipo parámetro de **Comparator** puede ser cualquier clase base de **T**.
- Así, en lugar de ésta definición:

```
<T> void sort(List<T> list, Comparator<T> c)
```

- Se indica que el tipo parámetro de **Comparator** es desconocido (?) pero base de **T**:

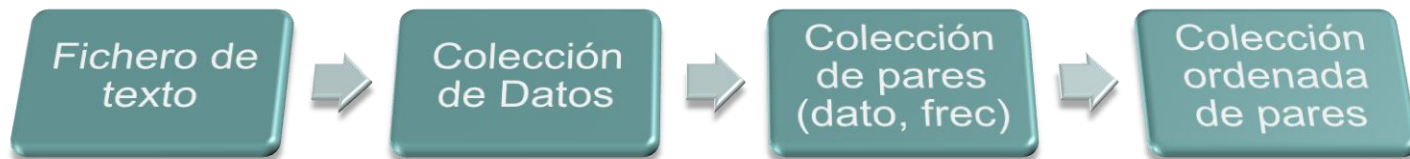
```
<T> void sort(List<T> list, Comparator<? super T> c)
```

EJEMPLO: FRECUENCIADOR



Problema:

- Dado un fichero de texto, se desea obtener la lista de pares (carácter, **frecuencia**) ordenada de mayor a menor por frecuencia, donde ésta indica el número de veces que aparece cada carácter en el texto.
- Se sabe que se va a pedir aplicar este problema a ficheros de texto que contengan otro **tipo de datos** (palabras, enteros, etc.) por lo que es esencial resolver el problema de la forma **más general** posible.





Solución Java (I)

- La clase parametrizada Cuenta<T> representa a un par (dato, frecuencia), donde T es el tipo del dato:

```
public class Cuenta<T> {  
    T elem;  
    int num;  
  
    public Cuenta(T elem) {  
        this.elem = elem;  
        num = 1;  
    }  
  
    public T getElem() { return elem; }  
    public int getNum() { return num; }  
    public void incr() { num++; }  
}
```



Solución Java (II)

```
public static <T> Cuenta<T> busqueda(List<Cuenta<T>> l, T e) {  
    for(Cuenta<T> c : l) {  
        if(c.getElem().equals(e)) { return c; }  
    }  
    return null;  
}  
  
public static <T> List<Cuenta<T>> recuento(List<T> lis) {  
    List<Cuenta<T>> res = new ArrayList();  
    for(T elem : lis) {  
        Cuenta<T> c = busqueda(res,elem);  
        if(c == null) { res.add(new Cuenta(elem)); }  
        else { c.incr(); }  
    }  
    return res;  
}
```



Algoritmo para caracteres

```
System.out.print("Introduzca texto: ");
// Usamos Scanner (expresiones regulares) para leer
// una linea
Scanner sc = new Scanner(System.in);
List<Character> lis = new ArrayList();
String lin = sc.nextLine();
for(i = 0; i < lin.length(); i++) { lis.add(lin.charAt(i)); }
List<Cuenta<Character>> lisc = recuento(lis);
// Para usar sort debemos hacer que Cuenta implemente
// la interfaz Comparable
Collections.sort(lisc);
for(Cuenta<Character> c : lisc) {
    System.out.println(c.getNum()+" : "+c.getElem());
}
```



Algoritmo para palabras

```
System.out.print("Introduzca texto: ");  
// Usamos Scanner (expresiones regulares) para trocear  
// el texto en palabras  
Scanner sc = new Scanner(System.in);  
List<String> lis = new ArrayList();  
String regex = "[A-Za-z]+";  
while(sc.hasNext(regex)) { lis.add(sc.next(regex)); }  
List<Cuenta<String>> lisc = recuento(lis);  
// Para usar sort debemos hacer que Cuenta implemente  
// la interfaz Comparable  
Collections.sort(lisc);  
for(Cuenta<String> c : lisc) {  
    System.out.println(c.getNum()+" : "+c.getElem());  
}
```




Algoritmo para enteros

```
System.out.print("Introduzca texto: ");  
// Usamos Scanner (expresiones regulares) para trocear  
// el texto en enteros  
Scanner sc = new Scanner(System.in);  
List<Integer> lis = new ArrayList();  
while(sc.hasNextInt()) { lis.add(sc.nextInt()); }  
List<Cuenta<Integer>> lisc = recuento(lis);  
// Para usar sort debemos hacer que Cuenta implemente  
// la interfaz Comparable  
Collections.sort(lisc);  
for(Cuenta<Integer> c : lisc) {  
    System.out.println(c.getNum()+" : "+c.getElem());  
}
```



Cuenta Comparable

```
public class Cuenta<T> implements Comparable<Cuenta> {
    T elem;
    int num;

    public Cuenta(T elem) { this.elem = elem; num = 1; }

    public T getElem() { return elem; }
    public int getNum() { return num; }
    public void incr() { num++; }

    public int compareTo(Cuenta o) {
        if(o.getNum() == num) { return 0; }
        else if(o.getNum() > num) { return 1; }
        else { return -1; }
    }
}
```