CLASES GENÉRICAS

Genérico.
"Genérico" se refiere a "tipo(de clase) Genérico". Java permite escribir código más genérico permitiendo en el código declarar los atributos referencia de un tipo de objeto genérico en lugar de un tipo de objeto concreto. Esto es especialmente útil en algunos contextos, como cuando se trabaja con colecciones.

Ejemplo:
Supongamos que en una aplicación tenemos la necesidad de trabajar con pares. Un par está formado por dos referencias a objetos. Esos objetos pueden ser de diversos tipos(Integer, Double, Piedras, Manzanas, ...). En este contexto, utilizar genéricos es la mejor solución.

Lo más importante, observa en el ejemplo estas tres cosas:

- Un tipo genérico se representa por una letra mayúscula como T.
- El tipo genérico se usa en la declaración de la clase genérica
- El objeto de una clase genérica no es genérico ¡Tiene un tipo concreto!, ya que al crear un objeto se indica un tipo concreto al lado del nombre del constructor.

```
class Pair<T> {
   private T first;
   private T second;

   public T getFirst() {
      return first;
   }

   public T getSecond() {
      return second;
   }

   public void setFirst(T first) {
      this.first = first;
   }

   public void setSecond(T second) {
      this.second = second;
   }
}
```

Un Par es un contenedor para almacenar una tupla de dos objetos

A continuación veremos una serie de ejemplos con comportamientos que funcionan y comportamientos que dan error(comentados).

```
class Persona{
   String nombre;
   int edad;

   Persona(String nombre,int edad){
      this.nombre=nombre;
      this.edad=edad;
   }
}
```

```
class App {
   public static void main(String[] args) {
      Pair<Integer> parDeInteger=new Pair<Integer>();//Observa cómo al crear el objeto se concreta el tipo
      parDeInteger.setFirst(4);
      //parDeInteger.setSecond("hola"); //ERROR se espera un objeto Integer


      Pair<String> parDeString=new Pair<String>();
      parDeString.setFirst("hola");
      //parDeString.setSecond(4); //ERROR se espera un objeto String



      Persona p1= new Persona("Elías",5);
      Persona p2= new Persona("Román",4);
      Pair<Persona> parDePersona=new Pair<Persona>();
      parDePersona.setFirst(p1);
      parDePersona.setSecond(p2);
      System.out.println("el nombre de la primera persona es: " +parDePersona.getFirst().nombre);
      System.out.println("el nombre de la segunda persona es: " +parDePersona.getSecond().nombre);
   }
}
```

Convenciones para los nombres de parámetro

Por convención, los tipos de parámetros son letras mayúsculas. Es al revés que la convención para nombres de variables (minúsculas) precisamente para evitar en el código que los tipos se confundan con variables en los parámetros.

Normalmente, se suele utilizar las letras de la siguiente forma(lo ponemos literal del doc oracle):

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

MÉTODO GENÉRICO

Acabamos de escribir una clase genérica indicando en la cabecera de la clase <T>

Los métodos también pueden ser genéricos indicando de forma parecida en su cabecera un <T>. En el siguiente ejemplo, una clase no genérica contiene un método genérico

```
class Util {
   static <T> void imprimirArray (T[] t) {
      for(int i=0;i<t.length;i++)
         System.out.println(t[i].toString());
   }
   //aquí vendrían por ejemplo otros métodos utilidad no necesariamente genéricos
}
```

Imagina que la clase Util tiene muchos métodos, y que sólo uno de esos métodos, el método imprimirArray() necesita un tipo genérico, no es necesario ni apropiado definir el tipo genérico a nivel de clase, se hace directamente en el método.

No confundas método genérico con método que está dentro de una clase genérica. Observa este método de Pair

```
public T getFirst() {
```

```
        return first;
    }
```

No es genérico, es un método de una clase genérica y que utiliza el tipo genérico definido en su clase. Para que un método se le pueda llamar genérico debe el propio método indicar que va a usar su propio tipo genérico con <> en la cabecera.

Observa los detalles de sintaxis para indicar un genérico en clase y en método:

- Para las clases *NombreClase<LetraGenérico>* ejemplo Pair<T>
- Para los métodos *<LetraGenérico> tipoRetorno NombreMétodo()* (luego, si nos hace falta podemos usar argumentos del tipo genérico) ejemplo <T> void imprimir()

INFERENCIA DE TIPOS(Type Inference)

https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

El compilador java tiene la habilidad de inferir (deducir) los tipos con lo que va a trabajar una clase/interface/genérico analizando el contexto. Veremos varios ejemplos, algunos te pueden resultar duros pero hay que echarles un vistazo para apreciar la complejidad de la inferencia de tipos.

La situación más básica de inferencia de tipos es la creación de un objeto de una clase genérica.

Ya lo vimos, recuerda la clase Pair<T>. Al hacer esto

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

Podemos decir que el compilador infiere T como tipo String.
A veces a esto no se le considera inferencia sino asignación de tipo directa, pero en definitiva inferir un tipo es averiguar un tipo y esto es una forma de averiguarlo.

En las instrucciones de asignación la parte izquierda puede determinar los tipos de la derecha, es decir, los tipos de la derecha se infieren de la izquierda

Consideremos el método Collections.emptyList, cuya definición en el API es:

```
static <T> List<T> emptyList();
```

En el siguiente contexto:

```
List<String> listOne = Collections.emptyList();
```

T es de tipo String

El operador diamante

Es un caso particular de lo anterior, java infiere tipos de la derecha analizando los tipos de la parte izquierda, pero aplicado a la creación de objetos

El operador diamante <>, es decir los signos para indicar tipo, pero vacíos, sin tipo es una consecuencia de la inferencia de tipos

No siempre es necesario indicar los tipos al crear un objeto de forma que

```
Pair<Persona> parDePersona=new Pair<Persona>();
```

Se puede escribir

```
Pair<Persona> parDePersona=new Pair<>();
```

<> no significa "sin tipo" o "tipo vacío". Significa que el tipo debe inferirse de la expresión.

La segunda forma se incorporó más recientemente y por eso aún se ve la primera, aunque se prefiere la segunda.
En esta segunda forma los tipos de Pair<> (con diamante vacío) se infieren de la definición de la variable referencia, por tanto, si no defino una variable referencia y hago un new "aislado", entonces sí que es necesario especificar los tipos en el constructor

```
new Pair<Integer>();
```

Otro ejemplo. La clase MyClass tiene un genérico X pero además su constructor es un método genérico que tiene un tipo T

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Si creamos una instancia

```
new MyClass<Integer>("")
```

Está claro que X es Integer. Por otro lado T se infiere como String ya que se pasa al constructor un "" de parámetro

En la siguiente creación de instancia, se infieren los dos tipos. Ahora al usar el operador diamante también inferimos el tipo de X:

```
MyClass<Integer> myObject = new MyClass<>("");
```

En contextos de mezcla de tipos la solución de la inferencia es inferir un supertipo

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Para empezar consulta en el API que un String y un ArrayList implementan Serializable

T es forzosamente del tipo Serializable, es el único tipo que tiene sentido en ese contexto.
Observa que se manejan 3 tipos: Serializable, String y ArrayList<String> pero sólo hay una T. La solución es que T es serializable pues un String y ArrayList implementan el interface Serializable y por tanto si a1 y a2 son serializable pueden recibir un String y una ArrayList respectivamente.

Tipo objetivo(Target type).
Después del ejemplo anterior, decimos más formalmente que el compilador para hacer la inferencia de tipos se basa en en el concepto de tipo objetivo de una expresión, que es el tipo de datos que el compilador java espera dependiendo donde aparezca la expresión(dependiendo del contexto).

Inferencia de tipos en variables locales
Las variables locales también pueden ser una especie de variables genéricas de forma que se puede deducir su tipo al declararlas.
Observa la nueva palabra reservada "var"

```
class App {
   public static void main(String[] args) {
      var x=3;
      x=6;
      //el tipo de x es int, no se le puede asignar ahora un String
      //x="hola";
```

```
    }
}
```

El uso de var puede ser cómodo en ciertos contextos como en el for mejorado, para despreocuparme de definir el tipo de la variable.

```
class App {

    public static void main(String[] args) {
      int[] listaEnteros={1,2,3,4,5};
      boolean[] listaBoolean={true,false,false,true};
      for(var e:listaEnteros){
         System.out.print(e+" ");
      }
      System.out.println("");
      for(var e:listaBoolean){
         System.out.print(e+" ");
      }
      System.out.println("");
     }
}
```

CLASE GENÉRICA CON VARIOS PARÁMETROS

Si por ejemplo, quisiéramos que el par de elementos fuesen de dos tipos, independientes el uno del otro, podríamos redefinir la clase Pair como:

```
class Pair<T, V> {
    private T first;
    private V second;

    public T getFirst() {
        return first;
    }
    public V getSecond() {
        return second;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setSecond(V second) {
        this.second = second;
    }
}
```

Ejercicio U6_B2_E1: Utilizando la definición de clase anterior Pair, escribe un main en App de forma que cree :

- un par de Integer y Float
- un par de String y array de enteros.
- Un par de Persona y Persona. Utiliza la clase Persona del ejemplo de más arriba en este documento.

e inicialice los valores de los pares con los métodos set()

Ejercicio U6_B2_E2: Añade a la clase genérica anterior un constructor de forma que al crear los pares se pueda pasar al constructor los valores de first y second, de forma que en el ejercicio anterior no es necesario utilizar los métodos Set.

Ejercicio U6_B2_E3: Con la clase

```
class Util {
   static <T> void imprimirArray (T[] t) {
      for(int i=0;i<t.length;i++)
         System.out.println(t[i].toString());
   }
}
```

App utiliza el método anterior de forma que imprime un array de Integer, un array de Double y un array de personas . Crea tú tres arrays sencillitos de las clases anteriores e invoca al método.

GENÉRICOS Y HERENCIA: OBSERVACIONES.

Hay muchas consideraciones pero sobre todo nos centramos en que una clase genérica al extenderse puede seguir siendo genérica o no:

```
class MyGeneric<T, E> {}
class Extend1<T, E> extends MyGeneric<T, E> {}
class Extend2 extends MyGeneric<String, Object> {}
```

Observa que Extend1 es genérica, pero Extend2 no lo es, ya se usa un MyGeneric con tipos concretos
Esta idea también es aplicable a la implementación de interfaces genéricos.

A partir de aquí surgen más combinaciones posibles ...

Ejemplos de cómo una clase no genérica puede extender a una clase genérica.

A este proceso le podemos llamar "particularizar una clase genérica".

Al margen de que sea útil o no, observa como

```
class Pair<T, V> {
   private T first;
   private V second;

   public T getFirst() {
      return first;
   }
   public V getSecond() {
      return second;
   }
   public void setFirst(T first) {
      this.first = first;
   }
   public void setSecond(V second) {
      this.second = second;
   }
}

class PairInteger extends Pair<Integer,Integer>{

}
public class App{
```

```
   public static void main(String args[]){
      PairInteger pi=new PairInteger();
      pi.setFirst(2);
      pi.setSecond(4);
   }
}
```

La misma idea se aplica al implementar interfaces

```
interface Pair<T, V> {
    public T getFirst();
    public V getSecond();
    public void setFirst(T first);
    public void setSecond(V second);
}

class PairInteger implements  Pair<Integer,Integer>{
    private Integer first;
    private Integer second;

    @Override
    public Integer getFirst() {
        return first;
    }

    @Override
    public Integer getSecond() {
        return second;
    }

    @Override
    public void setFirst(Integer first) {
        this.first=first;
    }

    @Override
    public void setSecond(Integer second) {
        this.second=second;
    }


}
public class App{
   public static void main(String args[]){
      PairInteger pi=new PairInteger();
      pi.setFirst(2);
      pi.setSecond(4);
   }
}
```

Ejemplo de cómo una clase genérica puede extender/implementar a una clase/interfaz genérica.

Simplemente reescribimos el ejemplo anterior

```
 interface Pair<T, V> {
    public T getFirst();
    public V getSecond();
```

```
    public void setFirst(T first);
    public void setSecond(V second);
}

class MiClasePar<T,V> implements  Pair<T,V>{
    private T first;
    private V second;

    @Override
    public T getFirst() {
        return first;
    }

    @Override
    public V getSecond() {
        return second;
    }

    @Override
    public void setFirst(T first) {
        this.first=first;
    }

    @Override
    public void setSecond(V second) {
        this.second=second;
    }


}
public class App{
   public static void main(String args[]){
      MiClasePar<Integer,Integer> pi=new MiClasePar<>();
      pi.setFirst(2);
      pi.setSecond(4);
   }
}
```

Parametro delimitado (Bounded parameter).

La palabra extends tiene un uso especial con genéricos. Dado un tipo *T* si escribimos

T extends NombreClase
T puede tomar cualquier Tipo que extienda a Clase
O bien
T extends NombreInterface
T puede tomar cualquier Tipo que implemente al Interface

La utilidad de este mecanismo es para obligar a los que usan nuestra clase genérica a utilizar un conjunto más concretos de tipos, ya que si no lo hacen así tendrán errores en tiempo de compilación.

Ejemplo
Lo que hacen las clases no es significativo. Nos centramos en el aspecto sintáctico. Es importante pegar este código en el IDE y ver como el cuarto caso DA ERROR debido a que

estamos intentando que el tipo T tome el valor String y esto no es permitido por la definición T extends A . En este ejemplo los únicos tipos que cumplen esta definición son A, B y C

```
class ConParametroLimitado<T extends A> {
   private T x;
   public ConParametroLimitado(T x) {
      this.x = x;
   }
}

class A {
   private int a = 1;
}

class B extends A {
   private int b = 2;
}

class C extends A {
   private int c = 3;
}

public class App {
   public static void main(String a[]) {
      new ConParametroLimitado<A>(new A());
      new ConParametroLimitado<B>(new B());
      new ConParametroLimitado<C>(new C());

      //ERROR
      new ConParametroLimitado<String>(new String("hola"));

   }
}
```

Tipos crudos (Raw types)

Observa que una clase puede contener uno (o varios) tipos "genéricos". Pero los objetos(instancias) de una clase NO SON GENÉRICOS ya que su tipo se concreta al crear la instancia. ¿Qué pasa si no indicamos el tipo al crear un objeto de una clase genérica? Esperaríamos un error de compilación, pero por cuestiones de compatibilidad anticuadas para nosotros, no se genera error de compilación si no que se asigna un "tipo crudo" (una especie de tipo por defecto)

```
class Pair<T> {
   private T first;
   private T second;

   public T getFirst() {
      return first;
   }

   public T getSecond() {
      return second;
   }

   public void setFirst(T first) {
      this.first = first;
   }

   public void setSecond(T second) {
      this.second = second;
   }
}
class MiClase{
```

```
   int x;
}
public class App{
  public static void main(String args[]){
     Pair<Integer> parInteger= new Pair<>(); //tipo parametrizado OK
     Pair parRaw= new Pair<>();//tipo crudo  ¡EVITAR AUNQUE NO DE ERROR!
     Pair<Object> parObject= new Pair<>();//tipo parametrizado OK

     //esto provoca que javac me recomiende recompilar con  -Xlint para ver warnings
     parInteger=parRaw;

  }
}
```

Los tipos crudos se mantienen por compatibilidad con las aplicaciones anteriores a la aparición de genéricos (en JDK antiguos no podíamos especificar un tipo genérico en las clases porque no existían genéricos). En el código actual, y por tanto el que escribimos nosotros, siempre se debe evitar trabajar con tipos crudos. De hecho si utilizamos tipos crudos recibiremos advertencias (warnings) del compilador.

Compilar con -X

La opción -X (la "X" mayúscula) indica a javac que quiero compilar de forma "no standard". Luego a este -X se le indica el matiz que quiero, concretamente consulto

https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html

Y veo que Xlint es para ver todos los warnings.

-Xlint

Enables all recommended warnings. In this release, enabling all available warnings is recommended.

Entre estos warnings se encontrarán la omisión de tipos

Los warnings no son errores, son advertencias de posibles problemas que pueden manifestarse en tiempo de ejecución. Al darle a "play" en un IDE no se aprecian estos warnings. Para ver los warnings necesitamos configurar el IDE o hacer manualmente el proceso javac/java en la consola

Compilamos el ejemplo anterior:

```
C:\Users\donlo\Documents\proyectos visual studio code\java\MiApp>cd src

C:\Users\donlo\Documents\proyectos visual studio code\java\MiApp\src>javac App.java
Note: App.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\donlo\Documents\proyectos visual studio code\java\MiApp\src>javac App.java -Xlint
App.java:27: warning: [rawtypes] found raw type: Pair
       Pair parRaw= new Pair<>();//tipo crudo  ¡EVITAR AUNQUE NO DE ERROR!
       ^
  missing type arguments for generic class Pair<T>
  where T is a type-variable:
    T extends Object declared in class Pair
App.java:31: warning: [unchecked] unchecked conversion
       parInteger=parRaw;
                  ^
  required: Pair<Integer>
  found:    Pair
2 warnings

C:\Users\donlo\Documents\proyectos visual studio code\java\MiApp\src>
```

Ejercicio U6_B2_E4: Recuerda la implementación de lista enlazada de enteros que vimos al estudiar estructuras dinámicas. En el siguiente ejemplo la lista está modificada de forma que es genérica y puede trabajar con cualquier tipo de dato, puedo por ejemplo tener un lista de objetos Persona. El siguiente código no obstante, por despiste, usa tipos crudos. La ejecución es correcta pero ya se discutió que deben evitarse.

Compila con y sin -Xlint y corrige para evitar warnings.

```
class Persona{
    String nombre;
    int edad;

    Persona(String nombre,int edad){
        this.nombre=nombre;
        this.edad=edad;
    }
    @Override
    public String toString(){
        return   "("+nombre+", "+edad+")";
    }

}

class Nodo<T>{
  private Nodo sig;
  private T dato;
  public Nodo(T dato) {
    this.dato = dato;
    this.sig=null;
  }
  public Nodo(T dato, Nodo sig) {
    this.dato = dato;
    this.sig = sig;
```

```
  }
  public void setSiguiente(Nodo sig) {
    this.sig = sig;
  }
  public Nodo getSiguiente() {
    return sig;
  }
  public T getDato() {
    return dato;
  }
}
class MiListaEnlazada<T>{
    private Nodo<T> primero=null;
    public void insertar(T dato){
        if(primero==null){
            primero=new Nodo<>(dato);
        }else{
            Nodo<T> temp= new Nodo<>(dato,primero);
            primero=temp;
        }

    }
    public int tamano(){
        int i=0;
        Nodo<T> temp=primero;
        while(temp!=null){
            i++;
            temp=temp.getSiguiente();
        }
        return i;
    }
    public T obtener(int indice){
        Nodo<T> temp=primero;
        int i=0;
        while(i<indice){
            temp=temp.getSiguiente();
            i++;
        }

        return temp.getDato();
    }
}



public class App {

    public static void main(String[] args) {

        MiListaEnlazada<Persona> lp= new MiListaEnlazada<>();
        lp.insertar(new Persona("yo",23));
        lp.insertar(new Persona("tu",24));
        lp.insertar(new Persona("el",25));

        for(int i=0;i<lp.tamano();i++)
            System.out.print(lp.obtener(i)+"  ");

        //ERROR: int es tipo primitivo. Los genéricos tiene que ser Clases
        //MiListaEnlazada<int> li= new MiListaEnlazada<>();
        //Integer O.K. Observa autoboxing
        MiListaEnlazada<Integer> li= new MiListaEnlazada<>();
        li.insertar(4);li.insertar(5);li.insertar(6);

       System.out.println();
        for(int i=0;i<li.tamano();i++)
            System.out.print(li.obtener(i)+"  ");
    }
}
```