# UTILIZACIÓN AVANZADA DE CLASES(III)

En este documento trataremos:

- Clase **Object**.
- **InstanceOf**
- Conversión entre tipos de objetos: **Casting**.
- Clase **Genéricas** o parametrizadas.

Como hemos visto en el 3.3. un mecanismo utilizado en POO es el uso de "Polimorfismo", que se puede aplicar tanto entre clase padres – clases hijas y clases que utilicen una interfaz común.

El polimorfismo introduce ciertas ventajas de cara a la generalización, atribuyendo a un mismo identificador definido mediante una clase común diversas clases que comparten dicho nexo en común.

De este modo se gana tanto generalización a la hora de instanciar, como especialización dentro de los métodos comunes. A costa de perder acceso a atributos o métodos codificados en las clases situadas en una jerarquía inferior.

Por lo tanto, hay que tener cuidado a la hora de utilizarlo. Por ejemplo, si luego queremos utilizar un método definido exclusivamente dentro de una clase hija, aunque esta sea instancia como se ha definido utilizando la clase padre no contaríamos/podríamos acceder a dichos métodos específicos.

A la hora de trabajar con polimorfismo contaremos con diversas herramientas/utilidades como son: "instanceOf()" y el "casting de Objetos".

También es importante entender que Java siempre le ha dado la habilidad de crear clases, interfaces y métodos generalizados operando a través de referencias del tipo **Object**. Debido a que ***Object*** es la superclase de todas las demás clases, una referencia de *Object* puede referirse a cualquier tipo de objeto

En los primeros apartados utilizaremos 3 clases propias como ejemplo sencillo del uso de herencia: ClasePadre, ClaseHija1, ClaseHija2.

```java
public class ClaseHija1 extends ClasePadre{

    private int atrHija1;


    public ClaseHija1(int atrPadre, int atrHija1) {
        super(atrPadre);
        this.atrHija1 = atrHija1;
    }

    public int getatrHija1() {
        return atrHija1;
    }

    public void setatrHija1(int atrHija1) {
        this.atrHija1 = atrHija1;
    }

    public void hola() {
        System.out.println("hola hija1");
    }

    public void hija1() {
        System.out.println("hija1 solo");
    }

    @Override
    public String toString() {
        return "ClaseHija1 [atrHija1=" + atrHija1 +
                ", getAtrPadre()=" + getAtrPadre() + "]";
    }

}
```

```java
public class ClaseHija2 extends ClasePadre{

    private int atrHija2;

    public ClaseHija2(int atrPadre, int atrHija2) {
        super(atrPadre);
        this.atrHija2 = atrHija2;
    }

    public int getatrHija2() {
        return atrHija2;
    }

    public void setatrHija2(int atrHija2) {
        this.atrHija2 = atrHija2;
    }

    public void hola() {
        System.out.println("hola hija2");
    }

    public void hija2() {
        System.out.println("hija2 solo");
    }

    @Override
    public String toString() {
        return "ClaseHija2 [atrHija2=" + atrHija2 +
                ", getAtrPadre()=" + getAtrPadre() + "]";
    }

}
```

```java
public class ClasePadre {

    private int atrPadre;

    public ClasePadre(int atrPadre) {
        this.atrPadre = atrPadre;
    }

    public int getAtrPadre() {
        return atrPadre;
    }

    public void setAtrPadre(int atrPadre) {
        this.atrPadre = atrPadre;
    }

    public void hola() {
        System.out.println("hola padre");
    }

    @Override
    public String toString() {
        return "ClasePadre [atrPadre=" + atrPadre + "]";
    }

}
```

# 6. Clase Object:

https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html

La clase Object está situada en la parte más alta del árbol de la herencia en el entorno de desarrollo de Java. Todas las clases del sistema Java son descendentes (directos o indirectos) de la clase Object. Esta clase define los estados y comportamientos básicos que todos los objetos deben tener, como la posibilidad de compararse unos con otros, de convertirse a cadenas, de esperar una condición variable, de notificar a otros objetos que la condición variable ha cambiado y devolver la clase del objeto.

Por lo tanto, todas las clases pueden instanciarse a partir de la definición de un objeto que utilice la clase Object.

```
Object cp = new ClasePadre(0);
Object ch1 = new ClaseHija1(0, 1);
Object ch2 = new ClaseHija2(0, 2);
```

Siendo sus métodos propios (de la clase Object):

| Método | Sintaxis | Propósito |
|---|---|---|
| getClass | class _ getClass() | Obtiene la clase de un objeto en tiempo de ejecución |
| hashCode | int hashCode | Devuelve el código hash asociado con el objeto invocado |
| equals | boolean equals(Object obj) | Determina si un objeto es igual a otro |
| clone | Object clone() | Crea un nuevo objeto que es el mismo que el objeto que se está clonando |
| toString | String toString() | Devuelve una cadena que describe el objeto |
| notify | void notify() | Reanuda la ejecución de un hilo esperando en el objeto invocado |
| notifyAll | void notifyAll() | Reanuda la ejecución de todo el hilo esperando en el objeto invocado |
| wait | void wait(long timeout) | Espera en otro hilo de ejecución |
| wait | void wait(long timeout,int nanos) | Espera en otro hilo de ejecución |
| wait() | void wait() | Espera en otro hilo de ejecución |
| finalize | void finalize() | Determina si un objeto es reciclado (obsoleto por JDK9) |

# 7. InstanceOf

El operador instanceof nos permite comprobar si un objeto es de una clase concreta u otra.

```
Object Ocp = new ClasePadre(0);
Object Och1 = new ClaseHija1(0, 1);
Object Och2 = new ClaseHija2(0, 2);


System.out.println(Ocp instanceof ClasePadre);
System.out.println(Och1 instanceof ClaseHija1);
System.out.println(Och2 instanceof ClaseHija2);
```

Por ejemplo, tenemos 3 clases: Empleado (clase padre), Comercial (clase hija de Empleado) y Repartidor (clase hija Empleado).

```
public class EmpleadoApp {

    public static void main(String[] args) {

        Empleado empleados[]=new Empleado[3];
        empleados[0]=new Empleado();
        empleados[1]=new Comercial();
        empleados[2]=new Repartidor();

        for(int i=0;i<empleados.length;i++){
            if(empleados[i] instanceof Empleado){
                System.out.println("El objeto en el indice "+i+" es de la clase Empleado");
            }
            if(empleados[i] instanceof Comercial){
                System.out.println("El objeto en el indice "+i+" es de la clase Comercial");
            }
            if(empleados[i] instanceof Repartidor){
                System.out.println("El objeto en el indice "+i+" es de la clase Repartidor");
            }

        }

    }

}
```

# 8. Casting de Objetos

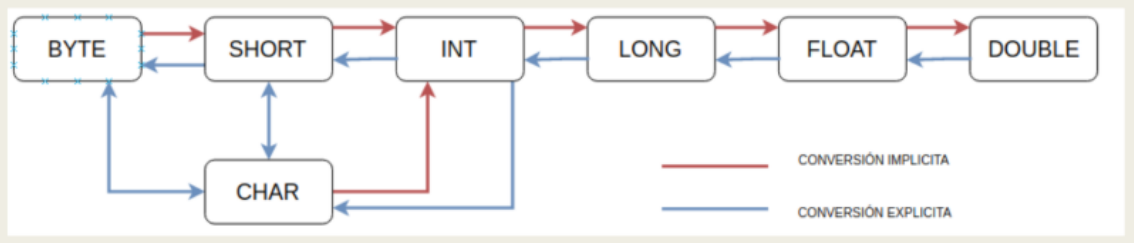
Dentro de los primeros conceptos básicos de programación vistos al inicio del curso vimos en que consistía la conversión de tipo o "Casting" de tipos.

## 8.1 USO DE CASTING PARA LA CONVERSIÓN DE TIPOS PRIMITIVOS (recordatorio).

En la conversión de tipos teníamos:

1. Conversión implícita.
2. Conversión explícita.

```
int attrInteger = 5;
double attrDouble = 5.5;
double atr2 = attrInteger;     // Conversión implicita.
int att2 = (int) attrDouble;   // Conversión explicita o Casting
```




## 8.2. USO DE CASTING PARA LA CONVERSIÓN DE OBJETOS.

¿Por qué querríamos hacer algo así? Por muchos motivos. Por ejemplo, en las siguientes situaciones:

- Cuando se trata de objetos creados mediante clases absolutamente diferentes entre sí, no conectadas por la jerarquía de clases, pero que comparten una o más Interfaces de clase.
- Cuando se trata de objetos creados a partir de varias subclases que tienen una misma clase padre común.

CASTING CONVERSIÓN HACIA ABAJO. DE LA CLASE PADRE HACIA LAS CLASES HIJAS.

En la conversión hacia arriba (polimorfismo) se gana generalidad, pero se pierde información

Por lo que si queremos convertir un objeto a otro objeto podemos utilizar el **casting**.

```
Figura f = new Circulo(); // Conversion hacia arriba
Circulo c = (Circulo)f; // Conversion hacia abajo mediante
// cambio de tipo (casting)
// Ahora mediante la referencia c podemos acceder a TODOS los miembros
// de la clase Circulo. Con la referencia f SOLO podemos acceder a los
// miembros declarados en la clase Figura
```

* También se puede hacer un **cambio de tipo temporal** sin necesidad de almacenar el nuevo tipo en otra variable referencia:

```
Figura f = new Circulo(); // Conversion hacia arriba
((Circulo)f).radio=1; // Acceso a miembros de la subclase
// A continuacion f sigue siendo de tipo Figura
```

En este caso mediante la sentencia ((Circulo)f) se cambia el tipo de f únicamente para esta instrucción, lo que permite acceder a cualquier miembro definido en Circulo (por ejemplo, al atributo radio). En sentencias posteriores f seguirá siendo de tipo Figura.

Aplicando este concepto al código que ya hemos generado anteriormente.

PROBLEMA: Acceso a los métodos hija1() y hija2() tras aplicar polimorfismo.

```
ClasePadre cp = new ClasePadre(0);
ClaseHija1 ch1 = new ClaseHija1(0, 1);
ClaseHija2 ch2 = new ClaseHija2(0, 2);

System.out.println(cp);
System.out.println(ch1);
System.out.println(ch2);

cp.hola();
ch1.hola();
ch1.hija1();
ch2.hola();
ch2.hija2();

System.out.println("*****************");

ClasePadre polcp = new ClasePadre(0);
ClasePadre polch1 = new ClaseHija1(0, 1);
ClasePadre polch2 = new ClaseHija2(0, 2);

System.out.println(cp);
System.out.println(ch1);
System.out.println(ch2);

polcp.hola();
polch1.hola();
polch1.hija1(); //No se puede acceder a este método pues la clase padre no lo tiene.
polch2.hola();
polch2.hija2(); //No se puede acceder a este método pues la clase padre no lo tiene.
```

SOLUCIÓN: Casting objeto.

```
ClaseHija1 castCh1 = (ClaseHija1)polch1;
castCh1.hija1();
((ClaseHija1)polch1).hija1();

ClaseHija2 castCh2 = (ClaseHija2)polch2;
castCh2.hija2();
((ClaseHija2)polch2).hija2();
```

Otro ejemplo utilizando en lugar de polimorfismo con la clase padre sino incluso una clase jerárquica superior y la más genérica como es Object, sucedería lo siguiente:

```
Object Ocp = new ClasePadre(0);
Object Och1 = new ClaseHija1(0, 1);
Object Och2 = new ClaseHija2(0, 2);

Och1.hija1();
ClaseHija1 Och2toClaseHija1 = (ClaseHija1)Och1;
Och2toClaseHija1.hija1();

((ClaseHija1)Och1).hija1();
```

# 7. CLASE / MÉTODOS GENÉRICOS O PARAMETRIZADOS.

En su esencia, el término genéricos significa **tipos parametrizados**. Los tipos parametrizados son importantes porque le permiten **crear clases, interfaces y métodos en los que el tipo de datos sobre los que operan se especifica como parámetro**. Una clase, interfaz o método que funciona con un tipo de parámetro se denomina genérico, como una **clase genérica** o **método genérico**.

## 7.1. EJEMPLO CLASE GENÉRICA SIMPLE:

```
public class ClaseGenerica1Tipo<T>{

    T atributo; // Se define un atributo pero sin especificar el tipo.

    // Constructor
    public ClaseGenerica1Tipo(T atributo) {
        super();
        this.atributo = atributo;
    }

    // Getters & Setters
    public T getAtributo() {
        return atributo;
    }

    public void setAtributo(T atributo) {
        this.atributo = atributo;
    }

}
```

```
// Usar Clase generica de 1 parámetro
ClaseGenerica1Tipo<Integer> objetoInteger = new ClaseGenerica1Tipo<Integer>(2);
ClaseGenerica1Tipo<Double> objetoDouble = new ClaseGenerica1Tipo<Double>(2.2);
ClaseGenerica1Tipo<String> objetoString = new ClaseGenerica1Tipo<String>("Dos");
```

Donde podemos ver como se define la clase genérica:

```
public class ClaseGenerica<T>{
```

T es el nombre de un parámetro de tipo y se especifica con los corchetes <>

Con la T el usuario puede indicar que tipo de parámetro utilizará en lugar dicha T. Por lo tanto, T se reemplazará por el tipo utilizado a la hora de definir e instanciar el objeto.

Recomendaciones:

- Una única letra por parámetro y este con una letra mayúsculas.
- Es común/tradicional utilizar la letra T y posteriormente V o E en caso de querer definir más de un parámetro.

## 7.2. EJEMPLO CLASE GENÉRICA MÚLTIPLE:

```
public class ClaseGenerica3Tipo<T,V,E> {

    private T attr1;
    private V attr2;
    private E attr3;

    public ClaseGenerica3Tipo(T attr1, V attr2, E attr3) {
        this.attr1 = attr1;
        this.attr2 = attr2;
        this.attr3 = attr3;
    }

    public T getAttr1() {
        return attr1;
    }

    public void setAttr1(T attr1) {
        this.attr1 = attr1;
    }

    public V getAttr2() {
        return attr2;
    }

    public void setAttr2(V attr2) {
        this.attr2 = attr2;
    }

    public E getAttr3() {
        return attr3;
    }

    public void setAttr3(E attr3) {
        this.attr3 = attr3;
    }

}
```

```
// Usar Clase generica de 3 parámetros
ClaseGenerica3Tipo<Integer, Integer, Double> objetoIntIntDouble = new ClaseGenerica3Tipo<Integer, Integer, Double>(2, 2, 2.2);
```

## 7.2. EJEMPLO CLASE GENÉRICA MÚLTIPLE:

## 7.3. EJEMPLO MÉTODO GENÉRICO:

```
public class MetodoGenerico {

    public <T,E> void mostrarVariables(T num1, E num2) {

        System.out.println(num1);
        System.out.println(num2);

    }

}
```

```
MetodoGenerico metodoGenerico = new MetodoGenerico();
metodoGenerico.mostrarVariables(2, 1);
```

## 7.4. TIPOS LIMITADOS:

En los ejemplos anteriores, los parámetros de tipo podrían reemplazarse por cualquier tipo de clase. Esto está bien para muchos propósitos, pero a veces es útil limitar los tipos que se pueden pasar a un parámetro de tipo

Java proporciona **tipos limitados** (bounded types). Al especificar un parámetro de tipo, puede crear un límite superior que declara la **superclase** de la cual se derivan todos los argumentos de tipo. Esto se logra mediante el uso de una cláusula **extends** al especificar el parámetro de tipo, como se muestra aquí:

```
<T extends superclass>
```

Si queremos limitar que sea un número, puesto que Integer, Double… derivan de la clase “Number” podríamos limitar el tipo admitido de este modo:

```
public class ClaseGenericaSingleBound<T extends Number> {
```

Incluso permitir que el tipo sea una de las siguientes clases

### 7.4.1 Compatiblidad de tipo

Los tipos delimitados son especialmente útiles cuando necesita asegurarse de que un parámetro de tipo sea compatible con otro

```
class Pareja<T,V extends T>{
```

De este modo tanto el tipo T como V tienen que ser del mismo tipo.