PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TEMA 1

PROGRAMACIÓN, JAVA. PARTE I

Contenido

1. Tipos de datos primitivos en Java

Los tipos primitivos de Java son tipos básicos, representan valores simples y se encuentran definidos por el lenguaje Java. Esto quiere decir que para cada tipo de dato básico existe un tipo equivalente en Java nombrado con una palabra, estos nombres se denominan "palabras reservadas" del lenguaje y no puedes ser utilizadas por el programador cuando cree nombres de variables. Los tipos primitivos en Java son los siguientes:

- **Numéricos enteros**
  - **Byte**: representa un tipo de dato de 8 bits con signo.
  - **Short**: representa un tipo de dato de 16 bits con signo.
  - **Int**: es un tipo de dato de 32 bits con signo para almacenar valores numéricos.
  - **Long**: es un tipo de dato de 64 bits con signo que almacena valores numéricos entre -263 a 263-1.
- **Numéricos reales:**
  - **Float**: es un tipo dato para almacenar números en coma flotante con precisión simple de 32 bits.
  - **Double**: es un tipo de dato para almacenar números en coma flotante con doble precisión de 64 bits.

- **Booleano → Boolean**: sirve para definir tipos de datos booleanos. Es decir, aquellos que tienen un valor de true o false. Ocupa 1 bit de información.
- **Carácter → Char**: es un tipo de datos que representa a un carácter Unicode sencillo de 16 bits.

Lista de tipos de datos primitivos del lenguaje Java			
Tipo	*Tamaño*	*Valor mínimo*	*Valor máximo*
`byte`	8 bits	-128	127
`short`	16 bits	-32768	32767
`int`	32 bits	-2147483648	2147483647
`long`	64 bits	-9223372036854775808	9223372036854775807
`float`	32 bits	-3.402823e38	3.402823e38
`double`	64 bits	-1.79769313486232e308	1.79769313486232e308
`char`	16 bits	`'\u0000'`	`'\uffff'`

Nota: un dato de tipo carácter se puede escribir entre comillas simples, por ejemplo `'a'`, o también indicando su valor Unicode, por ejemplo `'\u0061'`.

1.1 Valores por defecto de los datos primitivos y otros datos

Dato Primitivo	Valor por Defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'u0000'
String (o cualquier objeto)	null
boolean	false

1.2 Clases de Java relacionadas con los datos primitivos

El paquete Java.lang es uno de los paquetes más importantes de Java, ya que incluye muchas clases relevantes que forman parte del eje básico del lenguaje Java. No es necesario importar el paquete explícitamente, ya que siempre está incluido en cualquier aplicación. Clases del paquete Java.lang:

- **System**: proporciona objetos para entrada/salida, y funcionalidades del sistema.
- **Math**: proporciona funciones matemáticas usuales.
- **Envoltorios de tipos primitivos** (Character, Boolean, Integer, Double, etc.)
- **Object**: la raíz de la jerarquía de clases. Define el comportamiento base de todos los objetos.
- **String y StringBuilder**: manipulación de cadenas de caracteres.
- **Class**: proporciona información sobre las clases.

El paquete Java.lang, además:

- Contiene interfaces:
  - **Comparable**: especifica el método de ordenación natural entre objetos (compareTo()).
  - **Cloneable**: especifica que un objeto puede ser clonado.
  - **Runnable**: especifica el método a ejecutar por el gestor de hebras (run()).
- Contiene también **excepciones y errores**.

1.3 Palabras reservadas de Java

- **abstract**: Especifica la clase o método que se va a implementar más tarde en una subclase.
- **boolean**: Tipo de dato que sólo puede tomar los valores verdadero o falso.
- **break**: Sentencia de control para salirse de los bucles.
- **byte**: Tipo de dato que soporta valores en 8 bits.
- **byvalue**: Reservada para uso futuro.
- **case**: Se utiliza en las sentencias switch para indicar bloques de texto.
- **cast**: Reservada para uso futuro.
- **catch**: Captura las excepciones generadas por las sentencias try.
- **char**: Tipo de dato que puede soportar caracteres Unicode sin signo en 16 bits.
- **class**: Declara una clase nueva.
- **const**: Reservada para uso futuro.
- **continue**: Devuelve el control a la salida de un bucle.
- **default**: Indica el bloque de código por defecto en una sentencia switch.
- **do**: Inicia un bucle do-while.
- **double**: Tipo de dato que soporta números en coma flotante, 64 bits.
- **else**: Indica la opción alternativa en una sentencia if.
- **extends**: Indica que una clase es derivada de otra o de una interfaz.

- final: Indica que una variable soporta un valor constante o que un método no se sobrescribirá.
- **finally**: Indica un bloque de código en una estructura try – catch que siempre se ejecutará.
- **float**: Tipo de dato que soporta un número en coma flotante en 32 bits.
- **for**: Utilizado para iniciar un bucle for.
- **future**: Reservada para uso futuro.
- **generic**: Reservada para uso futuro.
- **goto**: Reservada para uso futuro.
- **if**: Evalúa si una expresión es verdadera o falsa y la dirige adecuadamente.
- **implements**: Especifica que una clase implementa una interfaz.
- **import**: Referencia a otras clases.
- **inner**: Reservada para uso futuro.
- **instanceof**: Indica si un objeto es una instancia de una clase específica o implementa una interfaz específica.
- **int**: Tipo de dato que puede soportar un entero con signo de 32 bits.
- **interface**: Declara una interfaz.
- **long**: Tipo de dato que soporta un entero de 64 bits.
- **native**: Especifica que un método está implementado con código nativo (específico de la plataforma).
- **new**: Crea objetos nuevos.
- **null**: Indica que una referencia no se refiere a nada.
- **operator**: Reservado para uso futuro. .

- **outer**: Reservado para uso futuro.
- **package**: Declara un paquete Java.
- **private**: Especificador de acceso que indica que un método o variable sólo puede ser accesible desde la clase en la que está declarado.
- **protected**: Especificador de acceso que indica que un método o variable sólo puede ser accesible desde la clase en la que está declarado (o una subclase de la clase en la que está declarada u otras clases del mismo paquete).
- **public**: Especificador de acceso utilizado para clases, interfaces, métodos y variables que indican que un tema es accesible desde la aplicación (o desde donde la clase defina que es accesible).
- **rest**: Reservada para uso futuro.
- **return**: Envía control y posiblemente devuelve un valor desde el método que fue invocado.
- **short**: Tipo de dato que puede soportar un entero de 16 bits.
- **static**: Indica que una variable o método sono de una clase (más que estar limitado a un objeto particular).
- **super**: Se refiere a una clase base de la clase (utilizado en un método o constructor de clase).
- **switch**: Sentencia que ejecuta código basándose en un valor.
- **synchronized**: Especifica secciones o métodos críticos de código multihilo.

- **this**: Se refiere al objeto actual en un método o constructor
- **throw**: Crea una excepción.
- **throws**: Indica qué excepciones puede proporcionar un método,
- **transient**: Especifica que una variable no es parte del estado persistente de un objeto.
- **try**: Inicia un bloque de código que es comprobado para las excepciones.
- **var**: Reservado para uso futuro.
- **void**: Especifica que un método no devuelve ningún valor.
- **volatile**: Indica que una variable puede cambiar de forma asíncrona.
- **while**: Inicia un bucle while.
- **//**: Comentario de línea.
- **/***: Apertura de comentario de bloque.
- ***/**: Cierre de comentario de bloque.

1.4 Creación de variables y constantes de datos primitivos

Variables

```
tipo identificador;
tipo identificador = expresión;
tipo identificador1, identificador2 = expresión;
```

Constantes

Declaración: `final tipo IDENTIFICADOR = expresión;`

Notas: las constantes, también denominadas literales, son "variables" de valor fijo. Una vez dado un valor inicial, este no se puede cambiar posteriormente. Su contenido puede ser de cualquier longitud y tipo único.

Operadores

Los operadores son símbolos que tienen asociada una operación, se emplean en expresiones para operar los elementos que relacionan. Existen varios tipos de operadores. A continuación se muestran los diferentes tipos existentes:

OPERADORES JAVA						
ARITMÉTICOS	RELACIONALES	LÓGICOS	UNITARIOS	A NIVEL DE BITS	ASIGNACIÓN	CONDICIONAL
+	<	&&	+	&	=	? :
-	<=	||	-	|	+=	
*	>	!	++	^	-=	
/	>=		--	<<	*=	
%	!=		~	>>	/=	
	==		!	>>>	%=	
					<<=	
					>>=	
					>>>=	
					&=	
					|=	
					^=	

Ejemplos sobre el uso de operadores:

int a = 10, b = 3;

double v1 = 12.5, v2 = 2.0;

char c1='P', c2='T';

Operación	Valor	Operación	Valor	Operación	Valor
a+b	13	v1+v2	14.5	c1	80
a-b	7	v1-v2	10.5	c1 + c2	164
a*b	30	v1*v2	25.0	c1 + c2 + 5	169
a/b	3	v1/v2	6.25	c1 + c2 + '5'	217
a%b	1	v1%v2	0.5		

int I = 7;

double f = 5.5;

char c = 'w';

byte b = 1;

Operación	Valor	Tipo
i + f	12.5	double
i + c	126	int
i + c - '0'	78	int
(i + c) - (2 * f / 5)	123.8	double
b + c	120	int

```
int a = 7, b = 9, c = 7;
```

Operación	Resultado
a==b	false
a >=c	true
b < c	false
a != c	false

Tablas de verdad para los operadores lógicos:

A	B	A OR B
F	F	F
F	V	V
V	F	V
V	V	V

A	B	A AND B
F	F	F
F	V	F
V	F	F
V	V	V

A	NOT A
F	V
V	F

```
int i = 7;
float f = 5.5F;
char c = 'w';
```

	Resultado
i >= 6 && c != 'w'	false
i >= 6 || c != 'w'	true
f < 10 && i > 100	false
!(c != 'p') || i % 2 == 0	false
i + f <= 10	false
i >= 6 && c == 'w' && f == 5	false
c != 'p' || i + f <= 10	true

2. Casting

La conversión de tipos se denomina casting, e incluye las transformaciones de tipos de datos primitivos, numéricos y no numéricos, y clases. A continuación, se detallan los tipos de casting diferentes que existen. Es importante resaltar que para que se produzca una conversión de tipos estos deben de ser compatibles.

2.1 Casting de tipos numéricos primitivos

- Implícito: ocurre cuando el tipo de dato de origen es menor que el tipo de dato de destino. La conversión se realiza automáticamente por el compilador. Ejemplos de casting implícito:

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

  - Promoción de tipos de datos básicos numéricos: el casting implícito permite transformar un dato de un tipo a otro, e incluye la asignación de datos y los operadores aplicados, que se denomina promoción de datos básicos numéricos. Ejemplo de promoción:

```java
char char1 = 1, char2 = 2;
short short1 = 1, short2 = 2;
int int1 = 1, int2 = 2;
float float1 = 1.0f, float2 = 2.0f;

// char1 = char1 + char2;      // Error: Cannot convert from int to char;
// short1 = short1 + short2;   // Error: Cannot convert from int to short;
int1 = char1 + char2;          // char is promoted to int.
int1 = short1 + short2;        // short is promoted to int.
int1 = char1 + short2;         // both char and short promoted to int.
float1 = short1 + float2;      // short is promoted to float.
int1 = int1 + int2;            // int is unchanged.
```

- Explícito: se produce cuando el tipo de dato de origen es mayor que el tipo de dato de destino. Para que se efectúe el casting se debe dejar escrito en el código el tipo de destino a representar. La conversión explicita de tipos de datos en coma flotante (double y float) a tipos de datos de números enteros primitivos produce un redondeo a la baja. Ejemplos de casting explicito:

```java
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

- Anotaciones para el casting de tipos de datos primitivos no numéricos:
    - El tipo de dato primitivo booleano no puede ser convertido a ningún otro tipo y viceversa.
    - Un **char** se puede convertir hacia/desde cualquier tipo numérico utilizando las asignaciones de puntos de código especificadas por Unicode. Un **char** es representado en la memoria como un valor entero de

16 bits sin signo (2 bytes), por lo que la conversión a **byte** (1 byte) eliminará 8 de esos bits (esto es seguro para caracteres ASCII). Los métodos de utilidad de la **clase Character** usan **int** (4 bytes) para transferir hacia/desde valores de puntos de código, pero un **short** (2 bytes) también sería suficiente para almacenar un punto de código Unicode. Ejemplos:

```
int badInt   = (int)  true; // Compiler error: incompatible types
```

```
char char1   = (char)   65; // A
byte byte1   = (byte)  'A'; // 65
short short1 = (short) 'A'; // 65
int int1     = (int)   'A'; // 65

char char2   = (char) 8253; // ‽
byte byte2   = (byte)  '‽'; // 61 (truncated code-point into the ASCII range)
short short2 = (short) '‽'; // 8253
int int2     = (int)   '‽'; // 8253
```

2.2 Casting de objetos

Al igual que con las primitivas, los objetos se pueden convertir tanto explícita como implícitamente.

La conversión implícita, también llamada upcasting o conversión ascendente, ocurre cuando el tipo de origen extiende o implementa el tipo de destino (conversión a una superclase o interface).

La conversión explícita, también llamada downcasting o conversión descendente, debe realizarse cuando el tipo de destino extiende o implementa el tipo de origen (transmisión a

un subtipo). Esto puede producir una excepción de tiempo de ejecución (**ClassCastException**) cuando el objeto que se está convirtiendo no es del mismo tipo de objetivo (o el subtipo del objetivo).

Ejemplos de conversión de objetos:

```
Float floatVar = new Float(42.0f);
Number n = floatVar;                //Implicit (Float implements Number)
Float floatVar2 = (Float) n;        //Explicit
Double doubleVar = (Double) n;      //Throws exception (the object is not Double)
```

2.3 Instanceof

Java proporciona el operador instanceof para probar si un objeto es de un determinado tipo o una subclase. El programa puede entonces elegir convertir o no ese objeto en consecuencia.

```
Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.
}
```

3. Caracteres de escape

Es un carácter precedido por una barra invertida (\) y tienen un significado especial para el compilador.

Secuencias de escape

Secuencia de escape	Descripción
\t	Inserte una pestaña en el texto en este punto.
\b	Inserte un retroceso en el texto en este punto.
\n	Inserte una nueva línea en el texto en este punto.
\r	Inserte un retorno de carro en el texto en este punto.
\f	Inserte un avance de formulario en el texto en este punto.
\'	Inserte una comilla simple en el texto en este punto.
\"	Inserte una comilla doble en el texto en este punto.
\\	Inserte un carácter de barra invertida en el texto en este punto.

4. Clases

Una clase es un patrón o plantilla que define un tipo complejo y responde a las necesidades del programa para manejar el dato complejo resultante del modelado. Los objetos son instancias de clases.

4.1 Sintaxis general de una clase en Java

```
[modificadorDeAcceso] class NombreClase [extends NombreSuperClase][implements Interface1, Interface2, … ]{

      //atributos de la clase (0 ó más atributos)
      [modificadorDeAcceso]  tipo nombreAtributo;


      //métodos de la clase (0 ó más métodos)
      [modificadorDeAcceso] tipoDevuelto nombreMetodo([lista parámetros]) [throws listaExcepciones]{
                  // instrucciones del método
                  [return valor;]
      }
}
```

Nota: Lo que aparece entre corchetes es opcional.

Detalle de la sintaxis:

Modificadores de acceso

Sirven para indicar el nivel de visibilidad de los miembros de la clase (atributos y métodos). Existen los siguientes tipos:

MODIFICADOR DE ACCESO	EFECTO	APLICABLE A
private	Restringe la visibilidad al **interior de la clase**. Un atributo o método definido como private solo puede ser usado en el interior de su propia clase.	Atributos Métodos
<Sin modificador>	Cuando no se especifica un modificador, el elemento adquiere el *acceso por defecto o friendly*. También se le conoce como acceso de package (paquete). Solo puede ser usado por las **clases dentro de su mismo paquete**.	Clases Atributos Métodos
protected	Se emplea en la herencia. El elemento puede ser utilizado por **cualquier clase dentro de su paquete y por cualquier subclase** independientemente del paquete donde se encuentre.	Atributos Métodos
public	Es el nivel máximo de visibilidad. El elemento es visible desde cualquier clase.	Clases Atributos Métodos

Otros modificadores

Unido a los modificadores de acceso pueden aparecer otros modificadores que se aplican a las clases, los atributos y los métodos.

MODIFICADOR	EFECTO	APLICABLE A
abstract	Aplicado a una clase, la declara como clase abstracta. No se pueden crear objetos de una clase abstracta. Solo pueden usarse como superclases. Aplicado a un método, la definición del método se hace en las subclases.	Clases Métodos
final	Aplicado a una clase significa que no se puede extender (heredar), es decir que no puede tener subclases. Aplicado a un método significa que no puede ser sobrescrito en las subclases. Aplicado a un atributo significa que contiene un valor constante que no se puede modificar	Clases Atributos Métodos
static	Aplicado a un atributo indica que es una variable de clase. Esta variable es única y compartida por todos los objetos de la clase. Aplicado a un método indica que se puede invocar sin crear ningún objeto de su clase.	Atributos Métodos
volatile	Un atributo volatile puede ser modificado por métodos no sincronizados en un entorno multihilo.	Atributos
transient	Un atributo transient no es parte del estado persistente de las instancias.	Atributos
Sincronizad	Métodos para entornos multihilo.	Métodos

Extends

Permite heredar de la clase "NombreSuperClase". Todas las clases heredan implicitamente de la superclase **Object** de Java**.**

Implements

Indica que la clase codifica los interfaces que recibe en la definición de la clase. Una interface es un conjunto de constantes y declaraciones de métodos. La clase receptora debe implementar todos los métodos de la interface.

Nota: al conjunto de métodos de una clase se le denomina interfaz de la clase.

Atributos

Permiten almacenar los datos relevantes del objeto. Generalmente, se definen al principio de la clase y se hace de la misma forma que una variable local en un método. *Pueden tener modificadores de acceso* y ser de tipo primitivo o referencias a objetos. El valor del atributo puede ser asignado en la misma línea de creación, aunque esto se hace en el constructor de la clase. Existen dos tipos de atributos, o variables:

- Atributos de instancia: no son declarados static y pertenecen a la instancia del objeto, es decir; son locales al objeto.

- Atributos de clase: son declarados static y pertenecen a la clase, es decir son globales y compartidos por todas las instancias de objeto. La notación de acceso es → Nombreclase.Atributo si el atributo es public, si el atributo es private el acceos se hace a través del método correspondiente.

Nota: no es necesario que existan objetos de la clase para acceder a los atributos estáticos.

Métodos

Establecen el comportamiento de los objetos de la clase y a través de ellos se accede a los datos de la clase. Representan el conjunto de mensajes a los que la clase puede responder, por ello es básico la implementación de, al menos, los métodos de acceso a la clase; también llamados getters/setters. Los métodos también pueden ser:

- Métodos de instancia: no declarados static. Operan sobre las variables del objeto y sobre las variables estáticas.
- Métodos de clase: declarados como static. Solo tienen acceso a los atributos estáticos de la clase.

Nota: la llamada a los métodos, en ambos casos, es igual. No es necesario que exista un objeto de la clase para llamar a los métodos estáticos.

Tipos de métodos

Constructores

Es un método especial para inicializar la instancia del objeto y asignar valores iniciales a sus variables. Los constructores no devuelven ningún valor. Tipos de constructores:

- Por defecto o predeterminado: es proporcionado automáticamente por Java, en caso de que no se defina explícitamente ningún constructor en la clase. Asigna los valores iniciales predeterminados a los miembros de datos

de la clase. El programador puede crear un constructor vacío, lo cual es altamente recomendable.

- Parametrizado: incluye parámetros en su signatura que recibirán los valores con los que se inicializarán los miembros de datos de la clase. El programador debe crear un constructor parametrizado, no es proporcionado automáticamente por Java. Se pueden crear tantos constructores en una clase como se desee, con una justificación de uso.

Nota importante: Hay que prestar especial atención a los constructores, el código de inicialización incluido, cuando tenemos miembros de datos que son referencias o estructuras complejas.

Nota: se denomina signatura a la cabecera de un método, de manera que dos métodos con el mismo nombre y distintos parámetros tienen distinta signatura. También recibe le nombre de firma.

Getters/setters

Permiten acceder a los valores o introducir nuevos valores respecto de los miembros de datos privados de la clase.

¿ Por qué los miembros de datos son privados ?

La naturaleza de la POO habla de **encapsulación** y **protección** de la información. Los datos privados mantienen la naturaleza de protección de los datos y los métodos permiten

interactuar con el objeto: evitando los accesos no autorizados, como dicta la naturaleza de la POO.

Existe otra razón para la existencia de los datos privados: la **extensibilidad**. Otro rasgo de la POO es la reutilización de código, la reusabilidad. Por ejemplo, si queremos sincronizar el acceso y la modificación de un miembro de datos, la forma de realizarlo es la siguiente:

- Datos privados:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

- Datos no privados:

Hay que controlar el acceso/modificación allí donde se produzca, lo cual resulta casi imposible. Si la implementación está en una librería consumida por terceros no existe la posibilidad de controlar el acceso y la modificación de los datos. En el caso de una API queda comprometido el mantenimiento de la implementación.

4.2. Referencias de tipos de datos

Referenciar un tipo de dato significa que el dato tiene asignado un espacio de memoria propio y una dirección que apunta a ese espacio, la dirección suele ser manejada por un nombre. La gestión interna habitual de las referencias es realizada por Java, pero la instanciación de la referencia de tipo debe ser hecha explícitamente por el programador. Los datos primitivos no se referencian, Java se encarga de ello. ¿ Por qué ? …

Ejemplo de instanciación de un tipo de datos:

```
Object obj = new Object(); // Note the 'new' keyword
```

Ejemplo de desreferencia de tipos de datos:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

La desreferenciación sigue la dirección de memoria almacenada en una referencia, hasta el lugar en la memoria donde se encuentra el objeto real. Cuando se encuentra un objeto, se llama al método solicitado y realmente asociado (toString en este caso).

Ejemplo de objeto no instanciado:

Un objeto que no ha sido instanciado no contiene una referencia de memoria, su referencia es nula (null). Esto provocará errores en tiempo de ejecución. Comentar alcance ….

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

4.3 Sobrecarga de métodos

Pueden existir diferentes versiones de un mismo método dentro de una clase, siempre que sean diferentes (número de parámetros y/o sus tipos, valores devueltos). Cuando tenemos una clase con versiones de un mismo método se denomina sobrecarga, Overloading o polimorfimo en tiempo de compilación.

5. Herencia

La herencia es una característica básica orientada a objetos en la que una clase *adquiere y extiende* las propiedades de otra clase, usando la palabra clave **extends**.

```
//Clase Alumno. Clase derivada de Persona
public class Alumno extends Persona{
    private String curso;
    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    @Override  //indica que se modifica un método heredado
    public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer(); //se llama al método leer de Persona para leer nif y nombre
        System.out.print("Curso: ");
        curso = sc.nextLine(); //se lee el curso
    }
}
```

El uso de la palabra clave **extends** entre clases hace que todas las propiedades de la superclase (también conocida como padre, Clase Padre, Clase Base) están presentes en la subclase (también conocida como Clase Secundaria o Clase Derivada).

Las modificaciones, ampliaciones que se hagan en una clase derivada sólo afectan a esa clase. No afectan a la clase padre u otras clases derivadas.

Java no permite la herencia múltiple. Una clase derivada puede contener múltiples ascendientes, solo uno por nivel, hasta llegar a la clase Object, que es la clase raíz de las clases en Java. Existe una simulación de herencia múltiple con clase e interfaces.

Redefinir atributos de la clase base en la clase derivada

Una clase derivada puede volver a declarar un atributo heredado (declarado como pubic o protected en la clase padre). Si el atributo es vuelto a declarar, el atributo de la clase padre queda oculto. Para acceder a un atributo de la clase padre, empleamos la siguiente notación: super.atributo.

Redefinir métodos de la clase base en la clase derivada

Esto quiere decir que vamos a modificar, ampliar el mismo método existente en la clase base, es decir debe la clase derivada tiene su propia implementación del método. Estamos redefiniendo el método. La redefinición conlleva una directiva del compilador en el código: "@Override", situada antes de la cabecera del método.

La redefinición sólo puede ser menos restrictiva. El método redefinido de la clase base y todas las sobrecargas del mismo quedan ocultos en la clase derivada, para acceder a ellos podemos emplear la notación super.metodo(). La redefinición se conoce como Overrriding o polimorfismo en tiempo de ejecución.

Se puede sobrecargar el método main, siempre que definamos correctamente los parámetros de entrada.

Ejemplo de sobrecarga del método "main".

```
class Demo{

    public static void main(String[] args) {
        System.out.println("Hello Folks"); // Hello Folks
        Demo.main("Ducks");
    }

    // Sobrecargando
    public static void main(String arg1) {
        System.out.println("Hello, " + arg1); // Hello Ducks
        Demo.main("Dogs","Cats");
    }
    public static void main(String arg1, String arg2) {
        System.out.println("Hello, " + arg1 + " and "  + arg2); // Hello Dogs and Cats
    }
}
```

Si queremos proteger de redefinición posterior a un método de una clase, empleamos el modificador final sobre dicho método.

```
public final void metodo(){
   ........
}
```

Ejemplo, resumen:

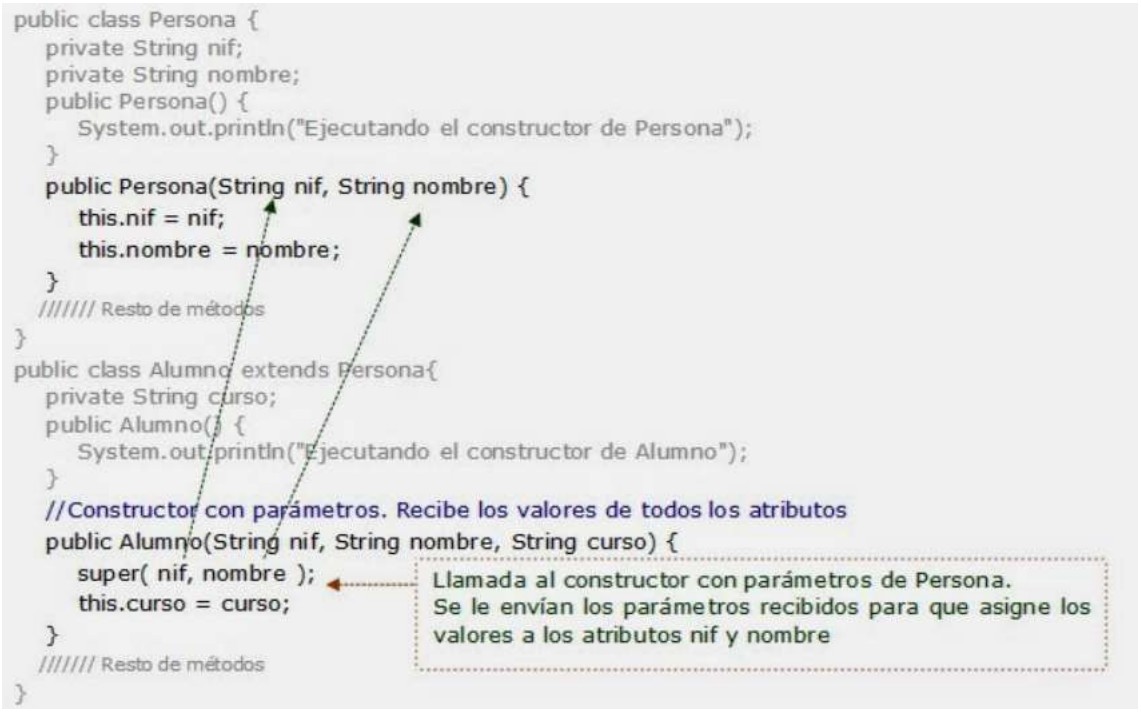
```
//Clase Alumno. Clase derivada de Persona
public class Alumno extends Persona{
    private String curso;
    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    @Override  //indica que se modifica un método heredado
    public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer(); //se llama al método leer de Persona para leer nif y nombre
        System.out.print("Curso: ");
        curso = sc.nextLine(); //se lee el curso
    }
}
```

5.1 Herencia y constructores

Características:

- Los constructores no se heredan.
- La clase base y la clase derivada inicializan sus propios atributos.
- La creación de un objeto de una clase derivada implica la ejecución del constructor de la clase base y después la ejecución del constructor de la clase derivada. No es necesario escribir la llamada al constructor padre, se llama implícitamente como primera instrucción en el constructor de la clase derivada.
- La llamada al constructor de la clase base, desde la clase derivada, se hace con la instrucción: super().
- Los constructores parametrizados deben recoger los atributos respectivos y pasarlo entre la clase base y la clase derivada. El orden correcto es: clase base y clase derivada.

La siguiente imagen muestra un resumen de las características.



5.2 Clase final

Si queremos que una clase no sea heredable debemos declararla con el modificador **final**.

```
public final class A{
        ........
}
```

5.3 Clases abstractas

Son clases que no se pueden instanciar. El sentido de su diseño es que otras clases hereden de ella. Son clases que normalmente son la raíz de la jerarquía de clases a desarrollar y tienen definido el comportamiento de todas las subclases. Las clases derivadas de la clase abstracta son las que contienen la implementación y deben hacerlo con todos los métodos abstractos heredados. Las clases abstractas:

- Pueden contener cero o más métodos abstractos.
- Pueden contener métodos no abstractos.
- Pueden contener atributos.
- Pueden contener constructores para inicializar sus atributos. Aunque no se instancien, las clases derivadas serán las que invoquen al constructor cuando sean instanciadas.
- Si una clase derivada no implementa algún método abstracto heredado, se convierte en una clase abstracta y debe ser declarada como tal.

La sintaxis para crear una clase abstracta es la siguiente:

```
[modificador] abstract class nombreClase{
    .......
}
```

Notas:

- Cuando manejamos un objeto a través de una referencia a una superclase, solo se pueden llamar a métodos de la superclase.
- Para objetos manejados con referencias: si el método llamado está en la superclase y ha sido redefinido en la subclase, entonces se ejecuta el método de la subclase. Es decir, se ejecuta el método del objeto no el de la referencia.
- El modelado de una jerarquía de objetos es fundamental para la posterior implementación. Todos los métodos por desarrollar en la posterior jerarquía de objetos deberán aparecer en la clase abstracta, raíz de dicha jerarquía.

5.4 Relaciones

Una relación representa una dependencia entre dos o mas clases, o una clase hacia si misma; denominada dependencia o relación reflexiva.

Propiedades de las relaciones

- Multiplicidad. Es decir, el número de elementos de una clase que participan en una relación. Se puede indicar un número en los diagramas UML, un rango… Se utiliza n o * para identificar un número cualquiera.
- Nombre de la asociación. En los diagramas UML , en ocasiones se escribe una indicación de la asociación que ayuda a entender la relación que tienen dos clases. Suelen utilizarse verbos como por ejemplo: «Una empresa contrata a n empleados». Java implementa las relaciones de diversas formas y no siempre tienen nombre.

Tipos de relaciones:

- **Asociación**: representa una dependencia semántica. En los diagramas UML se dibuja con una línea continua entre las clases de la asociación y pueden tener un nombre. Ejemplo: una mascota pertenece a una persona. Dos instancias de objetos en Java pueden existir independientemente la una de la otra. La relación se establece por los datos que se transmiten, los servicios que se prestan. Así dos objetos pueden tener varias relaciones y cada una de ellas se identifica por su rol. Java implementa una asociación cuando
  - Un objeto de una clase utiliza un método o propiedad de otra clase.
  - Un objeto de una clase crea otro objeto de otra clase.
  - Un método de un objeto de una clase recibe parámetros de objetos de otra clase.

- **Agregación**: representa una relación jerárquica de una clase y otras clases que la componen. En los diagramas UML se dibuja con una línea con un rombo en la parte de la clase compuesta. Ejemplo: una mesa está formada por tablas y tornillos. Existen agregaciones de objetos. Son relaciones de la forma todo-parte (pertenece a, tiene un, es parte de). Por ejemplo, un taller mecánico tiene clientes.
- **Composición**: es parecida a la agregación, pero aquí la relación es más fuerte; ya que una clase "parte" no existe sin la clase compuesta. Otra característica es que las clases "parte" no se comparten en otras relaciones. El tiempo de vida de la clase compuesta y las clases parte son el mismo. En los diagramas UML se representan con una línea continua y un rombo relleno en el extremo que va a la clase compuesta. Ejemplo; un vuelo de una compañía aérea este compuesto por varios pasajeros. Java representa la composición como un atributo dentro de una clase, que a su vez es otra clase.
- **Dependencia**: permite representar que una clase necesita de otra clase para realizar sus funcionalidades. En los diagramas UML se dibuja con una flecha de línea discontinua desde la clase que necesita la funcionalidad hasta la otra clase que suministra. Ejemplo: gastos de pagos y pasarela de pagos. Una clase utiliza los datos que genera otra clase; los necesita para operar y no los tienen como propios.
- **Herencia**: permite reflejar que una clase hija recibe atributos y/o operaciones de otra clase padre. En los diagramas UML se dibujan con una línea continua y un triángulo hacia la clase padre. Ejemplo: animal, pez, perro, gato. Java implementa la herencia con la palabra clave "extends".

Nota: la diferencia entre agregación y composición es la contención. La representación de la composición en UML es

útil para detectar limitaciones en el diseño y evitar problemas en el posterior desarrollo del sistema.

6. Polimorfismo

El polimorfismo es una característica de la POO que trabaja sobre los métodos. Un método presta un servicio en el programa, contiene acciones y define un comportamiento deseado o requerido del objeto. La ejecución de un método trivial es inmediata, produce el efecto buscado. ¿ Es posible definir varias versiones de un mismo método ? Si lo es, ya hemos hablado del overloading o sobrecarga de métodos. Si se hereda un método ¿ lo puedo volver a implementar adaptándolo a la clase derivada ? Si, es posible y aconsejable. Se denomina overriding o redefinición de métodos. Así, la ejecución de un método depende de la llamada o del objeto que llama al método. El comportamiento del flujo del programa varia, adopta diferentes formas durante los pasos de ejecución, es decir, es polimórfico y puede responder a las necesidades de la realidad programada.

Existen dos tipos de polimorfismo:

- **Polimorfismo estático o de sobrecarga**: el overloading permite que una clase tenga varios métodos con el mismo nombre, pero con diferente número de parámetros. El compilador conoce el método invocado por el número de parámetros empleados. La sobrecarga es una característica de la POO en Java, toda clase puede sobrecargar cualquier método(s).
- **Polimorfismo dinámico o de redefinición**: el overriding permite redefinir métodos heredados de la superclase en la subclase. Las redefiniciones de la subclase

sustituyen a la implementación original de la superclase y es el código que se ejecuta. El overriding necesita de una jerarquía de objetos y herencia, todos los elementos han construido el polimorfismo dinámico.

Ventajas del polimorfismo:

- Limpieza de código.
- Mantenimiento de código.
- Generalización del codigo.
- Fácil extensibilidad del código.
- Simplificación de código.
- Ocultación de detalles.

Cuadro resumen sobre las diferencias entre herencia y polimorfismo

Herencia	Polimorfismo
La herencia es aquella en la que se crea una nueva clase (clase derivada) que hereda las características de la clase ya existente (clase base).	Mientras que el polimorfismo es aquello que se puede definir en múltiples formas.
Básicamente se aplica a las clases.	Mientras que básicamente se aplica a funciones o métodos.
La herencia respalda el concepto de reutilización y reduce la longitud del código en la programación orientada a objetos.	El polimorfismo permite que el objeto decida qué forma de función implementar en tiempo de compilación (sobrecarga) y en tiempo de ejecución (anulación).
La herencia puede ser única, híbrida, múltiple, jerárquica y multinivel.	Mientras que puede ser polimorfismo en tiempo de compilación (sobrecarga), así como polimorfismo en tiempo de ejecución (anulación).
Se utiliza en el diseño de patrones.	Si bien también se utiliza en el diseño de patrones.

Ejemplo de polimorfismo paramétrico – Overloading

```
class Polymorph {

    public int add(int a, int b){
        return a + b;
    }

    public int add(int a, int b, int c){
        return a + b + c;
    }

    public float add(float a, float b){
        return a + b;
    }

    public static void main(String... args){
        Polymorph poly = new Polymorph();
        int a = 1, b = 2, c = 3;
        float d = 1.5, e = 2.5;

        System.out.println(poly.add(a, b));
        System.out.println(poly.add(a, b, c));
        System.out.println(poly.add(d, e));
    }

}
```

Ejemplo de polimorfismo de redefinición – Overrriding

```
class Animal
{
        public void mover()
        {
                System.out.println("Los animales se mueven");
        }
        public void ladrar() {}
}

class Perro extends Animal
{
        public void mover()
        {
                System.out.println("Los perros se mueven tambien");
        }

        public void ladrar()
        {
                System.out.println("Guauuuu");
        }
}

public class PruebaPerro
{
        public static void main(String[] args)
        {
                Animal a = new Animal();
                Animal b = new Perro();

                a.mover();
                b.mover();
                b.ladrar();
        }
}
```

7. Eficiencia

La medida del software en el uso de los recursos del sistema, referido al tiempo y los recursos empleados por el software en ejecución, es la *eficiencia*. Su estudio esta incluido dentro de la calidad del software y corresponde a una disciplina de la informática.

Este apartado es una sencilla aproximación a la medida de la eficiencia por medio del cálculo del tiempo transcurrido de un programa ejecutado.

La medida del tiempo de ejecución esta directamente relacionada con los datos. Si la carga de datos es máxima, el tiempo es mayor y a la inversa. El tiempo de ejecución de un algoritmo se mide con la carga de datos, el volumen. Así aparecen tres casos posibles: carga completa (n), carga media (n/2) y carga baja. El comportamiento del algoritmo puede variar en función de los datos y así las necesidades de respuesta aceptaran una solución algorítmica u otra. El estudio de la complejidad algorítmica y la eficiencia depende del contexto.

La aproximación que vamos a ver en este apartado simplemente mide el tiempo de ejecución de un algoritmo. Utilizamos el método nanoTime() de la clase System del paquete java.lang. Capturaremos sendos valores al comienzo y al final de la aplicación. La diferencia entre los valores representa el tiempo transcurrido, que se puede expresar en milisegundos (diferencia/1000000). Vemos un ejemplo.

```
public static void main(String[] args) {
    // TODO Auto-generated method stub

    long startTime = System.nanoTime();

    Frase Frase = new Frase();
    Frase Frase2 = new Frase();
    Frase.setTexto("Programando en Java");

    System.out.println("Caracter :  " + Frase.getCaracter());
    Frase.mostrarFrase();
    System.out.println("Tiene: " + Frase.contar() + " palabras.");
    System.out.println("Tiene: " + Frase.contar(true) + " caracteres " + 'a');
    Frase2.setTexto("Programando en Java, segunda frase");
    Frase2.mostrarFrase();
    System.out.println("Tiene: " + Frase2.contar() + " palabras.");
    System.out.println("Tiene: " + Frase2.contar(true) + " caracteres " + 'a');
    System.out.println(Frase.toString());


    long endTime = System.nanoTime();
    long timeElapsed = endTime - startTime;

    System.out.println("Execution time in milliseconds: " + timeElapsed / 1000000);
}
```

Existe varias formas de obtener el tiempo de ejecución. Realizar un ejercicio probando, al menos, tres formas diferentes de obtener el tiempo de ejecución de un algoritmo.