Licenciatura en Sistemas - Orientación a Objetos II – 2022

Equipo docente: Mg. María Alejandra Vranic
Lic. Romina Mansilla
Lic. Gustavo Siciliano
Lic. Ezequiel Scordamaglia

Introducción al paradigma de objetos

Teoría (Parte 2)

Estructuras de control básicas

De decisión (condicionales)

La estructura de decisión básica en Java es la sentencia if-then. La sintaxis de la misma es la siguiente:

```
if (condición){
        // sentencias que se ejecutan si condición es verdadera
}


if (condición){
        // sentencias que se ejecutan si condición es verdadera
} else {
        // sentencias que se ejecutan si condición es falsa
}


if (condición1){
        // sentencias que se ejecutan si condición1 es verdadera
} else if (condicion2){
        // sentencias que se ejecutan si condición1 es falsa y condicion2
        // verdadera
} else if(condición3) {
        // sentencias que se ejecutan si condición1 y condicion2 son
        // falsas y condición3 verdadera

        [... tantos else if como se quiera … ]

} else {
        // sentencias que se ejecutan si todas las condiciones son falsas
}
```

Condición es cualquier expresión que devuelva un valor de tipo booleano. Puede ser tanto una comparación como un método que devuelva un booleano, como una variable que contenga un booleano.

Otra estructura condicional es switch:

```
switch (variable){
        case valor1:
                // acción1;
            break;
        case valor2:
                //accion2;
            break;

        [... tantas cláusulas case como sean necesarias …]

        default:
                // acción a ejecutar por defecto
                break;
}
```

El funcionamiento es simple: Dependiendo del valor que tome la variable, es la cláusula case que se ejecuta. la sentencia break está allí porque si no la ejecución sigue con el case siguiente. La cláusula default se ejecuta si el valor que toma la variable no se encuentra en ninguna de las cláusulas case.

De iteración

Cuando necesitamos iterar (ejecutar una serie de sentencias de manera repetitiva) Java nos ofrece una serie de estructuras de control, cada una más adecuada a una forma de iterar distinta.

while / do while

Las estructuras while y do-while se utilizan para iterar mientras se cumpla una condición. La diferencia entre ambas es que en el caso de while la condición se verifica al comienzo de cada iteración, mientra que en el do-while se verifica al final.

```
while (condición) {
    // sentencias a ejecutar
}

do {
    // sentencias a ejecutar
} while (condición);
```

for/ for-next

La última estructura de iteración se denomina for o for-next. Se utiliza generalmente para iterar por un rango de valores:

```
for (inicialización; condición_finalizacion; incremento) {
        // sentencias a ejecutar
}
```

La cláusula de inicialización asigna o declara y asigna la variable que funcionará como contador de ciclos. La condición_finalización es una condición que devuelve un booleano que si es verdadero causa la terminación del ciclo. Incremento indica la forma de incrementar el contador.

Algunos ejemplos para que quede más claro:

```
int contador;
for(contador=0; contador<10; contador++){
        // sentencias
}
```

En este caso declaramos la variable contador fuera del ciclo for y la inicializamos en el mismo ¿Cuantas veces iterará?

```
for(int contador=0; contador<10; contador++){
        // sentencias
}
```

En este caso, la variable se declara dentro del for. Podría parecer lo mismo, pero hay una diferencia fundamental: Como la sentencia for declara un bloque de código y su ámbito, la variable contador no será visible fuera del ámbito del ciclo. Es decir, si en este caso intentamos acceder a la variable contador luego de terminado el ciclo el compilador dará un error del tipo de variable no definida. En el primer caso sí se puede acceder, ya que la declaramos fuera del for.

Terminantemente prohibido:

```
for(int contador=0; contador<10; contador++){
        // sentencias
        if(alguna condición){
                break; //no!
        }
}
```

Lo mismo en el caso de los while y do while. La sentencia break termina la ejecución del ciclo de manera abrupta. Los ciclos deben terminar cuando su condición lo indique, es mucho más complicado y proclive a errores debuguear un ciclo con múltiples salidas que uno con un sólo punto de salida. Si nos vemos en la necesidad de tener que detener la ejecución de un for con un break, significa que tenemos que componer la condición de terminación con un operador lógico. En caso de ser así, preferimos utilizar un while con un contador que se incremente en el ciclo y hacer la condición compuesta. La sentencia for sólo la emplearemos para los casos en que el intervalo a iterar esté bien definido y sólo se salga cuando éste se recorrió por completo.

Lo dicho con respecto al único punto de salida también es válido para los métodos: Sólo debe haber una sentencia return en un método, y debe estar en el punto de salida, ya que return termina la ejecución del método y devuelve el valor que recibe como argumento.

REALIZAR PRÁCTICA: 2.A (Números)

Comparaciones

Para comparar igualdad de valores primitivos numéricos utilizamos ==. El signo = sólo se usa para asignar valores

```
if(paciente1.getPeso==80){
        // lo que corresponda
}
```

Los Strings dijimos que son objetos, y los objetos no pueden compararse con ==. La comparación == devuelve verdadero si los dos objetos siendo comparados son en realidad el mismo objeto. Vamos a aclarar un poco esto:

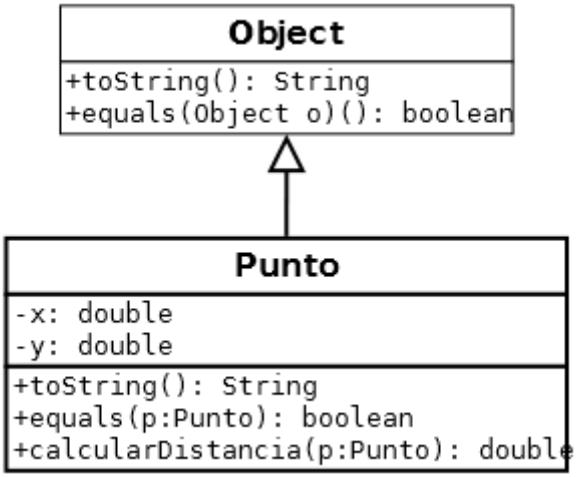
```
Paciente paciente1 = new Paciente("José", "Pérez",1.80f,85);
Paciente paciente2 = paciente1;
Paciente paciente3 = new Paciente("José", "Pérez",1.80f,85);
System.out.println(paciente1);
System.out.println(paciente2);
System.out.println(paciente3);
System.out.println(paciente1==paciente2); //imprime true
System.out.println(paciente1==paciente3); //imprime false
```

¿Porque las dos últimas impresiones no muestran ambas true? Esto ocurre porque paciente 1 y 2 tienen la misma identidad, es decir, son el mismo objeto: ocupan el mismo lugar en la memoria de la JVM. Sólo se creó el objeto una vez (con new), en paciente1 y luego se asignó a paciente2. En cambio, paciente3 fue creado, ocupa otro lugar en memoria y para java son distintos objetos, aunque sus atributos sean los mismos. Los String también son objetos y si bien la comparación entre “esto es un string”==“esto es un string” devuelve verdadero porque el compilador java internaliza strings literales iguales como un solo string (un solo objeto), fallará si utilizamos un string construido en tiempo de ejecución en la comparación.

¿Cómo solucionamos este problema?

Todas las clases definidas heredan de la clase Object en forma implícita. Por ahora solo vamos decir que cuando una clase hereda de otra “hereda” todos los atributos y métodos. Vamos a verlo con un ejemplo:

Creamos un proyecto Geometria, donde en modelo definimos la clase Punto:



Para comparar 2 objetos lo debemos hacer con el método equals. Este método siempre va a recibir como parámetro un objeto del mismo tipo de la clase que lo contiene. Pero para que equals devuelva true si las coordenadas del punto son las mismas debemos re-definir el método que hereda de la clase Object (que por defecto compara referencias).

Por otra parte si queremos que cuando imprimamos el objeto se muestre como el valor de los atributos (un par ordenado x,y) debemos re-definir el método toString de la clase Object.

```
package modelo;
public class Punto  {

	//atributos
	private double x;
	private double y;

	//constructor
	public Punto(double x, double y) {
	   this.x = x;
	   this.y = y;
	}

	//métodos getter y setter
	public double getX() {
	   return x;
	}

	public void setX(double x) {
	   this.x = x;
	}

	public double getY() {
	   return y;
	}

	public void setY(double y) {
	   this.y = y;
	}

	//re-definición de métodos de la clase Object
	// sobrecarga
```

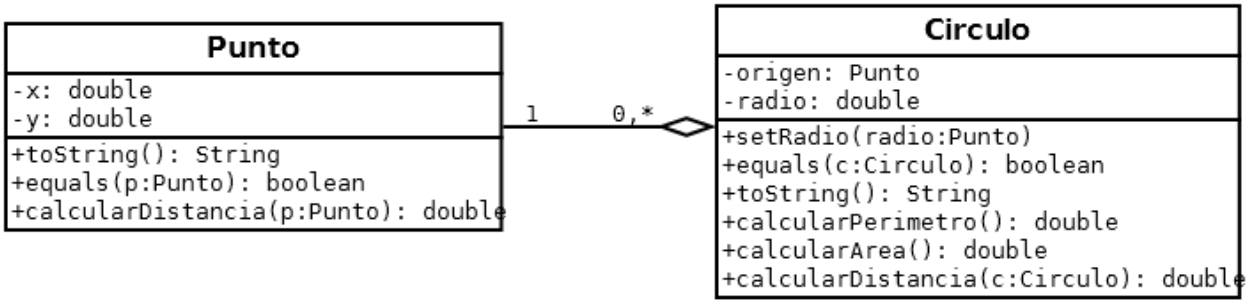
```
        public boolean equals(Punto p){
        return ((x==p.getX())&&(y==p.getY()));
        }

        // re-definición
        @Override
        public String toString(){
        return "("+x+","+y+")";
        }
}
```

Cuando hablamos de re-definir un método estamos aplicando **polimorfismo**. El método toString de la clase Object (padre) tiene la misma forma que el de Punto; es decir, el mismo nombre del método y los mismos parámetros (en este caso ninguno). Esto es lo que denominamos la firma o signatura del método. Al ser las signaturas iguales, lo que efectivamente estamos haciendo es sobreescribir el método heredado. Esto implica que el código del nuevo método reeemplazará al anterior. En el caso de equals, lo que estamos haciendo es cambiar la signatura del método de equals(Object) en la clase Object a equals(Punto) en la clase Punto. Esta forma de redefinir se denomina sobrecarga y nos permite tener dos métodos con el mismo nombre pero con distinto comportamiento dependiendo de los parámetros que pasemos. Así, si invocamos Punto.equals con un objeto de la clase Object tendremos el comportamiento estándar definido en Object. Si lo invocamos con un objeto de la clase Punto se ejecutará el comportamiento del nuevo método ingresado (comparación por coordenadas del punto).

REALIZAR PRÁCTICA: 2.B (Geometria - Parte 1)

Clases que contienen otras clases



En este diagrama especificamos los métodos toString y equals pero más adelante serán obvios como los getters y setters.

Pero sí hay un método especificado que merece la atención que es el método +setRadio(Punto radio) que recibe un punto como parámetro. El método por defecto del radio es +setRadio(double radio) que recibe como parámetro un valor del mismo tipo que el atributo.

La flecha con el diamante vacío indica una **Relación de Agregación** entre Punto y Circulo; tal como lo muestra el diagrama de clase, Circulo “tiene un” Punto. La relación es de agregación ya que la destrucción de un objeto Circulo no implica la destrucción del objeto Punto (si bien en el diseño especificamos si la relación es de composición o agregación, en la implementación no debemos encargarnos de “destruir” los objetos, de esto se encarga Java).

Otro tipo de relación posible es la de **Composición**. Decimos que este tipo de relación es más fuerte que la de asociación porque el ciclo de vida de el objeto de un lado de la relación depende del ciclo de vida de el o los objetos del otro lado de la misma. Cuando la relación es de composición, al desaparecer el objeto contenedor desaparecen también todos sus objetos asociados (contenidos). Podemos suponer un ejemplo: Una agenda con sus contactos: podemos eliminar (destruir) o agregar contactos a la agenda sin que ésta vea afectado su ciclo de vida; pero en cambio, si eliminamos la agenda deben destruirse también todos sus contactos, ya que los contactos de una agenda no tienen razón de ser si no existe la agenda que los contiene.

¿Por qué dijimos antes que la relación entre Punto y Circulo es de agregación (más débil)? ¿Para qué quiero el punto definido si ya no existe el círculo del que es origen? Todo depende (como siempre) de cómo esté planteado el problema: por ejemplo, podemos suponer que existen dos círculos concéntricos, es decir, que comparten el mismo centro. ¿Qué haríamos en este caso? Dos opciones posibles:

1. Generamos dos objetos Punto distintos pero con los mismos atributos y asignamos cada uno a un Circulo.
2. Generamos un solo objeto Punto y lo asignamos a los dos objetos Circulo.

Podemos argumentar que la opción 1 no es conveniente porque utiliza más memoria, haciendo que nuestro diseño sea ineficiente. En el primer caso tendríamos una relación de agregación, en el segundo de composición. Podemos asignar el mismo Punto a dos círculos porque los atributos y variables en Java no contienen otros objetos estrictamente, sino **referencias** a los mismos. Decimos que "la clase Círculo contiene un Punto" de manera coloquial, pero la forma más adecuada sería decir que "la clase Circulo contiene una referencia a un Punto". Podemos imaginarnos una referencia como un medio de comunicación entre objetos: un objeto que tiene una referencia a otro puede invocar sus métodos (en la jerga del paradigma de orientación a objetos se dice que cuando un objeto invoca los métodos de otro le está enviando un mensaje). Así, dos objetos distintos (dos círculos) pueden invocar métodos del mismo punto porque tienen una referencia al mismo, y si podemos invocar los métodos de un objeto podemos hacer cualquier cosa con él. Pero cuidado: Si dentro de un círculo tuviéramos un método que cambiase las coordenadas del centro (es decir, cambia el estado del punto que contiene, un método mover() de la clase Circulo, por ejemplo) también las cambiaría para el otro círculo, ya que ambos tienen una referencia al mismo objeto Punto.

Entonces, lo que diferencia una relación más fuerte de otra más débil es el ciclo de vida de los objetos que participan en ella. Esto era fundamental en los lenguajes en los que el manejo de memoria se hacía a mano, ya que el programador debía liberar la memoria asignada al objeto en el momento adecuado, pero en Java el manejo de la memoria es automático. Posee un mecanismo denominado "recolección de basura" (garbage collection) que se encarga de recuperar la memoria de los objetos que ya no se utilizan: cuando ya no existen referencias a un objeto - esto implica que nadie puede acceder a él - el mismo se marca para ser recolectado. Al recolectar un objeto se libera su memoria y queda disponible para la creación de nuevos objetos; cuando la JVM necesita más memoria, ejecuta una "recolección de basura", que es simplemente recorrer la memoria de trabajo liberando la memoria que tienen ocupada los objetos marcados para ser recolectados. Esto hace que la distinción entre una relación débil y una fuerte desde el punto de vista del manejo de la memoria ya no sea tan importante. Sin embargo, es importante para comprender el funcionamiento de los objetos que componen nuestro sistema y lo es más cuando incorporamos persistencia a nuestro modelo: es necesario saber si cuando se elimina un objeto de la base de datos hay que eliminar otros.

Los atributos de tipo objeto o clase, lo mismo que las variables y los parámetros que reciben los métodos, contienen referencias a los objetos. Por eso hay que tener cuidado de no cambiar dentro de un método los objetos que recibimos como parámetros: Al pasar un objeto como parámetro en realidad estamos pasando una referencia al mismo, así que las modificaciones que hagamos dentro del alcance del método luego serán visibles fuera de éste.

Entonces, ¿No es conveniente modificar un objeto dentro de un método? Bueno, depende. Si el propósito del método es modificar el estado del objeto recibido como parámetro, está documentado que es así, tiene un nombre adecuado y por lo tanto ningún programador que lo utilice se va a llevar una sorpresa, no está mal que así sea. De lo contrario, al modificar los objetos que recibimos como parámetros sin advertencia alguna, quienes utilicen este método de nuestra clase pueden introducir bugs sin darse cuenta. Un ejemplo: En la clase Punto:

```
public void mover(double desplazamientoX, double desplazamientoY){
        x = x + desplazamientoX;
        y = y + desplazamientoY;
}
```

Es evidente que lo que este método hace es "mover" al punto cambiando sus coordenadas en un desplazamiento determinado. Así, si tenemos:

```
Punto p1 = new Punto(10,10);
p1.mover(5,2)
```

nuestro nuevo punto tendrá las coordenadas (15,12).

Además, en la clase Círculo

```
public void mover(double desplazamientoX, double desplazamientoY){
        origen.mover(desplazamientoX,desplazamientoY);
}
```

la idea es la misma: movemos el círculo cambiando las coordenadas de su origen. Notemos cómo reutilizamos el método mover de la clase Punto para conseguirlo.

En la clase Test:

```
Punto punto1 = new Punto(10,10);
Circulo circulo1 = new Circulo(punto1, 10); // origen y radio
Circulo circulo2 = new Circulo(punto1, 20);
```

Este código genera dos círculos concéntricos de radios 10 y 20 con origen en (10,10). Ahora hacemos:

```
circulo1.mover(5,0);
```

Los círculos, ¿Siguen siendo concéntricos? ¿Por qué?

Tomemos un momento para pensarlo. Es importante comprender lo que está ocurriendo con este código.

Ahora que ya lo pensaron, si los círculos siguen siendo concéntricos, está mal: Que invoquemos un método para mover a circulo1 no debería mover también a circulo2. Eso se denomina efecto colateral (side effect) y suele ser fuente de bugs difíciles de encontrar. En este caso es sencillo de ver, pero imaginen un sistema donde tenemos una cadena de objetos que se contienen los unos a los otros, hacemos un cambio en uno y encontramos que cambia un atributo de alguno los demás sorpresivamente. Sorpresivamente porque al fin y al cabo sólo queríamos cambiar lo que queríamos cambiar. Para evitarlo, deberíamos cambiar la implementación del método como:

```
public void mover(double desplazamientoX,
                                        double desplazamientoY){
        // hacemos una copia del origen para no modificar el atributo de
        // la clase
        Punto nuevoOrigen = new Punto(origen.getX(),origen.getY());

        // asignamos el nuevo origen de este círculo
        origen = nuevoOrigen;

         // ahora sí, movemos el círculo
        origen.mover(desplazamientoX,desplazamientoY);
}
```

Al asignar un nuevo punto al origen, lo que en realidad estamos haciendo es cambiar la referencia que teníamos con el punto original a una nueva con el punto nuevo. De este modo cada círculo tendrá su propio punto origen y ya no serán concéntricos, que es lo que buscábamos al desplazar circulo1.

Como norma: un método sólo debe cambiar aquellos objetos que se hayan creado dentro de su ámbito. Modificar el resto es riesgoso, a menos que ese sea el propósito del método y como dijimos, esté documentado. Nuestros métodos deben respetar el principio de mínima sorpresa: El comportamiento de los mismos no debe causar efectos colaterales inesperados, tal como ocurre con los medicamentos. Si

tomáramos una aspirina para curarnos el dolor de cabeza y nos causara una erupción, sin duda estaríamos frente a un efecto colateral inesperado (y desagradable). Lo mismo ocurre con los métodos: Un programador que utilizara nuestro método mover y se le movieran varios círculos al querer mover uno, también estaría frente a un efecto colateral inesperado, se sorprendería y nos recordaría con bastante poco afecto.

Como ya podrán imaginar, también está prohibido pasar información entre objetos a través de variables globales de cualquier tipo. Para eso deben usarse los parámetros de los métodos. El problema es que, además del riesgo de que un método modifique una variable global provocando una sorpresa al ejecutar otro método que la referencie (por efecto colateral), el paso de información a través de globales o atributos de clase no queda documentado. En la signatura de un método debe estar toda la información que un programador necesita para comprender qué hace ese método: Que información debe recibir, qué debe devolver y a través del nombre, que hace. Si con eso no alcanza, se aclarará en los comentarios. El pase de información vía variables globales rompe esta convención, haciendo que el código sea más difícil de comprender y más proclive a errores.

Programar es una actividad que, cuando se realiza en serio, conlleva una gran carga intelectual; es por ello que se trata siempre de disminuirla. Tener que estar alerta a las sorpresas que el código de otros programadores (¡O el propio!) pudiera causarnos, produce un aumento de dicha carga. Debemos tener esto siempre presente al programar, ya que un programa de calidad no sólo lo es por lo que hace y que tan eficientemente lo hace, sino que también lo es por la calidad de su código fuente. Abelson y Sussman dicen en “Estructura e interpretación de los programas de computadora” (SICP por su sigla en inglés), que los programas *deben escribirse primero para ser leídos por las personas e incidentalmente para ser ejecutados en una computadora*. Hay pocas excepciones a esta regla, y cuando se rompe debe ser por razones de eficiencia bien fundadas.

Sobrecarga de métodos

Los métodos están sobrecargados cuando resuelven el mismo caso de uso y tienen el mismo nombre y retorno pero reciben distintos parámetros.
Entonces +setRadio(double radio) y +setRadio(Punto radio) están sobrecargados (retorno void).

Nota: retornar como resultado objeto o una lista de objetos es el mismo retorno (listas los lo veremos más adelante, pero vale la aclaración)

REALIZAR PRÁCTICA: 2.C (Geometria - Parte 2)

Array unidimensional

Un arreglo (array) en Java es una estructura de datos que nos permite almacenar un conjunto de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución. Tenemos dos maneras de definir un array:

1. Array sin inicializar.
2. Array inicializado.

Definición un array sin inicializar:

Para hacer esto usamos la forma “tipoDeDato[] nombreArray = new tipoDeDato[longitud]”.

```
int[] arrayInt = new int[3];
```

En este caso estamos definiendo un array de enteros de tres posiciones. Para poder modificar alguna de las posiciones es necesario primero acceder a ella, por ejemplo supongamos que queremos llenar el array con 1, 2 y 3. Para esto hacemos lo siguiente:

```
arrayInt[0] = 1;
```

```
arrayInt[1] = 2;
arrayInt[2] = 3;
```

Nota: Lo arrays siempre arrancan su índice en 0, por eso para acceder a la primer posición del array usamos el índice 0.

Este ejemplo con int aplica para el resto de los tipos de datos que conocemos.

```
byte[] arrayByte = new byte[3];
short[] arrayShort = new short[3];
long[] arrayLong = new long[3];
float[] arrayFloat = new float[3];
double[] arrayDouble = new double[3];
boolean[] arrayBoolean = new boolean[3];
char[] arrayChar = new char[3];
String[] arrayString = new String[3];
```

Definición un array con datos:

Para hacer esto usamos la forma “tipoDeDato[] nombreArray = {datos}”.

```
int[] arrayInt = {1, 2, 3};
```

En este caso definimos un array de enteros de tres posiciones y lo inicializamos con los números 1, 2, 3.
El mecanismo de acceso es igual que para los arrays sin inicializar.

Nota: En el caso de los array sin inicializar, al momento de crearlos se cargan valores por defecto.

- Para números el valor cero “0”.
- Para cadenas y letras el valor vacío.
- Para booleanos el valor false.

Array bidimensional

En Java es posible crear arrays con más de una dimensión para, por ejemplo, pasar de la idea de lista o vector a la de matriz de N por M elementos (siendo N las filas y M las columnas). Al igual que con los arrays unidimensionales tenemos dos maneras de definir un array bidimensional:

1. Array sin inicializar.
2. Array inicializado.

Definición un array bidimensional sin inicializar:

Para hacer esto usamos la forma “tipoDeDato[][] nombreArray = new tipoDeDato[longitud N][longitud M]”.

```
int[][] arrayInt = new int[3][3];
```

Como podemos ver es muy similar a la forma de trabajo con arrays unidimensionales, sólo que en este caso tenemos que agregar la longitud de las columnas. En el ejemplo creamos una matriz de enteros de 3 filas por 3 columnas.

Definición un array bidimensional inicializado:

Para hacer esto usamos la forma “tipoDeDato[][] nombreArray = {{datos}}”.

```
int[][] arrayInt = {{1,2,3},{4,5,6},{7,8,9}};
```

En este caso la primer tupla ({1,2,3}) corresponde a la primer fila, la segunda ({4,5,6}) a la segunda fila y la última a la tercer fila. Entonces al hacer esto:

```
System.out.print(arrayInt[0][0]);
System.out.print(arrayInt[0][1]);
System.out.print(arrayInt[0][2]);
System.out.print(arrayInt[1][0]);
System.out.print(arrayInt[1][1]);
System.out.print(arrayInt[1][2]);
System.out.print(arrayInt[2][0]);
System.out.print(arrayInt[2][1]);
System.out.print(arrayInt[2][2]);
```

La salida va a ser “1 2 3 4 5 6 7 8 9”. La manera de acceder a una posición del array bidimensional es igual que con los unidimensionales, sólo que agregando el índice de la columna.

Java™ Platform, Standard Edition 7 API Specification

La plataforma Java ofrece muchas clases para ser reutilizadas, está por defecto disponible al crear un nuevo proyecto, por lo tanto no es necesario incluirla con import en la clase.

Las pueden consultar en:

https://docs.oracle.com/javase/7/docs/api/

Por ejemplo: la clase Math que utilizamos el método pow para calcular potencia la pueden consultar en: https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

REALIZAR PRÁCTICA: 2.D (Array Unidimensional)

REALIZAR PRÁCTICA: 2.E (Array bidimensional)

REALIZAR PRÁCTICA: 2.F (Array Consultorio)

Repaso

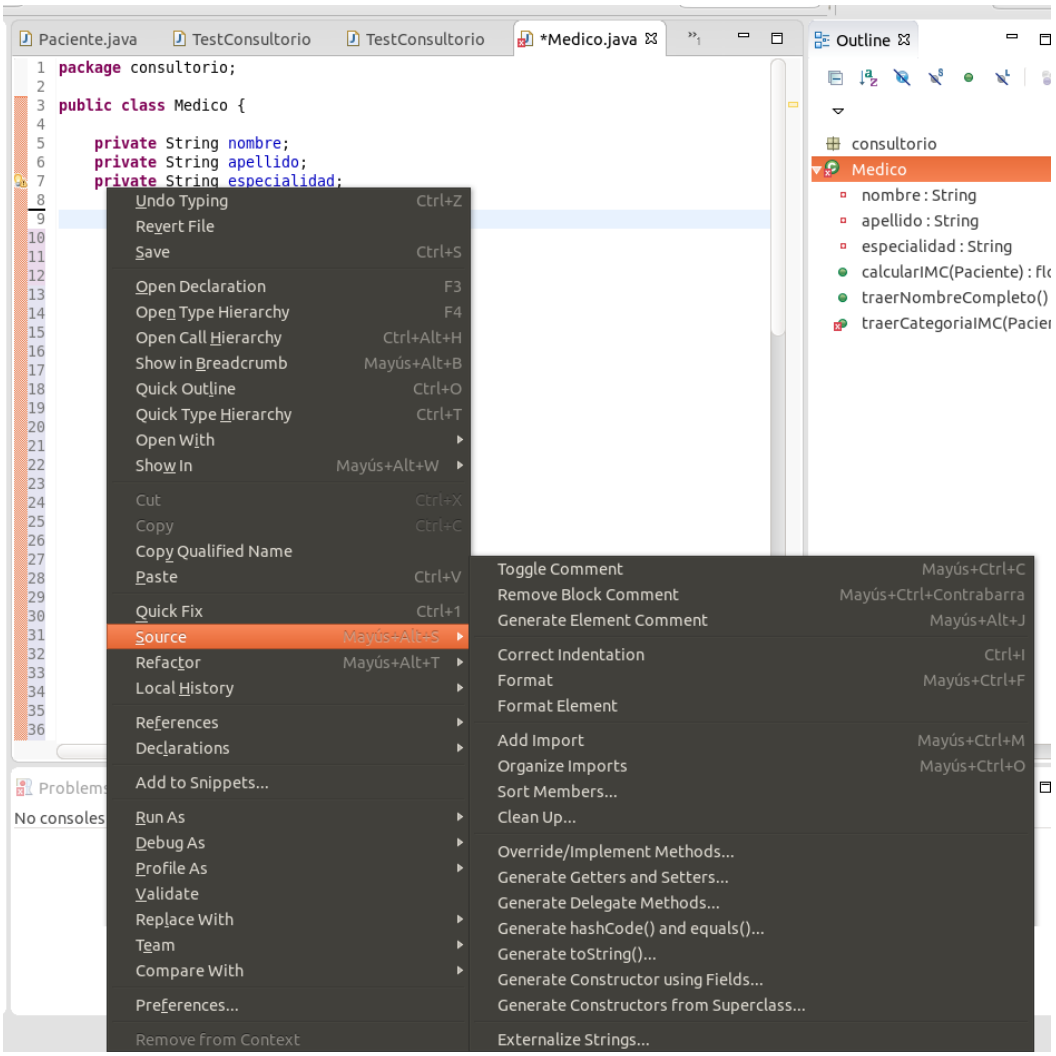
1) Cuando un clase encapsula otro objeto entre estas clases existe un relación de agregación o composición ¿Cuál es la diferencia entre estas relaciones?
2) ¿Qué es polimorfismo? ¿Para qué métodos hasta ahora aplicamos polimorfismo y para qué en cada caso?
3) ¿Cuándo un método está sobrecargado?
4) Tanto la clase Punto como la clase Circulo tienen un método llamado calcularDistancia ¿Esto es un ejemplo de Polimorfismo? ¿Es un ejemplo de Sobrecarga?
5) ¿Qué puede suceder si dos objetos encapsulan el mismo objeto?
6) En un sistema en producción es conveniente modificar la “firma” de un método (su nombre, su tipo de retorno y sus parámetros)?

HELP: Código de Infraestructura en Eclipse

Ya vimos cómo crear nuestros primeros programas utilizado Eclipse. Pero hasta ahora escribimos todo el código a mano y no ha sido un mayor problema, ya que nuestras clases resultan pequeñas en cantidad de atributos y métodos; pero podemos imaginar un escenario con cincuenta clases (no es raro), con cinco atributos promedio cada una: eso da un total de doscientos cincuenta getters, otros tantos setters y alrededor de cien constructores, ya que los mismos (como cualquier método) pueden sobrecargarse y suele hacerse. Sumando toString y equals tendremos alrededor de setecientos métodos y nuestro programa aún no hace nada de lo que nos interesa.

Todo este trabajo, que es percibido como una sobrecarga por los programadores, suele denominarse “boilerplate code” o simplemente “boilerplate” en inglés, y puede traducirse como “código de infraestructura”. Es código que es necesario que esté allí porque contribuye de manera indirecta a la calidad del código en general pero que no resuelve activamente el problema que nos interesa. Además, escribirlo es una tarea repetitiva y tediosa, y las tareas de ese tipo las resuelve el IDE.

Para ello, Eclipse nos ofrece una serie de facilidades que nos aliviarán esta tarea a través de la escritura automática de código. Estas facilidades pueden encontrarse en el menú “Source”, accesibles también con click derecho en la ventana del editor:



Vamos a mencionar que hacen algunas de las opciones y les dejaremos a ustedes la **tarea** de investigar cómo utilizarlas:

- **Toggle comment:** Convierte la(s) línea(s) seleccionada(s) en comentarios. Útil para dejar sin efecto alguna porción del código mientras realizamos pruebas.
- **Correct Indentation:** Corrige la indentación del código. Si no tienen costumbre de indentar el código, úsenlo. No entreguen código sin indentar, es más difícil de leer y da una impresión de desprolijidad.
- **Generate Getters and Setters:** Como su nombre lo indica, genera los getters y setters para los atributos que tengamos definidos. Si agregamos un atributo más tarde, se da cuenta de cuales setters y getters ya están definidos y no los redefine. Permite seleccionar de qué atributos queremos generarlos y generar los getters y los setters por separados si es necesario.
- **Generate toString():** Genera el método toString() utilizando los atributos que indiquemos.
- **Generate Constructor using fields:** Genera el constructor de la clase definiendo como parámetros los atributos que le indiquemos. Podemos indicar distintos atributos cada vez y generar constructores sobrecargados.
- **Surround with:** Ofrece varias opciones para encerrar el código marcado con distintas construcciones de control e iteración.

Otra opción útil del menú principal es **Refactor/rename**: Es una especialización de find and replace. Permite cambiar el nombre de un paquete, clase, atributo, método en un solo sitio y Eclipse buscará todas las apariciones del mismo para cambiarlo. También está disponible en el menú contextual de la ventana del editor.

También dentro del menú refactor encontraremos la opción **Move**, que nos permite mover clases a distintos paquetes reorganizando los imports necesarios automáticamente.

Algo que suele ocurrir es que nuestro proyecto comience a dar errores que no comprendemos qué los causa, ya que nuestro código parece ser correcto. Esto puede ocurrir al importar proyectos que se compilaron con otra versión de java y que esto esté causando problemas de compatibilidad binaria. Para resolverlo, utilizaremos la opción **Project/Clean** del menú principal. Esta opción elimina todos los archivos .class compilados obligando a que se generen nuevamente en la próxima corrida.

Práctica

2.A (Números)

Dada la clase Numero (paquete modelo), con el atributo n de tipo entero, implementar los siguientes métodos:

1) +sumar(int n1) : int
2) +multiplicar (int n1) : int
3) +esPar(): boolean //Utilizar operador % ej: 5%2 devuelve el resto de 5/2 ----> 1
4) +esPrimo (): boolean
5) +convertirAString(): String //Ver String.valueOf
6) +convertirDouble(): double //Ver Double.parseDouble
7) +calcularPotencia (int exp): double //ver Math.pow (double base, double exponente) : double ; Double
8) +pasarBase2 (): String
9) +calcularFactorial(): int (si el número si n>0 es el producto 1*2..*n, si n es 0 el factorial es 1, si el n<0 retorne 1
10) +numeroCombinatorio(int n1): int // para n y n1 positivos; n>n1 devuelve $Cn; n1$

En la clase TestNumero.java (paquete test) generar los resultados de los 10 métodos.

2.B (Geometria - Parte 1)

A la implementación vista, agregar lo siguiente:

1) Crear un TestPunto

Escenario 1: crear dos instancias (objetos) de Punto distintas. Imprimir ambas. Imprimir el resultado de equals entre los puntos.

Escenario 2: crear dos objetos de Punto iguales, imprimir ambos. Imprimir el resultado de equals entre los puntos.

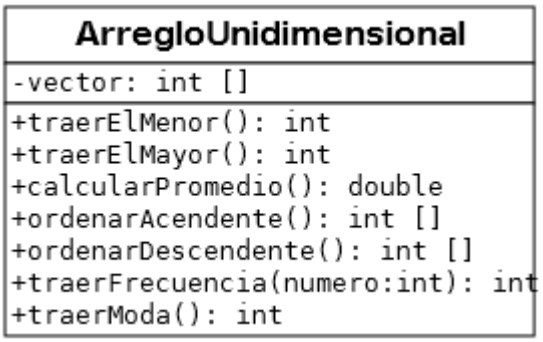
2) Implementar el método para calcular la distancia entre dos puntos como lo muestra el diagrama de clases.

2.C (Geometria - Parte 2)

Implementar los casos de uso pendientes en el modelo de Punto y Círculo.

2.D (Array Unidimensional)

Modelo:



+ ordenar(): int []

Ordenar por el Método Burbuja: recorrer el vector comparando cada elemento con el siguiente, que de resultar mayor se intercambian de lo contrario sigue, se vuelve a recorrer hasta que no se realicen ningún intercambio.

+ traerFrecuencia(int numero): double

Es el cociente entre cantidad de veces que aparece el parámetro y la cantidad de valores del arreglo

+ traerModa(): int

Como lo dice la palabra lo que está de “Moda” el valor que más aparece en el arreglo, utilizado en todas las estrategias de promoción, cual es la pelicula mas vista esta semana, el libro más vendido etc. En otras palabras la moda es el elemento que tiene mayor frecuencia.

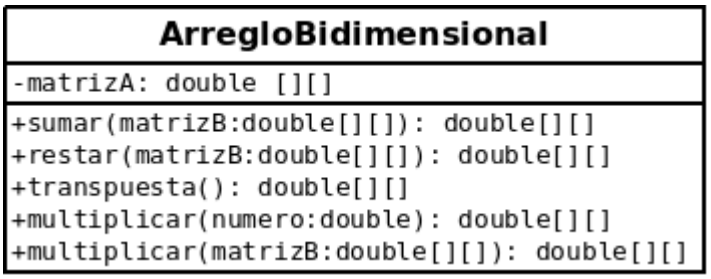
Ver método .length para obtener la longitud del arreglo.

Test:

En la clase **ArregloUnidimensionalTest.java**, testear el comportamiento de los casos de uso de la clase ArregloUnidimensional

2.E (Array bidimensional)

Modelo:



$$\begin{pmatrix} a_{11} & a_{12} & a_{13....} & a_{1n} \\ a_{21} & a_{22} & a_{13....} & a_{2n} \\ & ..... & & \\ a_{m1} & a_{m2} & a_{m3....} & a_{mn} \end{pmatrix}$$

+ sumar(double [][] matrizB): double [][]

Las matrices deben tener la mismas dimensiones y los elementos de la matriz resultante es el resultado de $c_{i,j} = a_{i,j} + c_{i,j}$

+ restar(double [][] matrizB): double [][]

Las matrices deben tener la mismas dimensiones y los elementos de la matriz resultante es el resultado de $c_{i,j} = a_{i,j} - c_{i,j}$

+ transpuesta(): double [][]

La matriz transpuesta se obtiene convirtiendo las filas en columnas.

$$A_{m\ n} = (a_{ii}) \mid A^{t}_{n,m} = (a_{ji})$$

$$A = \begin{pmatrix} 2 & 1 & 0 & 7 \\ 3 & 4 & 2 & -1 \\ 1 & 0 & 5 & 8 \end{pmatrix} \quad A^{t} = \begin{pmatrix} 2 & 3 & 1 \\ 1 & 4 & 0 \\ 0 & 2 & 5 \\ 7 & -1 & 8 \end{pmatrix}$$

+ multiplicar(numero:double): double [][]

Producto de un escalar por una matriz: $K * (a_{ij}) = (K * a_{ij})$

+ multiplicar(double [][] matrizB): double [][]

Método sobrecargado, para poder realizar el producto el número de columnas de la primera matriz tiene que ser igual al número de filas de la segunda.

$$c_{i,j} = a_{i,1} * b_{1,j} + a_{i,2} * b_{2,j} + ... + a_{i,n} * b_{n,j} = \sum_{k=1}^{n} a_{i,k} * b_{k,j}$$

Ejemplo:

$$\begin{pmatrix} 2 & 3 & 5 & 1 \\ 7 & 2 & 4 & 3 \\ -1 & 5 & 0 & 8 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 7 & 2 \\ 0 & -5 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 27 & -17 \\ 33 & -9 \\ 66 & 9 \end{pmatrix}$$

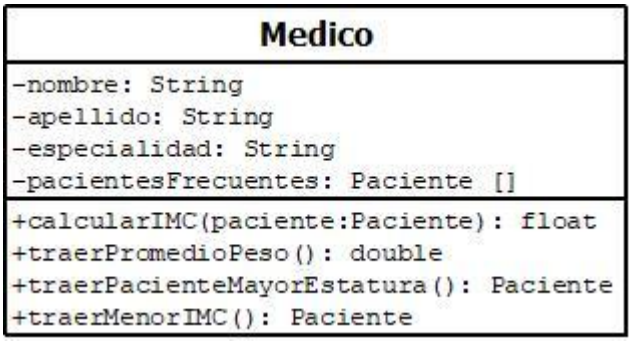
En todos los casos de uso si no es posible realizarlo por no cumplir el parámetro las condiciones de comportamiento, retornar null

Test:

En la clase **ArregloBidimensionalTest.java**, testear el comportamiento de los casos de uso de la clase ArregloBidimensional

2.F (Array Consultorio)

Modelo:



Para este ejercicio vamos a retomar el proyecto de consultorio que estuvimos trabajando la clase pasada y sumaremos funcionalidad.

+ traerMenorIMC(): Paciente

Tener en cuenta que aquí podemos reutilizar el método que calcula el IMC para obtener el valor.

Test:

En la clase **TestConsultorio.java**, testear el comportamiento de los casos de uso agregados.