Licenciatura en Sistemas - Orientación a Objetos I – 2018

Prof. Titular:	Lic. María Alejandra Vranić	alejandravranic@gmail.com
Prof. Ayudantes:	Lic. Romina Mansilla	romina.e.mansilla@gmail.com
	Lic. Leandro Ríos	leandro.rios.unla@gmail.com
	Lic Gustavo Siciliano	gussiciliano@gmail.com

Introducción al paradigma de objetos

En este curso vamos a aprender una nueva forma de abordar y resolver los problemas, distinta del enfoque que veníamos empleando hasta ahora. En pocas palabras vamos a aprender a desarrollar software usando el paradigma de orientación a objetos. Es por eso que vamos a pedirles que se olviden momentáneamente de lo que saben para que podamos encarar este nuevo aprendizaje con la mente lo más despejada posible. Más adelante veremos cómo los conceptos que aprendimos hasta ahora siguen teniendo vigencia, pero de una manera distinta y dentro de un marco más general de desarrollo.

Ahora bien: ¿De qué se trata? ¿Qué es esto que denominamos "objetos"? Vamos a esbozar algunas definiciones sueltas y luego veremos como las enlazamos para arribar a una definición del paradigma que nos sirva para comenzar a estudiarlo en más profundidad.

Objetos y clases de objetos

Desde la óptica del mundo real, un objeto es todo aquello que tienen entidad (que es) y se diferencia claramente de su entorno, pero en sentido general: Los seres vivos también son objetos de acuerdo a la misma: Una piedra, una taza, un auto, un perro, una noticia y Leandro Ríos son objetos.

Veamos algunas cosas que podemos decir de acuerdo a esta definición: un objeto exhibe algunas características propias: una taza tiene una altura, un color y un volumen neto (la cantidad de líquido que podemos poner sin que se derrame), y una persona tendrá una estatura, una edad, y un color de pelo, por nombrar algunas. Este conjunto de características o atributos define a cada **objeto** y nos permiten diferenciar entre las distintas **clases** de objetos.

Por otro lado, los objetos además hacen cosas: Leandro Ríos respira y da clase, por ejemplo. Una piedra no hace mucho, en este momento no se me ocurre que cosas puede hacer una piedra. Pero está bien, porque hay objetos que no hacen nada.

Entonces, podemos afinar un poco más nuestra definición y decir que un **objeto** es todo aquello que tiene entidad, que se diferencia claramente de su entorno, que se distingue por un conjunto de **atributos** y que puede tener un **comportamiento** asociado (hacer cosas o no). Así, un objeto queda definido por lo que llamaremos su **estado** (sus atributos) y su comportamiento.

Al conjunto de los atributos se lo denomina estado porque además de proveer información estática del objeto (color, forma, etc) nos ofrecen información dinámica (que cambia con el tiempo): Si está caliente o frío, si está parado o sentado, etc. Podemos decir que Leandro Ríos de pie es el mismo objeto que Leandro Ríos sentado pero con otro estado.

Podemos además notar que hay ciertas características comunes a Leandro Ríos y a Gustavo Siciliano, por ejemplo: una cabeza, dos pies, dos ojos, entre otras. Si abstraemos las diferencias entre ambos y entre el resto de nosotros, podemos definir una **clase** de **objetos** con la que podremos clasificar a todos los seres humanos de este curso y por qué no del planeta: podemos decir, casi diría que sin equivocarnos, que todos somos personas.

Entonces definimos a la clase Persona con los siguientes atributos: nombre, género, estatura, fecha de nacimiento, domicilio.

Y nos detenemos allí. Una persona tiene infinidad de características; pero con las que enumeramos basta para el ejemplo. Además, una persona hace cosas, entre otras: respirar, comer, caminar.

Habiendo definido esta clase Persona, podemos tener distintos objetos que pertenezcan a la misma:
nombre: Gustavo Siciliano, género: Masculino, estatura: 1,81 mts, fecha de nacimiento: 10/10/1990 domicilio: Avellaneda.
nombre: Leandro Ríos, género: Masculino, estatura: 1,87 mts, fecha de nacimiento: 16/05/1966, domicilio: Lanús.
nombre: Alejandra Vranić, género: Femenino, estatura: 1,62, fecha de nacimiento: 30/09/1961, domicilio: Temperley

Decimos que cada uno de estos **objetos** es una **instancia** de la **clase** Persona. Los tres comen, respiran y caminan.

En realidad estamos clasificando nuestra percepción de la realidad de acuerdo a nuestro criterio (la palabra clasificar viene de clase). A alguien se le podría haber ocurrido decir que esos tres objetos en realidad pertenecen a dos clases distintas: hombre y mujer por ejemplo. No está bien una y mal la otra, la elección depende del objetivo para qué estamos clasificando la realidad.

Ahora: ¿Qué tiene que ver toda esta discusión cuasi filosófica con el asunto que nos reúne en un curso de la carrera de sistemas? Buena pregunta: La respuesta es que cuando intentamos resolver un problema del mundo real con una computadora, *comenzamos por realizar una clasificación de algún tipo*. Si queremos hacer un sistema de facturación por ejemplo, tenemos que considerar artículos, clientes, facturas, recibos, proveedores, empleados, etcétera, etcétera -un etcétera casi infinito. Cuando miramos de cerca a una factura vemos que tiene una fecha, un cliente, una cantidad de ítems y un total. Cuando miramos a un ítem de cerca vemos que tiene un artículo, una cantidad y un precio. Cuando miramos un artículo vemos que tiene una descripción y un precio unitario. También podemos ver que todos estos objetos que acabo de describir se contienen y relacionan de manera armoniosa para llevar adelante una tarea que es que alguien que compra algo se vaya con su factura y la AFIP no tenga de qué quejarse. Tomemos nota de que esto es así aunque no exista un sistema informático de facturación: Si facturamos a mano también tendremos facturas, artículos, clientes, etc.

Hasta ahora, comenzábamos pensando a los sistemas a partir de qué cosas tenían que hacer y los diseñábamos como una secuencia de instrucciones de alto nivel que descomponíamos en instrucciones más simples hasta que llegábamos al nivel más bajo que es el del lenguaje de programación del que disponíamos; mientras que los datos eran entidades ajenas que eran manipulados por esas instrucciones. El paradigma de objetos, por otro lado, comienza por pensar cuál es la mejor manera de representar la realidad para generar un **modelo de clases** que pueda manipular una computadora para solucionar un problema.

Desde el paradigma de objetos, para abordar un problema se descompone el dominio del problema (el dominio del problema es la parte de la realidad en la que ocurre: las cosas que están involucradas, es decir, la porción del universo a la que afecta ese problema a resolver) en clases que definirán objetos y de la interacción entre dichos objetos surge la solución al problema. Esos objetos pueden actuar de manera independiente o a través de sus relaciones con otros objetos. Como en la vida real, cada objeto posee cierta información en su estado y puede realizar distintas tareas a partir de esa información o de información que le proveen otros objetos. Las tareas que no pueda realizar, ya sea por falta de información o de algún comportamiento, las delegará en otro u otros objetos que sean capaces de llevarlas a cabo y con los que se encuentre relacionado.

Piensen en un equipo de desarrollo: El analista con la información que tiene disponible del entorno y la propia (su saber) genera un diseño (ejecuta un comportamiento: diseñar), que será la información que le pasará al programador, quien con esa información recibida y la propia (su saber también) generará un programa (otro comportamiento: programar), que si ambos son buenos profesionales, el cliente conoce su negocio y los planetas se alinean, hará lo que este último desea.

Diseñar un programa siguiendo el paradigma de objetos se reduce entonces a realizar una clasificación de la realidad y asignar los atributos y comportamientos que corresponden a cada clase. Pero cuidado: dijimos que hay muchas maneras de clasificar la realidad y que no podemos determinar que alguna sea mejor por su propio mérito. Hay un ensayo de Jorge Luis Borges llamado “El idioma analítico de John Wilkins”, en el que imagina una enciclopedia china que, entre otras cosas, contiene una clasificación de los animales como sigue:

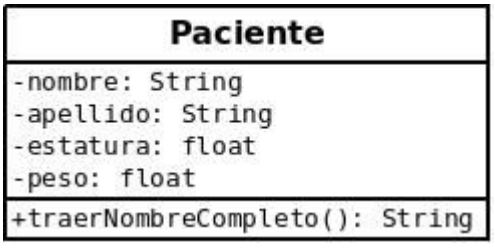
(a) pertenecientes al emperador,
(b) embalsamados,
(c) amaestrados,
(d) lechones,
(e) sirenas,
(f) fabulosos,
(g) perros sueltos,
(h) incluidos en esta clasificación,
(i) que tiemblan como enojados,
(j) innumerables,
(k) dibujados con un pincel finísimo de pelo de camello,
(l) etcétera,
(m) que acaban de romper un jarrón,
(n) que de lejos parecen moscas.

Concluye Borges en el relato: *"[...] notoriamente no hay clasificación del universo que no sea arbitraria y conjetural."*. Si bien esto es cierto, nosotros sí tenemos una medida de la pertinencia o calidad de nuestra clasificación: En primer lugar, que es adecuada para resolver nuestro problema y en segundo, que lo hace de manera eficiente y eficaz. Otras consideraciones incluyen que el diseño que resulte sea extensible, mantenible y que posibilite la reutilización. Esto quiere decir que si bien desde una perspectiva filosófica todas las clasificaciones son igualmente buenas, desde el punto de vista del diseño de sistemas de información no es así. A lo largo del curso iremos aprendiendo los criterios que nos permitirán distinguir una clasificación buena de otras, ya que una clasificación buena (que es parte del análisis) dará lugar a un buen diseño, y éste a su vez a un buen sistema.

Estructura de una clase

Dijimos que una clase tiene estado y comportamiento. El estado se compone de **atributos** y el comportamiento está representado por **métodos**.

Vamos a imaginar que tenemos que implementar un sistema para un consultorio médico. Para ello, comenzaremos definiendo la clase Paciente, con la que representaremos a todos los pacientes del consultorio. Nuestra primera definición tendrá lo mínimo necesario y la iremos ampliando.



Estado (Atributos)

Comportamiento (Métodos)

Este diagrama que ya conocemos de algún otro lado se denomina diagrama de clases y se utiliza para representar las clases con las que modelamos nuestra solución y las relaciones entre las mismas. Utilizamos el programa Dia para generar los diagramas de clase, es código libre y se encuentra disponible en http://dia-installer.de/index.html.es

Vemos que cada atributo es de un tipo de dato determinado. El tipo de dato que se indica en el método traerNombreCompleto() es el que devuelve el método.

Podemos ver que al definir la clase Paciente (los nombres de las clases siempre comienzan con mayúscula) lo que estamos haciendo es definir un molde o plano con el que vamos a generar objetos de tipo paciente. Las distintas **instancias** de Paciente tendrán distintos valores de sus atributos y distinta **identidad**. Los métodos de la clase paciente sólo podrán operar sobre los valores actuales de los atributos de la clase y los argumentos que pudieran recibir. Por ejemplo, si tuviéramos una instancia de Paciente con nombre “Leandro” y apellido “Ríos”, el método traerNombreCompleto() devolverá “Leandro Ríos”: es decir, opera sobre los atributos nombre y apellido al concatenarlos, retornando un nuevo string.

Ahora vamos a relacionar todo esto que venimos viendo con lo que ya sabían de antes: lo que antes se denominaba procedimientos y funciones ahora serán métodos, y lo que antes eran variables ahora serán atributos o variables. La diferencia está en cómo se organizan esos métodos, atributos y variables: mientras antes el foco estaba puesto en los procedimientos y funciones (lo que "hace" el programa) y los datos eran entidades que andaban flotando más o menos por ahí, ahora todos ellos se estructuran en clases que los contienen y a las que pertenecen. Esto tiene una gran relevancia con respecto a la reusabilidad de nuestro código, ya que una clase bien diseñada es un módulo independiente que podremos reutilizar en cualquiera de nuestros programas (o de otros, por qué no).

Por ahora sólo tenemos la clase paciente y con ella comenzaremos a aprender el lenguaje de programación y el entorno integrado de desarrollo que utilizaremos durante la cursada: Java y Eclipse.

El lenguaje de programación Java

Java es un lenguaje de programación que nació con la idea de que fuera portable y fácilmente embebible en electrodomésticos. Esto quiere decir que los programas escritos en él se pudieran ejecutar en cualquier combinación de arquitectura y sistema operativo sin necesidad de recompilar. Sabemos que cuando escribimos un programa en un lenguaje compilado, por ejemplo C, sólo correrá en la arquitectura y sistema operativo para el que lo hayamos compilado. Si queremos que corra en otra arquitectura (o sistema operativo) deberemos recompilarlo para la misma. Esto implica también que si el programa hace llamadas al sistema operativo directamente, deba modificarse el código fuente del mismo para que pueda correr en la nueva plataforma. Los programas C que están pensados para ser portables, están llenos de cosas como ésta:

```
#ifdef WIN32
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <tchar.h>
static HMODULE sGLESDLL = NULL;
#endif // WIN32

#ifdef LINUX
#include <stdlib.h>
#include <dlfcn.h>
static void *sGLESSO = NULL;
#endif // LINUX
```

Todo lo que hay que entender son los ifdef: indican si se ha definido para qué plataforma se está compilando el programa. Si se trata de WIN32 (Windows de 32 bits) se incluyen ciertos archivos cabecera y se definen algunas constantes. Si se trata de LINUX, serán otras; de este modo el compilador sabrá cuales archivos incluir en la compilación. Podemos ver que hay que mantener toda una serie de archivos distintos dependiendo de las plataformas en las que corra nuestro programa, y hay que repetir los ifdefs por todos los módulos de código fuente donde haya que considerar la portabilidad. Esto es engorroso y proclive a errores.

¿Cómo hacer entonces para que los problemas de portabilidad no afecten la escritura de programas? Podemos reconocer que se trata de dos problemas distintos: Por un lado el problema que realmente tenemos que resolver (aquello que nuestra aplicación lleve a cabo) y por otro, el de la portabilidad.

Los ingenieros de Sun, creador de Java (James Gosling, Mike Sheridan, y Patrick Naughton) decidieron que al tratarse de dos problemas distintos, deberían ser resueltos por programas distintos: Para ello, se eligió definir una máquina virtual (VM, virtual machine), que se encargaría de "traducir" entre un formato determinado (el de la VM, denominado bytecode) al que se compila nuestro programa y el de la arquitectura/Sistema Operativo (HW/SW en adelante) en el que la VM corriera. De este modo, para correr un programa Java en una combinación HW/SW determinada, basta con asegurarse de que exista una VM para el mismo.

Este concepto de VM está relacionado con el de las máquinas virtuales tipo VirtualBox, VMWare o Qemu, pero no es lo mismo. Éstas últimas virtualizan el hardware de una máquina determinada, de forma de poder instalarle un sistema operativo y correr programas para el mismo. La VM de java, por otro lado, define una máquina estándar que no existe físicamente (la especificación de la VM Java -la JVM- permite crear dispositivos hardware basados en ella, y de hecho existen algunos, pero no nos ocuparemos de ellos), sino que sólo sirve para ofrecer un ambiente definido en el que se ejecuten los programas Java.

Correr sobre una VM les da a los programas java la ventaja de la portabilidad: desde un smartphone y dispositivos embebidos hasta los servidores de aplicaciones empresariales pueden correr programas escritos en Java sin recompilar, en tanto exista una JVM que corra en ellos. Existen JVM para prácticamente todas las arquitecturas y sistemas operativos; en algunos casos (como el del sistema operativo Android) lo ofrecen como lenguaje “oficial” de la plataforma.

Si bien la portabilidad es un ventaja importante, existen algunas desventajas, las principales son dos: Una es el espacio de memoria (antes de correr cualquier programa hay que cargar la JVM en memoria) y otra es la velocidad comparada con C: es más lento, ya que antes de ejecutar una instrucción hay que traducirla al código ejecutable de la plataforma en la que corre la JVM, lo que consume tiempo. Para reducir esta demora, algunas JVM desde hace un tiempo vienen equipadas con lo que se denomina un compilador JIT (Just in time, “justo a tiempo” o por demanda), que no es más que un compilador que transforma la representación en bytecode que corre en la JVM, en la representación objeto la primera vez que se ejecuta la instrucción, reemplazando el bytecode en memoria por la versión compilada más eficiente. De este modo, la próxima vez que deba ejecutarse la instrucción se ejecutará la versión compilada sin necesidad de traducirla. La JVM HotSpot de Oracle cuenta con un compilador JIT. También ofrece “optimización adaptativa”: es un proceso que inspecciona constantemente el programa en ejecución buscando aquellos lugares en los que el código pasa más tiempo (“hot spots”) y por lo tanto son candidatos para ser optimizados automáticamente.

Java es entonces un lenguaje orientado a objetos, estáticamente tipado, compilado a bytecode, interpretado y cuya sintaxis es muy similar a la de C/C++. Esta sintaxis se eligió en su momento para atraer a los programadores C/C++ que eran mayoría en esa época (mediados de los años 90)

Tipos de datos en Java

Java divide los tipos de datos en dos clases fundamentales: Los tipos de datos primitivos y los tipos de datos por referencia.

Los tipos de datos primitivos son como los que conocemos de otros lenguajes: básicamente tipos de datos numéricos (entero, float, double, byte, char, etcétera). Los tipos de datos por referencia son todos los tipos de datos que son objetos y los arrays.

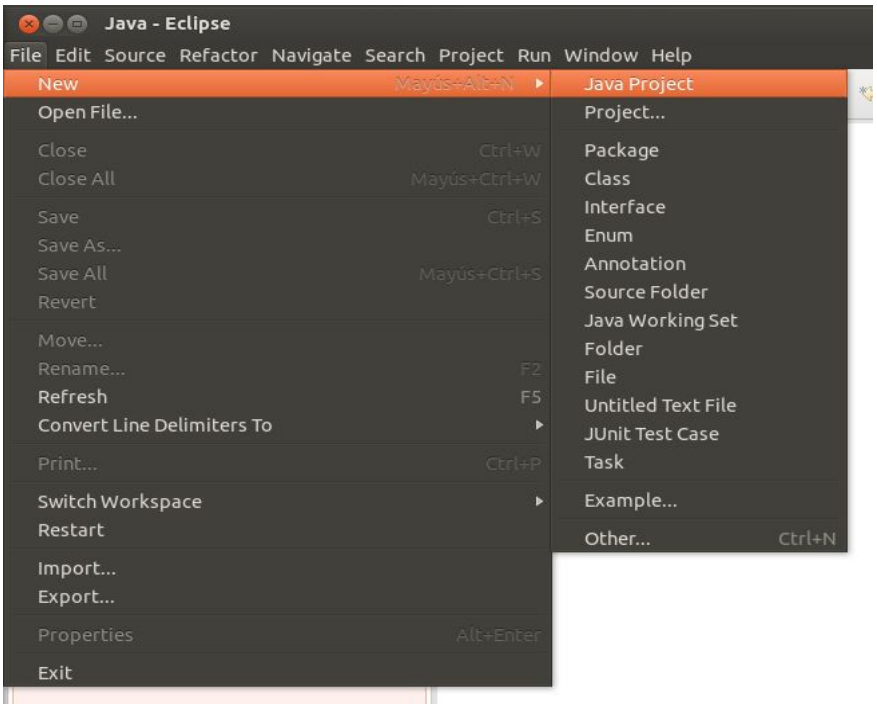
Los lenguajes de programación orientados a objetos más estrictos no hacen esta distinción: Para ellos, todos son objetos y pertenecen a alguna clase. Esto no deja de tener su lógica, ya que en algún lugar deben estar las operaciones que afectan a esos tipos de datos, y el lugar más adecuado de acuerdo al paradigma de objetos es la clase que los define. Tener tipos de datos primitivos obliga a tener librerías de funciones para manipularlos (como por ejemplo la clase Math en Java). A partir de Java 5 se agregaron las clases que encapsulan los distintos tipos primitivos, abriendo la posibilidad de que puedan usarse ambos. Existe una gran controversia alrededor de este asunto; el mayor argumento por la utilización de tipos primitivos es la eficiencia que éstos proveen, ya que consumen menos memoria y su utilización resulta en un menor tiempo de ejecución. Nosotros adoptaremos el camino más conservador y emplearemos tipos de datos primitivos siempre que sea posible.

1° programa Java

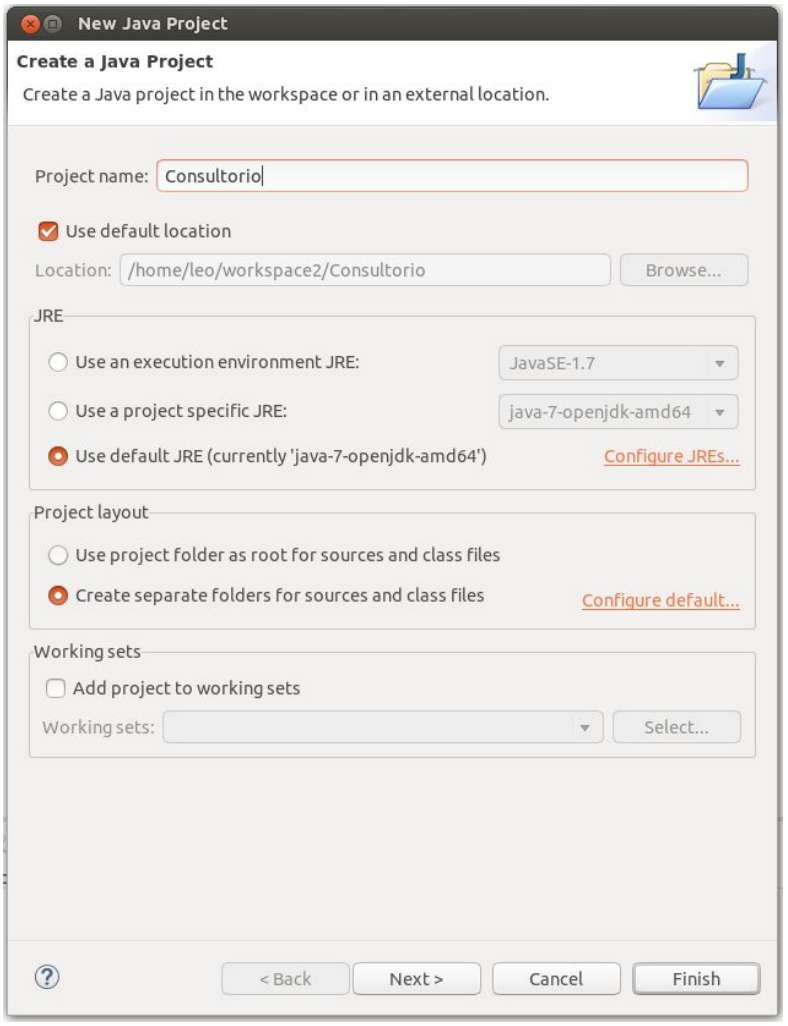
Para todas los trabajos prácticos del curso utilizaremos el entorno de desarrollo (IDE, Integrated Development Environment) Eclipse. Las máquinas del laboratorio tienen el icono para iniciar en el escritorio. Deben investigar cómo instalarlo en sus casas para poder realizar los trabajos prácticos domiciliarios. Pueden bajar Eclipse de https://eclipse.org/downloads/. también van a necesitar tener instalado el JDK (Java Development Kit), si no lo tiene ya instalado. Pueden bajarlo de:

http://www.oracle.com/technetwork/es/java/javase/downloads/index.html

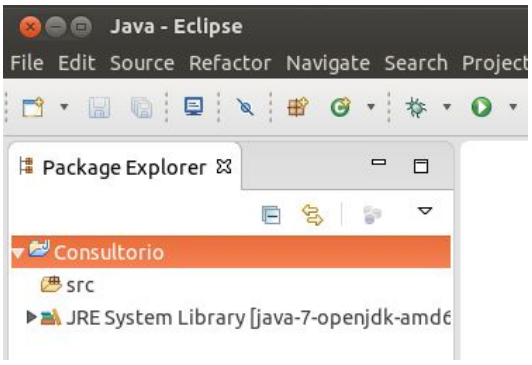
Vamos a comenzar por crear un nuevo proyecto en Eclipse:



Se abrirá la siguiente ventana, la completamos como en la figura y hacemos click sobre “Finish” (Los apartados “JRE” y Location los dejamos como los ofrece Eclipse)



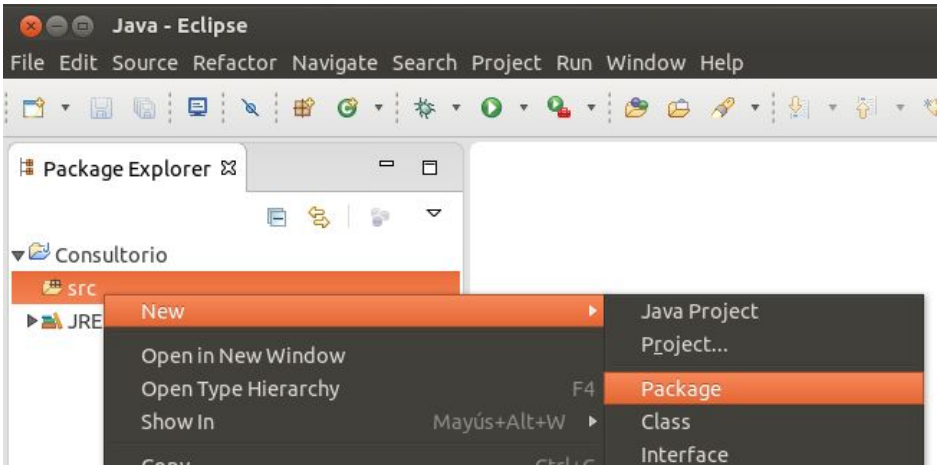
Luego de dar click sobre Finish, debería quedarnos una estructura de proyecto como la siguiente:



Antes de continuar es necesario definir algunos conceptos:

Ya dijimos que en el paradigma de la programación orientada a objetos, todo el código se organiza en clases, y no puede existir código fuera de una clase. Un paquete es un conjunto de clases que se asocian por alguna causa. En Java, las clases siempre se organizan en paquetes y las clases que se encuentran en el mismo paquete es porque forman parte del mismo subsistema. Entonces, vemos que si queremos escribir código Java, tenemos que hacerlo en una clase. Y si queremos tener una clase, ésta debe pertenecer a algún paquete. Así que comenzaremos definiendo un paquete para nuestra clase, para luego definir la clase misma y finalmente, escribir el código de nuestra clase.

En un proyecto Java en Eclipse, todos los paquetes se definen en la carpeta “src”, ya que es en la misma donde el compilador espera encontrar el código fuente (src es abreviatura de source, que significa fuente en inglés). Entonces, para agregar un paquete a la carpeta src, hacemos click derecho sobre el icono que lo representa:

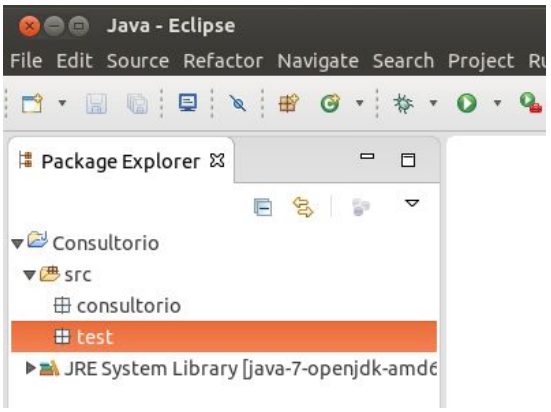


Le pondremos como nombre “consultorio” y daremos click sobre “Finish”.

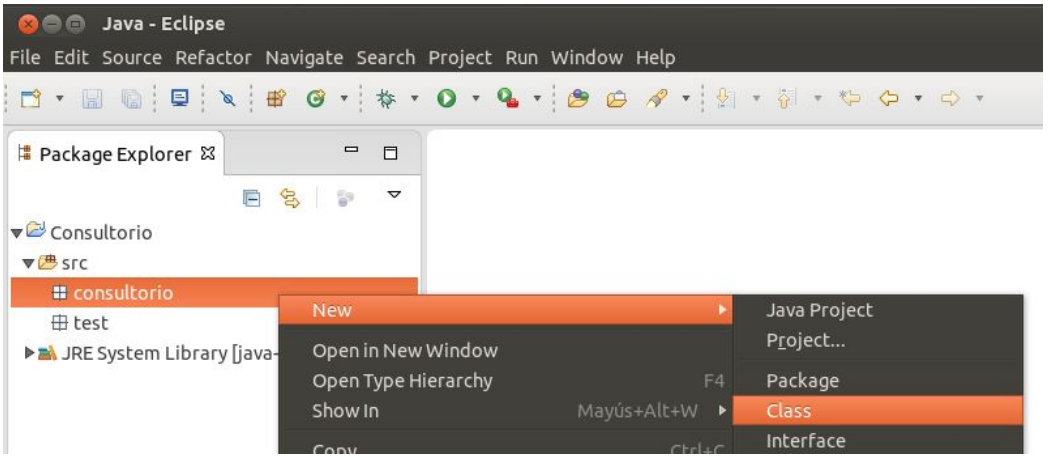
Dijimos que no puede existir código fuera de una clase, así que si queremos invocar los métodos de nuestros objetos Paciente deberemos tener una clase que lo haga. Todos los programas Java tienen por lo menos una clase con un método que pueda invocar el entorno Java para iniciar el programa, y diremos que esa clase es ejecutable.

Durante la cursada, todas las clases ejecutables pertenecerán a algún paquete que en su nombre incluya la palabra test. Entonces, del mismo modo que para el paquete consultorio agregamos un paquete test.

La estructura de nuestro proyecto deberá quedar de la siguiente manera:

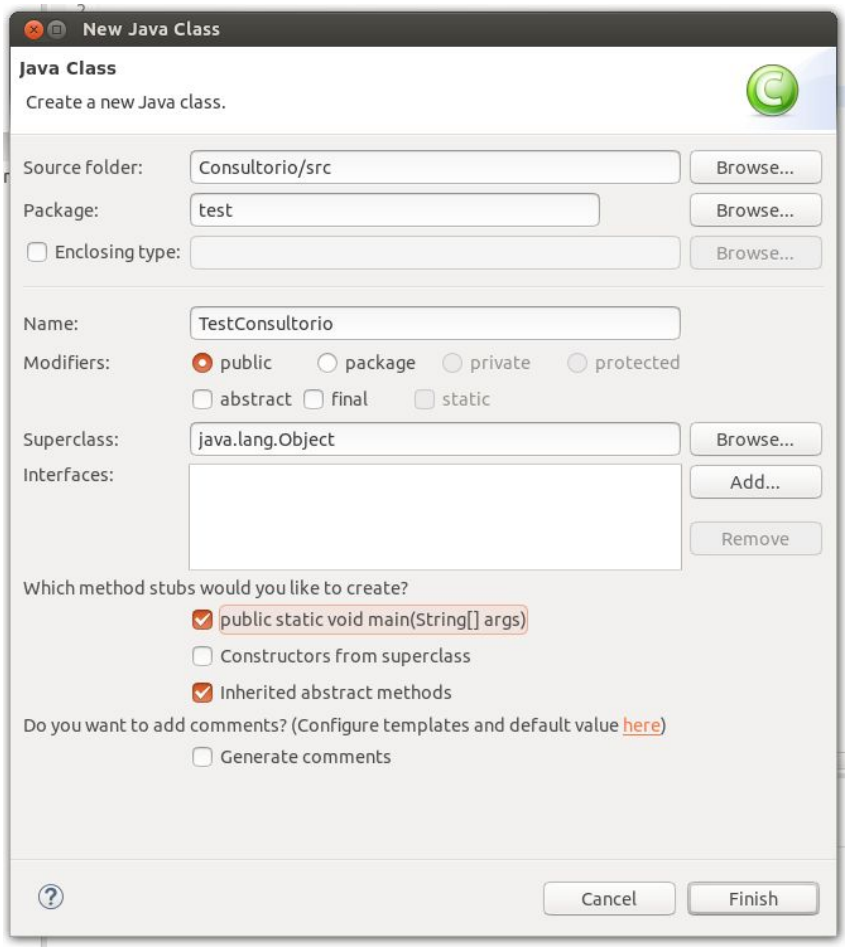


A continuación agregaremos nuestra clase Paciente en el paquete consultorio. Para ello, damos click derecho sobre el paquete:

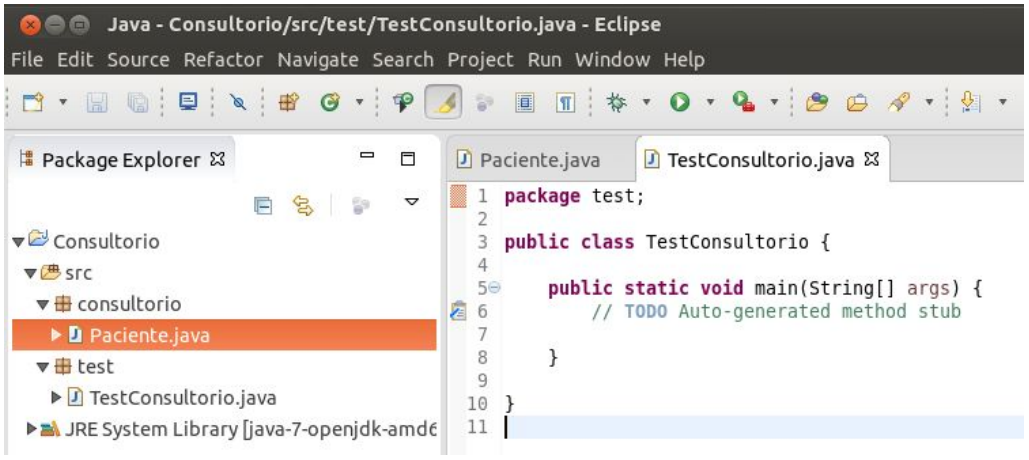


Le ponemos como nombre “Paciente”. Los nombres de las clases comienzan siempre en mayúscula y respetan la convención CamelCase: la primera letra de cada palabra que compone el nombre va siempre en mayúscula. Así, si tuviéramos una clase que represente a pacientes de obra social, su nombre podría ser PacienteObraSocial.

Creamos del mismo modo, pero sobre el paquete test otra clase TestConsultorio, que será ejecutable. En este caso, además de introducir el nombre, también marcamos el checkbox con la leyenda “public static void main(String args[])”. Eso le indica al compilador que la clase es ejecutable y main es el método que se invocará para iniciar la ejecución (como en C y C++).



La estructura de nuestro proyecto deberá quedar ahora como sigue:



Vemos que Eclipse ha generado las declaraciones de las clases Paciente y TestConsultorio. hacemos doble click sobre la clase paciente (como el la figura de arriba) y deberá abrirse el código de la misma en el editor.

Notamos también que Eclipse ha generado un archivo .java por cada una de las clases creadas: en Java, cada clase tiene su propio archivo, esto facilita su reutilización en distintos proyectos.

Comenzaremos a completar la definición de nuestra clase Paciente. En el editor introduciremos código para que quede de la siguiente manera:

```
package consultorio;

public class Paciente {

    private String nombre;
    private String apellido;
    private float estatura;
    private float peso;

    public Paciente(String nombre, String apellido, float estatura,
            float peso) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.estatura = estatura;
        this.peso = peso;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public float getEstatura() {
        return estatura;
    }

    public void setEstatura(float estatura) {
        this.estatura = estatura;
    }

    public float getPeso() {
        return peso;
    }

    public void setPeso(float peso) {
        this.peso = peso;
    }

    public String traerNombreCompleto(){
        String resultado;
        resultado= nombre+" "+apellido;
        return resultado;
    }
}
```

Lo primero que notamos es la similitud del código Java con C/C++. Utilizamos punto y coma para finalizar sentencias, las variables y atributos se declaran con el tipo primero, y utilizamos llaves para delimitar bloques de código.

La primera sentencia, package consultorio; define que la clase que se está por definir pertenece al paquete consultorio. Todas las clases deben pertenecer a algún paquete, si intentamos definir una clase en Eclipse sin haber declarado un paquete antes, creará uno por defecto.

A continuación viene la declaración de la clase propiamente dicha. Comienza con la sentencia public class Paciente, public para que la clase sea visible fuera del paquete consultorio. Si no existiera el modificador public, la clase sólo sería accesible dentro del paquete al que pertenece. Como planeamos accederla desde el paquete test, necesitamos que sea visible fuera de consultorio.

A continuación declaramos los atributos: primero la visibilidad, luego el tipo de datos y luego el nombre del atributo. Cuidado con el modificador de visibilidad private: Java nos permite declarar un atributo como public, pero eso permitiría que se pueda modificar el contenido del atributo desde fuera de la clase y de ese modo, vulnerar el encapsulamiento de los atributos. Aparte: Observen como String se escribe con mayúscula: Eso se debe a que no se trata de un tipo de datos primitivo, sino de clase.

Todo muy bien, pero se preguntarán: ¿Qué quiere decir encapsulamiento? El encapsulamiento es una de las características del paradigma de objetos: Decimos que las clases encapsulan sus atributos porque controlan el acceso a los mismos. Un atributo no debe poder modificarse sin que la clase lo sepa y pueda realizar validaciones antes de permitir la modificación. También es posible que un atributo dependa del valor de otro y que la clase necesite enterarse de cuando se modifica el segundo para modificar al primero en consecuencia. Si bien Java nos permite declarar un atributo como public, nosotros siempre los declararemos como private, ya que nos atenemos al paradigma a pesar de estas particularidades de Java.

Aparte: Este es un buen ejemplo de la disciplina que debe tener un programador: No siempre es válido hacer lo que el lenguaje nos permite, y no es una buena idea aprender un paradigma de programación a partir del lenguaje de programación que lo implementa. Es mejor al revés, ya que éstas cosas suelen ocurrir. También es una buena idea averiguar cuales son las prácticas aceptadas por la comunidad del lenguaje que estemos utilizando. Así podremos enterarnos de buenas prácticas, soluciones aceptadas a problemas comunes y estilos de codificación aceptados. Hay que tener en mente que el código que escribimos hoy podría tener que ser mantenido por otro y viceversa, y si todos programamos según las mismas reglas, el código resulta más fácil de comprender y cambiar. Ya bastante complicado es programar como para que lo compliquemos aún más programando cada uno con su propio estilo.

Sigamos: Como declaramos los atributos como private necesitamos algún mecanismo para cargarles los valores que correspondan a cada instancia. Hay dos mecanismos para ello:

```
public Paciente(String nombre, String apellido, float estatura,
        float peso) {
    this.nombre = nombre;
    this.apellido = apellido;
    this.estatura = estatura;
    this.peso = peso;
}
```

Este método que declaramos, que tiene el mismo nombre de la clase y que por ello es el único método cuyo nombre comienza con mayúscula se denomina **constructor** de la clase. Es el método que se invoca al crear una nueva instancia de la clase. Siempre es una buena idea que el constructor inicialice la instancia completamente, para que no nos queden instancias de la clase sin inicializar (un paciente sin nombre y apellido, por ejemplo). Utilizar un constructor que reciba como argumentos los valores para los atributos de la clase nos lo asegura.

¿Qué ocurre dentro de este método? Simple: Se asignan los valores que nos llegan en los parámetros del método a los atributos de la clase. La forma de diferenciarlos es utilizando la referencia **this.** Esta es una referencia al objeto que estamos creando; la sintaxis this.nombre significa que nos estamos refiriendo al valor que tiene el atributo nombre en el objeto que estamos creando. Si utilizamos nombre sin la referencia this, nos estamos refiriendo al valor local, es decir, al que viene como parámetro. Por eso this.nombre = nombre; significa “Asigne el valor del parámetro nombre al atributo nombre de la clase”.

Alguien se puede preguntar (esperamos que alguien se lo pregunte) *¿qué ocurre si queremos cambiar el valor de algún atributo de un objeto?* (el peso de un paciente puede variar de una consulta a otra, por ejemplo), tarea que resulta imposible porque los atributos son privados y no podemos accederlos desde afuera. Dijimos que los atributos sólo pueden modificarse bajo el control de la clase; para ello existen

métodos especiales denominados accesores (y éste es el segundo método), que sirven para obtener (to get en inglés) como para asignar (to set) el valor de un atributo. A los primeros se los denomina getters y a los segundos setters:

```
public float getPeso() {
        return peso;
}
public void setPeso(float peso) {
        this.peso = peso;
}
```

Existe un par de accesores por cada atributo que deba accederse desde fuera y por convención se los denomina get[Atributo] y set[Atributo]. getPeso devuelve el atributo peso del objeto. No usamos el modificador this porque no existe ambigüedad con respecto a qué peso estamos refiriéndonos: en el ámbito del método getPeso sólo existe una entidad denominada peso y es el atributo de la clase, ya que los atributos de una clase son visibles para todos los métodos de la misma.

El ámbito de un método define la visibilidad y duración de las variables que declaremos en el mismo. Así, si declaramos una variable como en el método traerNombreCompleto:

```
public String traerNombreCompleto(){
        String resultado;
        resultado= nombre+" "+apellido;
        return resultado;
}
```

La variable resultado, de tipo String, sólo es visible en el ámbito del método. Para sacar su valor y utilizarlo fuera del método lo devolvemos con la sentencia return. Esta sentencia termina la ejecución del método y devuelve al llamador el valor de su argumento, en nuestro caso el de la variable resultado.

Hay varias construcciones que definen ámbitos: La declaración de una clase, la de un método y en general cualquier bloque de código (un bloque de código es lo que hay entre dos llaves). Así, los atributos de la clase están definidos en el ámbito de la misma, lo mismo que sus métodos y es por ello que los primeros son visibles en el ámbito de los segundos

En el caso del método setPeso, recibe como argumento un valor del tipo float (punto flotante simple precisión) y lo asigna al atributo peso. Si quisiéramos hacer una validación del valor recibido, éste sería el lugar para hacerla (por ejemplo, que no sea mayor o menor que ciertos límites). Vemos entonces que así es como la clase tiene control sobre la asignación de los atributos.

Vamos ahora a hacer algo con nuestra clase. Para ello, completamos el código de la clase TestConsultorio de la siguiente manera:

```
package test;

import consultorio.Paciente;

public class TestConsultorio {

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

        Paciente paciente1 = new Paciente("José", "Pérez",1.80f,85);
        Paciente paciente2 = new Paciente("Jorge", "Fernández",1.60f,90);

        System.out.println("Pacientes:");
        System.out.println(paciente1.traerNombreCompleto());
        System.out.println(paciente2.traerNombreCompleto());
   }
  }
```

La línea // TODO Auto-generated method stub es un comentario. Los comentarios son similares a los de C: con doble barra para los comentarios de una línea y con /* … */ para los de bloque. Ese comentario lo agregó Eclipse al generar la clase, y es para avisarnos que hace falta completar el método. Se puede ver una lista de todos los TODO (to do, para hacer) seleccionando Window → Show View → Tasks del menú principal. La pestaña de las tareas aparecerá en la misma ventana en la que se encuentra la consola.

Algunas cosas para notar: La sentencia import consultorio.Paciente;
importa la definición de la clase Paciente, definida en el paquete consultorio al paquete en el que nos encontramos (test), de lo contrario no podremos utilizarla.

La sentencia public static void main(String[] args) declara el método main. Indicamos con void que el método no devolverá ningún valor, es decir, se comporta como un procedimiento. El modificador public como ya vimos, indica que el método es visible (es decir que se lo puede llamar) desde fuera de la clase y finalmente, static indica que no es necesario instanciar la clase para invocarlo. Esto quiere decir que podemos invocar al método sin tener un objeto de la clase.

Las sentencias

```
Paciente paciente1 = new Paciente("José", "Pérez",1.80f,85);
Paciente paciente2 = new Paciente("Jorge", "Fernández",1.60f,90);
```

hacen varias cosas a la vez: por un lado, declaran dos variables de tipo Paciente, paciente1 y paciente2; y por el otro se encargan de instanciar objetos de la clase Paciente y asignarlos a las variables recién creadas. Para ello se emplea la sentencia new: ésta crea la instancia de la clase e invoca al constructor de la misma con los argumentos que le pasamos (nombre, apellido, altura y peso) inicializándola. El valor de la altura tiene una f detrás para indicarle a Java que se trata de un float literal, de lo contrario asumirá que es un double (punto flotante doble precisión) y nos dará un error porque el tipo del argumento que estamos pasando no corresponde al tipo declarado.

Las sentencias:

```
System.out.println("Pacientes:");
System.out.println(paciente1.traerNombreCompleto());
System.out.println(paciente2.traerNombreCompleto());
```

muestran una lista de los pacientes que tenemos definidos. Por el momento sólo diremos que System.out.println es similar a la sentencia printf de C o print o write de otros lenguajes: muestra su argumento por consola. Más adelante veremos el porqué de esta sintaxis.

Lo que ocurre aquí es que System.out.println recibe una llamada a un método de paciente1, invoca ese método, que devuelve un String, que System.out.println recibe como parámetro y lo muestra por pantalla. Lo mismo con paciente2. La primera sentencia System.out.println simplemente recibe un String y lo muestra.

Para correr nuestro programa, click en el botón verde con el símbolo de “play” en la barra de herramientas o hacemos click derecho sobre la clase TestConsultorio, elegimos la opción Run as/Java application y veremos en la ventana de la consola (inferior derecha en eclipse) la respuesta de nuestro programa:

Preguntas:

1) ¿Cómo se resuelve un problema desde el paradigma de objetos?
2) ¿Cuales son las ventajas que brinda Java?
3) ¿Qué es una clase? ¿En el 1° Programa Java, cúal es la diferencia entre la clase Paciente y TestConsultorio?
4) ¿Qué representa un atributo?
5) ¿Cúal es la visibilidad de los atributos?
6) ¿Cúal es la forma de acceder al valor de un atributo, desde otra clase?
7) ¿Qué funcionalidad nos brinda el constructor?
8) ¿Para qué utilizamos “this”?
9) ¿Para que se utilizan los métodos “setter”? ¿Cómo instanciar una clase sin utilizar el constructor?
10) ¿En el 1° Programa Java definimos un caso de uso, cúal es? ¿qué parámetros recibe? ¿qué resultado retorna?

Trabajo práctico: ampliando un poco nuestro sistema

Vamos a agregar un poco más de funcionalidad a nuestro sistema, ya que un consultorio que solo tiene pacientes es más una reunión de achacados que un consultorio. Para eso, vamos a incorporar una nueva clase: La clase Médico.

Para implementar esta clase van a trabajar un poco más ustedes: Les vamos a dar algunas pistas para que puedan resolverlo, pero la idea es que lo hagan ustedes.

La clase médico, por lo pronto, solo hace una cosa: Calcula el índice de masa corporal del paciente. El índice de masa corporal lo utilizan los médicos para evaluar si un paciente está excedido de peso o excesivamente delgado. Se calcula a partir del peso y la estatura del paciente y su fórmula es

$$imc = peso/estatura^2$$

El peso y la altura sabemos que son atributos del paciente. Por ello, el método calcularIMC recibe un Paciente como parámetro. Lo que el método debe hacer es:

1- Obtener el peso y la estatura del paciente.
2- Calcular el IMC con ambos aplicando la fórmula.
3- Devolverlo.

Con lo que llevamos visto hasta ahora, tienen todas las herramientas necesarias para implementarlo. Sólo necesitan conocer los operadores matemáticos básicos: +, -, * ,/. Ninguna sorpresa aquí. Para elevar al cuadrado por el momento hagan estatura*estatura. Van a necesitar declarar variables, operar, devolver resultados, instanciar clases y mostrar por consola. para concatenar strings y números se utiliza + (por ejemplo: “Edad: “+32 devolverá el String “Edad: 32”).

Vamos a generar, del mismo modo que con TestConsultorio, otra clase TestConsultorio2, en la que implementaremos la prueba correspondiente. La ejecución de TestConsultorio deberá mostrar por consola lo siguiente:

Visita 1:
Médico: Daniel López
Paciente José Pérez: IMC 26.23457
Paciente Jorge Fernández: IMC 35.156246

Luego en TestConsultorio2 modificar el peso para los dos pacientes utilizando el método set y volver a generar la siguiente vista por consola:
Visita 2:
Médico: Daniel López
Paciente José Pérez: IMC …….
Paciente Jorge Fernández: IMC …….

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);
```

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 es 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 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.

Trabajo Práctico

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) // para n y n1 positivos; n>n1 devuelve $Cn;n1$

TestNumero.java (paquete test)
Generar los resultados de los 10 métodos.

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
```

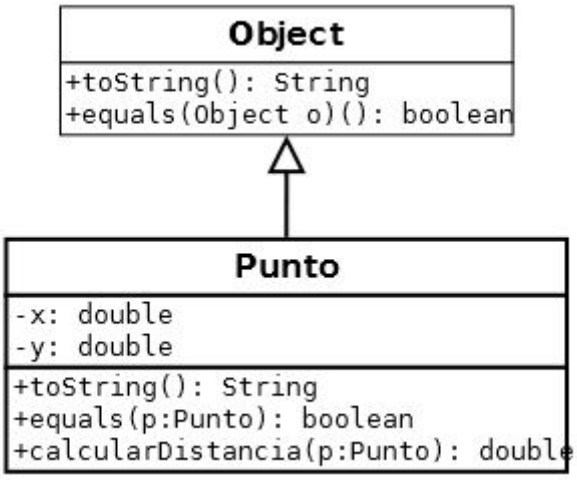
¿Porque las dos últimas impresiones no muestran ambas true? Esto ocurre porque paciente 1 y 2 tienen la misma identidad, es decir, son el mismo objeto: ocupan el mismo lugar en la memoria de la JVM. Sólo se creó el objeto una vez (con new), en paciente1 y luego se asignó a paciente2. En cambio, paciente3 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).

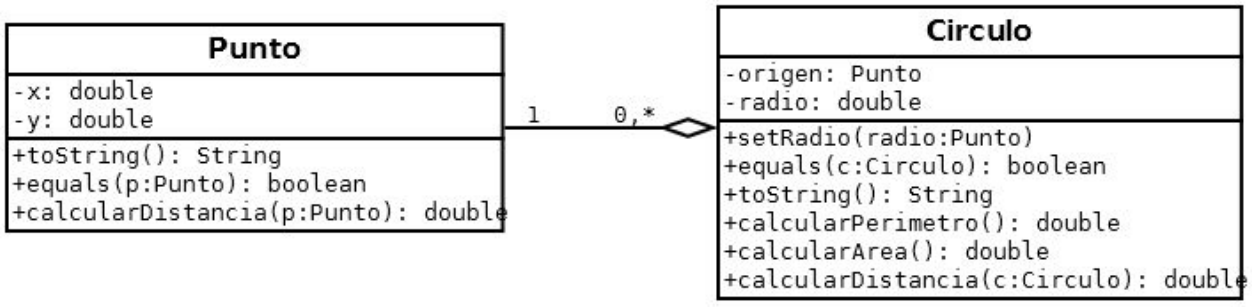
Implementación pendiente:

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.

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: En nuestro sistema de geometría podemos suponer que pueden existir 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 Circulo 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 ya 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. Es peligroso modificar parámetros del método y atributos de la clase. 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 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)

Trabajo Práctico

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

Preguntas:

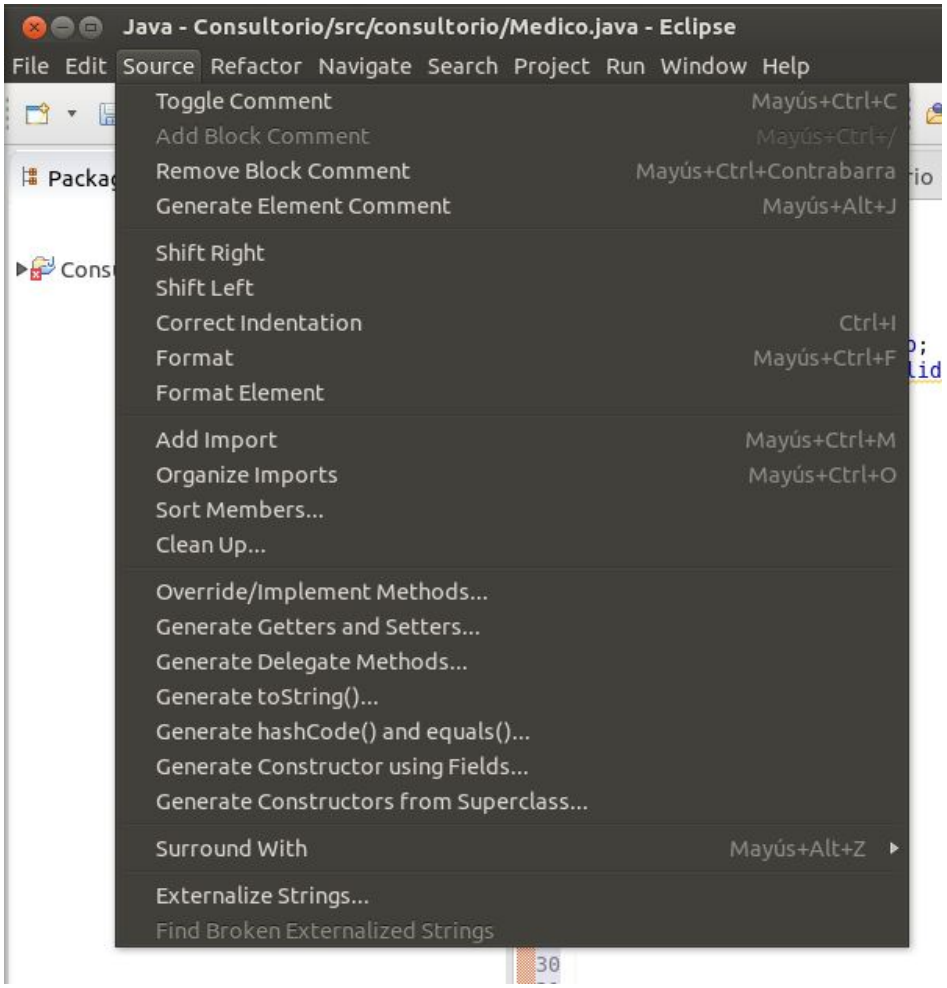
11) Cuando un clase encapsula otro objeto entre estas clases existe un relación de agregación o composición ¿Cuál es la diferencia entre estas relaciones?
12) ¿Qué es polimorfismo? ¿Para qué métodos hasta ahora aplicamos polimorfismo y para qué en cada caso?
13) ¿Cuándo un método está sobrecargado?
14) ¿Qué puede suceder si 2 objetos encapsulan el mismo objeto?
15) En un sistema en producción es conveniente modificar la “firma” de un método (su nombre, su tipo de retorno y sus parámetros)?

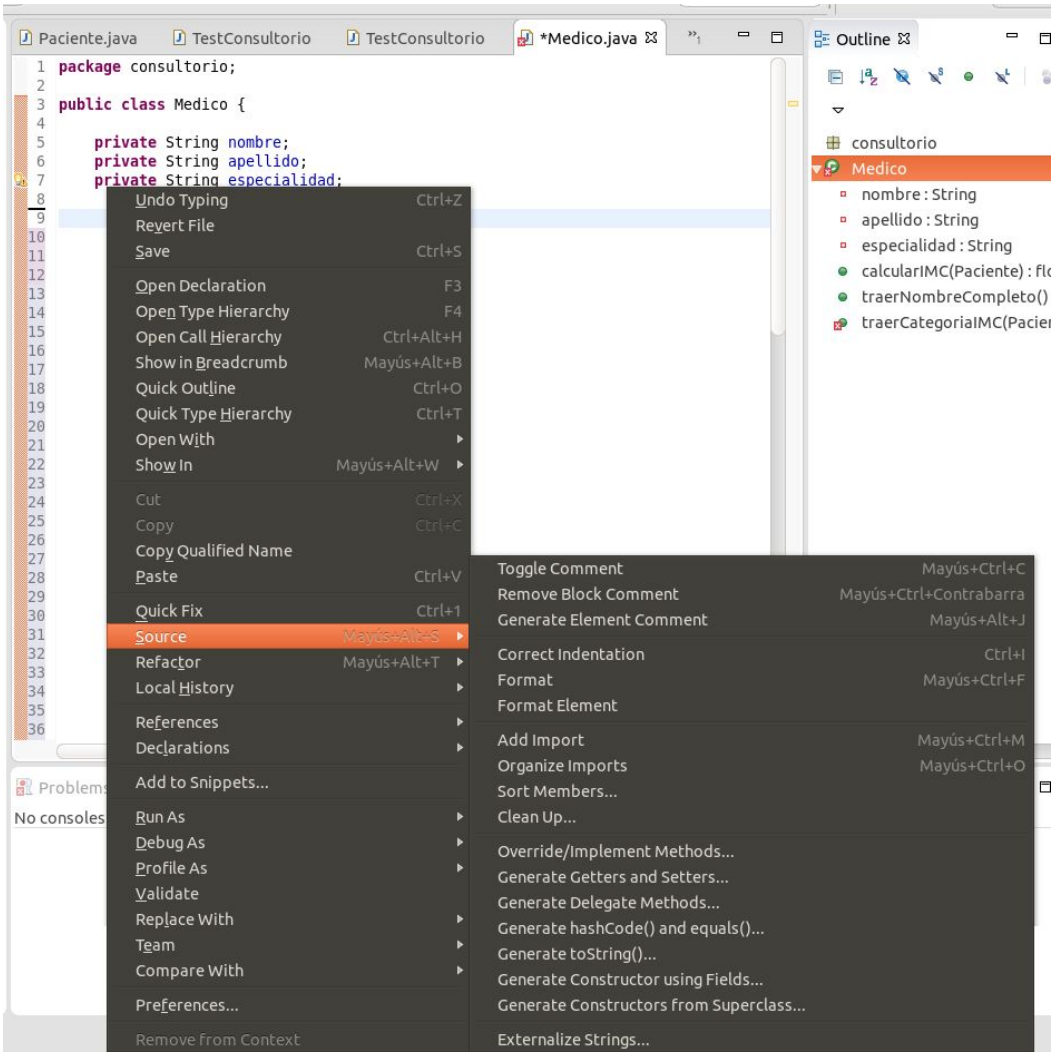
Utilizando Eclipse

Hasta ahora escribimos todo el código a mano y no es un mayor problema, ya que nuestras clases son 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. Sumemos los toString, los equals y cuando queramos acordarnos 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 mencionarles 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 ocurrirnos 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.

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}) con la segunda fila y la última con 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.

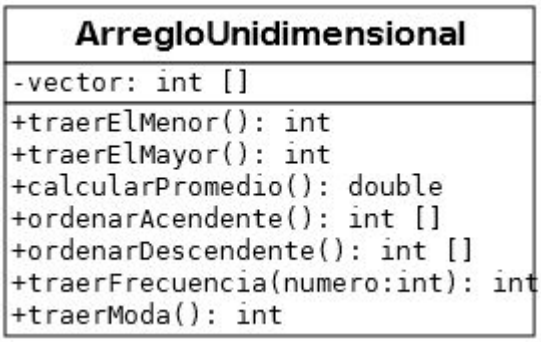
Las pueden consultar en:

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

Trabajo Práctico

Problema 1
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:

ArregloUnidimensionalTest.java

Testear el comportamiento de los casos de uso de la clase ArregloUnidimensional

Problema 2

Modelo:

ArregloBiderecional
-matrizA: double [] []
+sumar(matrizB:double [] []): double [] [] +restar(matrizB:double [] []): double [] [] +transpuesta(): double [] [] +multiplicar(numero:double): double [] [] +multiplicar(matrizB:double [] []): double [] []

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

+ 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 [][]

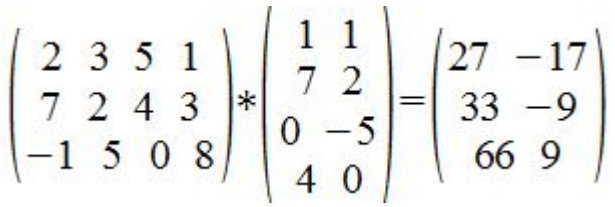
Producto de un escalar por una matriz: $K * (a_{ij}) = (K * a_{ij})$

+ 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:



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

Test:

ArregloBidimensionalTest.java

Testear el comportamiento de los casos de uso de la clase ArregloBidimensional