Programación I

Aprender programación
orientada a objetos desde cero

Dra. Ing. Inés Friss de Kereki
Catedrática de Programación

Material completo del Curso de **Programación I** Escuela de Ingeniería

UNIVERSIDAD ORT
Uruguay
Facultad de Ingeniería
Bernard Wand - Polak

Universidad ORT Uruguay
Facultad de Ingeniería
Bernard Wand - Polak

PROGRAMACIÓN I

Aprender programación orientada a objetos desde cero

Dra. Ing. Inés Friss de Kereki
Catedrática de Programación
kereki_i@ort.edu.uy
3era. edición - 2011

ISBN 978-9974-7704-5-4

Gráfica Don Bosco
Depósito Legal:

Julio 2011

Contenido

Indice de Ilustraciones

Objetivo del material

Este material, resultado de más de 10 años de experiencia en la enseñanza inicial de la Programación Orientada a Objetos en la Universidad ORT Uruguay, está dirigido a estudiantes universitarios sin conocimientos previos de Programación. Tiene por objetivo brindar un material completo y concreto para el aprendizaje inicial de la Programación basada en el paradigma de objetos. Se utiliza Java como lenguaje de aplicación. Los conceptos se presentan a medida que se van precisando para el avance del curso. En general se realiza una introducción al concepto específico y más adelante se profundiza y, o, completa. Los temas a tratar en el curso incluyen: conceptos básicos de programación, conceptos básicos de programación orientada a objetos, resolución de problemas simples y su implementación en un lenguaje orientado a objetos.

El libro se organiza por “semanas”. Cada una corresponde a una semana de un curso normal de programación. Un curso normal consta de 15 semanas, se dictan 4 horas de clase teórica en un salón de clase y 2 horas más de práctica en un laboratorio. Las primeras semanas son de mayor contenido teórico. A medida que avanza el curso se hace más énfasis en los contenidos prácticos. Se incluyen multiplicidad de ejercicios, prácticos y cuestionarios de autoevaluación. Varios ejemplos y ejercicios presentados han sido adaptados de los textos referidos en la bibliografía.

En las primeras semanas, se presentan los conceptos básicos de programación y su implementación en Java. El objetivo es permitir un rápido acercamiento a la programación real y efectiva en máquina y enfrentarse en forma temprana a problemas típicos como guardar un programa fuente, corregir errores de compilación o ajustar detalles de ejecución.

Luego de superada esta etapa inicial, se presentan los conceptos propios de la orientación a objetos. Inicialmente se muestran ejemplos amplios cuya finalidad es "mostrar el panorama", para luego concentrarse en cada uno de sus componentes. Así, posteriormente se presenta en forma detallada el concepto de objeto, clase, atributo, método y más adelante se incluyen relaciones como asociación o generalización-especialización. Finalmente, se introducen diversos tipos de colecciones. Al final del curso se está capacitado para resolver problemas en forma satisfactoria en dominios sencillos. En este primer semestre la interfaz con el usuario se realiza en estilo consola. No se incluye el manejo de ventanas a los efectos de no incorporar complejidad innecesariamente.

El curso de Programación I se continúa en segundo semestre con Programación II, donde los desafíos son modelar e implementar problemas en dominios de mayor amplitud y complejidad. El curso de Programación II tiene entre sus temas: metodologías para análisis y diseño de problemas orientados a objetos (casos de uso, metodología CRH: clases, relaciones y *helpers*), concepto de separación de dominio e interfaz, manejo de interfaz estilo *Windows*, nociones de patrones (patrón *Observer*), conceptos de persistencia, manejo básico de archivos y profundización en algoritmia.

En la carrera de Ingeniería en Sistemas en la Universidad ORT posteriormente se incluyen 2 cursos de Estructuras de Datos y Algoritmos, donde se estudia toda la problemática del análisis de algoritmos y de las estructuras de datos y se realiza la implementación de las soluciones en C++. Continuando la línea de Programación orientada a objetos, se dictan luego 2 cursos de Diseño, donde se estudian diversas herramientas de diseño y patrones.

Se agradece especialmente la colaboración de los docentes Carlos Berrutti, Andrés de Sosa, Alejandro Fernández, Nicolás Fornaro, Gonzalo Guadalupe, Mariana Lasarte, Pablo Martínez,

Santiago Matalonga, Claudio Montañés, Gastón Mousqués, Agustín Napoleone, Matías Núñez, Pablo Píriz, Michel Rener, Marcelo Rubino, Juan José Silveira, Alvaro Tasistro y a todos los ayudantes de Cátedra y demás docentes que han participado en la elaboración y revisión de los diversos materiales.

En la segunda edición se agregaron ejercicios y ejemplos. En esta tercera edición se realizaron ajustes y modificaciones menores.

Semana 1

1.1 Introducción

El objetivo de este material es iniciar la enseñanza de la programación utilizando técnicas de programación orientada a objetos y desarrollar aplicaciones en Java para poner en práctica los conceptos presentados.

A efectos de introducción, inicialmente se brindan conceptos sobre los diferentes roles de los profesionales vinculados al área de Programación así como conceptos básicos.

1.2 Rol de los Profesionales de Sistemas

Ingeniero de Sistemas, Licenciado en Sistemas, Programador... muchas veces no está claro el rol de cada uno de estos profesionales. Comenzaremos por describir en términos generales y a través de un ejemplo simplificado el rol del *licenciado* en Sistemas. Por ejemplo, una empresa le plantea al licenciado que desea "instalar computadoras para facturar". El licenciado actúa de intermediario entre la empresa y el software, realiza entrevistas con las distintas personas relacionadas con ese proceso, analiza y modela la realidad.

Tiene como reglas ser objetivo, diplomático, verificar toda la información y asegurarse de tener la mayor cantidad de detalle. En función de este análisis realiza un diseño preliminar y elabora un informe o propuesta.

Si la empresa está de acuerdo con la propuesta, el licenciado describe y asigna los programas a los *programadores*, quienes escribirán los programas. Una vez que están prontos esos programas, se pone en marcha el sistema. Esto se puede hacer de varias formas, por ejemplo en paralelo, de golpe, o en etapas. En *paralelo* quiere decir que se pondrá en funcionamiento el nuevo sistema pero también se realizará simultáneamente el trabajo en la forma anterior. De *golpe* refiere a que se sustituye completamente el sistema manual por el nuevo sistema. En *etapas* quiere decir que, por ejemplo, en primera instancia se pondrá en marcha la facturación contado, en otra etapa posterior la facturación crédito.

En general un licenciado es capaz de:

- insertarse en el mercado nacional y regional, como consultores en computación, directores de proyectos o gerentes de sistemas;
- identificar requerimientos organizacionales e informáticos que aumenten la competitividad y productividad de empresas y organizaciones;
- intervenir activamente en planes de reconversión en industrias y organizaciones;
- analizar, diseñar e implementar sistemas informáticos, reestructuras organizacionales y planes de calidad; y
- trabajar en equipos multidisciplinarios con otros profesionales y desempeñarse en tareas de investigación.

El perfil del Ingeniero de Sistemas está más orientado al análisis y desarrollo de sistemas grandes y además, la formación matemática que recibe le permite vincularse fácilmente con otros ingenieros. Un ingeniero de sistemas es capaz de:

- diseñar y desarrollar sistemas de alta escala y complejidad
- desempeñarse como desarrollador de software, líder de proyectos o gerente de sistemas;

- insertarse en el mercado regional e internacional;
- adaptarse al constante cambio; e
- integrarse con otros profesionales.

1.3 Definiciones Básicas

Hay varios términos que frecuentemente se usan en computación, como por ejemplo: sistema, *hardware* y *software*. Brindaremos definiciones básicas y concretas de los mismos.

1.3.1 Sistema

Hay varias definiciones posibles de sistema. Algunas de ellas tomadas de diccionarios indican:

- Conjunto u ordenación de cosas relacionadas de tal manera que forman una unidad o un todo.
- Conjunto de hechos, principios, reglas, etc. clasificadas y ordenadas de tal manera que muestran un plan lógico.
- Una forma establecida de hacer algo.

En computación tomaremos como definición de sistema: conjunto u ordenación de elementos organizados para llevar a cabo algún método, procedimiento o control mediante procesamiento de la información.

1.3.2 Hardware

Hardware refiere a todos los elementos físicos de una computadora. Se incluyen los discos, los circuitos, cables, tarjetas, monitor, teclado, *mouse*, etc.

Se puede clasificar en hardware fundamental y accesorio. El fundamental es el imprescindible para que funcione la computadora, en general se refiere a monitor, unidad de proceso y teclado. Dentro del accesorio se podría incluir por ejemplo una cámara de video o un escaner.

También se pueden clasificar como dispositivos de entrada (*mouse*, teclado, micrófono), de salida (impresora, monitor) o de entrada salida (*modem*, *diskette*, *cd-rom*, *pendrives*, disco duro, etc.).

1.3.3 Software

Comprende los programas. Un programa puede ser de uso general, como un procesador de texto o una planilla electrónica o de uso específico, por ejemplo un sistema de cálculo de sueldos. Otra clasificación lo divide en software de base (incluye lo necesario para que funcione el hardware) y software de aplicación.

1.3.4 Programar

Es el arte de dar comandos a algo o alguien que pueden ser ejecutados después. Se puede especificar de distintas formas: botones, mapa, lista. Un comando es una orden para algo o alguien para realizar determinada acción. Puede ser oral (“levántate”, “acomoda”) o manual (presionar un botón).

1.3.5 Diseñar

Es el acto de organizar los comandos. Así como un novelista probablemente antes de escribir un libro estructure que temas tratará y, o, elabore un índice, en forma similar antes de programar se diseña, "se hacen los planos" del software.

1.3.6 Computador

Es una máquina capaz de almacenar y ejecutar comandos. Tiene unidad central de proceso, disco, teclado, monitor, etc.

1.3.7 Máquina

Cada problema es representable por un conjunto de instrucciones tal que pueden ser ejecutadas por una máquina o persona que tenga como único conocimiento el poder de ejecución de esas instrucciones. El conjunto de instrucciones está definido de antemano.

1.3.8 Sistema operativo

Son programas que vinculan el hardware con el resto del software. Ejemplos son: Windows, Linux, DOS.

1.4 Componentes de un sistema de computación

En un sistema de computación no están solamente los programas. El sistema está formado por:

- Software;
- Hardware;
- Gente;
- Datos;
- Documentación: manuales, impresos; y
- Procedimientos: pasos que definen cómo usar cada elemento.

1.5 Características del software

El software presenta algunas características interesantes:

- Se desarrolla, no se fabrica en sentido clásico.
- No se estropea a diferencia del hardware.
- Es complejo: los dominios son complejos, las aplicaciones son complejas. El software modela la realidad y la realidad en general es compleja. Por ello se hacen abstracciones, así como un mapa en geografía toma los principales elementos relevantes para su objetivo, lo mismo se hace en computación.

1.6 Ingeniería de software

Es la aplicación de un enfoque sistemático, disciplinado y cuantificable hacia el desarrollo, operación y mantenimiento del software. Tiene un proceso (marco de trabajo), métodos (cómo construir software) y herramientas (dan soporte a proceso y métodos).

1.7 Ciclo de vida del software

Habitualmente, el proceso que se sigue para desarrollar software es: análisis o relevamiento de requisitos, diseño, implementación, prueba y mantenimiento. Se presenta gráficamente en la Ilustración 1 Ciclo de vida del software.

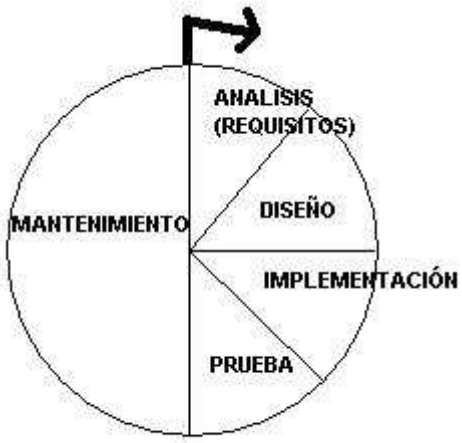


Ilustración 1 Ciclo de vida del software

Durante el *análisis o especificación de requisitos* se trata de determinar claramente cuáles son los requisitos que debe satisfacer el sistema, qué características debe tener. En el *diseño* se realizan los "planos" del sistema. En la *implementación* se desarrolla efectivamente el software, se escriben los programas. Durante la *prueba* se trata de eliminar los errores. Existen dos tipos de pruebas: alfa y beta. La prueba alfa habitualmente se hace dentro del equipo de desarrollo. Otro programador verifica los programas. En la prueba denominada beta se les pide colaboración a los clientes. Se les entrega en forma adelantada el software a cambio de que colaboren en su verificación. Durante el *mantenimiento* se hacen los ajustes del software debido a errores no detectados antes, cambios en la realidad (por ejemplo, creación de un nuevo impuesto sobre la facturación) y, o ampliaciones al sistema original.

1.8 Lenguajes de Programación

Nos detendremos en la escritura de programas. Inicialmente los programas se escribían en *lenguaje de máquina.* Este lenguaje es una combinación de 0 y 1 (el 0 representa que no hay corriente, el 1 que sí hay corriente) y sirvió para mostrar que era posible escribir programas, pero el nivel de abstracción de cada instrucción es muy bajo.

Por esta razón surgieron lenguajes del tipo "Assembler", en el cual por ejemplo se escribe "LOAD X", que correspondería a la instrucción de máquina 001100101011. Debido a que las máquinas únicamente entienden el lenguaje de 0 y 1, es necesario hacer la traducción a ese lenguaje. Este proceso de traducción de lenguajes tipo "Assembler" a lenguaje de máquina se realizar por un programa llamado Ensamblador. En general, la correspondencia de sentencias es 1 a 1, una de lenguaje assembler corresponde a una de máquina. Tanto el lenguaje de máquina como el lenguaje "Assembler" se denominan de *bajo nivel*, refiriéndose a bajo nivel de abstracción. Cada sentencia representa "poco".

Posteriormente surgieron los lenguajes de *alto nivel* de abstracción, en los cuales una sentencia representa muchas de máquina. Sigue siendo necesario realizar la traducción de ese lenguaje al de máquina. Este proceso se puede hacer de dos formas: compilar e interpretar. En el caso de los

lenguajes compilados, el compilador (programa que realiza esa traducción), toma todo el programa y lo traduce al lenguaje de máquina. Posteriormente se ejecutará. El intérprete toma cada línea del programa, la traduce y ejecuta, una a una. Es decisión de diseño del lenguaje si será interpretado o compilado.

Ejemplo:

(sentencia en alto nivel)		(en lenguaje de máquina)
SUM 1 TO TOTAL	----- compilador/intérprete ----	010001.....
		11110101...

Hay gran cantidad de lenguajes de programación: Java, Smalltalk, Pascal, Delphi, C, C++, Modula, Ada, COBOL, Fortran, Basic, Visual Basic.NET (VB.NET), etc., etc., etc. Cada uno de ellos está orientado a distintos fines. La elección de un lenguaje de programación depende de muchos factores, como por ejemplo disponibilidad, finalidad, costo y, o, facilidad. Java es un excelente lenguaje para comenzar el aprendizaje de la programación y aplicar los conceptos de programación orientada a objetos.

1.9 Resolución de problemas

Se verá una forma de pensar y resolver problemas. En primera instancia, se definirá "problema complejo".

1.9.1 Problema Complejo

Es un problema cuya solución no es inmediata o evidente.

Supongamos que se desea construir una casa. El arquitecto probablemente no se fije en los detalles chicos, como por ejemplo si el placard del dormitorio tendrá 3 ó 4 cajones. Seguramente dividirá el problema complejo de construir una casa por ejemplo en albañilería, eléctrica, sanitaria, etc. O sea, enfrentado a problema complejo la solución es dividirlo en más chicos y luego combinar sus soluciones. ¿Hasta cuándo continuar la división? Hasta que la solución sea evidente. Esta estrategia se denomina **dividir para conquistar.**

Como ejemplo, consideramos la planificación del próximo trimestre. Inicialmente se puede considerar un problema complejo: no se sabe cómo resolverlo. Una posible división es por meses.

1.9.2 Niveles

1.9.2.1 Primer nivel

mayo – tareas habituales
junio – viaje a Río
julio – examen de inglés

Esta división se llama de *Primer Nivel*, indica qué cosas hacer. Cada una de esas tareas puede considerarse como un problema complejo (que será necesario dividir) o como un problema simple, ya resuelto. Dependerá de cada uno que interpretación tendrá. Tener en cuenta que esta dualidad ocurre pues el problema planteado es muy general, cuando se desarrollan sistemas reales la división es más clara.

1.9.2.2 Segundo nivel

Si se considera el viaje a Río como un problema complejo, será necesario dividirlo en subproblemas. Se presenta así en el siguiente punto.

1.9.2.2.1 Viaje a Río

```
ir al Aeropuerto Internacional de Carrasco el 7 de junio hora 8
tomar el avión de 9 hs a Rio
llegar a Río 12 hs.
ir al hotel Columbia
registrarse
....
```

Se podría ampliar más el detalle. Cada uno de esos nuevos problemas también es pasible de ampliación o división. Por ejemplo, el problema de tomar el avión de las 9 hs a Río podría dividirse.

1.9.2.2.2 Tomar avión de las 9 hs a Río

Este proceso involucra el despacho de valijas y el embarque en sí.

```
si tengo valijas
        despachar valijas
embarcar
subir al avión
desayunar en el avión
```

1.9.2.2.3 Despachar valijas

A su vez, despachar valijas podría detallarse en:

```
mientras quede alguna valija
        tomar una valija
        pesarla
        ponerle el sticker
        despacharla
si hay sobrepeso
        pagar recargo
```

1.9.2.2.4 Desayunar

Un caso similar ocurre con Desayunar:

```
si hay buen tiempo y tengo mucha hambre
        pedir jugo y sandwiches
en otro caso
        pedir sólo café
```

Notar que la división no es standard, otra persona podría haber hecho una división diferente del problema.

Todos estos problemas (tomar el avión, despachar valijas, desayunar) corresponden al *segundo nivel*, que explica cómo hacer las cosas.

En resumen, la estrategia para resolver un problema complejo es la de dividir para conquistar: dividir el problema en subproblemas, resolver cada uno de ellos y combinar sus soluciones para resolver el problema original.

1.9.3 Seudocódigo

En el ejemplo utilizamos **seudocódigo.** No es código estricto que pueda ser utilizado por ninguna computadora, es una versión similar a código de un lenguaje. Es una forma de expresar qué cosas hay que hacer y cómo.

1.9.4 Estructuras de control

El flujo es el orden en que se ejecutan las instrucciones. Analizando más en detalle el ejemplo, se puede observar que se utilizan 3 estructuras de control de flujo del programa: secuencia, decisión e iteración.

1.9.4.1 Secuencia

La secuencia aparece en:

```
ir al Aeropuerto Internacional de Carrasco el 7 de junio hora 8
tomar el avión de 9 hs a Río
llegar a Río 12 hs.
ir al hotel Columbia
```

Es una lista de pasos, uno tras otro. Genéricamente se ve:

```
.....
....
....
```

1.9.4.2 Decisión

La decisión se puede ver en:

```
si tengo valijas
        despachar valijas
```

```
si hay buen tiempo y tengo mucha hambre
        pedir jugo y waffles
en otro caso
        pedir sólo café
```

El flujo de la ejecución se altera en función de una decisión. Los formatos posibles son:

a) si (condición)
        .....

b) si (condición)

```
        .....
en otro caso
        ......
```

En inglés es "*if* (si) ..." o "*if* (si) ....*else* (en otro caso)..."

1.9.4.3 Iteración

La iteración aparece en:

```
mientras quede alguna valija
        tomar una
        pesarla
        ponerle el sticker
        despacharla
```

El formato es:

```
mientras (condición)
        .....
```

En inglés se denomina "*while* (mientras)".

Esta estructura se utiliza cuando se desea repetir un proceso. Hay más versiones de la iteración:

```
repetir X cantidad de veces
        .....
```

Esta estructura se llama en inglés "*for*". Se utiliza cuando se conoce la cantidad de veces que se desea repetir.

Otra estructura de iteración es la "repetir".

```
repetir
        ......
hasta (condición)
```

El nombre en inglés de esta última estructura es "*repeat.... until*" (repetir .... hasta). La diferencia entre "repetir" y "mientras" es que el bloque dentro del mientras se ejecutará 0, 1, o más veces; en el repetir se ejecutará como mínimo 1 vez.

En los "...." puede ir cualquier otra sentencia, inclusive otras estructuras repetitivas, condiciones o secuencias.

No es necesario identificar explícitamente cada estructura de control cuando se realiza el programa, simplemente se utilizan.

1.9.5 Ejercicios básicos en seudocódigo

1.9.5.1 Semáforo

Se dispone de la siguiente esquina (ver Ilustración 2 Semáforo). Se desea cruzar de A a D respetando los semáforos. Para simplificar, considerar que únicamente hay luz verde y roja. Anotar en seudocódigo la lista de pasos necesarios para describir el proceso.

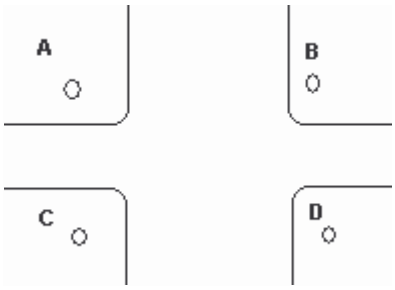


Ilustración 2 Semáforo

Una posible solución es:

```
mirar a C
si está el semáforo en rojo
        mirar a B
cruzar
mirar a D
mientras esté rojo el semáforo
        esperar
cruzar
```

1.9.5.2 Cocinar panqueques

Se tiene un recipiente con la preparación pronta para panqueques, se desea cocinarla. Anotar los pasos en seudocódigo que describan el proceso.
Una posible solución es:

```
prender fuego
calentar sartén
mientras quede preparación
        poner un poco de manteca en la sartén
        poner una cucharada de preparación
        cocinar 30''
        dar vuelta
        cocinar 30''
        retirarlo
apagar el fuego
limpiar la cocina
```

1.9.5.3 Cambiar una rueda de un auto

Anotar los pasos necesarios para explicar el proceso de cambiar una rueda pinchada del auto.
Una posible solución es:

```
si tengo tiempo, ganas, sé hacerlo y tengo todos los materiales
        sacar los elementos de la valija
        aflojar las tuercas
        poner el gato hidráulico
        subir el auto
        sacar las tuercas
        sacar la rueda
        poner la nueva rueda
        poner las tuercas
        bajar el auto
```

```
        sacar el gato hidráulico
        apretar las tuercas
        guardar todo
en otro caso
        llamar al Automóvil Club y esperar que ellos la cambien
```

1.9.5.4 Jugar al Ahorcado

Representar los pasos necesarios para jugar al Ahorcado (Ilustración 3 Ahorcado). En el Ahorcado un jugador piensa una palabra de un tema o pista dada, la representa con líneas (una por cada letra). El otro jugador va diciendo letras. Si la letra está en la palabra, se sustituye la línea por la letra. Si no está en la palabra, se agrega una componente más al muñequito que será ahorcado. El juego finaliza cuando se adivina la palabra o se completó el muñequito.

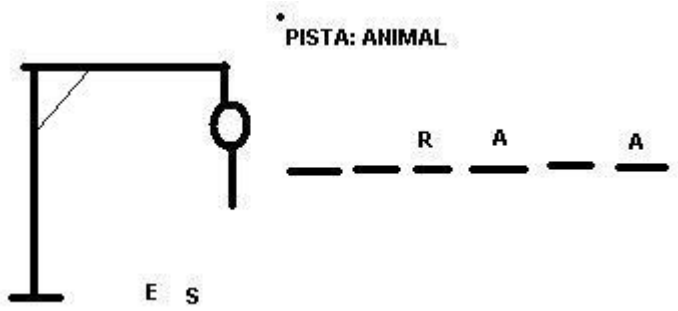


Ilustración 3 Ahorcado

Una posible solución es:

```
repetir
        repetir
                pienso letra
        hasta encontrar una que falte
        la digo
        si está
                la anoto
        en otro caso
                hacer una rayita
hasta que acerté o complete el macaco
si acerte
        mostrar GANE
en otro caso
        mostrar PERDI
```

1.10 Algoritmo

Un algoritmo es un conjunto de reglas que determinan una secuencia para resolver un tipo específico de problema.
Las propiedades de un algoritmo son:

- finito: como opuesto a infinito. Si es infinito, se denomina proceso computacional.
- entrada: el algoritmo tiene datos de entrada. Por ejemplo, en el caso del algoritmo de Euclides los datos de entrada son los dos números
- salida: el algoritmo tiene datos de salida. En el mismo ejemplo anterior, la salida es el MCD (máximo común divisor).

- efectivo: todas las operaciones son lo suficientemente básicas para que puedan ser realizadas por una persona en tiempo finito. Por ejemplo, si se debe dividir, indicar la cantidad de cifras a tomar.
- preciso: todos los pasos están bien detallados. En el caso de una receta de cocina, por ejemplo si se indica “dar un golpe de horno” no se cumpliría con la precisión, pues no se detalla si un golpe de horno es un breve momento, minutos u horas.

1.11 Ejercicios para resolver

Anotar el seudocódigo para cada uno de los siguientes problemas:

- realizar una llamada internacional con una tarjeta telefónica
- anotarse en la Universidad al curso de Programación I
- pintar un cuarto
- despachar una carta por correo certificado
- tirar una moneda, si sale cara estudio para el examen, si sale cruz me voy al cine y después estudio.
- dar las instrucciones para comprar una entrada a un concierto
- buscar una palabra en el diccionario
- un cliente solicita un préstamo a un banco
- reclamar en una empresa por una reparación para una computadora
- mandar un mensaje de texto por un celular

Semana 2

2.1 Introducción a Variables

A efectos de introducir el concepto de variable, consideremos el siguiente problema: se desea sumar 10 números con una calculadora sencilla, utilizando la memoria de la máquina. El proceso podría describirse así:

```
CM (Clear Memory, limpiar la memoria)
repetir 10 veces
        poner número
        apretar M+
RM (Recall Memory, mostrar la memoria)
```

¿Qué es la memoria de la calculadora? Es un lugar para almacenar un dato que luego puedo recuperar. Si en vez de una calculadora común se utilizara una científica -donde hay varias memorias- se diferenciaría cada memoria por su nombre: ejemplo M01, M02, ...

Esta idea de "memoria" se puede ampliar hacia el concepto de variable.

2.2 Variable

Una **variable** es un lugar de la memoria de la computadora para almacenar un dato. Tiene un nombre y es para un tipo de dato (ejemplo: es para un número entero, para un número con punto decimal, para texto, etc.). El nombre se recomienda que sea nemotécnico, o sea, que refleje su contenido. Así, si en una variable acumularemos el total de sueldos, es más claro ponerle a la variable como nombre "totalSueldos" que "XR86".

Se transcribirá el programa anterior a un lenguaje más parecido a los lenguajes de programación.
La sentencia "CM" se escribe: *M = 0* o *M ←0*. Se "asigna" el valor 0 a la variable M. Se lee de derecha a izquierda. M es la variable que representa el lugar de almacenamiento, el nombre "M" es el nombre elegido para la variable, como nenómico de Memoria. Se podría poner también más específicamente "acum", como nenómico de Acumulador. Así, *acum = 0.*

La sentencia "poner número" se escribe como "leer dato". Esa sentencia solicitará el ingreso de un dato por parte del usuario y se guardará automáticamente en la variable *dato*.

La sentencia M+ corresponde a: *M = M+ dato*. El valor de la derecha se asigna en la variable *M* más el dato que se ingrese. Para la variable *acum* se escribiría: *acum = acum + dato*.

RM corresponde a *mostrar M*, o sea, desplegar su valor por pantalla. Para este caso, se aplicaría *mostrar acum.*

Así, el mismo ejercicio, escrito utilizando variables quedaría:

```
acum = 0
repetir 10 veces
        leer dato
        acum = acum + dato
```

```
mostrar acum
```

acum es la variable donde se acumularán los valores. *dato* representa la variable donde se ingresará cada dato a sumar.

En forma más detallada, la estructura "repetir X veces" se puede indicar:

```
para (i= 1; i <= X; i = i + 1)
```

i representa la variable que controla la estructura. *i=1* indica que el valor inicial de la variable *i* es 1; *i<= X* es la condición de trabajo, mientras se cumpla esta condición se realizará el proceso; *i = i + 1* es el incremento que se realizará con cada nueva repetición. En este caso X corresponde al valor 10.

Así:

```
acum = 0
para (i= 1; i <= 10; i = i + 1)
        leer dato
        acum = acum + dato
mostrar acum
```

El mismo ejercicio hecho en vez de con la estructura de control "*for*" (para, repetir), hecho con "*while*" quedaría:

```
acum = 0
van = 0
mientras van <10
        leo dato
        acum = acum + dato
        van = van + 1
mostrar acum
```

Fue necesario agregar una variable donde se lleva la cuenta de cuántos números se han ingresado (variable van).

Si se desea con "*repeat*" (repetir), la implementación sería:

```
acum = 0
van = 0
repetir
        leo dato
        acum = acum + dato
        van = van + 1
hasta (van=10)
mostrar acum
```

2.3 Corrida a mano

Hacer la corrida a mano de un programa implica simular la ejecución del programa por parte de la máquina. Permite detectar posibles errores.

Tomemos por ejemplo el código anterior. ¿Pedirá 10 números? ¿Mostrará bien el total? Se hará la corrida a mano. Para simplificar, en vez de hacerlo con 10 números, se supondrá que hay solamente 2 números. Así, el programa sería:

```
acum = 0
van = 0
repetir
        leo dato
        acum = acum + dato
        van = van + 1
hasta (van=2)
mostrar acum
```

Para realizar la corrida se anotan las variables:
acum = 0, empieza en 0
van = 0, empieza en 0
Luego viene la sentencia "leo dato". La variable dato toma el valor que ingrese el usuario, por ejemplo 5. Se anota:
dato = 5
Después aparece la sentencia "acum = acum + dato".
La variable acum quedaría entonces en 5. Se representa así:
acum= 0 / 5
La barra vertical indica el nuevo valor.
La siguiente instrucción es "van = van + 1". Así: *van = 0 / 1*
Se verifica la condición de si van es 2, lo cual no es cierto, por lo cual sigue en la estructura repetitiva.
Se solicita un nuevo dato, el usuario ingresa el valor 8: *dato = 5 / 8*
Nuevamente, la siguiente instrucción es: "acum = acum + dato". Así: *acum = 0 / 5/ 13*
La próxima es "van = van + 1".
van = 0 / 1 / 2
Al chequearse la condición, se cumple. La siguiente línea es " mostrar acum", por lo que en pantalla aparecerá el valor 13.
Revisando el proceso, el programa debía pedir 2 datos y mostrar su suma; pidió dos valores (que fueron el 5 y 8) y mostró correctamente su suma (13). Se puede inferir que para el caso de 10 valores (donde se utilizará el valor 10 en lugar del modificado 2), el programa funcionará bien: pedirá 10 valores y mostrará su suma.

Es posible también hacer demostraciones matemáticas para determinar la corrección del programa. Se ve en cursos más avanzados.

2.4 Ejercicios básicos con variables

Realizar el seudocódigo para cada uno de los siguientes ejercicios.

1 – Ingresar 3 valores enteros y mostrar el mayor de ellos. Ej. si se lee: 1, -3, 5, mostrar: 5
2 – Recibir valores y sumarlos tomando en cuenta las siguientes convenciones:
Si se recibe un 0 se termina la suma y se devuelve el total;
Si se recibe un 2 se suma y luego se duplica el total acumulado; y
Si se recibe un número negativo se ignora.
3 – Recibir 4 valores y mostrarlos ordenados de menor a mayor. Ej. si se lee: 8, 5, 9, 1; se muestra: 1, 5, 8, 9
4 – Leer 5 valores y mostrar los siguientes resultados:
la suma de los dos primeros;
la multiplicación del 3ro y 4to;

el promedio de los 5; y
la suma de todos los resultados anteriores.

5- Leer un valor y mostrar el inverso del opuesto Ej. se lee: 8; se muestra: -1/8

6- Recibir 6 valores y mostrar su suma en valor absoluto. Ej. si se lee -1, 2,-3, 4,-5,6, mostrar 21

2.4.1 Producto

Leer 20 datos y mostrar el producto de esos 20 números. Una posible solución es:

```
prod = 1
van = 0
mientras van < 20
        leo dato
        prod = prod * dato
        van = van + 1
mostrar prod
```

Es similar al ejercicio de la suma visto antes, en este caso los cambios son: se utiliza multiplicación en vez de suma y se inicializa la variable *prod* en 1 (neutro del producto). Observar que también se ha modificado el nombre de la variable para reflejar que en este caso se realiza un producto, no una suma.

2.4.2 Valores repetidos

Leer 3 datos, se sabe que 2 son iguales y el tercero es diferente. Mostrar el valor repetido.
En este caso, una posible solución es:

```
        leer A, B, C
        si (A = B)
                mostrar repetido A
        en otro caso
                mostrar repetido C
```

2.4.3 Intercambio de variables

Un problema muy frecuente es el de intercambiar dos variables. Por ejemplo, se tiene una variable A que vale 20 y una variable B que vale 10. Se pide intercambiar el contenido de ambas variables (resolverlo en forma genérica, no en particular para los valores 10 y 20).

Así, la solución sería:

```
AUX = A
A = B
B = AUX
```

Para pensar: ¿es posible hacerlo sin utilizar una variable auxiliar?.

2.4.4 Promedio

Leer datos hasta que se ingrese el número 99 e imprimir el promedio.

```
total = 0
van = 0
ingresar dato
mientras (dato != 99)
        total = total + dato
        van = van + 1
        ingresar dato
si van > 0
        mostrar (total / van)
en otro caso
        mostrar "sin datos"
```

En este caso, es necesario llevar una variable donde se cuenta la cantidad de datos (van) y otra donde se acumulan todos los datos (total).

Notar que no se podría haber hecho fácilmente con una estructura de tipo "*repeat*".

2.4.5 Otros ejercicios de estructuras de control

1. Imprimir 10, 20, ..., 1000
2. Leer 50 datos, imprimir el promedio de los números pares mayores que 20.
3. Calcular las raíces de una ecuación de 2do grado. Para ello, ingresar los coeficientes y realizar los cálculos necesarios para indicar si la ecuación tiene 2 raíces reales diferentes, una raíz real doble o no tiene raíces reales.
4. Generar y mostrar la secuencia: 0, 1, 2, 4, 8, 16, 32, ..., 1024.
5. Generar los primeros n números de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Por ejemplo si se indica n=5, se debe mostrar: 0, 1, 1, 2, 3.
6. Imprimir los números múltiplos de 7 entre 2 números que se indicarán.

2.5 Concepto de "loop"

Reconsideremos el ejercicio de leer 20 datos y mostrar el producto:

```
prod = 1
van = 0
mientras (van < 20)
        leo dato
        prod = prod * dato
        van = van + 1
mostrar prod
```

¿Qué pasaría si por error omitimos la sentencia "van= van + 1"? El programa quedaría en "*loop*", en circuito infinito. No terminaría. Es ciertamente recomendable asegurarse que los ciclos iterativos tengan terminación.

2.6 Programa

Es un conjunto de instrucciones en orden lógico que describen un algoritmo determinado.
Las características esperables de un programa son:

- eficaz: funciona bien;
- entendible: el código es claro;

- modificable: el código puede ser fácilmente modificado, para posibles ajustes o ampliaciones; y
- eficiente: ejecuta en el menor tiempo posible. Notar que la eficiencia va después de la eficacia.

2.7 Expresiones aritméticas

Una expresión aritmética es una constante, o variable, o combinación de constantes y variables vinculadas por los operadores de + - * / y otros operadores. Tiene resultado numérico. Puede tener paréntesis.

Ejemplos.

4 + 3
a + 5 * 8 * i
50 % 3 nota: el operador % representa el "módulo", el resto de la división entera. 50%3 es el resto de la división de 50 entre 3. 50 dividido 3 da cociente 16 y resto 2. 50% 3 es 2.
(total * 1.23+5)+ (precio*0.20-0.4)

Ejercicio: ¿Qué errores tienen las siguientes líneas?

a) aux = 4 * - 3
b) m = 3(i+j)
c) total = total45
d) suma = suma + ((3+suma)/2

Solución:

a) falta paréntesis, no se pueden poner 2 operadores juntos
b) no se puede omitir ningún operador, falta el * luego del valor 3
c) estaría correcto si total45 es una variable.
d) falta un paréntesis

2.8 Expresiones lógicas

2.8.1 Definiciones básicas

2.8.1.1 Variable booleana

Una variable booleana (*boolean*) es una variable que solamente vale verdadero (*true*) o falso (*false*).

2.8.1.2 Operadores relacionales

Los operadores relacionales son:

menor: <
mayor: >
mayor o igual: >=
menor o igual: <=
igualdad: ==
diferentes: !=

Notar que el operador para preguntar por igualdad consta de "==", la asignación es "=".

2.8.1.3 Operadores lógicos

Los operadores lógicos son: *and* (y), *or* (o), y *not* (no). Considerando el estilo Java, el *and* se representa por &&, el *or* por || y el *not* por !. En las tablas siguientes se explica cada uno.

El operador *not* invierte el valor verdadero en falso y viceversa:

	not
verdadero	falso
falso	verdadero

El operador *and* (y) tiene resultado cierto cuando las condiciones sobre las que se aplica son ciertas. Por ejemplo, si para ir al cine las condiciones son a) tener tiempo y b) tener dinero, sólo se irá al cine cuando ocurran las 2 condiciones simultáneamente.

ejemplo: tener tiempo	ejemplo: tener dinero	and ejemplo: tener tiempo y tener dinero
falso	falso	falso
falso	verdadero	falso
verdadero	falso	falso
verdadero	verdadero	verdadero

El operador *or* (y) tiene resultado cierto cuando alguna de las condiciones son ciertas. Por ejemplo, si para enterarnos de una noticia se puede hacer a través de a) la radio b) la TV, basta con escuchar la radio, ver la TV o ambas cosas para enterarnos.

ejemplo: escuchar radio	ejemplo: ver TV	or ejemplo: escuchar radio o ver TV
falso	falso	falso
falso	verdadero	verdadero
verdadero	falso	verdadero
verdadero	verdadero	verdadero

2.8.1.4 Constantes lógicas

Las constantes lógicas son: verdadero (*true*) y falso (*false*).

2.8.1.5 Expresiones lógicas

Una expresión lógica es una constante lógica, o variable lógica, o 2 expresiones aritméticas separadas por operadores relacionales, u operadores lógicos actuando sobre constantes lógicas, variables lógicas, expresiones relacionales u otras expresiones lógicas. El resultado es verdadero o falso.

2.8.1.6 Precedencia

Así como en matemáticas se sabe que las operaciones de multiplicar y dividir se ejecutan antes que la suma y la resta, la precedencia de los operadores lógicos es: *not*, *and*, *or*. Primero se evaluará el *not*, luego el *and* y finalmente el *or*. Se recomienda, a los efectos de clarificar, utilizar paréntesis.

Ejemplos de expresiones lógicas:

Sean:
x = 3
y = 4
z = 2
f = false
¿Cuánto vale cada una de las siguientes expresiones?

a) (x>z) && (y>z)
b) (x + y / 2) <= 3.5
c) ! f
d) ! f || ((x<z) && (z >= 3*y))
e) f && (x > z + y)

Respuestas:

a) verdadero
b) falso, 5 no es menor que 3.5
c) verdadero
d) verdadero. Ya al evaluar "not f" se sabe que la expresión entera es verdadera pues es un "or".
e) falso. El primer término es falso, por lo cual, dado que es un "and", la expresión entera es falsa. Se dice que se hace evaluación de "corto circuito" cuando apenas se detecta el resultado de la expresión se deja de hacer la evaluación. Tiene como ventaja el ahorro de tiempo.

2.8.2 Uso de variables booleanas

Se desea un programa que lea datos hasta que se ingrese el valor 99 e indicar *al final* si pasó el valor 37 entre los datos ingresados. Analizar la siguiente solución:

```
paso = false
leo dato
mientras (dato != 99)
        si (dato == 37)
                paso = true
        en otro caso
                paso= false
        leo dato
si (paso)
        mostrar “SI”
en otro caso
        mostrar “NO”
```

Realizar la corrida a mano. ¿Qué sucede? En realidad, el programa mostrará si el último dato antes del 99 era o no el 37.

Una solución correcta es:

```
paso = false
leo dato
mientras (dato != 99)
        si (dato == 37)
                paso = true
        leo dato
si (paso)
        mostrar “SI”
en otro caso
```

```
        mostrar "NO"
```

Observar que no es necesario preguntar: "si (paso == true)", con poner "si (paso)" alcanza, pues dado que es una variable booleana sus únicas opciones son verdadero o falso. Si nos interesara preguntar por el valor falso, en vez de poner si (paso== false) poner si (!paso).

En la línea "mostrar 'SI'", el texto 'SI' es un *String*, una secuencia de caracteres. Por pantalla aparecerá la palabra SI.

2.9 Máximo

Se desea leer números hasta que se ingrese el valor 0. Al final, mostrar el mayor dato ingresado (excluyendo el 0). Analizaremos varias posibles soluciones.

```
max = 0
leo dato
mientras (dato !=0)
        si (dato > max)
                max  = dato
        leo dato
mostrar max
```

¿Qué pasa si como primer dato se ingresa el valor 0? Debería aparecer un texto indicando esta situación. La versión corregida sería:

```
max = 0
huboDatos = false
leo dato
mientras (dato !=0)
        huboDatos = true
        si (dato > max)
                max = dato
        leo dato
si (huboDatos)
        mostrar max
en otro caso
        mostrar "no hubo datos"
```

Consideremos otro caso más. ¿Qué pasa si los datos son todos negativos? El programa indicaría que el máximo es 0, lo cual no es correcto. Una alternativa es inicializar la variable max con el menor valor posible: *LowValue*. Este valor dependerá del tipo de datos que se considere, pero está perfectamente definido para cada uno de ellos.

```
max = LowValue
huboDatos = false
leo dato
mientras (dato !=0)
        huboDatos = true
        si (dato > max)
                max = dato
        leo dato
si (huboDatos)
        mostrar max
```

```
en otro caso
        mostrar “no hubo datos”
```

Otra alternativa sería inicializar *max* con el primer dato válido. Notar que la sentencia “mostrar max” despliega el valor de la variable max, o sea, su contenido.

2.10 Práctico 1

Práctico: 1
Tema: Ejercicios básicos - seudocódigo

1 Ingresar n números (n será ingresado inicialmente), e indicar su promedio, cantidad de números pares y de múltiplos de 3.

2 Dada una serie de números (terminada por el ingreso del número 0) indicar los dos números mayores de la misma.

3 Aceptar un número y mostrar sus dígitos.

4 Procesar 15 números e indicar la cantidad de secuencias estrictamente ascendentes que contenga.

Ej: 2 3 5 7 2 5 3 3 6 8 9 11 2 3 4
Contiene 4 secuencias

5 Pedir un número n, luego una serie de n números más, e imprimir el mayor de la misma y la suma de todos los números desde el principio hasta ese máximo inclusive.

Ej: n = 7 Serie: 22 3 5 4 24 17 19 Mayor 24, Suma 58

6 Ingresar la hora y minutos de entrada de un funcionario a la empresa, también la hora y minutos de salida. Indicar la cantidad de horas y minutos que trabajó.

7 Ingresar un número n y mostrar la suma de todos los impares entre 1 y n.

8 Leer 100 números y mostrar el promedio de aquellos que sean menores de 70 y mayores de 50.

Semana 3

3.1 Autoevaluación de seudocódigo

Se recomienda realizar la siguiente autoevaluación de seudocódigo.

1) Un algoritmo es :
   a) Una estructura de control.
   b) Un tipo de variable
   c) Un conjunto de reglas para resolver un problema
2) Una variable es :
   a) Un nombre que representa un valor
   b) Un tipo de datos
   c) Un valor constante asociado a un nombre
3) Verdadero y Falso (True y False) son valores de:
   a) Tipos de datos numéricos
   b) Tipos de datos Lógicos o booleanos
   c) Tipos de datos alfanuméricos o strings
4) Siendo A = 3 y B = 5, indique el resultado de la siguiente expresión : (A/3) >= (B-4)
   a) 10
   b) Verdadero
   c) Falso
5) Siendo A = 7, B = 3 y C = -1, indique el resultado de la siguiente expresión : A>B && A>C
   a) Verdadero
   b) -1
   c) Falso
6) Siendo A = 3, B = -1 y C = 0, indique el resultado de la siguiente expresión : A<= B || B>C
   a) 0
   b) Falso
   c) Verdadero
7) Señale la expresión que representa la siguiente afirmación : "El valor N está entre los valores A y B", siendo A <= B:
   a) N >= A || N <= B
   b) N <= A && N >= B
   c) N >= A && N <= B
   d) N >= A || N >= B
8) Marque la expresión que representa al siguiente problema : "Si el curso es Java, o VB o C y la nota es mayor o igual a 86 el alumno exonera..."
   a) (Curso="Java" || Curso="VB" || Curso="C") && Nota >= 86
   b) (Curso="Java" && Curso="VB" && Curso="C") || Nota >= 86
   c) (Curso="Java" || Curso="VB" || Curso="C") || Nota >= 86
9) Dado el siguiente algoritmo, si el usuario ingresa como valor el 3, decir que imprimirá :

```
leer a
a = a + 1
b = a + a
imprimir a
imprimir b
```

   a) 3 y 5
   b) 4 y 8

c) 4 y 6

10) Correr a mano es:

a) Escribir un algoritmo
b) Ejecutar un algoritmo como lo haría la máquina
c) Una forma de programar

11) Se desea saber si dado un número es o no es alguno de los predeterminados 3, 5 ó 10; indique el algoritmo que resuelve el problema :

a)
```
Leer A
si A==5 && A==3 && A==10 entonces
        imprimir "Si es"
en otro caso
        imprimir "No es"
```

b)
```
Leer A
si A==5 || A==3 || A==10 entonces
        imprimir "Si es"
en otro caso
        imprimir "No es"
```

c)
```
Leer A
si A==3 || A==5 && A==10 entonces
        imprimir "Si es"
en otro caso
        imprimir "No es"
```

12) Indicar los siguientes algoritmos, cuál muestra como resultado : 5, 10, 15, 20, 25, 30

a)
```
repetir desde x=1 hasta 30 Incremento 5
        imprimir x
```

b)
```
x = 0
mientras x <= 30
        x = x+5
        imprimir x
```

c)
```
x=0
repetir
        x = x + 5
        imprimir x
hasta x == 30
```

13) Dado el siguiente algoritmo

```
x = 1
mientras x <= 10
        si x % 2 != 0 entonces
                imprimir x
```

Imprime:

a) 1,3,5,7,9 b) 2,4,6,8,10 c) 1,1,1,1,1,1.... sin fin d) 1 y se *cuelga* la máquina

14) Dado el siguiente algoritmo, indicar su resultado :

```
 x =1
repetir
        Si x % 2 == 0 entonces
                x = x+1
        en otro caso
                imprimir x

        x = x+1
hasta x == 10
```

a) 1,3,7,9
b) 1
c) 2,4,6,8,10
d) 10
15) Dado el siguiente algoritmo que suma edades hasta que se ingrese 99, indique el error :

```
Suma ← 0
Repetir
        Leer Edad
        Suma ← Suma + Edad
Hasta Edad = 99
```

a) El último valor no se suma
b) El 99 se suma como otra edad
c) No se repite el ingreso

Las respuestas correctas son:
1) c, 2) a; 3) b; 4) b; 5) a; 6) b; 7) c; 8) a; 9) b; 10) b; 11) b; 12) c; 13) c; 14) b; 15) b.

3.2 Introducción a Java

Java es un lenguaje de programación de alto nivel, desarrollado por Sun en 1995. Es un lenguaje simple, robusto, orientado a objetos, seguro. Los programas escritos en Java se compilan y ejecutan.

Una *plataforma* es el ambiente requerido de hardware y, o, software para que ejecute un cierto programa. La plataforma de Java solamente consiste en software: la *JVM* (*Java Virtual Machine*) y *API* (*Applications Programming Interface*). En Java es posible crear aplicaciones y applets. Una aplicación es un programa independiente que corre en la plataforma Java. Un applet es un programa que es corrido en un navegador. Trabajaremos con aplicaciones.

Hay distintas versiones de Java. Utilizaremos Java 6 o Java 5 (también denominado Java 1.5), ambas versiones se adaptan perfectamente a los requerimientos de este curso. Para instalar Java, bajar de la página de Sun (www.java.sun.com) la versión correspondiente al sistema operativo que se tenga y luego ejecutar la instalación.

Se irán presentando los conceptos necesarios en Java para poder codificar y ejecutar los programas. Tener en cuenta que este material está orientado a la enseñanza de la programación, no es un libro sobre Java. Para familiarizarse con todos los detalles de Java es recomendable consultar el libro "*Thinking in Java*" de Bruce Eckel, disponible electrónicamente en www.bruceeckel.com.

La estructura de un programa Java incluye clases, métodos y sentencias. Un programa consiste de una o más clases. No hemos presentado aún el concepto de clase, por lo cual dejaremos esto pendiente. Una clase tiene uno o más métodos. Tampoco se ha presentado el concepto de método, así que también queda pendiente. Todas las aplicaciones Java tienen un método *main* (public static void main(String args[]). ¡Otra cosa más pendiente!. Un método contiene sentencias. Aquí sí ya es familiar: sentencias: ¡líneas del programa!.

3.3 Primer ejemplo de codificación Java

Consideremos el programa que permite mostrar los números del 1 al 10. En seudocódigo sería:

```
repetir desde dato==1 a 10
        mostrar dato
```

- En Java, todas las variables deben ser definidas previamente. El nombre de la variable puede contener letras, números, el signo de $ y el _. Debe comenzar con letra. Hay varios tipos posibles, según se desee almacenar un número entero, un número decimal, un texto, etc. Por ahora, utilizaremos tipo entero (int). Así, para definir una variable entera de nombre dato se escribe:

  int dato;

- Hay palabras reservadas, o sea, nombres que no se pueden utilizar como nombres de variables. Algunas palabras reservadas son "*while*", "*for*", "*do*", "*repeat*".

- La sentencia "*for*" se codifica:

  - for (inicio; condición; incremento) {
    .....
    }

    inicio corresponde al valor inicial para la repetición. Por ejemplo i = 1. La variable debe estar ya definida o definirse ahí mismo (int i).
    condición está vinculado a la terminación, por ejemplo i <= 10
    incremento refiere a la variación que se realizará en cada bucle, por ejemplo i = i + 1

- Para mostrar un dato por pantalla se utiliza *System.out.println(...)*. En el paréntesis, indicar que se desea mostrar. Si es un texto (*String*) indicarlo entre comillas:
  System.out.println("Texto de ejemplo");

Se recomienda incluir comentarios en el código, que simplifican la lectura y comprensión del programa. Se pueden poner, para el caso de una única línea:

```
// este es una línea de comentario
```

y para el caso de múltiples líneas:

```
/* este es un comentario
   largo que incluye
    múltiples líneas */
```

En resumen, este ejemplo presenta: comentarios, variables, for, salida y manejo simple de *String*.

El código completo es:

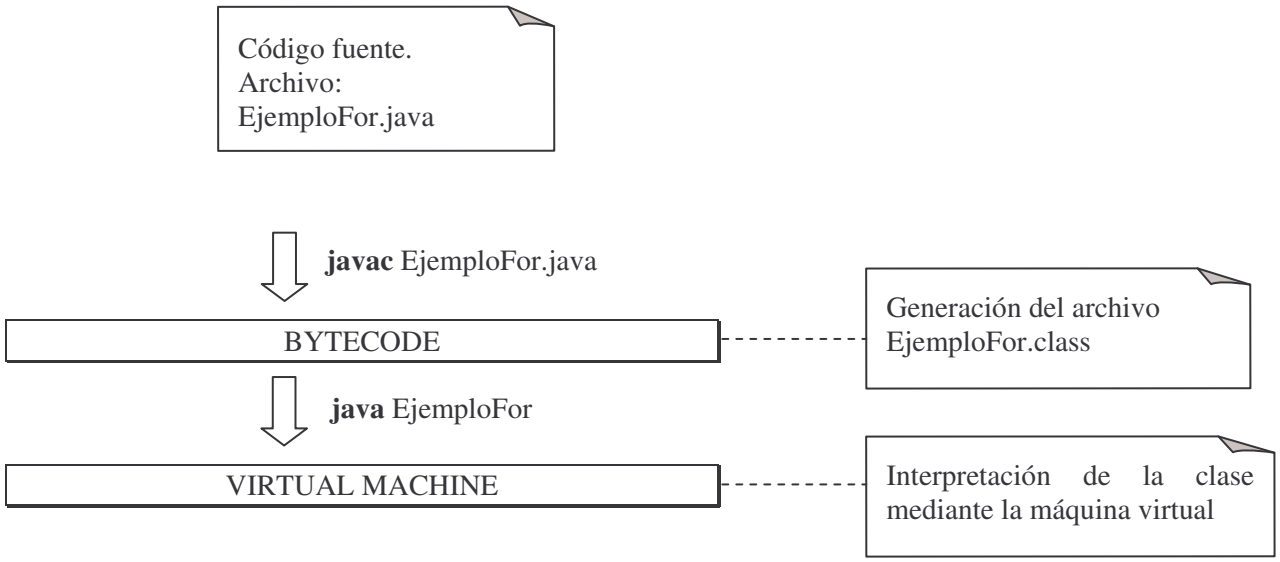
```
// primer ejemplo
public class EjemploFor {
        public static void main(String[] args){
                int i;
                for (i = 1; i <= 10; i= i+1) {
                    System.out.println("El valor es "+ i);
                }
        }
}
```

En términos generales, los pasos para la creación de una aplicación Java son los siguientes:

1. Escribir el código fuente. Puede realizarse en cualquier editor de texto, como por ejemplo el Edit de DOS o el Bloc de Notas. La extensión de los archivos fuentes deberá ser **.java.** Ejemplo: EjemploFor.java. El nombre debe coincidir con el que se puso en la definición "public class".

2. Generación del *byteCode* a través del comando, para este ejemplo, **javac EjemploFor.java**. El *byteCode* es una representación del código fuente en un lenguaje de más bajo nivel. Los errores encontrados en este proceso se muestran por pantalla indicando el archivo que contiene el error, el número de línea y la línea que lo contiene indicando la posición del error en la misma.

3. Ejecución de la aplicación Java a través del comando: **java NombreClase**. El intérprete de Java traduce el ByteCode a código de máquina y lo ejecuta. En particular, para este ejemplo es java EjemploFor.



3.3.1 Representación de datos y Casting

Abrir el Edit o el editor de su preferencia. Ingresar el código fuente siguiente y guardarlo en un archivo de nombre SumaEnteros.java.

```
import java.util.*;
public class SumaEnteros {
  public static void main(String [] args) {
    private int a;
    private int b;
    private int resultado;
    a = 0;
    b = 0;
    resultado = 0;
    Scanner in = new Scanner(System.in);
    System.out.println("Suma de Números");
    System.out.println("Ingrese el primer número: ");
    a = in.nextInt();
    System.out.println("Ingrese el segundo número: ");
    b = in.nextInt();
    resultado = a + b;
    System.out.println("El resultado de la suma es:  "+ resultado);
```

```
  }
}
```

En forma breve, pues se verá en detalle más adelante, se puede destacar:

import java.util.* La sentencia import es utilizada para hacer referencia a clases de un paquete externo. Un paquete es una forma de agrupación de clases.

a = 0; inicializa el valor de la variable entera a.

Scanner in = new Scanner(System.in); se utiliza para la lectura de datos de entrada.

a = in.nextInt(); este método lee una cadena de caracteres (*String*) de la entrada estándar (teclado) y la transforma a un entero.

System.out.println("El resultado de la suma es: "+ resultado); aquí se está utilizando el operador + para concatenar la cadena "El resultado de la suma es:" con el valor de la variable resultado.

Luego de guardar el archivo como SumaEnteros.java, ir a la línea de comandos e ingresar: javac SumaEnteros.java (se genera el *bytecode*). Ingresar el comando: java SumaEnteros para ejecutar la aplicación realizada.

Se desea ahora agregar lo necesario para mostrar el promedio. Así se incluiría que se muestre a/b y observar qué pasa si, por ejemplo, ingreso como datos los valores 3 y 2. El promedio es 2 y no 2.5 como esperaríamos. Esto tiene que ver con los tipos de datos que maneja Java.

Los principales tipos son:

int (para valores enteros)
short (para valores enteros pequeños)
long (para valores enteros más grandes)
float (decimales)
double (decimales, permite mayor tamaño)
String (para textos)
char (para caracteres)
boolean (para valores booleanos)

3.3.2 Conversiones entre tipos de datos

Las conversiones se pueden hacer:
a) por asignación (cuando por ejemplo asigno un dato *int* a una variable *float*),
b) por promoción aritmética, o
c) por "*casting*". Hacer "*cast*" significa que se especifica el tipo de datos que se considerará.

```
Ej:int i = 2;
   int j = 3;
   int k = (i + j ) / 2;
   float p = (i+ j ) / 2;              // ejemplo de asignación
   float q = ((float) i + j )/ 2;      // ejemplo de casting
   System.out.println("Promedio de 2 y 3 guardado en integer "+k);  // Caso 1
   System.out.println("Promedio de 2 y 3 guardado en float   "+p);   // Caso 2
   System.out.println("promedio de 2 y 3 con cast y en float "+q);    // Caso 3
```

¿Qué imprimirá en cada caso?
En el caso 1, saldrá 2, pues hace la cuenta en enteros. En el caso 2, saldrá 2.0, hizo la cuenta en enteros pero el valor se asignó a un float.
En el caso 3, hizo la cuenta en float (porque específicamente se le indicó que considere el valor i como float, se puso un operador de cast). Imprime 2.5.

3.3.3 Errores

Hay 3 tipos posibles de errores al escribir un programa. Ellos son:
a) error de lógica: por ejemplo, se debía solicitar 10 datos y se piden 11
b) error de sintaxis: por ejemplo, se escribió "forr" en vez de "for"
c) error de ejecución: por ejemplo, se hizo una división por 0

Todos los errores de sintaxis deben ser corregidos antes de la ejecución del programa. El compilador informa de los errores de sintaxis.
Para ayudar a prever los posibles errores de lógica y de ejecución se puede realizar la corrida a mano.

3.4 Bibliotecas de Java

Java tiene múltiples bibliotecas. Una biblioteca es un conjunto de utilidades o herramientas. Están agrupadas en "*packages*" o paquetes. Cada paquete tiene un conjunto de clases relacionadas. Por ahora, y dado que aún no se presentó el concepto de clase, simplemente se indica que cada paquete tiene un conjunto de utilidades. Por ejemplo, hay un paquete que tiene elementos relacionados con el manejo de interfaz de usuario (*java.awt*), otro con operaciones de entrada y salida (*java.io*). A medida que se vaya necesitando, se irá presentando el paquete respectivo. Para incluir un paquete en un programa se utiliza la sentencia import.

3.5 Codificación del Ejercicio del Máximo

Modificaremos el ejercicio del máximo, para incorporar que se pueda incluir cualquier valor, inclusive el 0.

```
max = LowValue
huboDatos = false
mostrar "quiere ingresar datos?"
leo resp
mientras (resp == "S")
        mostrar "ingrese dato"
        leo dato
        huboDatos = true
        if (dato > max)
                max = dato
        mostrar "mas datos?"
        leo resp
si (huboDatos)
        mostrar "Máximo vale" + max
en otro caso
        mostrar “no hubo datos”
```

La codificación de este ejercicio presenta varios elementos nuevos:
- definición y uso de variable booleana
- ingreso de datos desde el teclado
- mientras (*while*)
- decisión (*if*)
- *Low Value*
- operadores

Observar que en la condición se utilizó == en vez de =.

3.5.1 Definición de variable booleana

Para definir una variable booleana se indica su tipo y el nombre, por ejemplo: boolean hubo.
Se puede inicializarla también en la misma línea:
boolean hubo = false;

3.5.2 Lectura de datos desde el teclado

Para ingresar datos desde el teclado, desde la versión Java 5 se utiliza la clase Scanner, que está en el paquete java.util.
Se define:

```
Scanner input = new Scanner(System.in);
```

Luego, para leer un dato entero:

```
int valor = input.nextInt();
```

Para leer un dato float:

```
float dato = input.nextFloat();
```

Para leer un texto:

```
String texto = input.nextLine();
```

Para esta altura del curso de Programación I se asumirá que el usuario ingresará los valores correctamente, esto es, cuando debe ingresar un valor entero, digitará un número entero (y no un número con punto decimal o letras). Más adelante se verá el manejo de excepciones, para tratar esos casos erróneos.

Al final del programa, se estila poner input.close();

3.5.3 Codificación de estructuras de control (mientras, repetir)

La estructura mientras se codifica:

```
while (condición) {
        Código a ejecutar
}
```

La estructura repetir mientras se codifica:

```
do {
        Código a ejecutar
} while (condición);
```

En Java no existe la estructura “repetir hasta”, es “repetir mientras”.

3.5.4 Codificación de decisiones

Las decisiones se codifican:

a)
```
if (condición) {
        sentencias
}
else {
        sentencias
}
```

b)
```
if (condición) {
        sentencias
}
```

3.5.5 Operadores aritméticos

En la siguiente tabla se presentan algunos de los operadores disponibles en Java:

Operador	Descripción
++ / --	Auto incremento/decremento en 1
+ / -	Operador unario (mas/menos)
*	Multiplicación
/	División
%	Módulo (resto de división entera)
+ / -	Suma y resta

El operador ++ incrementa la variable en 1. Por ejemplo:

```
int dato= 6;
dato++;
```

La variable dato queda con el valor 7.

Un error frecuente es codificar lo siguiente:

```
dato = dato++;
```

¡La variable dato quedará con el valor original!

3.5.6 Operadores lógicos

Los operadores lógicos son:

Operador	Descripción
!	not
&&	and
||	or

3.5.7 Operadores de comparación

Los operadores de comparación en Java son:

Operador	Descripción
<	Menor que

>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual
!=	Distinto

Como se observa en la tabla anterior, el símbolo == corresponde al operador de comparación por igualdad. Es diferente a =, que representa una asignación.

3.5.8 Codificación del máximo

La codificación de este programa es:

```
import java.util.*;

public class Prueba{

public static void main (String args[]){

        int max = Integer.MIN_VALUE;  // es el Low Value
        int dato;
        boolean hubo = false;
        String resp;
        Scanner input = new Scanner(System.in);

        System.out.println("Quiere ingresar algo?");
        resp = input.nextLine();
        while (resp.equals("S")){
                hubo = true;
                System.out.println("Ingrese dato");
                dato = input.nextInt();
                if (dato > max){
                        max = dato;
                }
                System.out.println("Mas datos?");
                resp = input.nextLine();
        }
        if (hubo){
                System.out.println("Máximo vale "+max);
        }
        else {
                System.out.println("No hubo datos");
        }
        input.close();
   }
}
```

Notar la correspondencia prácticamente línea a línea del seudocódigo al código. Para preguntar si un *String* es igual a otro se utiliza equals, se verá detalladamente más adelante.

Semana 4

4.1 Autoevaluación de codificación Java

Realizar el siguiente ejercicio de autoevaluación:

1) Se quiere definir una variable de nombre **contador**, de tipo entera e inicializarla en 1. La forma correcta de hacerlo en Java es:

a) Int contador = 1;
b) int contador = = 1;
c) int contador = 1;
d) integer contador = 1;

2) La línea: **string nombre = "Juan",** no compila. ¿Cuál o cuáles son las causas?
causa 1) String está con minúscula.
causa 2) La coma al final es incorrecta, debe ser punto y coma

a) las 2 causas
b) sólo la *causa 1*
c) sólo la *causa 2*
d) otro motivo

3) Dado el siguiente código:

```
int i = 5;
if (i = 6){
        System.out.println("i es 6");
}
else {
        System.out.println("i no es 6");
}
```

a) No compilaría
b) Si se ejecuta se obtiene: i no es 6
c) Si se ejecuta se obtiene: i es 6
d) Si se ejecuta se obtiene: 5 es 6

4) ¿Cuál será la salida del siguiente programa?

```
int do = 8;
while (do <= 10) {
        do = do + 1;
        System.out.println(do);
}
```

a) Loop infinito
b) 9 10 11
c) Error de sintaxis

5) ¿Cuál de los siguientes NO es un nombre válido de variable?

a) contadorDeNombreMuyLargo
b) numero_1
c) –numero_1
d) numero_a

6) En una variable definida de tipo booleano se puede asignar:

a) true o false
b) cualquier número
c) cualquier texto

7) ¿Qué se mostrará con el siguiente código?

```
int val = 11;
System.out.println(“Resultado: ” + val +1 );
```

a) Resultado: 12
b) Resultado: 11 12
c) Resultado: 111
d) Resultado: val 1

8) ¿Qué hace el siguiente código?

```
int i =0;
while (i<= 10){
        i= i+1;
        System.out.println(i);
}
```

a) Generaría loop infinito
b) No compila
c) Imprime los números del 1 al 10
d) Imprime los números del 1 al 11

9) ¿Qué contiene un archivo .java?

a) Código de máquina pronto para ser ejecutado
b) Código solamente ejecutable por la Java Virtual Machine
c) Código fuente escrito en Java

10) ¿Cuál será la salida del siguiente programa?

```
int x = 5;
while (x != 0) {
        x= x-1;
        System.out.print("x="+x);
}
```

a) No compila
b) Loop infinito
c) x=4 x=3 x=2 x=1 x=0
d) x=4 x=3 x=2 x=1

Las respuestas son: 1) c; 2) a; 3) a; 4) c; 5) c; 6) a; 7) c; 8) d; 9) c; 10) c.
La pregunta 3 no compilaría pues en el *if* se utilizó el operador de asignación (=) y no el de comparación (==).
En particular, la pregunta 4) da error de sintaxis porque no es posible utilizar como nombre de variable la palabra "do", pues es una palabra reservada del lenguaje.

4.2 Ejercicio de detección de errores

A los efectos de repasar codificación y corrida de programas, realizar el siguiente ejercicio.

Ejercicio: Indicar los errores del siguiente código

```
package prueba;
public class MiClase {
public static void main(String args[]) {
int x,y;
x = 1;
while (x <= 10) ;{
y = 1;
while (y <= 10) {
System.out.print( x + " * " + y +" = " + x*y);
System.out.println();
}
x = x+ 1;
}
```

```
}
}
```

Algunos errores son:
1) Faltan comentarios que indiquen el objetivo y características del programa. Analizando el código se interpreta que probablemente se desea imprimir las tablas de multiplicar.
2) La indentación es inadecuada.
3) El ";" luego del primer while genera ¡loop!.
4) Falta incrementar *y* en el segundo while, lo que genera ¡loop!
5) No es necesario usar *print* y *println* en este caso. El método *print* imprime lo indicado entre paréntesis y deja el cursor luego de esa impresión, el método *println* imprime y luego avanza el cursor a la siguiente línea.
6) Las variables no son nemotécnicas.

Una posible versión corregida:

```
package ejemplo;
public class EjemploMultiplicacion {
   public static void main(String args[]) {
        // Muestra las tablas de multiplicar
        int fila, col;
        fila = 1;
        // por cada fila
        while (fila <= 10) {
                col = 1;
                // por cada columna
                while (col <= 10) {
                        System.out.println(fila + " * " + col + " = " + fila * col);
                        col = col + 1;
                }
                fila = fila + 1;
        }
   }
}
```

4.3 Práctico 2

Práctico: 2
Tema: Ejercicios básicos en Java

Implementar en Java todos los ejercicios vistos, en particular los del práctico 1. Además:

1 Ejecutar el siguiente programa en Java:

```
import java.io.*;
public class PruebaErrores {
        public static void main (String [] args) throws IOException{
                System.out.println("Mensaje de Ejercicio de Programación  " + " I ");
                System.out.println("Para continuar, dar enter);
                System.in.read();
        }
}
```

Incorporar los siguientes cambios de a uno por vez, analizando y anotando el mensaje que produce, corregirlo si es necesario y probar el siguiente:
a) sacar throws IOException
b) sacar la primera “ del literal
c) sacar el primer ;
d) sacar System.in.read();
e) cambiar "de Programación" por "del curso de Semestre 1"
f) sacar la última " del literal
g) sacar la última }
h) sacar import java.io.*;

2 Ingresar 10 datos y mostrar la suma

3 Ingresar datos hasta el número 0 y mostrar promedio de esos datos.

4 Ingresar 3 datos, 1 de ellos estará repetido. Indicar cuál es el repetido.

5 Imprimir 10, 20, ..., 1000.

6 ¿Qué tipos de comentarios existen en Java?

7 Anotar 4 enunciados diferentes en Java que sumen 1 a la variable entera x.

8 Inicialmente x vale 5, producto 10 y cociente 20. ¿Qué valor tienen finalmente x, producto y cociente?

```
producto *= x++;                    cociente / = ++x;
```

9 ¿Qué error tiene la siguiente estructura?

```
while (z >0)
sum += z;
```

4.4 Introducción a clases y objetos: Florista

En este punto se presentarán los conceptos de clase, objeto, método, instancia, jerarquía, herencia y muchos términos más. Supongamos que queremos enviarle flores a mi abuela que vive en Colonia y no puedo llevarlas personalmente pues estoy en Montevideo (a 177 km). [1]¿Qué opciones hay entonces? Algunas son:

- ir a una florería y encargarlas
- pedirle a mi esposo que las mande
- pedirle a un vecino que las mande

El mecanismo en orientación a objetos es encontrar el agente apropiado y darle el *mensaje* con mi requerimiento. Es responsabilidad del que recibe el mensaje, si lo acepta, llevar a cabo el *método* o algoritmo.

La acción en orientación a objetos comienza con el envío de un mensaje a un *objeto*. El mensaje es el pedido de la acción más información adicional. El receptor, si acepta el mensaje, es responsable de la resolución y ejecuta un método. Cómo lo lleva adelante no lo sabemos, está *encapsulado* el mecanismo.

En el caso del ejemplo, voy a la florería, hablo con la florista Florencia y le solicito que envíe determinadas flores a la dirección de mi abuela el día que le indique. O sea, le damos el mensaje

[1] Ejemplo adaptado de Budd[2001].

de enviar las flores con información adicional (qué flores, a qué dirección, qué día). Cómo lo lleva adelante no lo sabemos y, en realidad, tampoco es relevante. Por ejemplo, Florencia podría llevarlas ella misma o llamar por teléfono a un conocido de ella en Colonia o ir a la Terminal de Buses y despacharlas. Por eso decimos que el mecanismo está "encapsulado".

Si el mensaje se lo hubiera dado a mi esposo, probablemente el método que ejecutara fuera otro: quizás él mismo fuera a la florería o las llevara personalmente. Es un caso de *polimorfismo*, el mensaje es el mismo ("enviar flores a la abuela"), pero la implementación depende del receptor.

La florista Florencia puede usar cualquiera de las técnicas vistas para resolver el envío, pero en realidad, dado que uno ha tratado en alguna otra ocasión con floristas, se tiene una idea de cuál es el comportamiento esperado. En términos de programación orientada a objetos, se dice que Florencia es una *instancia* de la clase *Florista*. Todo objeto es instancia de una clase.

Sabemos también que Florencia nos pedirá dinero por la transacción y nos emitirá un recibo. Esta información la sabemos porque además de ser florista, en forma general es comerciante. Los comerciantes piden dinero por las transacciones y emiten recibos. Los floristas son una especialización de los comerciantes. Mi conocimiento de los floristas lo puedo expresar en términos de *jerarquía de clases*. Realizando una abstracción, o sea, tomando los elementos más relevantes, una posible jerarquía (Ilustración 4 Jerarquía) es:

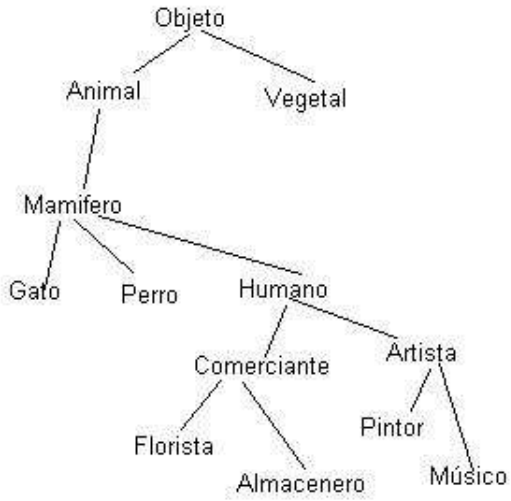


Ilustración 4 Jerarquía

Florencia, instancia de la clase Florista, *heredará* todos los comportamientos de los comerciantes. Florista es una *subclase* de la *clase* Comerciante. Comerciante es *superclase* de Florista y de Almacenero. Las subclases heredan de las superclases.

Florista es una clase concreta, pues se espera que haya instancias de ella (por ejemplo Florencia). Ejemplos de instancias de la clase Pintor podría ser Dalí y Picasso; ejemplos de instancias de la clase Perro podrían ser Pluto, Firulais y Lassie; de la clase Gato sería Garfield. No se espera que haya instancias de la clase Animal, se espera que haya de Gato o de Perro. Así, Animal es una clase *abstracta* y Gato es una clase *concreta*.

4.5 Introducción a clases y objetos: Biblioteca

Supongamos que se desea modelar la Biblioteca de la Universidad. En primera instancia, se trata de identificar los principales conceptos y cómo se relacionan. Probablemente aparezcan muchos términos: libro, revista, diario, mesa, silla, computadora, estantería, funcionario, etc.

Dado que la letra del ejercicio es tan amplia y con poco detalle, podría interpretarse de muchas formas. En particular nos referiremos al sistema para retirar libros. Ahí seguramente las estanterías y mesas no sean relevantes.

Se identifican así los siguientes conceptos:
- Funcionario (quien hace los préstamos);
- Socio (quien retira un material). Hay socios estudiantes y socios profesores, que tienen distintos derechos;
- Libro;
- Revista; y
- Video.

Se trata de intentar definir la relación o la vinculación entre esos conceptos y también las propiedades de cada uno de los conceptos. Así, identificamos que el libro tiene ISBN (número único de identificación), título, cantidad de páginas y un número de inventario, entre otras propiedades. Cualquier revista tiene también título, número, ISSN (número único de identificación de revistas) y cantidad de páginas. Es posible identificar que hay en general materiales, que son publicaciones impresas (como libro y revista) o video. Otros elementos como diarios, CD o cassettes no se consideran en esta solución. Al modelar un problema, se realiza una abstracción, se toman los elementos que se consideran más relevantes para el caso.

También hay distintos tipos de persona en el problema: funcionarios (que son quienes realizan los préstamos), socio Profesor y socio Estudiante. Toda persona tiene nombre y dirección. Todo socio tiene un número, además los profesores tienen número de profesor y los estudiantes número de estudiante. Otros datos, como por ejemplo el tipo de sangre de la persona, no son relevantes para este problema en particular, por eso no se modelan.

La biblioteca tiene un conjunto de socios, materiales y funcionarios. Al analizar más en detalle, aparece el concepto de Préstamo, que vincula a una persona con un material. El préstamo es realizado por un funcionario en una cierta fecha. La biblioteca tiene la lista de préstamos.

Una forma de representar todos estos elementos es con el *diagrama de clases*. Biblioteca, Socio, Funcionario, Material y Libro son ejemplos de clases. Nombre, dirección, número de inventario y título son ejemplos de atributos. Un *atributo* es una característica o propiedad de la clase. Una clase puede tener múltiples atributos.

Una primera aproximación al diagrama de clases del problema se presenta en la siguiente imagen (Ilustración 5 Biblioteca). Las líneas entre clases muestran la relación entre ellas. Cada préstamo está asociado a Material, Funcionario y Socio (se marca con una línea). Publicación es un tipo de Material (se marca con un triángulo). La Biblioteca tiene un conjunto de socios (se marca con un rombo). Estudiante y Profesor son un tipo de socio, "*heredan*" de socio todas las características, son subclases de Socio. (Notar que el nombre de la clase se pone con su inicial en mayúscula y va en singular). El signo de "menos" puesto antes de cada atributo tiene que ver con la "*visibilidad*", concepto que se presentará más adelante.

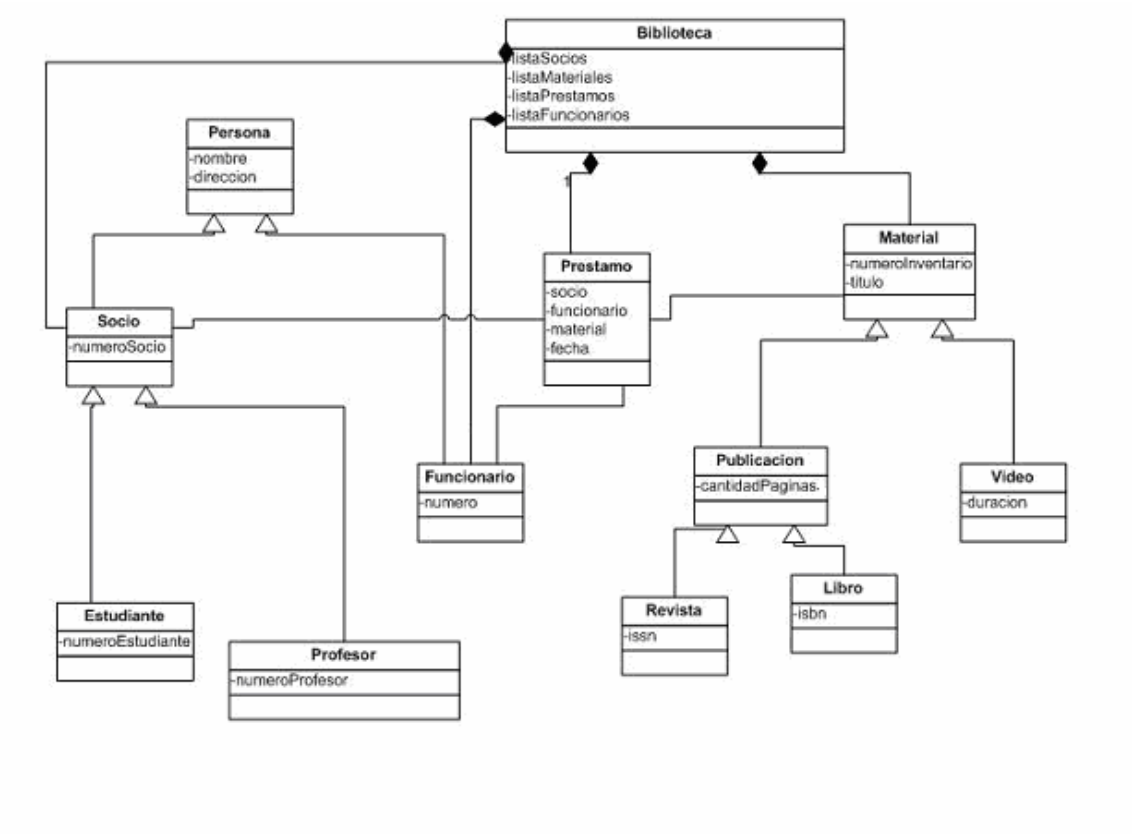


Ilustración 5 Biblioteca

Semana 5

5.1 Programación Orientada a Objetos

La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno instancia de algunas clases y cuyas clases son miembros de una jerarquía de clases unidas mediante relación de herencia.

El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del dominio del problema.

5.2 UML: notación básica

UML (*unified modelling language)* es el lenguaje unificado de modelado. Es un lenguaje que permite visualizar, especificar, documentar y construir los distintos elementos del software. Se utiliza en especial en este curso para representar las clases y sus relaciones. A esta altura del curso de Programación I, solamente se verán los elementos fundamentales de la notación, con simplificaciones.

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema. Veremos como representar una clase.

5.2.1 Notación de Clase, Atributos y Métodos

La clase se representa con una cajita con 3 partes:

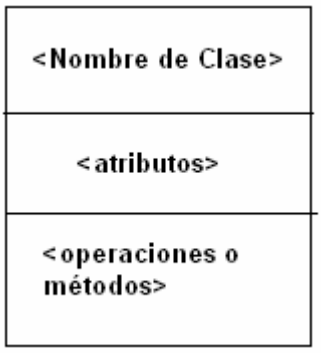


Ilustración 6 Clase

El área superior contiene el nombre de la Clase.
El área intermedia contiene los atributos (o variables) que caracterizan a la clase (pueden ser *private*, *protected* o *public*).
El área inferior contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: privada, protegida o pública).

Los atributos se indican con su visibilidad, nombre y tipo. La visibilidad es:

privada (private):	Se antecede el signo de menos (-) al atributo.
pública (public):	Se antecede el signo de más (+) al atributo.
protegida (protected):	Se antecede el signo numeral (#) al atributo.

Los métodos (“cosas que sabe hacer la clase”) se escriben indicando su visibilidad, el nombre, la lista de parámetros que reciben y la indicación del retorno, *void* si no retornan nada. Se verá con más detalle más adelante.

Se presenta un ejemplo en la Ilustración 7 Clase Persona:

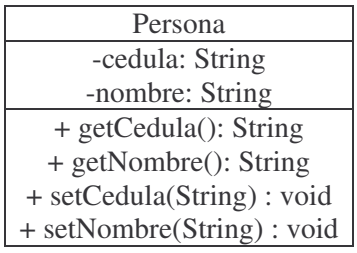


Ilustración 7 Clase Persona

Para el caso de métodos y, o, atributos de clase (concepto que se verá luego), se utiliza la convención de subrayarlos y de utilizar nombres que comiencen en mayúscula.

5.2.2 Relación de Asociación

Entre Préstamo y Socio existe una relación denominada asociación. El préstamo corresponde a un socio, el socio participa del préstamo. También existe asociación entre Préstamo y Funcionario, y entre Préstamo y Material. En particular, el vínculo entre Préstamo y Material se presenta en la figura Ilustración 8 Asociación:

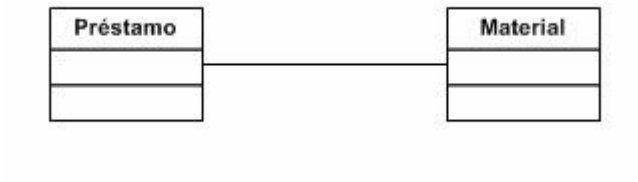


Ilustración 8 Asociación

La forma de implementarla efectivamente y usarla en los programas se verá más adelante.

5.2.3 Herencia

Como se vio en el ejemplo de la Biblioteca, una Publicación es un tipo de Material. Esa relación se denomina generalización-especialización: la publicación es un caso específico de material y el material generaliza el concepto de publicación. Publicación es subclase de Material, tiene todas las características del Material y puede incluir características propias. Para indicar la relación se utiliza un triángulo (Ilustración 9 Herencia). A través del mecanismo de herencia se comparten elementos de la relación de generalización-especialización.

Todos estos conceptos serán presentados en forma detallada más adelante.

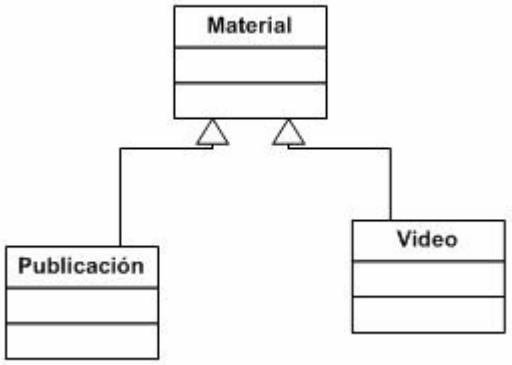


Ilustración 9 Herencia

5.2.4 Agregación y composición

En el ejemplo de la Biblioteca se vio que ella contenía la información de un conjunto de socios. Para los fines del curso de Programación I, no se hará distinción entre agregación y composición, esta diferenciación es apropiada para cursos posteriores de diseño orientado a objetos.

La Biblioteca contiene una lista de socios. Se indica con un rombo (Ilustración 10 Agregación y composición). En Programación 1 no haremos diferencia si el rombo es blanco o negro (casos específicos que incluye UML).

Ejemplo:

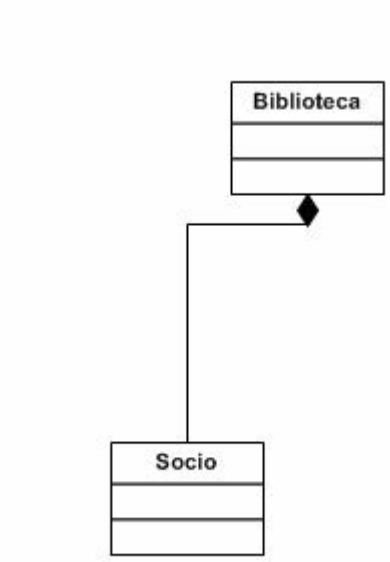


Ilustración 10 Agregación y composición

5.3 Práctico 3

Práctico No. 3 Tema: Objetos, clases, relaciones

Los siguientes ejercicios tienen como finalidad poder identificar objetos, clases, relaciones, atributos y operaciones.

1) Hay dos listas. La primera es una lista de clases y la segunda una lista de operaciones. A cada clase, asociar las operaciones que tendrían sentido:
Lista de clases:
- tabla de símbolos (una tabla que asocia palabras claves con sus descripciones)
- "array" de largo variable (una colección ordenada de objetos, indexada por un número, cuyo tamaño puede cambiar)
- set (conjunto desordenado sin duplicados)
Lista de operaciones:
- append (agregar un objeto al final de la colección)
- copy (hacer una copia de una colección)
- count (retorna la cantidad de elementos de una colección)
- delete (borra un miembro de la colección)
- index (devuelve el elemento de una colección en esa posición)
- intersect (determina los miembros comunes de dos colecciones)

2) ¿Qué tienen en común? (indicar distintas agrupaciones posibles):
a) bicicleta, auto, camión, aeroplano, moto, caballo
b) raíz cuadrada, exponencial, seno, coseno
c) microscopio electrónico, telescopio, binoculares, lentes

3) El alambre se usa para varias aplicaciones. Anotar qué características son importantes cuando se desarrollan aplicaciones para:
a) elegir alambre para un cable transatlántico
b) diseñar el sistema eléctrico de un aeroplano
c) sostener una jaula de un árbol
d) diseñar un filamento para una bombita eléctrica

4) Indicar si las siguientes relaciones son de generalización o de asociación.
a) un país tiene varias ciudades
b) hay archivos de texto, de directorios, de objetos gráficos
c) una carretera une 2 ciudades
d) un estudiante cursa una materia

5) Un almacén de venta por mayor vende productos alimenticios perecederos y no perecederos y artículos de limpieza. Tiene dos vendedores. Desarrollar un diagrama de clases que modele esta situación. Incluir las clases que se consideren necesarias con sus atributos y métodos.

6) Una empresa de courier tiene varios camiones, camionetas y motos. Los envíos pueden ser cartas, paquetes o cajas grandes, hasta 70 kgs. Cada envío tiene: remitente, destino y peso. Desarrollar un diagrama de clases que modele esta situación. Incluir las clases que se consideren necesarias con sus atributos y métodos.

5.4 Uso de clases standard

Hasta ahora, vimos cómo probar pequeños programas en Java que permitan pedir datos y procesarlos, por ejemplo calcular el promedio o el máximo, o solamente mostrar datos, por ejemplo, mostrar los números 10,20, …, 1000.

Se trataron ejemplos de clases e instancias (con el caso de la florista) y se hizo el ejemplo de la biblioteca. También se planteó el práctico 3. Se presentaron conceptos como clase, instancia,

método, mensaje, herencia, polimorfismo, clase abstracta. Son casi todos los conceptos básicos de la orientación a objetos.

Ahora, para empezar a profundizar, veremos el uso de clases *standard* y luego crearemos clases e instancias capaces de reconocer mensajes, y, en respuesta a esos mensajes, ejecutar métodos.

5.4.1 Clase String

En Java existen tipos de datos primitivos (como *int* o *boolean*) y tipo Clase. Un *String* en Java es una secuencia de caracteres. No es un tipo primitivo, es tipo Clase.

Son especiales porque se pueden crear de dos formas:

```
String s1 = "Ana";
String s2 = new String ("Ana");
```

Para referirse a los caracteres se utiliza la posición. La primera posición es 0. Las operaciones comunes en un *string*, por ejemplo *s* son:

s.length() (da el largo)
s.toUpperCase() (lo pasa a mayúsculas)
s.indexOf(“A”) (retorna la posición de la letra "A" en *s*)

Un tema importante tiene que ver con la comparación de *strings*. Supongamos que se tienen las siguientes definiciones:

```
String s1, s2, s3;
s1 = "Ana";
s2 = "Ana";
s3 = new String ("Ana");

String s4;
s4 = s3;
```

Si se hacen las comparaciones:

s1 == s2 es TRUE
s1 == s3 es FALSE
s1.equals(s2) es TRUE
s1.equals(s3) es TRUE
s1.compareTo(s2) es 0
s1.compareTo(s3) es 0
s4 == s3 es TRUE

s3 y s4 son alias: refieren o "miran" al mismo objeto.

Si se pone: s1 == s2, se compara referencias, no el valor. Pregunta si refieren al mismo lugar en memoria. La equivalencia se puede preguntar:

s1.equals (s2)
s1.equalsIgnoreCase(s2) (devuelve *true* si son iguales ignorando mayúsculas y minúsculas)
s1.compareTo(s2)

Tomar nota que compara lexicográficamente. El resultado es:

<0 : s1 va en forma lexicográfica antes que s2
0: s1 y s2 son iguales
>0: s1 va después s2

Se sugiere probar en Java los ejemplos para analizar cuidadosamente todos los casos presentados.

5.4.2 Clase Math

Muchas veces es necesario realizar operaciones matemáticas como por ejemplo calcular la raíz cuadrada o el máximo de 2 números. Se dispone de la clase *Math* (en *java.lang*). Esta clase *Math* es una agrupación de métodos útiles. Aunque podría considerarse en un sentido estricto que no es un "buen ejemplo" de definición de una clase pues no representa dicho concepto de clase como molde de creación de objetos, igualmente es de utilidad conocerla.

Veamos el siguiente ejemplo. Dentro de un main se tienen las siguientes definiciones:

```
int i, j;
float p;
double q;
i= 10;
j = 20;
p = 14.5f;  // la "f" indica considerar el número anterior como float
q = -45.67;
```

Probar que imprimen:

```
System.out.println("Máximo de "+i+ " y "+j + " vale "+ Math.max(i,j));
System.out.println("Máximo de "+i+ " y "+p + " vale "+ Math.max(i,p));
System.out.println("Raíz de "+p+ " vale "+ Math.sqrt(p));
System.out.println("Elevo al cubo a "+i+ " y  vale "+ Math.pow(i,3));
System.out.println("El valor absoluto de  "+q+ " es "+ Math.abs(q));
```

Se sugiere investigar la clase Math: una buena forma de conocer una clase en profundidad es hacer un pequeño programa y realizar pruebas de los diferentes métodos que dispone la clase en cuestión.

5.5 Creación de una clase y su prueba: Camión

El objetivo de este tema es presentar la construcción en forma simplificada de una clase y su prueba.

Veremos cómo:
1) Definir una clase;
2) Ponerle atributos ("variables de instancia");
3) Responder mensajes ("métodos", "parámetros", "*return*").

Además:
4) ¿Cómo se prueba? ¿Cómo se crea un objeto? (*new*)
5) ¿Cómo se muestra? ("*to String*")
6) ¿Cómo se inicializa por defecto? (Constructores)

En el ejercicio 6 del práctico 3 aparece la clase Camión. Se decidió guardar la chapa y el color del camión como atributos relevantes al problema. Como primera aproximación a la definición y prueba de clases, se implementará esa clase Camión y una de prueba.

La lista de pasos que se seguirá es:

1. Definir la clase en Java;
2. Poner los atributos (variables de instancia);
3. Agregarle que pueda responder mensajes;
4. Ver cómo probarla, cómo se crean objetos;
5. Ver cómo se muestran los objetos;
6. Analizar cómo se puede comparar;
7. Estudiar cómo se inicializa por defecto; y
8. Presentar variables de clase

5.5.1 Definición de la clase Camión

Se verán los elementos de la clase Camión incluidos en la Ilustración 11 Clase Camión.

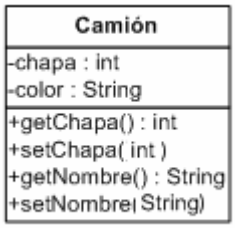


Ilustración 11 Clase Camión

Para definir la clase, crear una clase de nombre Camión dentro de un *package.*

```
package prueba;
public class Camion {

}
```

5.5.2 Atributos o variables de instancia

Los atributos o variables de instancia se definen:

```
public class Camion {
        // variables de instancia
        private int chapa;
        private String color;
}
```

De cada uno se indica la visibilidad, tipo y nombre. La visibilidad, como se comentó, puede ser *public* o *private*, en general se ponen *private*. Se amplía en el siguiente punto. El tipo puede ser, por ejemplo, *int*, *String*, *boolean*, etc.

5.5.3 Métodos de acceso y modificación

Por cada variable de instancia se definen métodos de acceso y modificación. En forma simplificada, un **método** es un conjunto de sentencias con un nombre. Para definirlo se debe incluir la visibilidad, retorno, nombre y opcionalmente la lista de parámetros. El método de acceso permite conocer el valor de la variable y el método de modificación permite cambiarlo.

La visibilidad, al igual que en los atributos según se vio recién, puede ser, entre otras opciones, *public* o *private*. La palabra *public* (que se puede poner delante de variables, métodos, clases)

representa la visibilidad de la variable, métodos, clase, respectivamente. Con ello, se puede controlar el alcance. Si algo es *public*, puede ser referido desde dentro o fuera del objeto. Si algo es *private*, sólo se puede acceder desde dentro. Las variables en general son *private*. Los métodos en general son *public*.

El retorno ("*return*") puede ser por ejemplo *int*, *String* o *void* (nulo). Refiere a lo que será devuelto por el método.

Los nombres de los métodos de acceso y de modificación son comúnmente *get*... y *set*.... Por ejemplo, el método para obtener la chapa es *getChapa* y el método para modificarla es *setChapa*. El parámetro de este método habitualmente es "unXX", así, para el caso de la chapa se pone *int unaChapa.* Se recomienda consultar los estándares de Java en cuanto a codificación.

```
public class Camion {
        // variables de instancia
        private int chapa;
        private String color;
        // métodos de acceso y modificación
        public int getChapa() {
                return chapa;
        }
        public void setChapa(int unaChapa) {
                chapa = unaChapa;
        }
        public String getColor() {
                return color;
        }
        public void setColor(String unColor) {
                color = unColor;
        }
}
```

Para definir un parámetro se indica el tipo y el nombre. *String unColor* refiere a un parámetro, un parámetro es información que le paso al método. Cuando se define el método, se pone el tipo del parámetro y un nombre genérico, cuando se usa se le pasa el objeto o valor real. Internamente al método, usa el nombre definido en el cabezal. Uno se llama parámetro formal, el de la definición; el otro es el parámetro actual, el de la llamada.

5.5.4 Prueba y creación de objetos

Se hace otra clase aparte para probar. Se incluye el método *main*. Para crear un objeto se utiliza el operador **new**.

```
package prueba;

public class ClasePruebaCamion {
  public static void main (String args[]) {
    Camion c1, c2;
    c1 = new Camion();
    c1.setColor("Rojo");
    c1.setChapa (1234);
    c2 = new Camion();
    c2.setColor("Rojo");
```

```
    c2.setChapa (5555);
    System.out.println(c1.getColor());
    System.out.println("Chapa del camion 2 es "+c2.getChapa());
  }
}
```

Ejecutarlo.
Se crearon 2 instancias de Camión, le hemos enviado mensajes y han respondido.

5.5.5 Impresión de los objetos (toString)

Probar qué pasa si se sustituye: System.out.println(c1.getColor()) por: System.out.println(c1);
Se obtiene una descripción del camión no demasiado útil, pues sale el nombre de la clase, el símbolo @ y probablemente números y letras. Para poder definir un formato más cómodo es necesario reescribir el método *toString* en la clase Camión. Dicho método *toString* devuelve una representación en *String* que representa al objeto. Está definido en *Object*, y como todas las clases derivan de ella, lo pueden usar.

Habitualmente se redefine este método, para que muestre lo que necesito. Cuando se redefine un método se dice que se hace **override,** se sobreescribe. El *toString* es **polimórfico**, tiene muchas formas. Depende de la clase, cuál forma tiene.

Así:

```
public class Camion {
        // variables de instancia
        private int chapa;
        private String color;
        // métodos de acceso y modificación
        public int getChapa() {
                return chapa;
        }
        public void setChapa(int unaChapa) {
                chapa = unaChapa;
        }
        public String getColor() {
                return color;
        }
        public void setColor(String unColor) {
                color = unColor;
        }
        // Para imprimir
        @ Override
        public String toString() {
                return “Camion de color “ + this.getColor() +  “ con chapa “+ this.getChapa();
        }
}
```

El método *toString* es invocado automáticamente al imprimir. La línea *@Override* se utiliza para indicarle explícitamente al compilador que se está sobreescribiendo un método, en este caso el método toString.

Para poder referirse en la clase Camión al propio objeto, es decir, al que recibió el mensaje se utiliza la palabra *this*.

5.5.6 Comparación de objetos

Se desea comparar camiones por su color. En el programa de prueba se querría poder poner:

```
if (c1.tieneMismoColor(c2)) {
        System.out.println("Son del mismo color ");
}
else {
        System.out.println("Son de diferente color ");
}
```

El método tieneMismoColor(Camion) debe agregarse en la clase Camión. Recibe como parámetro el objeto camión con el cual quiero comparar.

En la clase Camión:

```
public class Camion {
        // variables de instancia
        private int chapa;
        private String color;
        // métodos de acceso y modificación
        public int getChapa() {
                return chapa;
        }
        public void setChapa(int unaChapa) {
                chapa = unaChapa;
        }
        public String getColor() {
                return color;
        }
        public void setColor(String unColor) {
                color = unColor;
        }
        // Para imprimir
        @ Override
        public String toString() {
                return "Camion de color " + this.getColor() +  " con chapa "+ this.getChapa();
        }
        // Para comparar
        public boolean tieneMismoColor(Camion unCamion) {
                return this.getColor().equals(unCamion.getColor());
        }
}
```

5.5.7 Inicialización: constructores

Se desea que al crear el camión ya venga con valores iniciales: color blanco y chapa 100. Para ello es necesario definir los **constructores**. Dichos métodos son invocados automáticamente al crear objetos de esa clase.

Es posible definir varios constructores, con diferentes parámetros si se desea inicializar con otros valores. La *firma* de un método incluye el nombre y los respectivos tipos de parámetros. Cuando tengo un mismo método pero tiene diferentes firmas se habla de **overload** (sobrecarga).

```
public class Camion {
	// variables de instancia
	private int chapa;
	private String color;

	// Constructores
	public Camion() {
		this.setColor("blanco");
		this.setChapa(100);
	}

	public Camion(String unColor, int unaChapa){
		this.setColor(unColor);
		this.setChapa(unaChapa);
	}

	// métodos de acceso y modificación
	public int getChapa() {
		return chapa;
	}
	public void setChapa(int unaChapa) {
		chapa = unaChapa;
	}
	public String getColor() {
		return color;
	}
	public void setColor(String unColor) {
		color = unColor;
	}
	// Para imprimir
	@ Override
	public String toString() {
		return "Camion de color " + this.getColor() +  " con chapa "+ this.getChapa();
	}
	// Para comparar
	public boolean tieneMismoColor(Camion unCamion) {
		return this.getColor().equals(unCamion.getColor());
	}
}
```

Para probarlos, en la clase Prueba:

```
package prueba;

public class ClasePruebaCamion {
  public static void main (String args[]) {
    Camion c1, c2;
    c1 = new Camion();
    c2 = new Camion("rojo",1990);
    System.out.println(c1.getColor());
```

```
    System.out.println("Chapa del camion 2 es "+c2.getChapa());
}
}
```

5.5.8 Variables de clase

Supongamos que se desea almacenar el año a partir del cuál se debe realizar el chequeo obligatorio en la Intendencia. Este valor es único para todos los camiones, pero también cada camión debe conocer su año o modelo. Así el año general es una variable de clase y el modelo es una variable de instancia. Para definir una variable de clase o un método de clase, se agrega la palabra *static*. En general, las variables de clase se escriben con la inicial en mayúscula. Para indicar que un método es de clase también se utiliza la palabra *static*. Cuando se desea utilizar un método *static*, el mensaje se le envía a la clase.

Se agregará un método de instancia para determinar si a un camión le toca o no revisión (leTocaRevision()).

En la clase Camión:

```
public class Camion {
        // variables de clase
        private static int Año;

        // variables de instancia
        private int chapa;
        private String color;
        private int modelo;

        // Constructores
        public Camion() {
                this.setColor("blanco");
                this.setChapa(100);
                this.setModelo(2004);
        }

        public Camion(String unColor, int unaChapa){
                this.setColor(unColor);
                this.setChapa(unaChapa);
                this.setModelo(2004);
        }

        // métodos de acceso y modificación de variables de clase
        public static int getAño() {
                return Año;
        }
        public static void setAño(int unAño) {
                Año = unAño;
        }
        // métodos de acceso y modificación de variables de instancia
        public int getChapa() {
                return chapa;
        }
        public void setChapa(int unaChapa) {
```

```
                chapa = unaChapa;
        }
        public String getColor() {
                return color;
        }
        public void setColor(String unColor) {
                color = unColor;
        }
        public int getModelo() {
                return modelo;
        }
        public void setModelo(int unModelo) {
                modelo = unModelo;
        }

        // Para imprimir
        @ Override
        public String toString() {
                return "Camion de color " + this.getColor() +  " con chapa "+ this.getChapa();
        }
        // Para comparar
        public boolean tieneMismoColor(Camion unCamion) {
                return this.getColor().equals(unCamion.getColor());
        }
        // Para saber si le toca revisión
        public boolean leTocaRevision() {
                return this.getModelo() <= Camion.getAño();
        }
}
```

Para probar:

```
public class ClasePruebaCamion {
  public static void main (String args[])  {
    Camion c1;
    c1= new Camion("rojo",1234);
    c1.setModelo(1990);
    Camion.setAño(1995);
    if (c1.leTocaRevision()) {
          System.out.println("Le toca revisión");
     }
    else{
          System.out.println("No le toca revisión");
    }
}
```

5.6 Práctico 4

Práctico No. 4
Tema: Clases y objetos
Objetivo: Uso de clases predefinidas y creación de nuevas clases.

1. Escribir el código necesario que ejemplifique el uso de los siguientes métodos de String: equals, compareTo, indexOf, length, toUpperCase, toLowerCase, equalsIgnoreCase, trim.

2. Escribir el código necesario para mostrar el uso de los siguientes métodos de Math: max, min, abs, pow, sqrt. Investigar cuáles otros métodos están disponibles.

3. Sean las variables enteras: largo=18 y ancho=3 y la variable double altura = 2.25.
Indicar el valor de:
a) largo / ancho
b) largo / altura
c) largo % ancho
d) ancho % altura
e) largo * ancho + altura
f) largo + ancho * altura

Verificarlo en Java.

4. ¿Hay errores en el siguiente código? Indicar cuáles son y cómo corregirlos.

```
contador == 50;
wihle (contador <= 60) {
        System.out.println(contador)
        contador = contador ++;
}
```

5. A la clase Camión definida agregarle la marca (variable y métodos correspondientes).
Crear otra clase para probarla. Se deben crear 2 camiones, solicitando los datos al operador y mostrarlos.

6. Una biblioteca maneja libros, algunos de ellos son originales y otros son fotocopias. No todos los libros se prestan.
a) Crear la clase Libro
b) Agregar atributos: título, original y prestable
c) Agregar métodos de instancia: ‘getOriginal’, ‘getTitulo’ y ‘getPrestable’.
d) Agregar métodos de instancia 'esOriginal' y 'sePresta' que retornen el valor booleano correspondiente.
e) Agregar métodos de instancia ‘setTítulo’ , ‘setOriginal’ y ‘setPrestable’
f) Crear un método main en una clase para prueba que permita obtener 2 instancias de Libro, uno de ellos es original y no se presta, el otro es fotocopia y se presta. Utilizar los métodos de instancia para realizar estas operaciones. Mostrar los libros creados.

Semana 6

6.1 Ejercicios de práctica de estructuras de control

Es importante seguir realizando ejercicios que utilicen las estructuras de control, con la finalidad de familiarizarse con ellas.
Así se propone:

1) Imprimir:
+-----
++----
+++---
++++--
+++++-
++++++

2) Leer n (entero entre 1 y 9) y mostrar:
1
22
333
...
nnnnn

3) Imprimir:
****/
***/*
\/
/
*/***
/****\

6.2 Asociación: Persona y Contrato

Supongamos que se definieron la clase Persona y la clase Contrato, cuyas instancias representan contratos de alquiler. Inicialmente, se escribió el siguiente código:

```
package empresa;
public class Persona{
	private String nombre;

	public void setNombre(String unNombre) {
		nombre = unNombre;
	}
	public String getNombre() {
		return nombre;
	}
}

package empresa;
public class Contrato{
```

```
        private int monto;
        private Persona cliente;

}
```

Observar que en la clase Contrato la vinculación al cliente respectivo es a través de una variable de instancia de tipo Persona, es una asociación entre las clases. Gráficamente la jerarquía se ve según la siguiente figura (Ilustración 12 Persona y Contrato):

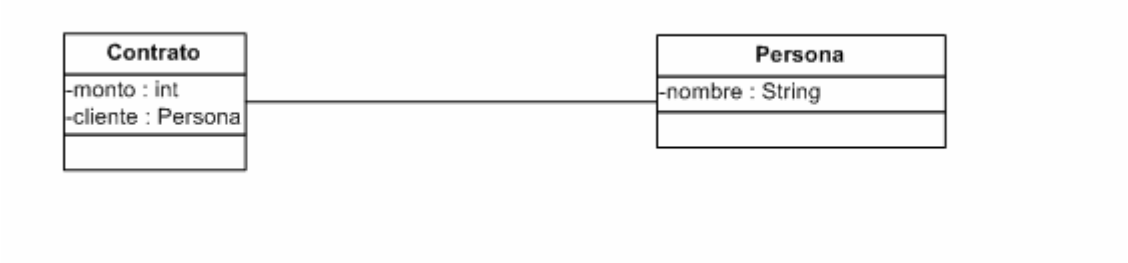


Ilustración 12 Persona y Contrato

Responder el siguiente cuestionario y completar el código requerido:

1) ¿Qué tipo de variable es *monto*?¿Por qué?

2) ¿Qué representa *cliente* en la clase Contrato?

3) Agregar los métodos de acceso y modificación de variables de instancia en la clase Contrato.

4) Reescribir el método de impresión de Persona para que muestre el nombre.

5) Reescribir el método de impresión de Contrato para que muestre el monto y el cliente.

6) Agregar un método en la clase Contrato para poder comparar dos contratos por su monto.

7) Agregar constructores en ambas clases.

8) Crear una clase de prueba que ejemplifique el uso de las clases anteriores.

9) Agregar año de vencimiento al contrato.

10) Indicar si un contrato está vencido a un cierto año.

11) Incrementar el monto en un porcentaje dado.

Una posible versión del código -en la que se incluyeron las respuestas a los planteos anteriores- es:

```
package empresa;

public class Persona {

        private String nombre;

        // pedido 7) Constructores
        public Persona(String unNombre){
```

```
                this.setNombre(unNombre);
        }
        public Persona() {
                this.setNombre("Sin definir");
        }

        public String getNombre(){
                return nombre;
        }

        public void setNombre(String unNombre){
                nombre=unNombre;
        }

        // pedido 4) Impresión de persona
        @Override
        public String toString(){
                return "Nombre: " + this.getNombre();
        }
}

package empresa;

public class Contrato {
        // pedido 1: monto es una variable de instancia, privada, representa el
        // monto del contrato. Es de instancia pues cada contrato tendrá su propio valor
        private int monto;

        // pedido 2: cliente corresponde al cliente del contrato. Observar que es de tipo
        // Persona.
        private Persona cliente;

        // pedido 9) agregar año de vencimiento
        private int añoVencimiento;

        // pedido 3: Acceso y modificación
        public Persona getCliente() {
                return cliente;
        }
        public void setCliente(Persona unCliente) {
                cliente = unCliente;
        }
        public int getMonto() {
                return monto;
        }
        public void setMonto(int unMonto) {
                monto = unMonto;
        }

        public int getAñoVencimiento() {
                return añoVencimiento;
        }
        public void setAñoVencimiento(int unAñoVencimiento) {
                añoVencimiento = unAñoVencimiento;
        }
```

```
	// pedido 5: Impresión de contrato
	@Override
	public String toString(){
		return this.getCliente() + " Monto: " +  this.getMonto();
	}
	// pedido 7: Constructores
	public Contrato(int unMonto, Persona unCliente){
		this.setMonto(unMonto);
		this.setCliente(unCliente);
	}

	// pedido 6: comparación de contratos por monto
	public boolean tieneMismoMonto(Contrato unContrato) {
		return this.getMonto()== unContrato.getMonto();
	}

	// pedido 10: indicar si un contrato está vencido a un cierto año
	public boolean estaVencido(int unAño){
		return this.getAñoVencimiento()<unAño;
	}
	// pedido 11: incrementar el monto de un contrato
	public void incrementar(int unPorcentaje){
		this.setMonto(this.getMonto()+ this.getMonto()* unPorcentaje/100);
	}
}

package empresa;

public class Prueba {

	public static void main(String[] args) {
		// pedido 8: clase de Prueba para ejemplificar uso
		Persona p1=new Persona();
		p1.setNombre("Juan");

		Persona p2=new Persona();
		p2.setNombre("Ana");

		Contrato c1=new Contrato(200, p1);
		c1.setAñoVencimiento(2000);
		Contrato c2=new Contrato(400, p2);
		c2.setAñoVencimiento(1995);

		System.out.println(c1);
		c1.incrementar(20);
		System.out.println(c1);

		System.out.println(c2);

		p1.setNombre("Pedro");
		System.out.println(c1);
```

```
                if(c1.estaVencido(2005)){
                        System.out.println(c1+  “está vencido");
                }

                if(c2.estaVencido(1990)){
                        System.out.println(c2+ “ está vencido");
                }
        }
}
```

6.3 Ejemplo: Temperatura

Este ejemplo tiene como objetivos: definir una clase, discutir posibles diseños, mostrar una implementación y analizar los problemas que tiene.

Quiero manejar temperaturas -en celsius y fahrenheit- pero no solamente como números. Quiero poder crear objetos, comparar, imprimirlos. Así, me gustaría poner en un programa:

```
Temperatura t1 = new Temperatura (100, ‘c’);
Temperatura t2 = new Temperatura (212, ‘f’);
```

Poder asimismo realizar comparaciones:

```
if (t1.equals (t2)).....
```

También queremos que se impriman adecuadamente: 100 grados centígrados. Necesitaré definir una escala por defecto para la impresión.

Para comparar, debo llevar a la misma unidad. Eso implica definir cómo me conviene almacenar la temperatura.

¿Cómo los puedo almacenar? Analizaremos distintos diseños. Podría internamente:
- llevar 3 números y 3 unidades: un posible problema es la redundancia;
- un número y su unidad: problema: deberá convertir muchas veces; ó
- llevar siempre en una única unidad, por ejemplo, celsius.

6.4 Código de Temperatura

Analizar el siguiente código.

```
package temperatura;
public class ClasePruebaTemperatura {

public static void main (String args[] ) {
   Temperatura unaT, unaT2;
   Temperatura. setEscalaPorDefecto("Celsius");
   unaT = new Temperatura(100,'c');
   unaT2 = new Temperatura(212,'f');
   System.out.println("Comparo y debe ser verdadero:  "+unaT.equals(unaT2));
   System.out.println("La primera temperatura es "+unaT);
   unaT2.setCelsius(200);
   System.out.println("Comparo Primera temperatura con la segunda y es
"+unaT.equals(unaT2)); // sale false
   unaT.setCelsius(100+60+40);
   System.out.println("Luego de cambiar el valor "+ unaT.equals(unaT2)); // sale true
```

```
    unaT2.setKelvin(373);
    unaT.setCelsius(100);
    System.out.println("Debe ser verdadero: " + unaT.equals(unaT2)); // sale true
}
}

package temperatura;
public class Temperatura {
  // variable de clase: Escala por Defecto
  private static String EscalaPorDefecto;
  // variables de instancia: Celsius
  private double celsius;
  // constructores
  public Temperatura() {
    this.setCelsius(0.00);
  }
  public Temperatura(float unaT, char unidad) {
    switch (unidad) {
      case 'f': {this.setFahrenheit(unaT); break;  }
      case 'c': {this.setCelsius(unaT); break;    }
      case 'k': {this.setKelvin(unaT);break; }
     }
   }
  // acceso y modificación var de clase
  public static void setEscalaPorDefecto (String unaE) {
    EscalaPorDefecto = unaE;
  }
  public static String getEscalaPorDefecto() {
    return EscalaPorDefecto;
  }
   // acceso y modificación var instancia celsius
  public double getCelsius() {
    return celsius;
  }
  public void setCelsius(double unaC) {
    celsius = unaC;
  }
  public double getFahrenheit() {
    return this.getCelsius()  * 9 / 5 + 32;
  }
  public void setFahrenheit(double unaF) {
    this.setCelsius ((unaF - 32)*5/9);
  }
  public void setKelvin (double unaK ) {
    this.setCelsius(unaK - 273.00);
  }
  public double getKelvin() {
    return this.getCelsius() + 273.00;
  }
  // comparación
  public boolean equals(Temperatura unaT) {
        return unaT.getCelsius() == this.getCelsius();
  }
```

```
  // Impresión
  @Override
  public String toString() {
    String texto;
    texto = "";
    if (Temperatura.EscalaPorDefecto.equals("Celsius")) {
      texto= this.getCelsius() + " grados celsius";
    }
    if (Temperatura.EscalaPorDefecto.equals("Fahrenheit")) {
       texto= this.getFahrenheit() + " grados fahrenheit";
     }
    if (Temperatura.EscalaPorDefecto.equals("Kelvin") ){
        texto=this.getKelvin() + " grados kelvin";
     }
    return texto;
}
}
```

¿Qué problemas y, o, elementos nuevos tiene? Algunos son:

- no es un buen diseño poner 'c' o 'f', pues quien utilice la clase no tiene forma de saber (salvo previa inspección del código) qué letra debe usar;
- la variable de clase está sin inicializar (escala por defecto);
- el nombre de la variable de clase abarca mucho: "escala por defecto", no es claro que es por defecto para la impresión;
- aparece la sentencia *switch*. La veremos detalladamente enseguida;
- se recomienda por claridad siempre poner una única sentencia en cada línea;
- se usan tipos de parámetros diferentes para la misma cosa: a veces *float*, a veces *int*. No es conveniente hacer esto;
- la firma del método *equals* será revisada luego.

6.5 Sentencia switch

El formato genérico de la sentencia switch es:

```
   switch (variable) {
     case a: {
           ....
           break; }
     case b: {
           ....
           break; }
     case c: {
           ....
           break; }
           ....
     }
   }
```

variable refiere a una variable de tipo int, char, long, short o byte. *a, b,* y *c* refieren a posibles valores de la variable dada. Se pueden poner más opciones. El funcionamiento de esta estructura es: evalúa el valor de la variable, en función de este resultado ejecuta la opción a, b,… según el valor. Es sumamente recomendable poner *break* en cada salida porque sino sigue ejecutando las otras opciones. Se verá más adelante un ejemplo detallado.

6.6 Práctico 5

Práctico No. 5
Tema: Clases y objetos

1. Crear la clase Triángulo. Debe incluir los siguientes métodos que devuelven un valor booleano:
a) esEscaleno b) esIsósceles c) esEquilátero d) tieneAnguloRecto
Agregar el código necesario para probarla.

2. Un club tiene socios.
a) Crear la clase Socio con variables de instancia: nombre, número y variable de Clase: PróximoNúmero.
b) Agregar los métodos de acceso y modificación
c) Inicializar en 1 el próximo número.
d) Crear un socio, mostrar sus datos
e) Crear otro socio, mostrar sus datos

3) Crear la clase Funcionario con atributos nombre y sueldo.
Incluir métodos de acceso y modificación. Agregar dos métodos de clase para prueba:
- ejemplo1: crea un funcionario con datos a ingresar por el operador y lo muestra
- ejemplo2: crea dos funcionarios (usando el ejemplo1) e indica además cuál gana más.

4) Agregar teléfono en la clase funcionario. Hacer que el método toString lo incluya.

5) Crear la clase Distancia que se puede usar para especificar distancias en kilómetros, metros y millas. Debe ser posible crear objetos en cualquier unidad e imprimirlas en cualquier otra unidad. Debe ser posible comparar, sumar y restar objetos.

Semana 7

7.1 Ejemplo: Monedas

El objetivo es diseñar una clase para manejo de monedas y ver diferentes opciones. (Esta clase se usa luego con otro ejemplo que trata de Cuentas Bancarias).

Cuando se comienza a analizar una clase, debe definirse qué características y qué comportamientos esperamos que tenga, qué cosas debe poder hacer.

Las monedas sirven para representar montos, como por ejemplo el saldo de una cuenta bancaria o el precio de un auto. Pueden ser positivas (ganancia) o negativas (pérdida). Deben poderse imprimir, se deben poder sumar y restar, aunque no tendría sentido que se pudiera dividir o multiplicar una moneda por otra. También debe ser posible compararlas. Podría analizarse si se desea llevar monedas de cualquier tipo o específicamente dólares, por ejemplo. En este caso consideraremos monedas en dólares.

Querría poder utilizar en un main la clase Moneda así:

```
public class ClasePruebaMoneda {
  public static void main (String args[]){
    Moneda m1, m2;
    m1 = new Moneda(276);
    m2 = new Moneda(1,23);
    System.out.println(m1);
    System.out.println(m1.sumar(m2));
  }
}
```

En una primera implementación de la clase Moneda podría decidirse llevar los dólares por un lado y los centavos por otro. Así se podría tener:

```
package moneda;

public class Moneda {
  //
  // 1era opción: llevar por separado dólares y centavos
  //

  // variables de instancia
private int dolares;
private int centavos;

  // constructores
public Moneda() {
    this.setDolares(0);
    this.setCentavos(0);
}

  public Moneda (int unMontoDolar, int unMontoCentavos) {
```

```
    this.setDolares(unMontoDolar);
    this.setCentavos(unMontoCentavos);
   }

   // acceso y modificación
   public int getCentavos() {
    return centavos;
   }

   public int getDolares() {
    return dolares;
   }
   public void setCentavos(int unMontoCentavos) {
    centavos = unMontoCentavos;
   }
   public void setDolares(int unMontoDolares) {
    dolares = unMontoDolares;
   }

   // impresion
  @Override
   public String toString() {
    return "$"  + this.getDolares()+ "."+this.getCentavos();
   }

   // suma
   public Moneda sumar(MonedaS unaMoneda) {
    return new Moneda(this.getDolares()+unaMoneda.getDolares(),
        this.getCentavos()+   unaMoneda.getCentavos());
   }
}
```

Observemos en particular qué ocurre cuando se crea una moneda con 10 dólares y 4 centavos. ¿Cómo se imprimirá? Saldrá $10.4, en vez de $ 10.04. Es necesario ajustar el método *toString*. También analicemos qué ocurre si sumamos 5 dólares 90 centavos con 4 dólares 50 centavos. Daría 9 dólares con 140 centavos. Es necesario entonces corregir el método de suma. Pensando a futuro, cuando se implemente el método para restar será necesario incluir la lógica para contemplar el caso de "pedirle" a los dólares, por ejemplo, si se desea restar 5 dólares 90 centavos de 10 dólares, 3 centavos.

Ante la complejidad que está tomando el diseño de esta clase es bueno replantearse si esta forma de almacenar por separado dólares y centavos resulta apropiada. Otra alternativa sería llevar "todo junto", en centavos:

```
package moneda;

public class Moneda {
  private int totalCentavos;

  public Moneda() {
    this.setTotalCentavos (0);
  }
  public Moneda(int centavos) {
    this.setTotalCentavos (centavos);
```

```
  }
  public Moneda (int dolares, int centavos) {
    this.setTotalCentavos (dolares * 100 + centavos);
  }

  private int getTotalCentavos() {
    return totalCentavos;
  }

   private void  setTotalCentavos(int unaCantidad) {
    totalCentavos = unaCantidad;
  }

  public int getDolares() {
    return this.getTotalCentavos() / 100;
  }
  public int getCentavos() {
    return this.getTotalCentavos() % 100;
  }
  @Override
  public String toString() {
   String dato = "";
   if (this.getCentavos()<10){
      dato = "0";
   }
    return "$"  + this.getDolares()+ "."+dato+ this.getCentavos();
  }

  public Moneda sumar(Moneda unaMoneda) {
    return new Moneda(this.getTotalCentavos()+ unaMoneda.getTotalCentavos());
  }

}
```

En esta versión ya se realizó el ajuste al método *toString*. Observar que el método de suma es más simple que en la otra versión. Observar además que los métodos de acceso y de modificación de totalCentavos son privados pues no se desea que se acceda o modifique este atributo desde fuera de la clase.

Se deberían agregar métodos para comparar y para realizar otras operaciones como por ejemplo multiplicar por un escalar o restar monedas. Así:

```
  public boolean mayorIgual(Moneda unaMoneda) {
    return this.getTotalCentavos() >= unaMoneda.getTotalCentavos();
  }
  public Moneda restar(Moneda unaMoneda) {
    return new Moneda(this.getTotalCentavos()- unaMoneda.getTotalCentavos());
  }
  public Moneda multiplicar(int unaCantidad) {
    return new Moneda(this.getTotalCentavos()* unaCantidad);
  }
  public Moneda multiplicar(float unaCantidad) {
    return new Moneda((int) (this.getTotalCentavos()* unaCantidad));
```

```
}
```

Como "moraleja" de este ejercicio, cuando se resuelve un problema muchas veces se toma la primera alternativa que se nos ocurre (aquí fue llevar por separado dólares y centavos) que eventualmente no resulta ser la más adecuada. Un análisis más profundo de la solución nos lleva a pensar en alternativas. No hay una forma única de resolver los problemas, pero es recomendable en general pensar varias alternativas antes de ponernos a programar efectivamente.

7.2 Práctico 6

Práctico No. 6
Tema: Repaso

1) Para que un objeto responda a un pedido, hay que enviar un ..... y el objeto ejecuta un .....
Los objetos son ..... de una .....
Las clases se organizan en ....., donde se comparten comportamientos y datos a través del mecanismo de la .....

2) En el siguiente extracto de código:

```
public class Club{
  private static int Ultimo;  // línea 1
  private String nombre;   // línea 2
            public static int getUltimo() {
                              return Ultimo;
           }
           .....
           }
```

explicar las dos líneas resaltadas.

3) Se desea calcular el área de un trapecio a partir de las bases y altura. Escribir un programa en Java que solicite las bases y altura y muestre el área. (Fórmula: (base mayor + base menor) * altura / 2)

4) Sea el salón de clases, en horario del curso. Identificar las clases, con sus atributos y métodos.

7.3 Prueba sobre objetos y clases

1. Las Clases contienen principalmente _____ y _____ y los _____ son instancias de las mismas.

2. Queremos realizar el manejo de funcionarios de un supermercado, de los mismos se conoce su nombre y documento de identidad. Por lo tanto será necesario una Clase _____, con los atributos _____ y _____

3. Se tiene una Clase Avión con tres atributos: velocidadMaxima, nombreCompañia (nombre de la Compañía Aérea al que pertenece) y altitudMáxima. Los valores por defecto son:

velocidadMaxima = 1200km/h
nombreCompañia = “EmpresaXX”

altitudMaxima = 10000 m

Existe un constructor sin parámetros que inicializa los valores anteriores y otro con parámetros.

La clase posee los siguientes métodos:

```
setVelocidadMaxima(int unaVelocidadMaxima)
getVelocidadMaxima()
setNombreCompañia(String unNombreCompañia)
getNombreCompañia()
setAltitudMaxima(int unaAltitudMaxima)
getAltitudMaxima()
```

a) ¿ Cómo se crea un Objeto llamado "elAvion" de la Clase Avión?, Indique la(s) correcta(s):

```
a1) Avion elAvion = new Avion();
a2) elAvion Avion = new Avion();
a3) Avion elAvion = Avion();
a4) Avion elAvion = new Avion(1200,"EmpresaXX", 10000);
```

b) El objeto "elAvion" modifica su velocidad máxima a 1350 km/h y la altitud máxima a 15000 m, escriba que métodos son necesarios invocar y cuales deben ser los parámetros requeridos.

c) Supongamos que se crean dos Objetos de la Clase Avión llamados "elAvion1" y "elAvion2". Indicar qué imprime el siguiente código:

```
elAvion1.setVelocidadMaxima(1500);
elAvion2.setVelocidadMaxima(1400);
elAvion1.setVelocidadMaxima(1550);
System.out.println(elAvion1.getVelocidadMaxima() );
System.out.println(elAvion2.getVelocidadMaxima());
```

d) Supongamos que se crean tres nuevos objetos de la clase Avión denominados "elAvion3", "elAvion4" y "elAvion5" y luego se realizan las siguientes invocaciones:

```
elAvion3.setNombreCompañia("Iberia");
elAvion4.setNombreCompañia ("Pluna");
elAvion5.setNombreCompañia ("Iberia");
```

NOTA: Suponer la siguiente definición del método esDeLaMismaCompañia dentro de la Clase Avion:

```
public boolean esDeLaMismaCompañia(Avion unAvion) {
        return (unAvion.getNombreCompañia().equals (this.getNombreCompañia());
}
```

Indicar que imprimen los siguientes códigos:

```
d1) if (elAvion3.esDeLaMismaCompañia(elAvion4)){
        System.out.println("Son iguales");
    }
    else{
        System.out.println("Son diferentes");
    }

d2) if (elAvion3 == elAvion5){
        System.out.println("Son iguales");
     }
```

```
    else{
        System.out.println("Son diferentes");
    }

d3) if (elAvion3.esDeLaMismaCompañia(elAvion5){
        System.out.println("Son iguales");
    }
    else{
        System.out.println("Son diferentes");
    }
```

Solución:
1) métodos, atributos, objetos
2) Funcionario, nombre, documento
3) a) a1 y a4; b) elAvion.setVelocidad(1350); elAvion.setAltitudMaxima(15000); c) 1550, 1400 d) d1) diferentes, d2) diferentes, d3) iguales

Semana 8

8.1 Prueba de Programas

Los programas se prueban para ver si están mal. Si se hace una prueba y no encontré ningún error, probablemente esté mal probado. Los principios que guían la prueba de programas son:

- Definir los resultados esperados: tener definidos claramente qué es lo que se quiere probar, qué se ingresará y qué se espera obtener.
- Otra persona que pruebe: los programas no deberían ser probados sólo por quien los hace. Es altamente recomendable que otra persona pruebe en forma independiente el programa.
- Ver a fondo cada salida o resultado. Debe verificarse cuidadosamente cada una de las salidas o resultados obtenidos.
- Probar casos esperados y correctos así como inesperados o incorrectos. Por ejemplo, si se está probando el ingreso de la edad de un funcionario y el rango es entre 18 y 65 años, hay que probar ingresar valores correctos, como 20 o 45; valores de borde: 18, 65; y valores no válidos, como por ejemplo 17, -3, 66, 100.
- Ver que el programa no haga cosas "de más". Por ejemplo, que el sistema luego de calcular los sueldos de los funcionarios borre todos los datos…
- Las pruebas deben ser completas. Siempre debe probarse en su totalidad.
- Pensar en probabilidad de más errores. Hay que pensar que si "pude cometer x errores, ¡quizás haya x+1!".

8.1.1 Estrategias de Prueba de Programas

Planteemos la siguiente analogía: se tiene un reloj de agujas y se desea saber si funciona bien. Una alternativa es mirar la hora que marca, esperar una hora y ver nuevamente la hora. O sea, considerar el reloj como una "caja negra" y analizar las salidas. Otra opción es abrir el reloj y estudiar si todos los engranajes funcionan bien. Es una prueba de "caja blanca". Estos dos enfoques, de caja negra y de caja blanca, se aplican para la prueba de programas.

8.1.1.1 Caja Negra

Para hacer pruebas de caja negra, se elaboran datos de prueba, indicando que es lo que se quiere probar, qué se espera obtener y qué se obtiene realmente. Siempre se debe indicar el valor con el que efectivamente se realizó cada prueba.

Así, en el caso de la edad del funcionario se podría armar una tabla como la de la Ilustración 13 Tabla de datos de prueba:

Qué se quiere probar	Qué se espera obtener	Qué se obtiene
se quiere ingresar un funcionario de edad 24, para lo cual se digita el valor 24 al solicitar la edad	que informe que está correcto	se acepta el resultado
se quiere incorrectamente ingresar un funcionario de edad negativa, por ejemplo -3	que informe que no es correcto	se obtiene lo esperado

Ilustración 13 Tabla de datos de prueba

8.1.1.2 Caja Blanca

La prueba de caja blanca implica una revisión "dentro" del programa. Se define cobertura como el grado de revisión del programa. Se puede hacer cobertura de sentencias, de decisiones o caminos. En el caso de sentencias, se trata de asegurar que toda sentencia es ejecutada al menos una vez. En el de decisiones se debe verificar que toda decisión alguna vez sea verdadera y alguna vez sea falsa. En la de caminos, se trata de chequear que todos los caminos posibles en el programa se dan alguna vez.

8.1.2 Prueba de métodos

Cuando se diseña un método, se puede realizar de dos formas: tratar de desarrollar desde el comienzo la versión lo más completa posible o realizar aproximaciones sucesivas al método final, desde una versión inicial simplificada. Así, se podría realizar una primera versión que devuelva, por ejemplo, un valor constante, un cálculo mínimo o que le solicite al operador el valor. Cuando se prueban los métodos debe tenerse en cuenta si ya es la versión final o una intermedia. En el caso de tener versiones intermedias preliminares se tiene la ventaja de que se puede ir avanzando en diferentes tareas y luego se completa; la desventaja es que interpretar los resultados puede ser más complicado debido a esos valores ficticios. En el caso que se opte por poner la versión definitiva, tiene la ventaja de que es más fácil revisar las salidas pero puede demorarse más en tener la versión completa del sistema.

8.1.3 Pasos para encontrar errores

La depuración de un programa es algo no sistemático y no garantizado. Los pasos sugeridos para encontrar/corregir errores son:

- Determinar el error;
- Ubicarlo en el programa;
- Corregirlo; y
- Verificarlo.

Se recomienda:
- Llevar un registro: ir anotando qué cosas se han probado, qué falta, qué se descarta, etc.
- Trabajar con listado del código fuente actualizado: en la medida que se realizan modificaciones al programa y se van realizando anotaciones en los listados impresos, se complica entender o descubrir errores, pues la versión en papel difiere con la real en la máquina.
- Guardar la versión anterior, antes de hacer "cambios geniales". Recordar guardar una versión del código completo hasta ese momento.
- Pensar en la posibilidad de reescribir: a veces es más fácil y seguro reescribir el método problemático que seguir corrigiendo errores sobre el mismo.
- Aprender de los resultados negativos: cuando se supuso que un error estaba en determinado lugar y se comprueba que no es así, es un resultado negativo pero a su vez permite descartar ese lugar como posible fuente del error.

¿Cómo se buscan?
- Pensar: analizar cuidadosamente qué es lo que pasa, en qué casos ocurre, descubrir la situación que lo genera.
- Descansar: luego de dedicar un rato a tratar de entender que es lo que pasa, conviene "alejarse" del código para luego, al retornar descansado, seguramente se encontrará el error;
- Pedir auxilio: recurrir a la ayuda de otro programador.
- Usar herramientas (*debugger*): hay software que permite realizar el seguimiento paso a paso del programa, mostrando las variables y la secuencia de instrucciones que se van ejecutando. Esta herramienta es útil si se ha pasado por las etapas de pensar, descansar y pedir auxilio. De

otra manera, toda la información que genera la herramienta no nos será aprovechable, pues no estamos lo suficientemente familiarizados con el programa.
- Experimentar: si nada de lo anterior funciona... probar cambiar algo "por las dudas"...

Al corregir errores, pensar en la posibilidad de múltiples errores. Arreglar los errores, no los síntomas. Tener previsto que la corrección también puede ser errónea. Rediseñar si es necesario.

8.2 Práctico 7 - Repaso de Conceptos

Práctico No. 7
Tema: Clase Camión

Considerar la siguiente definición de la clase Camión:

```
public class Camion {
  // variables de instancia
  private int chapa;
  private String color;
  // variable de clase
  private static int Año;

  // constructores
  public Camion (){
    this.setColor("Sin color");
    this.setChapa(0);

  }
  public Camion(String unColor, int unaChapa) {
    this.setChapa (unaChapa);
    this.setColor(unColor);
  }

  // metodos de acceso y modificacion
  public void  setColor(String unColor) {
    color = unColor;
  }
  public String getColor() {
    return color;
  }
  public void  setChapa(int unaChapa) {
    chapa = unaChapa;
  }
  public int getChapa() {
    return chapa;
  }
// impresion
  @Override
  public String toString() {
    return "Un camion de color "+this.getColor()+
        " y chapa "+this.getChapa();
  }

  // comparacion
  public boolean mismoColor(Camion unCamion) {
    return (this.getColor() == unCamion.getColor());
```

```
  }

  // metodos de clase
  public  static void setAño(int unAño) {
    Año = unAño;
  }

  public static int getAño(){
    return Año;
  }
}
```

1) ¿Cuál es la función del constructor Camion (String unColor, int unaChapa) ? Ejemplificar, anotando de dos formas diferentes el código de prueba necesario para crear un camión de color rojo y chapa 123.

2) El camión creado en el punto anterior es pintado de verde. Anotar el código que refleje esto.

3) ¿Qué puede representar la variable de clase Año? ¿Cómo se puede inicializar?

4) ¿Cómo se puede asegurar que la chapa, una vez que fue asignada, no se modifique?

5) Se quiere registrar si un camión está a la venta. Agregar el código y ejemplos de su uso.

6) Guardar el modelo del camión (año de construcción).

7) Agregar un método que permita saber si al camión le toca revisión (si el año del camión es anterior al Año, le toca).

8) ¿Cuántos camiones fueron creados en esa clase? Analizar cómo se puede llevar esta información.

9) ¿Qué pasaría si en el método toString se cambia la definición de public por private?

10) Crear la clase Motor. Un motor tiene como atributos la cilindrada y el tipo de combustible.

11) La impresión del motor debe mostrar todas sus características. Implementar el método adecuado.

12) Agregar al camión un motor. Indicar el código necesario para acceder y modificar dicho atributo. Anotar también código para probarlo.

13) La impresión del camión debe incluir también el detalle del motor.

14) Al camión creado en 1), ponerle que el motor es a gasoil.

Notar que la implementación de la asociación se realiza a través de objetos enteros, nunca por "parte" de objetos (como por ejemplo el nombre o número del objeto). Así, para asociar el camión y el motor, la clase Camión puede incluir una variable de tipo Motor. Si se asocia incorrectamente en la clase Camión un atributo de motor, como por ejemplo, el número de motor, se pierde la vinculación al objeto en sí (Ilustración 14 ¡Siempre referir al objeto entero!).

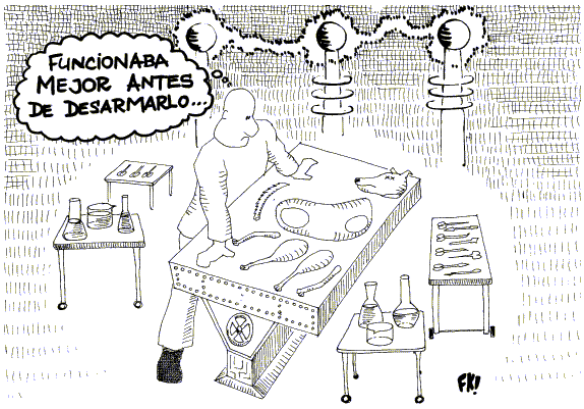


Ilustración 14 ¡Siempre referir al objeto entero!

8.3 Herencia

Conceptualizaremos acerca de herencia. En el ejemplo de la biblioteca aparecían las relaciones:
- Asociación (Préstamo-Material-Socio), visto con el ejemplo del Camión y Motor.
- Generalización-especialización: Persona-Socio-Funcionario.
- Agregación (se verá la semana próxima)

Ahora veremos en detalle la relación de jerarquía o herencia: "ES UN". La finalidad de la herencia es re-usar código, compartir atributos y, o, métodos.

Una jerarquía es un ranking u organización de elementos con la finalidad de resaltar la relación entre ellos. Se comparte por mecanismo de herencia. Está la *parent class* (super clase) y la *child class* (sub clase).

8.3.1 Ejemplo: Material, Publicación y Libro

En el ejemplo de la biblioteca, teníamos las clases Material, Publicación, Libro, Revista y Video:

Material (número de inventario, título)
Publicación (cantidad de páginas)
Libro (ISBN)
Revista (ISSN)
Video (duración en minutos)

¿Cómo implemento en Java? Estudiar el siguiente código (que contiene errores para analizar):

```
package PaqueteHerencia;
public class Material {
   private String titulo;

   public Material() {
      titulo = "Sin titulo";
   }

   public String getTitulo() {
      return titulo;
   }

   public void setTitulo(String unTitulo) {
```

```
        titulo = unTitulo;
    }
    @Override
    public String toString() {
        return this.getClass() + this.getTitulo();
    }

}

package PaqueteHerencia;
public class Publicacion extends Material {
    private int cantidadPaginas;

    public Publicacion() {
        cantidadPaginas  = 1;
        titulo = "Sin titulo";
    }

    public int getCantidadPaginas() {
        return cantidadPaginas;
    }
    public void setCantidadPaginas(int unaCantidadPaginas) {
        cantidadPaginas = unaCantidadPaginas;
    }
    @Override
    public String toString() {
        return this.getClass()+this.getTitulo()   + "-"+this.getCantidadPaginas();
    }
}
package PaqueteHerencia;
public class Libro extends Publicacion {
    private String isbn;

    public Libro() {
          titulo = “Sin titulo”;
          cantidadPaginas = 1;
           isbn=”Sin ISBN”;
    }

    public void setISBN(String unIsbn) {
        isbn = unIsbn;
    }
    public String getIsbn() {
        return isbn;
    }
    @Override
    public String toString() {
        return this.getClass()+this.getTitulo()+     this.getCantidadPaginas()+this.getIsbn();
    }

}
package PaqueteHerencia;
public class ClasePrueba {
```

```
    public static void main(String[] args) {
        Material m = new Material();
        Publicacion p = new Publicacion();
        Libro l = new Libro();
        System.out.println(m);
        System.out.println(p);
        System.out.println(l);

    }

}
```

Algunas de las cosas que se observan son:
- se accede directamente a variables, no usa se usa métodos (ejemplo: en el constructor de Publicación).
- Aparece palabra *extends*. Indica que una clase extiende a otra, es subclase.

Si se intenta compilar este código, no anda. El problema es por intentar acceder a un atributo privado desde otra clase. Profundizaremos en la visibilidad.

8.3.2 Visibilidad en profundidad

Hay distintos modificadores de visibilidad. Los modificadores de visibilidad tienen dos aspectos:
- herencia: determina si un método o variable puede ser referido en una subclase;
- acceso: determina el grado de encapsulación, o sea el alcance en el cual el método o variable puede ser directamente referido. Todos los métodos y variables son accesibles desde la propia clase en que son declarados.

modificador	clases	métodos y variables
no poner ninguno	desde el package	heredado por subclases del mismo package accesible desde mismo package
public	visible de todos lados	heredado por toda subclase accesible de cualquier lado
private	no aplicable	no heredado por subclase no accesible desde ninguna otra clase
protected	no aplicable	heredado por todas las subclases (en cualquier lado) accesible desde clases del package

Probar pasar a *protected* y observar que anda. ¿Cómo se que querré usar directamente una variable en una subclase? La mejor opción es siempre usar métodos para acceder a las variables. En general se recomienda poner las variables privadas y agregar métodos públicos para su acceso.

8.3.3 Constructores en detalle

Observar los constructores. ¿Queda práctico tener que inicializar todo cada vez? Hagamos la siguiente prueba: agregar *writes de control* (sentencias de impresión que indican por dónde se está ejecutando) en los constructores. Por ejemplo, en el constructor de material agregar: *System.out.println("Estoy en el constructor de Material");*. En forma similar, agregar en Publicación y en Libro. Observar que automáticamente los llama cuando creo un libro. Modificar los constructores:

```
public Material() {
        this.setTitulo("Sin titulo");
}

// Constructor de Publicación:
public Publicacion() {
    this.setCantidadPaginas(1);
}

// Constructor de Libro:
public Libro() {
    this.setISBN("Sin ISBN");
}
```

Automáticamente invoca primero al constructor de la superclase.

8.3.4 toString y super

¿Ocurrirá lo mismo con el método *toString*? O sea, para imprimir, ¿el *toString* también llama al mismo método de la superclase? ¡No!. Solamente el constructor sin parámetros es llamado automáticamente. Aún así, no queda práctico tener que volver a poner todo. Además si después agrego un atributo, ¿debo modificar todo?

La forma de referirme a un método de la superclase es con la palabra reservada *super*. Reescribiremos el método *toString* con super. Como estamos sobreescribiendo, agregamos también @Override.

```
@Override
public String toString() {
        return super.toString() + "-"+this.getCantidadPaginas();
    }

// Método en clase Libro
@Override
public String toString() {
        return super.toString() + this.getIsbn();
    }
```

El uso de *super* debería ser solamente cuando es imperioso, o sea, cuando se desea acceder explícitamente al método respectivo de la superclase y se desea que la búsqueda de dicho método comience en la superclase, salteando el de la propia clase.

8.3.5 Clases Abstractas

Probar cambiar *public class Publicacion* por *public abstract class Publicacion.* ¿Se puede crear instancias de Publicación? No, si se indica que una clase es abstracta no se pueden crear instancias de ella. Se profundizará más adelante.

8.3.6 Object

En Java, toda clase hereda automáticamente de una clase denominada *Object*. No es necesario indicar que se extiende *Object*. La clase *Object* tiene un conjunto de métodos útiles que se heredan y que pueden ser redefinidos según las necesidades. Un par de métodos interesantes son el método *toString* de impresión ya referido y el método *equals*, utilizado para chequear si dos objetos "lucen" iguales.

8.3.7 Abstract

Una clase abstracta es una clase que no se espera que tenga instancias. Si intento crear instancias, no lo permite. Por ejemplo, queremos que la clase Material sea abstracta:

```
public abstract class Material {
```

8.3.8 Métodos abstractos

En el caso de Material, suponer que quiero mostrar un slogan para Libro, Revista, Diario, etc. Quiero que todas las clases tengan el método, pero en Material que no diga nada.

¿Cómo lo hago? Una solución es poner que en Material devuelva un *String* vacío y que en cada subclase ponga lo necesario. Funciona, pero no obliga a que lo ponga en todas las subclases.

Una opción mejor es poner en Material:

```
    public abstract String slogan();
```

Un método abstracto es un método que no contiene implementación. Si se intenta poner código, no compila. Obliga a que se indique que la clase es abstracta.

Las subclases no abstractas derivadas de una clase abstracta deben hacer el *override* (sobreescritura) de todos los métodos abstractos de la clase padre (si no se definen, la considera abstracta). O sea, el método abstracto obliga a definir en la subclase ese mismo método con una implementación.

En el ejemplo, la clase Publicación se pasa a *abstract* y en Libro se agrega:

```
    public String slogan() {
       return "¡Nada como un buen libro!";
     }
```

8.3.9 Final

Como opción opuesta, para evitar que se redefina un cierto método, puedo definirlo con *final*. Por ejemplo, si en la clase Material se indica:

```
    public final String general() {
            return "general";
     }
```

y se intenta redefirlo en Publicacion, dice que no puede ser sobreescrito.

8.3.10 Upcast, downcast y polimorfismo

8.3.10.1 Upcast

Sean las definiciones:

```
// Clase Persona:

package pruebaPolimorfismo;
public class Persona {
  private int edad;

  public void setEdad(int unaEdad) {
    edad = unaEdad;
  }
  public int getEdad() {
    return edad;
  }
  public boolean esMenorQue(Persona unaPersona) {
    return (this.getEdad()) <= unaPersona.getEdad();
  }
  public String mensaje() {
    return "Es Persona";
  }
}

// Clase Empleado: subclase de Persona

package pruebaPolimorfismo;
public class Empleado extends Persona {
  int sueldo;

  public void setSueldo(int unSueldo) {
    sueldo = unSueldo;
  }
  public int getSueldo() {
    return sueldo;
  }
  public String mensaje() {
    return "Empleado ";
  }
}
```

¿Funciona el siguiente código? ¿Qué imprime?

```
package pruebaPolimorfismo;

public class PruebaUp {
  public static void main(String[] args) {
    Empleado e1, e2;
    e1 = new Empleado();
    e1.setEdad(18);
```

```
    e2 = new Empleado();
    e2.setEdad(20) ;
    if (e1.esMenorQue(e2)) {
      System.out.println("E1 es menor" );
    }
    else {
      System.out.println("E2 es menor");
    }
  }
}
```

Notar que el método esMenorQue(Persona) recibe un objeto Persona, no Empleado.

¿Por qué funciona? Funciona debido al *upcasting*. Todo empleado es una persona, el hecho de considerar un empleado como persona se hace de forma automática y se llama *upcasting*.

La clase derivada es un super conjunto de la clase base, así puede contener más métodos pero debe tener al menos los de la clase base. Por eso el compilador deja hacerlo sin problemas.

En el ejemplo, Empleado es un tipo de Persona. Cualquier mensaje que acepte un objeto Persona, aceptará también un objeto Empleado.

8.3.10.2 Polimorfismo

¿Qué imprime el siguiente código?

```
Persona pers1, pers2;
persona1 = new Persona();
System.out.println(persona1.mensaje());
persona2 = new Empleado();
System.out.println(persona2.mensaje());
```

Imprime:

Es persona
Empleado

Al ejecutar, se da cuenta que la variable persona2 "mira" a un objeto Empleado, usa método de empleado. El método mensaje es polimórfico, usa el adecuado dependiendo del receptor.

Polimorfismo es una técnica por la cual una referencia que es usada para invocar a un método puede resultar en diferentes métodos invocados en diferentes momentos.

8.3.10.3 Downcasting

Ejemplo 1: Analizar qué pasa si se escribe en el main:

```
Empleado e3;
e3 = new Persona();
```

No compila pues una persona podría no ser empleado.

Ejemplo 2: Si creo un empleado y quiero setear su sueldo:

```
Persona p4;
p4 = new Empleado();
p4.setSueldo(100);
```

Así no funciona, informa algo similar a que el "método setSueldo no está en Persona". Para Java es un objeto Persona, aunque en realidad es un Empleado. Para indicarle que lo mire como empleado es necesario hacer *downcast*. Para ello, se especifica entre paréntesis el nombre de la clase.

```
((Empleado)p4).setSueldo(100);
```

Ejemplo 3: ¿Funciona? ¿Qué sale?

```
Persona p2 = new Persona();
Empleado e3 = (Empleado) p2;
System.out.println(e3.mensaje());
```

Da error en ejecución, no puede hacer el *cast* de Persona a Empleado, pues no toda persona es empleado.

8.4 Práctico 8

Práctico No. 8
Tema: Herencia

1) Definir la clase Estudiante como subclase de Persona. El estudiante tiene además número.
Verificar que los métodos y variables de instancia definidos como protected se heredan.
Probar con private y public.

2) En el práctico 5 se definió la clase Funcionario. Extenderla para incluir mensuales y jornaleros (el cálculo del sueldo es diferente: el mensual tiene un valor fijo por mes, el jornalero tiene un valor diario y se le paga según la cantidad de días trabajados ese mes).
Agregar las clases, métodos y variables necesarias. Probarla.

3) Experimentar con la derivación simple: escribir un constructor de la subclase que no refiera explícitamente al constructor de la superclase. Explicar qué ocurre.

4) Los métodos y variables de clase, ¿se heredan?

5) Explicar, usando la API de Java, las siguientes líneas. Identificar packages, clases, objetos y métodos.

a) import java.util.*;
b) int number = Math.abs(-12) ;
c) System.out.println("hola");
d) BufferedReader in = new BufferedReader (new InputStreamReader(System.in));
e) Scanner in = new Scanner(System.in);

Semana 9

9.1 Repaso de Upcast, Polimorfismo, Abstract

Se disponen de las siguientes definiciones vistas la semana anterior:

Persona	
int edad	métodos setEdad (int unaEdad) int getEdad() boolean esMenorQue(Persona) String mensaje() (retorna "En Persona")
Empleado (subclase de Persona)	
int sueldo	métodos setSueldo(int unSueldo) int getSueldo() String mensaje() (retorna "En Empleado")

Indicar en cada uno de los ejemplos que pasaría:

		En el *main*:	¿Resultado? ¿Funciona?

1		Empleado e1;
		e1 = new Empleado();
		e1.setEdad(39);
			Funciona por herencia

2		Empleado e2 = new Empleado();
		e2.setEdad(25);
		if (e1.es MenorQue (e2)) {
		…
		}
			Funciona por *upcast*: todo empleado es Persona (Observar que el método esMenorQue está definido para recibir un objeto Persona).

3		Persona p1 = new Empleado(); System.out.println(p1.mensaje());	Compila (pues en Persona esta el método mensaje). En ejecución sale "En Empleado", porque es polimórfico
4		p1.setSueldo(100);	No funciona, en tiempo de compilación asume que p1 es Persona, no sabe que luego será empleado
5		(Empleado)p1.setSueldo(100);	Anda, pues se hizo explícitamente el *cast*
6	Agrego en Persona: public abstract class Persona		
		Persona p2 = new Persona();	No anda, si la clase es *abstract* no se puede crear instancias.
7	Agrego en Persona: public abstract int totalImpuestos();		
		Empleado e3 = new Empleado();	No funciona, debo incluir en Empleado el método totalImpuestos(), sino la clase queda *abstract*. Debería incluir el método en Empleado.

9.2 Colecciones: Ejemplo Banco

Trabajaremos ahora con cuentas bancarias. ¿Qué operaciones se puede hacer? Depósitos, retiros, transferencias, averiguar saldo, abrir, cerrar, etc. ¿Qué tipos de cuentas hay? Algunos son: Caja de ahorro, Cuenta corriente y Depósito a Plazo Fijo. En este caso consideraremos Caja de Ahorro y Cuenta Corriente. Una caja de ahorro permite depositar dinero y retirarlo, y da interés. Una cuenta corriente permite depósitos y retiros y también el uso de cheques. No da interés.

Analizaremos cómo organizar esas clases.

Una opción es poner Cuenta Corriente como subclase de Caja de Ahorro (Ilustración 15 Cuenta Corriente y Caja de Ahorro):

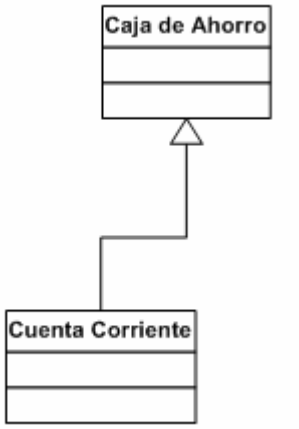


Ilustración 15 Cuenta Corriente y Caja de Ahorro

La desventaja en este caso es que la Cuenta Corriente tendría interés, lo que no es cierto.
Otra alternativa es (Ilustración 16 Caja de Ahorro y Cuenta Corriente):

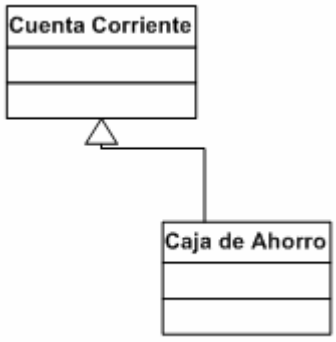


Ilustración 16 Caja de Ahorro y Cuenta Corriente

Aquí el problema es que Caja de Ahorro permitiría tener cheques, lo que es incorrecto.

Finalmente, otra alternativa es agrega una clase abstracta Cuenta, de la que deriven Caja de Ahorros y Cuenta Corriente (Ilustración 17 Cuenta).

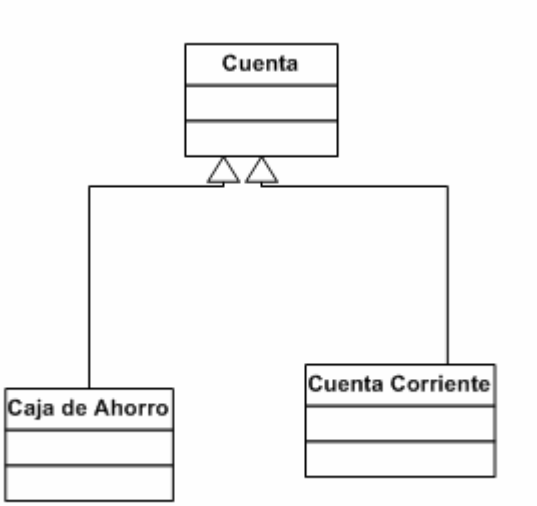


Ilustración 17 Cuenta

Este proceso es diseñar: ver qué clases necesito, analizar diferentes opciones y definir cómo se relacionan.

9.2.1 Clase Cuenta

El código de esta clase es:

```
package banco;
import moneda.*;
public abstract class Cuenta {
  // cada cuenta sabe el nombre de su propietario y saldo
  private String nombre;
  private Moneda saldo;

  public Cuenta() {
  this.setNombre ( "sin nombre");
  this.setSaldo ( new Moneda(0));
  }

  public void setNombre(String unNombre) {
    nombre = unNombre;
  }
  public String getNombre() {
    return nombre;
  }

  public Moneda getSaldo() {
    return saldo;

  }
  private void setSaldo(Moneda unSaldo){
    saldo = unSaldo;
  }

  public void depositar(Moneda unMonto) {
    this.setSaldo(this.getSaldo().sumar(unMonto));
```

```
  }

  public boolean retirar(Moneda unMonto) {
    boolean ok;
    ok = false;
    if (this.getSaldo().mayorIgual( unMonto)) {
      this.setSaldo(this.getSaldo().restar(unMonto));
      ok = true;

    }
    return ok;
  }
  @Override
  public String toString() {
    return "\n"+this.getNombre()+ " Saldo "+ this.getSaldo();}
}
```

Para poder utilizar la clase Moneda definida antes, se agrega un *import*.
Observar que no se definió el método "setSaldo()" como público. La razón de ponerlo privado es no permitir que desde afuera se modifique el saldo. El saldo puede variarse solamente a través de depósitos o de retiros.

9.2.2 Clase Caja de Ahorro

La Caja de Ahorro lleva una tasa de interés que es de clase. El banco otorga a cualquier caja de ahorro la misma tasa de interés. En este ejemplo, la consideramos fija en 12%. Como este valor es para toda caja de ahorro, es un atributo de la clase; no de instancia.

También se dispone del método acumularInteres, que acredita los intereses según la tasa (una versión simplificada).

```
package banco;
import moneda.*;
public class CajaAhorro extends Cuenta {
  // las cajas de ahorros tienen una tasa de interes
  private static float Tasa=0.12f;

  public CajaAhorro(Moneda unaMoneda) {
    this.depositar(unaMoneda);
  }

  public static void setTasa(float unaTasa) {
    Tasa = unaTasa;
  }
  public static float getTasa() {
    return Tasa;
  }
  public void acumularInteres() {
    Moneda aux;
    aux = this.getSaldo();
    aux = aux.multiplicar(CajaAhorro.getTasa() / 12);
    this.depositar(aux);
  }
  }
```

9.2.3 Prueba Caja de Ahorros

Para poder probar el código hecho hasta aquí, hacer una clase de Prueba:

```
package banco;
import moneda.*;
public class PruebaBanco {

  public static void main (String args[])  {
    // Prueba caja de ahorros
    CajaAhorro ca;
    ca = new CajaAhorro(new Moneda(200));
    ca.setNombre("Caja de ahorros de Ismael");
    ca.depositar(new Moneda(100));
    System.out.println("Antes de acumular interés" + ca);
    ca.acumularInteres();
    System.out.println("Saldo "+ca);

    if (ca.retirar(new Moneda(4442)) ){
      System.out.println("Retiro correcto!");
    }
    else {
      System.out.println("No se retira ");
    }

    System.out.println(ca);
   }
 }
```

9.3 Clase Cuenta Corriente: primera versión

Para llevar la cuenta corriente, querría llevar una bitácora de los cheques emitidos (en una primera versión simplificada). ¿Cómo llevo una lista? ¡Con ArrayList!. En la semana próxima se verá este tema.

Semana 10

10.1 ArrayList

Un *ArrayList* permite contener una colección de objetos. La clase *ArrayList* está definida en el paquete java.util.*.

Algunos métodos útiles son:

para agregar: *add*(unObjeto)
para sacar: *remove*(unObjeto)
para saber si contiene un elemento: *contains*(unObjeto)
para conocer el tamaño: *size*()

A modo de prueba, crearemos algunos objetos y los agregaremos en una lista. En el código se utilizan algunos métodos más de ArrayList.

```
import java.util.*;

public class PruebaArrayList {
   public static void main(String[] args) {
      // defino el ArrayList para contener objetos de tipo Camión
      ArrayList<Camion> lista = new ArrayList<Camion>();

      Camion c1 = new Camion();
      Camion c2 = new Camion();
      c1.setColor("rojo");
      c2.setColor("azul");

      // agrego elementos con add
      lista.add(c1);
      lista.add(c2);

     // para ver si esta vacio:
     System.out.println("esta vacio? "+ lista.isEmpty());


     // cantidad de elementos
     System.out.println("Tamaño "+lista.size());

     // ubicacion de un elemento determinado
     System.out.println("Posicion de c1"+lista.indexOf(c1));
     // para recuperar el elemento de una posicion: get
     System.out.println("Objeto de posicion 1"+lista.get(1));


     // puedo cambiar el elemento de una posicion
     //por otro dado
     lista.set(1, c1);
```

Para listar todos los elementos hay varias opciones. Una es mostrarlos todos juntos:

```
// Para listar todos los elementos
   // 1era opcion: listarlo todo junto
          System.out.println(lista);
```

Otra es recorrer a mano la colección:

```
// 2da. opción- recorrer a mano toda la coleccion
  for (int i = 0; i < lista.size(); i++) {
          System.out.println("Elemento del lugar" + i+ " " + lista.get(i));
       }
```

Una más es utilizar iteradores. Los iteradores permiten recorrer fácilmente una colección. Es necesario "cargar" el iterador y luego para recorrerlo se utiliza el método *hasNext()* (que retorna verdadero si hay más elementos) y el método *next()* que retorna el siguiente elemento. Luego que se "termina" de utilizar, es necesario recargarlo nuevamente.

```
// 3era. opción- usar iterator

Iterator<Camion> it = lista.iterator();
       while (it.hasNext()){
           Camion c = it.next();
                              .......
}
```

<Camion> le indica tanto al iterador como a la lista que los elementos que contendrá serán camiones. <> es un elemento incorporado por la versión de Java 5 y se denomina *generics*.

10.2 Clase Cuenta Corriente: segunda versión

La cuenta corriente lleva una lista de cheques. En esta versión simplificada, los cheques simplemente se representan por un *String*. Otra alternativa válida es definir una clase Cheque con atributos por ejemplo: fecha, monto, moneda.

```
package banco;
import moneda.*;
import java.util.*;

public class CuentaCorriente extends Cuenta {
   // las cuentas corrientes llevan un registro de los textos de los cheques (podria hacerse una
clase Cheque, para no ampliar ni complicar no se hace aquí).

   private ArrayList <String> listaCheques;

   public CuentaCorriente() {
          listaCheques = new ArrayList<String>();
   }
   public boolean emitirCheque(Moneda unaM, String unTexto) {
      boolean ok = false;
      if (this.retirar(unaM))  {
         // agrego el texto a la lista;
         this.agregarCheque(unTexto);
         ok = true;
```

```
        }
        return ok;
    }
    private void agregarCheque(String unT) {
        this.getListaCheques().add(unT);
    }

    public ArrayList<String> getListaCheques() {
        return listaCheques;
    }
    @Override
    public String toString() {
        return super.toString() +"\n"+ "***lista cheques "+this.getListaCheques();
    }
}
```

10.3 Agregación: Clase Banco

El Banco tiene una lista de cuentas. Para representar esta relación se utiliza el rombo. No diferenciaremos si el rombo es blanco o negro para el curso de Programación I.

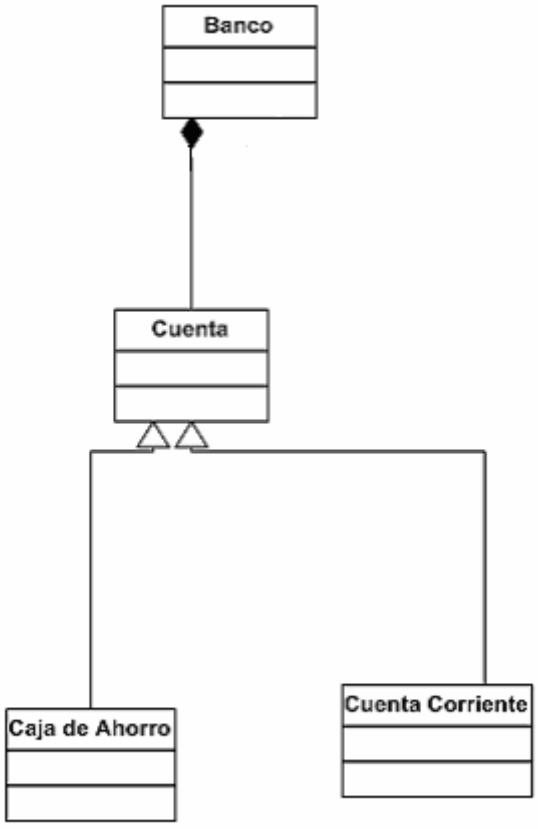


Ilustración 18 Banco

La relación entre la clase Banco y Cuenta es de agregación. El banco conoce a sus cuentas.

```
package banco;
import java.util.*;

public class Banco {
   // el banco tiene un nombre y una lista de cuentas
   private ArrayList<Cuenta> listaCuentas;
   private String nombreBanco;

   public Banco() {
      this.setNombreBanco("sin nombre");
      listaCuentas = new ArrayList<Cuenta>();
   }
   public void setNombreBanco(String unNombre) {
      nombreBanco = unNombre;
   }
   public String getNombreBanco() {
      return nombreBanco;
   }
   public void agregarCuenta(Cuenta unaCuenta) {
      this.getListaCuentas().add(unaCuenta);
   }
   public void eliminarCuenta(Cuenta unaCuenta) {
      this.getListaCuentas().remove(unaCuenta);
   }
   public ArrayList<Cuenta> getListaCuentas() {
      return listaCuentas;
   }
   @Override
   public String toString() {
      return this.getNombreBanco() +"\n"+ "lista cuentas " + this.getListaCuentas();
   }

}
```

Ejercicio: Agregar un método que retorne una lista de las cuentas con saldo 0. ¿Dónde se ubicará este método? Quien tiene toda la información de las cuentas es el Banco, por lo cual este método va en esa clase.

```
public ArrayList<Cuenta> cuentasSinSaldo() {
        ArrayList<Cuenta> lista = new ArrayList<Cuenta>();
        Iterator<Cuenta> it = this.getListaCuentas().iterator();
        while (it.hasNext()){
                Cuenta c = it.next();
                if (c.getSaldo().equals(new Moneda(0))){
                        lista.add(c);
                }
        }
        return lista;
}
```

10.4 Práctico 9

Práctico No. 9
Tema: Colecciones

1) Investigar las clases de java.util.

2) Crear un programa que permita probar la clase ArrayList (con generics).
En particular mostrar cómo:
- definir un ArrayList
- borrar el primer elemento
- ver si incluye un determinado objeto Moneda
- agregar el número entero 18
- agregar otros objetos
- verificar si está vacío
- recorrerlo e imprimir cada elemento

3) Se necesita un programa para llevar la lista de CD de una persona. De cada CD se guarda: titulo y número.
Hacer un programa en Java que permita guardar una colección de CD (utilizando ArryaList) y además sea posible: agregar un CD, indicar si está el CD de título "Mocedades 20 años", cambiar el número al CD de título dado, listar todos, eliminar un CD y listar todos ordenados por título. (Nota: La ordenación se ve en próximas semanas).

4) Definir las clases, con atributos y métodos apropiados para representar la siguiente situación:
Una empresa vende dos tipos de nylon: nacional e importado.
Se debe poder realizar las siguientes tareas:
- ingreso stock inicial de nacional e importado
- venta nacional (nombre cliente, metros, total)
- venta importado (nombre cliente, metros, total)
- listado de clientes
- listado de ventas nacionales
- listado de ventas importadas
- stock de nacional e importado

Analizar diferentes alternativas de diseño.

Semana 11

11.1 Arrays

Un *array* es una lista de datos del mismo tipo. Cada dato es almacenado en una posición específica y numerada del *array*. El número correspondiente a cada lugar se llama índice o posición.

Ejemplo:

lista (contendrá números enteros)

índice	dato
0	23
1	45
2	54
3	-15

La estructura de nombre *lista* contiene 4 elementos. El índice comienza en 0. Para obtener el valor de la segunda posición se debe indicar el nombre del *array* y la posición:

lista[1]

Para modificar ese valor, por ejemplo, por el número 115:

```
lista[1] = 115;
```

Este mismo *array* en Java se define:

```
int [] lista = new int[4];
```

o en forma equivalente:

```
int lista[] = new int[4];
```

Para saber el largo actual de la estructura se utiliza *length*.

11.2 Ejercicios básicos de arrays

Se incluye el ejercicio e inmediatamente una posible solución.

1) a) Anotar la definición en Java de un array de nombre sueldo para contener 10 datos de tipo float.
   b) Asignar el valor 13.45 a la cuarta posición del array
   c) Mostrar el valor de la cuarta posición del array

   Solución:
   a) float[] sueldo = new float[10];
   b) sueldo[3] = 13.45;
   c) System.out.println(sueldo[3]);

2) Indicar qué errores tiene el siguiente fragmento de código:

```
int[] horas = new int[3];
horas[0] = 12;
horas[3] = 4;
```

Solución:
Intenta acceder al índice 3, que está fuera del rango. Se caerá el programa.

3) Solicitar al operador una cantidad y definir un array de enteros de nombre **lista** para contener esa cantidad de datos.

Solución:

```
import java.util.*;
public class Prueba{
  public static void main(String args[]) {
        Scanner in = new Scanner(System.in);
        System.out.println("Ingresar tamaño");
        int tamano = in.nextInt();
        int lista[] = new int[tamano];     // defino el array

        // aquí irá el código de los siguientes ejercicios
  }
}
```

4) Cargar el array definido en el ejercicio anterior, pidiendo los datos al operador.

Solución: En el lugar indicado en el ejercicio anterior, agregar:

```
for (int i = 0; i < lista.length; i++) {
      System.out.println("Ingresar el dato número " + i);
      lista[i] = in.nextInt();
}
```

5) Mostrar el array recién cargado.

Solución:

```
for (int i = 0; i < lista.length; i++) {
      System.out.println("El dato número " + i + "  es " + lista[i]);
}
```

6) Mostrar el promedio del array anterior.

Solución:

```
// Promedio
float suma = 0;
for (int i = 0; i < lista.length; i++) {
      suma = suma + lista[i];
}
float p = suma / lista.length;
System.out.println("Promedio " + p);
```

7) Indicar el máximo valor y la posición en la que ocurre.

Solución:

```
// máximo
int max = lista[0];
int pos = 0;
for (int i = 1; i < lista.length; i++) {
        if (lista[i] > max) {
            max = lista[i];
            pos = i;
        }
}
System.out.println("El máximo es " + max + "y se encuentra en la posición " + pos);
```

8) Indicar si en el array se encuentra un valor que ingresará el usuario.

Solución:

```
// búsqueda de un dato particular
int dato = in.nextInt();
boolean termine = false;
boolean esta = false;
pos = 0;
while (!termine) {
            // verifico si ya recorrí todo
        if (pos == lista.length) {
            termine = true;
            esta = false;
        }
            // si está entre los elementos válidos, verifico si es el dato pedido
        if (pos < lista.length) {
            if (lista[pos] == dato) {
                esta = true;
                termine = true;
            }
        }
        pos = pos + 1;
}
if (esta) {
    System.out.println("Sí!" );
}
else {
     System.out.println("No!");
}
```

9) Ordenar en forma creciente los valores del array.

Solución:

```
// ordenación
for (int i = lista.length - 1; i >= 0; i--) {
        // se busca el máximo
        int posmax = 0;
```

```
        int valmax = lista[0];
        for (int j = 0; j <= i; j++) { // busco el más grande entre el 1er. elemento y el último considerado
            if (lista[j] > valmax) {
                valmax = lista[j];
                posmax = j;
            }
        }
       // el máximo encontrado debe ir al "final"
        // intercambio último con el de la posición donde encontré el máximo
        int aux = lista[i];
        lista[i] = lista[posmax];
        lista[posmax] = aux;

}
```

11.3 Prueba sobre Arrays

1. En un *array*, el valor del índice de la primera posición es:
   a) Depende de cómo fue inicializado el *array*
   b) 0
   c) 1
   d) -1

2. Si se tiene el *array*: int [] num = {1, 2, 3, 4, 5}, que inicializa con los respectivos valores. ¿Cuál de las siguientes afirmaciones es verdadera?
   a) num[0] vale 2
   b) El elemento en num[1] es 1
   c) num[5] no es válido
   d) Todas las anteriores

3. ¿Cuál de los siguientes *for* recorre completamente un array int[] enteros?
   a) for(int i = 0; i <= enteros.length; i++) {…}
   b) for(int i = 1; i < enteros.length; i++) {…}
   c) for(int i = 0; i < enteros.length; i++) {…}

4. En un *array* se puede almacenar:
   a) Sólo tipos primitivos
   b) Sólo objetos
   c) Sólo ints
   d) Ninguna de las anteriores

5. ¿Cómo se obtiene el tamaño de un array int[] unArray?
   a) unArray.size()
   b) unArray.length
   c) unArray.size
   d) unArray.tamaño

Las respuestas correctas son: 1) b, 2) c, 3) c, 4) d, 5) b.

11.4 Ejercicios avanzados de arrays

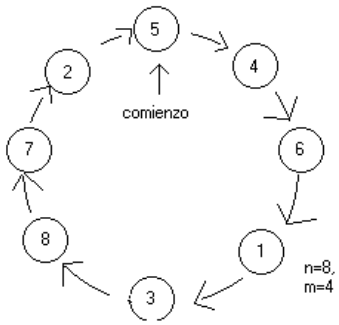
1) Indicar el máximo valor y, en el caso de que esté repetido, mostrar todas las posiciones en las que se presenta ese valor.

2) Indicar el "rango" de los valores del array, es decir, el valor mínimo y máximo.
3) Solicitar un valor e indicar la posición del elemento más cercano (por exceso o por defecto).
4) Invertir el array. Ej: si los datos cargados son: (10, 20, 4, 6), debe quedar: (6, 4, 20, 10).
5) Asumir que el array tiene valores positivos entre 1 y 15. Mostrar como histograma los datos del array. Ej: (5,4,1,2) desplegaría:
   5 *****
   4 ****
   1 *
   2 **
6) Leer un número y rotar esa cantidad de posiciones el array.

Ej: Si el array tiene como datos: (10, 20, 4, -5, 1) y se ingresa el número 2 se obtiene: (-5, 1, 10, 20, 4).

7) Suponiendo que el array esté ordenado, indicar cuál elemento se repite más veces. Ej: (10, 10, 20, 20, 20, 50), muestra 20. Si hay más de uno con la misma cantidad de repeticiones, mostrar el primero de ellos.
8) Suponer se dispone de dos arrays ordenados. Generar un tercer array con el resultado de la intercalación ordenada. Ej: sean los arrays: (10, 20, 50), (1, 2, 20, 24, 80), el resultado sería (1, 2, 10, 20, 20, 24, 50, 80)
9) ¡¡¡Un asesino serial anda suelto!!! Es una persona sistemática, que gusta de elegir asesinar a sus víctimas de acuerdo a un criterio matemático. Ha seleccionado una cantidad *n* de personas, les tomó fotos, y pegó esas fotos en un círculo. A partir de una cierta posición, cada *m*-ésima persona es asesinada y así va disminuyendo el círculo.

Por ejemplo, si *n*=8 y *m*=4, el turno en que serán ejecutadas es 54613872, es decir la primera persona del círculo original será asesinada en 5to. lugar, la segunda persona en 4to. lugar, etc. La última persona será la que estaba en el lugar 6to.



Leer n y m, e indicar cuál sería la mejor posición para estar ubicado en ese círculo – con la intención de estar en la última foto ¡¡y quizás así salvar su vida!!.

11.5 public static void main

El método *public static void main (String args[])* es el punto de entrada al programa. Es *público* porque debe ser posible accederlo desde fuera de la clase; es *estático* porque no es necesario tener una instancia para acceder al método, *main* es el nombre del método y *String args[]* refiere a que está recibiendo un array de String que es la lista de parámetros. Para reconocer cada uno de esos valores se puede hacer:

```
public static void main (String args[]) {
        if (args.length==0) {
            System.out.println("Sin parámetros");
        }
        else {
            for (int i = 0; i < args.length; i++) {
                System.out.println("El parámetro "+i + " vale "+args[i]);
            }
    }
}
```

Para enviar efectivamente los parámetros al *main*, al ejecutar se indica el nombre de la clase que tiene el *main* y se incluye la lista de valores separadas por uno o más espacios en blanco. Ejemplo: java Prueba valor1 valor2 valor 3

Semana 12

12.1 Ejercicio: Gastos de la casa

Este ejercicio plantea que se desea llevar los gastos de una casa. De cada gasto se sabe el día, monto y descripción. Armar un programa con menú que permita ingresar un gasto, obtener un listado detallado y la descripción del gasto mayor.

Los objetivos del ejercicio son:
analizar posibles diseños;
uso de *ArrayList*: búsqueda, máximo, ordenación;
manejar excepciones;
presentar el concepto de *Interface*;
menúes y *arrays*.
Se presentan los pasos a seguir para una posible solución.

12.1.1 Diseño

Cuando se resuelve un problema, se trata de identificar los conceptos o clases. También, para esas clases, se deben indicar los atributos. En este ejercicio aparecen los conceptos de Gasto y de Casa. Cada uno será una clase.

Así, una posible solución es:
Clase Gasto: atributos: int monto, int día, String descripción.
Los métodos básicos que debe incluir son:
- acceso y modificación de atributos;
- toString

Clase Casa: atributos: lista de gastos, String dirección.
Los métodos que debería incluir son:
- agregar un Gasto;
- dar la lista de todos los gastos;
- indicar gasto mayor; y
- acceso y modificación de atributos.

La lista de gastos se puede llevar en un ArrayList. Así permite agregar un elemento, obtener el de una posición o todos.

Una versión preliminar de la jerarquía podría ser como se presenta en la Ilustración 19 Gastos de la Casa:

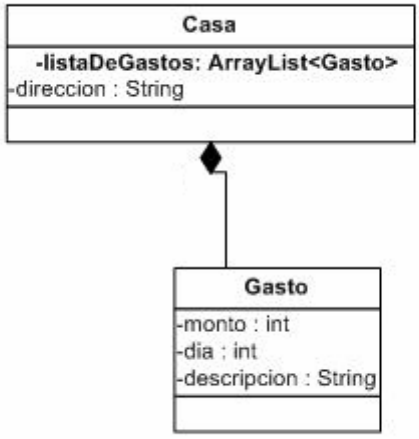


Ilustración 19 Gastos de la Casa

12.1.2 Clase Casa

El código sería:

```
package gastosCasa;
import java.util.*;
public class Casa {
	// variables de instancia
	private ArrayList<Gasto> conjuntoGastos;
	private String direccion;

	// Constructor
	public Casa() {
		conjuntoGastos = new ArrayList<Gasto>();
		this.setDireccion("Sin direccion");
	}
	// acceso y modificacion
	public void setDireccion(String unaDireccion) {
		direccion = unaDireccion;
	}
	public String getDireccion() {
		return direccion;
	}
	public void agregarUnGasto(Gasto unGasto) {
		this.devolverTodosGastos().add(unGasto);
	}
	public ArrayList<Gasto> devolverTodosGastos() {
		return conjuntoGastos;
	}

	// devuelve cantidad de elementos
	public int cantidadGastos() {
		return this.devolverTodosGastos().size();
	}
```

12.1.3 Clase Gasto

```
package gastosCasa;
public class Gasto {


	// variables de instancia
	private int monto;
	private int dia;
	private String descripcion;

	// Constructores

	public Gasto() {
		this.setDia(1);
		this.setDescripcion("Sin datos");
		this.setMonto(0);
	}
	public Gasto(int unDia, String unaD, int unM) {
		this.setDia(unDia);
		this.setDescripcion(unaD);
		this.setMonto(unM);
	}
	// acceso y modificacion
	public void setDescripcion(String unaDescripcion) {
		descripcion = unaDescripcion;
	}
	public String getDescripcion() {
		return descripcion;
	}
	public int getDia() {
		return dia;
	}
	public void setDia(int unDia) {
		dia = unDia;
	}
	public int getMonto() {
		return monto;
	}
	public void setMonto(int unMonto) {
		monto = unMonto;
	}
	// Impresion
	@Override
	public String toString() {
		return "Gasto " + this.getDescripcion() + " $ " + this.getMonto() + " Dia: " +
this.getDia();
	}
```

12.1.4 Máximo gasto

Se desea saber cuál fue el máximo gasto. Es la casa la que debe saber cuál es el mayor gasto. Hay distintas opciones para este método: recorrer "a mano" la lista, recorrerla usando un iterador o investigar si existe algo ya hecho en Java. Veremos las dos primeras opciones.

Recorriendo “a mano”:

```
// MAXIMO
// VERSION 1: Máximo a mano - recorro con for
public Gasto devolverGastoMayor() {
	Gasto max = new Gasto();
	Gasto aux;
	for (int i = 0; i < this.cantidadGastos(); i = i + 1) {
		aux = this.devolverTodosGastos().get(i);
		if (aux.getMonto() > max.getMonto()) {
			max = aux;
		}
	}
	return max;
}
```

Otra versión utilizando Iterator:

```
// VERSION 2: Máximo - recorro con Iteracion
public Gasto devolverGastoMayor() {
	Gasto max = new Gasto();
	Gasto aux;
	Iterator<Gasto> lista =  this.devolverTodosGastos().iterator();
	while (lista.hasNext()) {
		aux = lista.next();
		if (aux.getMonto() > max.getMonto()) {
			max = aux;
		}
	}
	return max;
}
```

12.1.5 Clase de Prueba

¿Cómo pruebo estas clases? Es necesario armar una clase de prueba. En esa clase ubicaremos el menú. También allí crearemos la instancia de Casa sobre la que se probarán los mensajes. En esa clase se ubica el método *main*. Será necesario ofrecer la lista de opciones, para ello se puede:
1) mostrar con “System.out.println()” cada una

```
	// MENU - A mano
	// Una opción para mostrar el menú es desplegar cada línea armada a mano
	  System.out.println("opciones: "+"\n"+
			"1 - ingreso" +"\n"+
			"2 - mostrar todos" +"\n"+
			"3 - mayor" +"\n"+
```

```
                "4 - ordenar"+"\n"+
                "5 - buscar datos de un monto"  +"\n"+
                "6 - indicar si esta monto"+"\n"+
                "7 - fin");
```

ó
2) cargar un array con las opciones y desplegarlas.

Para definirlo e inicializarlo se utiliza:

```
// MENU
String lista[] = {"ingreso", "mostrar todos", "mayor", "ordenar", "buscar datos de un monto", "indicar si esta monto", "fin"};
```

Se muestra:

```
        for (int i = 0; i <= 6; i++) {
                System.out.println((i + 1) + "===" + lista[i]);
        }
```

En cualquiera de los dos casos, se solicita luego el ingreso de la opción y en función de ese valor se harán las opciones (dentro de una estructura repetitiva).

12.1.6 Excepciones

¿Qué pasa si en el menú cuando pide ingresar un dato ingresan una letra? Da una *exception*. ¿Cómo lo soluciono?

Una **excepción** es un objeto que define una situación no usual o errónea. Una excepción es "levantada", "lanzada" o "tirada" por el programa y puede ser capturada y manejada. Describe las características de esa situación.

Un **error** es similar a una excepción, salvo que generalmente no se puede recuperar la situación.

Ambos representan procesos o situaciones no válidas. La excepción la puedo capturar y manejar, el error no.

¿Cómo hacer para manejar las excepciones? La sentencia *try* identifica un conjunto de sentencias que pueden "tirar" una excepción. La sentencia *catch*, que sigue al bloque *try* define cómo se tratará cada tipo especial de excepción que indique.

```
try  {
        ...
}
catch (exception variable) {
        ...
}
catch (exception variable) {
        ...
}
```

Cada *catch* 'captura' un tipo especial de excepción. En el ejemplo quedaría, luego de estar definidos *opcion* e *input* (que corresponden respectivamente a la variable de tipo int en la que se leerá la opción y a la instancia de Scanner):

```
        // ingreso y valido la opción del usuario
        try {
                opcion = input.nextInt();
        }
        catch (InputMismatchException e) {
                System.out.println("Error, se ignora la entrada");
                opcion = 0;
                input.nextLine();
        }
```

En este caso se optó por tomar como que se hubiera ingresado un "0" (si dio error). Es fundamental colocar la sentencia *input.nextLine()* para poder luego leer el siguiente dato.

12.1.7 Ingreso de datos

Otro punto interesante para analizar es: ¿dónde va el código de solicitar datos por pantalla? Una opción es poner un método de clase que los pida. Este método podría ir en la clase de prueba, pero pensando en un posible re-uso, podría ir en la clase correspondiente. Formalmente convendría que fuera en otro paquete independiente, pero para esta altura del curso de Programación I se prefiere incluirlo en la propia clase para no agregar más clases.

En este ejemplo particular, para crear un gasto, se incluye en la clase Gasto:

```
public static Gasto crearGasto() {
        Gasto unG;
        boolean ok;
        Scanner input = new Scanner(System.in);
        unG = new Gasto();
        ok = false;
        // pide datos, validando formato ok
        while (!ok) {
                try {
                        System.out.println("Ingrese día ");
                        unG.setDia(input.nextInt());
                        System.out.println("Ingrese descripción ");
                        unG.setDescripcion(input.nextLine());
                        System.out.println("Ingrese monto ");
                        unG.setMonto(input.nextInt());
                        ok = true;
                } catch (InputMismatchException e) {
                        System.out.println("Datos erróneos, reingrese ");
                        input.nextLine();
                }
        }
        return unG;
}
```

12.1.8 Búsqueda

Se desea saber si está ingresado un gasto de un monto dado. Se puede hacer una búsqueda secuencial por monto, que devuelve *true* si encuentra un gasto del monto dado.

Así el código podría ser:

```
// VERSION 1:  recorre con Iteración
// Busca un objeto Gasto del monto indicado
public boolean estaElGasto(int unMonto) {
	Gasto unG;
	unG = new Gasto();
	unG.setMonto(unMonto);
	Iterator<Gasto> e = this.devolverTodosGastos().iterator();
	boolean esta = false;
	while (e.hasNext() && !esta) {
		Gasto g = e.next();
		if (g.equals(unG)) {
			esta = true;
		}
	}
	return esta;
}
```

Se creó un objeto de la clase Gasto, el cual se inicializó con el monto buscado. Se utiliza el método *equals* en Gasto, por lo cual hay que redefinirlo. El método *equals* está definido en *Object* y es el método que se utiliza para determinar si se es igual a otro objeto.

```
@Override
public boolean equals(Object parm1) {
	// Es necesario hacer cast porque el parámetro es Object
	return this.getMonto() == ((Gasto) parm1).getMonto();
}
```

12.1.8.1 Otra versión de búsqueda (contains)

Se utiliza el método *contains* que está definido en *ArrayList*. Este método chequea si hay algún objeto igual (*equals*) al que estoy buscando.

```
// VERSION 2:  usa Contains
// Busca un objeto Gasto del monto indicado
public boolean estaElGasto(int unMonto) {
	Gasto unG;
	// creo un objeto Gasto, similar al que busco
	unG = new Gasto();
	unG.setMonto(unMonto);
	// para comparar, internamente usa el método equals de Gasto
	return (this.devolverTodosGastos().contains(unG));
}
```

12.1.8.2 Otra versión de búsqueda (indexOf)

En vez de devolver *true* o *false*, que devuelva el objeto que estoy buscando.

```
// VERSION 3: usa indexOf
// Otra posible implementacion de buscar un Gasto
// a partir del monto, utilizando metodos de la coleccion
public Gasto buscarGastoDeMonto(int unMonto) {
        Gasto unG;
        // creo un objeto Gasto, similar al que busco
        // (similar implica que el método equals dara true)
        unG = new Gasto();
        unG.setMonto(unMonto);
        int aux;
        aux = (this.devolverTodosGastos().indexOf(unG));
        Gasto retorno = null;
        if (aux != -1) {
                retorno= (this.getListaGastos().get(aux));
        }
        return retorno;
}
```

El método *indexOf* devuelve la posición en la cual hay un objeto similar o el valor -1 si no lo hay. Aquí luego se accede a esa posición y se obtiene el objeto real.

12.1.9 Ordenación (sort)

Quiero agregar un método que muestre los gastos ordenados por monto. Veremos diferentes alternativas.

12.1.9.1 Implementación a mano

Una alternativa sería hacer mi propio método de ordenación, por ejemplo utilizando algoritmo de selección similar al utilizado al presentar los *arrays*. Este algoritmo recorre la lista buscando el mayor, cuando lo ubica lo coloca al final y repite esta operación, pero sin tener en cuenta el último. Repite la búsqueda y reubicación considerando la lista sin esos últimos elementos.

Existen otros posibles métodos para ordenar que se verán en cursos posteriores (por ej. *sort* -ordenar- por enumeración, intercambio, inserción, *quicksort*, *mergesort*). El estudio de la eficiencia de estos algoritmos es un tema de cursos más avanzados.

12.1.9.2 Reutilizar

Veamos primero cómo debería ser este método. Pensemos si quisiera ordenar varios libros. ¿Qué debo indicarle a alguien para que pueda ordenarlos? Si conoce algún algoritmo de ordenación como los nombrados arriba, en realidad lo único que necesita saber es: si dados dos libros, el primero es menor que el segundo o no. Debe tener un método que indique si un libro es menor que otro en algún sentido (por ej. cantidad páginas, edición, etc.).

Existe en la clase *Collections* un método llamado *sort*, que según indica utiliza el orden natural de los elementos. ¿Qué es el orden natural? Es el orden definido en la propia clase. El método por defecto se llama *compareTo*. ¿Cómo sé que se llama así? Aparece aquí el concepto de *interface*.

12.1.9.3 Nociones de *Interface*

Una *interface* es una colección de métodos abstractos y constantes. No son clases, pero se pueden usar como tal. Es un conjunto de prototipos de métodos, sin información de implementación asociada.

Se define:

```
interface nombreXX{
        constantes;
        metodos abstractos ;
}
```

Una clase implementa una *interface* dando la implementacion de cada uno de los métodos definidos en esa *interface*. Para indicarlo, se pone:

```
class YY implements nombreXX{
...}
```

Las *interface*s sirven para que objetos de diferentes clases, no relacionadas, se traten de la misma forma. Permite que dos clases completamente independientes tengan un mismo conjunto de métodos definidos en forma diferente.

No es herencia múltiple: en herencia múltiple recibo y re-uso código, acá no se recibe ni re-usa código.

Una clase puede implementar varias *interfaces*.

¿Para qué métodos abstractos e *interfaces*? En particular en este ejemplo, debo indicarle el criterio de ordenación. Por ejemplo el método que ordena (*Collections.sort*) tiene que saber si un objeto es menor que otro, que relación tienen entre ellos. Asume que se usa el orden natural. ¿Cómo se llama el método? ¿Qué formato tiene? Pensemos en el método *sort*. ¿Cómo hace para indicar que internamente cuando compara va a usar el método "X"?. Para ello es que están definidas las *interfaces*. En este caso se llama interface Comparable. Allí declaro cuál es el formato que espero que tenga un método (y así lo usa internamente en este caso el método *sort*). Aquel que quiera usar el método *sort* tendrá que dar el código de ese método. Para decir que está dando la implementación a ese método, se debe indicar que implementa esa interface y dar el código del método *compareTo*.

12.1.9.4 Uso del Collections.sort

En la clase Casa:

```
public ArrayList<Gasto> ordenar() {
        // Ordeno por orden natural
        // Internamente usa compareTo() definido en Comparable
        //
        Collections.sort(devolverTodosGastos());
        return devolverTodosGastos();
}
```

12.1.9.4.1 Interfaz Comparable

En la clase Gasto, se indica que implementa la *interface Comparable* y se da código al método *compareTo*.

```
// método compareTo
public int compareTo(java.lang.Object o) {
        return this.getMonto() - ((Gasto) o).getMonto();
}
```

El método compareTo retorna:

<0: si el objeto que recibió el mensaje precede al parámetro;
>0: si el parámetro precede al receptor del mensaje; o
=0: en otro caso.

Si se definió utilizando *generics*, o sea que la clase implementa *Comparable<Gasto>*, el método *compareTo* queda:

```
// método compareTo
public int compareTo(Gasto o) {
        return this.getMonto() - o.getMonto();
}
```

12.1.9.4.2 Otro orden

Supongamos que además del orden anterior se solicita otro orden diferente. Para ello dispongo de la interfaz *Comparator* que tiene el método *compare*.

En la clase Casa:

```
public ArrayList<Gasto> ordenarDecreciente() {
        // Usa orden arbitrario, definido en compare() en Comparator
        // El criterio por el cual comparar está implementado
        // (implements) en la
        // clase Criterio Decreciente. El método se llama compare.
        //
        // El sort lo usa cuando dados dos objetos, deba decidir cual va primero
        Collections.sort(this.devolverTodosGastos(), new CriterioDecreciente());
        return this.devolverTodosGastos();
}
```

Una forma es definir una clase auxiliar en la cual se indica el código del método compare.

```
package gastosCasa;

import java.util.Comparator;

// Clase definida para contener el criterio de comparación

public class CriterioDecreciente implements Comparator<Gasto>{

  // Este método es necesario definirlo para establecer el criterio por el
  // cual un objeto va antes que otro en la ordenación
```

```
  public int compare(Gasto parm1, Gasto parm2) {
    // uso criterio decreciente
    return parm2.getMonto() - parm1.getMonto();
  }

}
```

El método *compare* devuelve:
<0 si el primer parámetro precede al segundo
>0 si el segundo parámetro precede al primero
= en otro caso.

Sugerencia: investigar los demás métodos de *Collections*.

12.1.9.5 Código completo

Gasto

```
package gastosCasa;

import java.util.InputMismatchException;
import java.util.Scanner;

//al implementar Comparable, se incluye el método compareTo()
public class Gasto implements Comparable<Gasto> {
	// variables de instancia
	int monto, dia;
	String descripcion;

	// Constructores

	public Gasto() {
		this.setDia(1);
		this.setDescripcion("Sin datos");
		this.setMonto(0);
	}
	public Gasto(int unDia, String unaD, int unM) {
		this.setDia(unDia);
		this.setDescripcion(unaD);
		this.setMonto(unM);
	}
	// -----------------------------------------------------
	// acceso y modificación
	public void setDescripcion(String unaDescripcion) {
		descripcion = unaDescripcion;
	}

	public String getDescripcion() {
		return descripcion;
	}
	public int getDia() {
		return dia;
	}
```

```
	public void setDia(int unDia) {
		dia = unDia;
	}
	public int getMonto() {
		return monto;
	}
	public void setMonto(int unMonto) {
		monto = unMonto;
	}
	// -----------------------------------------------------
	// Impresión
	@Override
	public String toString() {
		return "Gasto " + this.getDescripcion() + " $ " + this.getMonto()
		+ " Dia: " + this.getDia();
	}

	// método compareTo
	public int compareTo(Gasto o) {
		return this.getMonto() - o.getMonto();
	}

	// método auxiliar para pedir datos de pantalla para crear un Gasto
	public static Gasto crearGasto() {
		Gasto unG;
		boolean ok;
		Scanner input = new Scanner(System.in);
		unG = new Gasto();
		ok = false;
		// pide datos, validando formato ok
		while (!ok) {
			try {
				System.out.println("Ingrese dia ");
				unG.setDia(input.nextInt());
				System.out.println("Ingrese descripcion ");
				unG.setDescripcion(input.nextLine());
				System.out.println("Ingrese monto ");
				unG.setMonto(input.nextInt());
				ok = true;
			} catch (InputMismatchException e) {
				System.out.println("Datos erroneos, reingrese ");
				input.nextLine();
			}
		}
		return unG;
	}

	// Método equals
	// Es necesario redefinir el método equals para que cuando
	// le pida buscar alguno, compare por este criterio para ver si es =
	@Override
	public boolean equals(Object parm1) {
		// Es necesario hacer cast porque el parámetro es Object
		return this.getMonto() == ((Gasto) parm1).getMonto();
	}
```

```
}
```

Casa:

```
package gastosCasa;

import java.util.*;
public class Casa {
	// var de instancia
	ArrayList<Gasto> conjuntoGastos;
	String direccion;

	// Constructor
	public Casa() {
		conjuntoGastos = new ArrayList<Gasto>();
		this.setDireccion("Sin direccion");
	}
	// acceso y modificación
	public void setDireccion(String unaDireccion) {
		direccion = unaDireccion;
	}
	public String getDireccion() {
		return direccion;
	}
	public void agregarUnGasto(Gasto unGasto) {
		this.devolverTodosGastos().add(unGasto);
	}
	public ArrayList<Gasto> devolverTodosGastos() {
		return conjuntoGastos;
	}

	// devuelve cantidad de elementos
	public int cantidadGastos() {
		return this.devolverTodosGastos().size();
	}

	// MAXIMO

	// VERSION 1: Máximo a mano - recorro con for

	public Gasto devolverGastoMayorManualTotalmente() {
		Gasto max = new Gasto();
		Gasto aux;
		for (int i = 0; i < this.cantidadGastos(); i = i + 1) {
			aux = this.devolverTodosGastos().get(i);
			if (aux.getMonto() > max.getMonto()) {
				max = aux;
			}
		}
		return max;
	}

	// VERSION 2: Máximo - recorro con Iteración
	public Gasto devolverGastoMayorManual() {
		Gasto max = new Gasto();
		Gasto aux;
```

```
		Iterator<Gasto> lista;
		lista = this.devolverTodosGastos().iterator();
		while (lista.hasNext()) {
			aux = lista.next();
			if (aux.getMonto() > max.getMonto()) {
				max = aux;
			}
		}
		return max;
	}

	// BUSQUEDA DE UN GASTO DE MONTO DADO
	// VERSION 1: Método estaElGastoManualmente - recorre con Iteración
	// Busca un objeto Gasto del monto indicado
	public boolean estaElGastoManualmente(int unMonto) {
		Gasto unG;
		unG = new Gasto();
		unG.setMonto(unMonto);
		Iterator<Gasto> e = this.devolverTodosGastos().iterator();
		boolean esta = false;
		while (e.hasNext() && !esta) {
			Gasto g = e.next();
			if (g.equals(unG)) {
				esta = true;
			}
		}
		return esta;
	}
	// VERSION 2: Método estaElGasto - usa Contains
	// Busca un objeto Gasto del monto indicado
	public boolean estaElGasto(int unMonto) {
		Gasto unG;
		// creo un objeto Gasto, similar al que busco
		unG = new Gasto();
		unG.setMonto(unMonto);
		// para comparar, internamente usa el método equals de Gasto
		return (this.devolverTodosGastos().contains(unG));
	}

	// VERSION 3: Método buscarGastoDeMonto, usa indexOf
	// Otra posible implementación de buscar un Gasto
	// a partir del monto, utilizando métodos de la colección
	public Gasto buscarGastoDeMonto(int unMonto) {
		Gasto unG;
		// creo un objeto Gasto, similar al que busco
		// (similar implica que el método equals dara true)
		unG = new Gasto();
		unG.setMonto(unMonto);
		int aux;
		aux = (this.devolverTodosGastos().indexOf(unG));
		Gasto retorno = null;
		if (aux != -1) {
			retorno= (this.getListaGastos().get(aux));
		}
		return retorno;
```

```
	}

	// ORDENACION
	// VERSION 1: Método ordenar - orden natural
	// Para ordenar la lista de Gastos


	public ArrayList<Gasto> ordenar() {
		// Ordeno por orden natural
		// Internamente usa compareTo() definido en Comparable
		//
		Collections.sort(devolverTodosGastos());
		return devolverTodosGastos();
	}

	// VERSION 2: Método ordenarDecreciente - orden arbitrario
	// Para ordenar la lista de Gastos

	public ArrayList<Gasto> ordenarDecreciente() {
		// Usa orden arbitrario, definido en compare() en Comparator

		// El criterio por el cual comparar esta implementado
		// (implements) en la
		// clase Criterio Decreciente. El método se llama compare.
		//
		// El sort lo usa cuando dados dos objetos, deba decidir cual va primero

		Collections.sort(this.devolverTodosGastos(), new CriterioDecreciente());
		return this.devolverTodosGastos();
	}
}
```

Prueba

```
package gastosCasa;

import java.util.*;

public class ClasePruebaCasa {

	public static void main(String args[]) {
		// Creo objeto Casa, contendrá la lista de gastos
		Casa casa = new Casa();
		int opcion;
		Scanner input = new Scanner(System.in);
		opcion = 1;

		// MENU
		// El menú se puede armar con un array
		String lista[] = {"ingreso", "mostrar todos", "mayor", "ordenar", "buscar datos
de un monto", "indicar si esta monto", "fin"};
		while (opcion != 7) {
			// MENU - A mano
			// Una opciónpara mostrar el menú es desplegar cada línea armada a
			// mano
```

```
                        /*
                         * System.out.println("opciones: "+"\n"+ "1 - ingreso" +"\n"+ "2 -
                         * mostrar todos" +"\n"+ "3 - mayor" +"\n"+ "4 - ordenar"+"\n"+ "5 -
                         * buscar datos de un monto" +"\n"+ "6 - indicar si esta
                         * monto"+"\n"+ "7 - fin");
                         */
                        // Otra opción es mostrarla usando el array creado
                        for (int i = 0; i <= 6; i++) {
                                System.out.println((i + 1) + "===" + lista[i]);
                        }
                        // ingreso y valido la opción del usuario
                        try {
                                opcion = input.nextInt();

                        } catch (InputMismatchException e) {
                                System.out.println("Error, se ignora la entrada");
                                opcion = 0;
                                input.nextLine();
                        }
                        // Proceso entrada del menú
                        switch (opcion) {
                                case 1 :
                                        // ingreso
                                        System.out.println("Ingreso de gasto");
                                        casa.agregarUnGasto(Gasto.crearGasto());
                                        System.out.println("Ingreso ok");

                                        break;
                                case 2 :
                                        // listado general
                                        System.out.println("Listado general de gastos");
                                        System.out.println("Todos los gastos son" + "\n" +
casa.devolverTodosGastos());
                                        break;
                                case 3 :
                                        // máximo gasto
                                        System.out.println("Máximo gasto"                    +
casa.devolverGastoMayorManualTotalmente().toString());
                                        break;
                                case 4 :
                                        // ordenar
                                        // Criterio creciente
                                        System.out.println("Orden natural " + casa.ordenar());
                                        // Criterio decreciente
                                        System.out.println("Orden decreciente"               +
casa.ordenarDecreciente());
                                        break;
                                case 5 :
                                        // búsqueda de un gasto de monto dado
                                        int aux = 0;
                                        try{
                                                System.out.println("Ingrese monto");
                                                aux = input.nextInt();
                                                if (casa.estaElGasto(aux)) {
```

```
            System.out.println(casa.buscarGastoDeMonto(aux));
                                                } else {
                                                        System.out.println("No está");
                                                }
                                        }catch(InputMismatchException e){
                                                System.out.println("Número incorrecto.");
                                                input.nextLine();
                                        }
                                        break;
                                case 6 :
                                        // consulta de si esta un gasto de un monto dado
                                        System.out.println("Ingrese monto");
                                        int cualMonto = 0;
                                        try{
                                                cualMonto = input.nextInt();
                                                System.out.println("Este monto manualmente
está: " + casa.estaElGastoManualmente(cualMonto));
                                                // se podría utilizar también:
                                                System.out.println("Este monto está: " +
casa.estaElGasto(cualMonto));
                                        }catch(InputMismatchException e){
                                                System.out.println("Número incorrecto.");
                                                input.nextLine();
                                        }
                                        break;
                                }
                        }
                }
}
```

Criterio Decreciente

```
package gastosCasa;
import java.util.Comparator;

// Clase definida para contener el criterio de comparación

public class CriterioDecreciente implements Comparator<Gasto>{

  // Este método es necesario definirlo para establecer el criterio por el
  // cual un objeto va antes que otro en la ordenación
  public int compare(Gasto parm1, Gasto parm2) {
    // uso criterio decreciente
    return parm2.getMonto() - parm1.getMonto();
  }
}
```

Semana 13

13.1 Orden por dos campos

Supongamos que se desean ordenar los gastos por día y para el mismo día, por nombre. Se compara primero por día. Si ya es posible establecer el orden relativo entre esos dos elementos ya está resuelto. Si tienen el mismo día, es necesario comparar los nombres para determinar cuál va primero.

La forma de realizarlo es estableciendo el criterio de comparación así:

```
public class CriterioDoble implements Comparator<Gasto>{

  public int compare(Gasto parm1, Gasto parm2) {
    int diferencia = parm1.getDia() - parm2.getDia();
    if (diferencia==0) {
          diferencia = parm1.getNombre().compareTo(parm2.getNombre());
    }
    return diferencia;

}
```

13.2 Agenda Telefónica: HashMap

Consideremos el siguiente ejemplo: se quiere llevar una agenda telefónica, esto es una lista de personas con su número telefónico. Si se asume disponible una clase Agenda Telefónica, con los métodos:

- agregarPersona(String) que permite agregar una persona a la agenda
- darTelefonoDe(String): dado el nombre permite ubicar su teléfono
- estaElNombre(String): para consultar si está cierto nombre en la agenda
- darNombres(), darTelefonos(): iteradores para obtener una lista de nombres o los teléfonos respectivamente;
- sacarTeléfonoDe(String): para borrar una entrada en la agenda.

se podría utilizar de la siguiente manera:

Agenda Telefonica

```
package PaqueteAgendaTelefonica;
        import java.util.*;
        public class PruebaAgenda {
        public static void main (String args[]) {
            // creo la agenda
            AgendaTelefonica agenda;
            agenda = new AgendaTelefonica() ;
            // agrego varias personas
            agenda.agregarPersona("Suzi",3761241);
            agenda.agregarPersona("Patricia", 7117765);
            agenda.agregarPersona("Irene",1718);
            // consultas varias
```

```
            System.out.println("El teléfono de Patricia es "+agenda.darTelefonoDe("Patricia"));
            if (agenda.estaElNombre("Irene")) {
              System.out.println("El teléfono de Irene está en la Agenda");
            }
            if (agenda.estaElTelefono(3761241)) {
              System.out.println("Sí, está el  teléfono 3761241");
            }
            System.out.println("Lista de todos los telefonos "+agenda.darTelefonos());
            Iterator<Integer>  tels = agenda.darTelefonos();
            Iterator<String> nombres = agenda.darNombres();
            while (tels.hasNext() ) {
              System.out.println("Nombre "+nombres.next()+ " telefono "+tels.next());
            }
            System.out.println("Saco telefono de Patricia");
            agenda.sacarTelefonoDe("Patricia");
            agenda.cambiarTelefonoDe("Suzi",996892);
            tels = agenda.darTelefonos();
            nombres = agenda.darNombres();
            while (tels.hasNext() ) {
              System.out.println("Nombre "+nombres.next()+ " telefono "+tels.next());
            }

          }
}
```

En esta versión, para simplificar, se considera que la persona es simplemente representada por un String.

Consideremos ahora cómo sería la implementación de la clase Agenda Telefónica. Analizaremos de qué formas se podría almacenar los datos de nombre y teléfono. Una alternativa es llevar 2 *arrays* "paralelos", uno con las personas y otro con los números de teléfono. El problema de esta opción es que es necesario mantener manualmente la correspondencia siempre. Por ejemplo, si se elimina una persona, hay que eliminar el correspondiente teléfono en la otra lista.

Conviene investigar en Java a ver si existe algún tipo especial de colección aplicable para este caso. Hay estructuras apropiadas para este tipo de relación. Lo que se necesita es algo que tenga una lista de claves y de valores asociados. Hay pares clave/valor, con las claves como filas de una tabla y valores como columnas de la tabla. Esta estructura es el *HashMap* (que está en *java.util*).

Puedo averiguar cuántas "filas" hay ocupadas (*size*), saber si esta vacío (*isEmpty*), obtener las claves (*keySet*, devuelve una *Collection*, se puede pedir un *Iterator*), obtener los valores (*values*), obtener el objeto de una clave (*get*) y poner un objeto en una clave (*put*). También puedo sacar una clave (*remove*). También puedo preguntar si está determinada clave (*containsKey*) y también si está determinado valor (*containsValue*).

```
package PaqueteAgendaTelefonica;
          import java.util.*;
          public class AgendaTelefonica {
            private HashMap <String, Integer> agenda;
            public AgendaTelefonica() {
              agenda = new HashMap<String, Integer>();
            }
```

```
        public void agregarPersona( String nombre, int telefono) {
                agenda.put(nombre,telefono);
        }
        public int darTelefonoDe(String unNombre) {
          // el método get, si no esta el dato buscado, devuelve null
          return agenda.get(unNombre);
        }
        public Iterator<String> darNombres() {
          return agenda.keySet().iterator();
        }
        public Iterator<Integer> darTelefonos() {
          return agenda.values().iterator();
        }
        public boolean estaElNombre(String unNombre) {
          return agenda.containsKey(unNombre);
        }
        public boolean estaElTelefono(int unNúmero) {
          return agenda.containsValue(unNúmero);
        }
        public void sacarTelefonoDe(String unNombre) {
          agenda.remove(unNombre);
        }
        public void cambiarTelefonoDe(String unNombre, int nuevo) {
          agenda.put(unNombre,nuevo);
        }
      }
}
```

13.3 Wrapper Classes y Boxing

En un *hashmap*, el par clave y valor deben ser objetos. Los tipos primitivos (*int*, *boolean*, etc) son solamente valores. No tengo métodos en ellos, no son objetos. En las versiones anteriores a Java 5 se aplican las *wrapper classes* (*wrapper*: envolver, funda, cubierta) que son clases que corresponden a los tipos primitivos. Encapsulan un tipo primitivo dentro.

Así, en versiones anteriores de Java, para agregar un *int* a un *ArraList*, debía "envolverlo" como un objeto. Siendo lista un *ArrayList* ya definido, para agregar un entero se escribe:

```
lista.add(new Integer(2));
```

Java, por eficiencia, representa los tipos simples de forma diferente. No es orientada a objetos y los pone fuera de la jerarquía. Muchas veces se necesita sacrificar eficiencia por flexibilidad, y por esta razón se crearon clases estándar que permiten integrar esos tipos simples en la jerarquía. Esas son las *wrapper classes.* Están agrupadas bajo *Number* y tengo métodos: *intValue*, *doubleValue*, etc. Están en *java.lang*. A partir de la versión Java 1.5, se hace el *wrapper* en forma automática (*autoboxing*). Si quiero poner un entero, acepta que lo escriba directamente. Ejemplo: considerando la lista anterior, se puede escribir:

```
lista.add(2);
```

13.3.1 Null

Java inicializa las variables de tipo objeto con *null*, que significa que no fue inicializada o cargada. No tiene ningún valor. Es una palabra clave que se puede usar en lugar de un objeto. Por ejemplo:

```
String s
s = “hola”
s  = null;
```

La última línea saca la referencia al literal anterior, deja s sin valor alguno.

Cuando aparece el mensaje *NullPointerException* es que se trató de usar una variable de objeto que no refiere a ningún objeto.

Semana 14

El objetivo de esta semana es realizar prácticas que integren todos los contenidos del curso.

14.1 Ejercicio: Lleva y Trae

LLEVA Y TRAE SRL es una empresa dedicada al envío de paquetes. Realizar el diagrama de clases completo y la implementación en Java que permitan:
a) registrar cliente: se indican los datos personales del cliente.
b) ingresar paquetes. Se indica: número de paquete, cliente y destinatario (análogo a cliente, se asume que los destinatarios son clientes de la empresa).
c) ingreso de aviso de recepción; se indica el número de paquete que fue entregado.
d) consulta de un paquete: se ingresa un número de paquete y debe informarse si ya se ingresó su aviso de recepción o no.
e) listado de paquetes enviados que no tienen aviso de recepción.
f) listado de clientes que, a su vez, son destinatarios de algún paquete.

14.2 Solución Ejercicio Lleva y Trae

Como primer paso, identificaremos las clases necesarias. Aparecen los conceptos de: Paquete y Cliente. El destinatario también es un cliente. Además, aparece la clase Empresa quien será la responsable de la información de los paquetes y clientes.

Cada paquete tiene varios atributos: número, cliente, destinatario y la indicación de si fue recibido o no. Podría incluirse otra clase más para registrar el ingreso del aviso de recepción. Esta clase se utilizaría si se deseara guardar información sobre la fecha de la entrega, quién lo recibió o cualquier otro dato de la entrega en sí. En esta versión, dado que solamente parece ser necesario indicar si llegó o no, se guardará esta información como un atributo del paquete.

Una posible jerarquía de clases se presenta en la Ilustración 20 Lleva y Trae:

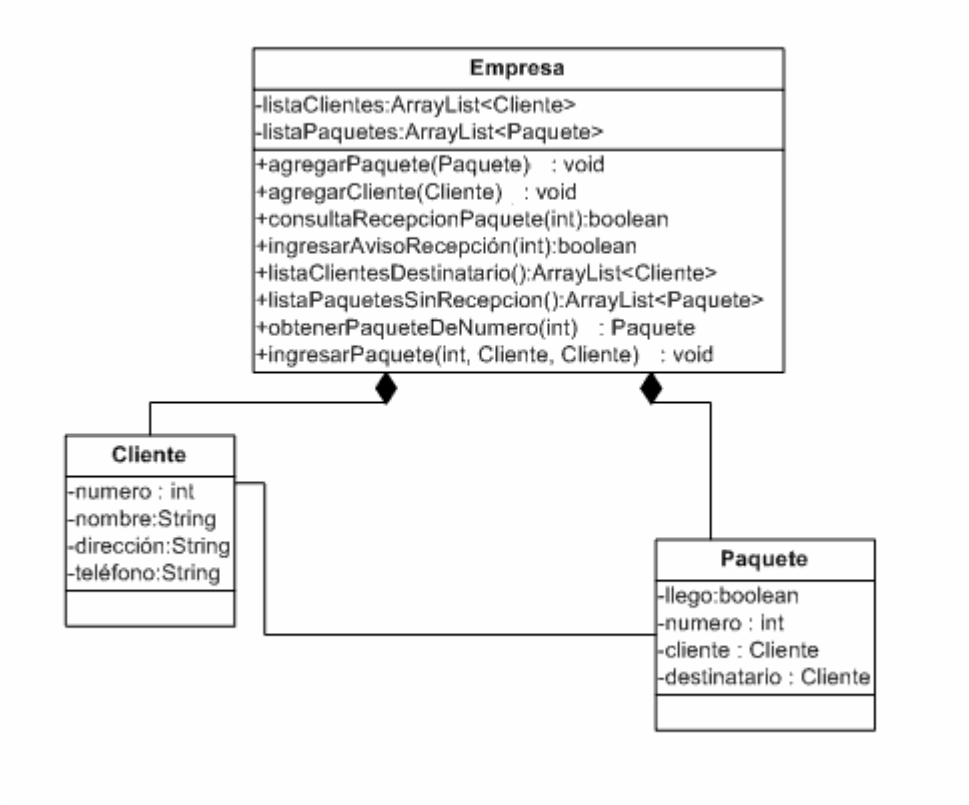


Ilustración 20 Lleva y Trae

Una posible implementación es:

```
//Clase Paquete

package llevaTrae;

public class Paquete {

	// Variables de instancia
	private boolean llego;
	private int numero;
	private Cliente cliente;
	private Cliente destinatario;

	// Constructores
	public Paquete() {
		this.setNumero(0);
		this.setCliente(null);
		this.setDestinatario(null);
		this.setLlego (false);
	}

	public Paquete(int unNumero, Cliente unC, Cliente unD) {
		this.setNumero(unNumero);
		this.setCliente(unC);
		this.setDestinatario(unD);
		this.setLlego (false);
```

```
        }

        // acceso y modificacion
        public Cliente getCliente() {
                return cliente;
        }

        public Cliente getDestinatario() {
                return destinatario;
        }

        public boolean getLlego() {
                return llego;
        }

        public int getNumero() {
                return numero;
        }

        public void seRecibio() {
                llego = true;
        }

        public void setCliente(Cliente unCliente) {
                cliente = unCliente;
        }

        public void setDestinatario(Cliente unDestinatario) {
                destinatario = unDestinatario;
        }

        public void setNumero(int unNumero) {
                numero = unNumero;
        }

        // Redefinición de métods
        @Override
        public boolean equals(Object parm1) {
                return this.getNumero() == ((Paquete) parm1).getNumero();
        }

        @Override
        public String toString() {
                String retorno = "Número de paquete: " + this.getNumero()
                                + "\nCliente " + this.getCliente() + "\nDestinatario "
                                + this.getDestinatario();
                if (this.getLlego()) {
                        retorno = retorno + "\nFue entregado";
                } else {
                        retorno = retorno + "\nNo fue entregado";
                }
                return retorno;
        }
}
```

```
//Clase Cliente

package llevaTrae;

public class Cliente {

	// Variables de instancia
	private int numero;
	private String nombre;
	private String direccion;
	private String telefono;

	// Constructor
	public Cliente() {
		this.setDireccion("Sin Dirección");
		this.setNombre("Sin nombre");
		this.setNumero(0);
		this.setTelefono("No tiene telefono");
	}

	// acceso y modificación
	public String getDireccion() {
		return direccion;
	}

	public String getNombre() {
		return nombre;
	}

	public int getNumero() {
		return numero;
	}

	public String getTelefono() {
		return telefono;
	}

	public void setDireccion(String unaDireccion) {
		direccion = unaDireccion;
	}

	public void setNombre(String unNombre) {
		nombre = unNombre;
	}

	public void setNumero(int unNumero) {
		numero = unNumero;
	}

	public void setTelefono(String unTelefono) {
		telefono = unTelefono;
	}

	// Redefinición de métodos
	@Override
```

```
	public boolean equals(Object parm1) {
		return this.getNumero() == ((Cliente) parm1).getNumero();
	}
	@Override
	public String toString() {
		return "Nombre: " + this.getNombre() + "\nDirección: "
				+ this.getDireccion() + "\nTeléfono: " + this.getTelefono()
				+ "\nNúmero de cliente: " + this.getNumero();
	}
}

//Clase Empresa

package llevaTrae;

import java.util.*;

public class Empresa {

	// Variables de instancia
	private ArrayList<Cliente> listaClientes;
	private ArrayList<Paquete> listaPaquetes;

	// Constructores
	public Empresa() {
		listaClientes = new ArrayList<Cliente>();
		listaPaquetes = new ArrayList<Paquete>();
	}

	// acceso y modificación
	public void agregarCliente(Cliente unCliente) {
		this.listaClientes.add(unCliente);
	}

	public void agregarPaquete(Paquete unPaquete) {
		this.listaPaquetes.add(unPaquete);
	}

	public ArrayList getListaClientes() {
		return listaClientes;
	}

	public ArrayList getListaPaquetes() {
		return listaPaquetes;
	}

	// métodos adicionales
	public boolean consultaRecepcionPaquete(int unNumero) {
		boolean retorno = false;
		Paquete auxBuscar = this.obtenerPaquetePorNumero(unNumero);
		if (auxBuscar != null) {
			retorno = auxBuscar.getLlego();
		}
		return retorno;
```

```
	}

	public boolean ingresarAvisoRecepcion(int unNumero) {
		boolean retorno = false;
		Paquete marcar = this.obtenerPaquetePorNumero(unNumero);
		if (marcar != null) {
			marcar.seRecibio();
			retorno = true;
		}
		return retorno;
	}

	public void ingresarPaquete(int unNumero, Cliente cli, Cliente destin) {
		Paquete nuevo = new Paquete(unNumero, cli, destin);
		this.listaPaquetes.add(nuevo);
	}

	public ArrayList<Cliente> listadoClientesDestinatarios() {
		ArrayList<Cliente> retorno = new ArrayList<Cliente>();
		// recorro todos los clientes
		Iterator<Cliente> iterCliente = this.getListaClientes().iterator();
		while (iterCliente.hasNext()) {
			Cliente unC = iterCliente.next();
			// por cada cliente, recorro todos los paquetes
			Iterator<Paquete> iterPaquete = this.getListaPaquetes().iterator();
			boolean envia= false;
			boolean recibe = false;
			while (iterPaquete.hasNext() && !(envia && recibe)){
				Paquete p = iterPaquete.next();
				if (p.getCliente().equals(unCliente)){
					envia = true;
				}
				if (p.getDestinatario().equals(unCliente)){
					recibe = true;
				}
			}
			if (envia && recibe){
				retorno.add(unC);
			}
		}
		return retorno;
	}

	public ArrayList<Paquete> listadoPaquetesSinRecepcion() {
		ArrayList<Paquete> retorno = new ArrayList<Paquete>();
		Iterator busco = this.getListaPaquetes().iterator();
		while (busco.hasNext()) {
			Paquete aux =  busco.next();
			if (!aux.getLlego()) {
				retorno.add(aux);
			}
		}
		return retorno;
	}
```

```
        public Paquete obtenerPaquetePorNumero(int unNumero) {
                Paquete retorno = null;
                Paquete auxBuscar = new Paquete();
                auxBuscar.setNumero(unNumero);
                int indice = this.getListaPaquetes().indexOf(auxBuscar);
                if (indice != -1) {
                        retorno = (Paquete) this.getListaPaquetes().get(indice);
                }
                return retorno;
        }
}
```

Supongamos que incorpora un requerimiento adicional: calcular el total de paquetes por cliente. Una forma de implementar este pedido es utilizando un *HashMap*, con clave el cliente y valor la cantidad de paquetes que tuvo. Así, en la clase Empresa se agrega:

```
  public HashMap totalPaquetesPorClienteTable() {
    HashMap<Cliente, Integer> map = new HashMap<Cliente,Integer>();
    Persona aux;
    for (int k=0; k < this.getListaPaquetes().size(); k++) {
      aux =(this.getListaPaquetes().get(k)).getCliente();
      if (!map.containsKey(aux)){
        // no esta el cliente, lo ingreso en el map
        map.put(aux,1);   // utiliza "autoboxing"
      }
      else {
        // encontre el cliente, le sumo 1
        map.put(aux,map.get(aux)+1);   // aquí hace el "autoboxing"
      }
    }
    return map;
   }
```

Semana 15

Al igual que la semana anterior, se trata de afianzar los conceptos básicos a través de la realización de ejercicios completos.

15.1 Ejercicio: HaceTodo

Realizar el diagrama de clases que represente la situación:
Una empresa de computación vende el paquete denominado HACETODO. El software HACETODO permite entre otras opciones, manejar stock, clientes, proveedores, logística, gestión del marketing, sueldos..., ¡todo!.
Como política de la empresa, cada CD con el paquete está numerado y se guarda información también acerca de quién preparó ese CD.

Desea poder realizar las siguientes operaciones:
a) ingreso al stock de un CD
b) venta de un CD a un cliente. Se ingresa el número de CD y el del cliente. Registrar el día.
c) dado un cliente, indicar que CD tiene.
d) devolución de un CD fallado por parte de un cliente. Se anula la venta.
e) listado de ventas no anuladas.
f) listado de funcionarios
g) listado ordenado por día de venta

Indicar claramente métodos y variables. Implementar en Java.

15.2 Solución Ejercicio HaceTodo

En este ejercicio, aparecen las clases: Cliente, CD, Venta, Funcionario y Empresa. Las descubrimos analizando los términos o sustantivos utilizados en la letra del ejercicio. Cliente y Funcionario podrían heredar de una clase Persona.

Una posible jerarquía de clases se presenta en la Ilustración 21 HaceTodo.

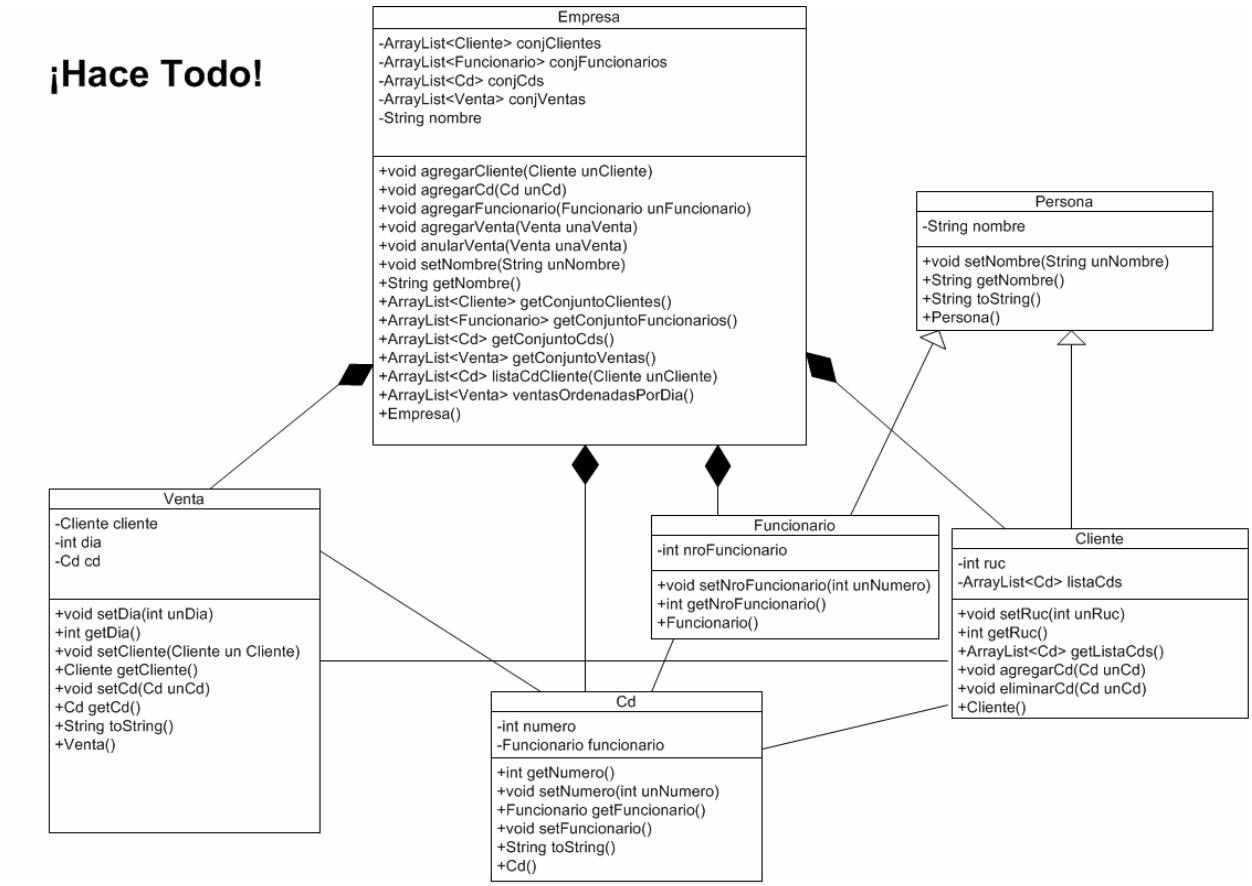


Ilustración 21 HaceTodo

Como posible implementación tenemos:

```
//CLASE PERSONA:

package hacetodo;

public class Persona {
	private String nombre;

	public Persona() {
		this.setNombre( "Sin nombre");
	}

	public java.lang.String getNombre() {
		return nombre;
	}

	public void setNombre(String unNombre) {
		nombre = unNombre;
	}
	@Override
	public String toString() {
		return this.getNombre();
	}
}
```

```
//CLASE FUNCIONARIO:

package hacetodo;

public class Funcionario extends Persona {
	private int nroFuncionario;

	public Funcionario() {
		this.setNroFuncionario(0);
	}

	public int getNroFuncionario() {
		return nroFuncionario;
	}

	public void setNroFuncionario(int unNroFuncionario) {
		nroFuncionario = unNroFuncionario;
	}

}

//CLASE CLIENTE:

package hacetodo;

import java.util.*;

public class Cliente extends Persona {

	private int ruc;

	private ArrayList<Cd> listaCds;

	public Cliente() {
		this.setRuc(0);
		listaCds = new ArrayList<Cd>();
	}

	public ArrayList<Cd> getListaCds() {
		return listaCds;
	}

	public int getRuc() {
		return ruc;
	}

	public void setRuc(int unRuc) {
		ruc = unRuc;
	}

	public void agregarCd(Cd unCd) {
		this.getListaCds().add(unCd);
	}
```

```
	public void eliminarCd(Cd unCd) {
		this.getListaCds().remove(unCd);
	}

}

//CLASE CD:

package hacetodo;

public class Cd {
   private int numero;
   private Funcionario funcionario;

   public Cd() {
      this.setNumero(0);
   }

   public int getNumero() {
      return numero;
   }

   public void setNumero(int unNumero) {
      numero = unNumero;
   }

   public Funcionario getFuncionario() {
      return funcionario;
   }

   public void setFuncionario(Funcionario unFuncionario) {
     funcionario = unFuncionario;
   }
   @Override
   public String toString() {
      return "Cd Nro. " + this.getNumero();
   }

}

//CLASE VENTA:
package hacetodo;

public class Venta {
	private Cliente cliente;
	private int dia;
	private Cd cd;

	public Venta(Cliente unCliente, int unDia, Cd unCd) {
		this.setCliente(unCliente);
		this.setDia(unDia);
		this.setCd(unCd);
	}
```

```
	public Cliente getCliente() {
		return cliente;
	}

	public void setCliente(Cliente unCliente) {
		cliente = unCliente;
	}

	public int getDia() {
		return this.dia;
	}

	public void setDia(int unDia) {
		dia = unDia;
	}

	public Cd getCd() {
		return cd;
	}

	public void setCd(Cd unCd) {
		cd = unCd;
	}
	@Override
	public String toString() {
		return "Venta " + this.getCd().toString() + " de cliente "
				+ this.getCliente();
	}

}

//CLASE EMPRESA:

package hacetodo;

import java.util.*;

public class Empresa {

	private ArrayList<Cliente> conjClientes;
	private ArrayList<Cd> conjCds;
	private ArrayList<Venta> conjVentas;
	private ArrayList<Funcionario> conjFuncionarios;
	private String nombre;

	public Empresa() {
		this.setNombre("sin nombre");
		conjClientes = new ArrayList<Cliente>();
		conjCds = new ArrayList<Cd>();
		conjVentas = new ArrayList<Venta>();
		conjFuncionarios = new ArrayList<Funcionario>();
	}

	public String getNombre() {
		return nombre;
```

```
}

public void setNombre(String unNombre) {
        nombre = unNombre;
}

public void agregarCliente(Cliente unCliente) {
        this.getConjClientes().add(unCliente);
}

public ArrayList<Cliente> getConjClientes() {
        return conjClientes;
}

public void agregarCd(Cd unCd) {
        this.getConjCds().add(unCd);
}

public ArrayList<Cd> getConjCds() {
        return conjCds;
}

public void agregarVenta(Venta unaVenta) {
        this.getConjVentas().add(unaVenta);
        unaVenta.getCliente().agregarCd(unaVenta.getCd());

}

// C) Lista de Cds de un cliente

public ArrayList<Cd> listaCdsCliente(Cliente unCliente) {
        return unCliente.getListaCds();
}

// D) Devolucion de venta

public void anularVenta(Venta unaVenta) {
        unaVenta.getCliente().eliminarCd(unaVenta.getCd());
        this.getConjVentas().remove(unaVenta);
}

// E) Listado de ventas

public ArrayList<Venta> getConjVentas() {
        return conjVentas;
}

public void agregarFuncionario(Funcionario unFuncionario) {
        this.getConjFuncionarios().add(unFuncionario);
}

// F) Lista de Funcionarios
```

```
        public ArrayList<Funcionario> getConjFuncionarios() {
                return conjFuncionarios;
        }

        // G) Listado ordenado por día

        public ArrayList<Venta> ventasOrdenadasPorDia() {
                Collections.sort(this.conjVentas, new CriterioVentaPorDia());
                return this.getConjVentas();
        }

}


//CLASE CRITERIOVENTAPORDIA:
package hacetodo;

import java.util.*;

public class CriterioVentaPorDia implements Comparator<Venta> {

        public int compare(Venta v1, Venta v2) {
                return v1.getDia() - v2.getDia();
        }

}
```

15.3 Práctico 10

Práctico No. 10
Tema: Colecciones: uso de clases para manejo de colecciones.

1) ArrayList
Sea el siguiente código.
ArrayList<Camion> miLista;
miLista = new ArrayList<Camion>();

Se guardarán objetos de la clase Camión. Agregar el código necesario para realizar las siguientes operaciones y consultas:

-¿Qué package es necesario referir?
Se incluye el siguiente código:
        Camion unCamion = new Camion("blanco",123)
-Agregar el objeto unCamion a la lista.
-Agregar 3 camiones más.
-Indicar la posición del elemento unCamion.
-Indicar si la lista contiene un objeto de chapa 123.
-Listar los elementos ordenados por chapa.

2) HashMap

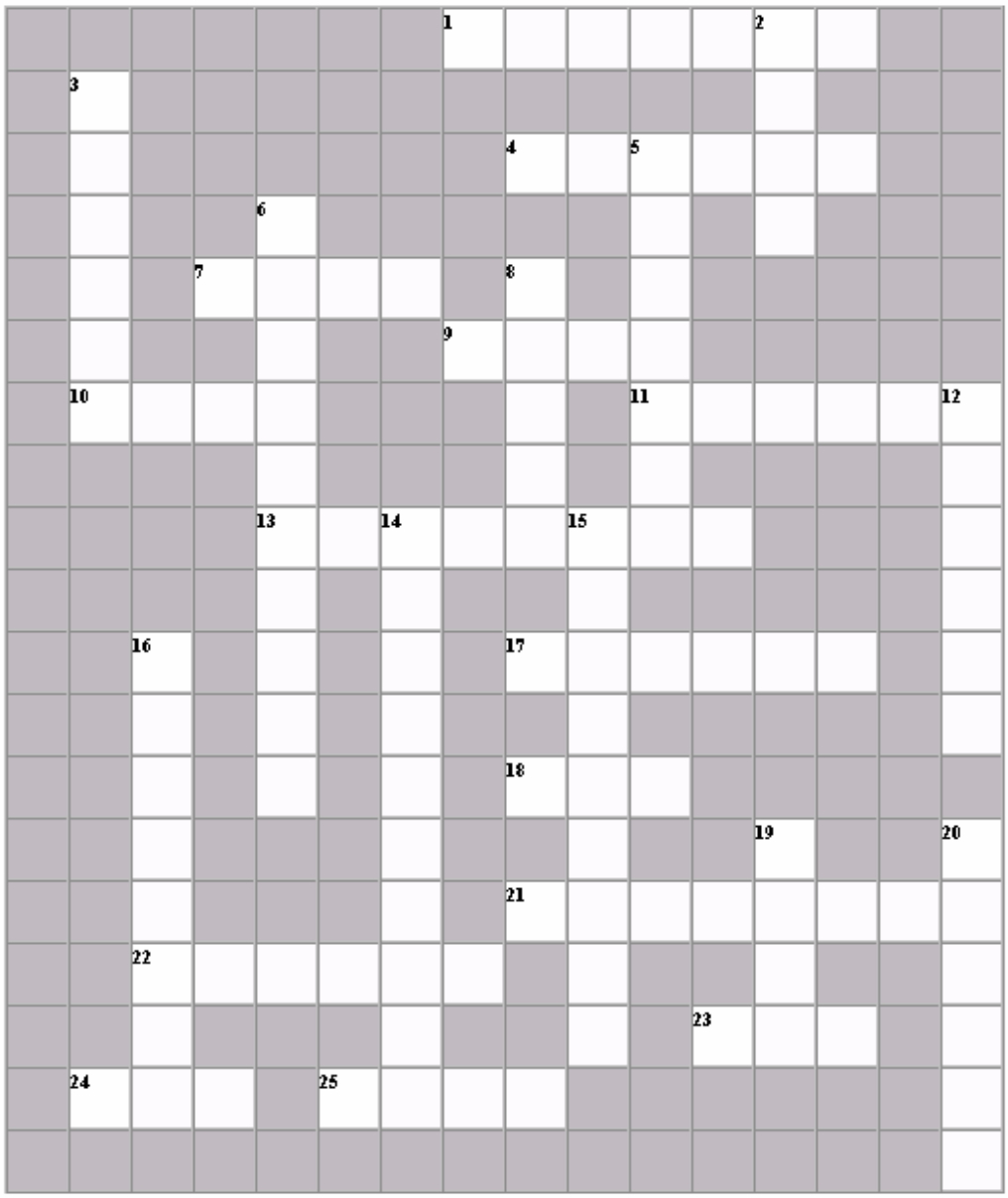
Definir un HashMap e incorporar 4 objetos Camion (c1, c2, c3, c4). Como clave poner cada camión y como valor la cantidad de revisiones del motor (0,0,2,4).

-Anotar el código necesario para realizar las siguientes operaciones y consultas:
-Indicar si el camión c2 está en la estructura.
-Agregar una revisión más al camión c4.
-Listar los elementos
-¿Hay algún camión con 3 revisiones?

3) Completar el siguiente cuadro:

	a) Array	b) ArrayList	c) HashMap
Elementos que permite contener:			
Para crearlos escribir:			
Package a incluir:			
Para averiguar el tamaño:			
¿Está el objeto XX?			
Obtener elemento de posición j			
Agregar elemento XX			
Eliminar objeto XX			
Ordenar			

15.4 Crucigrama



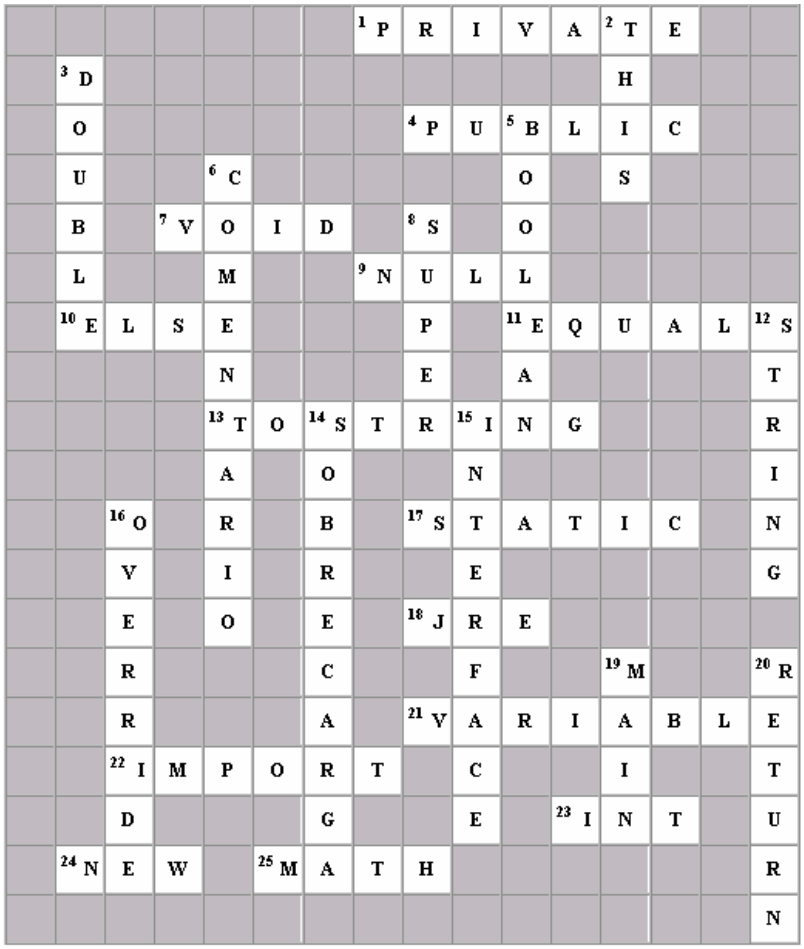
Verticales:

2. Palabra clave para referirse en la clase al objeto que recibió el mensaje
3. Tipos de datos para números decimales: float y .....
5. Puede ser verdadero o falso
6. Texto explicatorio dentro del código
8. Palabra clave para acceder a los elementos de una clase padre
12. Secuencia de caracteres
14. Cuando se define el mismo método con diferentes firmas se habla de ....
15. Define una colección de métodos y valores constantes
16. Dar una implementación diferente de un método heredado (en inglés)
19. Programa principal
20. Termina la ejecución de un método

Horizontales:

1. Significa que no es accesible desde otras clases
4. Significa que es accesible desde otras clases
7. Si un método no retorna nada, se indica a través de la palabra ...
9. Un valor vacío se dice que es ....
10. Se ejecuta si el if es falso
11. Para comparar si dos objetos son iguales se utiliza ...
13. El método por defecto de impresión es el ....
17. Se utiliza para indicar que una variable o método es de clase
18. Sigla de Java Runtime Environment.
21. Lugar de memoria para almacenar un dato. Se nombra con un identificador
22. Para incluir clases de otro paquete se debe poner ....
23. Tipo de datos entero
24. Crea una instancia de una clase
25. Clase que dispone de métodos como valor absoluto, raíz cuadrada, potencia

Solución:



Solución de ejercicios seleccionados

En todos los casos, se presenta una posible solución.

16.1 Solución del Práctico 1

Práctico: 1
Tema: Ejercicios básicos

Resolución de ejercicios seleccionados
(Se presenta una posible solución).

1- Seudocódigo:

```
leer n
suma = 0
cant2 = 0
cant3 = 0
si n > 0
        para (i = 1; i <=n; i++)
                pedir dato
                suma = suma + dato
                si dato % 2 == 0
                        cant2 ++
                si dato % 3 == 0
                        cant3 ++
        mostrar suma, suma / n, cant2, cant3
```

2-Seudocódigo:

```
dato = -1
mayor = 0
mayor2 = 0
mientras dato != 0
        pedir dato
        si dato> mayor
                mayor2 =mayor
                mayor=dato
        en otro caso
                si dato > mayor2
                        mayor2 = dato
mostrar mayor, mayor2
```

Analizar qué pasaría si los números son negativos.

3- Seudocódigo:

```
leer n
mostrar n %10
n = n / 10
mientras n > 0
        mostrar n % 10
        n = n / 10
```

```
5-Seudocódigo:
        sumatodo = 0 {suma todos los numeros}
        sumamax = 0 {suma hasta el máximo}
        max = minValue
        leer n
        para (i=1; i<=n; i++)
                pedir dato
                sumatodo = sumatodo + dato
                si dato > max
                        max = dato
                        sumamax = sumatodo
        si n > 0
                mostrar max, sumamax

6-Seudocódigo:
        (Se asume del mismo día)
        leer he, me, hs, ms "hora y minutos de entrada y salida"
        te = he*60 + me          "paso todo a minutos"
        ts = hs*60 + ms
        mostrar (ts-te)/60, (ts-te) % 60

7-Seudocódigo:
        leer n
        suma = 0
        para (i = 1; i<=n,i=i+2)
                suma = suma + i
        mostrar suma

8- Seudocódigo:
        suma=0
        dato=0
        contador=0
        para (i = 1; i <= 6; i = i + 1)
                leer dato
                si  50 < dato && dato < 70
                        suma = suma + dato
                        contador = contador + 1
        si contador != 0
                mostrar  (suma / contador)
        en otro caso
                mostrar ("Sin datos para el promedio")
```

16.2 Solución del Práctico 2

```
Solución Práctico No.  2
Tema:  Ejercicios básicos en Java

Se presentan soluciones a ejercicios seleccionados.

Soluciones de ejercicios del práctico 1:
package practicoDos;
import java.util.Scanner;
```

```
public class ClasePracticoDos {
	public static void main (String args[] ) {
		// ejercicio 1
		int i;
		int n;
		int dato;
		int suma;
		int cant2;
		int cant3;
		// ingreso cantidad
		Scanner input = new Scanner(System.in);
		System.out.println("Ingrese la cantidad");
		n = input.nextInt();
		suma = 0;
		cant2 = 0;
		cant3 = 0;
		for (i = 1; i<=n; i++)  {
			System.out.println("Ingrese Dato");
			dato = input.nextInt();
			suma = suma + dato;
			if (dato % 2 == 0) {
				cant2++;
			}
			if (dato % 3 == 0) {
				cant3++;
			}
		}
		if (n > 0) {
			System.out.println("La cantidad de datos es "+n);
			System.out.println("Hubo "+cant2 + "pares");
			System.out.println("Hubo "+cant3 + "multiplos de 3");
			System.out.println("Promedio "+(suma/n) );
		}
		else {
			System.out.println("Sin datos ");
		}



		// Ejercicio 3, práctico 1 (se asume n positivo)
		System.out.println("Ingrese el numero para mostrar las cifras ");
		n = input.nextInt();
		System.out.println(n % 10);
		n = n / 10;
		while (n > 0) {
			System.out.println(n % 10);
			n = n / 10;
		}

		// Ejercicio 6, práctico 1
		int he;
		int me;
		int hs;
		int ms;
```

```
                int te;
                int ts;
                System.out.println("Ingrese hora, minuto de ingreso y de egreso");
                he = input.nextInt();
                me = input.nextInt();
                hs = input.nextInt();
                ms = input.nextInt();
                // paso todo a minutos
                te = he * 60 + me;
                ts = hs * 60 + ms;
                System.out.println("Trabajo "+ ((te-ts)/60) + " horas "+ ((ts-te)%60) + "
minutos" );
                // Ejercicio 7, práctico 1
                int suma;
                System.out.println("Ingrese el numero");
                n = input.nextInt();
                suma = 0;
                for (i=1; i <=n; i = i+2) {
                        suma = suma + i;
                }
                System.out.println("La suma es "+suma);
        }
}
2.1)
sacar throws IOException: Error.
sacar la primera “ del literal:  Error
sacar el primer ; Error, espera ;
cambiar “de Programación ” por “ del curso de Semestre ” No hay problemas.
sacar la última ” del literal: Error, falta comilla
sacar la última } Error: falta }
sacar import.java.io.*; Da error, no encuentra clase IOException

2.2)    Ingreso de 10 datos y mostrar la suma
public static void main (String args[] ) {
        int suma = 0;
        Scanner input = new Scanner(System.in);
        for (int i = 1; i <= 10; i ++) {
                System.out.println("dato " + i);
                suma = input.nextInt() + suma;
        }
        System.out.println("suma "+suma);
}


2.5) El código dentro del método main  para imprimir 10,20, ..., 1000 es:

for (int i = 10; i <= 1000; i += 10) {
    System.out.println(i);
}

2.6) // es comentario de línea

        /* ....
        ...
```

```
        */  comentario de múltiples líneas

        /** ...
        ...
        */  comentario para documentación
```

2.7)

```
        x = x + 1;
        x ++;
        + + x;
        x +=1
```

2.8) x vale 7, producto 50 y cociente 2

2.9) Da loop.

16.3 Solución del Práctico 3

Solución Práctico No. 3
Tema: Objetos, clases, relaciones

1)
tabla de símbolos: copy, count, delete, intersect;
array largo variable: todas;
set: copy, count, delete, intersect

2)
a) son medios de transporte
b) son funciones de una variable.
c) son todos para ver. Binoculares y lentes para los dos ojos, el resto es para uno solo. Telescopios y binoculares para ver cosas de lejos, microscopio para agrandar cosas, lentes para agrandar o disminuir.

3) a) resistencia al agua b) vibración, peso c) flexible d) calor
Muestra un mismo objeto del cual considero diferentes aspectos según el problema.

4)
a) asociación
b) generalización-especialización: un archivo puede ser de texto, de directorios o de objetos gráficos
c) asociación
d) asociación

5)
Ideas sobre una posible resolución: En el almacén hay personas (que pueden ser vendedores o clientes), hay artículos (que pueden ser comestibles: perecederos o no perecederos) y de limpieza.
Tambien hay ventas, que asocian un cliente, un vendedor con artículos.
Los clientes saben comprar y pagar. Los vendedores saben vender y cobrar.

6)
Ideas sobre una posible resolución: Hay transporte: camiones, camionetas y motos. También hay personas: empleados y clientes; envíos: remitente (cliente), destino, peso.
Cada envío vincula un cliente con un transporte.

16.4 Solución del Práctico 4

```
Solución  Práctico No. 4
Tema:            Clases y objetos
Objetivo:        Uso de clases predefinidas y creación de nuevas clases.

Ejercicios 1., 2.

package pruebaStringsMath;
import java.io.*;
class ClasePruebaStringMath {
  public static void main (String args[]) throws IOException {

    // Práctico 4, ejercicio 1: prueba de String
    String s1, s2, s3, s4;
    s1 = "Texto de prueba";
    s2 = "texto de prueba con blancos    al final   ";
    s3 = "texto de prueba";

    System.out.println("Comparación de s1 y s3 con == "+ s1+" - " + s3 +" es "+ (s1==s3));
    System.out.println("Comparación de s1 y s3 con equals "+ s1+" - " + s2 + " es "+
(s1.equals(s3)));
    System.out.println("Comparación de s1 y s3 con equalsIgnoreCase "+ s1+" - " + s3 + " es "
+(s1.equalsIgnoreCase(s3)));
    System.out.println("Comparación de s1 y s2 con compareTo "+ s1+" - " + s2 + " es "+
(s1.compareTo(s2)));
    System.out.println("Largo del string "+s2 +" es "+s2.length());
    System.out.println("Paso a minusculas "+s1+ " es " + s1.toLowerCase());
    System.out.println("Paso a mayusculas "+s1+ " es " + s1.toUpperCase());
    System.out.println("Saco blancos a "+s2+" es " +s2.trim()+"-");
    System.out.println("Ubicacion de b en "+s2+" es " + s2.indexOf('b'));

    // Práctico 4, ejercicio 2: Prueba de Math
   int i,j,k;
   float p,r;
   double q;

   i = 10;
   j = 20;
   p = 14.5f;
   q = -45.67;
   System.out.println("El máximo entre "+i+ " y "+j+ " es "+Math.max(i,j));
   System.out.println("El máximo entre "+i+ " y "+p+ " es "+Math.max(i,p));
   System.out.println("El minimo entre "+i+ " y "+p+ " es "+Math.min(i,p));
   System.out.println("El valor absoluto de "+q +" es "+Math.abs(q));
   System.out.println("Potencia de "+i + " al cubo es "+Math.pow(i,3));
   System.out.println("Raíz de "+j + " es "+Math.sqrt(j));

  }
}


3.
```

a) largo / ancho vale 6
b) b) largo / altura vale 8
c) largo % ancho vale 0
d) ancho % altura vale 0.75
e) largo * ancho + altura vale 56.25
f) largo + ancho*altura vale 24.75

4.

```
contador == 50;                          // es contador = 50
wihle (contador <= 60) {                 // es while, no wihle
        System.out.println(contador)     // falta ;
        contador = contador ++;          // origina loop, poner solamente contador++
}
```

5.

```
package pruebaCamion3;

public class Camion {
  // variables de instancia
  private int chapa, modelo;
  private String color, marca;
  // variable de clase
  private static int Año;

  // constructores
  public Camion (){
    this.setColor( "Sin color");
    this.setChapa( 0);
  }
  public Camion(String unC, int unaChapa, int unModelo) {
    this.setChapa(unaChapa);
    this.setModelo( unModelo);
    this.setColor(unC);
  }

  // metodos de acceso y modificacion
  public void  setColor(String unColor) {
    color = unColor;
  }

  public String getColor() {
    return color;
  }

  public void  setChapa(int unaChapa) {
    chapa = unaChapa;
  }

  public int getChapa() {
    return chapa;
  }

  public void  setMarca(String unaMarca) {
    marca = unaMarca;
```

```
  }

  public String getMarca() {
    return marca;
  }

  public void setModelo(int unModelo) {
    modelo = unModelo;
  }

  public int getModelo() {
    return modelo;
  }

  // impresion
  @Override
  public String toString() {
    return "Un camion de color "+this.getColor();
  }

  // comparacion
  public boolean mismoColor(Camion unC) {
    return (this.getColor().equals(unC.getColor()));
  }
  // metodos de clase
  public  static void setAño(int unAño) {
    Año = unAño;
  }

  public static int getAño(){
    return Año;
  }

  public boolean leTocaRevision() {
    return this.modelo <= Camion.getAño();
  }

}


package pruebaCamion3;
import java.util.Scanner;

public class ClasePruebaCamion {
  public static void main (String args[]){

    Camion c1, c2;
    Camion.setAño(1960);
    c1 = new Camion();
    Scanner input = new Scanner(System.in);
    System.out.println("ingrese color y chapa");
    c1.setColor(input.nextLine());
    c1.setChapa (input.nextInt());
    // en forma similar ingresar datos del 2do camion o ponerlos fijos:
```

```
    c2 = new Camion("Rojo", 2222,1962);
    System.out.println(c1);
    System.out.println("Usando toString"+c1.toString());
    System.out.println(c1.mismoColor(c2));
    System.out.println("Chapa del camion 1 es "+c1.getChapa());
    System.out.println("Chapa del camion 2 es "+c2.getChapa());
    System.out.println("Color del camion 2 es "+c2.getColor());
    System.out.println("le toca revision "+c2.leTocaRevision());
  }
}
```

16.5 Solución del Práctico 5

```
Solución Práctico No.  5
Tema: Clases y objetos

1) Una posible implementación es:

package pruebas;

class Triangulo {
  // variables de instancia
  private int lado1;
  private int lado2;
  private int lado3;

  // Constructor
  public Triangulo(int l1, int l2, int l3) {
    this.setLado1(l1);
    this.setLado2(l2);
    this.setLado3(l3);
  }

  // acceso
  private int getLado1() {
    return lado1;
  }
  private int getLado2() {
    return lado2;
  }
  private int getLado3() {
    return lado3;
  }

  public void setLado1(int unLado){
          lado1 = unLado;
  }
  public void setLado2(int unLado){
          lado2 = unLado;
  }
  public void setLado3(int unLado){
          lado3 = unLado;
  }
```

```
  // métodos varios
  public boolean esEscaleno() {
    return (this.getLado1() != this. getLado2())  && (this. getLado1()!= this.getLado3()) &&
    (this. getLado2() != this. getLado3());
  }

   public boolean esEquilatero() {
    return (this. getLado1() == this. getLado2())    && (this. getLado1()== this. getLado3());
     }
    public boolean esIsosceles() {
     return !this.esEquilatero() &&
     ( (this. getLado1() == this. getLado2()) ||
       (this. getLado1() == this.getLado3()) ||
       (this. getLado2() == this.getLado3()) )      ;

  }

  // metodo de prueba
  public static void main (String args[] )  {
    Triangulo unEs = new Triangulo(10,20,30);
    Triangulo unIs = new Triangulo(10,20,10);
    Triangulo unEq = new Triangulo(10,10,10);
    System.out.println("Es escaleno "+ unEs.esEscaleno());
    System.out.println("Es isosceles "+ unIs.esIsosceles());
    System.out.println("Es equilatero "+ unEq.esEquilatero());
  }

}
```

3,4)

```
package funcionario;
import java.util.Scanner;

public class Funcionario {
  // variables de instancia
  private String nombre;
  private double sueldo;
  private String telefono;

  // acceso y modificacion
  public  void setNombre(String unNombre) {
    nombre = unNombre;
  }

  public String getNombre() {
    return nombre;

  }

  public void setSueldo(double unSueldo) {
    sueldo = unSueldo;
  }
```

```
 public double getSueldo() {
  return sueldo;
 }

 public void setTelefono(String unTelefono) {
  telefono = unTelefono;
 }

 public String getTelefono() {
  return telefono;
 }
 // impresion
 @Override
 public String toString() {
  return ("Funcionario de nombre "+this.getNombre() + " gana "+this.getSueldo()
  + "telefono"+this.getTelefono());
 }
 // comparacion
 public boolean ganaMas(Funcionario unFuncionario) {
  return (this.getSueldo()  > unFuncionario.getSueldo());
 }

 // metodos de prueba
 public static Funcionario ejemplo1(){
  Funcionario f;
  f = new Funcionario();
  System.out.println("ingrese nombre,sueldo y telefono ");
  Scanner input = new Scanner(System.in);
  f.setNombre(input.nextLine());
  f.setSueldo(input.nextFloat());
  f.setTelefono (input.nextLine());
  System.out.println(f);
  return f;
 }
 public static void  ejemplo2(){
  Funcionario f1, f2;
  f1 = Funcionario.ejemplo1();
  f2 = Funcionario.ejemplo1();
  System.out.println(f1.ganaMas(f2));

  }
}

package funcionario;
public class ClasePruebaFuncionario {
        public static void main (String args[]){
        Funcionario f1, f2;
        f1 = Funcionario.ejemplo1();
        System.out.println(f1);
        Funcionario.ejemplo2();
    }
}
```

16.6 Solución del Práctico 6

Solución Práctico No. 6

1) mensaje, método
instancias, clase
estructuras jerárquicas, herencia

2)
private static int Ultimo; define una variable de clase de nombre Ultimo y de tipo int, acceso privado.
private String nombre; define una variable de instancia denominada nombre y de tipo String, acceso privado.

3)

```
package area;
import java.util.Scanner;

public class ClaseAreaTrapecio {
  public static void main (String args[]){
    Scanner input = new Scanner(System.in);
    int basemayor;
    int basemenor;
    int altura;
    System.out.println("Ingrese base mayor, menor y altura");
    basemayor = input.nextInt();
    basemenor = input.nextInt();
    altura    = input.nextInt();
    System.out.println("El area es "+(basemayor + basemenor)*altura/2));

    }
}
```

4) En el salón hay personas y materiales. Las personas tienen nombre y dirección. Las subclases Estudiante y Profesor heredan de Persona. Los estudiantes tienen número de estudiante y el profesor tiene un número de funcionario.
Los materiales tienen número de inventario, funcionalidad. Hay sillas y mesas que son un material no eléctrico y hay video, PC y retro que son eléctricos.
Por cada atributo, hay un método que setea el valor y otro que lo retorna.

16.7 Solución del Práctico 7

Solución Práctico No. 7
Tema: Clase Camión

Se incluye la pregunta y su respuesta.

1) ¿Cuál es la función del constructor Camion (String unColor, int unaChapa) ? Ejemplificar, anotando de dos formas diferentes el código de prueba necesario para crear un camión de color rojo y chapa 123.

Permite que automáticamente, al crear un objeto utilizándolo, tome el color unColor y la chapa unaChapa.
Se podría usar:
Camion unCamion;
unCamion = new Camion ("Rojo",123);
o, usando el constructor sin parámetros:
unCamion = new Camion();
unCamion.setColor("Rojo");
unCamion.setChapa(123);

2) El camión creado en el punto anterior es pintado de verde. Anotar el código que refleje esto.
unCamion.setColor("verde")

3) ¿Qué puede representar la variable de clase Año? ¿Cómo se puede inicializar?
Puede representar el año a partir del cual deben chequearse o no los camiones. Es de clase porque todos los camiones tienen una única fecha, aplicable a todos.
Se puede inicializar, por ej. en 1996:

```
Camion.setAño(1996);
```

4) ¿Cómo se puede asegurar que la chapa, una vez que fue asignada, no se modifique?
Eliminando el método que permite modificarla, se permite setearla solamente en el constructor.

5) Se quiere registrar si un camión está a la venta. Agregar el código y ejemplos de su uso.
Agregar una variable de instancia boolean estaALaVenta con métodos de acceso y modificación.
Se podría poner dos métodos de seteo: setEstaALaVenta y setNoEstaALaVenta, ambos sin parámetros.
Se usaría:

```
unCamion.setEstaALaVenta();
```

6) Guardar el modelo del camión (año de construcción).
Agregar una variable de instancia modelo con los métodos de acceso y modificación.

7) Agregar un método que permita saber si al camión le toca revisión (si el año del camión es anterior al Año, le toca).

```
public boolean leTocaRevision() {
        return this.getModelo() < Camion.getAño();
}
```

8) ¿Cuántos camiones fueron creados en esa clase? Analizar cómo se puede llevar esta información.
Agregar una variable de clase, ya inicializada en 0. En los constructores, incrementarla.

9) ¿Qué pasaría si en el método toString se cambia la definición de public por private?
No compila.

10) Crear la clase Motor. Un motor tiene como atributos la cilindrada y el tipo de combustible.
Definir una clase con variables de instancia: cilindrada y tipo de combustible. También los métodos adecuados, por ejemplo: setMotorGasoil, setMotorDiesel, etc.

11) La impresión del motor debe mostrar todas sus características. Implementar el método adecuado.
Agregar el método toString.

12) Agregar al camión un motor. Indicar el código necesario para acceder y modificar dicho atributo. Anotar también código para probarlo.
Agregar una variable de instancia en la clase Camión de nombre motor y los métodos de acceso y modificación. En el constructor, agregar:
this.setMotor(new Motor())

13) La impresión del camión debe incluir también el detalle del motor.
Agregar en el método toString: this.getMotor().toString

14) Al camión creado en 1), ponerle que el motor es a gasoil.
Enviarle al camión el mensaje setMotorGasoil().

16.8 Solución del Práctico 8

Solución Práctico No. 8
Tema: Herencia

1)

```
package practico8;

public class Persona {
  private String nombre;
  public Persona() {
    this.setNombre ("Sin nombre");
  }
  public void setNombre(String unNombre) {
    nombre = unNombre;
  }
  public String getNombre() {
    return nombre;
  }
}

package practico8;
public class Estudiante extends Persona {
  private int numero;
  public Estudiante() {
    this.setNumero(0);
  }
  public void setNumero(int unNumero) {
    numero = unNumero;
  }
  public int getNumero() {
    return numero;
  }
  @Override
  public String toString () {
    return this.getNombre() + " nro "+ this.getNumero();
  }


}
```

```
package practico8;
public class ClasePruebaEstudiante {
  public static void main (String args[] )  {
    Estudiante e;
    e = new Estudiante();
    System.out.println(e);
    // accedo al método heredado
    e.setNombre("Juan");
    e.setNumero(12);
    System.out.println(e);
  }
}
```

Probar cambiando las definiciones public, protected y private.

2) Definir las subclases Mensual y Jornalero. Agregar el método calcularSueldo en cada clase de acuerdo a la definición.

3) En el ejercicio 1, en la clase Persona está definido el constructor sin parámetros, que inicializa el nombre.
En la clase Estudiante, también está el constructor sin parámetros que solamente inicializa el número.
Al crear un estudiante sin parámetros, automáticamente llama primero al constructor sin parámetros de la superclase (Persona) y luego ejecuta el de la propia clase.

4) Una variable de clase es compartida por todas las instancias de una clase.
Los métodos y variables de clase se heredan dependiendo en la forma en que se definieron. Si se puso protected o public, están disponibles en la subclase.

5) a) import java.util.*; Permite que todas las clases del package java.util sean visibles en la clase en la cual se puso esta definición.

b) int number = Math.abs(-12) ; Define una variable integer de nombre number y se inicializa con el valor absoluto de –12, obtenido a partir de la invocación al método abs de la clase Math.

c) System.out.println("hola"); Accede a la variable out de la clase System. Le envía el mensaje println. Investigar en la documentación de Java la definición completa de System.out y System.out.println().

d) BufferedReader in = new BufferedReader (new InputStreamReader(System.in));
Define la variable in de tipo *BufferedReader* y la carga con un objeto de dicha clase. La clase *BufferedReader* permite leer texto de un flujo de caracteres de entrada. Investigar en la documentación de Java para ver la descripción completa.

16.9 Solución del Práctico 9

Solución Práctico No. 9
Tema: Colecciones

2) Una posible implementación de algunos de los métodos pedidos es:

```
import java.util.*;
```

```
import camion.*;
import moneda.*;

public class ClasePruebaArrayList {

	public static void main(String[] args) {

		ArrayList<Object> a;
		Camion camion1, camion2, camion3;
		Moneda moneda;

		// definición del ArrayList
		a = new ArrayList<Object>();

		camion1 = new Camion("rojo", 1111,1);
		camion2 = new Camion("azul", 2222,2);
		camion3 = new Camion("verde", 333,3);

		moneda = new Moneda(100);

		// agregar elementos, puede contener distintos tipos de elementos
		a.add(camion1);
		a.add(camion2);
		a.add(camion3);
		a.add(moneda);

		// agregar el número entero 18
		// esta forma no funciona en versiones anteriores a Java 5,
		// sólo se pueden agregar objetos, no tipos primitivos
		// a partir de Java 5 funciona, pues hace el autoboxing.
		// a.add (18);
		// si quiero agregar el número entero 18 como objeto, creándolo a mano:
		a.add(new Integer(18));

		// pregunto si contiene el elemento moneda
		if (a.contains(moneda)) {
			System.out.println("El objeto moneda: " + moneda + " está");
		} else {
			System.out.println("El objeto moneda: " + moneda + " no está");
		}

		// borrar un elemento
		a.remove(camion1);

		// borrar el primer elemento
		a.remove(0);

		// recorro toda la colección
		for (int i = 0; i < a.size(); i = i + 1) {
			System.out.println("elemento " + i + " es " + a.get(i));
		}
		System.out.println("Imprimo todo " + a);

		// verifico si está vacío
```

```
                if (!a.isEmpty()) {
                        System.out.println("No está vacío");
                }
        }
}
```

3) Hay muchas implementaciones posibles. Una versión preliminar, de prueba de clases, sin menús, es:

```
public class CD implements Comparable<CD> {

        private int numero;
        private String titulo;

        public CD() {
                this.setTitulo("Sin título");
                this.setNumero(0);
        }

        public CD(String unTitulo, int unNumero) {
                this.setTitulo(unTitulo);
                this.setNumero(unNumero);
        }
        @Override
        public String toString() {
                return "CD: " + this.getTitulo() + " nro: " + this.getNumero();
        }

        public String getTitulo() {
                return titulo;
        }

        public void setTitulo(String unTitulo) {
                titulo = unTitulo;
        }

        public int getNumero() {
                return numero;
        }

        public void setNumero(int unNumero) {
                numero = unNumero;
        }
        @Override
        public boolean equals(Object parm1) {
                // Un CD es igual a otro si tiene el mismo título
                return this.getTitulo().equals(((CD) parm1).getTitulo());
        }

        public int compareTo(CD o) {
                // El criterio de ordenación es por el título
                return this.getTitulo().compareTo(o.getTitulo());
        }
}
```

```
import java.util.*;

public class ColeccionCD {

	private ArrayList<CD> coleccionCD;

	public ColeccionCD() {
		coleccionCD = new ArrayList<CD>();
	}

	public void agregarCD(CD unCD) {
		this.getColeccionCD().add(unCD);
	}

	public ArrayList<CD> listarTodos() {
		return getColeccionCD();
	}

	public ArrayList<CD> getColeccionCD() {
		return coleccionCD;
	}

	public boolean existeCD(String unTitulo) {
		CD unCD = new CD();
		unCD.setTitulo(unTitulo);
		return this.getColeccionCD().contains(unCD);
	}

	public void eliminarCD(CD unCD) {
		this.getColeccionCD().remove(unCD);
	}

	public ArrayList ordenar() {
		Collections.sort(this.coleccionCD);
		return this.getColeccionCD();
	}
}

public class PruebaColeccionCD {

	public static void main(String[] args) {

		ColeccionCD miColeccion;
		CD cd1, cd2, cd3;

		miColeccion = new ColeccionCD();

		cd1 = new CD("Mocedades 20 años", 11);
		cd2 = new CD("The Wall", 21);
		cd3 = new CD("Perales", 183);

		miColeccion.agregarCD(cd1);
```

```
                    miColeccion.agregarCD(cd2);
                    miColeccion.agregarCD(cd3);

                    // busco un elemento
                    if (miColeccion.existeCD("Mocedades 20 años")) {
                            System.out.println("En la colección está el CD de título \"Mocedades
20 años\" ");
                    }

                    if (miColeccion.existeCD("Beatles Help")) {
                            System.out.println("En la colección está Beatles Help");
                    } else {
                            System.out.println("En la colección  no está Beatles Help");
                    }

                    // Imprimo los elementos
                    System.out.println("Antes de ordenar " + miColeccion.listarTodos());
                    System.out.println("Luego de ordenar " + miColeccion.ordenar());

                    // cambio de número
                    cd1.setNumero(245);
                    System.out.println(cd1);

                    // Elimino un CD de la lista
                    miColeccion.eliminarCD(cd2);
                    System.out.println(miColeccion.listarTodos());
            }

}


4) Ideas de una posible definición:
Clase Empresa, variables: conjunto de ventas, conjunto de clientes, conjunto de Nylons.
Clase Nylon, variables: tipo, color, inventario, metrosActuales.
Clase Cliente: variables: nombre, direccion.
Clase Venta: variables: cliente, nylon, metros, total
```

16.10 Solución del Práctico 10

```
Solución Práctico No. 10
Tema: Colecciones: uso de clases para manejo de colecciones.

1) ArrayList

package practico10;

//Es necesario importar java.util.*
import java.util.*;
import camion.*;

public class ClaseArrayList {
   public static void main(String[] args) {
      ArrayList<Camion> miLista;
      miLista = new ArrayList<Camion>();
```

```
        Camion unCamion = new Camion( "blanco",123 );

        //Agrego el camion a la lista
        miLista.add( unCamion );

        //Creo otros tres camiones
        Camion camionDos = new Camion( "azul", 789 );
        Camion camionTres = new Camion( "verde", 456 );
        Camion camionCuatro = new Camion( "rojo", 567 );

        //Los agrego a la lista
        miLista.add( camionDos );
        miLista.add( camionTres );
        miLista.add( camionCuatro );

        //Indico la posición del elemento unCamion
        System.out.println( "La posición de unCamion es: "+miLista.indexOf( unCamion ) );

        //Indico si la lista tiene un objeto con chapa 123
        Camion aux = new Camion( "", 123 );
        if( miLista.contains( aux ) ) {
           System.out.println( "miLista tiene un objeto con chapa 123" );
        }
        else {
           System.out.println( "miLista NO tiene un objeto con chapa 123" );
        }



        //Muestro los elementos antes de ordenar usando for
        System.out.println( "--- Lista sin ordenar ---" );
        for( int i=0; i < miLista.size(); i++ ) {
           System.out.println( miLista.get(i) );
        }
        //Ordeno miLista aprovechando el método compareTo que agregué en Camion que
compara chapas.
        Collections.sort( miLista );
        //Muestro los elementos ordenados usando un iterator
        Iterator<Camion> miIterator = miLista.iterator();
        System.out.println( "--- Lista ordenada ---" );
        while( miIterator.hasNext() ){
           System.out.println( miIterator.next() );
        }
    }

}


En  la clase Camion se agregó:
import   java.util.*;
public class Camion implements Comparable<Camion>{
.... codigo original

  public int compareTo(Camion otro){
```

```
                return this.getChapa()-otro.getChapa();
}


2) HashMap

package practico12;

//Es necesario importar java.util.*
import java.util.*;
import camion.*;

public class ClaseHashMap {
        public static void main(String[] args) {
                // Defino un HashMap
                HashMap<Camion, Integer> miMapa = new HashMap<Camion, Integer>();

                // Creo cuatro camiones
                Camion c1 = new Camion("blanco", 123);
                Camion c2 = new Camion("azul", 789);
                Camion c3 = new Camion("verde", 456);
                Camion c4 = new Camion("rojo", 567);

                miMapa.put(c1, 0);
                miMapa.put(c2, 3);
                miMapa.put(c3, 2);
                miMapa.put(c4, 4);

                // Indico si c2 está en la estructura
                if (miMapa.containsKey(c2)) {
                        System.out.println("c2 está en la estructura");
                } else {
                        System.out.println("c2 no está en la estructura");
                }

                // Recupero el número anterior de revisiones
                int cantRevisionesC4 = miMapa.get(c4);
                // Agrego una revisión a c4
                miMapa.put(c4, cantRevisionesC4 + 1);

                // Listo los elementos
                Iterator miIterator = miMapa.entrySet().iterator();
                while (miIterator.hasNext()) {
                        Map.Entry entrada = (Map.Entry) miIterator.next();
                        System.out.println("Clave: " + entrada.getKey() + " Valor: "
                                        + entrada.getValue());
                }

                // ¿Hay algún Camion con 3 revisiones?
                if (miMapa.containsValue(3)) {
                        System.out.println("Hay algun camión con 3 revisiones");
                } else {
                        System.out.println("NO Hay ningun camión con 3 revisiones");
                }
```

```
        }

}
```

3)
Se presentan algunos de los métodos disponibles. En algunos casos, hay otras posibilidades.
a) array, b) ArrayList, c) HashMap
Permite contener
array: Elementos de la misma clase
ArrayList: Colección arbitraria de objetos
HashMap: Estructura de datos en la cual un elemento (clave) es usado para ubicar un segundo elemento (valor)

Creación:

```
Camion arrayCamion[];
arrayCamion = new Camion[3];

ArrayList<Camion> miLista;
miLista = new ArrayList<Camion>();

HashMap<Camion,Integer> hashCamion;
hashCamion = new HashMap<Camion,Integer> ();
```

Para averiguar el tamaño:
a) arrayCamion.length; (no es método) tamaño fijo
b) arrayListCamion.size(); tamaño dinámico
c) hashCamion.size(); tamaño dinámico

Para saber si está un cierto objeto:
a) NO
b) arrayListCamion.contains(XX);
c) hashCamion.containsValue(XX); (si es valor)
hashCamion.containsKey(XX); (si es clave)

Para ubicar el elemento de posición j:
a) arrayCamion[j]
b) arrayListCamion.get(j);
c) NO

Para agregar elemento XX
a) arrayCamion[...] = XX
b) arrayListCamion.add(XX);
c) hashCamion.put(clave, valor); indicar clave y valor

Para eliminar elemento XX
a) NO
b) arrayListCamion.remove(XX);
c) hashCamion.remove(XX); (XX debe ser clave)

Para ordenar:
a) algoritmos de ordenación
b) Collections.sort();
c) usar estructura auxiliar

Bibliografía básica recomendada y referencias

Budd, T. (2001). An introduction to Object Oriented Programming (3rd edition). USA: Addison Wesley Longman Inc.

Eckel, B. (2006). Thinking in Java (4th edition). USA: Prentice Hall PTR

Flanagan, J (2005). Java in a nutshell (5th edition). USA: O'Reilly Media.

Fowler, M.; Scott, K. (1999). UML distilled. (2nd Edition). USA: Addison Wesley Longman.

Lewis, J.; Loftus, W. (2008). Java Software Solutions. Foundations of program design (6th edition). USA: Addison-Wesley.

Savitch, W. (2005). Java: an introduction to computer science and programming. (4th edition). USA: Prentice Hall Inc.

http://www.java.sun.com La página de Sun. Ahí se puede bajar Java
http://www.jguru.com Contiene preguntas frecuentes y sus respuestas.
http://developer.java.sun.com Contiene varios tutoriales, glosario, quizzes, etc.
http://www.bruceeckel.com Libros electrónicos sobre Java.

Índice Alfabético

M

N

O

P

R

S

T

U

V

W

Educando para la vida

Cuareim 1451 Tel. 2902 15 05 Fax 2908 13 70
www.ort.edu.uy

Julio 2011