Ficha 3

Casting – Javadoc – Herencia

1.] Uso de la clase Scanner para leer archivos de texto.

Hemos visto que la clase *java.util.Scanner* brinda métodos directos para permitir la carga de datos desde el teclado de la consola estándar. Sin embargo, los objetos de la clase *Scanner* pueden también ser redireccionados para hacer que en lugar de tomar datos desde el teclado, los tome desde un archivo de texto que siga cierto formato en cuanto al espaciado interno.

Veremos que esto resulta de mucha utilidad cuando se presentan ocasiones en las que el programador deberá procesar secuencias de números que vendrán almacenados en archivos de texto plano. Lo común es que en esas circunstancias se tenga un archivo de texto con tantas líneas como números se suponen representados en él. Cada línea será en rigor una cadena de caracteres, terminada con un caracter de salto de línea, pero de modo que cada caracter es un dígito (ver el archivo Numeros.txt que acompaña al modelo TSB-Scanner).

Obviamente, la tarea consiste en leer cada una de la líneas del archivo de texto, convertir cada cadena leída a un número (int, long, float o del tipo que se requiera) y procesar cada número como tal, independientemente de si el programador decide (o no) almacenar primero esos números en un arreglo o en una lista.

Una forma de hacer esto, *a partir de la versión Java 1.5*, consiste (como dijimos) en usar un objeto de la clase *java.util.Scanner*. La idea es que al crear un objeto de la clase *Scanner*, se asocie al mismo un archivo o flujo de entrada desde el cual se tomarán los datos. Como ya sabemos, ese archivo o flujo de entrada normalmente es la entrada estándar *System.in*, y eso permite usar los métodos de la clase *Scanner* para tomar datos directamente desde el teclado, como se ve en el ejemplo siguiente:

```
Scanner sc = new Scanner(System.in);
System.out.print("Ingrese un numero entero: ");
int i = sc.nextInt();
```

Sin embargo, la aplicación que nos interesa mostrar aquí es la de poder tomar datos desde un *archivo de texto* (tal como *numeros.txt*). Para hacer eso, el constructor de la clase *Scanner* recibe como parámetro un objeto de la clase File que encapsula el nombre físico del archivo a abrir, y en caso de no existir arroja una excepción de la clase *FileNotFoundException* que debe ser prevista con un *catch()*. En general, el archivo que se desea abrir debería estar contenido en *la raiz de la misma carpeta del proyecto*. Más adelante veremos la forma de trabajar con *excepciones* en Java, pero por ahora bastará con tener ya definida la estructura básica necesaria para abrir el archivo usando un objeto de la clase *Scanner* (ver el modelo de código fuente que sigue más abajo).

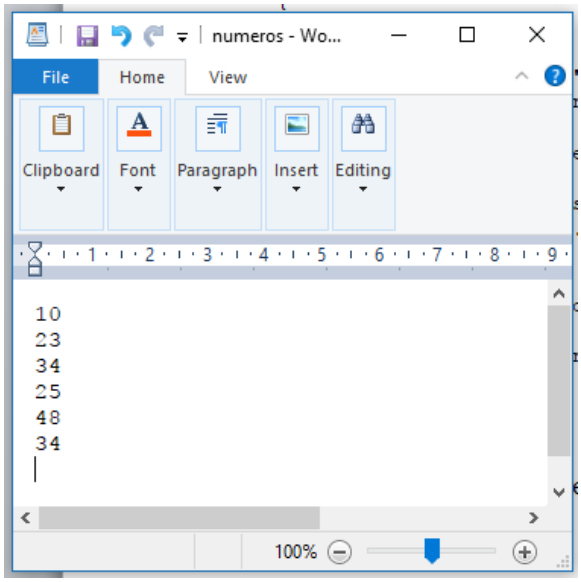
La clase *Scanner* también provee métodos de la forma *hasNextInt()*, *hasNextFloat()*, etc., que permiten preguntar si en la entrada existe un próximo valor del tipo deseado. Esto permite

entonces usar un ciclo para recorrer el archivo de entrada hasta que se llegue al final del mismo. Y lo más interesante, los métodos *nextInt()*, *nextFloat()*, etc., leen el siguiente valor y ya lo retornan convertido al tipo deseado, sin tener que usar métodos de conversión.

El resto, se deduce simplemente del esquema siguiente (y puede verse implementado en el modelo *TSB Scanner*):

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;

/**
 * Una clase de prueba para mostrar el uso de objetos Scanner para
 * leer archivos de texto que contienen números.
 *
 * @author Ing. Valerio Frittelli.
 * @version Agosto de 2017.
 */
public class Test
{
    public static void main(String args[])
    {
        System.out.println("Lectura de numeros en un archivo de texto...");

        File f = new File("numeros.txt");
        try (Scanner sc = new Scanner(f))
        {
            while (sc.hasNextInt())
            {
                int num = sc.nextInt();
                System.out.println(num);
            }
        }
        catch (FileNotFoundException ex)
        {
            System.out.println("No existe el archivo...");
        }
    }
}
```

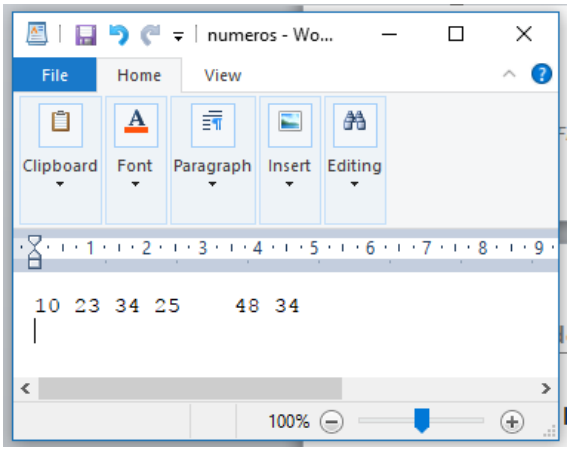
Si el archivo *numeros.txt* existe en la carpeta *TSB Scanner*, y contiene la secuencia:



entonces la salida producida por el programa que mostramos más arriba será de la forma:

```
Lectura de numeros en un archivo de texto...
10
23
34
25
48
34
BUILD SUCCESSFUL (total time: 0 seconds)
```

Notemos que los números contenidos en el archivo de entrada podrían estar separados entre sí por un salto de línea (como en el ejemplo anterior), pero también podrían estar separados por uno o más espacios en blanco y la clase Scanner de todos modos haría la lectura en forma correcta. Si el archivo fuese el siguiente:



el programa volvería a mostrar la misma salida:

```
Lectura de numeros en un archivo de texto...
10
23
34
25
48
34
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.] Asignación cruzada de valores: casting.

Hemos visto que Java presenta un conjunto de ocho tipos de datos llamados *tipos simples* o *tipos primitivos*. Cuando se declara una variable de alguno de esos tipos, se está reservando memoria para ella: tantos bytes como corresponda al tipo elegido. Luego de declarada, una variable puede recibir valores del tipo al que pertenece.

Pero en Java (como en muchos lenguajes) en ciertas situaciones se pueden hacer *asignaciones cruzadas*: valores de un tipo pueden asignarse en variables de otro tipo. Cuando esto está permitido por el compilador, se dice que los tipos implicados *son compatibles*. Por ejemplo, todos los tipos numéricos son compatibles entre sí: un valor de cualquier tipo entero (*byte*, *short*, *int* o *long*) se puede asignar en una variable de coma flotante (*float* o *double*) y viceversa (aunque en este caso, no en forma directa). Y el tipo *char* es compatible con los tipos numéricos. El único tipo primitivo que no es compatible con

ningún otro, es *boolean*: cualquier intento de asignar un valor *boolean* (*true* o *false*) en una variable de otro tipo, no compilará. Y tampoco podrá asignarse un valor que no sea *boolean* en una variable *boolean*. Las líneas de color azul en el siguiente ejemplo, provocarán un error de compilación por asignación de tipos incompatibles:

```
boolean r = true;
int x = r; // error de compilación

int a = 24;
boolean b = a;        // error de compilación
```

Si dos tipos son compatibles, se puede asignar valores de uno de esos tipos en variables del otro. Para poder hacerlo, el valor asignado se convierte en un valor del tipo de la variable receptora. Pero esta conversión de tipos (llamada *casting* en Java) a veces es *directa y automática*, y otras veces es *indirecta*: el programador debe solicitar la conversión de tipos.

Para entender en qué casos el *casting* es directo y en qué casos no, conviene asumir que entre los tipos compatibles hay una relación de "tamaños" comparativos. En principio, un tipo "es menor" que otro si sus variables ocupan menos bytes en memoria. Así, *byte* es menor que *short* (pues una variable *byte* ocupa un *byte* mientras que una variable *short* ocupa dos). Sin embargo, se asume que los tipos de coma flotante son siempre mayores que cualquier tipo entero (*float* es mayor que *long*, aunque *float* acupa 4 bytes por variable y *long* usa 8). En realidad, lo que se mide es la capacidad de cada tipo para representar valores en un rango mayor: una variable *float* (y por ende una *double*) puede representar valores numéricos en un rango mucho mayor que el máximo posible en *long*. Esto es debido a que la representanción binaria de coma flotante es diferente a la representación binaria entera. En esta escala de comparaciones de tipos, se puede asumir que *char* es menor que cualquier entero, y obtener la siguiente relación:

```
char < byte < short < int < long < float < double
```

Con lo anterior asumido, las reglas de *casting* son (en general) simples:

a. Si se quiere asignar una variable de un tipo menor a una variable de un tipo mayor (pero compatible) *entonces la asignación es directa* (el casting o conversión de tipos es *automático* e *implícito*). Los siguientes ejemplos son todos válidos:

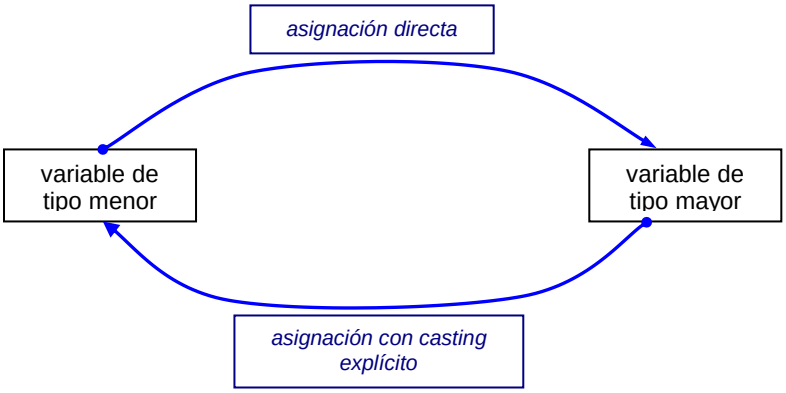
```
byte a = 13;
int b = a;
long c = b;
float d = a;
double e = b;
```

b. Si se quiere asignar una variable de un tipo mayor a una variable de tipo menor (pero compatible), *entonces la asignación requiere del operador de casting explícito* (el casting o conversión de tipos *debe ser requerido explícitamente por el programador*). El *operador de casting* consiste en colocar entre paréntesis el nombre del tipo hacia el cual se quiere convertir, delante del valor o expresión cuyo tipo se quiere convertir. Los siguientes ejemplos son todos válidos y en ellos los *operadores de casting* se colocaron en azul:

```
long f = 113;
int g = (int) f;
byte h = (byte) g;
double i = 98.456;
```

```
int j = (int) i;
```

La siguiente gráfica muestra un resumen de la idea general de ambas reglas:



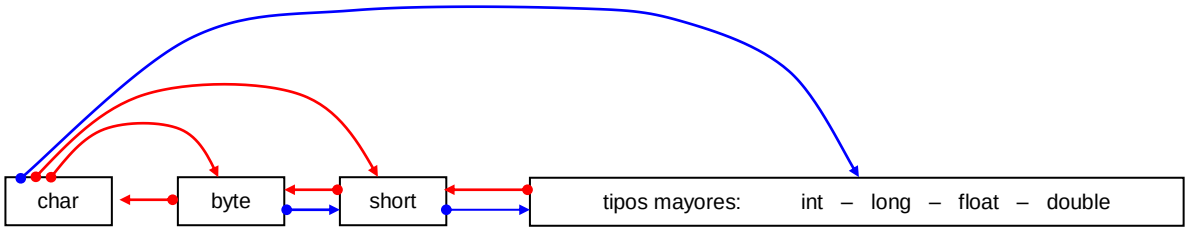
Al operar con el tipo char, las reglas generales son las mismas: una variable numérica de cualquier tipo puede asignarse en una variable *char*, usando ***siempre*** casting explícito (ya que *char* es menor que todos los numéricos):

```
int v1 = 65;
char c1 = (char) v1;

float v2 = 66.123f;
char c2 = (char) v2;

byte v3 = 68;
char c3 = (char) v3;
```

Sin embargo, hay un par de excepciones notables: si una variable *char* se asigna en una variable *byte* o *short* (y sólo para estos dos tipos), ***también*** debe usarse casting explícito. Si la variable *char* se asigna en una variable de tipo mayor a *short*, la asignación es directa (como se esperaba):

```
char c4 = 'A';
byte v4 = (byte) c4;

char c5 = 'C';
short v5 = (short) c5;

char c6 = 'B';
int v6 = c6;

char c7 = 'E';
float v7 = c7;
```

Finalmente, y en base a lo dicho, las reglas de asignación cruzada de variales pueden resumirse en este esquema, en el cual las flechas azules significan asignación directa, y las flechas rojas significan asignación con operador de casting explícito:



Debe notarse un detalle importante: si se usa casting explícito para convertir un valor de tipo mayor en un valor de tipo menor, *el lenguaje hará truncamiento de bits si el valor mayor no cabe en la variable receptora*. Dicho de otro modo, Java permitirá la asignación pero no garantizará que la asignación se haga correctamente, a menos que el valor asignado quepa en la variable receptora. Por ejemplo, sabemos que en una variable de tipo *byte* sólo pueden asignarse valores en el rango [-128 .. 127]. Si se intenta asignar un valor fuera de ese rango usando casting explícito, el valor asignado no coincidirá con el valor realmente alojado al terminar:

```
int x = 230;
byte y = (byte) x;
System.out.println("Y: " + y);
```

En el ejemplo mostrado, aparecerá el mensaje *Y: -26*: El valor 230 excede el rango válido para el tipo *byte*, *pero al ser forzada la asignación con casting explícito, se asigna una parte de la representación de bits del 230 en la variable y*, quedando el valor –26 en ella. Es responsabilidad del programador evitar estas situaciones, asegurándose que el casting explícito sólo se use si el valor a asignar realmente puede ser contenido en la variable receptora. En el ejemplo anterior, si el valor asignado fuera 100 o cualquier otro entre –128 y 127, no se habría presentado ningún problema.

Observemos además que si se asigna el valor de una variable numérica a una variable *char*, o viceversa, lo que realmente se está manejando es el *valor numérico* que corresponde a cada caracter en la tabla de representación usada por la computadora. Normalmente esos valores corresponden al sistema de codificación conocido como *ASCII* (por *American Standard Code for Information Interchage*: Código Estandar Americano (estadounidense) para el Intercambio de Información)[1]. En la tabla *ASCII*, por ejemplo, la letra 'A' tiene el número 65, la 'B' el 66, etc. y usando esos números convertidos a binario se representan internamente todos los caracteres posibles para la computadora. Por lo tanto, una secuencia de la forma:

```
int n1 = 65;
char c1 = (char) n1;
System.out.println("Símbolo: " + c1);
```

provocará una salida tal como: *Símbolo: A* en la consola estandar. La variable *c1* contiene realmente el valor 65, pero al manipularlo para salidas por consola el mismo se reemplaza por el caracter asociado. Lo mismo vale a la inversa:

```
char c2 = 'B';
int n2 = c2;
System.out.println("Valor: " + n2);
```

El mensaje mostrado será ahora: *Valor: 66* (pues 66 es el valor *ASCII* del caracter 'B').

Todas las explicaciones realizadas hasta aquí en relación al casting *son válidas si los valores asignados vienen dentro de otra variable*. Pero si el valor asignado ***es una constante*** (no almacenada en una variable, sino directamente asignada), las reglas de conversión pueden cambiar, pues ahora el compilador puede predecir si la asignación es válida o no de acuerdo

[1] En realidad, Java usa un sistema de codificación más amplio, llamado *Unicode*. El sistema *ASCII* requiere un byte por cada caracter, mientras que *Unicode* requiere dos. Pero el sistema *ASCII* es un subconjunto del sistema *Unicode*, por lo cual las explicaciones de esta sección son válidas de todos modos.

al valor de la constante asignada. Dos secuencias de instrucciones que a priori no parecerían diferentes, pueden provocar distintos tratamientos por parte del compilador. La siguiente no compila:

```
int a = 78;
byte b = a;      // no compila… debió ser:  byte b = (byte) a;
```

pero la siguiente, que aparentemente termina haciendo lo mismo, compila sin problemas:

```
byte b = 78;
```

La diferencia entre una secuencia y la otra, es que en la segunda *el valor asignado en b viene directamente como una constante*, y en ese caso el compilador puede determinar si el valor entra o no en la variable *b*. Si entra, compila. Si no entra, no compila. La siguiente asignación no compila, pues el valor 200 no entra en una variable byte:

```
byte b = 200;  // no compila...
```

Repetimos: si el compilador puede predecir la validez de la asignación, esta compilará o no sin que importen las reglas vistas de casting para asignaciones de variables. En los siguientes ejemplos, la primera linea compila pero la segunda no:

```
// ok... el compilador puede predecir que el resultado es 127...
byte b = 126 + 1;

// no compila... el compilador puede predecir que el resultado es 128...
byte c = 127 + 1;
```

En la primera, el compilador puede ver que el resultado será 127, y que ese valor cabe sin problemas en una variable *byte*. Por lo tanto, compila. En la segunda, el compilador predice que el resultado será 128, y como ese valor no entra en una variable *byte*, no compila.

Finalmente, notemos que si en una asignación hay una expresión en la parte derecha (una fórmula con varias variables y operadores como sumas, restas, etc.), entonces el resultado de la expresión será de diferente tipo de acuerdo a los tipos de los valores que intervienen en la expresión, de acuerdo a las siguientes reglas:

i. Si la expresión contiene variables (y eventualmente constantes), pero ***todas*** las variables y constantes son de tipo *int* o *menos que int*, entonces el resultado de la expresión será *int*: observe que en la última asignación del ejemplo que sigue, las variables *b*, *e*, *f* son de tipo *byte* o *short*: todas son de tipo *menor a int*, y según la regla, el resultado de operar con variables que son todas menos que *int* da *int*… La variable *a* en la cual se hizo la asignación no puede ser *byte* o *short*:

```
byte a, b;
int c, d;
short e, f;

// asumir que las variables son
// asignadas aquí con valores...

// algunas expresiones de prueba...
c = a + d + e;    // correcto
e = a + c + f;    // no compila...
a = b + e + f;    // no compila!!!!
```

ii. Si la expresión contiene variables y eventualmente constantes (***pero NO todas constantes***), el resultado será del tipo de la variable o constante *de mayor tipo que aparezca*. Recuerde que una constante de coma flotante en Java se considera *double*:

```
long f, g;
float h, i;
double j, k;
int x, y;

// asumir que las variables son
// asignadas aquí con valores...

// algunas expresiones de prueba...
k = j + h + f;        // correcto
h = g + j + x;        // no compila…
f = h + i + x + k;    // no compila…
h = f + i + 3.23;     // no compila…
```

3.] Clases de envoltorio (Wrapper classes).

El lenguaje Java provee una serie de clases que permiten representar valores de tipos primitivos como objetos. De esta forma, incluso los tipos primitivos pueden usarse en situaciones en que se requieran objetos y no valores primitivos de tipo simple: veremos que esa capacidad resulta muy útil y necesaria, por ejemplo, para manejar objetos de clases que representan estructuras de datos ya implementadas en Java, en el *package java.util*, o en estructuras de datos genéricas que serán diseñadas a lo largo del curso.

Esas clases se implementan definiendo simplemente un atributo del tipo al que se desea representar, y dotando a la clase de métodos para manejar el valor almacenado. Debido a que ese valor aparece como *envuelto en la clase*, esas clases se designan como *clases de envoltorio* (o *wrapper classes*), y existe una por cada tipo primitivo:

Tipo primitivo	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Todas estas clases proveen métodos para convertir el valor contenido a un *String* y viceversa, además de métodos para acceder al valor representado. Notar que estas clases están marcadas ellas mismas como *final* (no pueden ser derivadas) y sus atributos también son *final* (el valor de los atributos no puede modificarse una vez creado el objeto). Se usan diversos métodos de estas clases cuando se necesita convertir cadenas que contienen números hacia valores *int* o *float* u otros tipos numéricos (por ejemplo, para realizar la conversión de un *String* que contiene la *representación de un número* a un número mediante los métodos *Integer.parseInt()* o *Float.parseFloat()*).

En el siguiente esquema se crean objetos de algunas de estas clases primitivas, y se obtienen luego los valores contenidos para pasarlos a variables primitvas comunes. Note que cada clase de envoltorio provee al menos un método de la forma *intValue()*, *floatValue()*, *charValue()*, etc, que permite recuperar en forma de valor primitivo el valor envuelto por el objeto:

```
Integer i1 = new Integer(23);
Float f1 = new Float(2.34f);
Character c1 = new Character('$');
int i2 = i1.intValue();
float f2 = f1.floatValue();
char c2 = c1.charValue();
```

Note, sin embargo, que esta forma de pasar de un objeto wrapper a una variable primitiva y viceversa, era la única forma de hacerlo hasta la versión 1.4 (incluida) del lenguaje. Pero a partir de la versión 5.0 esto sigue siendo válido (lo anterior obviamente compilará en las nuevas versiones del lenguaje), pero también está disponible la posibilidad de pasar en forma directa entre los objetos wrapper y las variables primitivas: esta característica se conoce como *auto-boxing* (cuando se asigna valor primitivo directamente sobre una referencia de tipo wrapper) o como *auto-deboxing* (cuando se asigna un objeto wrapper directamente sobre una variable primitiva), y es por supuesto mucho más cómoda. Lo que sigue es equivalente al esquema anterior, pero con *auto-boxing*, y compilará si usted dispone de un compilador desde la versión 5.0 o posterior de Java:

```
// asignación de primitivos en objetos wrapper, con auto-boxing...
Integer i1 = 23;
Float f1 = 2.34f;
Character c1 = '$';

// extracción del valor contenido en el objeto, con auto-deboxing...
int i2 = i1;
float f2 = f1;
char c2 = c1;
```

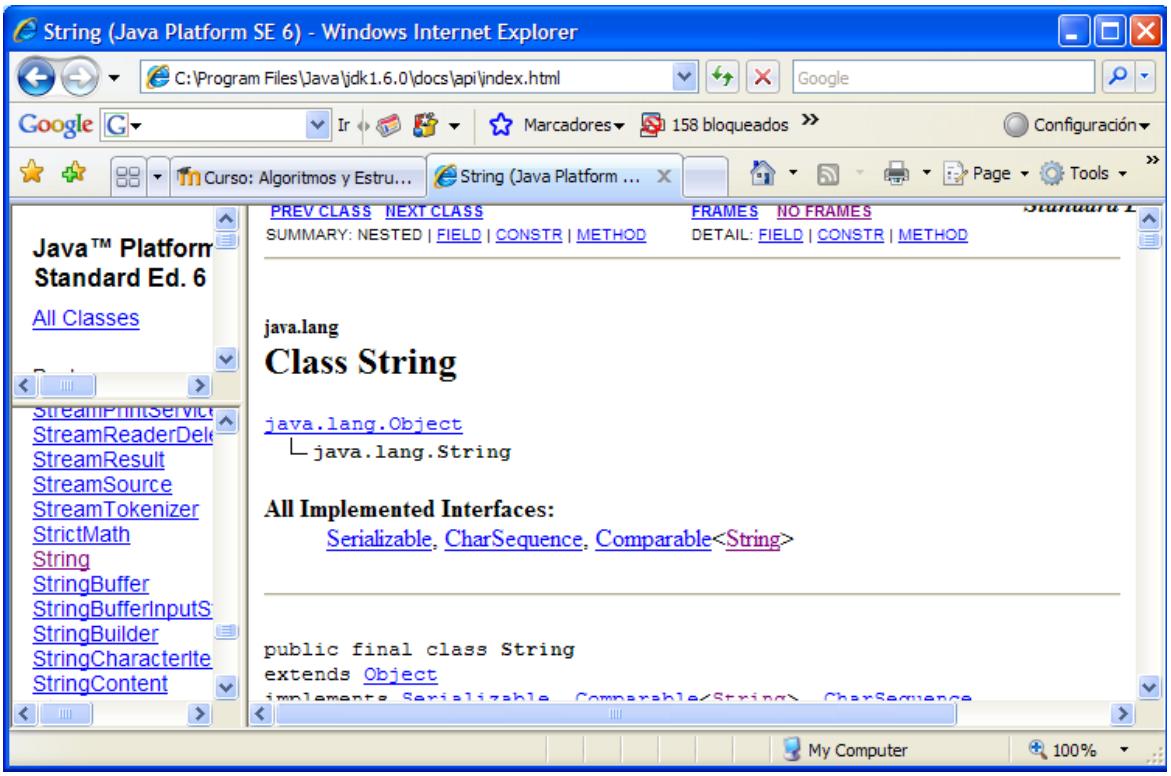
Observe que en las tres primeras lineas se están creando tres objetos de las clases *Integer*, *Float* y *Character*, pero el compilador ya no exige un *new* explícito: ese *new* es insertado en forma automática por el compilador, y el valor primitivo que está a la derecha del operador de asignación es automáticamente enviado al constructor de la clase wrapper que esté participando en forma implícita. En esto, el compilador realiza el mismo servicio que ya realiza cuando se asigna directamente una cadena sobre una referencia de tipo *String* (y los programadores, agradecidos... ☺).

4.] Documentación técnica de una clase: generación de documentación Javadoc.

Un elemento importante a considerar en la fase de desarrollo de un sistema informático es la documentación técnica del mismo. Sabemos que esa tarea es harto tediosa y larga, y pocos profesionales de la informática gustan de hacerla. Sin embargo, todos comprendemos la importancia de dejar claramente expresado para qué sirve cada clase, cuál es el objetivo de cada una, qué elementos públicos tiene, para qué sirve cada método público, qué retorna cada uno, o qué significa para cada método cada uno de los parámetros que toma. En otras palabras, es importante “dejar a la posteridad” los manuales de uso de nuestros desarrollos.

Hay muchas posibilidades en cuanto al formato de esos “manuales”, pero en la plataforma Java el formato viene ya definido casi como un estándar. Si no se trabaja on-line, la

documentación técnica de las clases nativas del lenguaje puede bajarse (download) desde el sitio oficial de la empresa Oracle, que es la propietaria de la plataforma Java[2], e instalarse en forma local. Al hacer el download, se obtiene un archivo *.zip* que puede ser descomprimido en donde prefiera el usuario. Sugerimos se haga dentro de la carpeta donde haya sido instalado Java. La descompresión genera una carpeta *docs*, dentro de la cual aparecen a su vez varias otras. Una de ellas se llama *api*, y contiene un archivo llamado *index.html*. Ese archivo es la página de entrada a la documentación de ayuda como la mostrada en la figura siguiente, con links a todas las demás.



Ahora bien: esa documentación muestra la forma en que están armados los manuales de uso de las clases predefinidas de Java. Pero ¿qué hay de las clases del programador? Por supuesto, lo deseable es que un programador arme sus manuales con el mismo formato, pero la buena noticia es que Java permite hacer eso de forma automática, si el programador tuvo el cuidado de introducir ciertos comentarios especiales en cada una de sus clases. Esos comentarios se llaman *comentarios de documentación*, o también, *comentarios javadoc*. Un *comentario javadoc* comienza con el signo /** (barra y doble asterisco) y finaliza con */ (asterisco y barra). En ese sentido, Java proporciona entonces tres tipos de comentarios:

- ✓ *Comentarios de línea simple o única*: cualquier texto que comience con // será considerado un comentario y el compilador ignorará ese texto. El efecto del comentario finaliza cuando termina la línea del texto que se inició con //.
- ✓ *Comentarios de párrafo*: cualquier texto de una o varias líneas que comience con /* y termine con */ también será considerado un comentario e ignorado por el compilador.

[2] URL: http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html

- ✓ *Comentarios de documentación o javadoc*: son similares a los comentarios de párrafo, pero comienzan con /** (dos asteriscos en lugar de uno sólo) y cierra con */ (igual que el de párrafo).

La diferencia entre los dos primeros y el último tipo de comentario, es que el último está para permitir generar documentación de ayuda y se siguen ciertas reglas más formales para introducir comentarios de esta clase que para los dos primeros. El objetivo de los dos primeros es ayudar al programador a poner en claro una idea o llamar la atención sobre una instrucción o varias, pero no generar documentación técnica de ninguna clase.

Cada entorno integrado de programación en Java puede generar archivos HTML con la documentación *javadoc* de nuestras clases, automáticamente. Lo que terminan haciendo todos estos entornos, es invocar a la aplicación *javadoc.exe* que viene en el JDK (en el mismo directorio donde está la máquina virtual y el compilador). Puede ejecutar ese programa desde la línea de órdenes del sistema operativo, pero también (por comodidad), el programador no necesita preocuparse por salir a la línea de órdenes sino que pude activarla desde alguna opción del interior del IDE que esté usando.

La aplicación *javadoc.exe* toma como entrada el archivo fuente de una clase (el archivo con extensión *.java*), analiza ese documento en busca de todos los *comentarios javadoc válidos*, y genera como salida otro archivo en formato HTML, llamado igual que el fuente, pero con extensión *.html* en lugar de *.java*. Si desde la línea de órdenes del sistema operativo se hace algo como esto:

```
C:\\javadoc Cuenta.java
```

entonces la herramienta *javadoc* generará un archivo *Cuenta.html* con la documentación de la clase *Cuenta* contenida en el archivo *Cuenta.java*, armada a partir de los *comentarios javadoc* colocados en ella por el programador.

Los *comentarios javadoc* no se colocan en cualquier parte de la clase. Deben ir en ciertos lugares predefinidos, y además, dentro de cada *comentario javadoc* pueden aparecer ciertas palabras reservadas que luego son tomadas para aparecer como texto resaltado en el HTML final. Lo normal que el primer comentario javadoc aparezca al principio del archivo de la clase, antes que ninguna otra cosa, y que en ese comentario se explique el objetivo de la clase, su forma de uso, etc. En ese mismo primer comentario, suelen aparecer dos palabras reservadas:

- **@author** para indicar el nombre del autor de la clase, y que luego aparecerá resaltado en negrita en el HTML.
- **@version** para indicar un número de versión o la fecha de última modificación o de creación de la clase, y que también aparecerá resaltado.

Por otra parte, también se coloca un comentario javadoc inmediatamente antes de cada miembro público de la clase, explicando para qué sirve ese miembro. Si el miembro es un método, se usan además las siguientes palabras reservadas:

- **@return** para indicar qué devuelve el método. Se estila hacer una breve explicación conceptual respecto del valor devuelto.
- **@param** se usa una de estas por cada parámetro que el método pide: si el método toma tres parámetros, deben usarse tres etiquetas @param. En cada una de ellas, debe indicarse el nombre del parámetro TAL Y COMO ESTA declarado en el método, y luego explicar su utilidad para el método.

- **@throws** para indicar si ese método puede lanzar alguna excepción. Normalmente se escribe el nombre de la clase de excepción lanzada.
- **@exception** idem *@throws*.
- **@deprecated** para indicar que ese método es ya anacrónico en esta versión de la clase, y que no debería usarse.

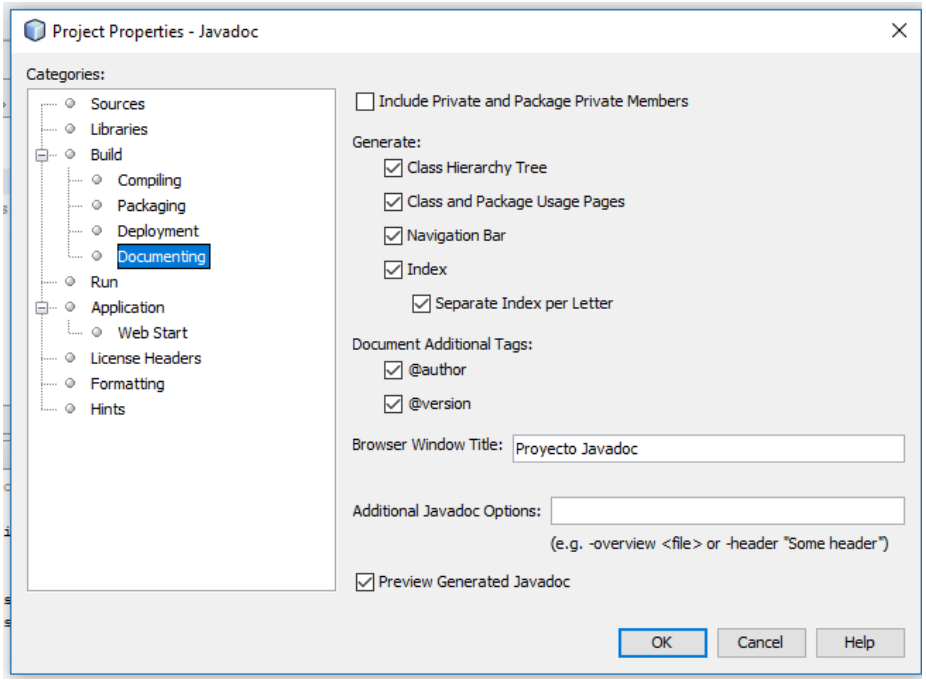
El siguiente ejemplo muestra una clase *Persona* con comentarios javadoc en regla:

```
/**
    Representa una Persona con datos básicos. Consideramos que una Persona
    tendrá un nombre y un número que indica su edad.
    @author   Ing. Valerio Frittelli.
    @version  Agosto de 2017
*/
Public class  Persona
{
   private String nombre;
   private int edad;

   /**
      Crea un objeto con valores tomados como parámetro.
      @param nom el nombre de la Persona.
      @param ed la edad de la Persona.
    */
   public Persona(String nom, int ed)
   {
      nombre = nom;
      edad   = ed;
   }

   /**
      Retorna el nombre de la Persona representada.
      @return un String con el nombre de la persona.
    */
   public String getNombre()
   {
      return nombre;
   }

   /**
      Retorna la edad de la Persona representada.
      @return un número entero con el valor de la edad de la persona.
    */
   public int getEdad()
   {
      return edad;
   }

   /**
      Permite cambiar el nombre de la Persona.
      @param nom el nuevo nombre.
    */
   public void setNombre(String nom)
   {
            nombre = nom;
   }

   /**
      Permite cambiar la edad de la Persona.
      @param ed la nueva edad.
```

```
     */
    public void setEdad(int ed)
    {
       edad = ed;
    }

    /**
       Arma y retorna una cadena sencilla con el contenido del objeto, lista
       para ser visualizada si fuera necesario.
       @return la representación como String del contenido del objeto.
     */
    public String toString()
    {
            return "\n\tNombre: " + nombre + "\tEdad: " + edad;
    }
}
```

El proyecto *Javadoc* que se adjunta con esta Lección, es un programa Java muy simple que contiene la clase *Persona* aquí mostrada, más una pequeña clase *Principal* con un método *main()* a modo de prueba. Cada una de esas clases tiene comentarios javadoc en regla, y usaremos ese proyecto para mostrar un ejemplo del proceso que debe llevarse a cabo para generar la documentación javadoc desde dentro de NetBeans (si el alumno usa otros entornos de programación, también podrá generar documentación javadoc pero deberá consultar los manuales de ayuda del entorno o el help del mismo para saber cómo hacerlo).

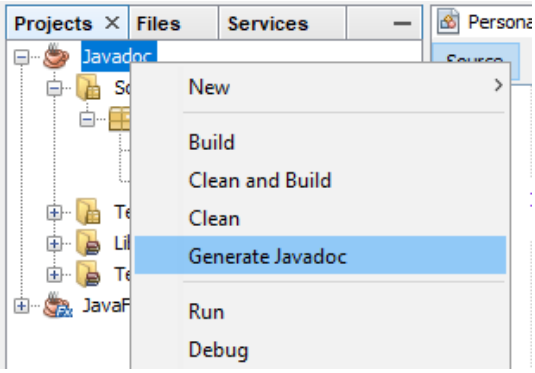
Si está usando NetBeans, asegúrese de tener compilado y sin errores el proyecto cuya documentación javadoc desea generar. Por supuesto, asegúrese también de haber incluido correctamente en cada clase de ese proyecto todos los comentarios javadoc que sean necesarios. Luego, ingrese en la opción *File* del menú superior de NetBeans, y dentro del menú contextual que aparece seleccione la opción *Project Properties*. En el cuadro de diálogo que aparece, haga click en la sección *Documenting* dentro de la rama *Build*.



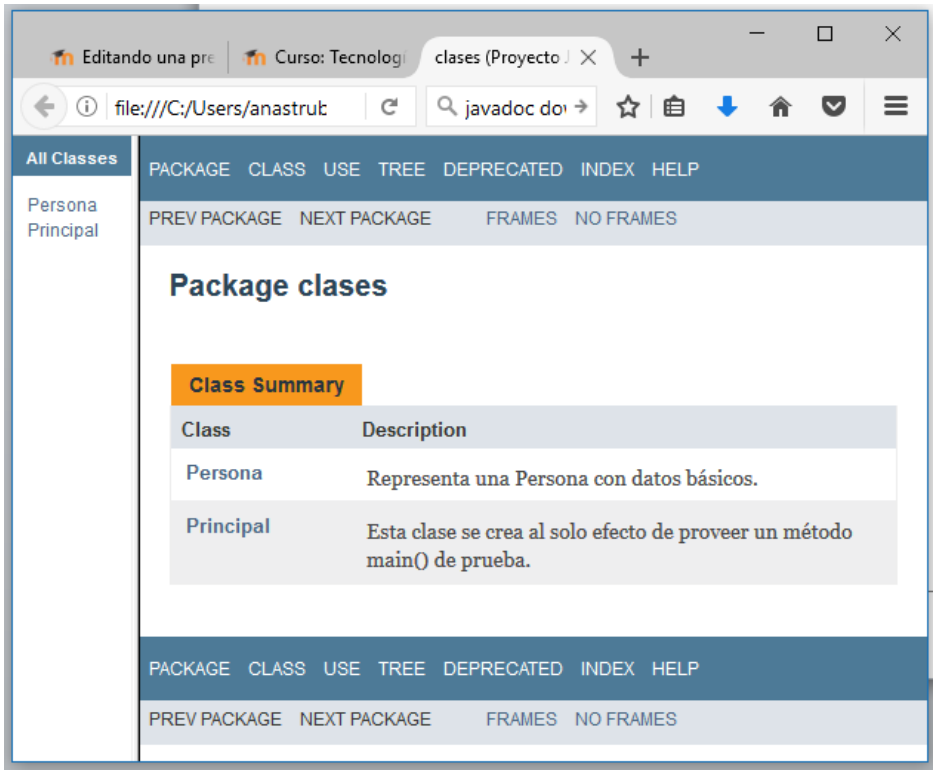
En ese cuadro de diálogo puede ajustar distintas opciones de configuración para la documentación javadoc que se está por generar. Las opciones que mostramos marcadas en

la figura anterior suelen ser las que se ajustan para una salida típica (invitamos al estudiante a probar combinaciones y efectos distintos). Cuando presione el botón Ok, los cambios en esta configuración serán guardados por NetBeans.

Lo único que resta por hacer, es solictar a NetBeans que efectivamente active el proceso de generación de la documentación javadoc. Para ello, apunte con el mouse al proyecto en el marco del Explorador de Proyectos, pulse el botón derecho del mouse, y en el menú contextual que aparece seleccione la opción *Generate Javadoc*:



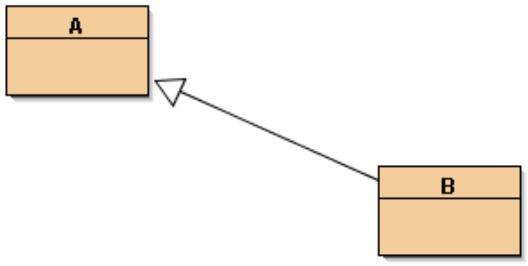
Al hacerlo, se abrirá el navegador web de su computadora y desplegará el contenido completo de la documentación javadoc de su proyecto (como se ve en el ejemplo de más abajo). Por otra parte, la documentación generada de esta forma es un conjunto de documentos HTML que se almacena dentro de la misma carpeta del proyecto, pero en la subcarpeta *distr*. Dentro de esa subcarpeta, aparece a su vez otra con el mismo nombre del proyecto, que contiene a su vez un documento llamado *index.html* que es la página de entrada a la documentación: usted puede hacer doble click sobre ese archivo, y su contenido se desplegará en el navegador Web.



5.] Herencia.

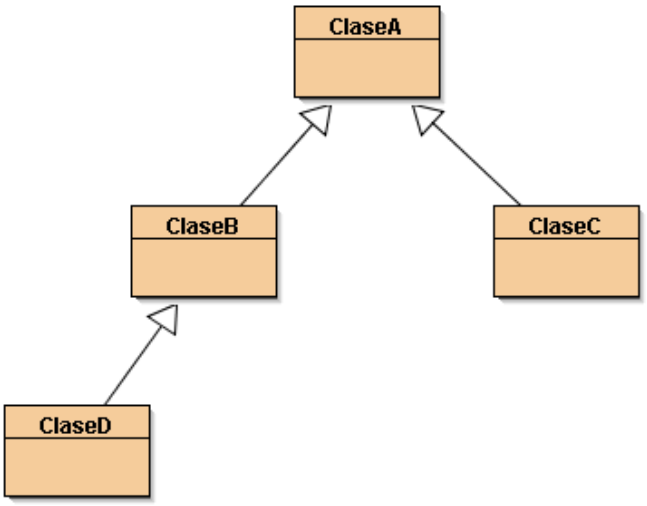
En Programación Orientada a Objetos se llama *herencia* al mecanismo por el cual se puede definir una nueva clase *B* en términos de otra clase *A* ya definida, pero de forma que la clase *B* obtiene todos los miembros definidos en la clase *A* sin necesidad de hacer una redeclaración explícita. El solo hecho de indicar que la *clase B hereda (o deriva) desde la clase A, hace que la clase B incluya todos los miembros de A como propios* (a los cuales podrá acceder en mayor o menor medida de acuerdo al calificador de acceso [*public*, *private*, *protected*, o *"default"*] que esos miembros tengan en *A*).

Cuando la clase *B* hereda de la clase *A*, se dice que hay una relación de herencia entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o *clase derivada*) que sería *B* en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es *A* en nuestro caso):



La clase desde la cual se hereda, se llama *super clase*, y las clases que heredan desde otra se llaman *subclases* o *clases derivadas*: de hecho, la herencia también se conoce como *derivación de clases*.

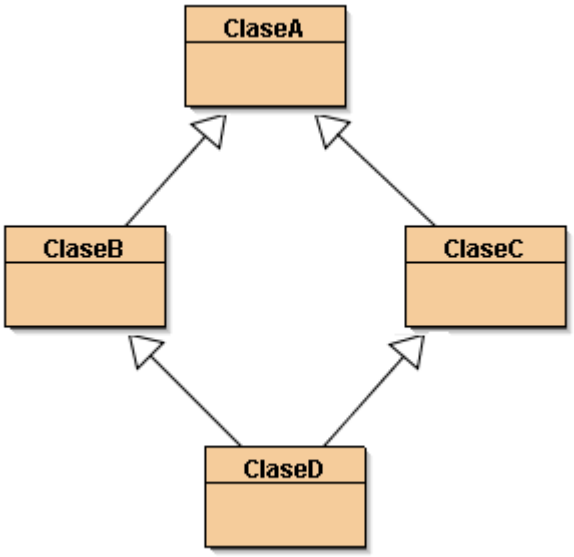
Una *jerarquía de clases* es un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como *clase base* de la jerarquía. La idea es que la clase base reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una pequeña jerarquía de clases:



En esta jerarquía, la *clase base* es *ClaseA*. Las clases *ClaseB* y *ClaseC* son *derivadas directas de ClaseA*. Note que *ClaseD* deriva en forma directa desde *ClaseB*, pero en *forma indirecta también deriva desde ClaseA*, por lo tanto todos los elementos definidos en *ClaseA* también estarán contenidos en *ClaseD*. Siguiendo con el ejemplo, *ClaseB es super clase de ClaseD*, y *ClaseA es super clase de ClaseB y ClaseC*.

En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

- ✓ ***Herencia simple:*** Si se siguen reglas de *herencia simple*, entonces *una clase puede tener una y sólo una super clase directa*. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen herencia simple. La clase *ClaseD* tiene una sóla super clase directa que es *ClaseB*. No hay problema en que a su vez esta última derive desde otra clase, como *ClaseA* en este caso. El hecho es que en herencia simple, a nivel de gráfico UML, sólo puede existir una flecha que parta desde la clase derivada hacia alguna super clase.
- ✓ ***Herencia múltiple:*** Si se siguen reglas de *herencia múltiple*, entonces *una clase puede tener tantas super clases directas como se desee*. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay herencia múltiple: note que *ClaseD* deriva en *forma directa* desde las clases *ClaseB* y *ClaseC*, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las clases *ClaseB* y *ClaseC* contra *ClaseA* es de herencia simple... tanto *ClaseB* como *ClaseC* tienen una y sólo una super clase directa: *ClaseA*.



No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la *implementación de clases de interface* (que oportunamente discutiremos). ***El caso es que Java NO SOPORTA herencia múltiple***: una jerarquía de clases como la mostrada en el gráfico anterior no es posible de ser definida en Java. Un ejemplo de un lenguaje que SÍ SOPORTA herencia múltiple es C++.

En general, y en términos básicos de UML, una *relación de uso* implica que *un objeto de una clase **usa** a un objeto de otra*. Pero una *relación de herencia* implica que un objeto *b* de una clase *B*, ***es** a su vez un objeto de otra clase A*. Cuando se diseñan clases, una pregunta que

puede orientar a definir si debe usarse o no herencia entre las clases *A* y *B*, siendo que *A* está ya definida, es la siguiente: *¿se puede decir que los objetos de la clase B son objetos de la clase A?* Por ejemplo, si tenemos diseñada una clase *Persona*, y queremos diseñar una clase *Cliente* (para un banco), la pregunta muestra en forma clara la relación: ¿Un *Cliente* es una *Persona*? Como la respuesta es sí, se podría hacer que la clase *Cliente herede de Persona*. Pero con respecto a las *Cuentas*... *¿un Cliente **es** una Cuenta?* ó *¿un Cliente **usa** una Cuenta?* Si bien para algunos bancos podría ser cierta la primera pregunta, en general es cierta la segunda, y por lo tanto se esperaría que un objeto de la clase *Cliente tenga un atributo que sea una referencia a un objeto de la clase Cuenta*, y NO que *Cliente herede de Cuenta*....

En el modelo TSB-Herencia entregado como anexo a esta ficha, asumimos una jerarquía sencilla de clases que representan cuentas bancarias. La clase base *Cuenta* contiene sólo un *número de cuenta* y un *saldo*, más los métodos para manejar esos atributos. El tipo de cada cuenta es lo que definirá a cada subclase, pues cada una tiene comportamiento diferente de acuerdo a su tipo. Por ahora asumimos que los métodos *depositar()* y *retirar()* serán iguales para todas las clases derivadas.

Cuando hay herencia (y por lo tanto una jerarquía de clases), cobra nueva importancia el *constructor por defecto* (es decir, el constructor sin parámetros y con bloque de acciones vacío). Por lo tanto veamos las reglas que sigue el compilador Java:

- Si un programador ***no incluye*** ningún constructor en una clase, el compilador incluirá un *constructor por defecto en el código de byte de la clase* (archivo con extensión *.class*), y por ello cualquier creación de instancias invocando al mismo, compilará.
- Si el programador ***incluye*** algún constructor que **NO** sea el constructor por defecto, entonces el compilador no incluirá el constructor por defecto en el *.class*, y cualquier intento de crear una instancia sin enviar parámetros a su constructor no compilará.

Para indicar que una clase hereda de otra, se usa en Java la palabra reservada *extends* al declarar la clase derivada, nombrando luego de ella a la super clase de la misma. Esto hace que todo el contenido de la super clase esté presente también en la derivada, sin tener que volver a definirlo. Como el lenguaje Java no soporta herencia múltiple, al declarar una clase derivada y luego de la palabra *extends* no puede escribirse más que el nombre una única clase. Las siguientes declaraciones, tomadas del modelo TSB-Herencia, muestran la forma de declarar que las clases *Inversion* y *Corriente* derivan de la clase *Cuenta*:

```
public class Inversion extends Cuenta
{
     // contenido de la clase Inversion
}

public class Corriente extends Cuenta
{
     // contenido de la clase Corriente
}
```

En Java, si al declarar una clase *no se indica si la misma deriva de otra* (o sea, no se escribe *extends*), entonces el lenguaje asume que esa clase hereda desde la clase *Object*, que es la base de toda la jerarquía de clases de Java (predefinidas o no: incluso las clases del programador derivan desde *Object* en Java). En otras palabras, si al definir, por ejemplo, una clase *Persona* no se indica de quien hereda:

```
public class Persona
```

entonces el compilador Java reemplaza la declaración anterior por la siguiente en el bytecode:

```
public class Persona extends Object
```

Este es el motivo por el cual el método *toString()* (y varios otros) están presentes en una clase aún cuando la misma no los redefina. La clase *Object* provee una serie de métodos que serán comunes a todo objeto. Algunos de esos métodos se usan tal cual vienen desde *Object*, sin redefinirlos, pero otros (*toString()*, por ejemplo) deberían ser redefinidos por cada clase para adecuar su funcionamiento al conjunto de atributos de la clase. La clase *Object* provee los siguientes métodos (los nombramos sólo a los efectos documentales): *toString() – finalize() – getClass() – clone() – equals() – hashCode() – wait() – notify()* y *notifyAll()*. Y todos estos métodos están entonces presentes en toda clase Java predefinida o definida por el programador.

Atención a los constructores: al invocar un constructor de una clase derivada, Java espera que ese constructor *invoque a su vez a algún constructor de la super clase*. Esa invocación debería ser incluida en la primera línea del bloque de acciones del constructor de la derivada. La palabra reservada *super* puede usarse para invocar explícitamente a un constructor de la *super clase*. Si el constructor de la clase derivada *no incluye* una llamada explícita a algún constructor de la super clase, entonces Java impone implícitamente una llamada al *constructor por default* de la super clase. Si ese constructor no estuviera en la super clase y no hubiera sido insertado automáticamente por el compilador, se provocará un error de compilación. Por ese motivo la inclusión del *constructor default* es recomendable, aunque pueda parecer innecesaria. Veamos los siguientes ejemplos, tomados del modelo *TSB-Herencia*:

```
public Corriente()
{
}

public Corriente(int num, float sal, boolean des)
{
    super(num, sal);
    descubierto = des;
}
```

El primer constructor mostrado es el *constructor default* de la clase *Corriente* (que sabemos que deriva de la clase *Cuenta*). Ese constructor no incluye ninguna instrucción explícita y por lo tanto tampoco incluye una llamada explícita a un constructor de la clase *Cuenta*. En este caso, el compilador reemplaza esa definición por esta otra cuando genera el archivo *Corriente.class*:

```
public Corriente()
{
   super();  // invocación explícita al constructor default de Cuenta
}
```

Note el uso de la palabra reservada *super* para hacer que un constructor de una clase derivada pueda invocar a un constructor de la super clase. La definición aquí mostrada puede ser indistintamente usada por el programador, en lugar de la originalmente mostrada: son equivalentes.

El segundo constructor mostrado para la clase *Corriente* asigna el parámetro *des* en el atributo *descubierto*, pero antes de eso invoca en forma explícita al constructor de la clase

Cuenta que recibe dos parámetros. En este caso el programador impone una llamada al constructor que necesita. Note que se pasan parámetros a *super* para elegir la sobrecarga correcta del constructor a invocar. También note que si la llamada explícita al constructor no se hubiera hecho, el compilador invocaría de nuevo al constructor *default* de la clase *Cuenta*. En ese sentido, la siguiente definición:

```
public Corriente(int num, float sal, boolean des)
{
   descubierto = des;
}
```

equivale por completo a esta otra (y es lo que entendería el compilador):

```
public Corriente(int num, float sal, boolean des)
{
        super();
        descubierto = des;
}
```

En el caso del modelo *TSB-Herencia*, la clase *Inversion* hereda desde *Cuenta* todos sus atributos y métodos, agrega nuevos métodos (*actualizar()*), y redefine algunos otros (*toString()*). La clase derivada podrá redefinir un método heredado si el planteo del método como está en la super clase no tuviera en cuenta el comportamiento y los atributos específicos de la derivada: el método *toString()* tal y como se hereda desde *Cuenta*, retorna una cadena que no incluye a la tasa de interés, y eso resulta incompleto para la clase *Inversion*. Por otra parte, los métodos *getNumero()*, *setNumero()*, *getSaldo()* y *setSaldo()* sirven tanto a la super clase como a la derivada, y por lo tanto no se redefinen: simplemente se usan. Si la clase derivada incluye métodos propios (caso: *actualizar()*), entoncesos esos métodos no pueden ser usados por la super clase: sólo existen en la derivada.

Atención a las diferencias:

- *Redefinir* un método es la acción que tiene lugar en una clase derivada, para volver a definir un método heredado desde la super clase pero tal que su especificación no es adecuada a la derivada. Para *redefinir* un método, la clase derivada debe volver a escribir su cabecera pero TAL CUAL COMO FIGURA DECLARADA EN LA SUPER CLASE, excluyendo eventualmente el calificador de acceso, y cambiando su implementación en el bloque de acciones. Al redefinir un método, en la clase derivada valdrá entonces el redefinido, y no el heredado.
- *Sobrecargar* un método es la ación que tiene lugar en una clase cualquiera, para agregar distintas versiones del mismo método en esa clase (caso: los constructores). Para *sobrecargar* un método, debe mantenerse el mismo nombre pero debe modificarse la forma de la lista de parámetros del mismo. No se toma como sobrecarga la diferenciación en el tipo de salida del método.

Observar el efecto de los *calificadores de acceso* cuando hay herencia: si en la super clase los atributos o métodos son *privados*, entonces son *inaccesibles* incluso para las clases derivadas. Ver el método *actualizar( )* de la clase *Inversión*, en el cual debimos usar *getSaldo()* y *depositar()* para acceder al atributo *saldo*. Si hubiésemos nombrado directamente a ese atributo en algún método de la clase *Inversión*, habríamos provocado un error de compilación. En cambio, si en la super clase los atributos fueran *protected*, serían accesibles para las clases derivadas como si fueran públicos, aunque no lo serían para clases que no deriven de esa super clase (hay otras reglas para el alcance de estos calificadores, que oportunamente discutiremos).

Una vez creada la jerarquía, podemos crear instancias de cualquiera de las clases en ella. Al invocar métodos, Java seguirá la pista del método invocado hasta la definición hecha en la clase del objeto, siempre y cuando ese método aparezca definido en la base de la jerarquía. Un objeto de una clase más arriba en la jerarquía, no puede invocar un método cuya primera definición aparezca en alguna clase más abajo. Pero lo contrario es válido: un objeto de una clase más abajo, puede invocar métodos definidos más arriba (el método *main()* de la clase *Principal* en el modelo *TSB-Herencia* muestra varios ejemplos de los dicho, y el estudiante puede probar distintas combinaciones para observar sus efectos).