Ficha 5

Miembros static y final – Excepciones – Listas Genéricas

1.] Miembros estáticos de una clase: el calificador static.

Sabemos que los distintos miembros (atributos y métodos) de una clase pueden marcarse como *public* o *private* (entre otros estados) según se desee imponer con más o menos fuerza el *Principio de Ocultamiento*. Sin embargo, existen otras palabras reservadas que pueden usarse para indicar la forma de acceder o compartir un miembro de la clase. Dos de ellas son las palabras *static* y *final*.

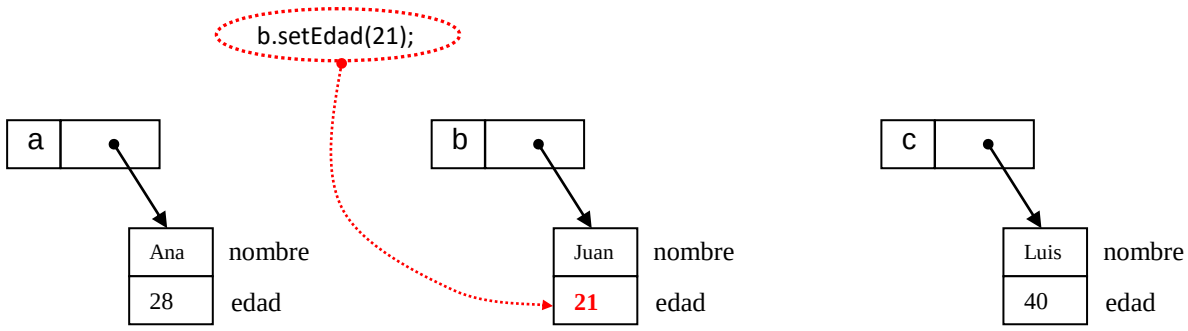
Si se tiene una clase cualquiera, cada vez que se crea un objeto mediante el operador *new* ese objeto tendrá dentro de sí una copia propia de todos los atributos de la clase. Cada objeto tendrá su propio juego de atributos y si un objeto modifica el valor de uno de sus atributos, eso no cambiará los valores de ese atributo en el resto de los objetos. Sea por ejemplo la clase *Persona* que usamos como modelo en el proyecto *TSB-Static* que acompaña a esta ficha:

```
class  Persona
{
   private String nombre;
   private int edad;

   public Persona(String nom, int ed)
   {
      nombre = nom;
      edad   = ed;
   }

   public String getNombre()
   {
      return nombre;
   }

   public int getEdad()
   {
      return edad;
   }

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

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

   public String toString()
   {
```

```
            return "\n\tNombre: " + nombre + "\tEdad: " + edad;
        }
    }
```

La clase tiene dos únicos atributos, declarados privados. Si ahora creamos algunos objetos, el gráfico que sigue muestra que cada uno tendrá acceso a su propio conjunto de atributos, que son una copia local de los que se declaran en la clase. Por lo tanto, si el objeto *b* invoca al método *setEdad()* y cambia su edad por otro valor, ese cambio sólo ocurrirá en el atributo *edad* contenido en *b*: nada ocurrirá con la edad de *a* o la de *c*:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
```



Ahora bien: si un miembro (atributo o método) se marca como estático (*static*), eso lo convierte en un *miembro compartido* por todas las instancias de la clase. Los miembros estáticos de una clase se cargan en memoria *antes* que se cree cualquier instancia de la clase, y todas las instancias que se creen luego, acceden exactamente al mismo elemento: no hay una copia de un atributo estático en cada instancia, sino que todas las instancias "ven" la misma variable (cada instancia posee un puntero a la única copia que hay de ese miembro). Esto puede parecer dificil de captar, o incluso puede parecer poco útil. Sin embargo, hay ciertas circunstancias en las que esta caracteristica es muy valiosa.

Por ejemplo, supongamos la misma clase *Persona* ya vista, pero ahora supongamos que se requiere poder llevar la cuenta de cuántas instancias de la clase se han ido creando (por caso, supongamos que se ha organizado un evento social en el cual sólo hay lugar para cierto número de Personas, y se quiere llevar la cuenta de cuantas Personas ya han entrado). Se podría llevar la cuenta con un contador implementado desde el método *main()*, y cada vez que se invoque a *new* para crear una *Persona*, incrementar el contador. Sin embargo, esto resultaría algo molesto para quien está haciendo el programa, pues debería llevar control permanente sobre ese contador en todos y cada uno de los métodos donde se pudiera invocar a *new*.

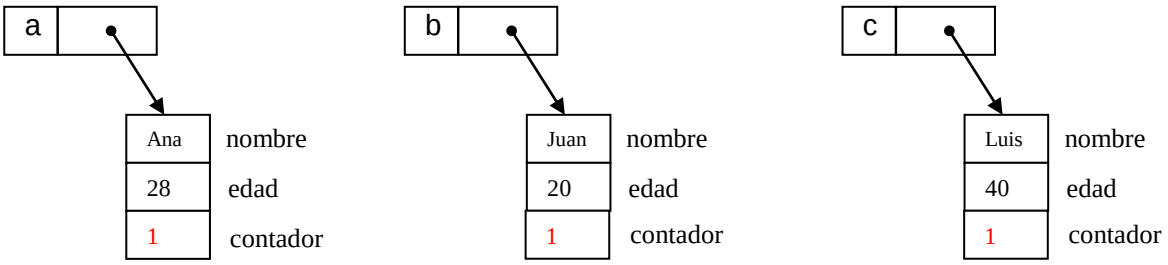
Una mejor solución es hacer que la propia clase *Persona* lleve la cuenta de la cantidad de instancias que van creando de ella. En principio, el *contador de instancias* debería ser ahora un atributo de la clase, y también en principio ese contador debería incrementarse dentro del constructor (o constructores) de la clase. Un metodo como *getInstanceCounter()* podría

usarse para retornar el valor de ese contador cada vez que se necesite saber la cantidad de instancias creadas:

```
class  Persona
{
   private String nombre;
   private int edad;

   // el contador de instancias de la clase
   private int contador = 0;


   public Persona(String nom, int ed)
   {
      nombre = nom;
      edad   = ed;
      contador++;
   }

   public int getInstanceCounter()
   {
      return contador;
   }

   public String getNombre()
   {
      return nombre;
   }

   public int getEdad()
   {
      return edad;
   }

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

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

   public String toString()
   {
      return "\n\tNombre: " + nombre + "\tEdad: " + edad;
   }
}
```

Puede verse fácilmente que esta solución es tristemente incorrecta: cada instancia que se cree tendrá su propio contador, que será puesto en cero al crear la instancia e incrementado a uno cuando se active el constructor. *En todas las instancias de la clase, habrá un contador distinto y el valor de cada uno de ellos será exactamente igual a uno...*

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
```

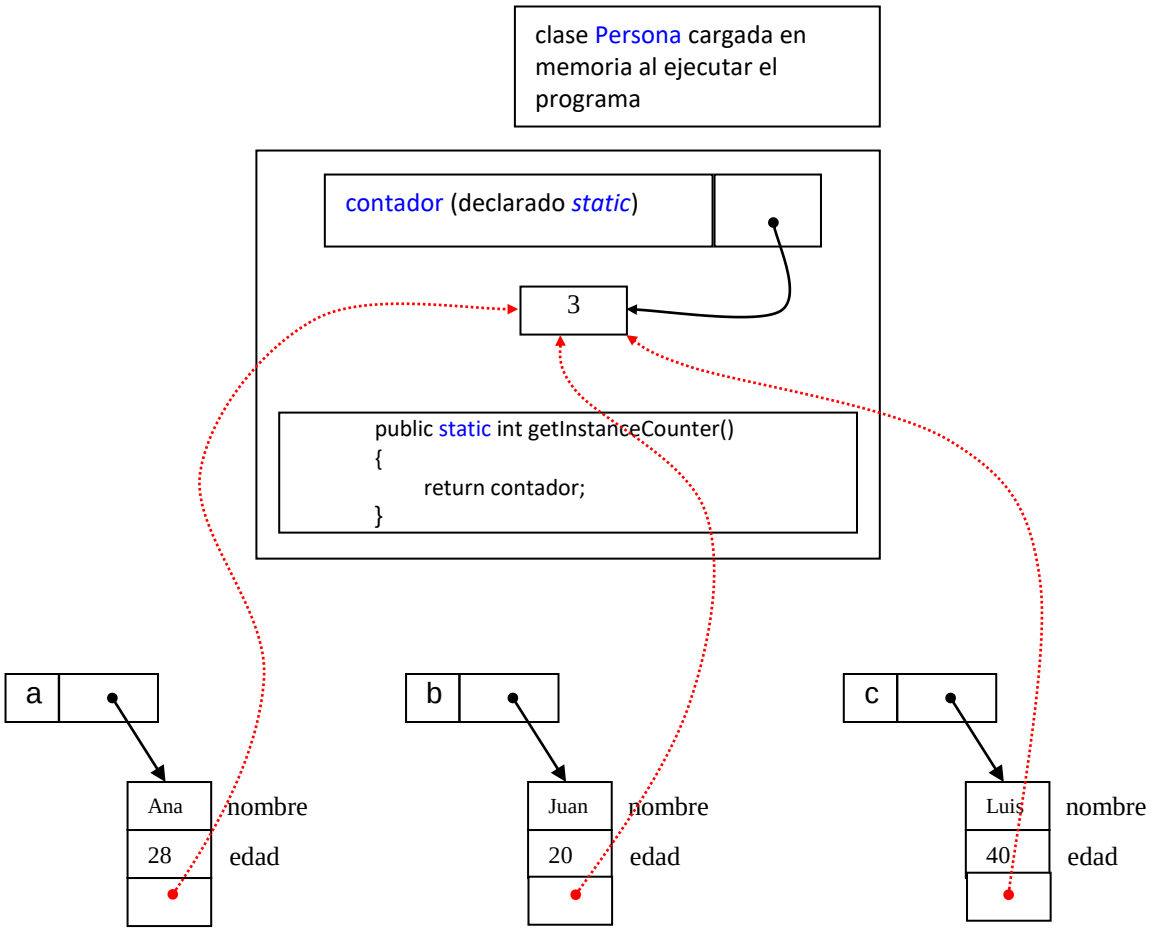


La solución sería hacer que el contador sea único: no que cada instancia tenga su propia copia, sino que exista una copia única de ese contador y que todas las instancias hagan referencia a esa única copia. Esto puede lograrse si el contador se declara static:

```
class  Persona
{
   private String nombre;
   private int edad;

   // el contador de instancias de la clase
   private static int contador = 0;


   public Persona(String nom, int ed)
   {
      nombre = nom;
      edad   = ed;
      contador++;
   }

   public static int getInstanceCounter()
   {
      return contador;
   }

   public String getNombre()
   {
      return nombre;
   }

   public int getEdad()
   {
      return edad;
   }

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

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

   public String toString()
   {
      return "\n\tNombre: " + nombre + "\tEdad: " + edad;
   }
}
```

Como dijimos, si un atributo se declara *static* existe una copia única del mismo, y cada instancia tendrá un puntero a esa copia única. Cuando el programa comienza a ejecutarse, la clase se carga en memoria *antes que se cree instancia alguna* y junto con ella se cargan los elementos que se hayan declarado *static* dentro de ella. En ese momento la única copia del atributo se inicializa en cero. Cuando luego se crean las instancias, cada una de ellas hace referencia a la unica copia del atributo mediante un puntero interno. Por lo tanto, luego de crear tres instancias el constructor se habrá invocado tres veces, y el único contador quedará valiendo tres:

```
Persona a, b, c;
int x = Persona.getInstanceCounter();
```



Como se ve, la única copia del atributo *contador* está en memoria junto con la clase, y no dentro de cada instancia. Por eso se suele decir que si un miembro (en este caso un atributo) es *estático* (declarado como *static*) entonces *le pertenece a la clase* y no a las instancias. Y también por eso, los *atributos estáticos* suelen designarse como *atributos de clase*. Si un atributo *no es static*, entonces cada instancia tendrá su propia copia: puede decirse que *son las instancias las dueñas de esos atributos* y eso hace que se los llame también *atributos de instancia*. En la clase *Persona*, los atributos *nombre* y *edad* son *atributos de instancia* (no son *static*) y el atributo *contador* es un *atributo de clase* (es *static*).

Notemos que también *un método puede declararse static*, y el efecto técnicamente es el mismo: si el método se declara *static* existe una única copia del mismo (que le pertenece a la clase) y se carga en memoria al cargarse la clase, del mismo modo que los atributos *static*. Si el método *NO* se declara *static*, entonces cada instancia posee su propia versión del método. Este hecho no parece tener mayor relevancia: al fin y al cabo que el método le pertenezca a la clase o a las instancias no cambia el proceso definido dentro de él. Sin embargo, hay una sutil (y útil) diferencia práctica: si un método es *static*, el mismo *existe en memoria aunque no se haya creado aún ninguna instancia*, y puede entonces ser invocado sin necesidad de crear instancias... Simplemente, se usa el nombre de la clase para invocar al método, en lugar de una referencia a una instancia. Notemos el método *getInstanceCounter()* de la clase *Persona*: está declarado *static* y por lo tanto podemos invocarlo de la siguiente forma:

```
int x = Persona.getInstanceCounter();
```

Esto tiene sentido: es la clase la que está llevando la cuenta de la cantidad de instancias creadas, y conceptualmente es válido que sea la propia clase la que llame al método... No obstante, tenga en cuenta que un *método static* también puede ser invocado sin problemas por una instancia de la clase, en forma normal. Lo siguiente es válido:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
int x = a.getInstanceCounter();
System.out.println("Cantidad de instancias creadas:" + x);
```

y se mostrará en consola un mensaje indicando que la cantidad de instancias creadas es 3. El mismo resultado se obtendria si en vez de la referencia a, se usara b o c para invocar a getInstanceCounter(), o si se invocara el método con el nombre de la clase en lugar de una referencia (puede ver esto ejecutando el modelo Static que acompaña a esta ficha).

Por cierto, si un método *NO es static* entones sólo existe dentro de una instancia, y por lo tanto no puede ser invocado mientras esa instancia no se cree. En otras palabras, sólo se puede invocar mediante una referencia a un objeto y no mediante el nombre de la clase. En la clase *Persona*, todos los métodos (salvo *getInstanceCounter()*) son *no – static*, y cualquier intento de invocarlos mediante el nombre de la clase producirá un error de compilación. Observe el siguiente ejemplo:

```
Persona a, b, c;
a = new Persona("Ana", 28);
b = new Persona("Juan", 20);
c = new Persona("Luis", 40);
int y = b.getEdad();   // correcto…
int z = Persona.getEdad();  // no compila: el método no es static...
```

También vale decir que si un método es *static*, entonces es un *método de clase* (como *getInstanceCounter()* en la clase *Persona*). Y si un método *no es static*, entonces es un *método de instancia* (como el resto de los métodos de la clase *Persona*).

Un análisis detallado del alcance del calificador *static* en una clase, muestra que si un método es *static*¸ entonces los atributos que ese método acceda dentro de la misma clase

también deben ser *static*, y si el método invoca a otros métodos de la misma clase esos métodos *deben* ser a su vez *static*... De otro modo, no compilará: el método estaría accediendo a elementos que no existen (o podrían no existir) al momento de la invocación. Por ejemplo, el método *getInstanceCounter()* no podría acceder al atributo *edad* o al atributo *nombre*, y tampoco podría invocar a ninguno de los otros métodos de la clase *Persona*:

```
// el método es de clase...
public static int getInstanceCounter()
{
   int x = edad + 1; // no compila: edad es atributo de instancia...
   String n = getNombre(); // no compila: getNombre() es método de instancia...
   return contador;
}
```

Notar por último, que el método *main()* (cuando está presente) se define como *static*: ***debe*** poder invocarse desde el sistema operativo o desde el entorno de programación que se esté usando, sin necesidad de crear una instancia de la clase que lo contiene (¡¡¡...!!!). Cuando se pide la ejecución de una aplicación, la máquina virtual Java debe saber cuál de todas las clases contiene al método *main()*, y esa máquina virtual se encarga de ejecutar el método. Como *main()* es estático, no puede invocar otros métodos de su misma clase que a su vez no sean estáticos, ni acceder a atributos de esa clase que no sean estáticos. Ese es el motivo por el cual hasta ahora, los programas realizados debían declarar *static* a los atributos y métodos de la clase *Principal* (o como fuese que se haya llamado la clase del *main()*) si esos atributos y métodos iban a ser accedidos desde *main()*...

2.] Declaración y uso de constantes: el calificador final en la declaración de atributos.

Supongamos que se desea implementar una clase para representar una Aerolínea (ver clase Aerolinea en el modelo *TSB-Static*). Si suponemos que la clase representará las oficinas comerciales de compañias aéreas que operan en Argentina, entonces podemos suponer que para todas esas compañías existe un ente regulador único, el mismo para todas. El nombre de ese ente podría almacenarse dentro de la clase en un atributo estático (*static*). De esta forma, si una instancia modifica el valor de ese atributo, la modificación valdrá para todas las demás.

Un atributo puede ser marcado también como *final*, lo cual en esencia lo convierte en una *constante*. Una vez asignado un valor inicial a una constante, el mismo no puede ser modificado durante el resto del programa (cualquier intento de hacerlo provocará un error de compilación). En nuestro ejemplo de la clase *Aerolínea*, el atributo *nombre* usado para almacenar el nombre de la compañia, se marcó *final* suponiendo que el mismo no será modificado de allí en adelante. Cualquier intento de modificar el valor de ese atributo luego de haberle dado su valor inicial, provoca un error de compilación.

Además, los calificadores *final* y *static* pueden combinarse: un atributo marcado con ambos calificadores, *será una constante (por ser final)*, y *también será compartida la misma copia de esa constante por todas las instancias de la clase (por ser static)*. El nombre del ente regulador en la clase *Aerolínea*, se marcó *static final*, y su valor se asignó inmediatamente al ser declarado. En casos como estos, en que el atributo es compartido y constante, se estila también declararlo *public* en lugar de *private*: al fin y al cabo, el *Principio de Ocultamiento*

busca evitar que se manipule de manera incorrecta el valor de un atributo de la clase, pero eso no podrá ocurrir si ese atributo es único para todas las instancias y marcado como constante... no hay nada que cada instancia pueda hacer con ese atributo salvo devolver su valor, y por lo tanto la práctica de declararlo *public* brinda flexibilidad para acceder a valores constantes de uso general que se almacenan en alguna clase, a manera de constantes globales para todo el programa.

Es común que una clase tenga varios atributos públicos constantes y estáticos para representar esa clase de valores. Un ejemplo son las constantes que se usan regularmente para ajustar el comportamiento de los métodos de la clase *JoptionPane* (que permite mostrar cuadros diálogo simples y de uso directo para visualizacion de mensajes y carga de datos): todas ellas son ejemplos de atributos públicos, constantes y estáticos. La clase *Color* de Java provee constantes públicas y estáticas para representar a los distintos colores que pueden usarse en una aplicación gráfica. Y la clase *Math* provee constantes públicas y estáticas para representar a los números *e* y *pi* (*Math.E* y *Math.PI* respectivamente). Y como el atributo es estático y público, puede ser accedido sin necesidad de crear instancias de esa clase, simplemente reemplazando la instancia por el nombre de la clase. Así, desde un método exterior a la clase *Aerolínea*, la constante *REGULADOR* declarada en ella puede accederse asi:

```
System.out.print(Aerolinea.REGULADOR);
```

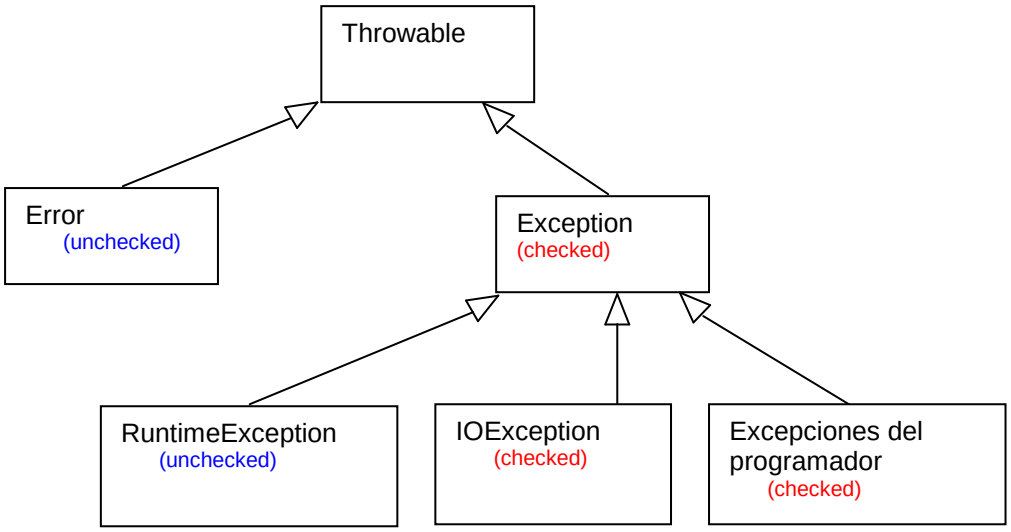
Hay ciertas reglas que conviene enumerar si se trabaja con atributos estáticos y/o finales para evitar errores de compilación (puede usar la clase *Aerolinea* del modelo *TSB-Static* para probar las distintas combinaciones que se sugieren a continuación):

- Un atributo declarado *static* (*pero sólo static y sin que haya combinación con final*) puede ser asignado en el momento de la declaración, o dentro de un constructor de la clase, o en cualquier método de la clase. Y una vez asignado, su valor puede ser cambiado desde cualquier otro método (a fin de cuentas, si el atributo sólo es *static* no es una constante: es sólo una variable de uso compartido y puede ser inicializada o cambiar su valor en cualquier parte).
- Un atributo declarado *final* (*pero sólo final y sin que haya combinación con static*) puede ser asignado en el momento de la declaración, o dentro de un constructor de la clase (pero no en ambos lugares: sólo en uno de ellos). Luego de realizada la asignación, el valor asignado no puede cambiarse (el compilador lanzará un error si se intenta)
- Si un atributo se declara *static final*, ***debe*** ser asignado al declararse (y no ya dentro de un constructor o en otro método de la clase). Si se intenta hacer la inicialización en otro lugar, se provocará un error de compilación. Esto se debe a que dicho atributo será cargado y mantenido inmodificable antes que cualquier instancia sea creada, y por lo tanto no se puede esperar a que alguna de ellas invoque a un constructor para asignarle un valor definitivo.

3.] Excepciones. Control de Excepciones en Java.

En Java, un error en tiempo de ejecución provocado por circunstancias anormales (errores matemáticos, índices fuera de rango, casting imposible de realizar, puntero nulo invocando a un método, etc.) se representa como un objeto. Así como la máquina virtual Java (JVM)

(por ejemplo) crea automáticamente objetos que representan a los eventos producidos por el usuario sobre los componentes de la interfaz visual de usuario, también genera automáticamente objetos que representan a un *error en tiempo de ejecución*, permitiendo que el programador pueda (si lo desea) intervenir en esa situación y eventualmente recuperarse de ella, incluso sin que la aplicación finalice de forma abrupta. Esos objetos que representan errores de tiempo de ejecución se llaman *objetos de excepción* o simplemente *excepciones* y son instancias de clases que forman parte de una jerarquía cuya clase base es la clase *Throwable* (o sea: "lanzable"):



Los errores de tiempo de ejecución representados por la clase *Error* son errores graves de hardware o de sistema frente a los cuales no se espera que el programador pueda hacer nada más que darse por notificado del hecho. Son manejados automáticamente por la JVM.

Los errores de tiempo de ejecución representados por la clase *Exception* son errores comunes de programación, algunos más graves que otros, que pueden requerir que el programador se vea obligado a escribir código de respuesta para esas excepciones, pues de lo contrario el programa no compilará. En ese sentido, las excepciones pueden clasificarse en *chequeadas (checked)* y en *no chequeadas (unchecked)*. El gráfico anterior muestra cuáles son de cada tipo.

Si un segmento de código puede llegar a lanzar una excepción del tipo *checked*, entonces el compilador obliga a que el programador tenga eso en cuenta, escribiendo código para tratar esa posible excepción, aunque luego en la práctica la misma no llegue a lanzarse.

En general, todas las clases de excepción que derivan de la clase *Exception* son *chequeadas* (salvo las que derivan a su vez de la clase *RuntimeException*). Por ejemplo, las excepciones de IO que pueden producirse al trabajar con entrada de datos desde distintos dispositivos, derivan todas de la clase *IOException*, que a su vez deriva desde *Exception*, y todas ellas son *chequeadas*: debemos escribir código para tratar las posibles excepciones de IO provocadas por nuestro código.

Si la excepción es *no chequeada*, el compilador no obliga a escribir código alguno de respuesta, y es decisión del programador el hacerlo o no. Todas las clases de excepción

derivadas desde *RuntimeException* son *no chequeadas*, y también la clase *Error*. Si el programador no incluye ningún código de tratamiento para estas clases de excepciones y alguna llega a producirse, simplemente el programa o el método finalizará mostrando un mensaje de error acorde a la excepción producida, pero no habrá problemas de compilación previa. Algunas de las clases de excepción más comunes derivadas de *RuntimeException* y de *IOException* son las siguientes:

`Derivadas más comunes de RuntimeException:`

- `NullPointerException`
- `ArithmeticException`
- `IndexOutBoundsException`
- `NegativeArraySizeException`

`Derivadas más comunes de IOException:`

- `EOFException`
- `FileNotFoundException`
- `InterruptedIOException`
- `ObjectStreamException`

Existen dos formas básicas de responder a una excepción (o sea: hay dos formas básicas de *código preventivo contra esa excepción*). Si la excepción es *chequeada*, el programador está *obligado a decidirse por alguna de las dos* que indicaremos. Si la excepción es *no chequeada*, el programador *puede optar por no usar ninguna y simplemente ignorarla, o puede tratarla con cualquiera de las dos formas* (podemos ver ejemplos aplicados en el modelo *TSB-Excepciones01* que acompaña a esta ficha).

Lo más sencillo, *es simplemente declarar la posibilidad de lanzamiento de la excepción en la cabecera del método que posea el bloque de código inseguro*:

```
public char leer () throws IOException
{
   System.out.print("Cargue un caracter:  ");
   char r = (char) System.in.read();  // puede lanzar una IOException
   return r;
}
```

Si el código inseguro pudiera lanzar más de un tipo de excepción, se pueden declarar todas en la cabecera:

```
public char leer () throws IOException, Exception
{
   System.out.print("Cargue un caracter: ");
   char r = (char) System.in.read();
   return r;
}
```

Lo anterior hace que el programador obligue a quien invoque al método *leer()* a tratar a su vez la excepción lanzada por él. En otras palabras, el programador de *leer()* se "saca el problema de encima" y lo transfiere a su "cliente".

La forma más natural (¡y posiblemente más comprometida!) de tratar una excepción (sobre todo si es chequeada), *consiste en capturarla con un bloque try - catch*:

```
public char leer ()
{
    char r = '';
    try
    {
       System.out.print("Cargue un caracter: ");
       r = (char) System.in.read();
    }
    catch(IOException ex)
    {
       JOptionPane.showMessageDialog(null, "Error de IO");
    }

    return r;
}
```

Si el bloque encerrado entre las llaves de *try* llega a disparar una excepción de tipo *IOException*, automáticamente la JVM creará un objeto de esa clase y buscará el bloque *catch* que tenga un parámetro *IOException*. Si lo encuentra, pasará el objeto creado a ese bloque, y se ejecutará el código encerrado entre las llaves de *catch*. Luego de ello, el programa seguirá ejecutando las instrucciones que se encuentran debajo del bloque *catch* (a menos que dentro del *catch* se haya lanzado un *System.exit()*). Como se ve, no necesariamente el programa termina si una excepción se produce y la misma es capturada. La decisión de terminarlo es del programador.

El objeto generado por la JVM para representar la excepción dispone de una serie de métodos que permiten que el programador tenga mayor conocimiento del error producido. Uno de esos métodos es *getMessage()* (heredado desde *Throwable*) que retorna un *String* con una descripción del error que provocó la excepción. El mismo puede usarse para mostrar un mensaje más claro:

```
public char leer ()
{
   char r = '';
   try
   {
     System.out.print("Cargue un caracter: ");
     char r = (char) System.in.read();
   }
   catch(IOException ex)
   {
     JOptionPane.showMessageDialog(null, "Error: " + ex.getMessage());
   }
   return r;
}
```

Si el bloque *try* puede llegar a lanzar excepciones de varias clases diferentes (aunque obviamente, sólo una en un momento dado), se pueden escribir varios bloques *catch*, cada uno con un parámetro que represente a la excepción esperada. Sólo debe tenerse en cuenta que la JVM recorre las definiciones *catch* por orden de escritura en el código, y al primero cuyo parámetro coincida con la excepción generada (incluidas las referencias polimórficas), lo aceptará. Por lo tanto, escriba los diversos *catch* comenzando por los menos polimorficos, y siga con los más polimorficos a continuación (de otro modo, no compilará...) El siguiente ejemplo muestra la forma correcta de hacerlo: El primer *catch* tiene un parámetro

IOException, y el segundo uno de tipo *Exception*. Por lo tanto, este último debe escribirse al final:

```
public char leer ()
{
   char r = '';

   try
   {
      System.out.print("Cargue un caracter: ");
      char r = (char) System.in.read();
   }

   catch(IOException ex)
   {
      JOptionPane.showMessageDialog(null, "Error: " + ex.getMessage());
   }

   /*
   * Si este bloque estuviera antes que el anterior, no dejaría pasar
   * ninguna excepción derivada de ella... y no compilaría...
   */
   catch(Exception ex)
   {
      JOptionPane.showMessageDialog(null, "Error:  " + ex.getMessage());
      System.exit(0);
   }

   return r;
}
```

Note que cuando una excepción se captura y procesa con un bloque *try – catch*, entonces el lanzamiento de una excepción en el bloque *try* provocará que la ejecución de la secuencia dentro del *try* se interrumpa, y se traslade el flujo de ejecución al *catch* que corresponda. Por el contrario, si el *try* se ejecuta en forma normal, entonces ningún bloque *catch* se ejecutará. Esto puede provocar que en determinadas circunstancias el programa no ejecute ciertas instrucciones críticas (por ejemplo, el cierre de un archivo o de una base de datos, o la liberación de recursos gráficos): si esas instrucciones están en el bloque *try* y se dispara una excepción antes de ejecutarlas, o están en un *catch* pero no se produce una excepción, entonces las instrucciones críticas no serán ejecutadas nunca.

Aunque veremos la forma de procesar archivos externos más adelante, los siguientes ejemplos sirven como modelos básicos: suponga que se desea almacenar algunos datos en un archivo (que manejaremos con un objeto de la clase *RandomAccessFile*). El siguiente bloque intenta abrir el archivo desde dentro de un bloque *try*, en ese mismo bloque intenta grabar los datos, y finalmente trata de cerrar el archivo. El bloque *catch()* solo pone un mensaje de error si hubo algún problema:

```
RandomAccessFile raf = null;
try
{
   raf = new RandomAccessFile("prueba.dat", "rw");
   raf.writeInt(120);
   raf.writeFloat(2000);
   raf.writeUTF("Juan Perez");
   raf.close();
}
```

```
catch(IOException e)
{
   System.out.println("Error:: " + e.getMessage());
}
```

Podemos ver que si durante la ejecución del *try* se produce una excepción, se interrumpirá el *try* y el archivo podría llegar a no cerrarse. Esto puede ser un gran problema si se supone que el programa continúa en ejecución (como en el ejemplo).

Una alternativa podría ser escribir la invocación a *close()* en el *catch()* (como se ve a continuación), pero eso tampoco sería correcto, ya que el cierre del archivo sólo se produciría en caso de lanzarse una excepción:

```
RandomAccessFile raf = null;
try
{
   raf = new RandomAccessFile("prueba.dat", "rw");
   raf.writeInt(120);
   raf.writeFloat(2000);
   raf.writeUTF("Juan Perez");
}

catch(IOException e)
{
   raf.close();
   System.out.println("Error:: " + e.getMessage());
}
```

Si bien el programador puede escribir su código fuente de forma que las operaciones críticas siempre se ejecuten (por ejemplo, en nuestro caso, podría escribir la invocación a *close()* tanto al final del bloque *try* como en el bloque *catch()*), lo cierto es que esto normalmente produce redundancia de código, así como bloques de código difíciles de interpretar y de mantener.

En casos así, se puede usar un bloque *finally* para complementar el *try – catch*. Un bloque *finally* se asemeja a un *catch()*, salvo por el hecho de que no se particulariza para un tipo específico de excepción, y por el hecho de que un bloque *finally* ***siempre se ejecuta***: si el bloque *try* se ejecuta sin problemas, entonces al finalizar el mismo se ejecutará el bloque *finally*, y si el *try* se interrumpe se activará un *catch()*, pero al terminar de ejecutarse ese *catch()* se ejecutará de todos modos el *finally*. El siguiente esquema muestra una forma correcta de solucionar nuestro problema del cierre del archivo:

```
RandomAccessFile raf = null;
try
{
   raf = new RandomAccessFile("prueba.dat", "rw");
   raf.writeInt(120);
   raf.writeFloat(2000);
   raf.writeUTF("Juan Perez");
}

catch(IOException e)
{
   System.out.println("Error:: " + e.getMessage());
}

finally
{
```

```
            if(raf != null)
            {
               raf.close();
            }
        }
```

Debe notar que el bloque *finally* será ejecutado prácticamente en toda combinación de situaciones que afecten al bloque *try – catch*. La ***única*** situación en la que *finally* no se ejecutará es aquella en la cual se ejecuta un *System.exit()* en el *try* o en el *catch()* (en casos así, la invocación a *exit()* provoca que el programa se interrumpa de inmediato, sin llegar a ejecutar ninguna otra instrucción). Considere las siguientes situaciones, que invitamos a que ponga a prueba (e investigue y pruebe otras muchas variantes según su curiosidad):

- Si el bloque try se ejecuta sin problemas, al terminar el try se ejecutará el finally.
- Si el bloque try provoca una excepción, saltará a un catch() y luego del mismo se ejecutará el finally.
- Si el bloque try contiene una instrucción return, primero se ejecutará el finally y luego el return del try.
- Si el bloque catch contiene una instrucción return, primero se ejecutará el finally y luego el return del catch.
- Si el bloque try contiene una invocación a System.exit(), no se ejecutará el finally.
- Si el bloque catch contiene una invocación a System.exit(), no se ejecutará el finally.

Para terminar esta sección, digamos brevemente que a partir de la versión 7 de Java se introdujo una nueva variante en cuanto a la posibilidad de escribir un bloque *try*, que se conoce como "*try con recursos*" (o "*try with resources*") Se trata de un bloque *try* en el cual se declaran en forma especial ciertos recursos. Esos recursos son objetos que deben ser cerrados al terminar el programa (como en el caso del ejemplo que hemos mostrado hasta aquí). El bloque *try with resources* garantiza que esos recursos serán efectivamente cerrados (en forma automática) al terminar de ejecutarse el bloque *try*. Técnicamente, los recursos declarados en un *try with resources* son objetos de *cualquier clase que implemente la interface java.lang.AutoCloseable* o su derivada *java.lang.Closeable* y el único método que estas declaran es justamente el método *close()*.

A modo de ejemplo de uso, volvamos sobre el ejemplo que hemos mostrado para abrir, grabar y cerrar un *RandomAccessFile*. En el último esquema hemos visto que toda la operación podría escribirse con un *try – catch – finally* sin problemas. Pero alternativamente, a partir de Java 7, podemos hacer en forma más compacta la misma operación mediante un *try with resources*:

```
try (RandomAccessFile raf = new RandomAccessFile("prueba.dat", "rw"))
{
   raf.writeInt(120);
   raf.writeFloat(2000);
   raf.writeUTF("Juan Perez");
}

catch(IOException e)
{
   System.out.println("Error:: " + e.getMessage());
}
```

En el ejemplo anterior mostramos el uso de un *try with resources*. El recurso asociado al bloque *try se declara entre paréntesis inmediatamente luego de la palabra try*. En nuestro caso se trata de un objeto de la clase *RandomAccessFile* (que implementa *AutoCloseable*, como todas las clases de gestión de archivos y flujos externos en Java). Tanto si el *try* se ejecuta sin problemas, como si el *try* provoca una excepción, el archivo representado por *raf* será cerrado automáticamente al terminar de ejecutarse el *try* o el *catch* (lo cual como primera ventaja, simplifica el bloque al no tener que escribir un *finally* explícito sólo para el cierre del archivo).

Note que un *try with resources* puede incluir bloques *catch* y *finally* en forma normal y cualquier bloque *catch* o *finally* será ejecutado ***después*** que el recurso asociado haya sido automáticamente cerrado.

4.] Excepciones definidas por el programador.

El programador puede crear sus propias clases de excepciones para representar errores de ejecución propios de su lógica y diseño. Para ello, lo común es declarar esas clases propias como derivadas de *Exception* (aunque podría crear clases de excepción derivadas de cualquier otra de la jerarquía de excepciones). No es necesario que esas clases sean complejas: suele bastar con simplemente declararlas para contar con una clase que describa una situación anormal en el sistema y luego confiar en el constructor de *Exception* y en los elementos heredados desde ella. Notar que al heredar desde *Exception*, la clase del programador será una excepción *chequeada* y el compilador obligará a tratarla.

Las clases de excepción que hemos creado en el modelo *TSB-Excepciones02* (provisto como modelo anexo en esta ficha) se llaman *InsercionHeterogeneaException* y *AccesoIncorrectoException*. La primera será lanzada cuando se intente insertar un objeto no compatible con los ya existentes en el objeto de clase *Listado* (o sea, si se intenta una *inserción heterogénea*), y la segunda si se intenta acceder a un elemento que no esté en el *Listado*:

```
public class InsercionHeterogeneaException extends Exception
{
      private String mensaje = "Error: Inserción Heterogénea";

      public InsercionHeterogeneaException()
      {
      }

      public InsercionHeterogeneaException(String mens)
      {
          mensaje = mens;
      }

      public String getMessage()
      {
         return mensaje;
      }
}


public class AccesoIncorrectoException extends Exception
{
```

```
    private String mensaje = "Error: Inserción Heterogénea";

        public AccesoIncorrectoException()
        {
        }

        public AccesoIncorrectoException(String mens)
        {
            mensaje = mens;
        }

        public String getMessage()
        {
           return mensaje;
        }
    }
```

Una vez definida una clase de excepción, lo siguiente es ubicar las situaciones en la lógica de negocio de la aplicación que merecerían que el programa lance una excepción de esa clase (y eventualmente se interrumpa si esa excepción no se captura).

El modelo *TSB-Excepciones02* incluye una clase llamada *Listado*, la cual contiene un arreglo de componentes polimórficos (objetos *Comparable*). La clase contiene un par de métodos *add()* y *get()*. El primero permite agregar un objeto al arreglo, pero controla que el objeto insertado sea de la misma clase que los que ya están en el arreglo. En caso de no ser así, crea un objeto de la clase *InsercionHeterogeneaException*, y usa la instrucción *throw* para lanzar ese objeto (como de hecho hace la máquina virtual con excepciones predefinidas del lenguaje en situaciones de error controladas por la plataforma...) El método obviamente se interrumpe si se activa la instrucción *throw*. Note que en ese caso, el método *add()* es un método inseguro: hay al menos una situación que podría hacer que el método se interrumpa lanzando una excepción, y como esa excepción es de una clase derivada de *Exception* y creada por el programador, es *chequeada*... por lo tanto, el método *add()* debe declarar en su cabecera que esa excepción puede llegar a producirse (de otro modo, no compilará...)

El otro método es *get()*, que retorna una referencia al objeto contenido en la casilla *i* del arreglo, siendo *i* un parámetro. El método simplemente controla que el valor de *i* sea válido de acuerdo al rango de índices del arreglo y a la cantidad de objetos que realmente tiene el arreglo en ese momento. Si el valor de *i* no fuera válido, se crea un objeto de la clase *AccesoIncorrectoException* y también se lanza con la instrucción *throw*, interrumpiendo el método. Otra vez, aquí hay que declarar la posibilidad de lanzar esa excepción en la cabecera de *get()*:

```
public final class Listado
{
   private Comparable v[];
   private int largo;
   private int libre;

   public Listado(int n)
   {
      if (n <= 0) { n = 10; }
      largo = n;
      libre = 0;
      v = new Comparable[largo];
   }
```

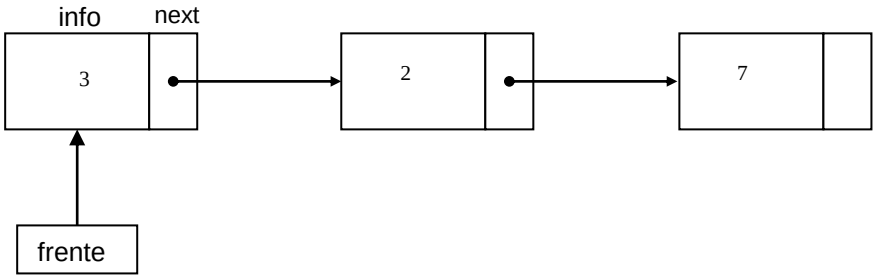
```
   public boolean add (Comparable x) throws InsercionHeterogeneaException
   {
      if (libre > 0 && x.getClass() != v[0].getClass())
      {
         throw   new InsercionHeterogeneaException("Objeto Incompatible");
      }

      boolean res = false;
      if (libre < largo)
      {
         v[libre] = x;
         libre++;
         res = true;
      }
      return res;
   }

   public Comparable get (int i) throws AccesoIncorrectoException
   {
      Comparable x = null;
      if (i >= libre)
      {
         throw new AccesoIncorrectoException("No existe objeto en posición " + i);
      }
      try
      {
         x = v[i];
      }
      catch(IndexOutOfBoundsException ex)
      {
          javax.swing.JOptionPane.showMessageDialog(null, "Indice negativo");
      }
      return x;
   }

}
```

Y finalmente, se debe incluir en el código fuente la correspondiente captura de estas excepciones, o la declaración en la cabecera de los métodos que invoquen a estos dos, de la posibilidad de encontrarlas al ejecutarlos: recordar que las excepciones del programador aquí mostradas son *chequeadas*. En el mismo modelo *TSB-Excepciones02* la clase *Principal* dispone de un método para cargar números. Este método toma el valor cargado por teclado e invoca al método *add()* de la clase *Listado* para añadir ese número al arreglo que contiene. La posible excepción de tipo *InsercionHeterogeneaException* lanzada por *add()*, es capturada con *try – catch()* (y algo similar puede hacerse desde otros procesos con la excepción *AccesoIncorrectoException* lanzada por *get()* en la clase *Listado*):

```
public static void cargar()
{
   try
   {
     System.out.print("Ingrese un numero para agregar al vector: ");
     int num = Consola.readInt();
     lis.add(num);
   }
   catch(InsercionHeterogeneaException ex)
   {
     JOptionPane.showMessageDialog(null, "Objeto incompatible");
   }
   System.out.println();
}
```

5.] Caso de Análisis: Desarrollo de una clase para manejar Listas Simples Genéricas.

Una *lista simplemente vinculada* (o *lista simple*) es una estructura lineal compuesta por una serie de elementos individuales llamados *nodos*. Cada nodo es un objeto formado, mínimamente, por dos campos: uno de ellos (que llamaremos *info*) contiene el valor o elemento de información que se almacena en el nodo; el otro (que llamaremos *next*) es un puntero con la dirección del nodo sucesor. El campo next del último nodo, normalmente vale *null*, indicando que el nodo no tiene sucesor. La dirección del primer nodo de la lista se almacena en una referencia separada, a la que podemos llamar *frente*. El gráfico siguiente muestra una lista simple en la que cada nodo almacena números enteros:



Básicamente, la implementación de una lista como la de la figura puede hacerse en base a dos clases sencillas: una para representar a los nodos y la otra para representar a la lista en sí misma, como conjunto de varios nodos ligados. En esta primera versión, el *info* de cada nodo será un simple atributo de tipo *int*:

```
public class TSBNode
{
      private int info;
      private TSBNode next;

      public TSBNode (int x, Node n)
      {
            info = x;
            next = n;
      }

      // resto de los métodos aquí…
}

public class TSBSimpleList
{
      private TSBNode frente;

      public Lista ()
      {
            frente = null;
      }

      public void addFirst (int x)
      {
            TSBNode p = new TSBNode(x, frente);
            frente = p;
      }

      // resto de los métodos aquí…
}
```

Esta versión preliminar sencilla de la clase *TSBSimpleList* incluye algún método para agregar un nodo en la lista. En el ejemplo, aparece el método *addFirst()* que añade un nodo con info igual a *x* (siendo *x* un parámetro), al principio de la lista.

Luego de definidas estas dos clases, se pretenderá crear instancias de la clase SimpleList y comenzar a usarlas para almacenar valores. Lo siguiente sería correcto:

```
public class Principal
{
      public static void main( String args[])
      {
            TSBSimpleList a = new TSBSimpleList();
            a.addFirst(7);
            a.addFirst(3);
            ...
      }
}
```

Sin embargo, si ahora se quiere crear una nueva instancia de la clase *TSBSimpleList* para guardar en ella cadenas de caracteres (o valores de cualquier tipo o clase que no sea *int*), se presentarán problemas de compilación:

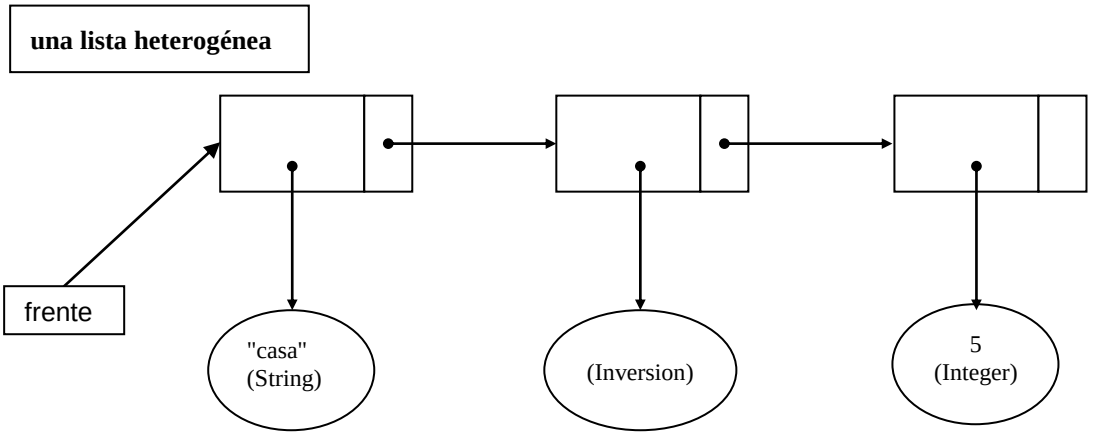
```
            TSBSimpleList b = new TSBSimpleList();
            b.addFirst("casa");  // no compila…
```

El problema es que tanto la clase *TSBNode* como la clase *TSBSimpleList* están declaradas de forma que el info de cada *nodo* se supone *int* y sólo *int*. En principio, si se quiere una lista de cadenas de caracteres deberíamos declarar otra clase *TSBNode* y otra clase *TSBSimpleList* iguales a las originales, pero cambiando el tipo del *info* en el *nodo* y el tipo del parámetro en el método *addFirst()*. Está claro que esta solución no es práctica... deberíamos tener tantas clases *TSBSimpleList* y *TSBNode* como tipos de datos diferentes querramos almacenar en nuestras listas.

Lo mejor sería definir una sóla clase *TSBNode* cuyo *info* sea *polimórfico*, y usar nodos de esa clase en una única clase *TSBSimpleList*. En ella, un método como *addFirst()* debería recibir un parámetro polimórfico e insertar eso en la lista. Entonces definimos: una lista cuyos nodos son capaces de aceptar *objetos de cualquier clase*, sin tener que declarar nuevamente la clase de la lista, se denomina *lista polimórfica* o también, *lista genérica*. Y en esa línea de conceptos, podemos pensar en dos tipos de listas genéricas, según las restricciones que el programador decida imponer a los datos que almacene en ellas:

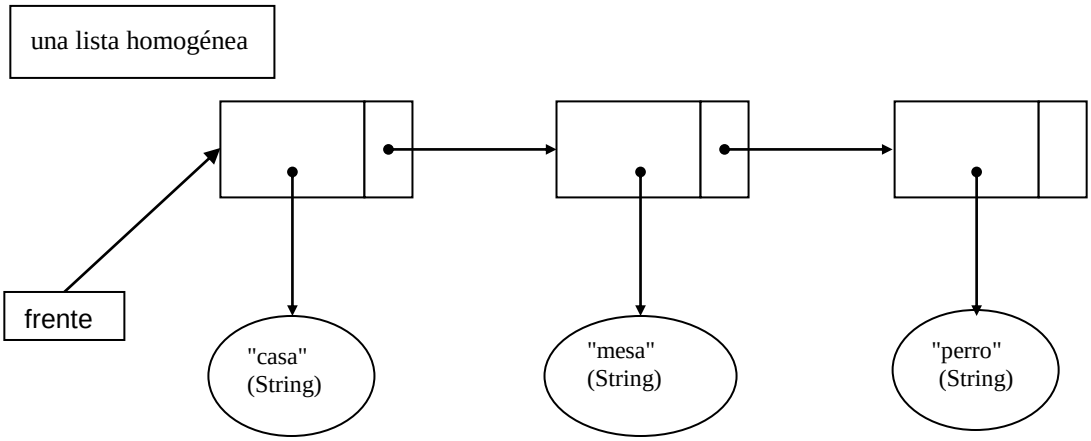
- *Listas Genéricas Heterogéneas*: una lista genérica se dice *heterogénea*, si admite que esa misma instancia de la lista pueda almacenar objetos de clases diferentes. Si el método *addFirst()* de la clase *TSBSimpleList* permitiese la inserción de objetos de clases diferentes, el siguiente esquema produciría una lista como la que se muestra debajo:

```
TSBSimpleList a = new TSBSimpleList();
a.addFirst( "casa" );
a.addFirst(new Inversion());
a.addFirst(new Integer(5));
```



- *Listas Genéricas Homogéneas*: una lista genérica se dice *homogénea*, si admite que esa misma instancia de la lista sólo pueda almacenar objetos de una misma y única clase. Si el método *addFirst()* de la clase *TSBSimpleList* no admite insertar objetos de clases diferentes en la misma lista, el siguiente segmento de código produciría una lista como la que se ve más abajo (note que la última llamada al método para insertar un objeto *Integer*, no produciría cambios en la lista: simplemente el método controlaría si el objeto que se quiere insertar es válido en esa lista, y lo ignoraría si no lo fuera):

```
TSBSimpleList a = new TSBSimpleList();
a.addFirst("casa");
a.addFirst("mesa");
a.addFirst("perro")
a.addFirst(new Integer(5));
```



Parece claro que el desarrollo de una clase que represente una lista genérica homogénea es más desafiante: requiere que el programador implemente los métodos de inserción de forma tal que el propio método controle que el objeto que se inserta sea de la misma clase que los que ya están en la lista. Y ese será nuestro objetivo: plantear la clase *TSBSimpleList* para que las listas creadas sean homogéneas (aunque iremos cumpliendo de a poco ese objetivo).

En principio, sólo es cuestión de redefinir la clase *TSBNode* para que el atributo *info* sea tan polimórfico como se pueda. Parece buena idea, entonces, que *info* sea una referencia a *Object* (ver modelo *TSB-SimpleList01*):

```
public class TSBNode
{
   private Object info;
   private TSBNode next;

   public TSBNode ( )
   {
   }

   public TSBNode (Object x, TSBNode p)
   {
      info = x;
      next = p;
   }

   public TSBNode getNext()
   {
      return next;
   }

   public void setNext(TSBNode p)
   {
      next = p;
   }

   public Object getInfo()
   {
      return info;
   }

   public void setInfo(Object p)
   {
      info = p;
   }

   public String toString()
   {
      return info.toString();
   }
}
```

Y en la clase *TSBSimpleList*, la idea es implementar sus métodos de forma que trabajen con parámetros polimórficos (referencias a *Object*) Por ejemplo, el método para insertar un nodo al frente de la lista es sencillo (aún no nos preocuparemos por cuidar la homogeneidad: sólo procuraremos que el nodo simplemente se inserte adelante) y otros métodos de uso básico también son directos. Una versión preliminar (e incompleta...) de la clase *TSBSimpleList* podría verse así (ver modelo *TSB-SimpleListo1*):

```
import java.util.NoSuchElementException;
public class TSBSimpleList
{
      private TSBNode frente;

      public SimpleList ()
      {
          frente = null;
      }
```

```
      public void addFirst(Object x)
      {
            if ( x != null )
            {
               TSBNode p = new TSBNode(x, frente);
               frente = p;
            }
      }

      public void clear( )
      {
         frente = null;
      }

      public Object getFirst()
      {
         if(frente == null) throw new NoSuchElementException("Error: vacía");
         return frente.getInfo();
      }

      public Object removeFirst()
      {
         if (frente == null) throw new NoSuchElementException("Error: vacía...");
         Object x = frente.getInfo();
         frente = frente.getNext();
         return x;
      }

      public String toString()
      {
            TSBNode p = frente;
            String res = "[ ";
            while( p != null )
            {
               if ( p.getNext() != null ) res = res + " - ";
               res = res + p.toString();
               p = p.getNext();
            }
            res = res + " ]";
            return res;
      }
}
```

Así planteada, la clase *TSBSimpleList* permite añadir objetos al principio de la lista, recuperar el primer objeto (con opción a removerlo o no de la lista), borrar todo el contenido de la lista, y finalmente retornar una cadena con el contenido de la lista completa. Todos los métodos planteados son bastante simples y directos.

Observe que el método *addFirst()* no verifica que la inserción sea homogénea: simplemente añade el objeto al inicio de la lista y sólo controla que la referencia tomada como parámetro no venga nula. Por lo tanto, es posible crear una lista con objetos de varias clases diferentes (como la lista *b* en el método *main()* que se muestra más abajo). También note que los métodos *getFirst()* y *removeFirst()* retornan una referencia a un *Object*, y para poder asignarla en una referencia a un *String* (o a cualquier otra clase), primero se debe hacer casting explícito hacia la clase *String*. La siguiente clase incluye un método *main()* simple para testear este desarrollo (ver modelo *TSB-SimpleList01*):

```
public class Principal
{
    public static void main(String args[])
    {
        // una lista de cadenas...
```

```
        TSBSimpleList a = new TSBSimpleList();
        a.addFirst("casa");
        a.addFirst("perro");
        a.addFirst("sol");
        System.out.println("Lista de cadenas: " + a.toString());

        // remueve y retorna la primera cadena
        String cad1 = (String) a.removeFirst();
        System.out.println("La primera cadena era: " + cad1);

        // retorna la primera cadena sin removerla
        String cad2 = (String) a.getFirst();
        System.out.println("La primera cadena es ahora: " + cad2);

        System.out.println("Lista de cadenas (luego del borrado): " + a.toString());

        // ahora una lista heterogénea...
        TSBSimpleList b = new TSBSimpleList();
        b.addFirst( new Cuenta (23, 2000) );
        b.addFirst( "juan" );
        b.addFirst( new Integer(5) );
        System.out.println( "\nLista heterogénea: " + b );
    }
}
```

En los desarrollos que hemos mostrado hasta aquí para implementar una lista genérica, hemos sugerido como primer intento que el *info* de cada nodo sea una referencia a *Object*, y en la clase *TSBSimpleList* hemos incluido métodos para insertar, borrar y convertir a *String* que realmente funcionaban mediante este mecanismo. El problema es que luego de creado un objeto de la clase *TSBSimpleList* querremos eventualmente usar métodos que trabajen con el *contenido* de cada nodo, por ejemplo para poder comparar sus valores (por caso, para mantener ordenada esa lista). Entonces, todo objeto almacenado en nodos de la lista debería proveer un método que permita compararlo con otro objeto que sea de su misma clase. Tal método podría ser el método *compareTo()* tal como lo impone la interface *Comparable*, según se vió en una ficha de estudio anterior.

Si queremos que los objetos almacenados en nuestras listas dispongan polimórficamente del método *compareTo()* y poder así establecer una relación de orden entre ellos, entonces nuestra clase *TSBNode*, deberíamos hacer que el *info* apunte a objetos de clases que hayan implementado *Comparable* y no a *Object*. Todo objeto que desee almacenarse en nuestras listas, deberá pertenecer a alguna clase que haya implementado antes la interface *Comparable*. De este modo, las listas podrán aprovechar la presencia de ese método para implementar otros métodos que puedan comparar objetos (y no sólo agregarlos o borrarlos en una posición fija)

En el proyecto *TSB-SimpleList02* mostramos que la clase *Cuenta* (abstracta y base de una jerarquía de clases que modelan cuentas bancarias), implementa la interface *Comparable*, y se desarrolla dentro de ella el método *compareTo()*. De esa clase heredan las clases *Corriente* e *Inversión*, y como a ambas el método definido en *Cuenta* les sirve, ambas usan ese mismo. La clase *Cliente* (también provista en el modelo) implementa a su vez *Comparable* y provee su propia versión del método *compareTo()*. Las dos clases pueden ser usadas para definir objetos que luego puedan ser incluidos dentro de una de nuestras listas. Veamos también que el info de la clase *TSBNode* se cambió para ser de tipo *Comparable*, y que cada método de la clase *TSBSimpleList* ahora opera con esa consigna.

En el modelo *TSB-SimpleList02* se ha agregado un método *contains()* a la clase *TSBSimpleList*. Este método verifica si el objeto tomado como parámetro está en la lista y retorna *true* en ese caso. El método recorre la lista y usa el método *compareTo()* (aunque también podría usar el método *equals()* heredado desde *Object* si ese método fuese adecuadamente redefinido en las clases derivadas) para hacer la comparación entre cada objeto de la lista y el objeto buscado:

```
public boolean contains (Comparable x)
{
    if (x == null) return false;

    TSBNode p = frente;
    while(p != null && x.compareTo( p.getInfo()) != 0)
    {
       p = p.getNext();
    }
    return (p != null);
}
```

En los modelos que siguen se incorporarán nuevos métodos a la clase *TSBSimpleList* que aprovechen la disponibilidad del método *compareTo()*. Un ejemplo de aplicación se muestra en el método *main()* de la clase *Principal* del modelo *TSB-SimpleList02*.

Por otra parte, la forma en que fueron desarrollados los métodos de inserción de la clase *TSBSimpleList* en los modelos *TSB-SimpleList01* y *TSB-SimpleList02* hacen que las listas representadas se comporten como *listas heterogéneas*: se puede insertar objetos de clases diferentes en la misma lista. Sin embargo, nos proponemos hacer que nuestras listas sean *homogéneas*: una misma lista sólo podrá contener objetos de la misma clase. Para esto, (ver modelo *TSB-SimpleList03*) los métodos de inserción deberían modificarse de forma que controlen que la clase del objeto que se inserta coincida con la clase de los que ya están en la lista. Está claro que sólo es necesario controlar la clase del que se quiere insertar contra la clase del primer objeto que ya esté en la lista (si la lista es homogénea, todos los que siguen son de la misma clase que el primero...) Un análisis más detallado sugiere los siguientes controles (suponemos que *x* es el parámetro que apunta al objeto que se quiere insertar):

- Si x es null, no hacer nada y salir sin hacer inserción ni tarea alguna.
- Si x no es null y la lista está vacía, inserte el objeto x en la lista sin problemas.
- Si x no es null y la lista no está vacía, inserte x en la lista pero sólo si la clase de x coincide con la clase del primer objeto ya está en la lista.

Se puede usar el método *getClass()* heredado desde *Object* para controlar la clase de *x* contra la clase del primero de la lista. De esta forma, la nueva versión del método *addFirst()* puede quedar así:

```
public void addFirst(Comparable x)
{
   if(x == null) return;
   if(frente != null && x.getClass() != frente.getInfo().getClass()) return;

   // si llegó acá, está todo ok... inserte tranquilo
   TSBNode p = new TSBNode(x, frente);
   frente = p;
}
```

El control de homogeneidad debe realizarse en varios de los métodos de la clase. Por lo pronto, en todos los métodos de inserción y en general en todos los métodos que tomen como parámetro un objeto para insertarlo, para removerlo o para buscarlo en la lista. Si la lista no fuera homogénea y se lanza una búsqueda de un objeto *x* en ella, el método *compareTo()* podría intentar comparar objetos que no sean de la misma clase y provocaría con ello una exception de la clase *ClassCastException*. Para evitar repetir las líneas de código que realizan el control, la clase *TSBSimpleList* incorpora un método privado *isHomogeneus()* que retorna *false* si *x* es *null* o no es homogénea con el primer elemento de la lista, y *true* en caso contrario:

```
private boolean isHomogeneus (Comparable x)
{
   if(x == null) return false;
   if(frente != null && x.getClass() != frente.getInfo().getClass()) return false;
   return true;
}
```

El modelo *TSB-SimpleList03* incorpora estos cambios y añade varios otros métodos para insertar al final (*addLast()*), insertar suponiendo que la lista está ordenada (*addInOrder()*), buscar y retornar un objeto tal como está en la lista (*search()*), remover el elemento del final (*removeLast()*), etc.

Por otra parte, se han incluido en la clase *TSBSimpleList* un grupo de métodos que por ahora (y hasta que se estudie la forma de implementar *iteradores* para hacer esta tarea en forma eficiente) podrán facilitar el trabajo de recorrer la lista cuando se esté programando un método de una clase que no sea la propia clase *TSBSimpleList* (por ejemplo, el método *main()*). Se trata de los métodos *add(i, x) – set(i, x) – remove(i) – get(i) – indexOf(x)* y otros similares, que básicamente consideran que los nodos de la lista están numerados con valores de cero en adelante, en forma semejante a los componentes de un arreglo. Así, el primer nodo de la lista es el nodo que está en la posición 0 de la lista (o también, el nodo con índice 0 en la lista), el segundo nodo tiene posición 1 (o índice 1), y así sucesivamente. Si bien esos índices realmente no existen en la lista ni están almacenados en ningún atributo de la clase *TSBNode*, la suposición de su existencia ayuda a simplificar los procesos de recorrido y manejo de una lista. El siguiente ejemplo ayuda a ilustrar el uso de los métodos citados:

```
TSBSimpleList lis1 = new TSBSimpleList();
lis1.add( 0, new Inversion (101, 2000, 2.1f) );
lis1.add( 1, new Inversion (212, 1000, 1.2f) );
lis1.add( 2, new Inversion (511, 3000, 3) );
System.out.println( "\nLista de Cuentas: " + lis1 );
if( lis1.isEmpty() ) System.out.println( "\nNo hay cuentas en la lista..." );
else
{
   float a = 0;
   int b = 0;
   for( int i = 0; i < lis1.size(); i++ )
   {
       Inversion x = ( Inversion ) lis1.get( i );
       a += x.getSaldo();
       b++;
   }
   float prom = 0;
   if( b != 0 ) prom = a / b;
   System.out.println( "\nSaldo promedio de las cuentas de Inversion: " + prom );
}
```

El método *add(i, x)* inserta el objeto *x* en la lista, en la posición cuyo índice es *i*. Los nodos que ya estaban en la lista a partir de esa posición, se mueven un lugar hacia la derecha (la posición de cada uno aumenta en 1). En el modelo, hemos invocado tres veces a ese método, para insertar en la lista *lis1* tres objetos de la clase *Inversion*. Las líneas:

```
lis1.add( 0, new Inversion (101, 2000, 2.1f) );
lis1.add( 1, new Inversion (212, 1000, 1.2f) );
lis1.add( 2, new Inversion (511, 3000, 3) );
```

agregan esos tres objetos en las posiciones 0, 1 y 2 respectivamente. Note que el método podrá agregar un nuevo nodo al final de la lista, pero no puede "saltearse" posiciones. Dada la lista anterior, la siguiente instrucción agregará un nodo en la posición 3:

```
lis1.add( 3, new Inversion (82, 1000, 2) );
```

Pero si la lista tiene ya cuatro nodos (el último entonces en la posición 3), la siguiente instrucción provocará una excepción por índice fuera de rango:

```
lis1.add( 8, new Inversion (12, 500, 1) );
```

En otras palabras: si la lista tiene *n* nodos, el método *add(i, x)* provocará una excepción si *i<0 || i >n*.

El método *get(i)* retorna el objeto que se encuentra en la posición *i* de la lista, pero sin removerlo (la cantidad de nodos de la lista no cambia). Igual que antes, el valor de *i* debe ser apropiado al rango de posiciones de la lista: si *i < 0 || i >= n* el método lanzará una excepción de índice fuera de rango. Considere que el método retorna una referencia de tipo *Comparable*, apuntando al objeto que se encontraba en el info del nodo accedido, y que debería usarse casting explícito para convertir esta referencia en otra del tipo que se requiera. Por ejemplo: en la lista *lis1* del modelo anterior almacenaron objetos de la clase Inversion, en las posiciones 0, 1 y 2 respectivamente. Si se invoca al método *get()* para obtener el objeto de la posición 1, la forma de hacerlo sería:

```
Inversion x = ( Inversion )  lis1.get( 1 );
```

De esta forma, la referencia polimórfica que *get()* retorna, se convierte en una referencia a un objeto de una clase específica, y a partir de ella el programador puede acceder a sus métodos particulares. En el modelo, este idea se está usando para recorrer la lista con un ciclo *for* (al estilo del recorrido de un vector...) y obtener el saldo promedio:

```
float a = 0;
int b = 0;
for( int i = 0; i < lis1.size(); i++ )
{
        Inversion x = ( Inversion ) lis1.get( i );
        a += x.getSaldo();
        b++;
}
float prom = 0;
if( b != 0 ) prom = a / b;
System.out.println( "\nSaldo promedio de las cuentas de Inversion: " + prom );
```

Note que la clase *TSBSimpleList* provee un método *size()* que retorna la cantidad de nodos que la lista tiene (o sea, el tamaño de la lista), y ese valor se usa para controlar el final del ciclo *for* de recorrido.

El método *set(i, x)* cambia el *info* del nodo que está en la posición *i*, por el objeto *x*. Note que a diferencia del método *add(i, x)*, el método *set(i, x)* no agrega un nuevo nodo a la lista (el tamaño de la lista no cambia): sólo reemplaza el objeto en la posición *i* por el objeto *x* que viene como parámetro. Por su parte, el método *remove(i)* retorna el objeto que estaba en la posición *i*, pero lo elimina de la lista (a diferencia de *get(i)* que lo retorna sin removerlo). Y el método *indexOf(x)* busca el objeto *x* en la lista, y en caso de encontrarlo retorna el índice del primer nodo que lo contenga (comenzando desde el frente de la lista). Si el objeto *x* no está en la ista, *indexOf()* retorna –1. Se deja para el estudiante el análisis del código fuente de estos métodos, y otros que aparecen en la clase.

Los métodos que permiten recorrer la lista "como si fuera un arreglo" (*get(i), set(i, x)*, *add(i, x)*, *remove(i)*, etc.) son muy útiles para facilitar el trabajo de recorrido de la lista cuando se programa ese recorrido desde fuera de la clase *TSBSimpleList*, pero debe entender el estudiante que esa sencillez tiene un precio: todos estos métodos recorren la lista (en el peor caso, hasta el final…) para llegar al nodo número *i* que queremos acceder. Eso implica que un ciclo for que recorra la lista completa (por ejemplo, para calcular el saldo promedio como vimos en el ejemplo anterior) será muy ineficiente, ya que en cada vuelta de ese ciclo comenzará el recorrido nuevamente, para acceder al siguiente nodo. En esta ficha de estudio no nos preocuparemos por ese hecho y aceptaremos la sencillez que obtenemos a cambio, pero es bueno que el alumno tome conciencia del problema. Solo nos limitaremos a decir que el problema puede resolverse correctamente implementando y usando un objeto *iterador*, el cual permite recorrer la lista sin tener que volver a comenzar el recorrido cada vez que queremos otro nodo. Pero la implementación de un iterador (por ahora) está fuera de nuestro objetivo. Volveremos sobre este tema en fichas posteriores.

En general, la denominación de los métodos se ha hecho de forma que sean análogos a los métodos que incluye la clase *LinkedList* del paquete *java.util* y que ya viene predefinida en el lenguaje. En ese sentido, Java incluye versiones predefinidas de varias clases que representan estructuras de datos básicas y la clase *LinkedList* representa una *lista genérica heterogénea*, aunque *doblemente vinculada* (cada nodo tiene un puntero al siguiente y al anterior) y con dos punteros de acceso: uno al primer nodo de la lista y otro al último. Oportunamente estudiaremos las clases del paquete *java.util* y su estructura y convenciones de trabajo.

Si bien se ha intentado seguir la clase *LinkedList* como modelo para el planteo de nuestra clase *TSBSimpleList*, note que algunos métodos que nuestra clase implementa no están en *LinkedList*: el método *addInOrder()* de nuestra clase *TSBSimpleList* agrega un objeto de forma que la lista se mantenga ordenada si es que ya estaba ordenada previamente. Pero ese método (al igual que nuestro método *search()*) no está presente en *LinkedList*: simplemente, esa clase está diseñada con otras convenciones y principios que son los que guiaron el diseño de todo el paquete *java.util*. La clase *LinkedList* está pensada como una colección heterogénea de objetos que brinda métodos para insertar al principio, atrás o en alguna posición intermedia, pero nunca teniendo en cuenta los contenidos de los objetos. En nuestra propuesta de *TSBSimpleList* hemos preferido incluir ese método *addInOrder()* no convencional, debido a que se trata de un algoritmo interesante y tratado históricamente en materias similares a la nuestra. En la medida que sea posible, trataremos de

mantenernos "a la par" de las convenciones y clases del paquete *java.util*, pero cuando las circunstancias nos sugieran incluir elementos que rompan esa paridad, lo haremos en búsqueda de elementos que aporten novedades, curiosidades o simplificaciones. No es objetivo de esta cátedra replicar el paquete *java.util* de Java (aunque podría hacerse) pero no hay problema en tomarlo como modelo y entender que quienes lo programaron aplicaron ideas y conocimientos que surgen de materias como la nuestra.