Capítulo 3

- **Excepciones**
- **Colecciones de Objetos**
- **Serialización y Flujos de I/O**
- **Manejo de Archivos**

Excepciones

- Una **excepción** es cualquier condición anormal que interrumpe el flujo normal de ejecución de un programa.
- Los errores ocurridos en tiempo de ejecución de un programa constituyen claros ejemplos de excepciones (más adelante veremos **otros** ejemplos de excepciones)

Ejemplos: Consideremos los siguientes errores en tiempo de ejecución:

- El programa pretende hacer una división entre cero.
- Se quiere invocar un método sobre un objeto que está referenciando a `null`.
- Se accede a una celda que está más allá del tamaño de un arreglo.

- En todos los ejemplos anteriores, el problema no es detectado en la compilación, sino durante la ejecución. Esto ocasiona que el programa termine de ejecutarse abruptamente, en forma prematura.

Excepciones

- Excepciones como las anteriores constituyen un tipo especial de Excepción conocido como **Runtime Exception**. Las excepciones de este tipo son todos aquellos errores en tiempo de ejecución que podrían evitarse si se programa **cuidadosamente**.

- Java provee un mecanismo conocido como **Exception Handling** (manejo de excepciones) que permite detectar una excepción, y si es posible, corregir el problema y permitir que el programa continúe ejecutándose normalmente.

- Por ahora vamos a dedicarnos solamente al estudio de las **Runtime Exceptions**. Más adelante veremos que la forma de manejar todas las excepciones en Java es idéntica a la forma de manejar éstas.

Excepciones

Runtime Exceptions

- Consideremos el siguiente ejemplo:

```
public class PruebaExcepciones
{
    public static void main (String [] args)
    {
        String arre[] = {"Uno", "Dos", "Tres"};

        for (int i = 0; i < 20; i++)
            System.out.println(arre[i]);
    }
}
```

- **¿ Hay errores de *compilación* en este programa ?**
- **¿ El programa se comportará correctamente ?**

Excepciones

Runtime Exceptions (continuación)

- La salida a pantalla desplegada por el programa anterior será:

```
Uno
Dos
Tres
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 3 at PruebaExcepciones.main
```

- Cuando ocurre un error de ejecución como el anterior, se dice que el programa **lanzó** una Runtime Exception. Existen muchas Runtime Exceptions predefinidas por Java. Por ejemplo, las siguientes:
  - **`ArithmeticException`**
  - **`NullPointerException`**
  - **`ArrayIndexOutOfBoundsException`**
  - **`StringIndexOutOfBoundsException`**

Excepciones

Runtime Exceptions (continuación)

- Generalmente, **no** es posible predecir si nuestro programa lanzará una Runtime Exception o no. En general, si uno programa **cuidadosamente**, tales excepciones nunca son lanzadas.
- De todas maneras, estudiaremos cómo podemos hacer para detectar estas excepciones y (si es posible) corregir los problemas asociados con ellas. Para ello, analizaremos el mecanismo de **Exception Handling**.

La instrucción try – catch

- Es una instrucción especial de Java que permite manejar excepciones (esto es, detectarlas y tratar de corregir el error asociado). Su sintaxis básica es:

```
try
 { ... // código "peligroso" }
catch (TipoExcepcion nombre)
 { ... // tratamiento del error }
```

Excepciones

La instrucción try – catch (continuación)

- Para manejar una Excepción en particular se debe encerrar el código “peligroso” (el que puede lanzar la excepción) dentro de un bloque **try**. El mismo debe contener a la instrucción “sospechosa”, pero también puede contener otras instrucciones.

- A continuación, se debe incluir un bloque **catch** donde:
  - Entre paréntesis se especifica el tipo de excepción a tratar.
  - Entre llaves se escribe el código a ejecutarse que permita corregir el error, o en su defecto, terminar el programa en forma “elegante” .

- Modifiquemos la clase **PruebaExcepciones** presentada antes a efectos de incorporarle una instrucción **try – catch**.

Excepciones

La instrucción try – catch (continuación)

Ejemplo:

```
public class PruebaExcepciones
{
    public static void main (String [] args)
    {
        String arre[] = {"Uno", "Dos", "Tres"};
        for (int i = 0; i < 20; i++)
        {
            try
            {  System.out.println(arre[i]); }
            catch (ArrayIndexOutOfBoundsException e)
            {  i = 20; }
        }
    }
}
```

Excepciones

La instrucción try – catch (continuación)

¿ Cómo funciona la instrucción try – catch ?

1º) Comienza a ejecutarse el código que está dentro del bloque **try**.

2º) Si la excepción especificada en los paréntesis del **catch** es lanzada:

a) Se detiene la ejecución de las instrucciones del bloque **try**.
b) Luego se ejecutan las instrucciones del bloque **catch**.

3º) Si dicha excepción **no** es lanzada, el código del **try** termina de ejecutarse normalmente y **no** se ejecutan las instrucciones del **catch**.

Observación:

Una instrucción **try – catch** maneja **solamente** la excepción indicada entre los paréntesis del **catch**. Si **otra** excepción es lanzada durante la ejecución del bloque **try**, la misma **no** será atrapada.

Excepciones

La instrucción try – catch (continuación)

- En el ejemplo presentado, la excepción especificada fue lanzada, pero enseguida fue **atrapada** y procesada dentro del bloque **catch**. Ahora la salida desplegada por el programa será simplemente la siguiente:

```
Uno
Dos
Tres
```

- Incluir una instrucción **try – catch** en un programa **no** significa que necesariamente la excepción va a ser lanzada. Quiere decir que, **en caso de lanzarse**, el programa estará preparado para manejarla.
- Si en el ejemplo anterior mantenemos el bloque **try – catch**, pero modificamos el encabezado del **`for`** para no excedernos del rango, la excepción especificada **no** será lanzada.

```
for (int i = 0; i < arre.length; i++)
```

Excepciones

La instrucción try – catch (continuación)

- Para tratar **más** de una excepción en una instrucción **try – catch**, podemos incluir más de un bloque **catch**, uno por cada excepción que queramos tratar

Ejemplo:

```
try
{    ...
     // código "peligroso"
}
catch (ArrayIndexOutOfBoundsException e)
{    ...
     // trato esta excepción, en caso de lanzarse
}
catch (NullPointerException e)
{    ...
     // trato esta otra excepción, en caso de lanzarse
}
```

Excepciones

La instrucción try – catch (continuación)

- También es posible atrapar **varias** excepciones diferentes dentro de un mismo bloque **catch**, separadas mediante |

Ejemplo:

```
try
{    ...
     // código "peligroso"
}
catch (ArrayIndexOutOfBoundsException |
       NullPointerException e)
{    ...
     // atrapa cualquiera de las dos excepciones
     // especificadas, en caso de lanzarse
}
```

Excepciones

La instrucción try – catch (continuación)

- En ocasiones, no tenemos interés en preocuparnos por ninguna excepción en particular. Simplemente queremos que, ante **cualquier** excepción, el programa tome una acción genérica.

  Ejemplo:

```
try
{    ...
     // código "peligroso"
}
catch (Exception e)
{    ...
     // trato esta excepción, sin importar cual sea
}
```

Este bloque catch atrapa **cualquier** Excepción porque toda excepción **hereda** de la clase base **Exception** en Java.

Excepciones

Runtime Exceptions (Moraleja)

- Hemos visto que para manejar Excepciones en Java contamos con la instrucción **try – catch**. En los ejemplos presentados las hemos usado solamente para manejar **Runtime Exceptions**, pero la misma sirve para manejar **cualquier** tipo de Excepción (enseguida veremos otras).

- De todas maneras, el compilador de Java **NO** nos obliga a manejar las **Runtime Exceptions**, dado que las mismas son en general **impredecibles**, pero en caso de que queramos hacerlo, contamos con la instrucción **try – catch**.

- Si programamos con cuidado y depuramos apropiadamente nuestro código, tales excepciones **no** ocurrirán, y no perjudicaremos la claridad del código al incluir muchos bloques **try – catch**.

Excepciones

Throws versus try – catch

- A veces no deseamos manejar una Excepción dentro del mismo método en el cual es lanzada (quizás porque queremos centralizar el manejo de errores de todo el programa en otra capa).

- Por lo tanto, Java nos permite “redireccionar” una Excepción hacia el método superior. Es decir, el que invoca al método dentro del cual la Excepción es lanzada.

- Esto puede hacerse usando la palabra reservada **throws** en lugar de atrapar la Excepción con una instrucción **try – catch**. Si optamos por esta alternativa, será el método superior quien deberá hacerse cargo de la Excepción.

- No hay una regla general acerca de dónde conviene manejar las excepciones. Eso debe definirse concretamente para cada diseño. Lo más frecuente es **lanzarlas** en la Fachada y **atraparlas** en la capa gráfica.

Excepciones

Throws versus try – catch (continuación)

Ejemplo: Mostramos dos alternativas para un mismo método

```
public void miMetodo(...)
{    ...
     try
     {  instrucción X;
        instrucción Y;
     }
     catch (TipoException nombre)
     { ... }
}

public void miMetodo(...) throws TipoException
{    ...
     instrucción X;
     instrucción Y;
}
```

Excepciones

Manejando otras Excepciones

- Hemos visto que las **Runtime Exceptions** son todos aquellos errores en tiempo de ejecución que podrían evitarse si se programa **cuidadosamente**.
- No obstante, existen **otras** situaciones que podrían provocar la finalización prematura de un programa y que **no** dependen del programador, sino de factores **externos** al programa en tiempo de ejecución.

  Ejemplos:

  - El programa pretende abrir un archivo que **no** existe en el disco.
  - El programa pretende enviar información hacia otra máquina en un momento en el cual la red está **caída**.
  - El programa pretende imprimir un texto en una impresora, pero la misma **no** está configurada.

Excepciones

Manejando otras Excepciones (continuación)

- Los ejemplos anteriores también son considerados **excepciones**, dado que constituyen situaciones anormales que pueden provocar la finalización abrupta de la ejecución del programa.

- Sin embargo, en Java se considera que tales excepciones son **predecibles** y por lo tanto el compilador **nos obliga a manejarlas**, lo cual hacemos con la instrucción **try – catch** tal y como la conocemos.

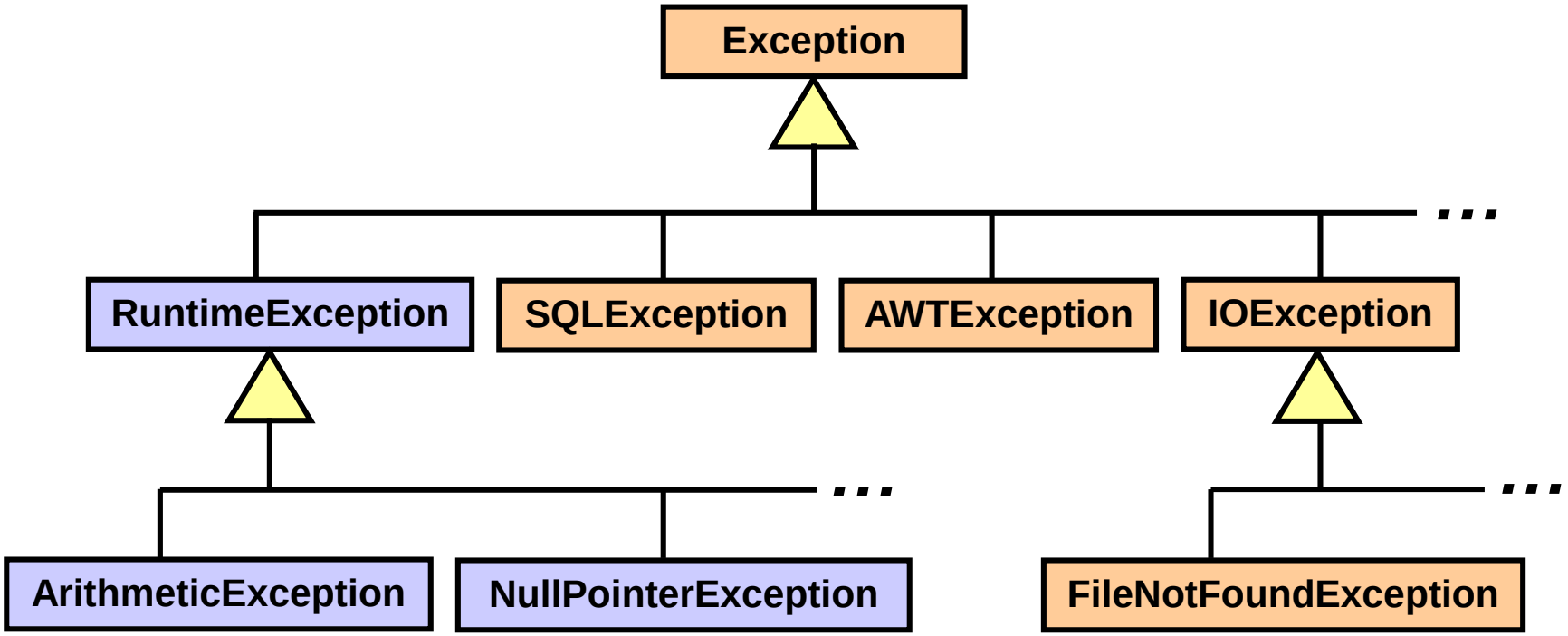
- En resumen, podemos decir que existen dos grandes tipos de Excepciones:
  - Las **Runtime Exceptions**
  - Todas las demás

- En Java, las Excepciones ***también son Objetos***. Las clases que las definen están organizadas en una enorme jerarquía de herencia, como podemos ver en la próxima diapositiva.

Excepciones

Manejando otras Excepciones (continuación)

- Mostramos **parte** de la jerarquía de Excepciones existentes en Java



Referencia: API - `java.lang.Exception`

Excepciones

Definiendo nuestras propias Excepciones

- En nuestros programas podemos definirnos **nuestras propias** excepciones con el fin de manejar situaciones que (en el contexto de nuestros programas) constituyan errores que deban ser tratados.
- Al hacerlo, muchas veces podemos **eliminar** precondiciones o chequeos previos que impliquen trabajo adicional para una operación. La idea básica es: ***"Trato de resolverlo y si no puedo, lanzo una excepción"***

Ejemplo: Supóngase un programa para el manejo de cuentas bancarias

```
public class MontoInvalidoException extends Exception
{   private String mensaje;

    public MontoInvalidoException (String mensaje)
    {   this.mensaje = mensaje;  }

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

Excepciones

Definiendo nuestras propias Excepciones (continuación)

```
public class FachadaCuentaBancaria
{    private double saldo;
     ...
     public void depositar (double monto)
                              throws MontoInvalidoException
     {  if (monto < 0)
        {
            String msg = "El monto debe ser positivo";
            throw new MontoInvalidoException(msg);
            // creo y lanzo una nueva Excepción
        }
        saldo = saldo + monto;
     }
}
```

Observación: Cualquiera que invoque al método **depositar**, deberá atrapar nuestra **MontoInvalidoException**

Colecciones de Objetos

- Hasta el momento, la única estructura que conocemos en Java para implementar una colección de objetos es el **arreglo** (estructura **acotada**)
- Para implementar colecciones **no** acotadas, Java provee un paquete predefinido llamado **`java.util`**, el cual contiene interfaces y clases **predefinidas** que implementan colecciones no acotadas de objetos.
- Esto significa que ***no necesitamos programar nuestras propias estructuras de datos*** cuando queramos implementar colecciones no acotadas. En lugar de eso usamos las colecciones predefinidas por Java.
- Hay **interfaces** en el paquete **`java.util`** que definen los métodos a ser implementados por las distintas colecciones según sus características y **clases** que los implementan usando diferentes estructuras de datos internas.
- Las colecciones del paquete **`java.util`** siempre son **polimórficas**, dado que permiten almacenar ***cualquier tipo de Objeto***. Al declararlas es posible indicar el tipo concreto de los objetos a guardar en ellas mediante **< >**.

Colecciones de Objetos

Algunas interfaces predefinidas del paquete `java.util`

- **Interface `List<TipoElem>`**

  Esta interface define métodos apropiados para cualquier colección (no acotada) que almacena sus valores en una estructura de datos ***lineal***.

- **Interface `Map<TipoKey,TipoElem>`**

  Esta interface define métodos apropiados para cualquier colección (no acotada) cuyos elementos poseen una ***clave***.

- **Interface `SortedMap<TipoKey,TipoElem>`**

  Esta interface (derivada de **`Map`**) define métodos apropiados para cualquier colección (no acotada y ordenada) cuyos elementos poseen una clave (y además los elementos se ***ordenan*** según su clave).

- **Interface `Set<TipoElem>`**

  Esta interface define métodos apropiados para cualquier colección (no acotada) que represente un ***conjunto***.

Colecciones de Objetos

Algunos métodos de la interface `List<tipoElem>`

- **`boolean add (TipoElem o)`**
  // inserta el objeto recibido al final de la colección. Siempre retorna true.

- **`TipoElem get (int index)`**
  // retorna el objeto almacenado en la posición dada por el índice. Si no
  // existe dicha posición en la colección, el método lanza una **excepción**.

- **`TipoElem remove (int index)`**
  // elimina el objeto almacenado en la posición dada por el índice y baja un
  // lugar a todos los objetos posteriores en la colección. Si no existe dicha
  // posición en la secuencia, el método lanza una **excepción**.

- **`int size()`**
  // retorna la cantidad de objetos almacenados en la colección (las
  // posiciones dentro de la secuencia se numeran a partir de cero).

Colecciones de Objetos

Algunos métodos de la interface `Map<TipoKey,TipoElem>`

- **`TipoElem put (TipoKey key, TipoElem value)`**
  // inserta en la colección al objeto **`value`** asignándole la clave **`key`**. Si ya
  // había otro objeto con esa clave, el mismo es retornado y suplantado.

- **`TipoElem get (TipoeKey key)`**
  // retorna el objeto identificado por la clave recibida. Si no había ninguno en
  // la colección, o el que había era nulo, el método retorna **`null`**.

- **`boolean containsKey (TipoKey key)`**
  // determina si hay algún objeto en la colección identificado por la clave
  // recibida.

- **`TipoElem remove (TipoKey key)`**
  // elimina de la colección al objeto identificado por la clave recibida y además
  // lo devuelve. Si no había un objeto con esa clave, el método no hace nada.

Colecciones de Objetos

Algunas clases predefinidas del paquete `java.util`

- **Clase `ArrayList<TipoElem>`**
  - Esta clase implementa una colección que implementa la interface `List`. Internamente usa una estructura de ***arreglo dinámico redimensionable*** para almacenar los elementos.
  - Las operaciones de ***acceso*** a elementos se realizan en **O(1) peor caso** (son muy eficientes, dado que acceden directamente a la posición dada).
  - Sin embargo, las operaciones de ***inserción*** y ***borrado*** potencialmente son ineficientes (se realizan en **O(n) peor caso**) dado que en ocasiones requieren ***redimensionar*** el arreglo.

Observación: Al usar **`ArrayList` NO** usamos la sintaxis usual de arreglos porque los mismos son usados **internamente** por la clase. Nos manejamos simplemente con los métodos provistos por la clase (nada de corchetes `[ ]`).

Colecciones de Objetos

Algunas clases predefinidas del paquete `java.util` (cont.)

- **Clase `LinkedList<TipoElem>`**
  - Esta clase implementa una colección que implementa la interface `List`. Internamente utiliza una estructura de ***lista doblemente encadenada con puntero al principio y al final*** para almacenar los elementos.
  - Las operaciones principales (***búsqueda***, ***inserción***, ***borrado***) se realizan en **O(n) peor caso** (son medianamente eficientes, dado que requieren recorrer nodo a nodo los elementos hasta llegar a la posición deseada). En una lista no existe el concepto de ***redimensionamiento***.

Observación: Al utilizar `LinkedList` **NO** usamos ninguna sintaxis para listas encadenadas porque las mismas son usadas **internamente** por la clase. Nos manejamos simplemente con los métodos provistos por la clase. De hecho, **NO** existe en Java una notación para listas similar a la de C++ (nada de flechas ni de asteriscos) dado que **no** existe notación de punteros en Java.

Colecciones de Objetos

Algunas clases predefinidas del paquete `java.util` (cont.)

- **Clase `Hashtable<TipoKey,TipoElem>`**
  - Esta clase implementa una colección que implementa la interface **`Map`**. Internamente utiliza una estructura de ***hash*** para almacenar los elementos. La cantidad de cubetas se define en tiempo de ejecución.
  - Las operaciones principales (***búsqueda***, ***inserción***, ***borrado***) se realizan en **O(1) caso promedio** (son muy eficientes).
  - Las **claves** utilizadas deben ser objetos que implementen los métodos **`hashCode()`** y **`equals()`** apropiadamente. Estos métodos están originalmente definidos en la clase **`Object`** y existen clases predefinidas que los sobre-escriben (Por ejemplo, la clase **`String`**).

Observación: Al utilizar **`Hashtable`** debemos manejarnos simplemente con los métodos provistos por la clase, de igual modo que en toda colección predefinida

Colecciones de Objetos

Algunas clases predefinidas del paquete `java.util` (cont.)

- **Clase `TreeMap<TipoKey,TipoElem>`**
  - Esta clase implementa una colección que implementa la interface `SortedMap`. Internamente utiliza un ***arbol binario de búsqueda balanceado*** como estructura para almacenar los elementos según el orden de sus claves.
  - Las operaciones principales (***búsqueda***, ***inserción***, ***borrado***) se realizan en **O(log(n)) caso promedio** (son bastante eficientes).
  - Las **claves** utilizadas deben ser objetos que implementen el método `compareTo()` apropiadamente. Dicho método está originalmente definido en la interface `Comparable` de Java y existen clases predefinidas que lo implementan (Por ejemplo, la clase `String`). De este modo, las claves utilizadas poseen una relación de ***orden***.

Colecciones de Objetos

Iteradores

- A diferencia de lo que ocurre en C++, la estructura de datos utilizada internamente por cada colección del paquete **`java.util`** (como ser ***ArrayList***, ***LinkedList***, ***Hashtable*** o ***TreeMap***) **no** es visible para el programador, puesto que está ***encapsulada*** dentro de la colección.
- Por lo tanto, **no** es posible acceder directamente a la estructura de datos para realizar una recorrida sobre los elementos de la colección. Resulta necesaria alguna alternativa para iterar sobre ellos de forma ***indirecta***.
- Una alternativa muy usada en Java (de hecho, es la más ***eficiente*** en la mayoría de los casos) es el uso de un ***iterador***. Se trata de un objeto que permite recorrer todos los elementos de la colección **sin** necesidad de conocer la estructura de datos interna en la cual se almacenan.
- Todas las colecciones del paquete **`java.util`** poseen ***iteradores*** asociados, los cuales permiten iterar sobre todos los elementos de la colección, mediante dos métodos llamados **`hasNext()`** y **`next()`**.

Colecciones de Objetos

Iteradores (continuación)

- Los ***iteradores*** también están predefinidos en el paquete **`java.util`** y poseen los siguientes métodos:
  - **`public boolean hasNext()`**
    // determina si quedan elementos por visitar en el iterador.
  - **`public Object next()`**
    // retorna el próximo elemento a visitar en el iterador.

- **Ejemplo:** Usamos un iterador para recorrer una ***LinkedList*** de Strings.

```
LinkedList<String> lista = new LinkedList<String>();
lista.add("Juan"); lista.add("Tom"); lista.add("Ana");
Iterator<String> iter = lista.iterator();
while (iter.hasNext())
{    String nombre = iter.next();
     System.out.println (nombre);
}
```

Colecciones de Objetos

Iteradores (continuación)

- **Ejemplo:** Usamos un iterador para recorrer un ***TreeMap*** que almacena objetos de tipo **`String`** (identificados mediante claves de tipo **`Integer`**).

```
TreeMap<Integer, String> arbol =
                new TreeMap<Integer, String> ();
arbol.put (222,"Juan");
arbol.put (111,"Tom");
arbol.put (333,"Ana");
Iterator<String> iter = arbol.values().iterator();
while (iter.hasNext())
{    String elem = iter.next();
     System.out.println (elem);
}
```

Observación: Por ser un ***TreeMap*** (árbol binario de búsqueda) los elementos se listan **ordenados** según sus claves (primero se lista **Tom**, luego **Juan** y por último **Ana**).

Colecciones de Objetos

Otras alternativas para recorrer colecciones

- Si bien los ***iteradores*** suelen ser la alternativa más eficiente para recorrer los elementos de una colección, Java provee otras alternativas.
- Una de ellas es una variante a la instrucción **`for`** tradicional que permite iterar sobre todos los elementos de cualquier ***secuencia*** (como ser un **arreglo** común y corriente, un ***ArrayList*** o una ***LinkedList***).

Ejemplo: Usamos el nuevo **`for`** para recorrer una ***LinkedList*** de Strings.

```
LinkedList<String> sec = new LinkedList<String>();
sec.add("Juan");
sec.add("Tom");
sec.add("Ana");
for (String elem: sec) // para cada elemento de sec
    System.out.println (elem);
```

Observación: Internamente, Java va asignando a **`elem`** una referencia a cada elemento de la lista conforme recorre, hasta haber pasado por todos.

Colecciones de Objetos

Otras alternativas para recorrer colecciones (continuación)

- Otra alternativa es otra variante a la instrucción **`for`** tradicional (llamada **`forEach`**) que permite iterar sobre todos los elementos de casi cualquier ***colección*** (sin importar cuál sea si estructura de datos interna).

  Ejemplo: Usamos **`forEach`** para recorrer una ***LinkedList*** de Strings.

```
LinkedList<String> lista = new LinkedList<String>();
lista.add("Juan");
lista.add("Tom");
lista.add("Ana");
lista.forEach(elem -> System.out.println (elem) );
```

  Observación: Internamente, Java va asignando a **`elem`** una referencia a cada valor de la lista conforme recorre, hasta haber pasado por todos.

  Nota: La instrucción **`forEach`** **no** sirve para recorrer **arreglos** comunes y corrientes. Para ellos, se puede usar tanto la instrucción **`for`** tradicional como la nueva variante de **`for`** presentada en la diapositiva anterior.

Colecciones de Objetos

Otras alternativas para recorrer colecciones (continuación)

- **Ejemplo:** Usamos **`forEach`** para recorrer un ***TreeMap*** que almacena objetos de tipo **`String`** (identificados mediante claves de tipo **`Integer`**)

```
TreeMap<Integer, String> arbol =
                    new TreeMap<Integer, String>();
arbol.put(222,"Juan");
arbol.put(111,"Tom");
arbol.put(333,"Ana");
arbol.forEach((key, elem) -> { System.out.print(key);
                               System.out.print(" - ");
                               System.out.println(elem);
                             }
                    );
```

Observación: Por ser un ***TreeMap*** (árbol binario de búsqueda) las parejas (clave, elemento) se listan **ordenadas** según las claves (primero **111 - Tom**, luego **222 - Juan** y por último **333 - Ana**).

Colecciones de Objetos

Un Ejemplo Completo

- Presentamos a continuación una clase que define un **Diccionario abstracto de objetos genéricos**. En ella implementamos los métodos primitivos de Diccionario. Presentamos luego un diccionario **específico** de **Vehiculos**, con los métodos que son específicos de la colección de Vehículos. Optamos por un **Hash** de 10 cubetas como estructura para el diccionario.

```
import java.util.Hashtable;
import java.util.Iterator;
...
public abstract class Diccionario <K,T>
{
  // usamos una estructura de hash para almacenamiento
  // la cantidad de cubetas se pasa por parámetro en el
  // constructor de Hashtable
  protected Hashtable <K,T> tabla;
  ...
}
```

Colecciones de Objetos

Un Ejemplo Completo (continuación)

- Implementamos los métodos de la clase **Diccionario**:

```
public Diccionario ()
{  tabla = new Hashtable <K,T> (10);  }

public boolean member (K clave)
{   return tabla.containsKey(clave);  }

public T find (String clave)
{   return tabla.get(clave);   }

public void insert (K clave, T objeto)
{   tabla.put(clave,objeto);   }

public void delete (K clave)
{   tabla.remove(clave);}
```

Colecciones de Objetos

Un Ejemplo Completo (continuación)

- Presentamos a continuación otra clase que define un **Diccionario específico de Vehículos**, la cual ***hereda*** del diccionario genérico En ella implementamos un método específico propio de esta nueva colección.

```
import java.util.Iterator;
...
public class Vehiculos
extends Diccionario <String, Vehiculo>
{
  /* heredo la estructura de hash junto con las */
  /* operaciones primitivas del diccionario genérico */
  /* defino además un método específico de esta clase */

 public double promedioPrecios() ...
 // calcula promedio de precios de todos los vehículos
 // usa un iterador para recorrer los vehículos del hash
}
```

Colecciones de Objetos

Un Ejemplo Completo (continuación)

```
public double promedioPrecios()
{    // obtengo el iterador de la tabla de hash
     Iterator<Vehiculo> iter = tabla.values().iterator();
     double suma = 0.0;

     // recorro los Vehículos a través del iterador
     while (iter.hasNext())
     {  Vehiculo ve = iter.next();
        suma = suma + ve.getPrecio();
     }

     // calculo el promedio
     if (tabla.size() > 0)
        return (suma / tabla.size());
     else
        return 0.0;
}
```

Serialización y Flujos de I/O

- En Java podemos enviar y recibir objetos hacia y desde diversos dispositivos de I/O (entrada y salida).

  Ejemplos:

  - Queremos respaldar un conjunto de objetos es un archivo en disco, para luego recuperarlos en una próxima ejecución.
  - Queremos enviar un objeto conteniendo ciertos datos hacia un programa que se encuentra corriendo en otra máquina de la red.

- En Java, la transferencia de información entre un programa y un dispositivo de I/O se realiza a través de unos objetos especiales denominados **flujos de entrada y salida** (definidos en el paquete **`java.io`**).

- Existen en Java diversos flujos de I/O definidos para diferentes propósitos. Por ejemplo, hay flujos especializados en lectura y escritura de texto en archivos, hay flujos especializados en transferir información a través de una red, y hay flujos especializados en el respaldo de objetos en disco.

Serialización y Flujos de I/O

- Dependiendo de lo que se quiera hacer, se deben elegir flujos de comunicación apropiados para el tipo de dispositivo de I/O a manipular y la tarea a desempeñar mediante dicho dispositivo.

  Ejemplos:

  - Si el dispositivo de I/O es un archivo para respaldo de datos y lo que se quiere es guardar objetos en él, los flujos a utilizar se llaman **`FileOutputStream`** y **`ObjectOutputStream`**. El primero se encarga de abrir el archivo para respaldo de objetos y el segundo se encarga de escribir tales objetos en el archivo.
  - Si el dispositivo de I/O es un archivo de texto y lo que se quiere hacer es leer el texto almacenado en él, los flujos a utilizar se llaman **`FileReader`** y **`BufferedReader.`** El primero se encarga de abrir el archivo de texto y el segundo se encarga de leer el texto desde él.

Serialización y Flujos de I/O

- En Java, para que un objeto pueda ser transferido a través de un flujo de comunicación entre nuestro programa y un dispositivo de I/O, dicho objeto debe ser ***serializable***.
- Que un objeto sea ***serializable*** significa que el objeto en cuestión debe poder ser transformado a un formato de bytes adecuado para poder "viajar" a través del flujo de comunicación.
- En Java, esto se logra implementando una interface predefinida llamada **`Serializable`** (definida en el paquete **`java.io`**).
- La interface **`Serializable`** no posee ningún método. Sólo se utiliza para indicar que los objetos de la clase que la implementa tendrán la propiedad de poder ser serializados.

Serialización y Flujos de I/O

- **Ejemplo**:

```
import java.io.Serializable;

public class Vehiculo implements Serializable
{
    private static final long serialVersionUID = 1L;
    private String modelo;
    private String matricula;
    private double precio;

    ...
}
```

- A partir de ahora, todo Vehículo será ***serializable***. Esto significa que, por ejemplo, es respaldable en un archivo en disco o enviable a través de la red.
- El atributo ***serialVersionUID*** es un "código de serialización" que Java define para la clase, como forma de identificar qué versión de la clase está siendo usada para serializar sus objetos (no le daremos mucho uso en la práctica).

Serialización y Flujos de I/O

- **Observaciones:**
  - Los objetos en Java son por defecto **NO** serializables. Si queremos poder transferirlo por un flujo de comunicación, debemos hacerlo explícitamente serializable. En caso contrario, ocurrirá una **Excepción**.
  - Si un objeto serializable posee otros objetos como atributos, los mismos **también** deberán ser serializables.
  - Cuando un Objeto serializable es transferido por un flujo de comunicación, también son transferidos **automáticamente** todos sus atributos. Si alguno de ellos **NO** es serializable, ocurrirá una **Excepción** y **toda** la transferencia será cancelada.
  - Toda clase derivada de una clase que implementa **`Serializable`** también se considera serializable, siempre y cuando **no** posea atributos adicionales que **no** sean serializables.
  - Hay clases predefinidas que implementan **`Serializable`** (Por ejemplo, **`String`**) y clases predefinidas que no lo hacen (Por ejemplo, **`Thread`**).

Manejo de Archivos

Archivos de Configuración

- Son archivos en los cuales guardamos datos de configuración que son de interés a la hora de ejecutar nuestro programa.

  Ejemplos:

  - El nombre y la ruta a un archivo donde vamos a respaldar los datos manejados por nuestra aplicación.
  - La dirección IP y el número de puerto de la máquina que alberga al programa servidor de nuestra aplicación.

- Los archivos de configuración son útiles para evitar tener datos **hard–code** en nuestro código fuente. Por ejemplo, si mañana cambio el servidor de mi aplicación a otra máquina **no** tendré que recompilar mi programa cliente.

- Los archivos de configuración son archivos de texto que generalmente tienen la extensión `.properties`. La idea es escribir en ellos los datos de configuración en forma **previa** a la ejecución del programa.

Manejo de Archivos

Archivos de Configuración (continuación)

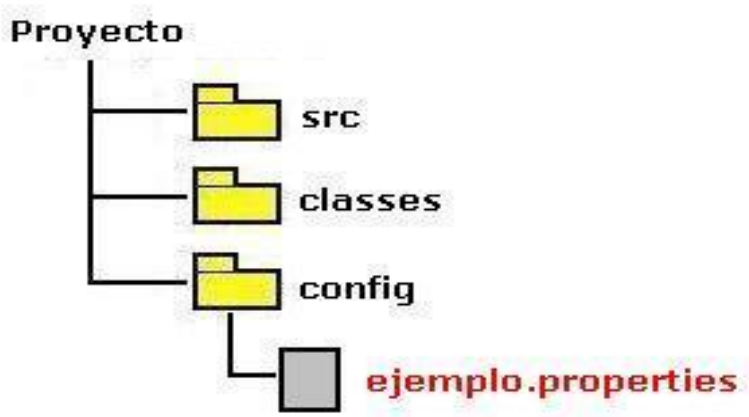
- Cada dato de configuración dentro del archivo tiene un nombre que lo identifica, un valor y debe ocupar una línea propia.

  Ejemplo: Tenemos el siguiente archivo **"ejemplo.properties"**

```
ipServidor = 127.0.0.1
puertoServidor = 5432
```

- Generalmente, se acostumbra definir alguna ubicación predefinida para colocar el o los archivos de configuración de nuestra aplicación, y luego acceder a ellos desde nuestro programa

  Ejemplo:



Manejo de Archivos

Archivos de Configuración (continuación)

```
import java.util.Properties;
import java.io.*;
...

public static void main (String args [])
{  try
    {  Properties p = new Properties();
       String nomArch = "config/ejemplo.properties";

       // Abro el archivo properties y leo los datos de configuración
       p.load (new FileInputStream (nomArch));
       String ip = p.getProperty("ipServidor");
       String puerto = p.getProperty("puertoServidor");
       ...
     }
     catch (IOException e)
     {  e.printStackTrace(); }
}
```

Manejo de Archivos

Archivos para Respaldo de datos

- Los archivos para respaldar los datos manejados por una aplicación pueden tener cualquier extensión que indique apropiadamente su utilidad. Generalmente las extensiones **`.txt`** y **`.dat`** son habituales usadas.

- Al igual que lo que ocurre en C++, el objetivo de estos archivos es solamente **respaldo** de datos. La idea no es que un usuario edite directamente dichos archivos. En Java también es posible trabajar con archivos que contengan texto leíble por un usuario (ver más adelante).

- Es conveniente que la ruta hacia el o los archivos que usemos para respaldo de datos sea **configurable**. Para ello, podemos usar archivos de configuración (**`.properties`**) del modo estudiado anteriormente.

- No debemos olvidar que para poder respaldar objetos en archivos, las clases que los definen deben implementar la interface **`Serializable`**.

Manejo de Archivos

Archivos para Respaldo de datos (continuación)

Ejemplo: Presentamos una clase que permite respaldar y recuperar desde disco un arreglo de **Vehículos**.

```
package sistema.persistencia;

import java.io.*;
import sistema.logica.vehiculos.Vehiculo;
import sistema.logica.excepciones.PersistenciaException;

public class Persistencia
{
   public void respaldar (String nomArch, Vehiculo arre[])
                              throws PersistenciaException ...

   public Vehiculo [] recuperar (String nomArch)
                              throws PersistenciaException ...
}
```

Manejo de Archivos

Archivos para Respaldo de datos (continuación)

```
public void respaldar (String nomArch, Vehiculo arre[])
                                    throws PersistenciaException
{  try
   {    // Abro el archivo y creo un flujo de comunicación hacia él
        FileOutputStream f = new FileOutputStream(nomArch);
        ObjectOutputStream o = new ObjectOutputStream(f);

        // Escribo el arreglo de vehículos en el archivo a través del flujo
        o.writeObject (arre);
        o.close();
        f.close();
   }
   catch (IOException e)
   {    e.printStackTrace();
        throw new PersistenciaException("error respaldar");
   }
}
```

Manejo de Archivos

Archivos para Respaldo de datos (continuación)

```
public Vehiculo [] recuperar (String nomArch)
                                throws PersistenciaException
{  try
   {    // Abro el archivo y creo un flujo de comunicación hacia él
        FileInputStream f = new FileInputStream(nomArch);
        ObjectInputStream o = new ObjectInputStream(f);

        // Leo el arreglo de vehículos desde el archivo a través del flujo
        Vehiculo arre[] = (Vehiculo []) o.readObject();
        o.close();
        f.close();
        return arre;
   }
   catch (IOException e)
   {    e.printStackTrace();
        throw new PersistenciaException("error recuperar");
   }
```

Manejo de Archivos

Archivos para lectura y escritura de texto

- Son archivos en los cuales podemos leer y escribir texto desde nuestros programas. La idea es que dichos archivos puedan ser luego editados por un usuario mediante algún editor de texto.

- Pueden tener cualquier extensión que se desee, aunque lo más apropiado es que tengan extensión **`.txt`**. Los datos a leer y escribir en estos archivos usualmente son manipulados siempre en forma de String y luego convertidos a algún tipo de datos apropiado.

- Al igual que con los archivos de respaldo de datos, es conveniente que la ruta hacia el o los archivos que usemos para leer y escribir texto sea **configurable** (usar archivos de configuración **`.properties`**).

- Obsérvese que al manejar estos archivos, siempre leemos y escribimos **Strings**, los cuales ya implementan por defecto la interfaz **`Serializable`**.

Manejo de Archivos

Archivos para lectura y escritura de texto (continuación)

Ejemplo: Presentamos una clase que permite escribir y leer texto (en forma de **String**) desde un archivo en disco.

```
package sistema.persistencia;

import java.io.*;

public class LecturaEscritura
{
   public void escribir (String nomArch, String texto)
                                 throws PersistenciaException ...

   public String leer (String nomArch)
                                 throws PersistenciaException ...
}
```

Manejo de Archivos

Archivos para lectura y escritura de texto (continuación)

```
public void escribir (String nomArch, String texto)
                                    throws PersistenciaException
{  try
   {    // Abro el archivo y creo un flujo de comunicación hacia él
        FileWriter f = new FileWriter(nomArch);
        BufferedWriter b = new BufferedWriter(f);
        // Escribo el texto del string en el archivo mediante el flujo
        b.write(texto);
        b.close();
        f.close();
   }
   catch (IOException e)
   {    e.printStackTrace();
        throw new PersistenciaException("error escritura");
   }
}
```

Manejo de Archivos

Archivos para lectura y escritura de texto (continuación)

```
public void leer (String nomArch, String texto)
                                  throws PersistenciaException
{  try
   {    // Abro el archivo y creo un flujo de comunicación hacia él
        FileReader f = new FileReader(nomArch);
        BufferedReader b = new BufferedReader(f);
        // Leo el texto del archivo y lo guardo en el string
        String resu = new String();
        String aux = b.readLine();
        while (aux != null)
        {
           resu = resu + aux;
           aux = b.readLine();
        }
        ...
```

Manejo de Archivos

Archivos para lectura y escritura de texto (continuación)

```
            ...
            b.close();
            f.close();
            return resu;
        }
        catch (IOException e)
        {   e.printStackTrace();
            throw new PersistenciaException("error escritura");
        }
}
```