# Reutilización: Clases predefinidas

Jacobson[1] sostenía que la reutilización del software es una de las mejores formas de hacer frente a la crisis del software, especialmente como compañera inseparable del paradigma orientado a objetos. La reutilización es considerada como una posible solución para dar respuesta a un mercado que exige productos y servicios cada vez más fiables, de bajo costo, y entregados a tiempo.
El proceso de desarrollo de software para el paradigma de orientación a objetos se comporta de forma diferente al paradigma de desarrollo clásico, si se considera la reutilización, pues ésta se encuentra incorporada en el desarrollo OO a través de mecanismos inherentes al paradigma, como la herencia.
La RAE[2] define reutilizar como "Utilizar algo, bien con la función que desempeñaba anteriormente o con otros fines".
La reutilización en Ingeniería del Software puede ser analizada desde dos puntos de vista:

- el desarrollo **para** reutilización: implica desarrollar módulos que podrán ser usados en otras ocasiones. El objetivo es elevar al máximo la cantidad de ocasiones y situaciones en que el módulo pueda ser usado.
- el desarrollo **con** reutilización: significa usar módulos ya desarrollados y probados.

Ambas, aunque estrechamente relacionadas, presentan características peculiares que involucran puntos de vista diversos que se complementan. En el primer caso el proceso de desarrollo supondrá un incremento de los costos, y el producto tendrá un valor agregado. Los desarrolladores que trabajan **para** reutilización deben poner el máximo interés para que los módulos que producen recuperen la inversión de tiempo y dinero a través de su reutilización.
En cuanto a los costos del desarrollo **con** reutilización, se estima una reducción en los costos, pues desarrollar con reutilización aumenta la productividad. Sin embargo, no todos los autores coinciden en este punto. Según McClure[3] los beneficios de la reutilización hay que buscarlos en la mejora de otros aspectos como calidad, fiabilidad, etc. y no sólo en los costos.
La programación orientada a objetos es un paradigma enfocado principalmente a la reutilización del código y a facilitar su mantenimiento. Proporciona un marco perfecto para la reutilización de las clases. El encapsulamiento y la modularidad permiten utilizar una y otra vez las mismas clases en aplicaciones distintas. Esto es así porque el aislamiento entre distintas clases significa que es posible añadir una nueva clase o un módulo nuevo sin afectar al resto de la aplicación. Esto se conoce como extensibilidad.
En este tema se abordarán **algunas** de las clases predefinidas de java que propician el desarrollo **con** reutilización. Estas clases permiten trabajar con colecciones, acceder a datos en periféricos, y trabajar con fechas.

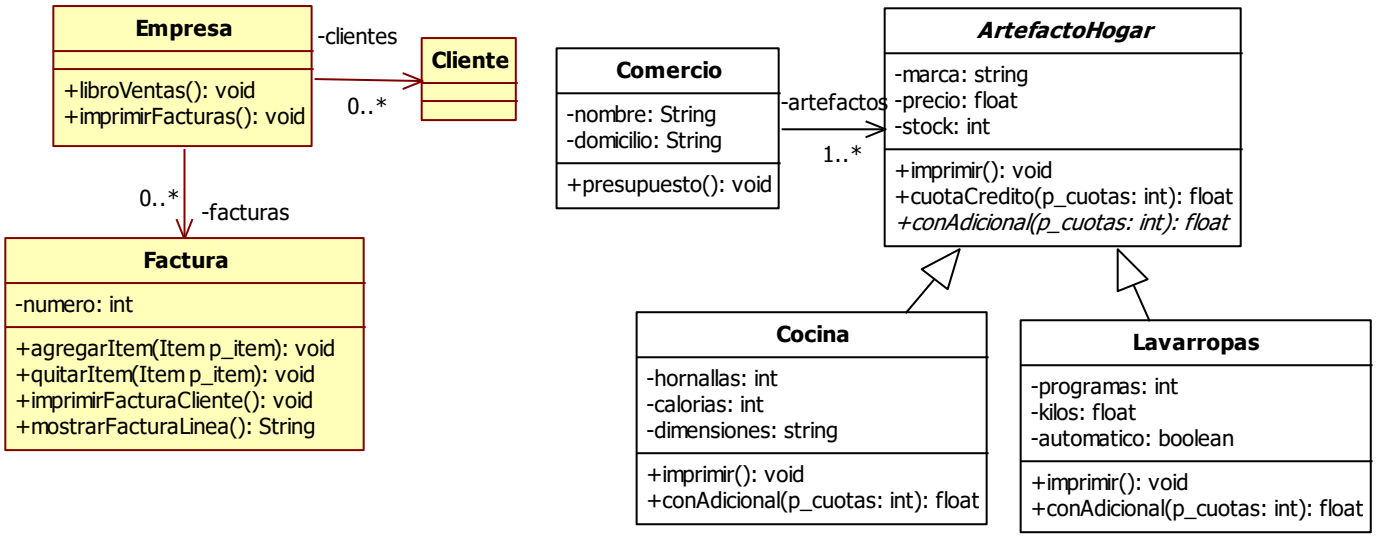
## Reutilizando Contenedores/Colecciones

El término colecciones es utilizado para hacer referencia a datos estructurados en los cuales cada elemento tiene un significado similar, aunque su valor dependa de la posición. También se los llama contenedores porque contienen otros objetos. Los ejemplos más conocidos son los arreglos, las listas enlazadas y los árboles.
Entre las ventajas que presentan los contenedores se encuentra la posibilidad de operar sobre un elemento en particular, sobre un subconjunto de elementos seleccionados mediante un filtro o sobre todos los elementos de la colección como conjunto.

1 Jacobson I.,Griss M. y Jonsson P. ,"Software reuse. Architecture, Process and Organization for Business Success", ACM Press. Addison Wesley Longman, 1997
2 Real Academia Española
3 McClure Carma, "Software Reuse Techniques. Adding Reuse to the Systems Development Process", Prentice Hall, 1997.

# Colecciones en los Lenguajes Orientados a Objetos

Desde que empezaron a surgir lenguajes de programación con cierto nivel de abstracción de datos, se soportaron las colecciones en forma estándar. Así, Fortran y Cobol manejaban arreglos, Pascal introdujo los conjuntos y varios lenguajes dieron facilidades para manejar estructuras dinámicas con la ayuda de punteros. Todo esto es previo a la orientación a objetos.
El problema que presentaba el enfoque que se le daba en esos lenguajes era la dificultad de la reutilización de código, a pesar de que semánticamente es totalmente genérico. Por ejemplo, si en Pascal se desarrolla un procedimiento que ordena un vector de enteros en forma creciente, será necesario reescribir el módulo si se desea trabajar con números reales o cadenas de caracteres.
Ada fue el primer lenguaje que encaró esta problemática, también antes de la adopción de la POO. Este lenguaje, pionero en tantos aspectos, implementó el concepto de *tipo genérico*, que admitía declarar un tipo de colección, con todas sus operaciones, difiriendo la definición del tipo del elemento hasta el momento de definir la variable de instancia de la colección. Algo parecido implementó luego C++ con sus clases *parametrizadas*, ya en el marco de la POO.
Todos los lenguajes de POO manejan arreglos en la forma tradicional, con la definición de una estructura de datos indexada, de tamaño fijo y con elementos del mismo tipo. Esta estructura, que es de las más antiguas de la programación, si bien es estática y limitada, es muy eficiente.
También se proveen, en todos los lenguajes, colecciones dinámicas. Cada colección tiene una interfaz diferente y un comportamiento también diferente, además de eficiencias diferentes para las distintas operaciones. Por ejemplo, una lista enlazada es muy eficiente para insertar y eliminar elementos, y para ser recorrida secuencialmente, pero es más lento acceder a un elemento individual que en el caso de un arreglo.
Pero lo más interesante de la POO en relación con las colecciones es la posibilidad de trabajar con elementos de varias clases, lo cual favorece la aplicación del polimorfismo y la reutilización, éste último uno de los objetivos más fuertes de la orientación a objetos.
Los siguientes diagramas de clase muestran ejemplos del uso de colecciones de objetos:




# Tipos de contenedores

**1.a) Homogéneos**
Los contenedores homogéneos son aquellos que contienen elementos de la misma clase, es decir que cada uno de sus elementos es de la misma clase.

*Ejemplo*: Un objeto de la clase Array en java, donde cada elemento del contenedor es un objeto de la clase String.

```
//La colección unArreglo contiene tres elementos del tipo String.
String unArreglo[] = {"AED1","AED2","POO"};
```

**1.b) Heterogéneos**
Un contenedor heterogéneo permite almacenar objetos de diferentes clases.

*Ejemplo:* Un objeto de la clase ArrayList en java, donde un elemento del contenedor es un objeto de tipo fecha y otro de la clase String.

```
//Objeto de la clase ArrayList (colección)
ArrayList miLista = new ArrayList();

//Objeto da la clase Calendar (fecha actual)
Calendar fecha = Calendar.getInstance();

//Objeto de la clase String (cadena "hola")
String cadena= "hola";

//Agrega cada objeto a la colección
miLista.add(fecha);
miLista.add(cadena);

La colección miLista almacena un elemento tipo Calendar y otro tipo String.
```

**2.a) Contenedores Estáticos**
Un contenedor es *estático* o *finito* si tiene un tamaño predefinido. En este caso, se le asigna un tamaño al momento de la declaración, y sólo puede almacenar un número fijo de elementos. No se puede modificar el tamaño durante la ejecución del programa. Esto es así porque se reserva para el contenedor un espacio finito y consecutivo en la memoria.
Un array es un contenedor estático que consiste en una secuencia de elementos que pueden contener: ó datos primitivos u objetos (en realidad, referencias a objetos), agrupados bajo un único identificador (o nombre). Los arrays de objetos y los arrays de tipos primitivos se usan de la misma manera. La única diferencia es que los arrays de objetos almacenan referencias, mientras que los arrays de datos primitivos guardan los valores primitivos directamente.
Todos los elementos del array deben ser del mismo tipo (homogéneo).
El identificador de array es a su vez un objeto. El único atributo de este objeto al que se puede acceder es el miembro *length* (de sólo lectura) que indica cuántos elementos pueden almacenarse en ese objeto array. No se puede saber directamente cuántos elementos hay contenidos en un array, puesto que length indica sólo cuántos elementos puede contener el array. Sin embargo, cuando se crea un objeto array sus elementos se inicializan automáticamente a *null*, por lo que se puede ver si una posición concreta del array tiene un objeto, comprobando si es o no es *null*. En el caso de un array de tipos primitivos se inicializa automáticamente a cero para datos numéricos, (char)0 (representación de cero como caracter) en el caso de caracteres, y false si se trata de valores lógicos.
El operador [ ] (corchetes) es el otro acceso que se tiene al objeto array, que permite su declaración. Las variables del tipo Array se declaran utilizando el operador [], del siguiente modo:

```
tipo_basico[]  nombre_variable;
```

Para crear el objeto, se utiliza el operador new de la forma:

```
new tipo_basico[numero_de_elementos]
```

Ejemplo:

```
String unArreglo[] = new String[10];
```

La gestión de información con el Array es muy eficiente. Se puede almacenar y acceder al azar a una secuencia de objetos. Es una secuencia lineal simple, que hace rápido el acceso a los elementos. El tiempo de acceso a los elementos del Array es el mismo para cualquiera de ellos.

**2.b) Contenedores Dinámicos**
También existen estructuras de datos de tipo *dinámicas*, es decir, colecciones de elementos (también denominados nodos), que tienen la particularidad de crecer durante la ejecución de un programa. Este tipo

de estructuras no reserva una zona estática (fija) de memoria para su almacenamiento, sino que el espacio ocupado crece o decrece según las necesidades en tiempo de ejecución.
Las estructuras de datos dinámicas se pueden dividir en dos grandes grupos:

- Lineales: Pilas, Colas, Listas
- No Lineales: Árboles, Grafos

Las estructuras de datos dinámicas son muy flexibles, por lo que son muy utilizadas para el almacenamiento de datos que cambian constantemente.

**3.a) Colecciones Ordenadas**
Es un tipo de colección que contiene elementos en una secuencia concreta. Las listas son ejemplos de colecciones ordenadas. Java incluye dos clases que permiten trabajar con listas: ArrayList y LinkedList.

**3.b) Colecciones No Ordenadas**
Es un tipo de colección de elementos que se almacenan sin ningún orden. Los conjuntos son ejemplos de colecciones no ordenadas. Java incluye los Set.

## Operaciones con colecciones

Si bien las distintas clases difieren en la disponibilidad de métodos, todas proveen como mínimo alguna forma de agregar, remover y obtener elementos.
Los métodos más comunes son:
– add(), añadir un elemento.
– contains(), comprueba si existe un elemento.
– isEmpty(), comprueba si está vacío.
– size(), devuelve el número de elementos del contenedor.
– remove(), elimina un elemento del contenedor.

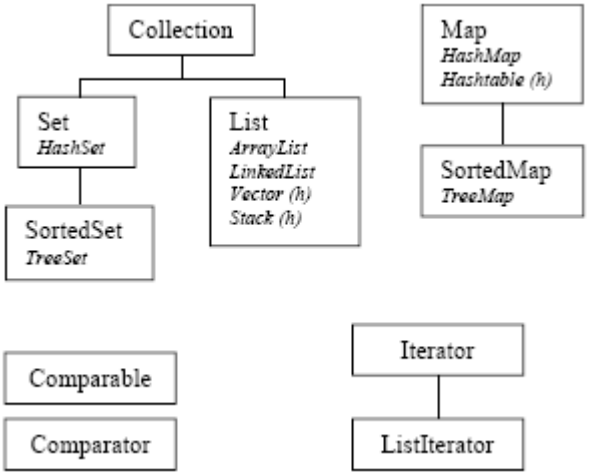
## Colecciones en Java

Java provee el *Java Collections Framework* (*JCF*) o "estructura de colecciones de Java". Es un conjunto de clases e interfaces que facilitan la manipulación de estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la programación orientada a objetos, ya que adopta el concepto de "contener objetos". A través del JCF se uniformiza la manera en que son manipulados grupos de objetos.
Todos estos contenedores contienen elementos de tipo *Object*. No permiten almacenar tipos primitivos. Para guardar tipos primitivos se deben usar clases conocidas como wrapper (envoltura): Byte para byte, Short para short, Integer para int, Long para long, Boolean para boolean, Float para float, Double para double y Character para char.

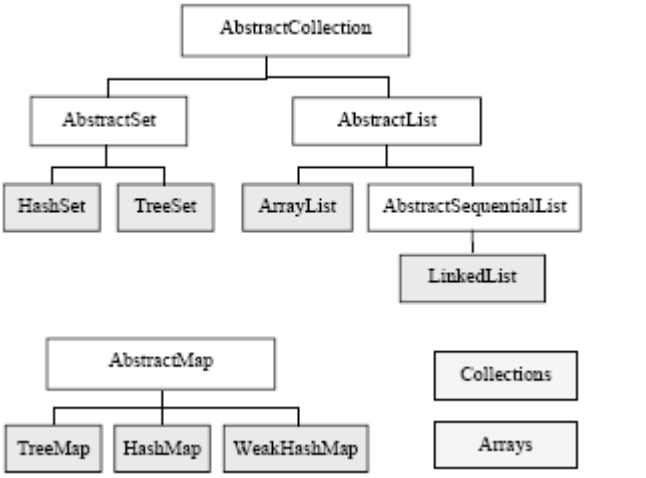
Las características principales del JCF son:

- *Interfases*: Posee una serie de interfases que permiten extender o generar clases específicas para manipular objetos en forma grupal.
- *Implementaciones*: Proporciona una serie de implementaciones (clases) concretas en las que se pueden almacenar objetos en forma grupal.
- *Algoritmos*: Ofrece una serie de métodos estándar para manipular los objetos dentro de una implementación, como ordenamiento o búsqueda.

La estructura del JCF es la siguiente:




**a) Interfaces de la Collection Frameworks**




**b) Jerarquía de clases de la Collection Frameworks**

Las interfaces tienen dos raíces en la jerarquía: *Collection* y *Map*. La diferencia entre ambas es: una *Collection* trabaja sobre conjuntos de elementos singulares o individuales, mientras que un *Map* trabaja sobre pares de valores Clave.Valor (Key.Value), que permiten buscar un valor usando una clave. Es también llamado diccionario porque se busca un valor como se busca una definición usando una palabra.
En la figura **a)**, en letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface Map: *HashMap y Hashtable*.
Existen clases que son llamadas clases "históricas", (versiones previas a la 1.2). Se denotan en la figura con la letra "h" entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF. Se recomienda utilizar las nuevas clases.
La figura **b)** muestra la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.
Las clases Collections y Arrays tienen una característica particular: no son abstract, pero no tienen constructores públicos con los cuales crear objetos. Fundamentalmente contienen métodos static para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

## Algunas estructuras de datos en Java

- ***LinkedList***

Es una lista enlazada de *Recipientes* (nodos) donde cada uno contiene elementos (objetos, otras listas, etc) y uno o dos punteros hacia posiciones de memoria que apuntan al anterior o siguiente nodo.
Útil cuando se quiere insertar o eliminar elementos al principio o al final de la lista. No permite acceder a un elemento en concreto de la lista directamente sin recorrer antes los anteriores.

- ***ArrayList***

Es una estructura de datos de tipo Array dinámica. A diferencia de los arrays clásicos (arrays estáticos), un ArrayList permite aumentar el tamaño del vector indefinidamente (hasta lo que la memoria permita) y agregar o quitar elementos.
A diferencia de la LinkedList, la ArrayList permite acceder a cualquier elemento de la lista directamente mediante su índice, lo que la hace especialmente adecuada para búsquedas rápidas.

- ***HashSet***

Un HashSet es una estructura de datos que contiene un conjunto de objetos. Permite buscar un objeto dentro del conjunto de forma rápida y fácil. Internamente gestiona un array y guarda los objetos utilizando un índice calculado con un código hash del objeto.

- Los elementos de un HashSet no están ordenados
- Para añadir un elemento al HashSet se utiliza el método **add(Object obj)**.
- Para borrar un elemento se utiliza **remove(Object obj)**.
- Para borrar todos los elementos se utiliza **clear()**.
- El tamaño del HashSet se puede obtener con la función **size()**

- ***HashMap***

Un HashMap permite guardar elementos, donde cada elemento es un par clave/valor. A diferencia de un array simple donde se guarda el valor en un índice en concreto, un HashMap determina el índice él mismo basándose en el valor hash (hashcode) generado a partir de la clave.

- ***TreeSet***

Un TreeSet mantiene los objetos ordenados en lo que se conoce como un red-black tree, es decir, en un árbol binario balanceado (cada padre tiene como máximo 2 hijos, y cuando se inserta una entrada se autobalancea de forma que quede un árbol binario simétrico).
Un TreeSet permite hacer búsquedas rápidas. No tanto como un HashMap, pero el TreeSet tiene la ventaja de estar ordenado por clave.

## Persistencia de objetos

Normalmente cuando se codifica un programa se hace con la intención de que el programa pueda interactuar con los usuarios del mismo, es decir, que el usuario pueda pedirle que realice determinada tarea, suministrándole datos con los que debe llevar a cabo la tarea solicitada. Se espera que el programa los manipule de alguna forma, proporcionando una respuesta a lo solicitado.

Por otra parte, en muchas ocasiones interesa que el programa guarde los datos que se le han introducido de forma que, al finalizar el proceso, los datos no se pierdan y puedan ser recuperados en una sesión posterior. La forma habitual de hacer esto es mediante la utilización de dispositivos de almacenamiento secundario o externo (normalmente un disco).

Se llama *persistencia* a la capacidad de una entidad de trascender el tiempo o el espacio. Es la acción de preservar información de un objeto de forma permanente, permitiendo la recuperación de la misma para que pueda ser empleada y actualizada.

Es un concepto importante, pues permite que un objeto pueda ser usado en diferentes momentos a lo largo del tiempo, por el mismo programa o por otros, así como en diferentes instalaciones de hardware. Esto es particularmente importante, dado que es habitual compartir archivos entre distintas plataformas, invocar procedimientos remotos o usar objetos distribuidos.

Los datos generalmente se almacenan con algún tipo de organización, por ejemplo, un registro, que agrupa datos de diferentes tipos (enteros, string, etc).

En los lenguajes de programación OO, como Java, C++, etc, los datos se organizan en objetos, que son instancias de clases. Esto implica que las bases de datos orientadas a objetos (BDOO) deben persistir tanto el estado de los objetos como su comportamiento. Es decir, los valores de sus atributos y la funcionalidad que tenga a cargo el objeto.

Los objetos pueden ser, según su persistencia:

- **Transitorios o efímeros**: el tiempo de vida de un objeto depende directamente del entorno del proceso que los instanció (tiempo en memoria).
- **Persistentes**: el tiempo de vida de un objeto es independiente del proceso que lo instanció, es decir, transciende el tiempo en memoria y es almacenado conservando su estado en un medio secundario de almacenaminto. Luego puede ser recuperado para su utilización o actualización, pudiendo ser reconstruido por el mismo u otro proceso

## Reutilizando clases para entrada/salida en Java

Java proporciona una extensa jerarquía de clases predefinidas, agrupadas en el paquete estándar de la API de Java, denominado java.io que incorpora interfaces, clases y excepciones, para acceder a distintos tipos de archivos. Estas clases predefinidas cuentan con todos los métodos necesarios para leer, grabar y manipular los datos.

La manera de representar las entradas y salidas en Java es a base de *streams* (flujos de datos). Un *stream* es una conexión entre el programa y la fuente o destino de los datos. La información se traslada en serie (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.

Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un stream que conecta el monitor al programa. Se da a ese stream la orden de escribir algo y éste lo traslada a la pantalla.

Existen dos tipos de Entrada/Salida:

1. **Entrada/Salida estándar:** teclado y pantalla

El acceso a la entrada y salida estándar es controlado por tres objetos que se crean automáticamente al iniciar la aplicación: System.in, System.out y System.err

***System.in***

Este objeto implementa la entrada estándar (normalmente teclado). Los métodos que proporciona son:

- *read():* Devuelve el carácter que se ha introducido por el teclado leyéndolo del buffer de entrada y lo elimina del buffer para que en la siguiente lectura sea leído el siguiente carácter. Si no se ha introducido ningún carácter por el teclado devuelve el valor -1.
- *skip(n):* Ignora los *n* caracteres siguientes de la entrada.

***System.out***

Este objeto implementa la salida estándar. Los métodos que proporciona son:

- *print(a):* Imprime *a* en la salida, donde *a* puede ser cualquier tipo básico Java ya que Java hace su conversión automática a cadena.
- *println(a):* Es idéntico a *print(a)* salvo que con *println()* se imprime un salto de línea al final de la impresión de *a*.

***System.err***

Este objeto implementa la salida en caso de error. Normalmente esta salida es la pantalla o la ventana de la terminal como con System.out, pero puede ser interesante redirigirlo, por ejemplo hacia un fichero, para diferenciar claramente ambos tipos de salidas.

Las funciones que ofrece este objeto son idénticas a las proporcionadas por System.out.

2. **Entrada/Salida a través de archivos:** se trabaja con archivos de disco.

### Tipos de ficheros

En Java es posible utilizar dos tipos de ficheros (de texto o binarios) y dos tipos de acceso a los ficheros (secuencial o aleatorio).

Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de datos (int, float, boolean,...).

Una lectura secuencial implica tener que acceder a un elemento antes de acceder al siguiente, es decir, de una manera lineal (sin saltos). Sin embargo, los ficheros de acceso aleatorio permiten acceder a sus datos de una forma aleatoria, es decir, indicando una determinada posición desde la que leer/escribir.

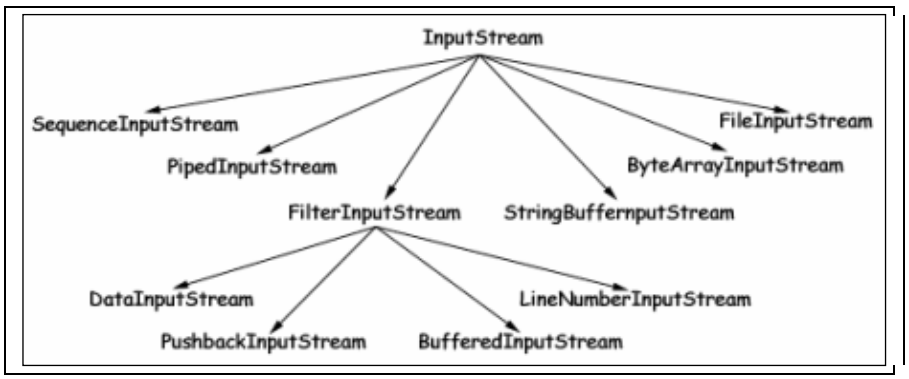
### Jerarquía de Clases

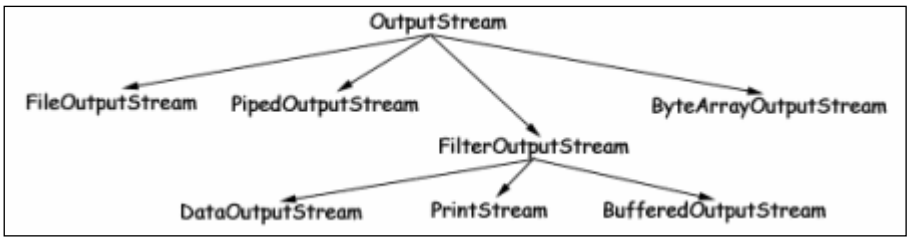
En el paquete java.io existen varias clases de las cuales se puede crear instancias para tratar todo tipo de archivos. Existen dos familias de jerarquías distintas para la entrada/salida de datos. La diferencia principal consiste en que una opera con bytes y la otra con caracteres (el carácter de Java está formado por dos bytes

porque sigue el código Unicode). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.

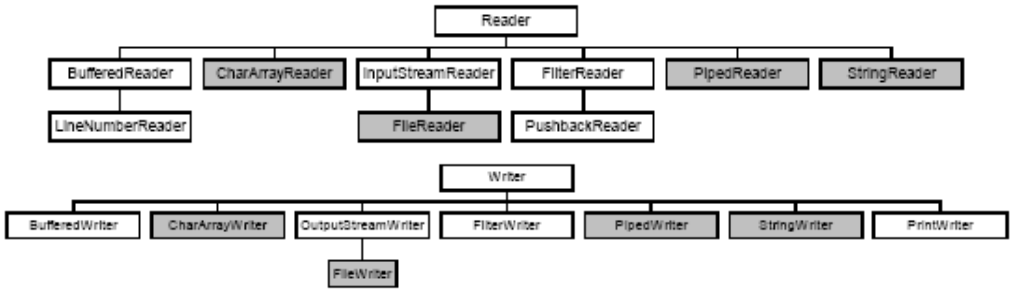
Desde Java 1.0, la entrada y salida de datos del programa se podía hacer con clases derivadas de InputStream (para lectura) y OutputStream (para escritura). Estas clases tienen los métodos básicos read() y write() que manejan bytes y que no se suelen utilizar directamente.

Las jerarquías de clases son las siguientes:




En Java 1.1 aparecieron dos nuevas familias de clases, derivadas de Reader y Writer, que manejan caracteres en vez de bytes. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las jerarquías son las siguientes:




Algunas clases de la jerarquía definen de dónde, o a dónde, se envían los datos, es decir, el dispositivo con que conecta el stream. Otras añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader entrada = new BufferedReader(new FileReader("Alumnos.txt"));
```

En primer lugar se crea un stream (flujo) de entrada de tipo FileReader, que permite leer del archivo Alumnos.txt. Luego se crea a partir de él un objeto BufferedReader, que aporta la característica de utilizar buffer. Los caracteres que lleguen a través del FileReader pasarán a través del BufferedReader, es decir utilizarán el buffer.

Del mismo modo, se pueden leer caracteres desde el teclado, pasándole un objeto de tipo System.in, que hace referencia a la entrada estándar.

```
BufferedReader tecla = new BufferedReader(new InputStreamReader(System.in));
```

De la amplia jerarquía de clases, las tres principales son:

- *FileOutputStream*: Para escritura de archivos de texto a los que se accede de forma secuencial.

- *FileInputStream*: Para lectura de archivos de texto a los que se accede de forma secuencial.
- *RandomAccessFile*: Archivo de entrada o salida binario con acceso aleatorio. Es la base para crear los objetos de tipo archivo acceso aleatorio. Estos archivos permiten gran variedad de operaciones: saltar hacia delante y hacia atrás para leer la información necesaria, e incluso leer o escribir partes del archivo sin necesidad de cerrarlo y volverlo a abrir en un modo distinto.

### Generalidades

Para trabajar con un archivo siempre se debe actuar de la misma manera:

### 1. Se abre el fichero.

Para ello hay que **crear un objeto de la clase predefinida** correspondiente al tipo de fichero a manejar, y el tipo de acceso a utilizar. Se deberá utilizar alguno de los métodos constructores que se observan en la documentación de java. El formato general será:

```
TipoDeFichero obj = new TipoDeFichero( ruta );
```

Donde *ruta* es la ruta de disco en que se encuentra el archivo o un descriptor de archivo válido.

Este formato es válido, excepto para los objetos de la clase RandomAccessFile (acceso aleatorio), para los que se debe instanciar de la siguiente forma:

```
RandomAccessFile obj = new RandomAccessFile( ruta, modo );
```

Donde modo es una cadena de texto que indica el modo en que se desea abrir el fichero: "**r**" para sólo lectura o "**rw**" para lectura y escritura.

### 2. Se utiliza el fichero.

Para ello cada clase presenta un conjunto de métodos de acceso para escribir o leer en el fichero. La variedad de métodos puede ser observados en la documentación de java.

La Clase *RandomAccessFile* presenta, además de los clásicos métodos de lectura/escritura, otro grupo de métodos que trabajan con un índice que proporciona esta clase, y que indica en qué posición del archivo se encuentra el puntero. Con él se puede trabajar para posicionarse en el archivo. Se debe tener en cuenta que cualquier lectura o escritura de datos se realizará a partir de la posición actual del puntero del fichero.

Los métodos de desplazamiento son los siguientes:

- **long getFilePointer():**Devuelve la posición actual del puntero del fichero.
- **void seek( long posi ):** Coloca el puntero del fichero en la posición indicada por **posi**. Un fichero siempre empieza en la posición 0.
- **int skipBytes( int n ):** Intenta saltar **n** bytes desde la posición actual.
- **long length():** Devuelve la longitud del fichero.

### 3. Gestión de excepciones

Se puede observar en la documentación de java que todos los métodos que utilicen clases de este paquete deben tener en su definición una cláusula throws IOException. Los métodos de estas clases pueden lanzar excepciones de esta clase (o sus hijas) en el transcurso de su ejecución, y dichas excepciones deben de ser capturadas y debidamente gestionadas para evitar problemas.

### 4. Se cierra el fichero y se destruye el objeto.

Para cerrar un fichero lo que hay que hacer es destruir el objeto. Esto se puede realizar de dos formas: dejando que sea el recolector de basura de Java el que lo destruya cuando no lo necesite (no se recomienda) o destruyendo el objeto explícitamente mediante el uso del método close() del objeto:

```
obj.close()
```

## Reutilizando clases para el tratamiento de Fechas en java

Java proporciona un conjunto de clases predefinidas que permiten manipular fechas. El procedimiento en todos los casos es el mismo:

- crear un objeto instanciando alguna de las clases que lo permiten
- usar el objeto enviándole mensajes según los métodos que proporcione cada clase

En las primeras versiones de java el manejo de fechas pasaba exclusivamente por el uso de la clase java.util.Date, pero a partir de la versión JDK 1.1 fueron añadidas nuevas funcionalidades en otras clases de apoyo como Calendar, TimeZone, Locale y DateFormat y descargando de funcionalidad a la clase Date, que de igual modo sigue siendo la clase base para el uso de fechas.

Existen tres clases en la documentación Java API que permiten manipular fechas:

- **Date** crea un objeto Date (paquete java.util)
- **Calendar** configura o cambia la fecha de un objeto Date (paquete java.util).
- **DateFormat** muestra la fecha en diferentes formatos (paquete java.text).

La jerarquía de clases es la siguiente:

***java.util***

java.lang.Object
  └ **java.util.Date**

***java.util***

java.lang.Object
  └ java.util.Calendar
    └ **java.util.GregorianCalendar**

***java.text***

java.lang.Object
  └ java.text.Format
    └ java.text.DateFormat
      └ **java.text.SimpleDateFormat**

Para trabajar con fechas se deben importar los paquetes correspondientes.

```
import java.util.*;
import java.text.*;
```

La clase Date representa un momento específico del tiempo en milisegundos, mientras que Calendar permite obtener números enteros que especifican el día, mes, año, hora, etc. de dicha fecha.