Programación Orientada a Objetos

Tema 7:
Estructuras de Datos y Librería I/O

Contenidos

- Tema 7: Estructuras de Datos y Librería I/O

Estructuras de Datos

Fundamentos de Ordenación y Búsqueda

- Los algoritmos de ordenación y búsqueda son **ampliamente utilizados** en las aplicaciones modernas.
- Existe una gran variedad de algoritmos que intentan **optimizar** distintas situaciones, aplicando **diferentes técnicas.**
- Muchos lenguajes de alto nivel proporcionan funciones que evitan que el usuario deba implementar estos algoritmos por sí mismo.
- Hay que tener en cuenta el coste computacional asociado a la búsqueda y ordenación sobre colecciones de tamaño N.

Búsqueda secuencial

Localizar, en una colección de elementos, el primer elemento que cumple una condición dada.

Ej: buscar aula libre en una determinada fecha y horario que tenga al menos n asientos

Con mucha frecuencia los elementos (objetos-sujetos) que interesa tratar informáticamente son identificables → **clave** =identificador unívoco

Ej: localizar **la** ficha del alumno con un determinado número de expediente

La búsqueda proporciona la posición, la referencia o una copia del **primer** elemento en la colección de elementos que satisface una condición determinada.

Si no hay elemento que satisfaga la condición, se devolverá una indicación de no encontrado.

Algoritmo de búsqueda secuencial

La implementación concreta depende de la estructura de datos

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION posicion( L, clave)
{
   actual←primero_de_L;
   MIENTRAS queden y clave≠actual.clave HAZ actual←siguiente;
   SI clave=actual.clave ENTONCES posicion←pos_de_actual
   EN CASO CONTRARIO posicion←NoEncontrado
}
```

Búsqueda binaria

Aplicabilidad
- La lista está ordenada por la clave de búsqueda
- Se conoce el número de elementos
- Se tiene acceso directo al elemento por posición en la lista

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION posicion_bin( L, clave)
{
   tam←tamaño_de_la_colección;
   inf←0;                                   //inf: limite inferior del intervalo
   sup←tam-1;                               //sup: limite superior del intervalo
   MIENTRAS inf<=sup HAZ
  {
        centro = ((sup + inf) / 2);         //centro: elemento central del intervalo
        SI L[centro].clave = clave ENTONCES
        {
                posicion←pos_de_L[centro];
                SALIR;
        }
       SI clave<L[centro].clave ENTONCES sup=centro-1
       EN CASO CONTRARIO inf=centro+1
  }
  posicion← NoEncontrado;
  SALIR;
}
```


Ordenación por inserción directa

La colección inicial de elementos se descompone en dos partes: Una parte cuyos elementos mantienen un orden y la parte restante.

Inicialmente la parte ordenada consta de un solo elemento (el primero)

En la iteración i, los i primeros elementos están ya ordenados. Se localiza la posición que correspondería en la parte ordenada al primer elemento de la parte desordenada (elemento i+1) y **se inserta** en esa posición.

Ejemplo (En cada iteración se marca la parte ordenada con subrayado):

i=1	44 **55** 12 42 94 18 06 67
2	44 55 **12** 42 94 18 06 67
3	12 44 55 **42** 94 18 06 67
4	12 42 44 55 **94** 18 06 67
5	12 42 44 55 94 **18** 06 67
6	12 18 42 44 55 94 **06** 67
7	06 12 18 42 44 55 94 **67**
9	06 12 18 42 44 55 67 94

Ordenación por selección directa

La colección inicial de elementos se descompone en dos partes: Una parte cuyos elementos mantienen un orden y la parte restante.

En la iteración i se busca dentro de la parte desordenada el elemento cuyo valor clave es menor (o mayor, según el criterio de ordenación) y se coloca en esa posición, de forma que deja de pertenecer a la parte desordenada de la colección.

Ejemplo (En cada iteración se marca la parte ordenada con subrayado):

i=1	44 55 12 42 94 18 **06** 67
2	06 55 **12** 42 94 18 44 67
3	06 12 55 42 94 **18** 44 67
4	06 12 18 **42** 94 55 44 67
5	06 12 18 42 94 55 **44** 67
6	06 12 18 42 44 **55** 94 67
7	06 12 18 42 44 55 94 **67**
9	06 12 18 42 44 55 67 94

Ordenación por intercambio directo

También es conocido como *método de la burbuja*.

Se basa en realizar pasadas sucesivas sobre todos los elementos de la colección. En cada pasada se compara cada uno de los elementos de la colección con su adyacente, y se realiza el intercambio entre ellos en caso de que el criterio de ordenación lo requiera (como si hubiera que ordenar sólo los dos elementos comparados).

PSEUDOCÓDIGO DEL ALGORITMO GENERAL

```
FUNCION ordenarBurbuja (L)
{
    PARA i=0 HASTA tamaño_de_la_colección-1 HAZ
    {
        PARA j=0 HASTA tamaño_de_la_colección-2 HAZ
            SI L[j].clave>L[j+1].clave ENTONCES          // caso de ordenación de menor a mayor
            {
                    // se intercambian los elementos consecutivos
                    aux←L[j];
                    L[j]←L[j+1];
                    L[j+1]←aux;
            }
    }
}
```

Ordenación por intercambio directo

Ejemplo *método de la burbuja*.

Inicio:	44 55 12 42 94 18 06 67
i=0	44 12 42 55 18 06 67 94
1	12 42 44 18 06 55 67 94
2	12 42 18 06 44 55 67 94
3	12 18 06 42 44 55 67 94
4	12 06 18 42 44 55 67 94
5	06 12 18 42 44 55 67 94
6	06 12 18 42 44 55 67 94
7	06 12 18 42 44 55 67 94

TIPOS ENUMERADOS

- **Tipos Enumerados:**
- Los tipos enumerados sirven para restringir la selección de valores a un conjunto previamente definido.
- Un tipo enumerado permite que una variable tenga solo un valor dentro de un conjunto de valores predefinidos, es decir, valores dentro de una lista enumerada.
- Los valores que se definen en un tipo enumerado son constantes y se suelen escribir en mayúsculas.
- La clase que representa los tipos enumerados en Java es java.lang.Enum.
- Posee una serie de métodos útiles como:
  - toString(): permite visualizar el nombre de las constantes de una enumeración.
  - ordinal(): permite saber el orden de declaración de las constantes.
  - values(): genera un vector con los valores de la enumeración.

TIPOS ENUMERADOS

- **Ejemplos Tipos Enumerados:**

```
public enum ColoresSemaforo {
    VERDE, NARANJA, ROJO
}

public class PruebaEnum {
    public static void main(String[] args) {
        ColoresSemaforo cs = ColoresSemaforo.VERDE;
        switch (cs) {
            case ROJO:
                System.out.println("No puedes pasar.");          break;
            case VERDE:
                System.out.println("Puedes pasar.");            break;
            case NARANJA:
                System.out.println("Cuidado al pasar.");        break;
        }
        if (cs.equals(ColoresSemaforo.VERDE)) {    cs = ColoresSemaforo.ROJO;    }
        System.out.println(cs.toString());
        for (ColoresSemaforo csf : ColoresSemaforo.values()) {
            System.out.println(csf + ", ordinal " + csf.ordinal());
        }
    }
}
```

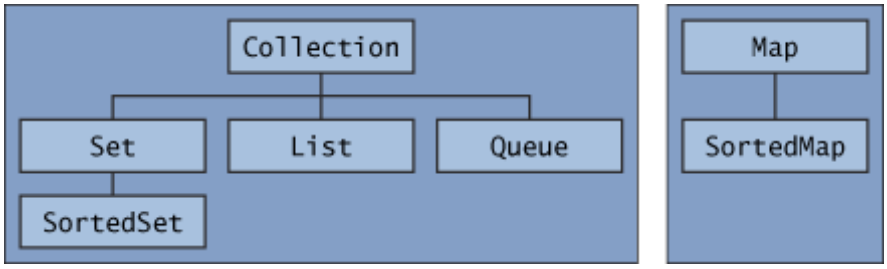
```
public class PruebaEnum2 {

    enum DiasSemana {
        L, M, X, J, V, S, D
    };

    public static void main(String[] args) {
        DiasSemana ds = DiasSemana.L;
        System.out.println(ds);
    }
}
```


COLECCIONES

- Una colección es un objeto que recopila y organiza otros objetos.
- La colección define las formas específicas en las que se puede acceder y con las que se pueden gestionar esos objetos, los cuales se denominan elementos de la colección.
- Existen muchos tipos específicos de colecciones que permiten resolver distintas problemáticas.
- Las colecciones se pueden clasificar en dos categorías generales: lineales y no lineales (ej.: jerárquica o en red).
- La organización relativa de los elementos de una colección está determinada usualmente por:
  - El orden en que se añaden los elementos a la colección.
  - Alguna relación inherente entre los propios elementos.

- Las *Colecciones* en Java ofrecen un mecanismo orientado a objetos para almacenar conjuntos de datos de tipo similar.
- Tienen su propia asignación de memoria y un conjunto de métodos para su iteración y recorrido.
- El **framework de las colecciones** en Java se basa en una arquitectura unificada que contiene:
  - **Interfaces**. Tipos abstractos de datos que representan a las colecciones. Permiten la manipulación de la colección de forma independiente de los detalles de su representación o implementación.
  - **Implementaciones**. Representan las implementaciones concretas de las interfaces. Son estructuras de datos reutilizables.
  - **Algoritmos**. Representan métodos de utilidad para realizar búsquedas y ordenación. Estos métodos son polimórficos.

- Las clases que representan colecciones en Java se encuentran dentro del paquete java.util.
- Las clases que representan colecciones se basan en una serie de interfaces que definen los métodos necesarios para su gestión y que estas clases implementarán.
- Las interfaces más importantes son las siguientes:



- Las interfaces de las colecciones se basan en **genéricos**. Los genéricos implementan el concepto de *tipos parametrizados*, que permiten crear colecciones que resulten fáciles de utilizar con múltiples tipos.

- El término “genérico” significa “perteneciente o apropiado para grandes grupos de clases”.

- Cuando nos encontramos con la definición de una interface o clase donde se utilice la sintaxis <E> nos está indicando que se basa en genéricos. Por ejemplo:

```
public interface Collection<E>
```


- La interface **Collection** representa una secuencia de elementos individuales a los que se aplica una o más reglas.

- Una colección **List** debe almacenar los elementos en la forma en que fueron insertados, una colección **Set** no puede tener elementos duplicados.

- La interface **Map** representa un grupo de parejas de objetos clave-valor, que permite realizar búsquedas de objetos. No se permiten claves duplicadas.

COLECCIONES: LISTAS

- **Listas:**
- Un **ArrayList** es un array de objetos cuyo tamaño puede variar en tiempo de ejecución. Implementa la interface **List**.
- Los objetos se pueden almacenar al final de la colección o en una posición concreta utilizando el método *add()* y borrarlos mediante *remove().*
- También podemos remplazar un elemento de la colección con el método *set()*.
- Se puede buscar un elemento en concreto utilizando los métodos *contains()*, *indexOf()* o *lastIndexOf()*.
- Se puede extraer un objeto de una posición específica utilizando el método *get()*.

EJEMPLO ARRAYLIST

```
import java.util.*;
public class PruebaArrayList {
   private static final String[] COLORES = {"rojo", "verde", "azul"};
   public static void main(String[] args) {
      ArrayList<String> array = new ArrayList<>();
      imprimeArrayList(array);
      // añadimos elementos al ArrayList
      array.add("amarillo");
      array.addAll(Arrays.asList(COLORES));
      array.add("blanco");
      imprimeArrayList(array);

      // primer y último elemento
      System.out.println("Primer elemento: " + array.get(0));
      System.out.println("Último elemento: " + array.get(array.size() - 1));

      // encontrar un elemento
      if (array.contains("rojo")) {
         System.out.println("\n\"rojo\" encontrado en el índice " +
               array.indexOf("rojo") + "\n");
      } else {
         System.out.println("\n\"rojo\" no encontrado\n");
      }
      // borrar un elemento
      array.remove("rojo");
      // imprimimos el contenido
      //System.out.println(array.toString());
      imprimeArrayList(array);
   }
```

```
private static void imprimeArrayList(ArrayList array) {
      if (array.isEmpty()) {
         System.out.print("el ArrayList está vacío");
      } else {
         System.out.print("Contenido del ArrayList: ");

         Iterator items = array.iterator();

         while (items.hasNext()) {
            System.out.print(items.next() + " ");
         }
      }

      System.out.println("\n");
   }
}
```

COLECCIONES: CONJUNTOS

- **Conjuntos:**

- **HashSet:**
  - Un **HashSet** se basa en una *tabla hash* para almacenar los objetos y es una implementación de la interface Set y representa un conjunto de valores que no admite duplicados.
  - Para almacenar un objeto utilizaremos el método *add()* y para borrarlo *remove().*

- *Nota: Una tabla hash o mapa hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (por ejemplo nombres de personas o saldos de cuentas) almacenados a partir de una clave generada (por ejemplo el dni o el número de cuenta). Funciona transformando la clave con una función hash en un hash, un número que la tabla hash utiliza para localizar el valor deseado.*

EJEMPLO HASHSET

```
import java.util.*;

public class PruebaHashSet {
   private static final String[] COLORES = {"rojo", "verde", "azul"};
   public static void main(String[] args) {
      HashSet<String> conjunto = new HashSet<>();
      // añadimos elementos al conjunto
      conjunto.add("amarillo");
      conjunto.add("amarillo"); //no se añade porque no admite duplicados
      conjunto.addAll(Arrays.asList(COLORES));
      conjunto.add("blanco");
      // imprimimos el contenido
      System.out.println(conjunto.toString());
      // encontrar un elemento
      if (conjunto.contains("rojo")) {
         System.out.println("\n\"rojo\" encontrado\n");
      } else {
         System.out.println("\n\"rojo\" no encontrado\n");
      }
      // borrar un elemento
      conjunto.remove("rojo");
      // imprimimos el contenido
      System.out.println(conjunto.toString());
   }
}
```

- **Mapas:**

- **HashMap:**

  - Una **HashMap**, es una *tabla hash*, realiza una implementación de la interface **Map**, por lo tanto, representa una colección para almacenar objetos claves/valores.

  - Para almacenar un objeto utilizaremos el método *put(clave,valor*). Después se puede utilizar *get(clave)* para recuperar el objeto. Para borrarlo utilizaremos *remove(clave).*


```
import java.time.*;
import java.util.*;
public class PruebaHashMap {
   public static void main(String[] args) {
      HashMap<String, Persona> personas = new HashMap<>();
      //Creamos personas
      LocalDate f1 = LocalDate.of(1965, Month.JANUARY, 1);
      LocalDate f2 = LocalDate.of(1975, Month.FEBRUARY, 10);
      Persona per1 = new Persona("06634246S", "Javier García", f1, "calle1");
      Persona per2 = new Persona("65834916K", "José Sánchez", f2, "calle2");
      //Las introducimos en la HashMap
      personas.put(per1.getDni(), per1);
      personas.put(per2.getDni(), per2);
      System.out.println(personas.toString());
      //Recuperamos una persona mediante su DNI
      Persona obj = personas.get("06634246S");
      System.out.println("Nombre: " + obj.getNombre());
      //Modificamos datos
      obj.setNombre("Nuevo Nombre");
      //Eliminamos una persona
      personas.remove("65834916K");
      if (personas.containsKey("65834916K")) {  System.out.println("No eliminada"); } else {
         System.out.println("Eliminada");
      }  //Presentamos la información
      System.out.println(personas.toString());
   }
}
```

COLECCIONES: ITERADORES

- **Iteradores:**
- **Iterator:**
  - **Iterator** es una interfaz simple que permite recorrer todos los elementos de una colección de objetos.
  - Especifica dos métodos para su recorrido *hasNext()* y *next()*.
- **ListIterator:**
  - **ListIterator** es un subtipo de Iterator y es más potente ya que permite realizar un recorrido bidireccional de la colección añadiendo el método *hasPrevious()* y *previus()*.

EJEMPLO ITERADORES

```
import java.time.*;
import java.util.*;
public class Iteradores {
   public static void main(String[] args) {
      //Fechas de nacimiento
      LocalDate f1 = LocalDate.of(1965, Month.JANUARY, 1);       LocalDate f2 = LocalDate.of(1975, Month.FEBRUARY, 10);
      LocalDate f3 = LocalDate.of(1980, Month.APRIL, 15);       LocalDate f4 = LocalDate.of(1985, Month.NOVEMBER, 25);
      //Diversos objetos de tipo persona
      Persona obj1 = new Profesor("06634246S", "Javier García", f1, "calle1", "Electrónica", 2000);
      Persona obj2 = new Profesor("65834916K", "José Sánchez", f2, "calle2", "Computación", 1500);
      Persona obj3 = new Alumno("91635476F", "María Rubio", f3, "calle3", "Informática", "Alcalá");
      Persona obj4 = new Alumno("15664386T", "Carmen Pérez", f4, "calle4", "Industriales", "Zaragoza");
      //Introducimos los objetos en un ArrayList
      ArrayList<Persona> personas = new ArrayList<>();
      personas.add(obj1);
      personas.add(obj2);
      personas.add(obj3);
      personas.add(obj4);
      ListIterator<Persona> li = personas.listIterator();
      System.out.println("Sentido directo:");
      while (li.hasNext()) {
         System.out.println(li.next());
      }
      System.out.println("Sentido inverso:");
      while (li.hasPrevious()){
         System.out.println(li.previous());
      }
   }
}
```

- **Algoritmos:**
- La clase **Collections** contiene una serie de algoritmos de utilidad para aplicarlos a las colecciones, estos algoritmos son polimórficos ya que se pueden aplicar a cualquier tipo de dato. Entre sus métodos destacan los de ordenación y búsqueda. Todos los métodos de la clase son estáticos. La mayoría se aplican sobre las colecciones de tipo lista.
- El método *sort()* nos permite ordenar una lista, bien por su orden natural, o bien a través de un comparador.
- El método *binarySearch()* nos permite buscar un elemento en una lista ordenada.
- El método *reverse()* nos permite ordenar una lista en orden inverso.

Universidad de Alcalá
COLECCIONES: ALGORITMOS
Departamento Ciencias de la Computación

- **Algoritmos:**
- Los métodos min() y max() nos permiten saber cual es el elemento menor y mayor de la lista.
- El método *frecuency()* nos permite saber el número de veces que se repite un elemento de una lista.
- El método *rotate()* nos permite mover todos los elementos de una lista hacia adelante un determinado número de posiciones, extrayéndolos por el extremo y colocándolos al principio.
- El método *suffle()* nos permite permutar una lista de manera aleatoria.
- El método swap() nos permite intercambiar dos elementos de una lista.

ORDENACIÓN DE OBJETOS

- Para ordenar objetos a través de alguno de sus atributos podemos utilizar objetos comparadores o implementar la interfaz **Comparable**.
- Un objeto **Comparator** nos proporcionará la forma en la que debemos ordenar la colección a través del método **compareTo()**.

```
//Comparador para ordenar las personas por su DNI
Comparator DniPerComp = new Comparator() {
      public int compare(Object o1, Object o2) {
            Persona per1 = (Persona) o1;
            Persona per2 = (Persona) o2;
            return per1.getDni().compareTo(per2.getDni());
      }
};

//Comparador para ordenar las personas por su edad
Comparator EdadPerComp = new Comparator() {
      public int compare(Object o1, Object o2) {
            Persona per1 = (Persona) o1;
            Persona per2 = (Persona) o2;
            Integer e1 = per1.getEdad();
            Integer e2 = per2.getEdad();
            return e1.compareTo(e2);
      }
};
```

```
public class Persona
      implements Comparable<Persona> {
   private String dni;
   private String nombre; …

    public int compareTo(Persona p) {
        return this.dni.compareTo(p.getDni());
    }
}
```


ORDENACIÓN DE OBJETOS

- Ordenación de los objetos de un **ArrayList** por el atributo que seleccionemos y búsqueda de un elemento.

```
//Introducimos los objetos en un ArrayList
ArrayList<Persona> personas = new ArrayList<>();
personas.add(obj1); …          personas.add(obj4);
//Ordenamos los objetos del array por el atributo Nombre
Collections.sort(personas, NomPerComp);
//Buscamos una persona por su nombre
BufferedReader entrada=
      new BufferedReader(new InputStreamReader(System.in));
System.out.println("\nIntroduce el nombre de la persona a buscar:");
String nombre = entrada.readLine();
//creamos una persona con el nombre a buscar
Persona p = new Persona();
p.setNombre(nombre);
int pos = Collections.binarySearch(personas, p, NomPerComp);
if (pos>=0) {
            System.out.println("\nDatos de la persona:");
            Persona per = personas.get(pos);
            System.out.println(per.toString());
} else System.out.println("\n Persona no encontrada.");
```

ORDENACIÓN DE OBJETOS

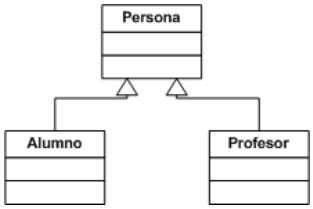
- Ordenación de los objetos de una **HashMap** a través de un ArrayList por el atributo que seleccionemos.

```
//Introducimos los objetos en una tabla hash
HashMap<String, Persona> personas = new HashMap<>();
personas.put(obj1.getDni(), obj1); …                    personas.put(obj4.getDni(), obj4);

//Convertimos la tabla hash en un ArrayList de objetos
Collection<Persona> colec = personas.values();
ArrayList<Persona> personasA = new ArrayList<>(colec);

//Ordenamos los objetos del array por el atributo nombre
Collections.sort(personasA, NomPerComp);

//Presentamos la información
System.out.println("\nPersonas ordenadas por Nombre:");
 for (Persona per : personasA) {
    System.out.println(per.toString());
 }
```


TIPOS DE OBJETOS EN COLECCIONES

- Para saber con que clase se instanció un objeto se debe utilizar el método *getClass* de la clase Object, si lo complementamos con *getSimpleName* de la clase Class, nos devuelve una cadena con el nombre de la clase.
- Ejemplo: Tenemos una jerarquía de clases e introducimos diversos objetos en una tabla hash de tipo Persona:

```
HashMap<String, Persona> personas = new HashMap<>();
Persona obj1 = new Profesor(…); …          Persona obj4 = new Alumno(…);
personas.put(obj1.getDni(), obj1); …       personas.put(obj4.getDni(), obj4);
//Buscamos una persona por su DNI
BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
System.out.println("\nIntroduce el dni de la persona a buscar:");
String dni = entrada.readLine();
Persona paux = personas.get(dni);
//Sacamos la clase para poder aplicarle sus métodos
String clase = paux.getClass().getSimpleName();
System.out.println("Clase: " + clase);
if (clase.equals("Alumno")) {
        Alumno alumno = (Alumno) paux;
        System.out.println("- Titulación: " + alumno.getTitulacion());
        System.out.println("- Universidad: " + alumno.getUniversidad());
} else if (clase.equals("Profesor")) {
        Profesor profe = (Profesor) paux;
        System.out.println("- Departamento: " + profe.getDepartamento());
        System.out.println("- Sueldo: " + profe.getSueldo());
}
```

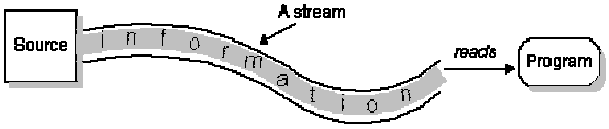


Librería I/O

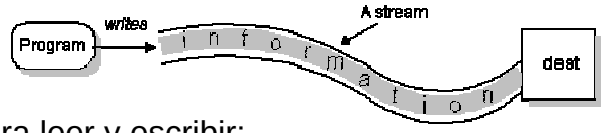
LIBRERÍA I/O

- Normalmente las aplicaciones necesitan **leer o escribir información desde o hacia una fuente externa de datos**.
- La información puede estar en cualquier parte, en un fichero, en disco, en algún lugar de la red, en memoria o en otro programa.
- También puede ser de cualquier tipo: objetos, caracteres, imágenes o sonidos.
- La comunicación entre el origen de cierta información y el destino se realiza mediante un **stream** (flujo) de información. Un stream es un objeto que hace de intermediario entre el programa y el origen o destino de la información.

- Para traer la información, un programa abre un stream sobre una fuente de información (un fichero, memoria, un socket) y lee la información, de esta forma:



- De igual forma, un programa puede enviar información a un destino externo abriendo un stream sobre un destino y escribiendo la información en este, de esta forma:

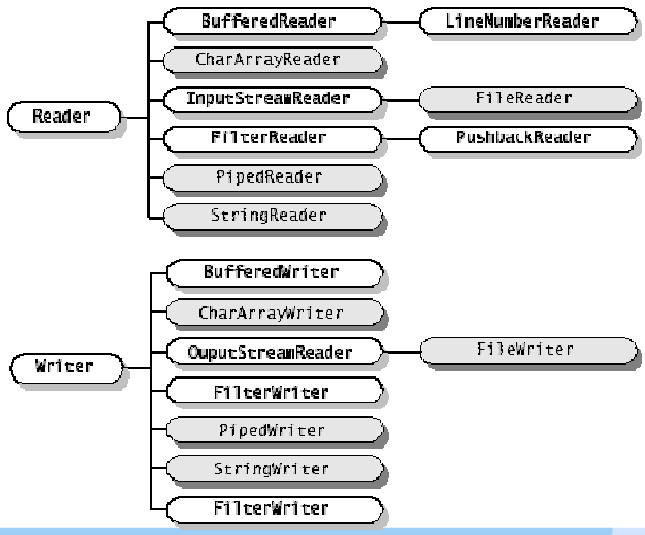


- Los algoritmos para leer y escribir:

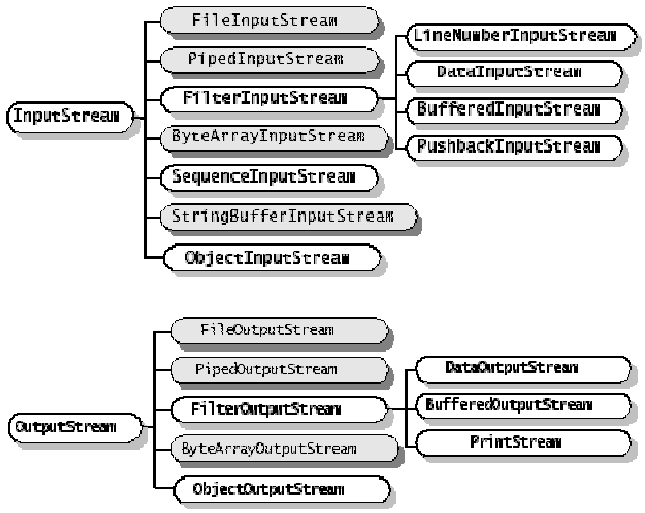
abrir un stream
mientras haya información
leer o escribir información
cerrar el stream

- El paquete *java.io* contiene una colección de clases stream que soportan estos algoritmos para leer y escribir. Estas clases están divididas en dos árboles basándose en los tipos de datos (caracteres o bytes) sobre los que opera.

- **Streams de Caracteres:**
  - Reader y Writer son las superclases abstractas para streams de caracteres en java.io. **Reader** proporciona el API y una implementación para readers (streams que leen caracteres de 16-bits) y **Writer** proporciona el API y una implementación para writers (streams que escriben caracteres de 16-bits).



- **Streams de Bytes:**
  - Los programas deberían usar los streams de bytes, descendientes de **InputStream** y **OutputStream**, para leer y escribir bytes de 8-bits. Estos streams se usan normalmente para leer y escribir datos binarios como imágenes y sonidos.



- **La clase File:**
  - Nos permite recuperar información acerca de un archivo o directorio.
  - Los objetos de la clase File no abren archivos ni proporcionan herramientas para procesar archivos. Se utilizan en combinación con objetos de otras clases de java.io para especificar los archivos o directorios que se van a manipular.
  - Ejemplo:

```
import java.io.File;
public class PruebaFile {
   //muestra información acerca de un fichero y un directorio
   public static void main(String[] args) {
      File fichero = new File("ejemplo.txt");
      if (fichero.exists() && fichero.isFile()) {
         System.out.println("\n- Información del fichero:");
         System.out.println("El fichero tiene el nombre: " + fichero.getName());
         System.out.println("El fichero tiene el path: " + fichero.getAbsolutePath());
         System.out.println("Longitud del fichero: " + fichero.length());
      }
      File directorio = new File("C:\\Program Files\\Java");
      if (directorio.exists() && directorio.isDirectory()) {
         String listado[] = directorio.list();
         System.out.println("\n- Listado del directorio:");
         for (int i = 0; i < listado.length; i++) { System.out.println(listado[i] + "\n"); }
      }
   }
}
```

FICHEROS DE TEXTO

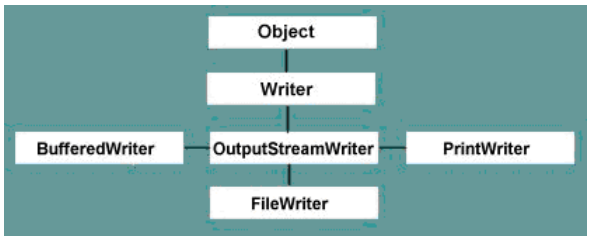
- **Lectura de un fichero de texto:**
  - Si necesitamos leer la información almacenada en un fichero de texto que contiene caracteres especiales tales como acentos y eñes debemos combinar las clases **FileInputStream, InputStreamReader** y **BufferedReader.**
  - Mediante la clase **FileInputStream** indicaremos el fichero a leer (es un stream de bytes).
  - La clase **InputStreamReader** se encarga de **leer bytes y convertirlos a carácter** según unas reglas de conversión definidas para cambiar entre 16-bit Unicode y otras representaciones específicas de la plataforma.
  - Mediante la clase **BufferedReader** leeremos el texto desde el **InputStreamReader** almacenando los caracteres leídos. Proporciona un buffer de almacenamiento temporal. Esta clase tiene el método *readLine()* para leer una línea de texto a la vez.

FICHEROS DE TEXTO

- **Lectura de un fichero de texto:**

```
import java.io.*;
public class LeeFicheroEspecial {
   public static void main(String[] args) {
      String cad;

      try {
         FileInputStream fis = new FileInputStream("ejemplo.txt");
         InputStreamReader isr = new InputStreamReader(fis, "ISO-8859-1");
         BufferedReader br = new BufferedReader(isr);
         while ((cad = br.readLine()) != null) {
            System.out.println(cad);
         }
         //Cerramos el stream
         br.close();
      } catch (IOException ioe) {
         System.out.println(ioe);
      }
   }
}
```

- **Escritura de un fichero de texto:**
  - Para crear un stream de salida, tenemos la clase **Writer** y sus descendientes.



  - La clase **PrintWriter** proporciona métodos que facilitan la escritura de valores de tipo primitivo y objetos en un stream de caracteres.
  - Los métodos principales que proporciona son ***print*** y ***println***. El método *println* añade una nueva línea después de escribir su parámetro.
  - La clase PrintWriter se basa en un objeto BufferedWriter para el almacenamiento temporal de los caracteres y su posterior escritura en el fichero.

- **Escritura de un fichero de texto:**

```
import java.io.*;
public class EscribeFichero {
    public static void main(String[] args)  {
        String cad1 = "Esto es una cadena.";
        String cad2 = "Esto es otra cadena.";

        try {
            PrintWriter salida
            = new PrintWriter(new BufferedWriter(new FileWriter("salida.txt")));
            salida.println(cad1);
            salida.println(cad2);

            //Cerramos el stream
            salida.close();
        } catch (IOException ioe) {
            System.out.println("Error IO: "+ioe.toString());
        }
    }
}
```

SERIALIZACIÓN DE OBJETOS

- Cuando ejecutamos una aplicación OO lo normal es crear **múltiples instancias de las clases** que tengamos definidas en el sistema. Cuando cerramos esta aplicación todos los objetos que tengamos en memoria se pierden.

- Para solucionar este problema los lenguajes de POO nos proporcionan unos mecanismos especiales para poder guardar y recuperar el estado de un objeto y de esa manera poder utilizarlo como si no lo hubiéramos eliminado de la memoria. Este tipo de mecanismos se conoce como **persistencia de los objetos**.

- En **Java** hay que implementar una interfaz y utilizar dos clases:
  - Interfaz Serializable (interfaz vacía, no hay que implementar ningún método)
  - Streams: ObjectOutputStream y ObjectInputStream.

- Por ejemplo: *class Clase implements Serializable*, a partir de esta declaración los objetos que se basen en esta clase pueden ser persistentes.

SERIALIZACIÓN DE OBJETOS

- ObjectOutputStream y ObjectInputStream permiten leer y escribir grafos de objetos, es decir, escribir y leer los bytes que representan al objeto. El proceso de transformación de un objeto en un stream de bytes se denomina **serialización**.
- Los objetos ObjectOutputStream y ObjectInputStream deben ser almacenados en ficheros, para hacerlo utilizaremos los streams de bytes FileOutputStream y FileInputStream, **ficheros de acceso secuencial**.
- Para serializar objetos necesitamos:
  - Un objeto FileOutputStream que nos permita escribir bytes en un fichero como por ejemplo:

    FileOutputStream fos = new FileOutputStream("fichero.dat");
  - Un objeto ObjectOutputStream al que le pasamos el objeto anterior de la siguiente forma:

    ObjectOutputStream oos = new ObjectOutputStream(fos);
  - Almacenar objetos mediante writeObject() como sigue:

    oos.writeObject(objeto);
  - Cuando terminemos, debemos cerrar el fichero escribiendo:

    fos.close();

SERIALIZACIÓN DE OBJETOS

- Los atributos **static** no se serializan de forma automática.
- Los atributos que pongan **transient** no se serializan.

- Para recuperar los objetos serializados necesitamos:
  - Un objeto FileInputStream que nos permita leer bytes de un fichero, como por ejemplo:

    FileInputStream fis = new FileInputStream("fichero.dat");

  - Un objeto ObjectInputStream al que le pasamos el objeto anterior de la siguiente forma:

    ObjectInputStream ois = new ObjectInputStream(fis);

  - Leer objetos mediante readObject() como sigue:

    (ClaseDestino) *ois.readObject();*

    Necesitamos realizar una conversión a la "ClaseDestino" debido a que Java solo guarda Objects en el fichero.

  - Cuando terminemos, debemos cerrar el fichero escribiendo:

    fis.close();

SERIALIZACIÓN DE OBJETOS

- Ejemplo de serialización de objetos de tipo persona 1:

```
public class Persona implements Serializable { ... }
/*************************************************************************************************/
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1"); ...
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4");
/*************************************************************************************************/
//Serialización de las personas 1
FileOutputStream fosPer = new FileOutputStream("copiasegPer.dat");
ObjectOutputStream oosPer = new ObjectOutputStream(fosPer);
oosPer.writeObject(obj1); ... oosPer.writeObject(obj4);
/*************************************************************************************************/
//Lectura de los objetos de tipo persona
FileInputStream fisPer  = new FileInputStream("copiasegPer.dat");
ObjectInputStream oisPer = new ObjectInputStream(fisPer);
try {
                while (true) {
                        Persona per = (Persona) oisPer.readObject();
                        System.out.println (per.toString());
                }
} catch (EOFException e) {
        System.out.println ("Lectura de los objetos de tipo Persona finalizada");
}
fisPer.close();
```

SERIALIZACIÓN DE OBJETOS

- Ejemplo de serialización de objetos de tipo persona 2:

```
public class Persona implements Serializable { ... }
/*************************************************************************************************/
Persona obj1 = new Persona( "06634246S", "Javier", f1, "calle1"); ...
Persona obj4 = new Persona( "15664386T", "Carmen", f4, "calle4");
//Introducimos los objetos en una tabla hash
HashMap<String, Persona> personas = new HashMap<>();
personas.put(obj1.getDni(), obj1); ...
personas.put(obj4.getDni(), obj4);
/*************************************************************************************************/
//Serialización de la tabla hash personas
FileOutputStream fosPer = new FileOutputStream("copiasegPer.dat");
ObjectOutputStream oosPer = new ObjectOutputStream(fosPer);
oosPer.writeObject(personas);
fosPer.close();
/*************************************************************************************************/
//Lectura de los objetos de tipo persona a través de la tabla hash personas
FileInputStream fisPer = new FileInputStream("copiasegPer.dat");
ObjectInputStream oisPer = new ObjectInputStream(fisPer);
try {
        while (true) {
                personas = (HashMap) oisPer.readObject();
                System.out.println (personas.toString());
        }
} catch (EOFException e) {
        System.out.println ("Lectura de los objetos de tipo Persona finalizada");
}
fisPer.close();
```


EJEMPLO USO DE COLECCIONES Y PERSISTENCIA

- El siguiente ejemplo utiliza un ArrayList para gestionar objetos de tipo Persona en un censo universitario con profesores y alumnos. También se utiliza la persistencia para almacenar los datos cuando la aplicación se cierra y la generación de ficheros de tipo texto.