Grado en Ingeniería Informática
Grado en Matemáticas e Informática

Asignatura: PROGRAMACIÓN II

Clases contenedoras

Clara Benac Earle, Manuel Collado,
Ángel Lucas González Martínez
Jaime Ramírez Rodríguez
Guillermo Román

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Noviembre 2013

Clases contenedoras

Colecciones de datos

Estructuras de datos

Contenedores

Colecciones de datos

- En la mayoría de las aplicaciones no se manejan sólo datos individuales, sino colecciones de ellos.
- Para almacenar la colección de datos hay que plantear una forma de organización, que será la estructura de datos.
- La organización de los datos y la forma de manipularlos puede depender de ciertas características de esos datos. Por ejemplo:
  - Si existe o no un orden preestablecido entre los valores de los datos, que debe ser tenido en cuenta.
  - Si la colección como tal contiene una copia de cada dato o sólo una referencia a los datos.

Estructuras de datos

- Existe una gran variedad de estructuras de datos interesantes, que pueden usarse en muchas aplicaciones.
- Cada estructura tiene características particulares que la hacen adecuada en ciertos casos y en otros no.
  - Estructuras lineales o no lineales
  - Ordenadas o no ordenadas
  - Acceso por posición o por valor del dato
  - Iterables o no iterables
  - Acotadas o no acotadas
- Cada estructura concreta tendrá una combinación de estas u otras características.

Clases contenedoras

- Los contenedores son clases cuyo contenido es una colección de datos, con una estructura determinada.
- Además de almacenar los datos de la colección, estas clases ofrecen un conjunto concreto de operaciones para manipular la colección.
- Puesto que las clases son un mecanismo de abstracción, se dice también que las clases contenedoras implementan Tipos Abstractos de Datos.
- Cada contenedor implementa una estructura de datos concreta y un conjunto concreto de operaciones.
- Muchos de estos contenedores tienen nombre propio, y están disponibles en las librerías asociadas a cada lenguaje de programación.

Estructuras lineales y no lineales

- Estructuras de datos lineales
  - Contienen una secuencia de elementos. La posición de cada uno es significativa:

$$\mathbf{e_1\ e_2\ e_3\ \ldots\ e_n}$$

  - Genéricamente se llaman *listas* (**ojo, nombre ambiguo**)
- Estructuras de datos no lineales
  - La estructura no es una simple secuencia, o bien la posición no es significativa:
  - Ejemplos:
    - árbol, grafo, tabla (*map*), conjunto, etc.

Contenedores lineales

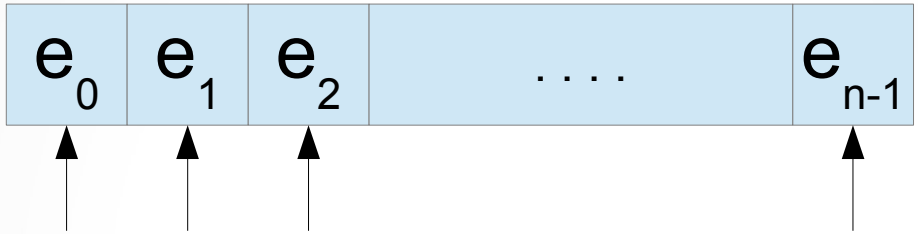
- Todos contienen una secuencia de elementos. Se distinguen por las operaciones concretas para manipular la colección.
- El acceso por posición, para insertar, modificar o extraer elementos, puede ser:
  - Por el principio
  - Por el final
  - Por ambos extremos
  - En cualquier posición
  - En una posición actual que se puede desplazar (cursor)
- Las combinaciones habituales se recogen en la siguiente tabla

Contenedores lineales

Insertar / **Extraer**	Cualquier posición	Posición actual	Ambos extremos	Principio	Final
Cualquier posición	Vector Lista-directa				
Posición actual		Lista-secuen.			
Ambos extremos			Cola doble		
Principio				Pila	Cola
Final				*Cola inversa*	*Pila inversa*

Vectores

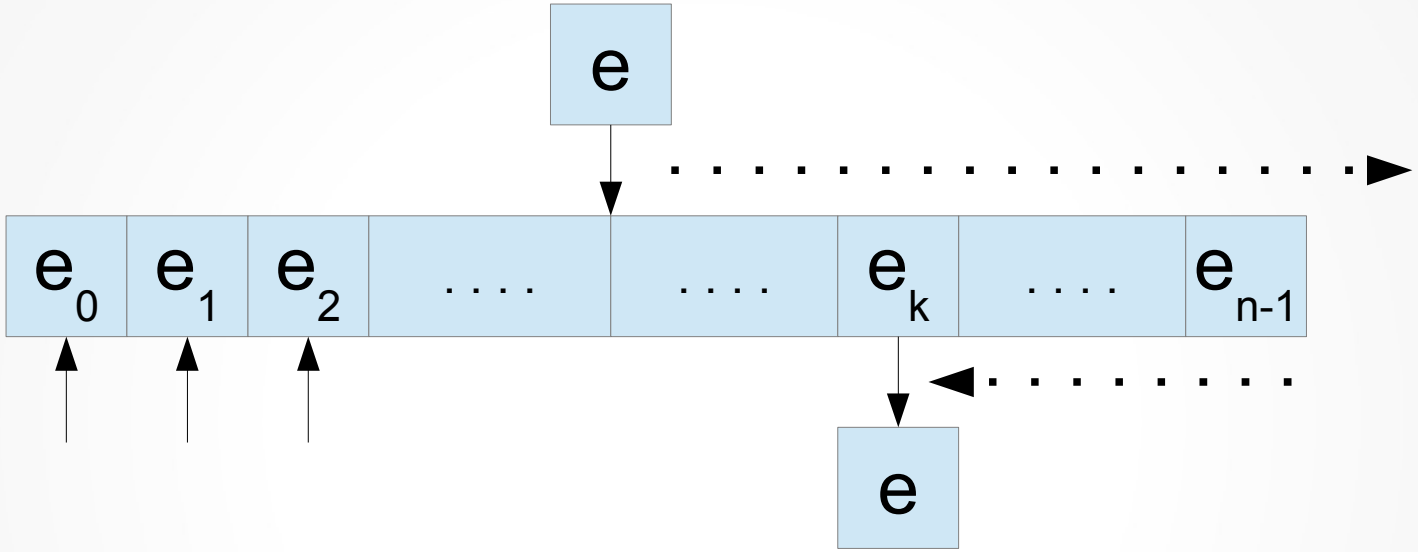
- El contenedor vector es esencialmente el *array*, disponible directamente en el lenguaje de programación.



- En Java los índices se numeran desde cero, aunque resultaría más intuitivo numerarlos desde uno.
- Se puede acceder directamente al elemento en cualquier posición, y recuperar su valor o modificarlo.
- El número de elementos es fijo. No se añaden ni retiran elementos.

Lista de acceso directo

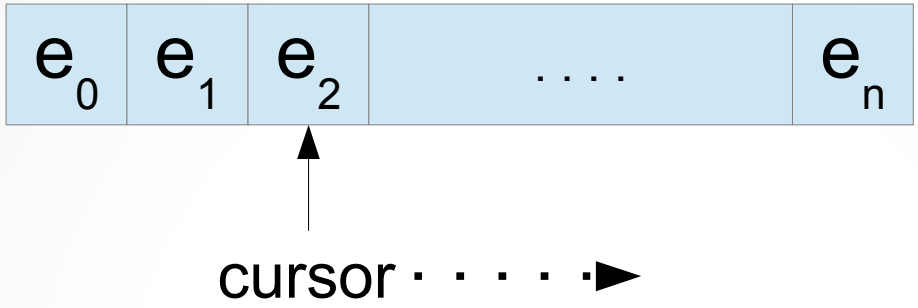
- Es como un vector de tamaño variable, porque además se pueden insertar o suprimir elementos.



- Los elementos posteriores se mueven adelante o atrás para hacer sitio o rellenar el hueco.
- Se conoce también como lista-vector (*ArrayList)*.

Lista-secuencial

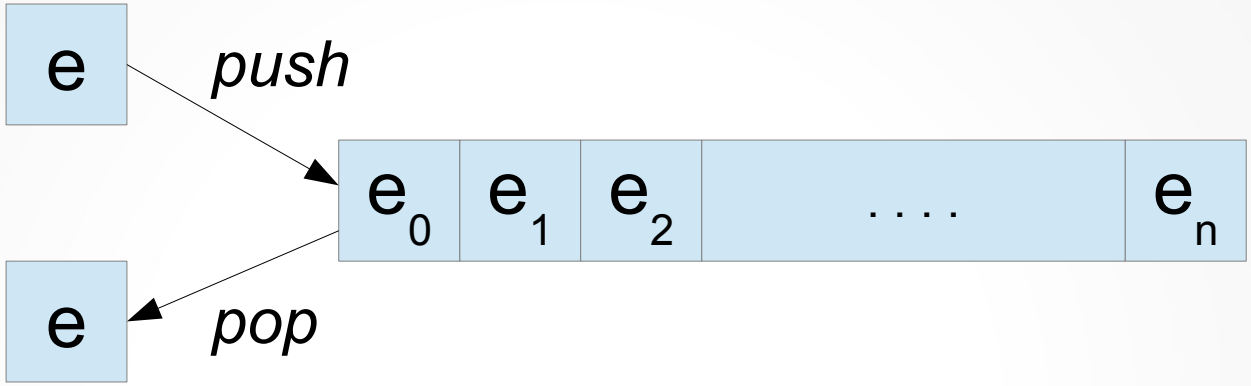
- Es una secuencia que permite acceder a los elementos de uno en uno, mediante un cursor.



- El cursor se puede poner al principio y hacerlo avanzar. A veces también retroceder.
- Se puede recuperar el elemento al que señala el cursor, o modificarlo.
- Se pueden añadir o eliminar elementos en la posición del cursor, moviendo los posteriores.
- Se pueden añadir elementos al final sin usar el cursor.

Pila (*stack*)

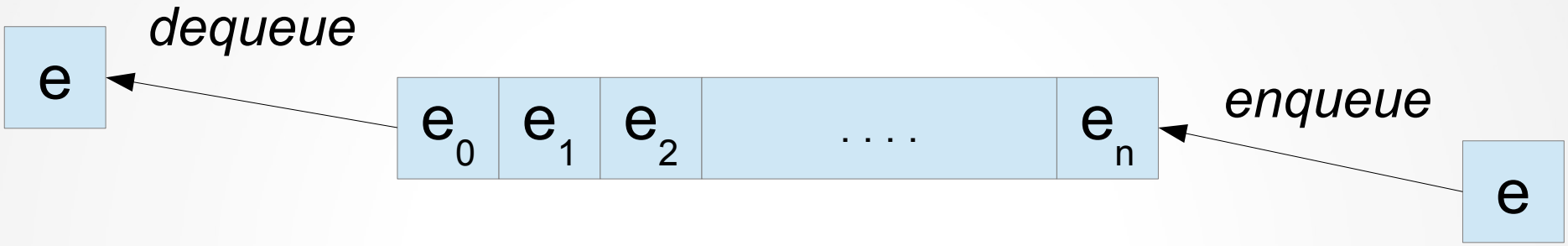
- Es una secuencia a la que se accede sólo por el principio.



- Se puede añadir un nuevo elemento delante de los demás (*push*).
- Se puede retirar el primer elemento (*pop*), sacándolo de la secuencia.

Cola (*queue*)

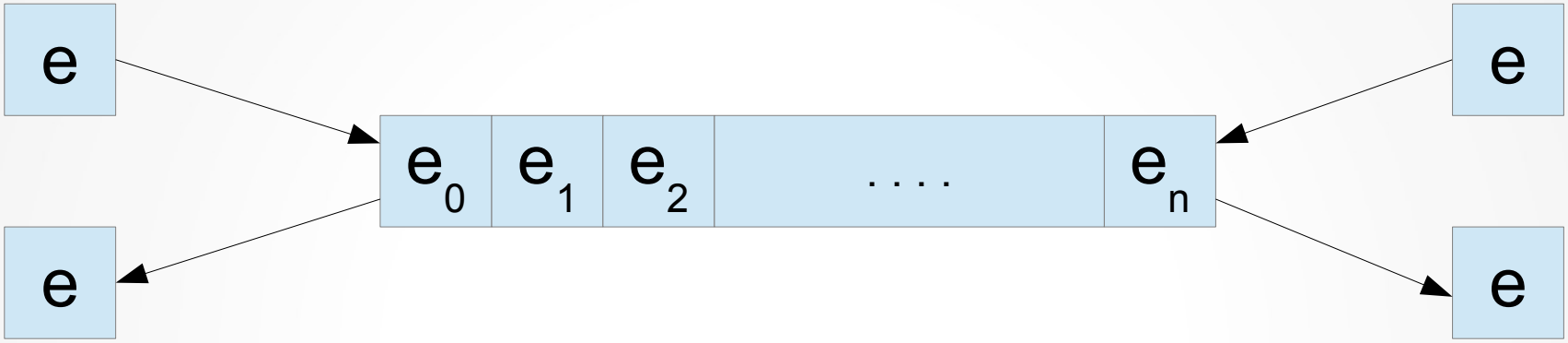
- Es una secuencia con acceso especial por cada extremo.



- Se puede añadir un nuevo elemento detrás de los demás (*enqueue*).
- Se puede retirar el primer elemento (*dequeue*), sacándolo de la secuencia.

Cola doble (*deque*)

- Es una secuencia con acceso general por ambos extremos.



- Se puede añadir un nuevo elemento delante o detrás de los demás.
- Se puede retirar el primer elemento o el último, sacándolo de la secuencia.
- *Deque: double-ended-queue*

Contenedores lineales ordenados

- Tienen la restricción de que los elementos están almacenados en orden.
- Sólo hay una forma de inserción, por valor. El nuevo elemento se coloca automáticamente en su lugar.
- La extracción, en cambio, se puede hacer por posición o por valor.

Extraer				
Cualquier posición	Posición actual	Ambos extremos	Principio	Final
Lista-orden. directa	Lista-orden. secuencial		Cola con prioridad	*Cola con prior. inv.*

Contenedores lineales acotados

- Tienen una capacidad limitada, definida habitualmente en el momento de crearlos.
- El tamaño puede ser el máximo número de elementos o el espacio de memoria reservado.
- Si se intenta insertar más elementos se produce una excepción.
- Casos concretos:
  - Vector: Acotado.
  - Lista de acceso directo: No acotada.
  - Lista secuencial: No acotada.
  - Pila: a veces se usan pilas acotadas.
  - Cola: la cola acotada se suele denominar *buffer*.

Contenedores lineales iterables

- Permiten recorrer uno a uno los elementos mediante un iterador.
- El iterador se puede colocar al principio y hacerlo avanzar.
- A veces también se permite colocarlo en determinada posición, y avanzar o retroceder.
- Casos concretos:
  - Vector: implícitamente iterable.
  - Lista de acceso directo: implícitamente iterable.
  - Lista secuencial: implícitamente iterable.
  - Pila: requiere iterador adicional.
  - Cola: requiere iterador adicional.

Documentación de clases

Documentación del código con Javadoc

Documentación con *Javadoc*

- Mediante comentarios entre **/** ... */**
  - El comentario se pone inmediatamente delante del elemento documentado.
  - Se pueden utilizar etiquetas (*tags*) predefinidas *@xxx* para identificar ciertos tipos de información.
  - Se permite la inserción de código HTML.
- El formato general de un comentario Javadoc es:

```
/**
 * Descripción general
 *
 * @xxx texto informativo de etiqueta de sección ...
 * @yyy texto informativo con {@zzz texto en línea} ...
 */
public Elemento java( documentado ) { ... }
```

Documentación con *Javadoc*

- Es útil para documentar las clases como parte del propio código.
- Especialmente interesante para documentar la interfaz:
  - Comentarios generales sobre la clase y los métodos.
  - PRE y POST de cada método.
- La herramienta javadoc extrae los comentarios y las cabeceras de los elementos documentados, y genera un documento de referencia:
  - Formato: páginas HTML enlazadas.
  - No hace falta mirar el código para conocer los servicios disponibles, sino que basta con leer las páginas HTML

Documentación con *Javadoc*

- Se puede encontrar más información en:

  http://download.oracle.com/javase/1.5.0/docs/guide/javadoc/index.html

- Para generar la documentación se utiliza la orden:

  javadoc –author -d docs –version –private -link http://java.sun.com/j2se/1.5.0/docs/api EjemploComentarios.java

- En eclipse: Project ⇒ Generate Javadoc

Comentarios para documentación

- Etiquetas de propósito general, para cualquier elemento:
  - **@see**: permite poner referencias en la sección *See Also*
  - **{@link** *enlace***}**: similar a *see*. El enlace es parte de un párrafo y no de una sección
- Documentación de una clase:
  - Después de import e inmediatamente antes de class.
  - Algunas etiquetas posibles:
    - **@author**: Indica el autor o autores
    - **@version**: Permite indicar la versión de la clase

Comentarios para documentación

- Documentación de un método:
  - Inmediatamente antes del método
  - Se pueden utilizar las etiquetas:
    - **@param** *nombre* significado del argumento ...
    - **@return** significado del valor de retorno ...
    - **@throws** *Excepción* porqué y cuándo se lanza ...
  - Conviene incluir la ***precondición*** y la ***postcondición***
    - No hay etiquetas predefinidas para eso

Condiciones PRE y POST

- Los métodos de las clases deben tener especificada su *precondición* y su *postcondición*.
  - Sólo los *getters* y *setters* pueden no tener documentación, aunque es conveniente.
- Como se puede incluir código HTML lo aprovechamos
  - ... ⇒ Para poner en negrita **PRE** o **POST**.
  - ... o
 ⇒ Para que incluya saltos de línea.
- La documentación queda mucho más clara.

Ejemplo de método (I)

```
public class Lista {
  ....
  /**
   * Retorna el primer elemento de la lista, sin
   * alterar la lista
   *
   * <p><b>PRE:</b> La lista no está vacía</p>
   * <p><b>POST:</b> Retorna una referencia al elemento
   *                 que ocupa la primera posición de la lista</p>
   * @return retorna el primer elemento de la lista
   * @throws ExcepcionListaVacia si se viola la precondición
   */
  public Informacion primero() throws ExcepcionListaVacia {
    ....
  }
  ....
}
```

Ejemplo de método (II)

Method Detail

primero

```
public Lista.Informacion primero()
                          throws ExcepcionListaVacia
```

Retorna el primer elemento de la lista, sin alterar la lista

PRE: La lista no está vacía

POST: Retorna una referencia al elemento que ocupa la primera posición de la lista

Returns:

retorna el primer elemento de la lista

Throws:

`ExcepcionListaVacia` - si se viola la precondición

Clases contenedoras

Contenedor Pila

Leer datos y escribirlos invertidos

- Un ejemplo de programa: Leer de la entrada estándar números enteros separados por espacio e imprimirlos en orden inverso.
  - Entrada: 12 13 14 15 16 17 18
  - Salida: 18 17 16 15 14 13 12
- Hace falta almacenar todos los valores antes de imprimirlos.
- Primera aproximación con un Vector (*array*)
  - ¿Limitaciones?
  - Mejor pensar en otra cosa ...

Solución: Contenedor Pila

- También llamada estructura LIFO (*Last In First Out*).
- Es la abstracción de un contenedor en el que el último elemento insertado, es el primero que se extrae, p.ej. un tubo de pastillas.
- Las operaciones típicas que se consideran son: *el constructor de pila vacía*, *apilar*, *desapilar*, *getCima*, y *esVacia*.
- Esta definición del contenedor no establece un tamaño máximo ⇒ tamaño potencialmente ilimitado

Implementación de una Pila en Java

```
public class PilaEnteros {
  ....
  /*--- Constructores ---*/
  public Pila()
    {....} // crear pila vacía

  public void apilar(int elemento)
    {....} // añadir un elemento

  /*--- Observadores ---*/
  public boolean estaVacia ()
    {....} // ¿está vacía?

  public int getCima ()
    throws ExcepcionPilaVacia
    {....} // observar la cima
```

```
  /*--- Modificadores ---*/
  public void desapilar()
    throws ExcepcionPilaVacia
    {....} // retirar la cima

  public void vaciar()
    {....} // retirar todos

  public int sacarCima()
    throws ExcepcionPilaVacia
    {....} // retirar la cima

} // PilaEnteros
```

Otro ejemplo: Invertir un fichero

- El programa acepta un argumento en la linea de comandos que indica el fichero a invertir (si no hay argumento se procede a la lectura desde la entrada estándar). El fichero invertido (orden de las líneas) se vuelca en la salida estándar.
- Después de desarrollar el programa nos damos cuenta de que hemos tenido que **duplicar**, completamente a mano, el mismo código anterior para las pilas.
- ¿Qué nos ofrecen los lenguajes de programación para resolver este problema?
- **Polimorfismo paramétrico** (*genéricos* en Java).

Una clase genérica de pilas

- Tomamos una de las clases no genéricas de pilas (*PilaEnteros*) y la modificamos hasta conseguir que sea una clase genérica:
  - Parametrizamos o abstraemos el tipo:
    - sustituyendo cada ocurrencia del tipo concreto (*int*) por un tipo genérico (*Informacion*), y
    - generalizando la definición de las clases auxiliares que utilicen el tipo concreto (*int*).
  - Actualizamos el código del cliente (leer datos y escribirlos invertidos) sustituyendo *PilaEnteros* por *Pila<Integer>*.

Contenedor Pila genérica en Java

```
public class Pila<Informacion> {
  ....
  /*--- Constructores ---*/
  public Pila()
    {....} // crear pila vacía
  public void apilar
    (Informacion elemento)
    {....} // añadir un elemento

  /*--- Observadores ---*/
  public boolean estaVacia ()
    {....} // ¿está vacía?

  public Informacion getCima ()
    throws ExcepcionPilaVacia
    {....} // observar la cima
```

```
  /*--- Modificadores ---*/
  public void desapilar()
    throws ExcepcionPilaVacia
    {....} // retirar la cima

  public void vaciar()
    {....} // retirar todos

  public Informacion sacarCima()
    throws ExcepcionPilaVacia
    {....} // retirar la cima

} // PilaEnteros
```

Clases contenedoras

Ejemplos de uso de Contenedores

Paréntesis “balanceados” (I)

- Dada una expresión en la que se usan paréntesis determinar si sus paréntesis son correctos.
- Ejemplo: sólo un tipo de paréntesis

Balanceados	No balanceados
A * (B + C / (X + Y))	A * B + C / (X + Y))
(E + F) * (X - Y)	E + F) * (X - Y

- Con un contador es suficiente.
- Pero, ¿qué ocurre con expresiones con más de un tipo de paréntesis?

Balanceados	No balanceados
A * [B + C / (X + Y)]	A * {B + C / (X + Y)]
([{([])}])	{([({[]})]}

Paréntesis “balanceados” (II)

- Se utiliza una pila para “recordar” el último (y anteriores) paréntesis abiertos.
- Cuando se encuentra un paréntesis cerrado se compara con la cima de la pila para comprobar que son compatibles y desapilamos.
- El resultado es incorrecto:
  - si intentamos desapilar y la pila está vacía,
  - no son compatibles o
  - al final la pila no está vacía

Expresiones en notación postfija (I)

- La notación aritmética *estándar* en matemáticas (y, actualmente, también de los lenguajes de programación) es la notación *infija*:

32 + 43 * 12 + 78 / 2 / 5 - 4
100 / 4 / 2

- La notación infija obliga a introducir reglas de precedencia de operadores y asociación de los mismos para evitar el uso de paréntesis.
- Aun así necesitamos los paréntesis para poder *evitar* las precedencias predefinidas:

(32 + 43) * (12 + 78) / 2 / (5 - 4)
100 / (4 / 2)

Expresiones en notación postfija (II)

- La notación *postfija* (también denominada notación polaca inversa — *RPN*) es una notación alternativa a la estándar que posee un par de propiedades muy interesantes para la informática:
  - No necesita uso de paréntesis.
  - Las reglas para su evaluación son muy sencillas.
- En RPN el operador se escribe inmediatamente después de los operandos y la precedencia la indica el orden de aparición de los operandos y de los operadores.
- Pensemos en cómo se evalúan las expresiones:

(5 + 7) * (6 - 2)
5 + 7 * 6 - 2

Expresiones en notación postfija (III)

- Y veamos lo fácil que sería automatizar si utilizamos RPN:

5 7 + 6 2 - *

5 7 6 * + 2 -

- Mientras se lean operandos se apilan en una pila.
- Cuando se encuentra un operador se sacan dos elementos de la pila, se realiza la operación con ellos y se deja el resultado en la pila.
- El resultado final está en la cima de la pila.

Ejercicios

- Calculadora RPN

```
public int evaluarRPN (String expresion)
  throws ErrorExpresionNoValida ...
private boolean isOperadorValido (char operador) ...
private int aplicar (int op1, int op2, char operador)
  throws ErrorOperadorNoValido ...
```

- Main

```
CalculadoraRPN calculadora = new CalculadoraRPN();
String linea = Consola.leerLinea();
while (!linea.equals("")) {
  try {
    System.out.println(calculadora.evaluarRPN(linea));
  } catch (ExpresionNoValidaException e) {...}
  linea = Consola.leerLinea();
}
```

Clases contenedoras

Contenedor Cola

Ejemplo: caja del supermercado

- El ejemplo clásico de uso de colas
- Los clientes van llegando a la cola y se sitúan detrás del último
  - Se van atendiendo en función del orden de llegada
- Comprobar la diferencia de funcionamiento con la pila
  - Que tiene la estructura contraria

Contenedor Cola

- También llamada estructura FIFO (*First In First Out*).
- Es la abstracción de un contenedor en el que el primer elemento insertado es el primero que se extrae, p.ej. la cola de la taquilla del cine.
- Las operaciones típicas que se consideran son: *el constructor de cola vacía*, *insertar*, *borrar*, *getPrimero*, y *esVacia*.
- Esta definición del contenedor no establece un tamaño máximo ⇒ tamaño potencialmente ilimitado

Ejemplo: mezclar series ordenadas

```
public class Mezclador {
    private Cola<Integer> serie1, serie2, mezcla;
    ....
    public void mezclar() {
      try {
        mezcla = new Cola<Integer>();
        while (!serie1.estaVacia() && !serie2.estaVacia()) {
          if (serie1.getPrimero() <= serie2.getPrimero()) {
              mezcla.insertar(serie1.sacarPrimero());
          } else {
              mezcla.insertar(serie2.sacarPrimero());
          }
        }
        while (!serie1.estaVacia()) {
            mezcla.insertar(serie1.sacarPrimero());
        }
        while (!serie2.estaVacia()) {
            mezcla.insertar(serie2.sacarPrimero());
        }
      } catch (ExcepcionColaVacia e) {
         e.printStackTrace();
      }
    }
```

Clases contenedoras

Contenedor Lista

Contenedor Lista

- Ya hemos estudiado algunos contenedores (Pilas, Colas) que pueden ser utilizados para almacenar colecciones de datos.
- Ahora bien, un programa cliente no puede recorrerlas sin destruirlas.
  - Por ejemplo: para buscar un elemento, etc.
- El contenedor Lista va a admitir recorridos que no modifiquen la estructura.

Contenedor Lista

- Representa una secuencia de valores que admite habitualmente las siguientes operaciones:
  - *Constructor de lista vacía*, *estaVacia*, *getPrimero*, *insertar* (al principio), *borrar* (el primero), *concatenar*.
- Nuestra implementación del contenedor Lista incluirá algunas operaciones más que facilitarán su utilización: *equals*, *vaciar*, etc.

Implementación del contenedor Lista

- Clase genérica ListaIterable<Informacion> basada en una cadena enlazada.
- **Recorrido de una lista:** se va a utilizar un iterador interno (atributo actual).
  - Un **iterador** es una referencia a un elemento de la lista que podemos mover a lo largo de la lista para acceder a cada uno de los elementos.
  - **Operaciones necesarias:** alPrincipio(), siguiente(), haySiguiente(), getActual()

```
// PRE: lista no está vacía

lista.alPrincipio();
while (lista.haySiguiente()) {
   System.out.println(lista.getActual());
   lista.siguiente();
}
System.out.print(lista.getActual());
```