

## Módulo 6 – Programación con Java

### Índice

#### Conociendo Java

¿Qué es Java? .....	2
Programar con Apache NetBeans .....	3
¿En Java cómo se escribe el código? .....	5

#### Estructuras Estáticas - Arreglos (Arrays)

Vectores y Matrices .....	11
---------------------------	----

#### Colecciones

Estructuras Dinámicas Collections .....	19
---	----

#### Excepciones

Manejo de Excepciones.....	27
Métodos en Java .....	31

## ¿Qué es Java?

Java es una tecnología que se usa para el desarrollo de aplicaciones multiplataforma. Esta tecnología se compone de un lenguaje de programación y una plataforma informática que se utiliza para el desarrollo de aplicaciones web, aplicaciones de escritorio o en aplicaciones móviles. Fue comercializada por primera vez en 1995 por Sun Microsystems.

### Hablemos de la Plataforma Java

La plataforma Java es el nombre de un entorno o plataforma de computación originaria de Sun Microsystems, capaz de ejecutar aplicaciones desarrolladas usando el lenguaje de programación Java. En este caso, la plataforma no es un hardware específico o un sistema operativo, sino más bien una máquina virtual encargada de la ejecución de las aplicaciones, y un conjunto de bibliotecas estándar que ofrecen una funcionalidad común. Esta plataforma se instala en los sistemas operativos dejando todo listo para que cualquier aplicación desarrollada con el lenguaje java pueda ejecutarse. Para que los programadores puedan comenzar a trabajar es necesario instalar el Java Development Kit o Java JDK en el sistema operativo que utilizamos. Nosotros nos centraremos más adelante en la instalación de Java JDK.

## ¿Qué necesitas saber o hacer con Java JDK?

Por ahora solo necesitamos saber instalarlo y después poco a poco iremos conociendo como son las librerías que tiene dentro.

### Hablemos del Lenguaje de Programación Java

Java también es un lenguaje de programación orientado a objeto que se utiliza para programar aplicaciones de escritorio, sitios web o móviles. Las aplicaciones desarrolladas con el lenguaje Java probablemente no funcionen si no está instalada la plataforma Java o llamada máquina virtual de java. Desde computadoras portátiles hasta servidores, consolas para juegos, teléfonos móviles o Internet, Java está en todas partes, si creaste una aplicación con este lenguaje que está ejecutándose en Windows y necesitas que se ejecute en Linux, no necesitas recompilar ni reescribir el código, para que funcione en el otro sistema operativo solo te aseguras de que tenga la máquina virtual de java instalada y listo.

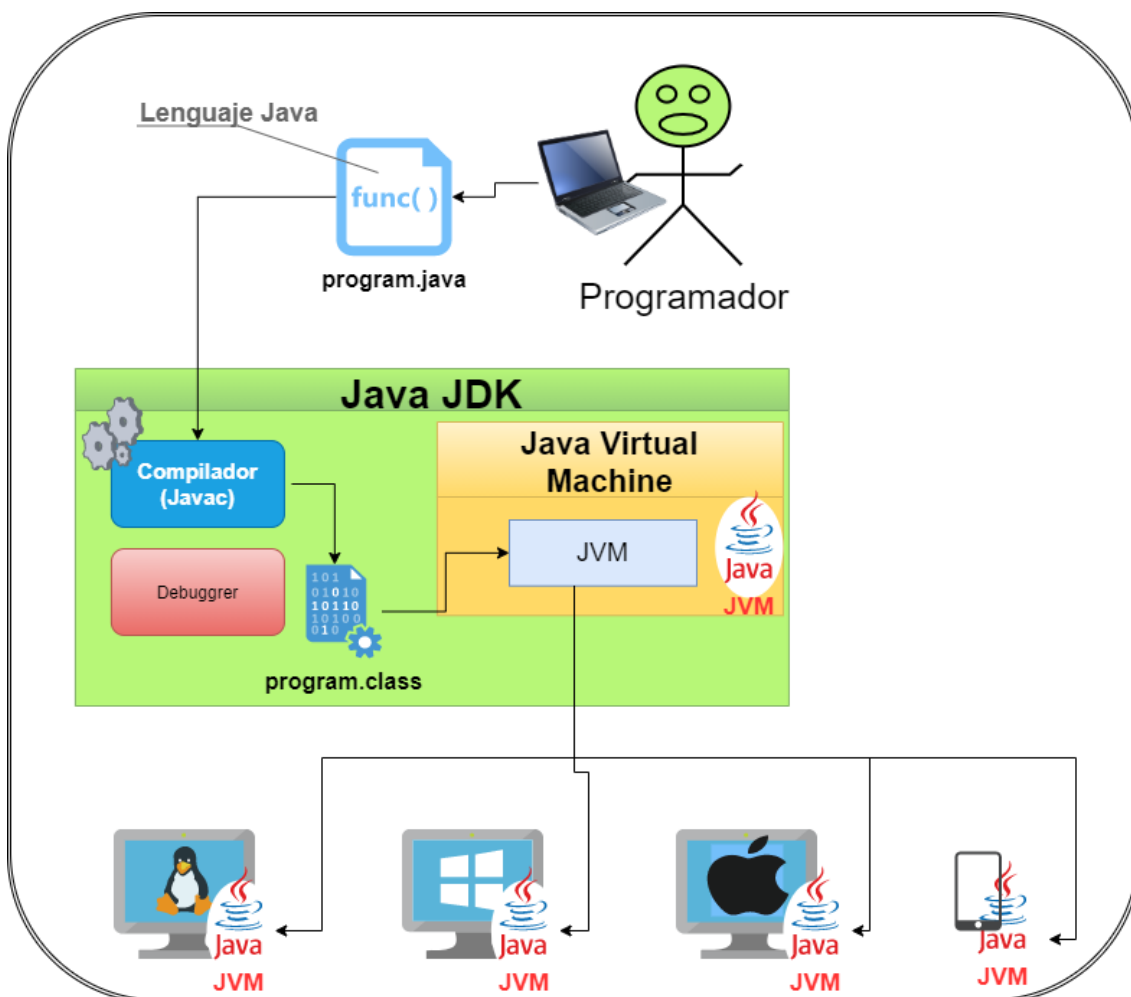
Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web. Nosotros conoceremos más a fondo este lenguaje de programación para poder usarlo.

## ¿Qué necesitas saber o hacer con el Lenguaje Java?

Debemos aprender cómo se escribe el código para que funcione y haga lo que necesitamos, es decir, aprender las reglas del lenguaje o sintaxis para que compile sin errores.

### ¿Cómo es que Java funciona en todas las plataformas?

Para responder esa pregunta te dejamos una grafico para que veas cómo ocurre, esto es posible por la Java Virtual Machine:



## Programar con Apache NetBeans

### ¿Qué es IDE?

Un IDE es la abreviación del inglés **I**ntegrated **D**evelopment **E**nvironment (**IDE**), la traducción en castellano es: **E**ntorno de **D**esarrollo **I**ntegrado, es una aplicación que proporciona servicios integrales para facilitarle al desarrollador el proceso de programar, detectar errores en el código, facilitar acceder a la documentación de las librerías, entre muchas otras funciones.

Normalmente, un IDE contiene un editor de código fuente, ventanas auxiliares para ver qué pasa en el código y un depurador que nos da más detalles cuando ocurre un error. La mayoría de los IDE tienen auto-completado inteligente de código (IntelliSense). Algunos IDE contienen un compilador, un intérprete, o ambos, tales como NetBeans y Eclipse. Nosotros nos enfocaremos en trabajar con el IDE

de **Apache NetBeans** para trabajar con Java en tu computadora, igualmente te recomendamos que todas las prácticas de código hagas que funcionen en NetBeans de tu computadora.

### ¿Qué necesitas saber o hacer con el NetBeans?

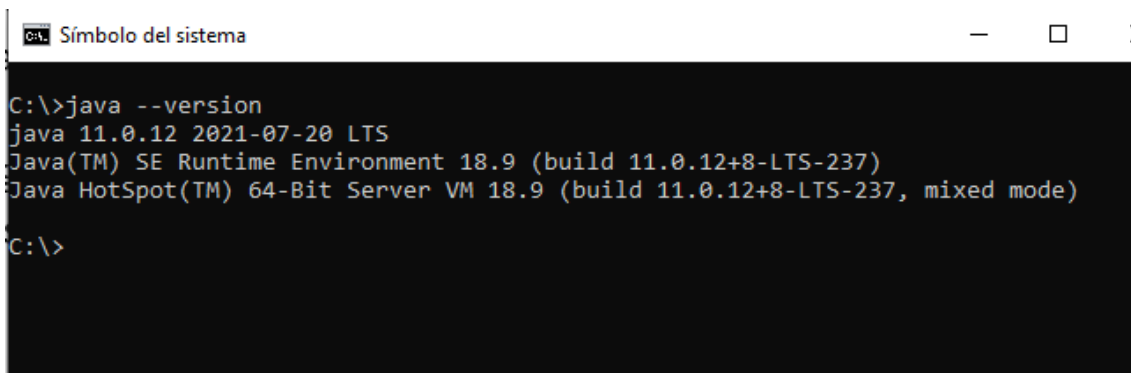
Deberás ir investigando poco a poco todo lo que ofrece el IDE NetBeans que puedan facilitarte las cosas al momento de programar, como por ejemplo:

- Cómo crear un proyecto.
- Cuáles son las ventanas más importantes.
- Cómo ejecutar el programa en modo Debug.
- Cómo interrumpir el código mientras está ejecutando.
- En qué parte del IDE muestra los errores.
- Accesos directos o teclas rápidas (combinaciones de tecla) para programar más rápido.
- Descubrir cómo te ayuda a aprender más rápido el lenguaje.

### Preparemos tú entorno de desarrollo

El plan para dejar tu computadora lista para programar en java es:

1. Instalar Java JDK
  1. Buscar en google “*descargar Java JDK*”, entrar al sitio oficial de java y buscar el link de descarga JDK.
  2. Crear un usuario en el sitio de Java para poder descargar el Java JDK.
  3. Buscar la última versión estable LTS y descargar el Java JDK para el sistema operativo que tengas.
  4. Una vez descargado el Java JDK, instalarlo en tu computadora.
  5. Comprueba si Java JDK quedo bien instalado con el comando `java --version`, como se ve en la siguiente imagen:



```
C:\>java --version
java 11.0.12 2021-07-20 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.12+8-LTS-237)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.12+8-LTS-237, mixed mode)
C:\>
```

2. Instalar NetBeans

1. Buscar en google “*descargar apache netbeans lts*”, entrar al sitio oficial para la descarga.
2. Asegúrate de descargar la versión estable o LTS.
3. También asegúrate de descargar el instalador correcto para tu sistema operativo.
4. Abrimos el Apache NetBeans y lo exploramos.

Una vez que hayamos explorado las ventanas y creado un proyecto en Apache NetBeans, veamos cómo el IDE te va a ayudar no solo a escribir el código sino también a aprender más sobre Java mostrándote documentación además de recordarte cómo se escribe el código. Así que cuanto más aprendas a usar el IDE, más fácil se te hará programar y aprender.

Alt-Insert: Genera código dependiendo de las posibilidades desplegando un menú de opciones.

Ctrl-Shift-I: Agrega las sentencias de importación de clases que hagan falta.

Alt-Shift-I: Arregla la importación de clases seleccionada.

Alt-Shift Left/Right/Up/Down: Mueve la línea de código en la dirección de las teclas.

Ctrl-Shift-R: Selecciona código de forma rectangular.

Ctrl-Shift-C: Agrega dobles diagonales para comentarios de una sola línea.

Ctrl-E: Borra la línea en la que está posicionada el cursor.

ifelse y a continuación la tecla Tab: NetBeans completará un bloque if con su else.

trycatch y a continuación la tecla Tab: NetBeans completará un bloque try-catch.

for y a continuación la tecla Tab: Obtienes un bloque de código para for listo para usarse.

fori y a continuación Tab: Crea el ciclo sobre un arreglo con índice.

## ¿En Java cómo se escribe el código?

### Sintaxis básica

Java es un lenguaje por lo cual hay que aprender cómo se escribe el código, para comenzar te dejamos una imagen de resumen y después desarrollaremos cada tema.

## SINTAXIS BÁSICA DE JAVA

El diagrama muestra un código de ejemplo en Java con varias anotaciones explicativas:

- Todos los archivos pertenecen a paquetes:** `package com.yoprogramo.miprimerproyecto;`
- Importando un paquete para usar en tu proyecto:** `import java.util.*;`
- Se declara el tipo de dato String de la variable:** `String` en `private String nombre;`
- Modificadores de acceso: private, public, protected, etc:** `public` en `public class Inquilino {` y `public static void main`.
- Palabra reservada new Crea un objeto:** `new` en `Propiedad prop = new Propiedad();`
- Java utiliza clases para ejecutar el código:** `class Inquilino {`
- main es el método principal en Java:** `main` en `public static void main`.
- Se usan {} llaves para identificar el bloque de código:** Las llaves que definen el cuerpo de la clase y el método `main`.
- Se utilizan para fin de cada sentencia:** Los puntos y coma `;` al final de cada línea de código.

Acerca de los programas Java, es muy importante tener en cuenta los siguientes puntos:

- **Sensibilidad a mayúsculas y minúsculas:** Java distingue entre mayúsculas y minúsculas, lo que significa que el identificador **Hola** y **hola** tendrían un significado diferente en Java.
- **Nombres de clases:** para todos los nombres de clases, la primera letra debe estar en mayúsculas. Si se utilizan varias palabras para formar un nombre de la clase, la primera letra de cada palabra interior debe estar en mayúsculas. Ejemplo: *clase MiPrimerClaseJava*
- **Nombres de métodos:** todos los nombres de métodos deben comenzar con una letra minúscula. Si se utilizan varias palabras para formar el nombre del método, la primera letra de cada palabra interior debe estar en mayúsculas. Ejemplo: *public void miPrimerMetodo ()*
- **public static void main (String args []):** el procesamiento del programa Java comienza desde el método `main ()` que es una parte obligatoria de cada programa Java. Veamos un ejemplo en la siguiente imagen:

```
public static void main(String[] args) {

    // Acá se escribe el código a ejecutar
}
```

### Palabras reservadas

Java como cualquier otro lenguaje de programación tiene palabras reservadas que no puedes utilizar porque son lenguaje utilizado internamente. En la imagen siguiente te dejamos la lista de estas palabras.

abstract	e return	static	s
confit	throw	tranfien	package
finally	break	t care	super
int	default	double	voil
pblic	fov long	if new	chaw
this	glont	scictpf	extendy
boolean	throw	try	import
continue	byte do	catch	private
float	goto	elye	switch
interfac	native	implemet	volatile

### Identificadores de Java

Todos los componentes de Java requieren nombres. Los nombres utilizados para clases, variables y métodos se denominan **identificadores**.

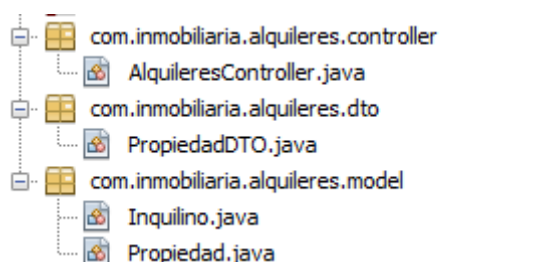
En Java, hay varios puntos para recordar acerca de los identificadores:

1. Todos los identificadores deben comenzar con una letra (de la A a la Z o de la a a la z), un carácter de moneda (\$) o un guión bajo (\_).
  2. Después del primer carácter, los identificadores pueden tener cualquier combinación de caracteres.
  3. No se puede utilizar una palabra clave como identificador.
  4. Lo más importante es que los identificadores distinguen entre mayúsculas y minúsculas.
- Ejemplos de identificadores legales: edad, \$ salario, \_valor, \_\_1\_valor.
  - Ejemplos de identificadores ilegales: 123abc, -salary.

### Paquetes Java

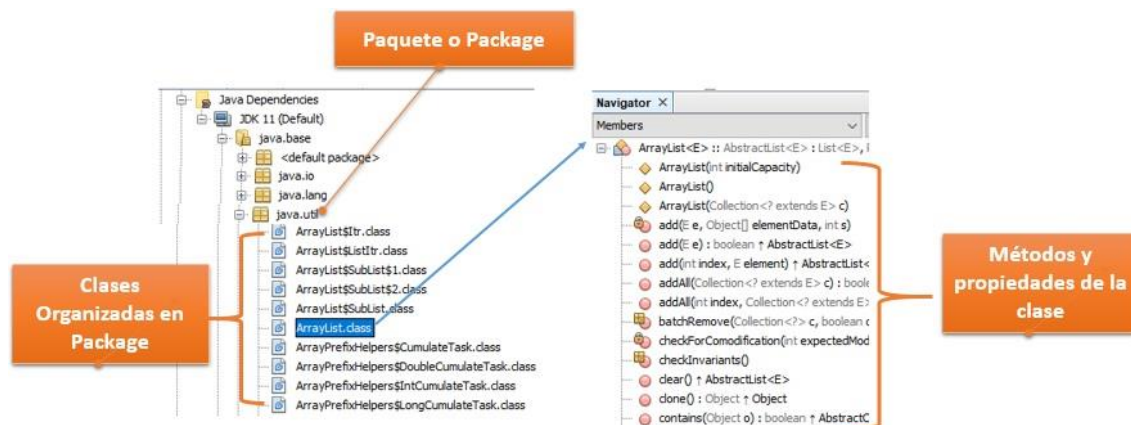
En palabras simples, es una forma de categorizar las clases y las interfaces. Al desarrollar aplicaciones en Java, se escribirán cientos de clases e interfaces en un programa, por lo que categorizar estas clases es una necesidad y hace la vida mucho más fácil.

Te mostramos en la siguiente imagen cómo se ven los paquetes creados en un proyecto para organizar las clases.





Java utiliza mucho estos paquetes para la organización de su código, miremos la siguiente imagen:



Lo último que tienes que saber de los paquetes es: para utilizar las clases que están organizadas en un paquete se deben importar, te dejamos una imagen en donde se muestra cómo se importa un paquete:

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

## Modificadores de Java

Al igual que otros lenguajes, es posible modificar **clases, métodos, etc.**, Mediante el uso de modificadores. Hay dos categorías de modificadores:

- **Modificadores de acceso:** para establecer niveles de acceso para clases, variables, métodos y constructores
  - defecto
  - público
  - protegido
  - privado
- **Modificadores sin acceso:** modificadores sin acceso para lograr muchas otras funciones
  - final,
  - abstrac
  - estático

Java proporciona una serie de modificadores de acceso para establecer niveles de acceso para clases, variables, métodos y constructores.

En la siguiente imagen te dejamos los modificadores de acceso en formato grilla:



		Modificadores			
Visibilidad		public	protected	private	Default
Desde la misma clase		SI	SI	SI	SI
Desde Mismo Paquete	desde cualquier clase del mismo paquete	SI	SI	SI	NO
	Desde una subClase DEL mismo paquete	SI	SI	SI	NO
Desde otro Paquete	Desde una subClase fuera del paquete	SI	SI, a través de una Herencia	NO	NO
	Desde cualquier clase fuera del paquete	SI	NO	NO	NO

## Variables en Java

Ya sabemos sobre las variables, lo que debemos aprender o conocer en el lenguaje Java son los tipos que podemos encontrar. Podemos agruparlos para recordarlos, por un lado, están los **tipos de datos primitivos** (no son objetos ni tampoco tienen métodos) y los **tipos de datos objeto** (que normalmente incluyen métodos). El siguiente esquema es importante tenerlo en cuenta al momento de programar para saber qué tipo de datos podemos usar en Java según sea necesario.

		NOMBRE	TIPO	OCUPA EN MEMORIA	RANGO APROXIMADO
TIPOS DE DATOS EN JAVA	TIPOS PRIMITIVOS (sin métodos; no son objetos; no necesitan una invocación para ser creados)	byte	Entero	1 byte	-128 a 127
		short	Entero	2 bytes	-32768 a 32767
		int	Entero	4 bytes	$2 \cdot 10^9$
		long	Entero	8 bytes	Muy grande
		float	Decimal simple	4 bytes	Muy grande
		double	Decimal doble	8 bytes	Muy grande
		char	Carácter simple	2 bytes	---
		boolean	Valor true o false	1 byte	---
	TIPOS OBJETO (con métodos, necesitan una invocación para ser creados)	String (cadenas de texto), Date (fechas)			
		Tipos de la biblioteca estándar de Java	Muchos otros (p.ej. Scanner, TreeSet, ArrayList...)		
		Tipos definidos por el programador / usuario	Cualquiera que necesites crear, por ejemplo Taxi, Autobus, Tranvia		
		arrays	Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.		
		Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos pero como objetos.)	Byte Short Integer Long Float Double Character Boolean		

Recordemos que las de variables tienen 3 momentos:

1. La declaración.
2. La asignación.
3. La utilización.

La forma de declarar una variable en Java es la siguiente:

```
int edad;           //Declaración
edad = 0;           //Asignación
edad = edad + 7;    //Utilización
```

En la siguiente imagen te mostramos ejemplos de cómo se declaran variables en Java:

```
int a, b, c;           // Declaración de tres variables a, b, c de tipo int
int a = 10, b = 10;    // Ejemplo de asignación
byte B = 22;           // Declaración y asignación de variable 'B' de tipo byte.
double pi = 3.14159;   // Declaración y asignación de variable 'pi' de tipo double.
char a = 't';          // Declaración y asignación de variable 'a' de tipo char
String nombre = "Fernanda"; // Declaración y asignación de variable 'nombre' de tipo String
```

Java proporciona la clase Date disponible en el paquete java.util, esta clase encapsula la fecha y hora actuales.

Este es un método muy sencillo para obtener la fecha y hora actual en Java. Puedes usar un objeto Date simple con el método toString () para imprimir la fecha y hora actuales de la siguiente manera:

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instancia del objeto
        Date date = new Date();

        // Mostrar la fecha convirtiendo a string
        System.out.println(date.toString());
    }
}
```

## Vectores y Matrices

### Estructuras Estáticas – Arreglos (Arrays)

Los arreglos (o arrays en inglés), son un conjunto de datos que se caracterizan por almacenarse en memoria de manera contigua, bajo un mismo nombre, pero con diferentes “índices” o “identificadores” para diferenciar la ubicación de cada uno de ellos.

Los arreglos son estructuras fijas, es decir, que una vez declarados e inicializados, mantienen su tamaño durante toda la ejecución del programa. ¿Qué quiere decir esto? Por ejemplo, si se declara e inicializa un arreglo con 5 posiciones, estas 5 se mantendrán de principio a fin de la ejecución del programa que se esté desarrollando sin la posibilidad de cambiar su tamaño.

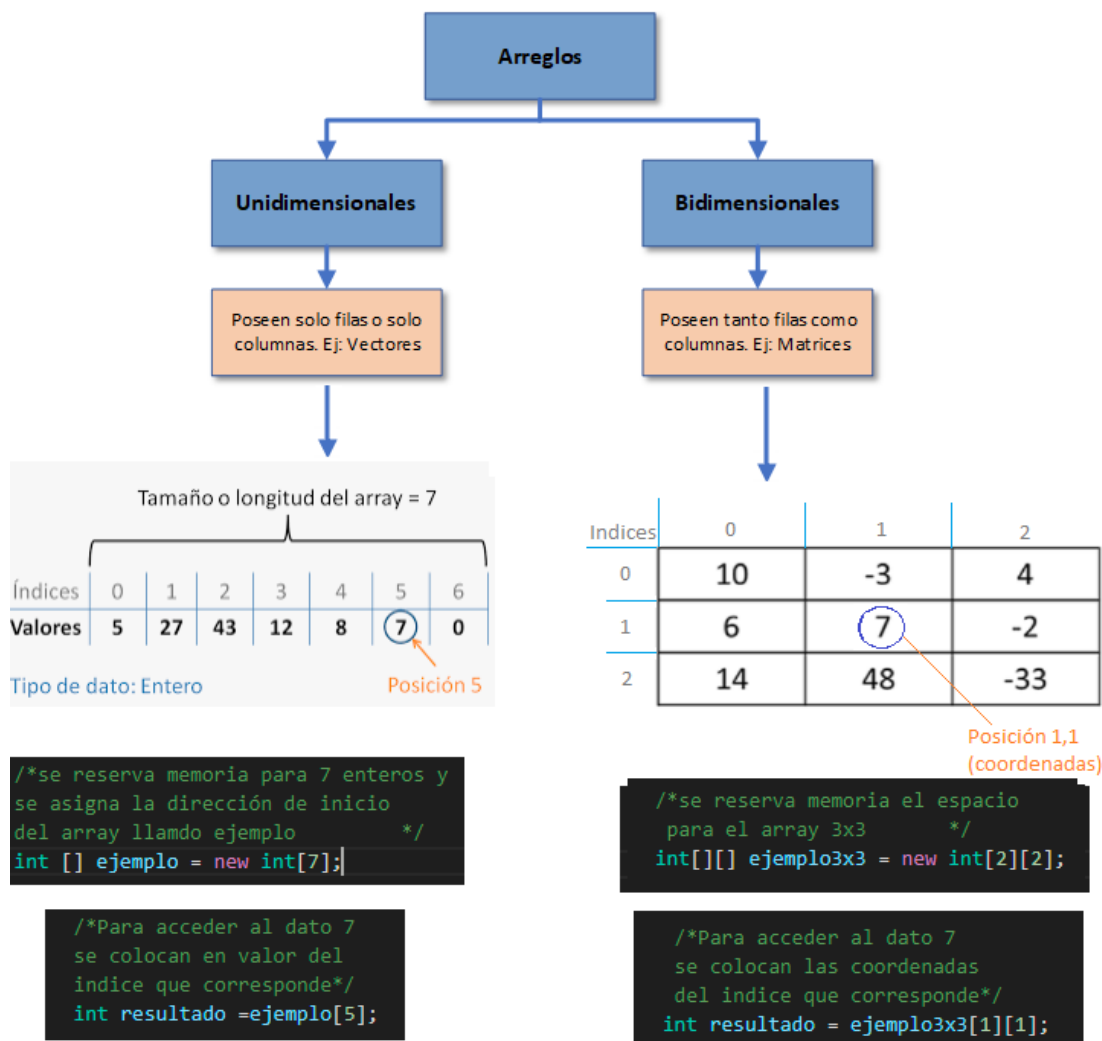
Los arreglos, al igual que las variables comunes, deben poseer **un solo tipo de dato** determinado. Este tipo de dato debe ser **único para todos los elementos** que conforman el array con el que se esté trabajando. Observemos un ejemplo de un arreglo de tipo numérico:



## Tipos de Arreglos

Existen dos tipos principales de arreglos, los unidimensionales y los bidimensionales. Los arreglos unidimensionales son aquellos que poseen una sola dimensión sin importar su disposición, es decir, posee solo una fila o una columna con un conjunto de valores. Un ejemplo por excelencia de este tipo de arreglos son los vectores.

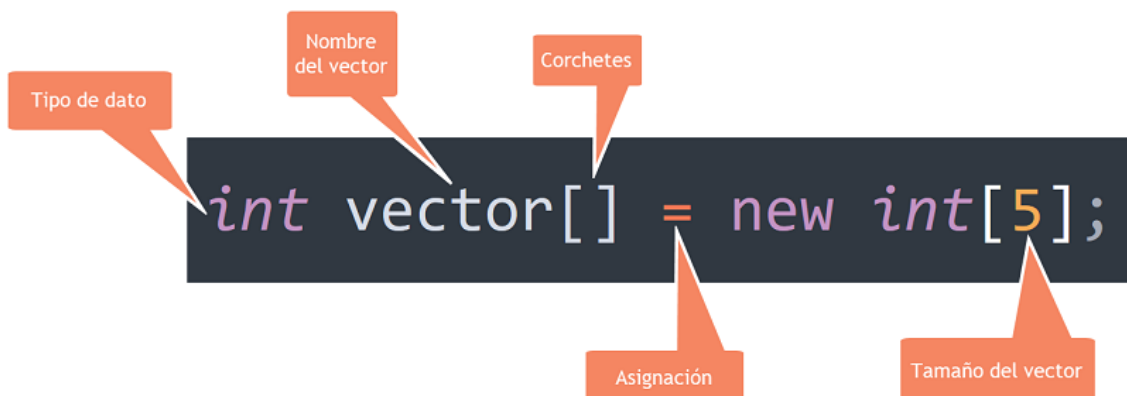
Por otro lado, los arreglos bidimensionales son aquellos que poseen dos dimensiones, es decir, tanto filas como columnas al mismo tiempo. Un ejemplo de esto son las reconocidas matrices:



## Arreglos Unidimensionales (Vectores)

### Vectores: Declaración, inicialización y Asignación

Los vectores son arreglos unidimensionales que permiten almacenamiento de datos de manera contigua. En Java se declaran especificando el tipo de dato que almacenarán, el nombre y la identificación [] la cual determina que se trata de un vector. Al mismo tiempo, para inicializar las posiciones de un vector, es necesario asignar al vector declarado la palabra new más el tipo de dato y nuevamente [], donde esta vez se especifica la longitud que tendrá el arreglo. Un ejemplo práctico de esto puede observarse en la siguiente ilustración:



Una vez declarado e inicializado un vector, es posible asignarle diferentes valores en cada una de sus posiciones a partir de dar a conocer el índice donde estos datos deberán ir. Un ejemplo de esto a nivel código puede verse a continuación, como así también a nivel vector:



*Resultado de la asignación de valores a un vector*

Un detalle muy importante a tener en cuenta es que, por convención mundial, los vectores comienzan su índice en el valor 0. ¿Qué quiere decir esto? Que si tenemos un vector de 5 posiciones, sus índices irán del 0 al 4, por lo que si hacemos referencia al índice 5, no estaríamos posicionados en la 5ta posición, sino en la sexta; esto, al tratarse de un vector de únicamente 5 posiciones provocaría **un error por desbordamiento**.



## Vectores: Recorrido y Carga

Un vector que ya se encuentra cargado o con valores asignados, puede ser recorrido tanto para mostrar los valores que contiene como así también para utilizarlos en caso de que sean necesarios. Para llevar a cabo este recorrido la mejor opción es utilizar la estructura repetitiva for.

Los vectores se recorren de manera secuencial, es decir, posición a posición según un determinado orden que se establezca. Suponiendo el mismo vector de la ilustración anterior, podríamos recorrerlo mediante la porción de código que se encuentra a continuación:



```
for (int i=0; i<vector.length; i++) {  
    System.out.println("Estoy en la posición " + i);  
    System.out.println("Contiene el número " + vector[i])  
}
```

Diagram illustrating the code execution with callouts:

- índice en el que inicia el vector (points to `i=0`)
- Obtenemos la longitud del vector para saber hasta donde recorrer (points to `vector.length`)
- Indicamos de cuánto en cuánto aumentará el índice (points to `i++`)
- Muestro por pantalla en que posición estoy (points to `"Estoy en la posición " + i`)
- Muestro por pantalla el valor que contiene mi (points to `vector[i]`)

El ciclo for contendrá tres parámetros, el primero corresponde a la inicialización de una variable “i” que representará el índice del vector. Como segundo parámetro, tenemos la condición de parada, la cual, mediante la función length podemos obtener la longitud exacta de nuestro vector, para asegurarnos de que no haya un error por desbordamiento y que el recorrido finalice cuando llegue a la última posición. Por último, como tercer parámetro, tenemos al incremento, es decir, de cuánto en cuánto queremos que crezca nuestro índice para así poder hacer el recorrido secuencial. En este caso particular, iremos posición por posición (1 a 1), es por ello que se especifica `i++` (equivalente a colocar `i=i+1`).

Recorrer un vector es bastante sencillo y la estructura repetitiva for nos brinda una gran ayuda. Ahora supongamos que el vector se encuentra vacío y lo que se desea realizar es permitirle al usuario la posibilidad de cargar los vectores por teclado. Para ello, será necesario utilizar una clase especial llamada Scanner.

El Scanner es una clase (concepto muy importante en la programación orientada a objetos) que permite el manejo de ingreso de información o datos a nuestros sistemas mediante algún periférico de entrada (principalmente por teclado). Para hacer uso de él y poder cargar nuestro vector en cada una de sus posiciones, es necesario recorrerlo e ir “Scanneando” los valores para cada posición. Un ejemplo de esto puede verse en la siguiente ilustración:

```
Scanner teclado = new Scanner (System.in);

for (int i=0; i<vector.lenght; i++) {

    System.out.println("Ingrese el número para la posición " + i);
    int tecla = teclado.nextInt();
    vector[i] = tecla;
}
```

Creamos el Scanner

Recorremos el vector

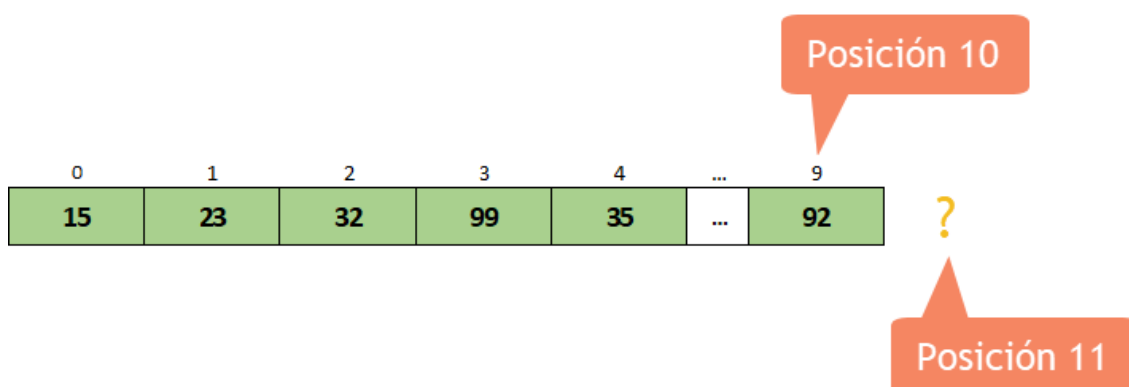
Se solicita al usuario que ingrese el valor que desea almacenar

Se lee por teclado el valor

Se asigna al vector el valor leído

## Error por desbordamiento

Un error por desbordamiento ocurre cuando queremos hacer referencia a una posición de un vector que en realidad no existe o no está contemplada para el tamaño que el mismo posee. Por ejemplo, si declaramos un vector de 10 posiciones e intentamos acceder a una posición 11, estaríamos provocando un error por desbordamiento:



A nivel de código existen diferentes maneras de que se pueda provocar un error por desbordamiento, sin embargo las dos situaciones más comunes son: colocar incorrectamente la condición de parada del bucle que recorre el vector o hacer referencia para asignación como lectura de una posición que no exista:

```
int vector [] = new int [3];

for (int i=0; i<=3; i++) {
    ...
}
```

Inicializo un vector de 3 posiciones

El for llega hasta el 3 (cuando el máximo índice es 2)

## Arreglos Bidimensionales (Matrices)

Las matrices son arreglos bidimensionales, es decir, poseen dos dimensiones para recorrer, en este caso filas y columnas. Así como en los vectores es necesario un índice para identificar cada una de sus posiciones, en las matrices se utilizan dos índices, uno para identificar las filas y otro para las columnas, de esta manera la posición de nuestro arreglo bidimensional estará dada por dos índices al mismo tiempo tal y como puede verse en la siguiente ilustración:

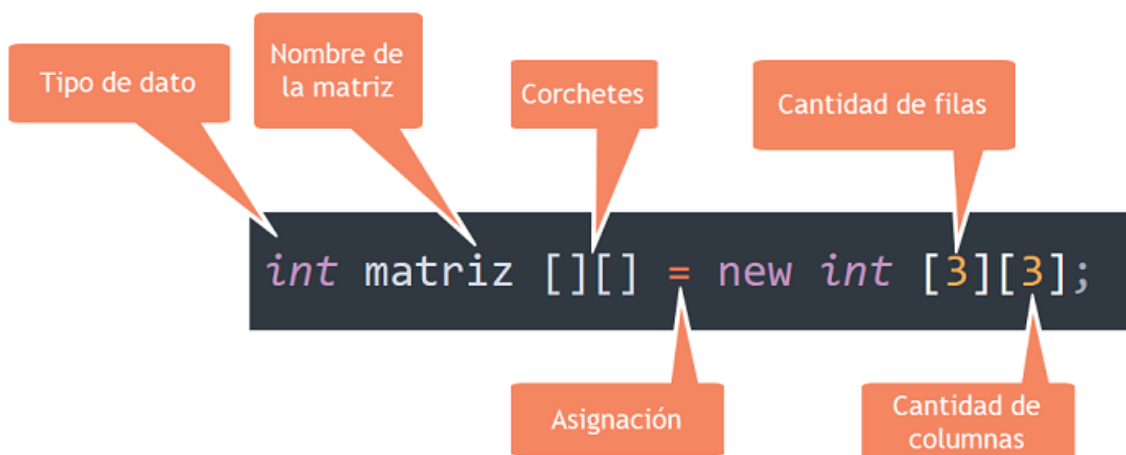
		Columna		
		0	1	2
Fila	0	10	23	32
	1	20	64	61
	2	91	12	14

Matriz de 3 filas y 3 columnas

## Matrices: Declaración, inicialización y Asignación

Las matrices se declaran, inicializan y asignan de forma muy similar a los vectores, con la diferencia de que en lugar de utilizar un solo conjunto de corchetes [] para

especificar las posiciones, ahora utilizaremos dos `[]`, uno para especificar el índice de las filas y otro para el índice de las columnas:



Una vez declarada e inicializada una matriz, es posible asignarle diferentes valores en cada una de sus posiciones a partir de dar a conocer el índice para las filas y para las columnas en conjunto con los valores que se asignarán. Un ejemplo de esto a nivel código puede verse en la continuación:

```
int matriz [][] = new int [3][3];

matriz[0][0] = 10;
matriz[0][1] = 23;
matriz[0][2] = 32;
matriz[1][0] = 20;
matriz[1][1] = 64;
matriz[1][2] = 61;
matriz[2][1] = 91;
matriz[2][2] = 12;
matriz[2][3] = 14;
```

Al igual que en los vectores, los valores de las filas y columnas siempre comienzan en 0, en donde el valor del primer par de corchetes pertenece siempre a las filas y el segundo a las columnas sin excepción.

## Matrices: Recorrido y Carga

El recorrido de las matrices es un poco más complejo que el de los vectores. Esto se debe a que no tenemos únicamente una dirección de recorrido, sino dos, tanto filas como columnas al mismo tiempo.

El algoritmo principal para poder recorrer matrices está compuesto por la utilización de dos estructuras repetitivas for, una para el control del recorrido de las filas, y otro para el control del recorrido de las columnas:

```
for (int f=0; f<3; i++) {  
    for (int c=0; c<3; i++) {  
        System.out.println("Estoy en la fila " + f + " columna: " + c);  
        System.out.println("Tiene guardado un " + matriz[f][c]);  
    }  
}
```

For para recorrer filas

For para recorrer columnas

Para mostrar el contenido de la matriz hacemos referencia a ambos índices

Así como en los vectores es posible realizar la carga de datos a través del teclado solicitándole a un usuario que lo haga, también se puede realizar la misma acción con las matrices mediante la clase Scanner:

```
Scanner teclado = new Scanner (System.in);  
for (int f=0; f<3; i++) {  
    for (int c=0; c<3; i++) {  
        System.out.println("Ingrese un valor para fila " + f + " columna: " + c);  
        int tecla = teclado.nextInt();  
        matriz[f][c] = tecla;  
    }  
}
```

## Estructuras Dinámicas Collections

### Collections

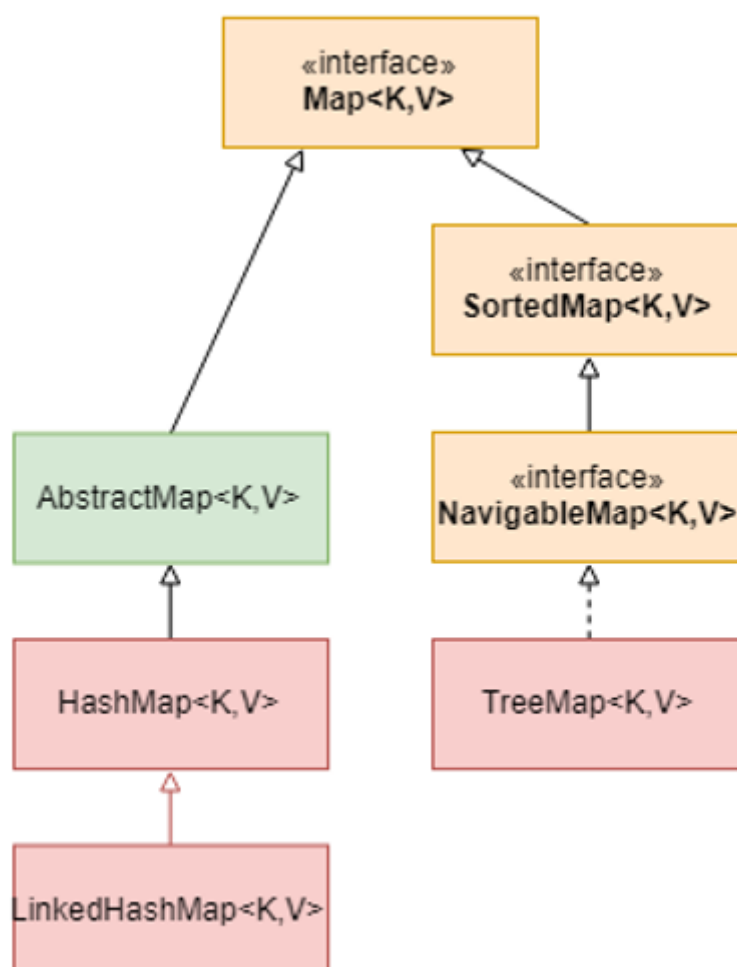
Las colecciones son estructuras similares a los arreglos, pero con la principal característica de que son dinámicas, es decir, que pueden variar su longitud o tamaño durante la ejecución de un programa.

Las Collections (como lo dice su nombre) son colecciones de objetos que se agrupan todos como si fuesen una misma unidad. Se basan en el mismo concepto de “colección” de la vida real, donde en lugar de tener por ejemplo una colección de estampillas, tendremos una colección de números u objetos de un determinado tipo.

Las colecciones en Java dependen de distintos tipos de implementaciones de interfaces, pero básicamente se puede decir que son 3 los tipos más utilizados:

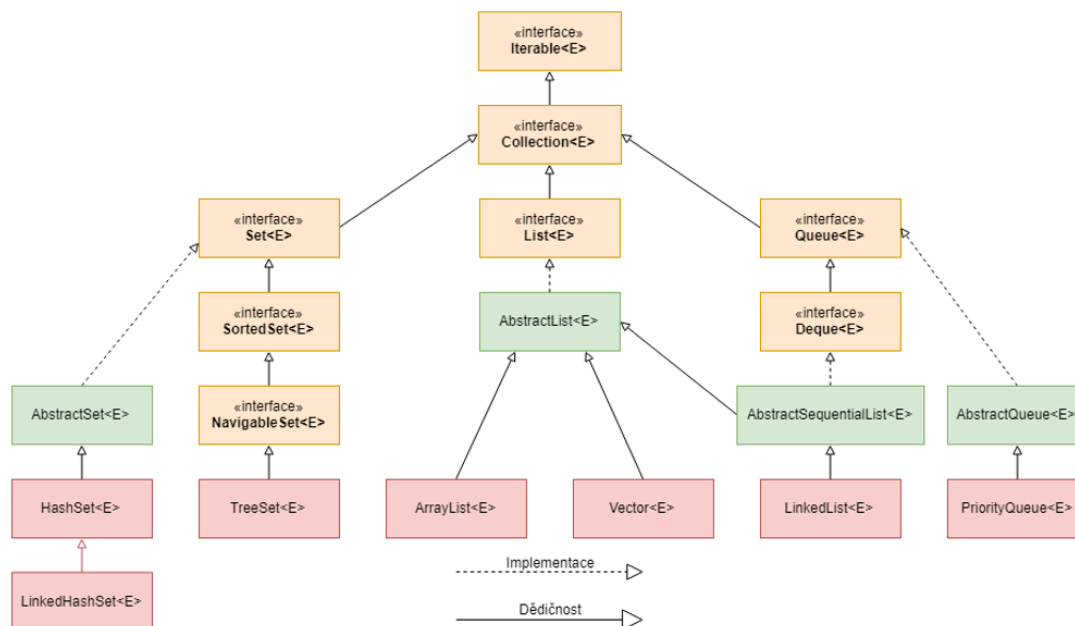
- **List**
- **Set**
- **Map**

Los árboles de jerarquía de implementación de las interfaces por cada uno de estos tipos de collections pueden verse reflejados en los siguientes diagramas:



*Jerarquía de Collection Map*





### *Jerarquía de Collections List, Set y Queue*

Si bien existen distintos tipos de Collections, las más utilizadas son sin duda las Listas, las cuales se especificarán en mayor detalle a continuación.

## Collection List

Las listas son consideradas como un conjunto de elementos relacionados entre sí que tienen un determinado orden. La principal característica de las listas (como toda colección) es el hecho de poseer un tamaño y estructura dinámicos, por lo que pueden cambiar durante el tiempo de ejecución de un programa.

En Java existen diferentes tipos de listas, entre las cuales se destacan:

- **ArrayList**
- **LinkedList**
- **Stack**

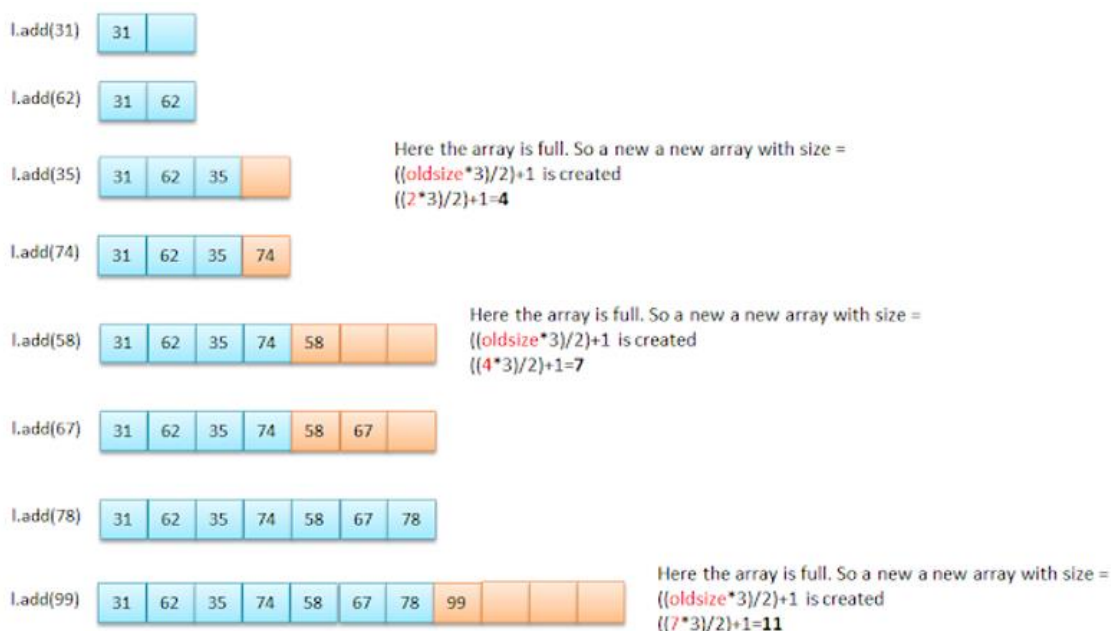
## ArrayList

Tal y como lo expresa su nombre, son considerados arreglos dinámicos que permiten almacenar elementos. Tiene la característica de heredar de la clase AbstractList la cual implementa al mismo tiempo la interface List, es por eso que es considerada como un tipo de lista.

Una de las principales características de los ArrayLists es la posibilidad de tener elementos duplicados, como así también cumplir con el orden FIFO (First In First Out), en donde el orden que tienen los registros dentro de la lista es el mismo en el que fueron insertados.

Como son arreglos, cada una de sus posiciones tiene un índice determinado, por lo cual se puede acceder a cualquier posición de forma aleatoria, como así también de manera secuencial recorriendo posición a posición.

Una contra de los ArrayLists es el hecho de que poseen una manipulación lenta, dado que generalmente para hacer algún cambio es necesario recorrer el arraylist completo. Un ejemplo del funcionamiento de los mismos podemos verlo a continuación:



### Ejemplo de funcionamiento de ArrayList

Los ArrayLists al igual que los arreglos comunes y corrientes, deben ser declarados e inicializados. Un ArrayList siempre será declarado como una Lista y especificado su tipo al crear una nueva instancia. Te mostramos un ejemplo a nivel código de la declaración y asignación:

Siempre se declaran como de tipo List y entre <> el tipo de dato u clase que almacenarán

Nombre de la lista

Se especifica que será de tipo ArrayList

Se especifica entre <> el tipo de dato o clase que permitirá almacenar

```
List <Persona> listaPersonas = new ArrayList<Persona> ();
```

En los arreglos estáticos es posible asignar valores a cada una de las posiciones, sin embargo, en los ArrayLists esto no es posible, sin embargo, existe una función específica para la incorporación de nuevos elementos a la lista. Esta función es conocida como add y un ejemplo de la misma puede verse en la siguiente imagen. Es importante recordar que en los ArrayLists el primer elemento que sea ingresado será el primero en ser tratado:

```
List <Persona> listaPersonas = new ArrayList<Persona> ();

listaPersonas.add(new Persona(1,"Gabriel", 30));
listaPersonas.add(new Persona(2,"Lucy", 56));
listaPersonas.add(new Persona(3,"Guillermo", 60));
listaPersonas.add(new Persona(4,"Luisina", 29));
```

Método add

Puedo pasar como parámetro la creación de un nuevo objeto como así también un objeto ya instanciado.

### Carga de ArrayLists

Por otro lado, los ArrayLists también pueden ser recorridos para visualizar cada uno de sus elementos y existen dos formas de hacerlo, ambas mediante la estructura repetitiva for, tal y como con los arreglos estáticos. Ejemplos de estas opciones de recorrido:

```
//opción 1
for (Persona pers : listaPersonas) {

    System.out.println("Nombre: " + pers.getNombre());
    System.out.println("Apellido: " + pers.getApellido());
}

//opción 2
for (int i=0; i<listaPersonas.size(); i++) {
    System.out.println("Persona: " + listaPersonas.get(i));
}
```

## Linked List

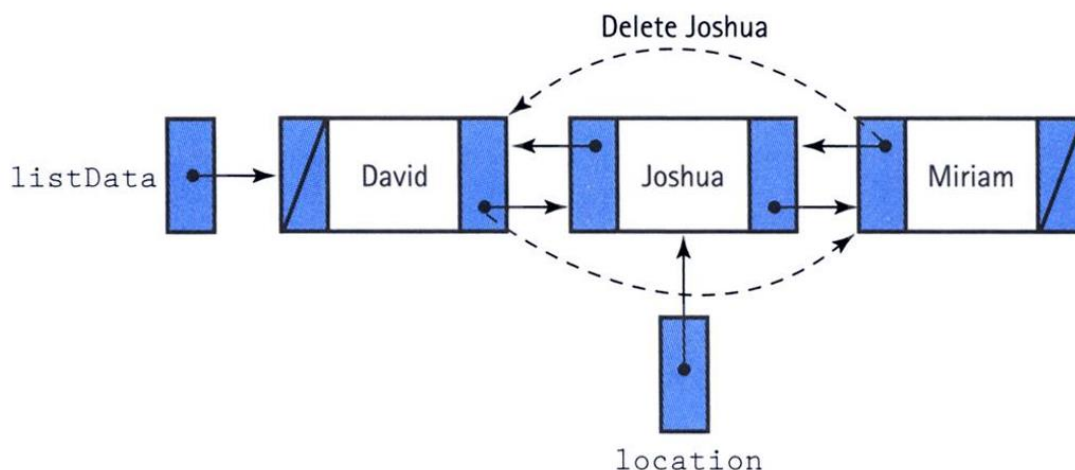
Las LinkedList o mejor conocidas como listas doblemente enlazadas (porque permiten una relación de “ida y vuelta” entre cada registro) son un tipo de lista que se caracterizan porque cada uno de sus elementos no se encuentran de forma contigua en memoria (como el caso de los ArrayLists) pero si poseen dos punteros que enlazan fuertemente a cada uno de ellos entre sí.

Hereda de Abstract Sequential List que al mismo tiempo hereda de AbstractList e implementa la interfaz List, tal y como se puede apreciar en la imagen de más abajo.

Por otro lado, las listas enlazadas también permiten duplicados entre sus elementos, con la ventaja, además, de que poseen una manipulación mucho más rápida que los ArrayLists.

Las LinkedLists tienen la característica de poder ser tratadas no solo como listas, sino también como colas o como pilas, teniendo la capacidad de permitir llevar a cabo inserciones o eliminaciones al principio o al final de la colección.

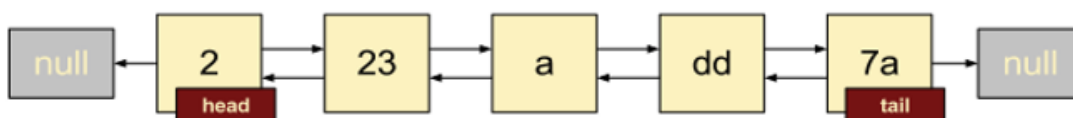
Un ejemplo de la distribución y punteros que utiliza una LinkedList :



Aunque parecen muy similares por las clases de las cuales heredan o por implementar la misma interfaz List, tanto ArrayLists como LinkedLists tienen sus diferencias, caracterizándose (en primera medida) por las posiciones contiguas o no de cada uno de sus elementos, como así también el hecho de que los ArrayLists utilizan índices y solo permiten inserción al final de la lista a diferencia de las listas enlazadas, que si bien no poseen índices, permiten inserciones al principio o al final. Te mostramos un ejemplo comparativo en la siguiente imagen:

## Array vs. Linked List

### Linked List



### Array



### Ejemplo comparativo

A nivel código, la implementación de una LinkedList es muy similar a la de una ArrayList, con la diferencia (como se mencionó anteriormente) que el método add permite también agregar elementos al principio de una lista que ya pudiera contener elementos o ya estar cargada, mediante un parámetro 0:

```
//Declaración e inicialización
List <Persona> listaPersonas = new LinkedList<Persona> ();

//carga al final
listaPersonas.add(new Persona(1,"Gabriel", 30));
listaPersonas.add(new Persona(2,"Lucy", 56));

//carga al principio (sobre lista ya cargada)
listaPersonas.add(0, new Persona(4,"Luisina", 29));

//recorrido
for (Persona pers : listaPersonas) {
    System.out.println("Nombre: " + pers.getNombre());
    System.out.println("Apellido: " + pers.getApellido());
}
```

*Declaración, inicialización, método add y recorrido de LinkedList*

## Stack

Una colección de tipo Stack representa una pila de objetos que se caracteriza por utilizar el modelo LIFO (Last In First out), es decir, el último elemento que ingresó será el primero en salir. Si nos imagináramos una pila de cajas que tengamos una encima de la otra, la primera que sacaríamos no sería la que está más cerca del suelo, sino la que esté más arriba de la pila. Una stack, cumple exactamente este mismo concepto. Stack extiende de la clase Vector, la cual implementa la interface List.

Stack tiene una serie de métodos propios que utiliza para realizar diferentes acciones. Entre algunos de estos métodos se encuentran:

- **Push:** Coloca un elemento al tope de la pila
- **Pop:** Borra el último elemento de la pila
- **isEmpty:** tal como lo dice su nombre, verifica si la pila está vacía o no (true si es así o false si está cargada).
- **Peek:** Muestra el valor que actualmente se encuentra en el tope de la pila, pero SIN ELIMINARLO.
- **Search:** Busca un elemento específico dentro de la pila.

Ejemplo práctico de utilización de cada uno de estos métodos:



```
Stack<Integer> pila = new Stack<Integer>(); //creo la pila

//verifica si está vacía la pila
System.out.println("¿Está vacía la pila?: " + pila.isEmpty());

//inserta registros al final de la pila
pila.push(1); //voy agregando números, dado que es un Integer
pila.push(2);
pila.push(3);
pila.push(4);

//recorre la pila
for (Integer pil : pila) {
    System.out.println(pil); //muestra el orden de la pila
}

//borra el elemento tope de la pila
pila.pop();

//buscar elemento "x"
//devuelve 1 o número positivo si encuentra el valor
//si no lo encuentra devuelve -1
System.out.println("¿Está el número en la pila?: " + pila.search(3));

//mostrar el elemento tope de la pila sin eliminarlo
System.out.println(pila.peek());
```

## Manejo de Excepciones

### Excepciones

Las excepciones en Java, al igual que en la vida real, son situaciones o eventos que se pueden producir y que rompen el flujo normal de un conjunto de acciones que se estén realizando. Por ejemplo, en la vida diaria si no pudimos presenciar un examen por estar enfermos, el profesor puede hacer una “excepción” y volver a tomarnos el examen otro día. No es una situación común (por suerte no es común enfermarse todos los días), pero si es una acción que cambiará totalmente los pasos que proseguirán. De igual manera, en Java, las excepciones rompen el flujo normal de ejecución de una aplicación.

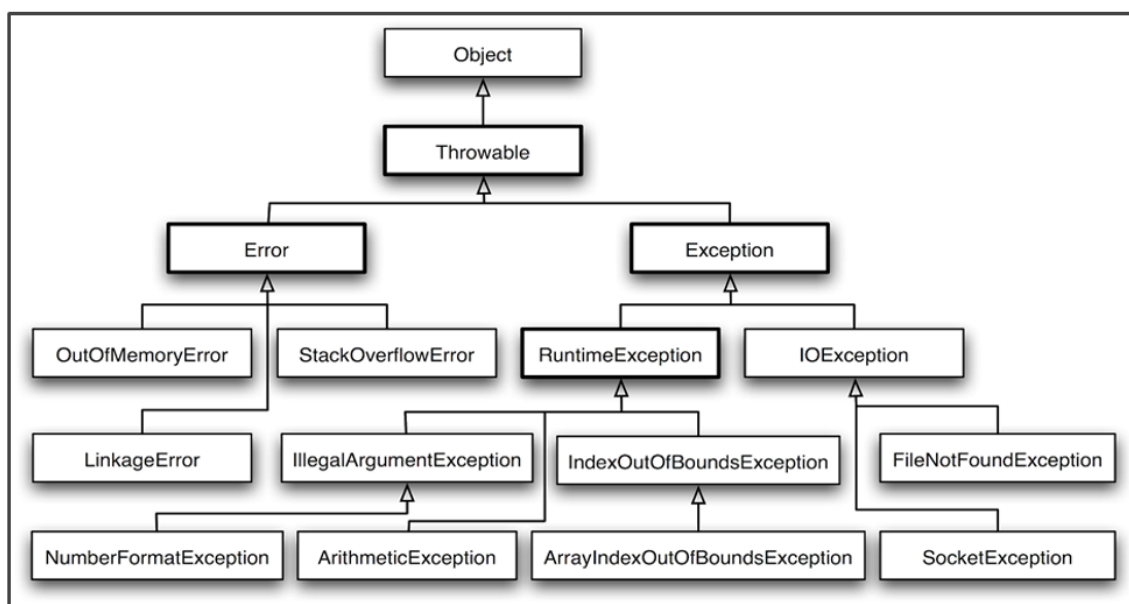
Los motivos de que se produzcan excepciones en la ejecución de un programa son muchos, entre ellos la falla de algún componente de hardware, operaciones matemáticas (por ejemplo, dividir por cero), errores generales de un programa (por ejemplo, error por desbordamiento de un arreglo), intento de apertura de un archivo que no existe, intento de acceso a una base de datos que no existe, entre otros.

## Tipos de Excepciones

En Java existen dos tipos principales de excepciones. Las **propias de Java** y las **personalizadas**. Las primeras conforman un conjunto de excepciones que por defecto Java nos brinda como lenguaje, donde entre ellas, las que más se destacan son:

- **RuntimeException:** Son aquellas que se producen en tiempo de ejecución, es decir, dentro de la máquina virtual Java cuando se está ejecutando la aplicación. Un ejemplo clásico de este tipo de excepción puede ser `NullPointerException`.
- **IOException:** Son aquellas que se producen al haber un error de tipo entrada/salida (in/out). Un ejemplo clásico de este tipo de excepción es cuando se intenta acceder a un archivo desde un programa para obtener sus datos como entrada. Este tipo de excepción debe ser tratada de forma obligatoria, ya sea en la cabeza del método con un “throws IOException” o con un bloque try y catch.

Toda la estructura de excepciones existente en Java por defecto puede visualizarse en la siguiente ilustración:



*Estructura de excepciones en Java*

En cuanto a las excepciones personalizadas, Java nos permite realizarlas estableciendo el mensaje personalizado que queremos mostrar al usuario, como así también la determinada situación donde podría llegar a producirse. Para poder crear una excepción personalizada es necesario crear una nueva clase que como condición indiscutible extienda de la clase `Exception` para que pueda implementar

cada una de sus características particulares de toda excepción. Un ejemplo puede visualizarse en la siguiente ilustración:

```
public class MiExcepcionPropia extends Exception {  
  
    public MyExcepcionPropia () {  
        super ();  
    }  
  
    public metodoUno (parámetros que recibo) {  
        hacerAlgo();  
    }  
  
    public metodoDos (parámetros que recibo) {  
        hacerAlgo2();  
    }  
  
}
```

### Captura de Excepciones

En Java para capturar las excepciones que se hayan podido producir se utilizan las expresiones “try” (intentar) y “catch” (atrapar) para delimitar el código que es necesario controlar ante la posible aparición de una excepción. Cuando se produce una excepción, el bloque “try” termina y “catch” captura o atrapa la excepción y muestra o ejecuta el código que está pensado para esta excepción en cuestión:

```
try {  
  
    localizar archivo;  
    abrir archivo;  
    leer archivo;  
    cerrar archivo;  
  
}  
catch (fallo de localizacion del archivo) {  
    hacerAlgo();  
}  
catch (fallo al abrir el archivo) {  
    hacerAlgo();  
}  
catch (fallo al leer el archivo) {  
    hacerAlgo();  
}  
catch (fallo al cerrar el archivo) {  
    hacerAlgo();  
}
```

Supongamos que estamos realizando un programa donde es posible que en algún momento se produzca una división por cero y queremos preverla para que, al producirse la excepción, no cause un error en la ejecución. Para ello, vamos a rodear la operación con un try, de esta forma, en caso de que se produzca la excepción, el catch la capturará y nos permitirá mostrar un mensaje informando esto. A nivel código podemos ver este ejemplo en la siguiente Ilustración:

```
try {  
    double division = 35 / 0;  
}  
catch (Exception e) {  
    System.out.println("¡¡No podés dividir por cero!!");  
}
```

Ahora supongamos otro caso común como es el de un posible error por desbordamiento cuando trabajamos con vectores o matrices. En el caso de que hagamos referencia a una posición que no existe, Java nos lanzará una excepción por el error por desbordamiento, sin embargo, mediante try y catch podemos prever esta situación para evitar que se finalice inesperadamente la ejecución de nuestro programa:

```
try {  
    int edades [] = {3,4,5,6};  
    //intentamos provocar un error por desbordamiento  
    System.out.println("La edad de la posición Nº 4 es: " + edades [4]);  
}  
catch (Exception e) {  
    System.out.println("¡¡Provocaste un error por desbordamiento!!");  
}
```

### Otras sintaxis de Excepciones

Además del try y catch que son las sentencias por excelencia del manejo de excepciones, también existen otros bloques y sentencias que nos permiten el manejo de las mismas. Entre ellos se encuentran:

- **finally:** Es un bloque OPCIONAL de código que se coloca al finalizar el catch. En el caso de que se encuentre implementado, es una porción de código que se ejecutará SIEMPRE, sin importar si se ejecutó el try o el catch.
- **throw:** Es una sentencia que permite arrojar a “propósito” una excepción en cualquier lugar del código que sea necesario.
- **throws:** Es utilizada siempre en las declaraciones de los métodos y se encarga de especificar en detalle qué excepciones puede lanzar el método en cuestión.

## Métodos en Java

Un método Java es una colección de declaraciones que se agrupan para realizar una operación. Cuando llamas al System.out. **println ()** , por ejemplo, el sistema en realidad ejecuta varias declaraciones para mostrar un mensaje en la consola.

Ahora aprenderás a crear tus propios métodos con o sin valores de retorno, a invocar un método con o sin parámetros y a aplicar la abstracción de métodos en el diseño del programa.

### Métodos vs Funciones

Hagamos una comparación simple para recordar la diferencia entre ambas:

Una **función** es un fragmento de código que se llama por su nombre. Se pueden pasar datos para operar (es decir, los parámetros) y, opcionalmente, se pueden devolver datos (el valor de retorno). Todos los datos que se pasan a una función se pasan explícitamente.

Un **método** es un fragmento de código que se llama por un nombre asociado con un objeto. En la mayoría de los aspectos, es idéntico a una función, excepto por dos diferencias clave:

1. A un método se le pasa implícitamente el objeto en el que se llamó.
2. Un método puede operar en datos contenidos dentro de la clase (recordando que un objeto es una instancia de una clase: la clase es la definición, el objeto es una instancia de esos datos).

Te mostramos un ejemplo de cómo se escribe:

```
public static int nombreMetodo(int a, int b) {  
    // bdy  
}
```

Analizando más en detalle podemos ver lo siguiente:

- **public static**-- modificador
- **int** - tipo de retorno
- **nombreMetodo**- nombre del método
- **int a, int b** - lista de parámetros con su tipo de datos