Unidad 2

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 3

Control del Programa y Vectores

Control del Programa y Vectores

En este Capítulo

- Sentencias if – else
- Sentencia switch
- La sentencia for
- Sentencia while
- Sentencia do-while
- Casos Especiales de Control del Flujo
- Vectores
- Matrices o Vectores Bidimensionales
- Límite de un Vector
- Redimensionamiento de un Vector
- Copia de Vectores

Sentencias `if – else`

Proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio booleano. Generalmente, la forma sencilla es

```
if (expresión booleana) sentencia;
```

Este formato es válido cuando se quiere ejecutar una sola sentencia dado que la condición es verdadera. Sin embargo, si se quiere usar más de una sentencia, se debe utilizar un bloque asociado al **if** para agruparlas. Su formato es

```
if (expresión booleana) {
      [sentencias;]
}
```

Cuando se quiere analizar también el caso en el cual la expresión booleana es falsa se puede utilizar la sentencia **else**. Por ejemplo

```
if (expresión booleana) {
      [sentencias;]
}
else sentencia;
```

Nuevamente, esto es útil si se ejecutan varias sentencias si la condición es verdadera y tan solo una en caso de ser falsa.

Para el caso de ejecutar varias sentencias cuando la condición es falsa y, por ejemplo, una sentencia sola si es verdadera, se puede volver a utilizar la técnica de asociar un bloque de sentencias, por ejemplo

```
if (expresión booleana) sentencia;
else {
[sentencias;]
}
```

El último caso a analizar es cuando se ejecutan muchas sentencias en ambas situaciones, el formato es

```
if (expresión booleana) {
[sentencias;]
}
else {
[sentencias;]
}
```

Ejemplo

```
if (res == 1) {
. . .
// Código para la acción en caso que res sea igual a 1
. . .
} else {
```

```
. . .
// código para la acción en caso que res no sea igual a 1
. . .
}
```

Existe otra forma de la sentencia **else**, **else if** que ejecuta una sentencia basada en otra expresión. No es más que una forma particular de combinar un **if** con un **else** (no es una sentencia diferente) y, aunque no es recomendable porque puede inducir a errores, muchos programadores la utilizan.

Ejemplo

```
int puntuacion;
String nota;
if (puntuacion >= 90) {
      nota = "Sobresaliente";
} else if (puntuacion >= 80) {
      nota = "Notable";
} else if (puntuacion >= 70) {
      nota = "Bien";
} else if (puntuacion >= 60) {
      nota = "Suficiente";
} else {
      nota = "Insuficiente";
}
```

Otro caso a tener en cuenta es el del anidamiento. Cuando se anidan sentencia **if** – **else** es importante tener en cuenta qué **if** se asocia con cuál **else**.

Para esto, se debe seguir la siguiente regla:

> *El **else** del anidamiento más interno se asocia con el primer **if** que encuentra, siguiendo esta asociación desde adentro hacia afuera*

La única forma de alterar esta regla es asociando las sentencias a bloques. Esto provoca que se evalúe el bloque como una unidad del **if** al que este asociado.

Ejemplo

```
if (expresión booleana) {
      [sentencias;]
      if (expresión booleana) sentencia;
            if (expresión booleana) {
                  [sentencias;]
            }
            else {
                  [sentencias;]
            }
      else {
            [sentencias;]
      }
else {
      [sentencias;]
```

```
}
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia **if - else** y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el **if - else**.

Sentencia switch

La sintaxis de la sentencia **switch** es el siguiente:

```
switch ( expr1) {
      case constante2:
            [sentencias;]
            break;
      case constante3:
            [sentencias;]
            break;
      default:
            [sentencias;]
            break;
}
```

La sentencia **switch** se utiliza para ejecutar sentencias en base al valor que arroja una expresión. Por ejemplo, si un programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha y se quiere mostrar el nombre del mes basándose en su número entero equivalente, se podría utilizar la sentencia **switch** de Java para realizar esta tarea.

Ejemplo

```
int mes;
. . .
switch (mes) {
      case 1: System.out.println("Enero"); break;
      case 2: System.out.println("Febrero"); break;
      case 3: System.out.println("Marzo"); break;
      case 4: System.out.println("Abril"); break;
      case 5: System.out.println("Mayo"); break;
      case 6: System.out.println("Junio"); break;
      case 7: System.out.println("Julio"); break;
      case 8: System.out.println("Agosto"); break;
      case 9: System.out.println("Septiembre"); break;
      case 10: System.out.println("Octubre"); break;
      case 11: System.out.println("Noviembre"); break;
      case 12: System.out.println("Diciembre"); break;
}
```

La sentencia **switch** evalúa la expresión, en este caso el valor de mes, y ejecuta la sentencia **case** apropiada. Cada sentencia case debe ser única y el valor proporcionado a cada sentencia **case** deberá cumplir dos condiciones: ser un valor constante y del mismo tipo de dato que el devuelto por la expresión.

Otro punto interesante son las sentencias **break** después de cada **case**. La sentencia **break** hace que el control salga de la sentencia **switch** y continúe con la siguiente línea. La sentencia **break** es necesaria porque las sentencias **case** se siguen ejecutando hacia abajo. Esto es, sin un **break** explícito, el flujo de control seguiría secuencialmente a través de las sentencias **case** siguientes. En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia case a otra, por eso se han tenido que poner las sentencias **break**.

Finalmente, puede utilizar la sentencia **default** al final de la sentencia **switch** para manejar los valores que no se han manejado explícitamente por una de las sentencias **case**.

Ejemplo

```
int mes;
. . .
switch (mes) {
      case 1: System.out.println("Enero"); break;
      case 2: System.out.println("Febrero"); break;
      case 3: System.out.println("Marzo"); break;
      case 4: System.out.println("Abril"); break;
      case 5: System.out.println("Mayo"); break;
      case 6: System.out.println("Junio"); break;
      case 7: System.out.println("Julio"); break;
      case 8: System.out.println("Agosto"); break;
      case 9: System.out.println("Septiembre"); break;
      case 10: System.out.println("Octubre"); break;
      case 11: System.out.println("Noviembre"); break;
      case 12: System.out.println("Diciembre"); break;
      default: System.out.println("Ese no es un mes válido!");
}
```

Hay ciertos escenarios en los que es preferible que el control proceda secuencialmente a través de las sentencias case. Como las sentencias case no dependen de ningún orden en particular respecto de las constantes que evalúan para ejecutar las sentencias asociadas al **case**, se puede aprovechar la falta de orden y la no interrupción de la ejecución del código por un **break**, como en este ejemplo, que calcula el número de días de un mes

Ejemplo

```
int mes;
int numeroDias;
. . .
switch (mes) {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
            numeroDias = 31;
            break;
      case 4:
```

```
		case 6:
		case 9:
		case 11:
			numeroDias = 30;
			break;
		case 2:
			if(((ano % 4 == 0) && !(ano % 100==0)) || ano % 400 == 0))
				numeroDias = 29;
			else
				numeroDias = 28;
			break;
}
```

El uso adecuado de cada **break** puede reducir notablemente la complejidad del código.

Ejemplo

```
package seleccionador;
public class Auto {

	public static final int DELUXE = 1;
	public static final int STANDARD = 2;

	void agregarMotor() {
		System.out.println("Agregando Motor...");
	}

	void agregarRuedas() {
		System.out.println("Agregando Ruedas...");
	}

	void agregarRadio() {
		System.out.println("Agregando Radio...");
	}

	void agregarAireAcondicionado() {
		System.out.println("Agregando Aire Acondicionado...");
	}
}
```

Y una clase que utiliza los servicios de Auto

```
public class VerificaAuto1 {
	public static void main(String[] args) {
		Auto a = new Auto();
		int modeloDeAuto = 1;
		switch (modeloDeAuto) {
		case Auto.DELUXE:
			a.agregarAireAcondicionado();
			a.agregarRadio();
			a.agregarRuedas();
			a.agregarMotor();
			break;
		case Auto.STANDARD:
			a.agregarRadio();
```

```
                a.agregarRuedas();
                a.agregarMotor();
                break;
            default:
                a.agregarRuedas();
                a.agregarMotor();
            }
        }
}
```

La salida es

```
Agregando Aire Acondicionado...
Agregando Radio...
Agregando Ruedas...
Agregando Motor...
```

Se puede notar rápidamente que existen varios llamados a métodos repetidos que podrían solucionarse con un replanteo del flujo del programa en los case que pertenecen al **switch**.

La siguiente clase aprovecha la capacidad de ejecución de los case hasta encontrar un break para reducir notablemente el código.

Ejemplo

```
package seleccionador;
public class VerificaAuto2 {

        public static void main(String[] args) {
            Auto a = new Auto();
            int modeloDeAuto = 1;
            switch (modeloDeAuto) {
            case Auto.DELUXE:
                a.agregarAireAcondicionado();
            case Auto.STANDARD:
                a.agregarRadio();
                a.agregarRuedas();
                a.agregarMotor();
            }
        }
}
```

Obteniendo el mismo resultado:

```
Agregando Aire Acondicionado...
Agregando Radio...
Agregando Ruedas...
Agregando Motor...
```

Java 7

Extensión de la sentencia `switch`

Desde la versión 7 de Java se pueden utilizar cadenas constantes como casos de evaluación dentro de la sentencia. Anteriormente, los únicos valores admitidos eran constantes numéricas.

Ejemplo

```
package seleccionador;
public class Cadenas {
	public static void main(String[] args) {
		String cadena = "prueba";

		switch (cadena) {
		case "contante1":
			System.out.println("Entré en constante1");
			break;
		case "contante2":
			System.out.println("Entré en constante2");
			break;
		case "contante3":
			System.out.println("Entré en constante3");
			break;
		case "prueba":
			System.out.println("Entré en prueba");
			break;
		default:
			System.out.println("No entré en ningún case");
		}
	}
}
```

Donde el resultado es

```
Entré en prueba
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia **`switch`** y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el **`switch`**.

La sentencia `for`

Al igual que otras sentencias explicadas previamente, el **`for`** puede ejecutarse sobre una única sentencia o sobre un bloque asociado. La sintaxis del ciclo **`for`** es:

```
for (expresión inicial; expresión booleana; expresión) sentencia;
for (expresión inicial; expresión booleana; expresión){
	[sentencias;]
}
```

Donde:

- **expresión inicial**: es la sentencia que se ejecuta una vez al iniciar el ciclo. Se suele utilizar para inicializar variables
- **expresión booleana**: es una sentencia que determina cuando se termina el ciclo. Esta expresión se evalúa al principio de cada iteración en el ciclo. Cuando la expresión se evalúa a **`false`** el ciclo se termina.
- **Expresión:** se invoca en cada interacción del ciclo a partir del segundo ciclo (inclusive) en adelante.

Cualquiera (o todos) de estos componentes pueden ser una sentencia vacía (un punto y coma)

Ejemplo

```
for (int i = 0; i < 10; i++) {
      System.out.println("Iterando");
}
System.out.println("Valor de I: " + i);
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia **`for`** y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el **`for`**.

Sentencia `while`

Una sentencia **`while`** realiza un ciclo mientras se cumpla una cierta condición (que la expresión que evalúa a un **`boolean`** sea verdadera). La sintaxis general de la sentencia **`while`** es:

```
while (expresión booleana) sentencia;
while (expresión booleana) {
      [sentencias;]
      }
```

Ejemplo

```
int i = 0;
while (i < 10) {
            System.out.println("Iterando …");
            i++;
}
System.out.println("Fin");
```

Al igual que en el **`for`**, en este ciclo se puede ejecutar una sola sentencia o asociarlo a un bloque.

Ejemplo

```
int i = 0;
while (i < 10) {
      System.out.println("¿No terminó todavía?");
      i++;
}
System.out.println("Listo");
```

Por convención la llave abierta '{' se coloca al final de la misma línea donde se encuentra la sentencia **while** y la llave cerrada '}' empieza una nueva línea con sangría (indentación) en la línea en la que se encuentra el **while**.

Sentencia do-while

Es conveniente utilizar la sentencia **do-while** cuando el ciclo debe ejecutarse al menos una vez.

La sintaxis del ciclo **do - while** es:

```
do {
      [sentencias;]
} while (expresión booleana) ;
```

Ejemplo

```
int i = 0;
do {
      System.out.println("Iterando…");
      i++;
} while (i < 10);
System.out.println("Fin");
```

Casos Especiales de Control del Flujo

Etiquetas en Sentencias if

Java admite la declaración de etiquetas a las cuales derivar el control de flujo de un programa siempre y cuando las mismas están asociadas a una sentencia o a un bloque de estas.

Las etiquetas tienen un beneficio dudoso y fomentan un estilo de programación de difícil mantenimiento. Sin embargo, como son cortes de control del flujo de los programas extremadamente rápidos, pueden ser utilizadas con ***sumo*** criterio cuando la velocidad de procesamiento sea crítica.

El Formato de declaración de una etiqueta es el siguiente:

```
Etiqueta: sentencia
```

Ó

```
Etiqueta:{

      }
```

La forma más simple de utilizar etiquetas es asociándolas a sentencias **if** que rompe el control del flujo de programa mediante una sentencia **break**.

Ejemplo

```
package etiquetas;
public class EtiqueteasConIf {
	public static void main(String[] args) {
		int contador = 0;

		// Verificar break en una etiqueta asociada a un bloque
		etiqueta1: {
			System.out.println("antes de etiqueta1");
			if (contador == 0) {
				break etiqueta1;
			}
			System.out.println("después de etiqueta1");
		}

		// Verificar break con una etiqueta asociada a una sentencia if
		System.out.println("antes de  etiqueta2");
		etiqueta2: if (contador == 0) {
			break etiqueta2;
		} else {
			System.out.println(
				"cláusula else; el break no salta a etiqueta2");
		}
		System.out.println("después de etiqueta2");
	}
}
```

La salida que presenta el programa es:

```
antes de etiqueta1
antes de  etiqueta2
después de etiqueta2
```

Sentencias `break` y `continue` en los Ciclos

Como se mostró en la sentencia **`switch`**, la sentencia **`break`** hace que el control del flujo salte a la sentencia siguiente respecto de la actual en ejecución.

El caso más simple de **`break`** interrumpe la sentencia en ejecución. Si se ejecuta dentro de un ciclo que tiene asociado un bloque, como por ejemplo

```
do {
	sentencia;
	if (expresión booleana) {
		break;
	}
	sentencia;
} while (expresión booleana);
Sentencia_fuera_de_bloque;
```

No ejecutará ninguna sentencia más dentro del bloque ni ningún otro ciclo, derivando el flujo del programa a Sentencia_fuera_de_bloque;

Ejemplo

```
package saltos;

public class CiclosConBreak {
	public static void main(String[] args) {
		for (int i = 0; i < 10; i++) {
			System.out.print("\t" + i);
			if(i==4) break;
		}

		System.out.print("\n");

		int j = 0;
		while (j < 10) {
			j++;
			System.out.print("\t" + j);
			if(j==5) break;
		}
		System.out.println("\nTerminé!!");
	}
}
```

La salida es:

```
	0	1	2	3	4
	1	2	3	4	5

Terminé!!
```

La sentencia **continue** es similar pero a diferencia de **break**, hace que se ejecute el próximo ciclo de una iteración.

Ambas sentencias interrumpen el flujo normal de la secuencia de sentencias, pero además, se las puede asociar a una etiqueta de manera que dicha interrupción derive el flujo del programa a la sentencia asociada a la etiqueta

El caso más simple de **continue** es forzar a un nuevo ciclo cuando se ejecuta ignorando el resto de las sentencias que se encuentran en ese ciclo posteriores a él. Si por ejemplo se tiene el siguiente formato:

```
		do {
			 sentencia;
			if (expresión booleana) {
				continue;
			}
			sentencia;
		} while (expresión booleana);
```

Y se ejecuta el **continue**, la sentencia que se encuentra a continuación del **if** no se ejecutará en ese ciclo

Ejemplo

```
package saltos;
public class CiclosConContinue {
	public static void main(String[] args) {
		for (int i = 0; i < 10; i++) {
			if (i % 2 != 0)
				continue;
			System.out.print("\t" + i);
		}
		System.out.print("\n");
		int j = 0;
		while (j < 10) {
			j++;
			if (j % 2 == 0)
				continue;
			System.out.print("\t" + j);
		}
		System.out.println("\nTerminé!!");
	}
}
```

La salida es

```
	0	2	4	6	8
	1	3	5	7	9
Terminé!!
```

Una forma bastante compleja de imprimir números pares e impares que puede sustituirse con código mucho más simple.

Se debe tener cuidado con el uso de estas sentencias. Por ejemplo si se pide a un programador que modifique el código anterior para que sólo se ejecute hasta el número 5, puede decidir modificarlo como se muestra a continuación, empeorando más aún la mala programación.

Ejemplo

```
package saltos;
public class CiclosConBreakYContinue{
	public static void main(String[] args) {
		for (int i = 0; i < 10; i++) {
			if (i % 2 != 0)
				continue;
			System.out.print("\t" + i);
			if(i==4) break;
		}
		System.out.print("\n");
		int j = 0;
		while (j < 10) {
			j++;
			if (j % 2 == 0)
				continue;
			System.out.print("\t" + j);
```

```
                if(j==5) break;
            }
            System.out.println("\nTerminé!!");
        }
}
```

La salida es

```
        0       2       4
        1       3       5
Terminé!!
```

Etiquetas con `break` y `continue`

Hay otra forma de **break** que hace que el flujo de control salte a una sentencia etiquetada, como se mencionó anteriormente. Se puede etiquetar una sentencia utilizando un identificador legal de Java (la etiqueta) seguido por dos puntos (:) antes de la sentencia:

SaltaAqui: algunaSentenciaJava

Para saltar a la sentencia etiquetada se debe utilizar esta forma de la sentencia **break**.

break SaltaAqui;

Las rupturas etiquetadas son una alternativa a la sentencia **goto**, la cual no está soportada por el lenguaje Java.

Por ejemplo, si se tiene el siguiente código

```
afuera :
        do {
                sentencia;
                do {
                        sentencia;
                    if (expresión booleana) {
                            break afuera;
                    }
                     sentencia;
                } while (expresión booleana);
                 sentencia;
        } while (expresión booleana);
         Sentencia_fuera_de_bloque;
```

Al ejecutarse, interrumpe la sentencia asociada a la etiqueta, por lo tanto la próxima sentencia a ejecutar será Sentencia_fuera_de_bloque;

En el siguiente código se muestra otro uso incorrecto de los saltos con la sentencia **break**. El ejemplo muestra a tres sentencias **for** anidadas, pero la disposición de la etiqueta sumada a la condición que ejecuta el **break** con la etiqueta tiene como consecuencia que la segunda sentencia **for** no llegue a incrementarse nunca

Ejemplo

```
package etiquetas;
public class EtiquetasConBreak {
	public static void main(String[] args) {
		for (int h = 0; h < 10; h++) {

			afuera: for (int j = 0; j < 10; j++) {
				System.out.println(j);
				for (int i = 0;; i++) {
					if (i % 2 != 0) {
						System.out.print("\t" + i);
						continue;
					} else if (i > 9) {
						System.out.println();
						break afuera;
					}
				}
			}
		System.out.println("for externo!");
		}
		System.out.println();
		System.out.println("Terminé!!");
	}
}
```

La salida del programa es

```
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
0
	1	3	5	7	9
for externo!
```

```
0
	1	3	5	7	9
for externo!

Terminé!!
```

Cuando se utiliza en **continue** con una etiqueta, se fuerza a un nuevo ciclo de la iteración asociada a la etiqueta, más allá de cuan anidada este la sentencia, irá a la etiqueta seleccionada. Por ejemplo, si se tiene el siguiente código

```
	test:
		do {
			sentencia;
			do {
				sentencia;
				if (expresión booleana) {
					continue test;
				}
				sentencia;
			} while (expresión booleana);
			sentencia;
		} while (expresión booleana);
```

Forzará un nuevo ciclo de la sentencia de iteración más externa, sin importar el estado en que se encuentre el ciclo anidado

Si a la sentencia **continue** se le suma una etiqueta, el control del programa deriva en esta, la cual estará asociada a una instrucción de ciclo forzándolo a un nuevo ciclo asociado a dicha etiqueta.

Ejemplo

```
package etiquetas;
public class EtiquetasConCiclos {
	public static void main(String[] args) {

		afuera: for (int j = 0; j < 10; j++) {
			System.out.print("\t" + j);
			for (int i = 0; i < 10; i++) {
				if (i % 2 != 0) {
					System.out.print("\t" + i);
					continue;
				} else
					continue afuera;
			}
		}
	System.out.println();
	System.out.println("Terminé!!");
	}
}
```

La salida es

```
      0     1     2     3     4     5     6     7     8     9

Terminé!!
```

Vectores

Declaración

En Java, como en cualquier otro lenguaje, un vector agrupa datos del mismo tipo bajo un mismo nombre. La salvedad en este lenguaje es que los datos pueden ser tipos primitivos o referencias.

Por otra parte, los vectores en sí mismos son referenciados. Esto quiere decir que cuando se declara un vector en realidad primero se declara una referencia a éste que debe ser inicializada posteriormente para poder obtener el vector en sí mismo. La forma de declarar referencias a vectores en Java es:

```
char s[];
Point p[];
char[] s;
Point[] p;
```

Se debe tener en cuenta que declaraciones como las anteriores sólo crea espacio para una referencia.

Para obtener el espacio de almacenamiento de los elementos del vector, este deberá ser solicitado en tiempo de ejecución, ya que un vector es un objeto y debe ser creado con **new**.

Creación

Cuando se crea un vector a partir de la referencia a este, se ejecuta el operador **new**. Por ejemplo, para un vector de un tipo primitivo (**char**) el siguiente código muestra la forma de realizarlo:

```
public char[] crearVector() {
      char[] s;
      s = new char[26];
      for ( int i=0; i<26; i++)     {
      s[i] = (char) ('A' + i);
      }
      return s;
}
```

La declaración

```
char[] s;
```

Reserva un espacio en memoria para la referencia s. Para ser más preciso, reserva un espacio en el stack para la referencia, pero todavía no existe un espacio de almacenamiento para los elementos del vector. Cuando se ejecuta la sentencia

```
s = new char[26];
```

Java reserva dinámicamente en memoria el espacio de almacenamiento para los 26 elementos del vector. El siguiente ejemplo muestra cómo se realiza la creación y asignación de elementos de un vector (el detalle de cómo se asignan valores se dará más adelante en este módulo). Notar que el programa también muestra como pasarlo como parámetro o retornarlo como valor.

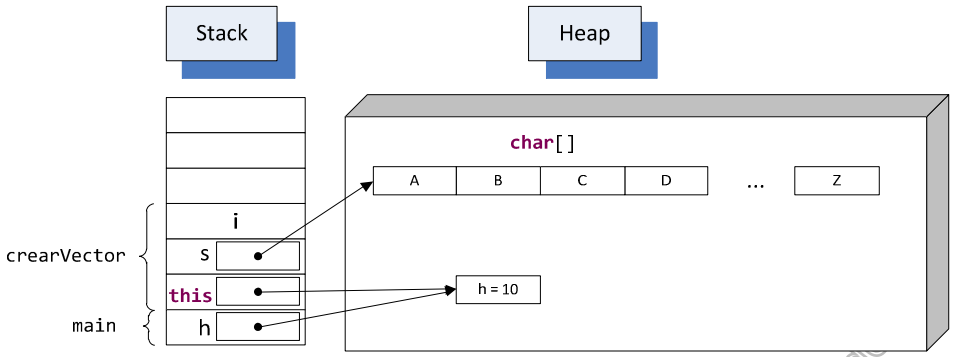
Ejemplo

```
package vectores;
public class VectoresDeTiposPrimitivos {
	private int h = 10;

	public char[] crearVector() {
		char[] s;

		s = new char[26];
		for (int i = 0; i < 26; i++) {
			s[i] = (char) ('A' + i);
		}
		return s;
	}

	public void imprimirVector(char[] vector) {
		System.out.print('<');
		for (int i = 0; i < vector.length; i++) {
			// imprimir un elemento
			System.out.print(vector[i]);
			// Imprimir una coma para delimitarlos
			// si no es el último elemento
			if ((i + 1) < vector.length) {
				System.out.print(", ");
			}
		}
		System.out.print('>');
	}

	public static void main(String[] args) {
		VectoresDeTiposPrimitivos v = new VectoresDeTiposPrimitivos();
	  char[] caracteres = v.crearVector();

	  v.imprimirVector(caracteres);
	  System.out.println();
	}
}
```


La salida del programa es

```
<A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z>
```

Cuando un programa se encuentra en ejecución, el único lugar donde se puede requerir memoria dinámicamente es en el heap, por lo tanto los espacios de almacenamiento quedan como muestra la figura.



El hecho de tener que pedir memoria dinámicamente en un vector tiene consecuencias directas cuando se necesita crear un vector de objetos.

En este caso, cuando se declara el espacio para los elementos del vector, cada uno de estos es a la vez una referencia a cada tipo de objeto a crear, por lo tanto, se deberá ejecutar un **new** por cada elemento. En el código se utiliza la clase Point del paquete awt de Java

Ejemplo

```
package vectores;
import java.awt.Point;

public class VectoresDeObjetos {
	private int h = 10;

	public Point[] crearVector() {
		Point[] p;
		p = new Point[10];
		for (int i = 0; i < 10; i++)
			p[i] = new Point(i, i + 1);
		return p;
	}

	public void imprimirVector(Point[] vector) {
		for (int i = 0; i < vector.length; i++) {
			// imprimir un elemento
			System.out.println(vector[i]);
		}
	}

	public static void main(String[] args) {
		VectoresDeObjetos v = new VectoresDeObjetos();
		Point[] puntos = v.crearVector();

		v.imprimirVector(puntos);
		System.out.println();
	}
}
```

La salida del programa es

```
java.awt.Point[x=0,y=1]
java.awt.Point[x=1,y=2]
java.awt.Point[x=2,y=3]
java.awt.Point[x=3,y=4]
java.awt.Point[x=4,y=5]
java.awt.Point[x=5,y=6]
java.awt.Point[x=6,y=7]
java.awt.Point[x=7,y=8]
java.awt.Point[x=8,y=9]
java.awt.Point[x=9,y=10]
```

Por lo tanto, la declaración

```
Point[] p;
```

Sólo crea una referencia en el stack a un vector cuyos elementos serán a la vez referencias de objetos del tipo Point. La sentencia

```
p = new Point[10];
```

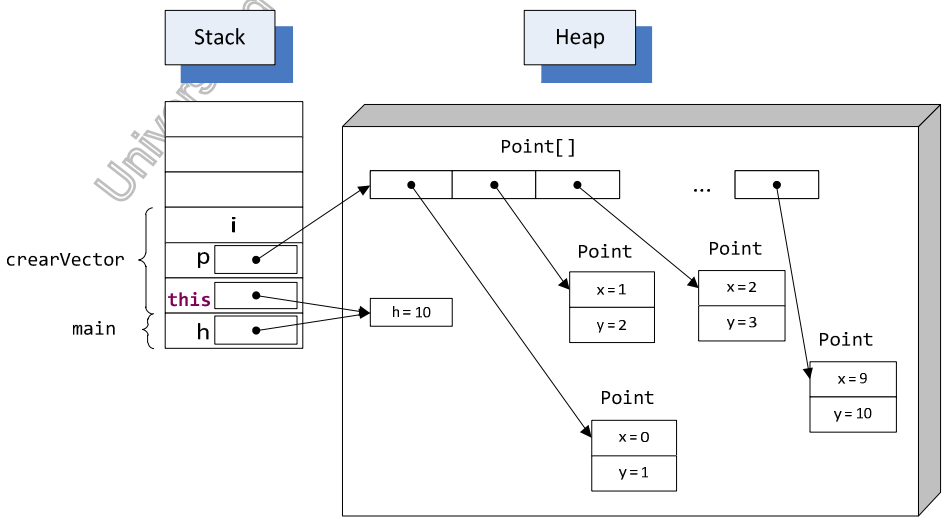
Crea el espacio de almacenamiento para 10 referencias.

El ciclo a continuación ejecuta la creación de cada elemento por el operador **new**

```
for (int i = 0; i < 10; i++)
      p[i] = new Point(i, i + 1);
```

Por lo tanto, es en este momento cuando cada elemento del vector p se va convirtiendo en un objeto.

La siguiente figura es un diagrama de lo expuesto:



Inicialización

Existen dos formas de inicializar los vectores: elemento a elemento (como se mostró anteriormente) o por un bloque de inicialización.

Cuando se inicializa elemento a elemento se necesita crear primero el espacio para almacenar los elementos del vector para luego asignar cada uno de ellos, tanto para los tipos primitivos como para los tipos referenciados. Por ejemplo, para crear e inicializar un vector de elementos primitivos se puede codificar lo siguiente

```
char[] s;
s = new char[4];
s[0] = 'A';
s[1] = 'B';
s[2] = 'C';
s[3] = 'D';
```

Sin embargo, los elementos de un vector de referencia necesitan un poco más de trabajo, ya que se debe crear en cada elemento el objeto acerca del cual se debe almacenar su referencia como un elemento del vector

```
Point[] p = new Point[4];
p[0] = new Point(0, 1);
p[1] = new Point(1, 2);
p[2] = new Point(2, 3);
p[3] = new Point(3, 4);
```

Cuando se inicializa un vector con un bloque de asignación, no se ve claramente que el espacio de almacenamiento se asigna como se lo hace con un objeto. Sin embargo el proceso es el mismo pero en lugar de declararlo explícitamente, el lenguaje lo hace automáticamente. Para el caso de tipos primitivos el código es el que se muestra a continuación. Notar el punto y coma luego del bloque. Esto es lo que diferencia un bloque de asignación de uno de declaración

```
char[] s = {
            'F',
            'G',
            'H',
            'I'
            };
```

Nuevamente, los tipos referenciados exigen el esfuerzo adicional de hacer un **new** para cada elemento de manera de crear así el objeto que será referenciado por éste

```
Point[] p = {
             new Point(4, 5),
             new Point(5, 6),
             new Point(6, 7),
             new Point(7, 8)
             };
```

Matrices o Vectores Bidimensionales

Java es un lenguaje que no fue pensado para realizar grandes operaciones con vectores multidimensionales. En realidad el lenguaje permite manejar dos dimensiones de cualquier tipo de datos, primitivos y referenciados.

El concepto de la declaración de la referencia se mantiene en estos casos, con la salvedad que en lugar de un par de corchetes se utilizan dos, de manera que el primer par especifica la primera dimensión y el otro la segunda.

Los vectores de dos dimensiones o matrices deben pensarse como un vector de vectores, de manera que cada elemento del primer vector mantiene una referencia al primer elemento del vector de la segunda dimensión, por lo tanto, cualquier intento de inicializar la segunda dimensión antes que la primera será un error en tiempo de compilación. Un ejemplo de las posibles declaraciones válidas y el error descripto se muestra a continuación

```
int dosDim[][] = new int [4][];
dosDim[0] = new int[5];
dosDim[1] = new int[5];
int dosDim[][] = new int [][4]; //ilegal
```

Las matrices no necesitan ser cuadradas en su creación. Recordando el concepto de considerarlas vectores de vectores, cada vector de la segunda dimensión se puede crear con los elementos que se necesiten. Por ejemplo

```
dosDim[0] = new int[2];
dosDim[1] = new int[4];
dosDim[2] = new int[6];
dosDim[3] = new int[8];
```

También el lenguaje permite la declaración de matrices cuadradas como se hace tradicionalmente en cualquier lenguaje. Si se necesitara crear un vector de 4 vectores de 5 enteros cada uno, la declaración sería

```
int dosDim [][] = new int[4][5];
```

Límite de un Vector

Lo siguiente deseable al declarar e inicializar un vector, tenga una o dos dimensiones, es recorrerlo. Para ello los principales temas a tener en cuenta son dónde empieza y hasta dónde llega. El primer tema es fácil, ya que todos los vectores en Java comienzan por el elemento cuyo subíndice es 0. El segundo tema no es más difícil, puesto que el lenguaje brinda la posibilidad de acceder a una variable que contiene la cantidad de elementos que el vector tiene la capacidad de almacenar.

Luego, si todos los elementos del vector poseen valores iniciales, recúrrelo se limita a establecer los límites superiores e inferiores y usar un ciclo de iteración. Se debe tener especial cuidado

cuando los elementos del vector sean referencias a objetos que los mismo guarden una referencia válida a un objeto antes de utilizarlo o generará un error en tiempo de ejecución.

Un ejemplo de recorrido de un vector, que sólo contiene los valores iniciales que le da el lenguaje a las variables de tipo entero en cada elemento, es el siguiente:

```
int list[] = new int [10];
      for (int i = 0; i < list.length; i++) {
                  System.out.println(list[i]);
      }
```

Redimensionamiento de un Vector

En Java no se puede cambiar el tamaño de la dimensión establecida en la declaración y creación de un vector. Si se intentara hacer esto, simplemente se crearía un nuevo vector que se asignaría a la variable de referencia y el vector anterior pasaría a ser seleccionable para el recolector de basura (quien se encarga de liberar la memoria no utilizada), perdiendo así su contenido. Este hecho se refleja en el siguiente ejemplo, en el cual el vector de 6 elementos es seleccionable para el recolector de basura cuando se asigna a la variable de referencia el vector de 10 elementos:

```
int miVec[] = new int[6];
miVec = new int[10];
```

Copia de Vectores

La alternativa que brinda al lenguaje al redimensionamiento es hacerlo explícitamente (en lugar de automáticamente) copiando el contenido en un nuevo vector y reasignando su contenido a la referencia. Esto, si bien es un nuevo vector, tiene como resultado final un "redimensionamiento". Para lograrlo, se provee un método que permite copiar un vector en otro a través de lo especificado en sus argumentos. Los argumentos del método arraycopy son los siguientes:

1. Vector fuente de los datos a copiar
2. Elemento desde el cual empieza a leerse para copiar
3. Vector destino de la copia
4. Elemento a partir del cual se empieza a escribir lo que se lee del primer vector
5. Cantidad de elementos a copiar

Por ejemplo

```
// vector original
  int vec1[] = { 1, 2, 3, 4, 5, 6 };

// nuevo vector más largo
  int vec2[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copiar todos los elementos del vector vec1
// en el vector vec2 comenzando por el subíndice 0
  System.arraycopy(vec1, 0, vec2, 0, vec1.length);
```