Memoria (dinámica)

Algoritmos y Estructuras de Datos

Hoja de ruta

La idea de la clase de hoy es charlar de:

- La relación entre los TADs y la OOP.
- Tipos complejos
- ¿Dónde se ubican los objetos y variables en memoria?
- *Stack* y *heap*
- ¿Cómo sabemos si un número en memoria es una variable o una referencia?
- *Aliasing*
- La igualdad en Java

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos
- Es abstracto ya que, para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos
- Es abstracto ya que, para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.
- Describe el "qué" y no el "cómo"

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos
- Es abstracto ya que, para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.
- Describe el "qué" y no el "cómo"
- Como usuarios nos interesa la "interfaz", o sea el contrato.

Breve repaso de TADs

En la teórica de esta semana vieron TADs.

- TAD quiere decir Tipo Abstracto de Datos
- ¿Qué es un Tipo Abstracto de Datos?
- Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos
- Es abstracto ya que, para utilizarlos, no se necesita conocer los detalles de la representación interna ni cómo están implementadas sus operaciones.
- Describe el "qué" y no el "cómo"
- Como usuarios nos interesa la "interfaz", o sea el contrato.
- Son una forma de modularizar a nivel de los datos

Breve repaso de clases

Las clases nos permiten agrupar datos y comportamiento (¿a qué se asemejan?).
Los atributos privados definen el *estado interno* del objeto. Los métodos públicos definen su *comportamiento*.
Dos conceptos clave se desprenden de este paradigma:

- Abstracción: ...
- Encapsulamiento: ...

Breve repaso de clases

Las clases nos permiten agrupar datos y comportamiento (¿a qué se asemejan?).
Los atributos privados definen el *estado interno* del objeto. Los métodos públicos definen su *comportamiento*.
Dos conceptos clave se desprenden de este paradigma:

- Abstracción: centrarse en el comportamiento común, no en su implementación.
- Encapsulamiento: ocultar el funcionamiento interno (que podría cambiar)

Breve repaso de clases

Las clases nos permiten agrupar datos y comportamiento (¿a qué se asemejan?).
Los atributos privados definen el *estado interno* del objeto. Los métodos públicos definen su *comportamiento*.
Dos conceptos clave se desprenden de este paradigma:

- Abstracción: centrarse en el comportamiento común, no en su implementación.
- Encapsulamiento: ocultar el funcionamiento interno (que podría cambiar)

No necesariamente hay un mapeo 1:1 entre clases y TADs

Operando con tipos complejos

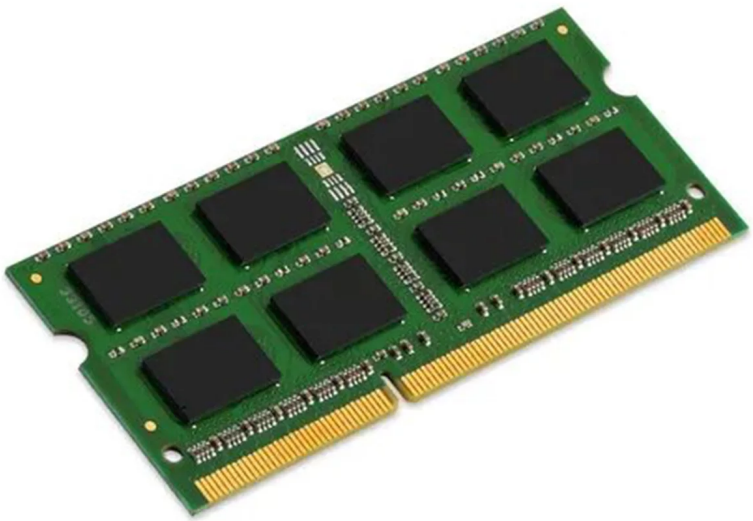
Un **tipo complejo** es aquel que es una combinación de otros tipos existentes. Los arreglos son un ejemplo de un tipo complejo.
Se diferencian de los **tipos primitivos** (int, char, etc.) en que pueden contener a otros tipos de datos y, como tales, su manejo en **memoria** es distinto.

Operando con tipos complejos

Un **tipo complejo** es aquel que es una combinación de otros tipos existentes. Los arreglos son un ejemplo de un tipo complejo.
Se diferencian de los **tipos primitivos** (int, char, etc.) en que pueden contener a otros tipos de datos y, como tales, su manejo en **memoria** es distinto.
Todas las clases que definan serán tipos complejos.

Memoria

Esto es una memoria:



Memoria

- Muchos de los *bugs* que suele haber al programar vienen de un mal manejo de la memoria.
- En los próximos talleres van a tener problemas cada vez más díficiles que requieren entender bien estos conceptos.
- Veremos lo mínimo necesario para la vida de un programador.
- Más en: Sistemas Digitales, Arquitectura y Organización de Computadoras, Sistemas Operativos.

Memoria

Todo el contenido de las variables ocupa "espacio" en la "memoria". Los sistemas operativos suelen separar la memoria en dos grandes regiones:

Contexto local ⇒ en la pila (*stack*)

Las variables locales viven únicamente dentro del scope local. Aquí se guardan tipos primitivos y *referencias* a arreglos y objetos.

Dinámica (manual) ⇒ en el *heap*

En el heap se almacenan los objetos. La memoria de un objeto se libera cuando se vuelve inalcanzable.

Ejemplo de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);
}
int doble(int x) {
    int res = 2 * x;
    return res;
}
```

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    // <---
    int d = doble(x);
}
int doble(int x) {
    int res = 2 * x;
    return res;
}
```

Stack	
x (ejemplo)	8

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);// <---
}
int doble(int x) {
    // <---
    int res = 2 * x;
    return res;
}
```

Stack	
x (doble)	8
x (ejemplo)	8

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);// <---
}
int doble(int x) {
    // <---
    int res = 2 * x;
    return res;
}
```

Stack	
x (doble)	8
x (ejemplo)	8

¿Qué pasaría si el código de `doble` modificara la variable `x`?

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);// <---
}
int doble(int x) {
    int res = 2 * x;
    // <---
    return res;
}
```

Stack	
res (doble)	16
x (doble)	8
x (ejemplo)	8

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);
    // <---
}
int doble(int x) {
    int res = 2 * x;
    return res;
}
```

Stack	
d (ejemplo)	16
x (ejemplo)	8

Seguimiento de código

```
void ejemplo() {
    int x = 8;
    int d = doble(x);
    // <---
}
int doble(int x) {
    int res = 2 * x;
    return res;
}
```

Stack	
d (ejemplo)	16
x (ejemplo)	8

¿Qué pasa con la memoria al llamar una función recursiva?

Un ejemplo distinto

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
    }
}
```

Un ejemplo distinto

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
    }
}
```

¿Dónde se guardará el arreglo? ¿Qué hace el `new`?

Un ejemplo distinto

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
    }
}
```

¿Dónde se guardará el arreglo? ¿Qué hace el `new`?
`new` reserva espacio en el heap para el arreglo y devuelve una referencia a la posición de memoria donde se guardó.

Seguimiento

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    // <---
    duplicar(xs);
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
    }
}
```

Stack	
xs (ejemplo)	1104

Heap	
Posición	Valor
...	
1104	[15, 8, 42]
...	

Seguimiento

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);// <---
}
void duplicar(int[] secu) {
    int i = 0;
    // <---
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
    }
}
```

Stack	
i (duplicar)	0
secu (duplicar)	1104
xs (ejemplo)	1104

Heap	
Posición	Valor
...	
1104	[15, 8, 42]
...	

Seguimiento

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);// <---
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
        // <---
    }
}
```

Stack	
i (duplicar)	1
secu (duplicar)	1104
xs (ejemplo)	1104

Heap	
Posición	Valor
...	
1104	[30, 8, 42]
...	

Seguimiento

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);// <---
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
        // <---
    }
}
```

Stack	
i (duplicar)	2
secu (duplicar)	1104
xs (ejemplo)	1104

Heap	
Posición	Valor
...	
1104	[30, 16, 42]
...	

Seguimiento

```
void ejemplo() {
    int[] xs = new int[]{15, 8, 42};
    duplicar(xs);// <---
}
void duplicar(int[] secu) {
    int i = 0;
    while (i < secu.length) {
        secu[i] = 2 * secu[i];
        i++;
        // <---
    }
}
```

Stack	
i (duplicar)	3
secu (duplicar)	1104
xs (ejemplo)	1104

Heap	
Posición	Valor
...	
1104	[30, 16, 84]
...	

Más punteros

```
class Estudiante {
    int edad;
    int[] notas;
    Estudiante (int e, int[] n) {
        edad = e;
        notas = n;
    }
}
void ejemplo() {
    Estudiante[] estudiantes = new Estudiante[2];
    estudiantes[0] = new Estudiante(20, new int[]{8, 7, 9});
    estudiantes[1] = new Estudiante(19, new int[]{6, 8, 4});
}
```

Más punteros

```
class Estudiante {
    int edad;
    int[] notas;
    Estudiante (int e, int[] n) {
        edad = e;
        notas = n;
    }
}
void ejemplo() {
    Estudiante[] estudiantes = new Estudiante[2];
    estudiantes[0] = new Estudiante(20, new int[]{8, 7, 9});
    estudiantes[1] = new Estudiante(19, new int[]{6, 8, 4});
}
```

¿Cómo queda organizada la memoria?

Seguimiento

```
class Estudiante {
    int edad;
    int[] notas;
    ...
}
void ejemplo() {
    Estudiante[] estudiantes = new Estudiante[2];
    // <---
    estudiantes[0] = new Estudiante(20, new int[]{8, 7, 9});
    estudiantes[1] = new Estudiante(19, new int[]{6, 8, 4});
}
```

Stack	
estudiantes	3054

Heap	
3054	[0, 0]

Seguimiento

```
class Estudiante {
    int edad;
    int[] notas;
    ...
}
void ejemplo() {
    Estudiante[] estudiantes = new Estudiante[2];
    estudiantes[0] = new Estudiante(20, new int[]{8, 7, 9});
    // <---
    estudiantes[1] = new Estudiante(19, new int[]{6, 8, 4});
}
```

Stack	
estudiantes	3054

Heap	
3054	[9416, 0]
...	
3107	[8, 7, 9]
...	
9416	Estudiante(edad: 20, notas: 3107)

Seguimiento

```
class Estudiante {
    int edad;
    int[] notas;
    ...
}
void ejemplo() {
    Estudiante[] estudiantes = new Estudiante[2];
    estudiantes[0] = new Estudiante(20, new int[]{8, 7, 9});
    estudiantes[1] = new Estudiante(19, new int[]{6, 8, 4});
    // <---
}
```

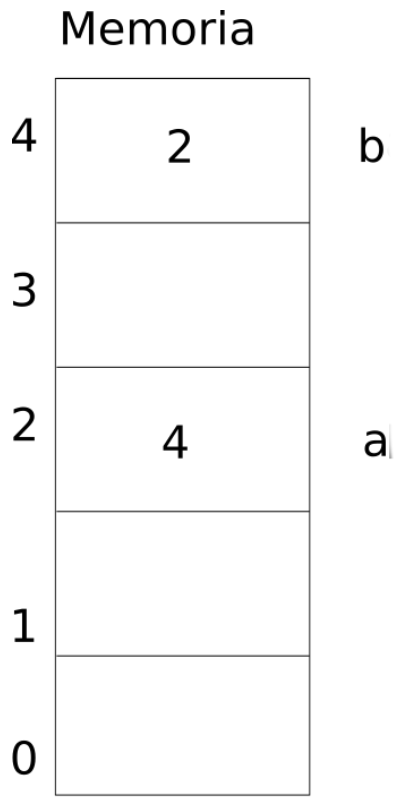
Stack	
estudiantes	3054

Heap	
3054	[9416, 5331]
...	
3107	[8, 7, 9]
...	
5331	Estudiante(edad: 19, notas: 6883)
...	
6883	[6, 8, 4]
...	
9416	Estudiante(edad: 20, notas: 3107)

La memoria son solo números

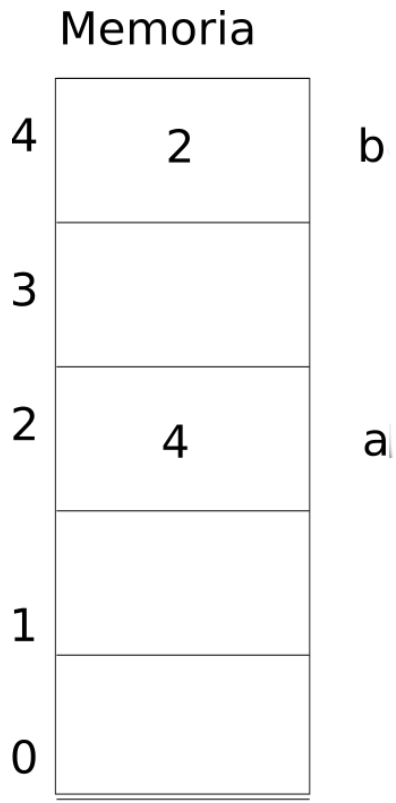
Supongamos que tenemos el siguiente código, y la memoria queda organizada de la siguiente manera:

```
void ejemplo3() {
    int a = 4;
    int b = 2;
}
```



¿Cómo lo interpreto? Si cambio alguno, ¿qué pasa con el otro?

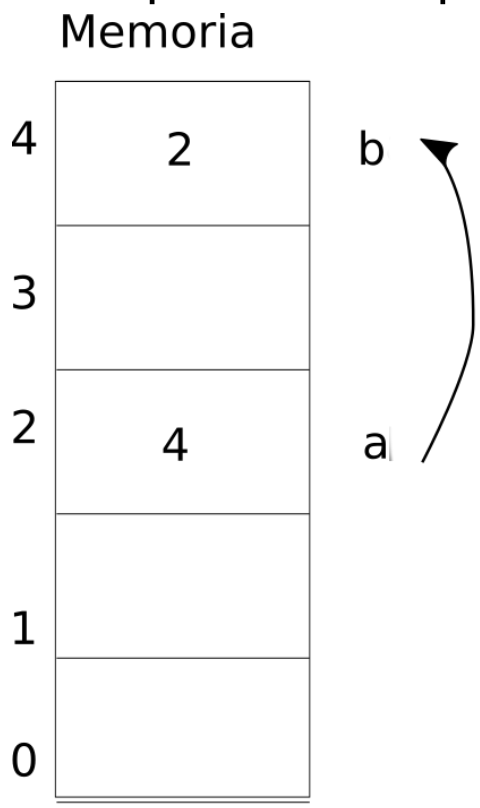
La memoria son solo números



¿Cómo lo interpreto? Si cambio alguno, ¿qué pasa con el otro?

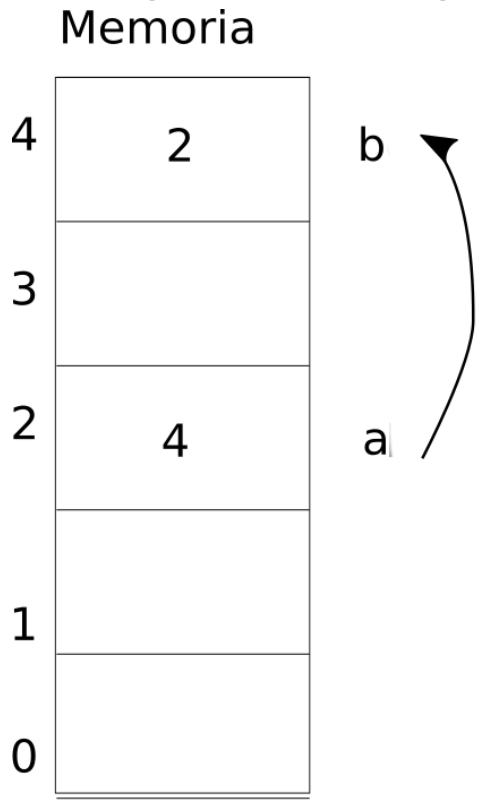
La memoria son solo números

Java podría interpretarlo así (o viceversa):



La memoria son solo números

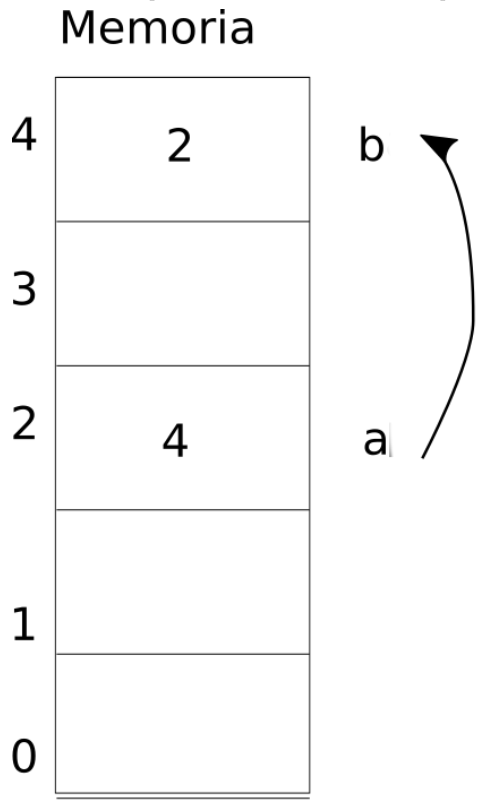
Java podría interpretarlo así (o viceversa):



¿Cómo sabe si es un número o una dirección de memoria?

La memoria son solo números

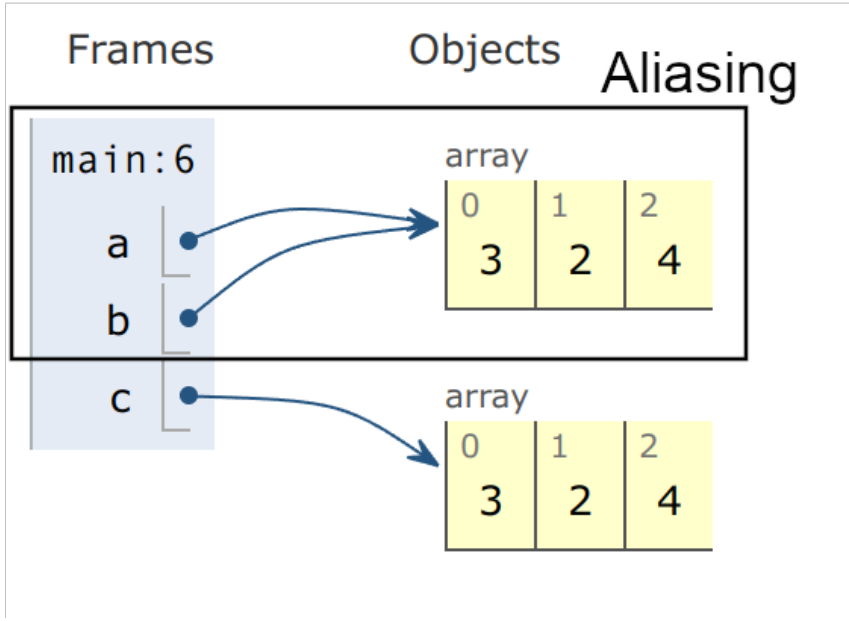
Java podría interpretarlo así (o viceversa):



¿Cómo sabe si es un número o una dirección de memoria?
¡Conociendo los **tipos**! Los tipos le dicen al compilador cómo interpretar y cuánto espacio ocupa cada variable.

Aliasing

```
public class Main {
    public void main() {
        int[] a = new int[]{3, 2, 4};
        int[] b = a;
        int[] c = a.clone();
    }
}
```



- Cuando dos variables referencian al mismo valor, decimos que hay *aliasing* entre ellas.
- En el caso de objetos/arreglos, lo podemos verificar con == (¿por qué?).
- El aliasing es un problema cuando se trata de objetos mutables.
- **Si exponemos referencias a los atributos de nuestra clase**, nos exponemos a que el usuario de la misma modifique los valores internos sin nuestro consentimiento.

Recuerden:

- Si el tipo es primitivo, `a == b` dice si "a" y "b" son *iguales*.
- Si el tipo es un objeto/arreglo, `a == b` dice si "a" y "b" son alias de un mismo objeto.
- Si el tipo es un objeto, `a.equals(b)` dice si "a" y "b" son *iguales*.

Recursos

- Visualizador de memoria en Java (con ejemplos)
- Otro visualizador, con varios lenguajes

Algunos ejemplos

- Uso del stack con factorial recursivo
- Ejemplo de heap con una clase básica