Programación Orientada a Objetos en Java

Algoritmos y Estructuras de Datos

Modelando problemas más complejos

Un problema común en la vida cotidiana es el de compartir gastos. Alicia y Bruno comparten un departamento y deciden anotar los gastos que realizan. Esto implica llevar un registro de los gastos de cada uno y saber cuánto le corresponde pagar a cada uno. Además, es importante saber quién debe compensar a quién.

Modelando problemas más complejos

Un problema común en la vida cotidiana es el de compartir gastos. Alicia y Bruno comparten un departamento y deciden anotar los gastos que realizan. Esto implica llevar un registro de los gastos de cada uno y saber cuánto le corresponde pagar a cada uno. Además, es importante saber quién debe compensar a quién.

De procedimientos a objetos

A veces los procedimientos resultan insuficientes para solucionar problemas complejos que requieren mantener un **estado**, el cual va cambiando a través de **operaciones (públicas)** invocadas por el usuario.

La especificación de este tipo de problemas lo verán la semana siguiente. Hoy nos concentraremos en su **implementación**.

Programación Orientada a Objetos

La programación orientada a objetos es un paradigma de programación que se basa en el concepto de **objetos**. Un objeto es una *entidad* que agrupa:

- **Estado:** atributos que describen al objeto.
- **Comportamiento:** operaciones (métodos) que el objeto puede realizar.

Los objetos modifican su estado interno a través de sus métodos públicos.

Programación Orientada a Objetos

La programación orientada a objetos es un paradigma de programación que se basa en el concepto de **objetos**. Un objeto es una *entidad* que agrupa:

- **Estado:** atributos que describen al objeto.
- **Comportamiento:** operaciones (métodos) que el objeto puede realizar.

Los objetos modifican su estado interno a través de sus métodos públicos.

¿Qué utilizaremos?

- Vamos a utilizar una pequeña parte de este paradigma.
- Utilizaremos clases de `Java` para modelar objetos.

Conceptos importantes

- **Abstracción**: Énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?".

Conceptos importantes

- **Abstracción**: Énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?".
- **Encapsulamiento**: Ocultar el funcionamiento interno (que podría cambiar), centrándonos en lo que necesita el usuario.

Conceptos importantes

- **Abstracción**: Énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?".
- **Encapsulamiento**: Ocultar el funcionamiento interno (que podría cambiar), centrándonos en lo que necesita el usuario.
- **Modularización**: Dividir el problema en partes más pequeñas y manejables.

Conceptos importantes

- **Abstracción**: Énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?".
- **Encapsulamiento**: Ocultar el funcionamiento interno (que podría cambiar), centrándonos en lo que necesita el usuario.
- **Modularización**: Dividir el problema en partes más pequeñas y manejables.
- ⚠Usamos una partecita y algunos conceptos....

Conceptos importantes

- **Abstracción**: Énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?".
- **Encapsulamiento**: Ocultar el funcionamiento interno (que podría cambiar), centrándonos en lo que necesita el usuario.
- **Modularización**: Dividir el problema en partes más pequeñas y manejables.
- ⚠Usamos una partecita y algunos conceptos....
- Más en ingeniería de software

Clases

Vamos a utilizar clases !

Definiendo clases

Una instancia de una clase es un objeto. Vamos a definir una clase en varias etapas:

1. Vamos a declarar los **métodos públicos**. O sea sus "operaciones". Conforman la *interfaz* de la clase.

Definiendo clases

Una instancia de una clase es un objeto. Vamos a definir una clase en varias etapas:

1. Vamos a declarar los **métodos públicos**. O sea sus "operaciones". Conforman la *interfaz* de la clase.
2. Vamos a declarar los **atributos privados** de la clase. Su "estado", oculto al usuario.

Definiendo clases

Una instancia de una clase es un objeto. Vamos a definir una clase en varias etapas:

1. Vamos a declarar los **métodos públicos**. O sea sus "operaciones". Conforman la *interfaz* de la clase.
2. Vamos a declarar los **atributos privados** de la clase. Su "estado", oculto al usuario.
3. Vamos a implementar los métodos públicos utilizando los atributos privados. Esto define el *comportamiento* de la clase.

Definiendo clases

Una instancia de una clase es un objeto. Vamos a definir una clase en varias etapas:

1. Vamos a declarar los **métodos públicos**. O sea sus "operaciones". Conforman la *interfaz* de la clase.
2. Vamos a declarar los **atributos privados** de la clase. Su "estado", oculto al usuario.
3. Vamos a implementar los métodos públicos utilizando los atributos privados. Esto define el *comportamiento* de la clase.
4. De particular importancia es el *constructor* de la clase (o los constructores), que permiten inicializar una instancia de la clase (es decir, un objeto).

Definiendo clases

Una instancia de una clase es un objeto. Vamos a definir una clase en varias etapas:

1. Vamos a declarar los **métodos públicos**. O sea sus "operaciones". Conforman la *interfaz* de la clase.
2. Vamos a declarar los **atributos privados** de la clase. Su "estado", oculto al usuario.
3. Vamos a implementar los métodos públicos utilizando los atributos privados. Esto define el *comportamiento* de la clase.
4. De particular importancia es el *constructor* de la clase (o los constructores), que permiten inicializar una instancia de la clase (es decir, un objeto).

- Se puede programar con clases pero no modularmente.

Sistema Cuanto Te Debo (CTD)

Un problema común en la vida cotidiana es el de compartir gastos. Alicia y Bruno comparten un departamento y deciden anotar los gastos que realizan. Esto implica llevar un registro de los gastos de cada uno y saber cuánto le corresponde pagar a cada uno. Además, es importante saber quién debe compensar a quién.

Sistema Cuanto Te Debo (CTD)

Un problema común en la vida cotidiana es el de compartir gastos. Alicia y Bruno comparten un departamento y deciden anotar los gastos que realizan. Esto implica llevar un registro de los gastos de cada uno y saber cuánto le corresponde pagar a cada uno. Además, es importante saber quién debe compensar a quién.

Necesitamos:

- Conocer los gastos de cada participante.
- Saber, hasta el momento, cuánto debe cada participante al otro.
- Saber qué participante debe compensar.

```java
public class CTD {
  private int[] gastosPersona1;
  private int[] gastosPersona2;

    public CTD() {
        //{...}
    }

  public void anotarGasto(int persona, int gasto) {
        //{...}
    }

  private int[] agregarGasto(int[] gasto_persona, int gasto) {
        //{...}
    }

  public int gastos(int persona) {
        //{...}
  }

  public int quienSeTieneQuePoner() {
        //{...}
  }
}
```

Si tuviéramos esta clase en Java, ¿sabríamos cómo usarla?

- ¿Qué gastos tiene una persona al iniciar?

Si tuviéramos esta clase en Java, ¿sabríamos cómo usarla?

- ¿Qué gastos tiene una persona al iniciar?
- ¿Qué devuelve `quienSeTieneQuePoner`?

Si tuviéramos esta clase en Java, ¿sabríamos cómo usarla?

- ¿Qué gastos tiene una persona al iniciar?
- ¿Qué devuelve `quienSeTieneQuePoner`?
- ¡Hace falta una especificación!

Clases e instancias

```
CTD c1 = new CTD();

c1.anotarGasto(0, 5);
c1.anotarGasto(1, 3);
c1.anotarGasto(0, 1);
c1.anotarGasto(0, 3);

c1.gastos(0); // 9
c1.gastos(1); // 3
```

```
CTD c2 = new CTD();

c2.anotarGasto(0, 8);
c2.anotarGasto(1, 2);
c2.anotarGasto(1, 3);
c2.anotarGasto(0, 7);

c2.gastos(0); // 15
c2.gastos(1); // 5
```

Clases e instancias

```
CTD c1 = new CTD();

c1.anotarGasto(0, 5);
c1.anotarGasto(1, 3);
c1.anotarGasto(0, 1);
c1.anotarGasto(0, 3);

c1.gastos(0); // 9
c1.gastos(1); // 3
```

```
CTD c2 = new CTD();

c2.anotarGasto(0, 8);
c2.anotarGasto(1, 2);
c2.anotarGasto(1, 3);
c2.anotarGasto(0, 7);

c2.gastos(0); // 15
c2.gastos(1); // 5
```

`class CTD` es la clase.

`c1` y `c2` son **objetos** distintos, pero ambos son **instancias** de la clase CTD.

Para que el comportamiento de la clase pueda llevarse a cabo, hay que implementarla.

La **implementación** está dada por:

- La **representación interna**: un conjunto de `atributos` (variables) que determina el estado interno de la instancia.
- Un conjunto de **algoritmos** que implementan cada una de las operaciones de la interfaz (métodos públicos), consultando y modificando las variables de la representación interna.

Declaración de atributos privados

```
class CTD {
  /* ... */
  private int[] gastosPersona1;
  private int[] gastosPersona2;

};
```

Estados internos

```
c1.anotarGasto(0, 5);
c1.anotarGasto(1, 3);
c1.anotarGasto(0, 1);
c1.anotarGasto(0, 3);

// Estado interno de c1
c1.gastosPersona1 == [5, 1, 3]
c1.gastosPersona2 == [3]
```

```
c2.anotarGasto(0, 8);
c2.anotarGasto(1, 2);
c2.anotarGasto(1, 3);
c2.anotarGasto(0, 7);

// Estado interno de c2
c2.gastosPersona1 == [8, 7]
c2.gastosPersona2 == [2, 3]
```

Estados internos

```
c1.anotarGasto(0, 5);
c1.anotarGasto(1, 3);
c1.anotarGasto(0, 1);
c1.anotarGasto(0, 3);

// Estado interno de c1
c1.gastosPersona1 == [5, 1, 3]
c1.gastosPersona2 == [3]
```

```
c2.anotarGasto(0, 8);
c2.anotarGasto(1, 2);
c2.anotarGasto(1, 3);
c2.anotarGasto(0, 7);

// Estado interno de c2
c2.gastosPersona1 == [8, 7]
c2.gastosPersona2 == [2, 3]
```

El estado interno **no es accesible** desde afuera del objeto (encapsulamiento).

⚠Un objeto siempre debe ser usado mediante su interfaz (métodos públicos).

Comportamiento

¿Cómo definimos comportamiento para las instancias?

Comportamiento

¿Cómo definimos comportamiento para las instancias?
A través de **métodos públicos** que acceden a los atributos privados:

```
public void anotarGasto(int persona, int gasto) {
  if (persona == 0) {
    gastosPersona1 = agregarGasto(gastosPersona1, gasto);
  } else {
    gastosPersona2 = agregarGasto(gastosPersona2, gasto);
  }
}
```

Comportamiento

Los métodos pueden utilizar métodos privados; es decir, métodos "internos" que sólo serán accesibles desde métodos de la clase.

Comportamiento

Los métodos pueden utilizar métodos privados; es decir, métodos "internos" que sólo serán accesibles desde métodos de la clase.
Pensemos un segundo cómo implementarían `agregarGasto`

Comportamiento

Los métodos pueden utilizar métodos privados; es decir, métodos "internos" que sólo serán accesibles desde métodos de la clase.
Pensemos un segundo cómo implementarían `agregarGasto`
⚠**Ojo:** ¡los arreglos de Java no son redimensionables!

Comportamiento

Los métodos pueden utilizar métodos privados; es decir, métodos "internos" que sólo serán accesibles desde métodos de la clase.

Pensemos un segundo cómo implementarían `agregarGasto`

⚠**Ojo:** ¡los arreglos de Java no son redimensionables!

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

Ejemplo

```
public static void main (String[] args) {
    CTD c1 = new CTD();
    CTD c2 = new CTD();
                        // <---
    c1.anotarGasto(0, 3);
    c1.anotarGasto(1, 1);
    c2.anotarGasto(1, 5);
}
```

Contexto

c1.gastosPersona1	[]
c1.gastosPersona2	[]
c2.gastosPersona1	[]
c2.gastosPersona2	[]

Ejemplo

```
public static void main (String[] args) {
    CTD c1 = new CTD();
    CTD c2 = new CTD();
    c1.anotarGasto(0, 3);
                        // <---
    c1.anotarGasto(1, 1);
    c2.anotarGasto(1, 5);

}
```

Contexto

c1.gastosPersona1	[3]
c1.gastosPersona2	[]
c2.gastosPersona1	[]
c2.gastosPersona2	[]

Ejemplo

```
public static void main (String[] args) {
    CTD c1 = new CTD();
    CTD c2 = new CTD();
    c1.anotarGasto(0, 3);
    c1.anotarGasto(1, 1);
                        // <---
    c2.anotarGasto(1, 5);

}
```

Contexto

c1.gastosPersona1	[3]
c1.gastosPersona2	[1]
c2.gastosPersona1	[]
c2.gastosPersona2	[]

Ejemplo

```
public static void main (String[] args) {
    CTD c1 = new CTD();
    CTD c2 = new CTD();
    c1.anotarGasto(0, 3);
    c1.anotarGasto(1, 1);
    c2.anotarGasto(1, 5);
                        // <---

}
```

Contexto

c1.gastosPersona1	[3]
c1.gastosPersona2	[1]
c2.gastosPersona1	[]
c2.gastosPersona2	[5]

El resto de los ingredientes

La interfaz de CTD tiene un método para ver el gasto de los compañeros:

```
public int gastos(int persona) {
  int total = 0;
  if (persona == 0) {
    for (int gasto : gastosPersona1) {
      total += gasto;
    }
  } else {
    for (int gasto : gastosPersona2) {
      total += gasto;
    }
  }
  return total;
}
```

El resto de los ingredientes

Pero los miembros privados de una clase no son accessibles desde afuera:

```
void ejemplo() {
  CTD c = new CTD();
  System.out.println(c.gastosPersona1);
    // error: The field c.gastosPersona1 is not visible.
}
```

El resto de los ingredientes

Debemos usar el método público de la interfaz para acceder a los gastos:

```
void ejemplo() {
  CTD c = new CTD() ;
  System.out.println(c.gastos(1));
}
```

Constructor

- Veamos nuevamente agregarGasto.

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

Constructor

- Veamos nuevamente agregarGasto.

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

- ¿Qué contiene gasto_persona al principio?

Constructor

- Veamos nuevamente agregarGasto.

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

- ¿Qué contiene `gasto_persona` al principio?
- Los constructores son funciones especiales para inicializar una nueva instancia de una clase: deben definir los valores iniciales de los atributos privados.

Constructor

- Veamos nuevamente agregarGasto.

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

- ¿Qué contiene `gasto_persona` al principio?
- Los constructores son funciones especiales para inicializar una nueva instancia de una clase: deben definir los valores iniciales de los atributos privados.
- Se escriben con el nombre de la clase.

Constructor

- Veamos nuevamente agregarGasto.

```
private int[] agregarGasto(int[] gasto_persona, int gasto) {
  int[] gasto_persona_nuevo = new int[gasto_persona.length + 1];
  for (int i = 0; i < gasto_persona.length; i++) {
    gasto_persona_nuevo[i] = gasto_persona[i];
  }
  gasto_persona_nuevo[gasto_persona_nuevo.length - 1] = gasto;
  return gasto_persona_nuevo;
}
```

- ¿Qué contiene `gasto_persona` al principio?
- Los constructores son funciones especiales para inicializar una nueva instancia de una clase: deben definir los valores iniciales de los atributos privados.
- Se escriben con el nombre de la clase.
- No tienen tipo de retorno (está implícito; en realidad, la clase es el "tipo").

Constructor

```
public class CTD {
    private int[] gastosPersona1;
    private int[] gastosPersona2;

    public CTD() {
        gastosPersona1 = new int[0];
        gastosPersona2 = new int[0];
    }

    /* ... */
}

void ejemplo() {
    CTD c = new CTD();
}
```

Constructor por copia

A veces nos interesa crear una nueva instancia de la clase que sea una *copia* de otra instancia ya creada (es decir, que posea los mismos valores en su representación interna).
No obstante, ejecutar `CTD c1 = c2` trae problemas: ambos objetos apuntarían a la misma representación interna (*aliasing*). Modificar uno modificaría al otro.

Constructor por copia

A veces nos interesa crear una nueva instancia de la clase que sea una *copia* de otra instancia ya creada (es decir, que posea los mismos valores en su representación interna).
No obstante, ejecutar `CTD c1 = c2` trae problemas: ambos objetos apuntarían a la misma representación interna (*aliasing*). Modificar uno modificaría al otro.
Por lo tanto, para copiar, necesitamos definir lo que se llama *constructor por copia*:

```
public CTD(CTD otro) {
  // Copia el CTD, copiando en "cascada"
  gastosPersona1 = otro.gastosPersona1.clone();
  gastosPersona2 = otro.gastosPersona2.clone();
}
```

Luego, para crear una instancia nueva que sea una copia de la otra:

```
{
    CTD c1 = new CTD(c2)
}
```

Aliasing

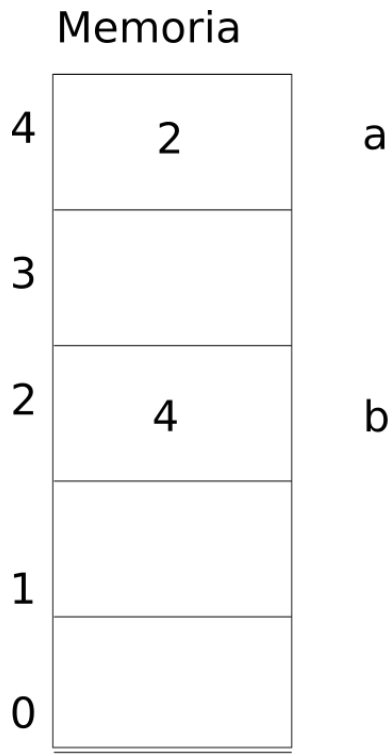
¿Qué recuerdan de IP?

Aliasing

¿Qué recuerdan de IP?

- 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 (modificables).
- Si exponemos referencias a los atributos de nuestra clase, nos exponemos a que el usuario de la misma nos modifique sin que nos demos cuenta (ojo en el taller).

La memoria son solo números



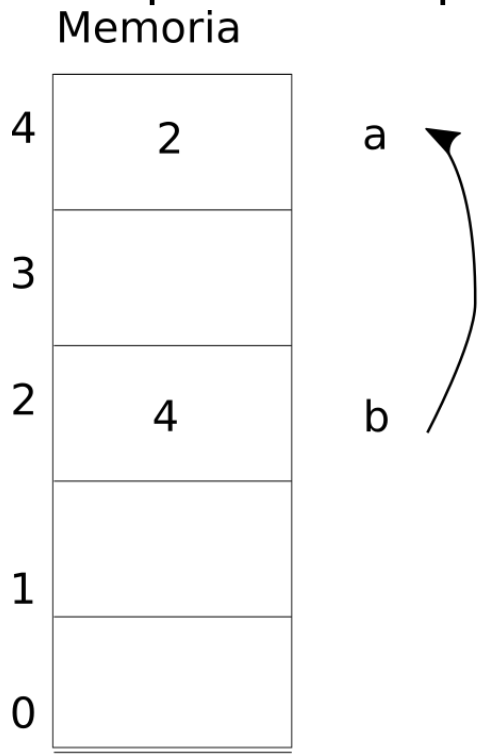
Sean estos espacios en memoria y **a** y **b** variables que apuntan a su espacio correspondiente
¿Cómo lo interpreto?

La memoria son solo números

Java podría interpretarlo así:

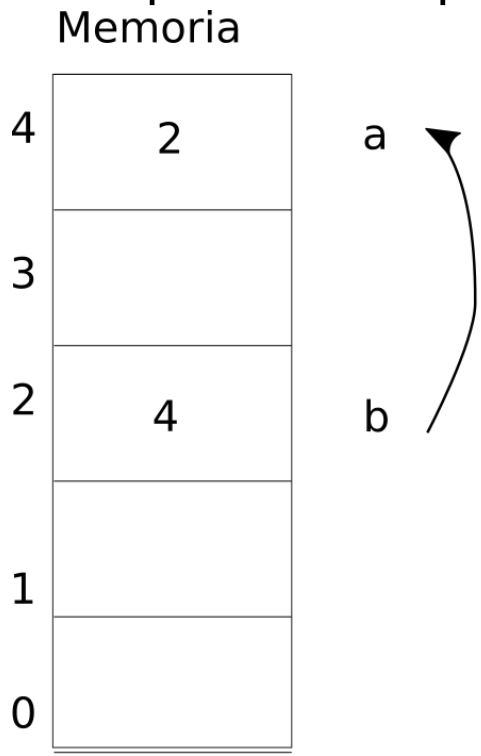
La memoria son solo números

Java podría interpretarlo así:



La memoria son solo números

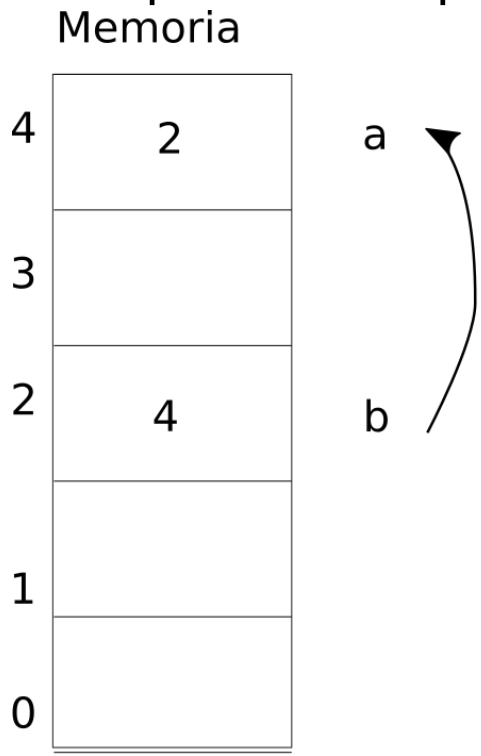
Java podría interpretarlo así:



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

La memoria son solo números

Java podría interpretarlo así:



¿Cómo sabe si es un número o una dirección de memoria?
¡Conociendo los **tipos**!
Si la variable **a** es un *int* y la variable **b** una referencia a un *int*, tenemos este resultado.

Recuerden:

- Si el tipo es primitivo, `a == b` dice si "a" y "b" son valores *iguales*.
- Si el tipo es un objeto/arreglo, `a == b` dice si "a" y "b" son alias de un mismo objeto (apuntan a la misma dirección de memoria).
- Si el tipo es un objeto, `a.equals(b)` dice si "a" y "b" son *iguales* o equivalentes.

Recuerden:

- Si el tipo es primitivo, `a == b` dice si "a" y "b" son valores *iguales*.
- Si el tipo es un objeto/arreglo, `a == b` dice si "a" y "b" son alias de un mismo objeto (apuntan a la misma dirección de memoria).
- Si el tipo es un objeto, `a.equals(b)` dice si "a" y "b" son *iguales* o equivalentes.

¡Muy importante declarar el `equals` cuando escribimos clases nuevas!

Método equals

Al crear clases propias, es nuestra responsabilidad definir cómo se comparan dos instancias de la clase.
Por ejemplo, si tenemos una clase `Persona`, ¿cuándo consideramos que dos personas son iguales?

Método equals

Al crear clases propias, es nuestra responsabilidad definir cómo se comparan dos instancias de la clase.
Por ejemplo, si tenemos una clase `Persona`, ¿cuándo consideramos que dos personas son iguales?
Para esto, debemos definir el método `equals` en nuestra clase.

Método equals

```java
@Override
public boolean equals(Object otro) {
    // Algunos chequeos burocraticos...
    boolean otroEsNull = (otro == null);
    boolean claseDistinta = otro.getClass() != this.getClass();

    if (otroEsNull || claseDistinta) {
        return false;
    }

    //  casting -> cambiar el tipo
    CTD otroCTD = (CTD) otro;

    return gastosPersona1 == otroCTD.gastosPersona1
      && gastosPersona2 == otroCTD.gastosPersona2;
}
```

Método equals

```java
@Override
public boolean equals(Object otro) {
    // Algunos chequeos burocraticos...
    boolean otroEsNull = (otro == null);
    boolean claseDistinta = otro.getClass() != this.getClass();

    if (otroEsNull || claseDistinta) {
        return false;
    }

    //  casting -> cambiar el tipo
    CTD otroCTD = (CTD) otro;

    return gastosPersona1 == otroCTD.gastosPersona1
      && gastosPersona2 == otroCTD.gastosPersona2;
}
```

Pero... ¿cómo se comparan los arreglos?

Método equals

```java
@Override
public boolean equals(Object otro) {
    // Algunos chequeos burocraticos...
    boolean otroEsNull = (otro == null);
    boolean claseDistinta = otro.getClass() != this.getClass();

    if (otroEsNull || claseDistinta) {
        return false;
    }

    //  casting -> cambiar el tipo
    CTD otroCTD = (CTD) otro;

    return arraysIguales(gastosPersona1, otroCTD.gastosPersona1)
      && arraysIguales(gastosPersona2, otroCTD.gastosPersona2);
}
```

Comparando arreglos

```
private boolean arraysIguales(int[] array1, int[] array2) {
    // comparar length
    if (array1.length != array2.length) {
        return false;
    }
    for (int i = 0; i < array1.length; i++) {
        if (array1[i] != array2[i]) return false;
    }
    return true;
}
```

Resumen

- Las clases son una forma de agrupar datos y comportamiento.
- Los objetos son instancias de una clase.
- Los atributos definen el estado interno de un objeto.
- Los métodos definen el comportamiento de un objeto.
- Los constructores inicializan un objeto.
- Cuando dos variables referencian al mismo valor, decimos que hay *aliasing* entre ellas.
- El método `equals` permite comparar dos objetos.

Manos a la obra: taller de POO

Ya pueden comenzar a programar el taller de Programación Orientada a Objetos. Recuerden que la entrega del mismo se hace a través del Campus y su fecha límite de entrega es el 06/04