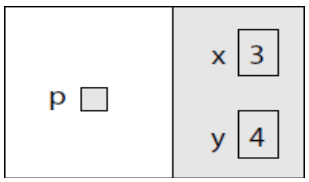
# Programación II
## Objetos en Java

**Definición**

Objeto: Conjunto de **datos** y **métodos** relacionados.

Los objetos se alojan en una parte de la RAM reservada al proceso denominada **memoria dinámica**, y son referenciados por las variables del programa o por otros objetos:

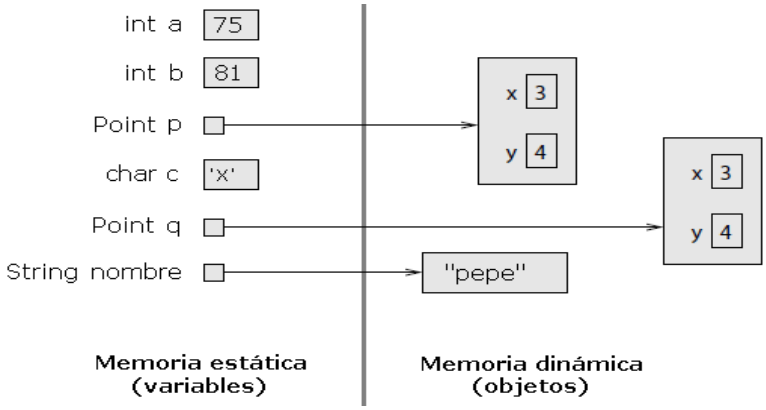



**Memoria dinámica**

Las variables se alojan en **memoria estática**.
1. Para los tipos básicos, las variables contienen el valor del tipo.
2. Para los objetos, las variables contienen referencias a memoria dinámica.

El **ciclo de vida** de las variables es distinto de acuerdo al tipo de memoria.

1. Las variables (en memoria estática) desaparecen de la memoria cuando se cierra el bloque en el cual fueron declaradas.
2. Los objetos se mantienen en memoria dinámica mientras haya alguna variable que los referencie.

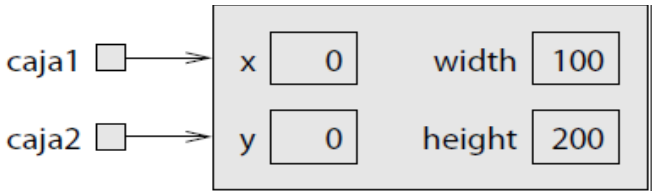


**Aliasing**

Consideremos este código:

```
Rectangle caja1 = new Rectangle(0, 0, 100, 200);
Rectangle caja2 = caja1;
```

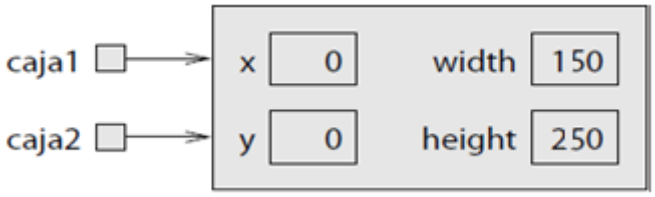
Esto genera que ambas variables referencien al mismo objeto:




Ahora, supongamos que invocamos al método grow() que ensancha y alarga el rectángulo en cada sentido, con las dimensiones especificadas.

```
System.out.println(caja2.width);
caja1.grow(50,50);
System.out.println(caja2.width);
```

Si bien llamamos a un método que modificaba a caja1, se modificó también caja2.



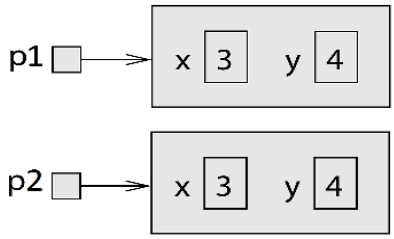

Esto se debe a que las dos variables referencian al mismo objeto. A esto se lo conoce como **aliasing**.

**Comparando objetos**

Consideremos este código:

```
Point p1 = new Point(3, 4);
Point p2 = new Point(3, 4);
```

Tenemos dos referencias que apuntan a objetos **iguales** pero no **idénticos**:

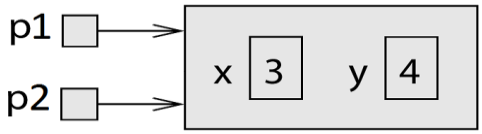



Cuál sería el resultado de la comparación p1==p2?

Consideremos este otro código:

```
Point p1 = new Point(3, 4);
Point p2 = p1;
```

Es decir, tenemos el siguiente diagrama:




Cuál sería el resultado de p1==p2 ahora?
Al igual que con los Strings, la forma de comparar objetos es a través del método equals().

**El método equals()**

Cada clase de Java cuenta con una implementación de este método que compara dos instancias de esa clase.

Cuando definimos nuestras propias clases, es buena práctica definir nuestro propio método **equals** para que se puedan comparar dos objetos de nuestra clase.

Eclipse implementa automáticamente este método con código por defecto!

**Clases**

Las clases son el “molde" a partir de la cual se crean los objetos, dado que especifican qué variables de instancia contienen y a qué métodos responden.

Los datos y métodos asociados a un tipo de objeto se definen en la clase, con lo cual podemos decir que la clase **representa** un tipo de objeto.

1. Las variables de instancia especifican los **valores** que contienen todas las instancias de la clase.
2. Los métodos de la clase especifican el **comportamiento** de las instancias de la clase, en función de las variables de instancia y otros parámetros.

**Conceptos generales**

**Variables de instancia**: Datos incluidos dentro de las instancias de una clase. Se tiene un juego de variables de instancia por cada objeto.

**Variables de clase (static)**: Datos asociados con la clase, comunes a todas las instancias de la clase. Se tiene un único juego de variables de clase para todos los objetos.

**Métodos de instancia**: Funciones que se ejecutan sobre instancias de la clase. Dentro del código del método accedemos a la instancia en cuestión (**parámetro implícito**) por medio de la referencia **this**.

**Métodos de clase**: Funciones asociadas con la clase, y que no se ejecutan sobre instancias de la clase. Dentro del código del método no se puede acceder a variables de instancia, porque no hay parámetro implícito.

**Miembros públicos**: Datos y métodos disponibles por usuarios de la clase, que el objeto “expone" hacia el exterior.

**Miembros privados**: Datos y métodos disponibles sólo por los métodos de la misma clase, habitualmente utilizados para representar el estado interno de la instancia.

**Miembros protegidos**: Datos y métodos disponibles sólo por los métodos de la clase y de sus subclases.

**Clase Fecha**

Ejemplo: supongamos que queremos definir una clase para representar fechas.
Esencialmente, una fecha es un día de un mes de un año.

```
class Fecha
 {
 int dia;
 int mes;
 int anio;
}
```

¿Cómo hacemos ahora para crear objetos de tipo Fecha?

**Constructores**

Cuando se crea un objeto de una clase se llama a un método especial llamado **constructor.**

El constructor se encarga de **inicializar** las variables de instancia.
Veamos un ejemplo de constructor para la clase Fecha.

```
Fecha()
 {
 this.dia=1;
 this.mes=1;
 this.anio=2000;
 }
```

Nótese que el constructor **no devuelve ningún valor**.
La palabra this es una referencia al objeto que está siendo "construido".

Cuando escribimos `new Fecha()` en una expresión, se crea un objeto de tipo Fecha y se llama a este método sobre el objeto para inicializarlo.

**Mas de un constructor**

Podemos definir más de un constructor, siempre con parámetros distintos.

Por ejemplo:

```
Fecha(int d, int m, int a)
 {
 this.dia = d;
 this.mes = m;
 this.anio = a;
 }
```

Este constructor es un **constructor trivial** en el que pasamos un valor para cada variable de instancia.
Es muy común contar con este tipo de constructores.

**Métodos de clase (o estáticos)**

Hagamos ahora un método de la clase Fecha que nos diga si un año es o no bisiesto.

```
static boolean bisiesto(int anio) {  //static denota que es un método de clase
 if (anio % 4 == 0 && anio % 100 != 0)
      return true;
 else if (anio % 400 == 0)
      return true;
 else
      return false;
}
```

En este caso el resultado del método **no depende de ningún objeto**, ya que un año es o no es bisiesto, sin importar de "a quién" se le pregunte.

El anterior es un **método de clase** ya que no depende de alguna instancia en particular.

En un método de clase no existe el parámetro implícito this y el método no se invoca "sobre" un objeto.

Por ejemplo, para listar los años bisiestos hasta el año 2000, podemos usar este método con la siguiente sintaxis:

```
for (int a = 1; a <= 2000; a++)
      if (Fecha.bisiesto(a)) // ← Llamado al método de clase!
            System.out.println("El año " + a + " es bisiesto");
```

**El valor null**

Cuando declaramos una variable del tipo de algún objeto y no le damos ningún valor inicial, éste, por defecto, es null.

Las siguientes declaraciones son equivalentes:

```
Point p1;
Point p2=null;
```

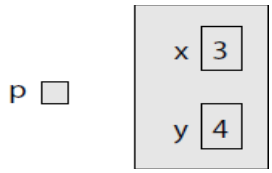
Si tratamos de acceder a un miembro o llamar a un método de un objeto null, se genera una excepción: NullPointerException.

**Garbage Collector**

El valor null nos sirve también para borrar referencias, por ejemplo:

```
Point p = new Point(3, 4);
p = null;
```

En este punto hay un objeto en memoria que no es referenciado por nadie, porque p ahora no apunta a nada:




Los objetos que dejan de ser referenciados por alguna variable, son liberados por el **garbage collector**.

El operador new reserva memoria para un nuevo objeto, y esa memoria está en uso hasta que el garbage collector la recupere.

Esto no ocurre así para las variables de los tipos nativos (int, double, boolean), porque no se almacenan en memoria dinámica.

Cuando el contexto en el cual se cre_o la variable (función, ciclo, etc) se termina, la memoria estática se libera y la variable desaparece.