Java FlyWeight Pattern es uno de los patrones de diseño que más cuesta entender ya que implica construir muchas clases para entender su funcionamiento y sus puntos fuertes. Vamos a intentar realizar un acercamiento a él a través de clases sencillas. Imaginemonos que disponemos de la clase MacBookAir.

```
package com.arquitecturajavaflighweight;

public class MacBookAir {

private String id;
 private int ram;
 private int disco;
 private static int contador;



 public MacBookAir(String id, int ram, int disco) {
 super();
 this.id = id;
 this.ram = ram;
 this.disco = disco;
 contador++;
 System.out.println(contador);
 }
 public String getId() {
 return id;
 }
 public void setId(String id) {
 this.id = id;
```

```
 }
 public int getRam() {
 return ram;
 }
 public void setRam(int ram) {
 this.ram = ram;
 }
 public int getDisco() {
 return disco;
 }
 public void setDisco(int disco) {
 this.disco = disco;
 }

}
```

```
<br data-mce-bogus="1">
```

Se trata de una clase normal y corriente a la que hemos añadido una variable estática “contador” que almacena el número de instancias que tenemos en memoria. Si construimos el siguiente programa principal:

```
package com.arquitecturajavaflighweight;

public class Principal {

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

```
 MacBookAir m1= new MacBookAir("1",4,128);
 MacBookAir m2= new MacBookAir("2",4,128);
 MacBookAir m3= new MacBookAir("3",8,256);
 MacBookAir m4= new MacBookAir("4",8,256);

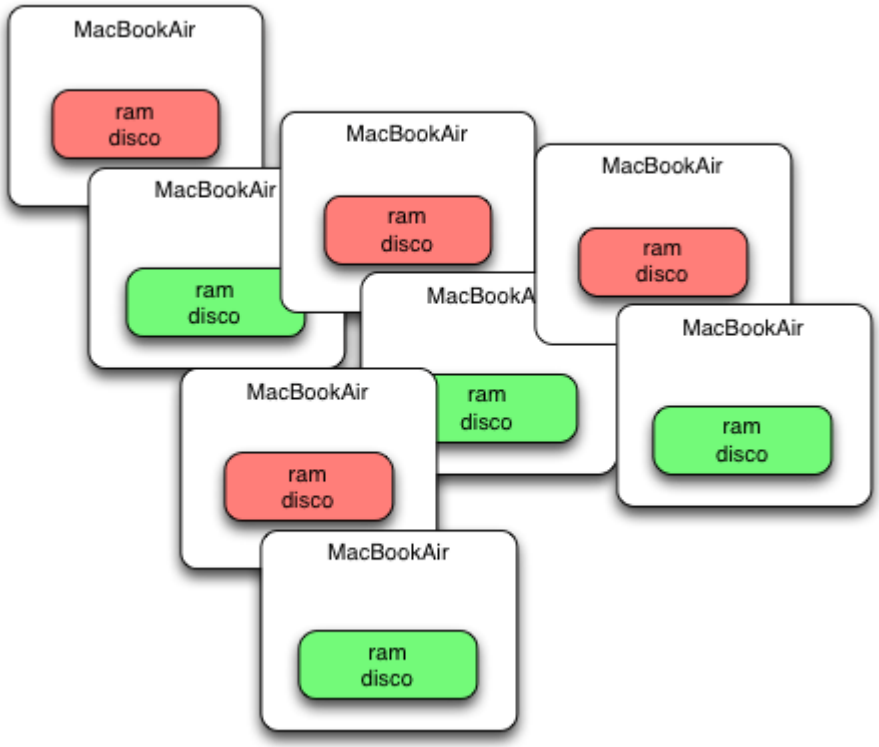
}

}
```

La consola nos mostrará por pantalla algo com esto:

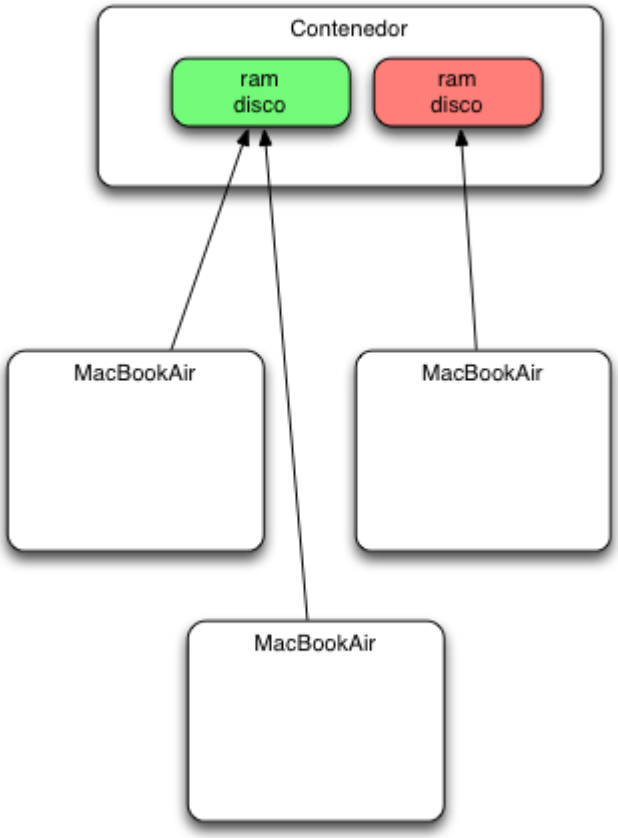


El código es correcto pero ,¿Que pasaría si tuviéramos que construir 100.000 Macbookairs?.



Java FlyWeight Pattern

La realidad es que solo están permitidas 3 o 4 configuraciones de RAM +DISCO en este tipo de equipos. Tenemos por lo tanto un problema de gestión de memoria ya que estamos almacenando los 8 Gb de RAM y 256 de SSD muchas veces. Una solución más elegante sería :



De esta forma solo se almacenaría el valor de cada tipo de MacBookAir una vez y cada ordenador con su numero de serie referencia al que le corresponde esto es un ejemplo Java FlyWeight Pattern. Para construir esta solución necesitaremos varias piezas. En primer lugar vamos a definir un interface y una nueva implementación de nuestro MacBookAir.

```
 

package com.arquitecturajavaflighweight2;

public interface MacBookAir {
```

```
public String getId();

public int getRam();

public int getDisco();

}
```

```
package com.arquitecturajavaflighweight2;

public class MacBookAirImplFlighWeight implements MacBookAir {

private String id;
private MacLigero macLigero;

public MacBookAirImplFlighWeight(String id, MacLigero macLigero) {
super();
this.id = id;

}

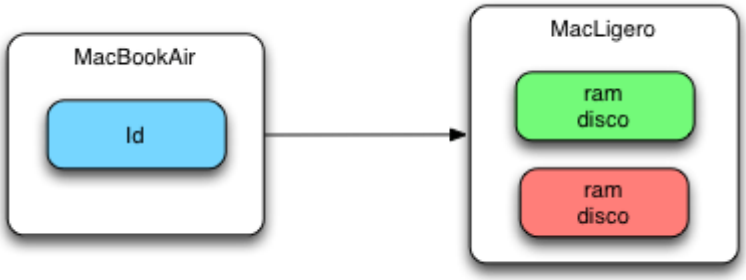
public int getRam() {
return macLigero.getRam();
}

public int getDisco() {
return macLigero.getDisco();
}
```

```
@Override
public String getId() {
return id;
}

}
```

La clase es diferente en muchos aspectos , el primero y más sencillo de entender es que no lleva métodos set ya que el MacBookAir no se le puede cambiar nada una vez comprado . Es una mejora a nivel de implementación , el segundo es que cada MacBookAir solo lleva un ID el resto de los campos los delega en otra clase que tendremos que construir. Estos campos son los comunes a todos los MacBookAir



Vamos a ver su código:

```
package com.arquitecturajavaflighweight2;

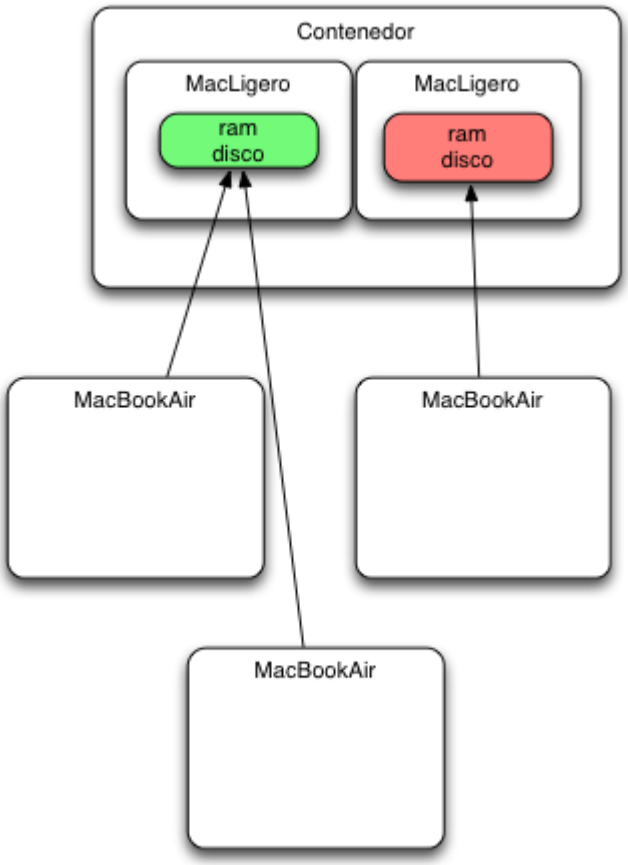
public class MacLigero {

private int ram;
```

```
private int disco;
private static int contador;

public MacLigero(int ram, int disco) {
super();
this.ram = ram;
this.disco = disco;
contador++;
System.out.println("contador ligero: "+ contador);
}
public int getRam() {
return ram;
}
public void setRam(int ram) {
this.ram = ram;
}
public int getDisco() {
return disco;
}
public void setDisco(int disco) {
this.disco = disco;
}
}
```

Como podemos ver la clase es realmente sencilla. La parte compleja de entender viene compuesta por dos elementos más . El primero es el contenedor de MacLigeros que almacenará los diferentes tipos.



Su código es:

```
package com.arquitecturajavaflighweight2;

import java.util.HashMap;
import java.util.Map;

public class ContenedorMacLigero {

private static Map<String,MacLigero> macLigeros= new HashMap<String,
```

```
MacLigero>();
public static MacLigero getMacLigero(int ram,int disco) {

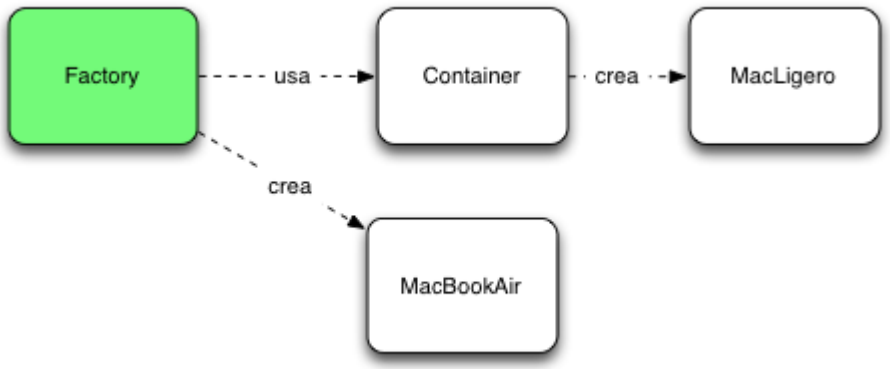
String clave=ram+":"+disco;

System.out.println(clave);

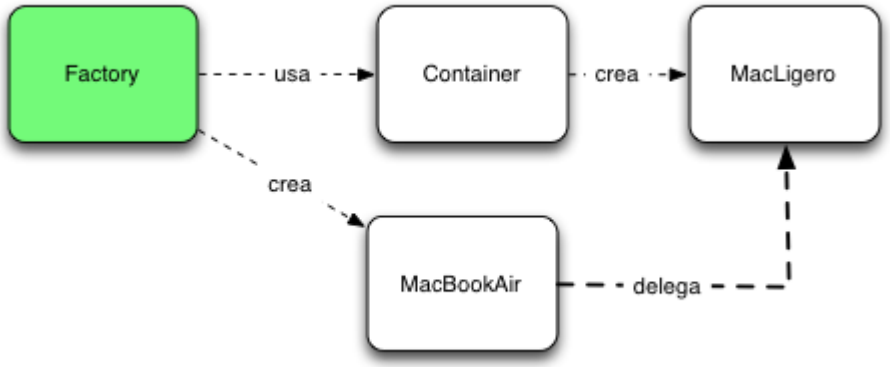
if (!macLigeros.containsKey(clave)) {

macLigeros.put(clave,new MacLigero(ram,disco));
}
return macLigeros.get(clave);
}
}
```

Se trata de un HashMap que almacena los diferentes tipos de MacLigeros no podrá haber más de 4-8 combinaciones. El último paso es crear la factoría que construye los Macs.



La factoría nos ayudará a crear de forma correcta los MacBookAirs apoyandose en los MacLigeros.




```
package com.arquitecturajavaflighweight2;

public class FactoriaMacs {
public static MacBookAir crearMacBookAir(String id, int ram ,int
disco) {

MacLigero macLigero=ContenedorMacLigero.getMacLigero(ram, disco);
MacBookAir macBookAir= new MacBookAirImplFlighWeight(id,macLigero);
return macBookAir;


}
}
```

El programa principal cambiará para adaptarse a la factoría:

```
package com.arquitecturajavaflighweight2;
```

```
public class Principal {

public static void main(String[] args) {



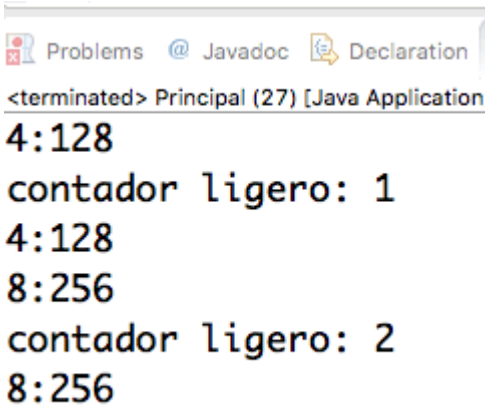
MacBookAir m1= FactoriaMacs.crearMacBookAir("1",4,128);
MacBookAir m2= FactoriaMacs.crearMacBookAir("2",4,128);
MacBookAir m3= FactoriaMacs.crearMacBookAir("3",8,256);
MacBookAir m4= FactoriaMacs.crearMacBookAir("4",8,256);

}

}

 
```

Sin embargo los cambios importantes vendrán cuando ejecutemos el programa:



Solo se han creado dos MacBooksLigeros aunque tengamos cuatro MacBooksAir hemos

aplicado Java FlyWeight Pattern a nuestro código.

Otros artículos relacionados: Java Proxy Pattern , Java Singleton Class Loaders