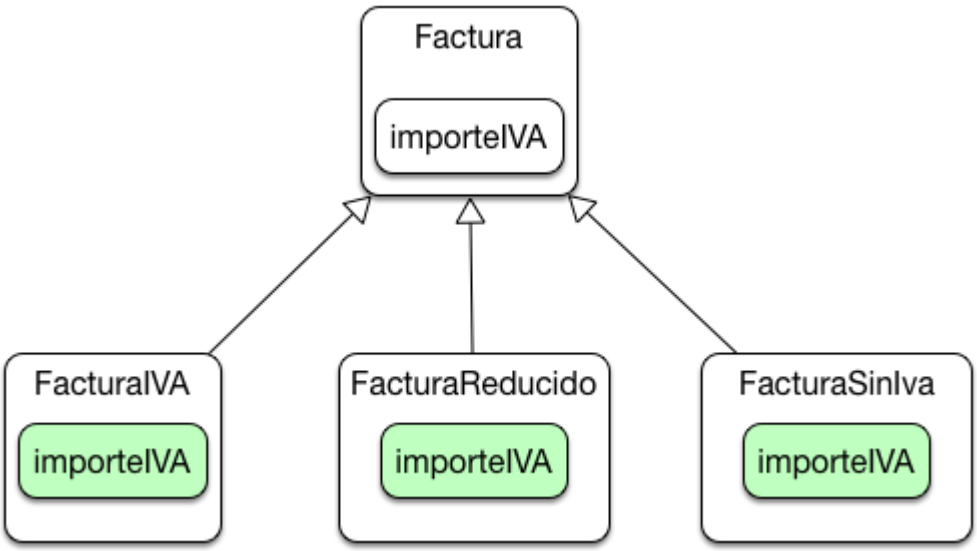
El concepto de Java Package encapsulation no es conocido por todos. Todos los desarrolladores conocemos que existe public , private , protected y package visibility como operadores de ámbito. Sin embargo no siempre sabemos como darles uso. Vamos a construir un ejemplo que nos ayude a diseñar un API utilizando el concepto de encapsulación y packages de una forma más sólida. Para ello vamos a partir de una jerarquía de clases de Facturas que tienen un importe y calculan el importe con IVA.



Veamos el código:

```
package com.arquitecturajava.ejemplo1;

public abstract class Factura {

        private int id;
        private String concepto;
        private double importe;
```

```
        public static Factura getFactura(int id, String concepto,
double importe,TipoFactura tf) {
                if (tf==TipoFactura.IVA) {
                        return new FacturaIva(id,concepto,importe);
                }else if (tf==TipoFactura.SINIVA) {
                        return new FacturaSinIVA(id,concepto,importe);
                }else {
                        return new
FacturaReducida(id,concepto,importe);
                }
        }
        public Factura(int id, String concepto, double importe) {
                super();
                this.id = id;
                this.concepto = concepto;
                this.importe = importe;
        }
        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getConcepto() {
                return concepto;
        }
        public void setConcepto(String concepto) {
                this.concepto = concepto;
        }
        public double getImporte() {
                return importe;
```

```
	}
	public void setImporte(double importe) {
		this.importe = importe;
	}
	public Factura() {
		// TODO Auto-generated constructor stub
	}
	public abstract double importeConIva() ;
}
```

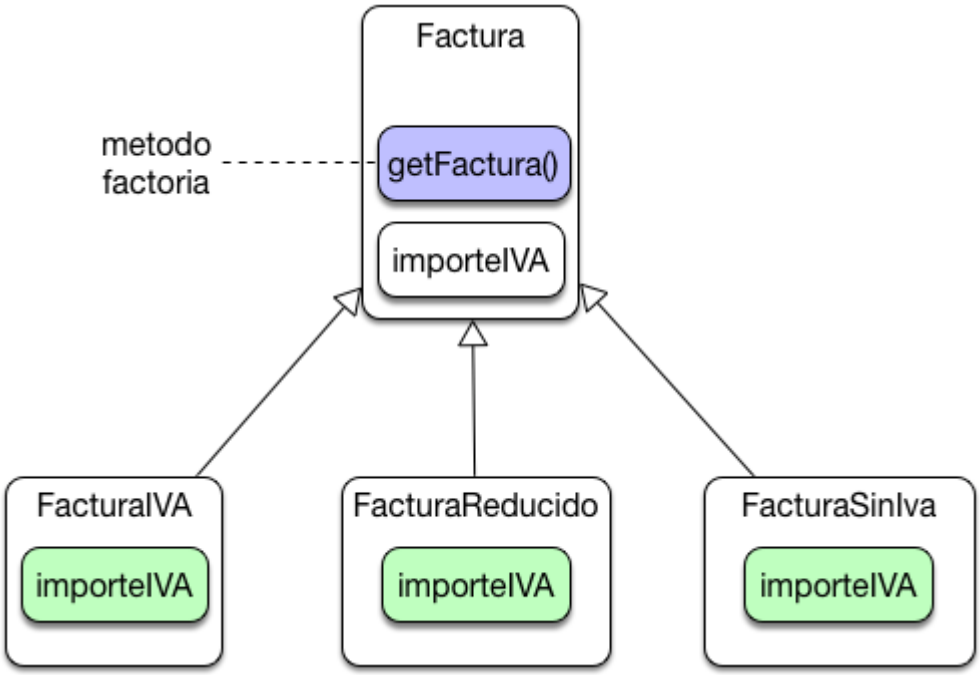
```
package com.arquitecturajava.ejemplo1;

public class FacturaIva  extends Factura{



	public FacturaIva(int id, String concepto, double importe) {
		super(id, concepto, importe);
		// TODO Auto-generated constructor stub
	}

	@Override
	public double importeConIva() {
		// TODO Auto-generated method stub
		return getImporte()*1.21;
	}

}
```

```
package com.arquitecturajava.ejemplo1;

public class FacturaReducida extends Factura{


	public FacturaReducida(int id, String concepto, double
importe) {
		super(id, concepto, importe);
	}

	@Override
	public double importeConIva() {
		// TODO Auto-generated method stub
		return getImporte()*1.07;
	}

}
```

```
package com.arquitecturajava.ejemplo1;

public class FacturaSinIVA extends Factura {



	public FacturaSinIVA(int id, String concepto, double importe)
{
		super(id, concepto, importe);
		// TODO Auto-generated constructor stub
```

```
	}

	@Override
	public double importeConIva() {
		// TODO Auto-generated method stub
		return getImporte();
	}

}
```

Acabamos de diseñar la jerarquía de clases completa utilizando el método getFactura() como factory method para generar la jerarquía completa de facturas.



Para ello nos hemos apoyado en una enumeración:

```
package com.arquitecturajava.ejemplo1;
```

```
public enum TipoFactura {
        IVA, REDUCIDA, SINIVA

}
```

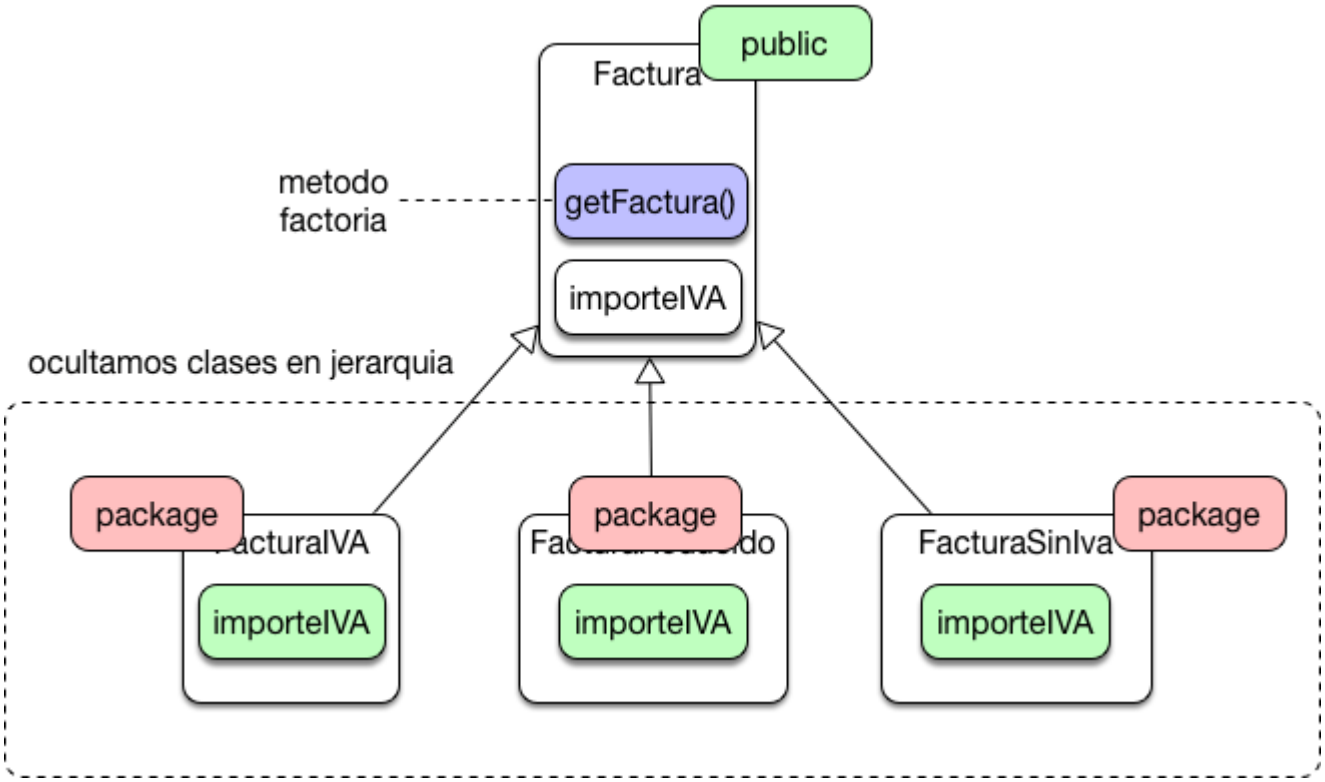
Java Package Encapsulation

El código se puede utilizar sin ningún problema con un Main y construir las Facturas para utilizarlas.

```
package com.arquitecturajava.main;

import com.arquitecturajava.ejemplo2.Factura;
import com.arquitecturajava.ejemplo2.TipoFactura;

public class Principal {


        public static void main(String[] args) {
                Factura f=Factura.getFactura(1, "ordenador", 300,
TipoFactura.IVA);
                System.out.println(f.getImporte());
                FacturaIva f2= new FacturaIva(1,"ordenador",300);
                System.out.println(f2.getImporte());
        }

}
```

¿Existe algún problema?.La realizad es que no. Ahora bien¿ es necesario que el que programa el Main sea consciente de la jerarquía de clases que existe?. La realidad es que no , con conocer la enumeración sería suficiente. Sin embargo la conoce muy bien ya que nada mas que escribimos la palabra Factura nos aparecen todas las opciones.

¿Podemos obligar al desarrollador a tener que construir siempre la Factura usando el método de Factoría ? .En estos momentos NO. Ahora bien si cambiamos la visibilidad de todas las clases hijas de Factura de public a package . Si podremos conseguir ese efecto . De tal forma que los desarrolladores únicamente conocerán el concepto de Factura y no la jerarquía completa viendose obligados a usar el método de Factoria.



Para ello será suficiente con mantener la clase Factura como acceso público y el resto de clases de la jerarquía eliminar el acceso publico y dejar por package.

```
public class Factura {

}
```

```
class FacturaIva extends Factura {

}
```

Realizadas estas operaciones el desarrollador no tendrá ninguna opción de crear Facturas de un tipo concreto, solamente podrá usar el método getFactura de la clase Factoria para crear los diferentes objetos. Esto nos ayudará a simplificar el como funcionan las cosas.

```
Factura f=Factura.getFactura(1, "ordenador", 300, TipoFactura.IVA);
System.out.println(f.getImporte());
```

Acabamos de utilizar el concepto de Java Package Encapsulation para mejorar el diseño de nuestra solución:

Otros artículos relacionados:

1. Java 8 Factory Pattern y su implementación
2. Java Herencia vs Interfaces
3. Java Herencia y sus limitaciones
4. Java Package