**EJERCICIO 1**

A - El esqueleto del template method suele definirse en una superclase abstracta.

B - Una vez que está definido lo ideal es que nadie tenga acceso a el/los método/s como para sobreescribrilo/s, es decir, que no sea posible redefinirlo/s.

C - Todo depende del diseño, si los metodos tienen la característica "final" no pueden ser redefenidos, si no tienen esta observación pueden modificarse.

D - Un hook method es un método que no tiene ningun comportamiento definido (no hace nada), pero queda disponible para que las subclases lo implementen a su conveniencia.

**EJERCICIO 4**

**public abstract class** LlamadaTelefonica { // Clase abstracta

**private int** tiempo ;

**private int** horaDelDia ;

**public** LlamadaTelefonica( **int** tiempo , **int** horaDelDia ){

**this** . tiempo = tiempo ;

**this** . horaDelDia = horaDelDia ;

} // Operación concreta

**public int** getTiempo(){

**return this** . tiempo ;

} // Operación primitiva

**public int** getHoraDelDia(){

**return this** . horaDelDia ;

} // Operación primitiva

**public abstract boolean** esHoraPico(); // Hook Method

**public float** costoFinal(){

**if** ( **this** .esHoraPico()){

**return this** .costoNeto()\*1.2f\* **this** .getTiempo();

} **else** {

**return this** .costoNeto()\* **this** .getTiempo();

}

} // Template method

**public float** costoNeto(){

**return this** .getTiempo()\*1;

} // Operación concreta

}

**public** classLlamadaDescuento **extends** LlamadaTelefonica{ // Clase concreta

**public** LlamadaDescuento( **int** tiempo, **int** horaDelDia) {

**super** (tiempo, horaDelDia);

} // Operación concreta

@Override

**public boolean** esHoraPico() {

**return false** ;

} // Operación concreta

@Override

**public float** costoNeto(){

**return this** .getTiempo()\*0.95f;

} // Operación concreta

}

**EJERCICIO 5.1**

**public abstract class** CuentaBancaria {

**public void** extraer(**int** monto) {

**if** (**this** .puedeExtraer(monto)){

**this** .setSaldo( **this** .getSaldo()- monto);

**this** .agregarMovimientos("Extraccion");

}

}

**public abstract boolean** puedeExtraer(**int** monto);

}

**public class** CuentaCorriente **extends** CuentaBancaria {

@Override

**public boolean** puedeExtraer(**int** monto) {

**return this**.getSaldo() + **this**.getDescubierto() >= monto;

}

}

**public class** CajaDeAhorro **extends** CuentaBancaria {

@Override

**public boolean** puedeExtraer(**int** monto) {

**return this**.getSaldo() >= monto && **this**.getLimite() >= monto;

}

}

**EJERCICIO 5.2**

**public abstract class** CuentaBancaria // Clase Abstracta

**public** CuentaBancaria(String titular ) // Operación concreta

**public** String getTitular() // Operación primitiva

**public int** getSaldo() // Operación primitiva

**protected void** setSaldo( **int** monto ) // Operación primitiva

**public void** agregarMovimientos(String movimiento ) // Operación primitiva

**public void** extraer(**int** monto) // Template method

**public abstract boolean** puedeExtraer(**int** monto) // Método abstractoM

**public class** CuentaCorriente **extends** CuentaBancaria // Clase concreta

**public** CuentaCorriente(String titular , **int** descubierto ) // Operación concreta

**public int** getDescubierto() // Operación primitiva

**public void** extraer( **int** monto ) // Operación concreta

**public boolean** puedeExtraer(**int** monto) // Hook method

**public class** CajaDeAhorro **extends** CuentaBancaria // Clase concreta

**public** CajaDeAhorro(String titular , **int** limite ) // Operación concreta

**public int** getLimite() // Operación primitiva

**public void** extraer( **int** monto ) // Operación concreta

**public boolean** puedeExtraer(**int** monto) // Hook method

**EJERCICIO 6**

1 – Existen dos tipos de adaptadores, los de **clase** que utilizan múltiples herencias para adaptar unas interfaces con otras. Y también están los adaptadores de **objeto**, que se implementan en base a la composición de un objeto concreto.

2 – El de clase adapta un *Adaptado* con un *Objetivo* comprometiéndose con una clase *Adaptadora* concreta. Como consecuencia, la clase adaptadora no va a funcionar cuando se quiere adaptar una clase y todas sus subclases. Deja que el *Adaptador* sobrescriba algunas cosas del comportamiento del *Adaptado*, ya que el primero resulta en una subclase del segundo. Introduce un único objeto y no se necesita ningún puntero indirecto adicional para llegar al *Adaptado*.

Por otro lado, el de objeto deja que un único *Adaptador* trabaje con múltiples *Adaptados* (él mismo y todas sus subclases). A su vez es posible agregar funcionalidad a todos los *Adaptados* a la vez. El problema es que se vuelve más complejo sobrescribir comportamiento del *Adaptado*, ya que esto requiere subclasificar al *Adaptado* y hacer que el *Adaptador* se refiera a la subclase en lugar del Adaptado mismo.

3 – La alternativa de adaptador de objeto se puede usar sin problemas en Java (es la que vimos en el ejemplo de la clase), por otro lado, la de clase presentaría algunos problemas: No se puede tener herencia múltiple en Java, solo se puede implementar múltiples interfaces (esto significa que las cosas a adaptar sean interfaces).

**EJERCICIO 9**

1 – El patrón Adapter en este caso se dá entre Vector (Adaptader) y Enumeration (Adaptee), ya que Vector define el comportamiento de los métodos de la interfaz Enumeration en base a otros métodos pertenecientes a la clase. El tipo de implementación es de clase, ya que implementa una interfaz (no directamente, sino a través del método elements()).

2 – En este caso el método iterator() de la clase ArrayList retorna una clase Itr que va a ser un Adapter de la interfaz Iterator, ya que dicha clase la implementa y a su vez define comportamiento adaptado a ArrayList. El ejemplo se encuentra en la clase “Prueba” dentro del paquete “tp7.colecciones”.