ALUMNO: Cuesta Alario David **DNI:** 20857516-N

Entorno Eclipse y perspectiva Java

Espacio de trabajo:

Definir el espacio de trabajo en el que almacenar los proyectos que se generen

$$File \rightarrow Switch\ Workspace.$$

Transportar los proyectos, paquetes y clases mediante las opciones de menú

$$File \rightarrow Export \rightarrow General \rightarrow File\ System$$

$$File \rightarrow Import \rightarrow General \rightarrow File$$

Se puede acceder a la perspectiva Java desde el menú

$$Window \rightarrow Open\ Perspective \rightarrow Java$$

Para evitar problemas con la herramienta Web-CAT conviene verificar que el editor de Eclipse utilice la codificación del texto UTF-8.

$$Window \rightarrow Preferences \rightarrow General \rightarrow Workspace \rightarrow Text\ File\ Encoding \rightarrow Others \rightarrow UTF\text{-}8$$

Mostrar por defecto los números de línea

$$Window \rightarrow References \rightarrow General \rightarrow Editors \rightarrow Text\ Editors \rightarrow Show\ line\ numbers.$$

Atajos:

Cualquier entorno de desarrollo proporciona atajos para ejecutar las acciones más importantes y habituales

$\boldsymbol{Ctrol - Shift - L}$ ⟶ Muestra la lista de todos los atajos

$\boldsymbol{Ctrol - Space}$ ⟶ Autocompletado (quizá se deba marcar la opción)

$$Preferences \rightarrow Java \rightarrow Editor \rightarrow Content\ Assist \rightarrow Advanced \rightarrow Java\ Proposals$$

Criterio Nombres:

Java es un lenguaje sensible a las letras mayúsculas y minúsculas, por lo que es muy importante seguir un criterio a la hora de nombrar los diferentes elementos.

- **Los paquetes:** siempre van escritos en minúsculas.
- **Los atributos, los métodos (excepto el método constructor), las variables y los parámetros:** Utilizaran el estilo "lowerCamelCase". En el caso de los parámetros comenzaran siempre por la letra minúscula 'p'
- **Las clases y las interfaces**: Utilizaran el estilo "UpperCamelCase" En el caso de los parámetros comenzaran siempre por la letra mayúscula 'I'
- **Las constantes:** siempre van escritos en mayúsculas

Elemento	Estilo	Ejemplos
Paquete	minúsculas (separado por puntos)	org.pmoo.packlaboratorio1
Clase	UpperCamelCase (sustantivos)	Persona, ListaVentas
Interfaz	UpperCamelCase (adjetivos)	IOrdenable, ISerializable
Método constructor	UpperCamelCase (nombre clase)	Persona, ListaVentas
Tipo enumerado	UpperCamelCase (sustantivos)	Color, Operando
Método	lowerCamelCase (verbos)	ordenar, getEdad
Atributo	lowerCamelCase (sustantivos)	pNombre, pListaAlumnos
Parámetro	lowerCamelCase (sustantivos)	longitud, coordX
Variable	lowerCamelCase (sustantivos)	max, notaMedia
Constante	MAYÚSCULAS (separado por _)	PI, MAX_ELEMS

Comentarios:

De una o más líneas ⟶ /*.....*/
De una sola linea ⟶ //.....
Dejar un recordatorio de una tarea pendiente ⟶ //TODO

Declaración de variables y objetos

Número entero ⟶ int
Número entero largo ⟶ long
Número real ⟶ float
Número real de doble precisión ⟶ doble
Caracter ⟶ char ⟶ 'caracter'
Cadena de caracteres ⟶ String ⟶ "cadena de caracteres"
Booleano ⟶ true | false

Tipo	Expresión en Java	Ejemplos

Tipo de acceso	(acceso sólo desde el paquete)
	public (acceso desde cualquier clase)
	private (acceso sólo desde su clase)
	protected (acceso desde clases que heredan)
		`long fact;`
		`public long fact;`
		`private long fact;`
		`protected void setUp();`

Entero	**byte** | **short** | **int** | **long** <nombre variable>	`int edad;` `long factorial;`
Real	**float** | **double** <nombre variable>	`float notaMedia;`
Carácter	**char** <nombre variable>	`char inicial;`
String	**String** <nombre variable>	`String tituloLibro;`
Booleano	**boolean** <nombre variable>	`boolean encontrado;`
Declaración múltiple	<tipo acc> <tipo el> <nombre1>, <nombre2>,....;	`private int i,j,k;`

Operaciones básicas

Operación	Operadores	Ejemplos
Asignación	=	`x=0;`
Operaciones aritméticas	+ - * / %	`i = i + 1;  resto = num % 2;`
Comparación	< <= > >= == !=	`if(num<0)` `while(numElems>=0)`

Operaciones lógicas	&& (and)
	|| (or)
	! (not)
		`if (x==0 && y==0)`
		`while(x<0 || x>MAX)`
		`while (!encontrado)`

Concatenación de Strings (acepta tipos básicos)	+	`String saludo = "Hola " + "mundo";` `String str = "El valor de i es " + i;`

Sentencias básicas

Imprimir por pantalla:

System. out. println("Cadena de texto" + Instancia_1. toString() + Variable);

Programa Principal:

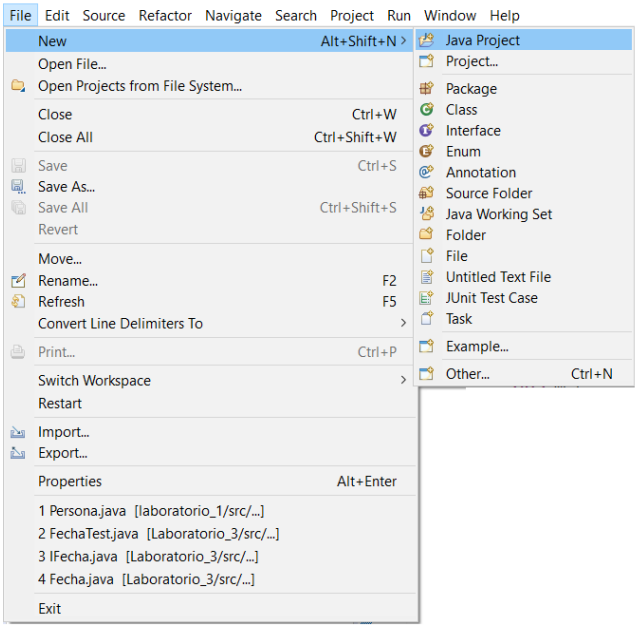
public static void main(String[] args) {

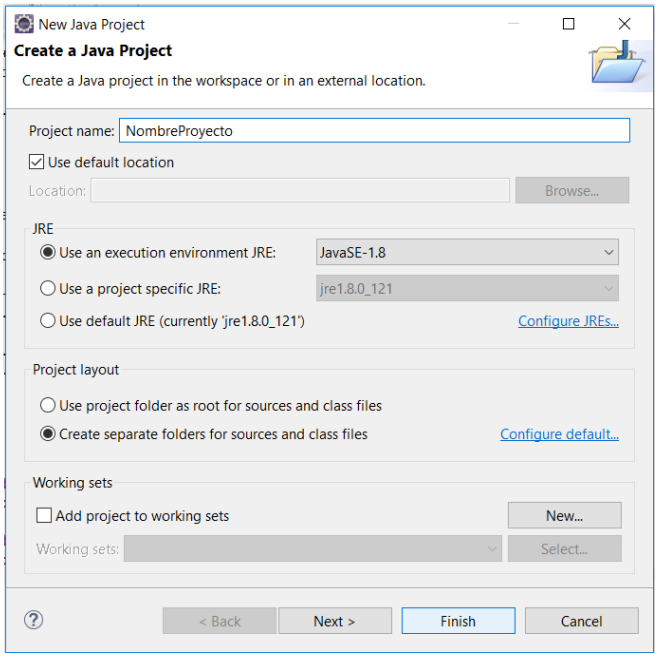
NombreClase InstanciaClase = new NombreClase (Parametros_Inicializacion);

Proyectos:

- Crear un proyecto

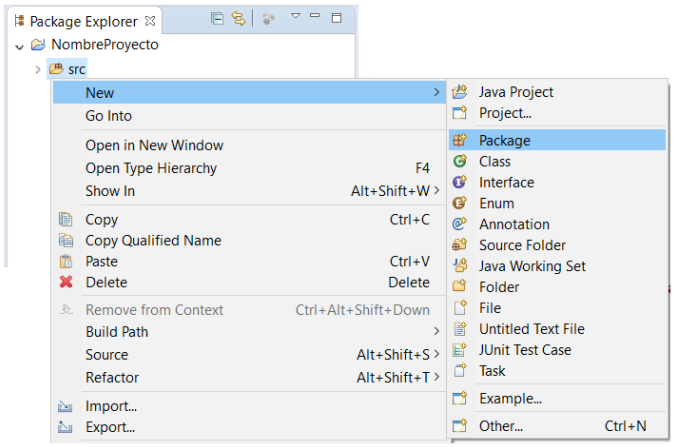
File → New → Project → Java → Java Project

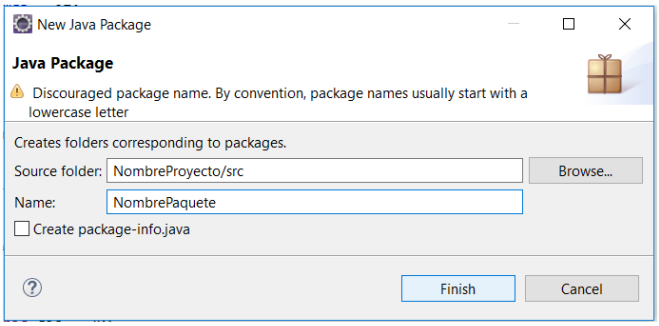




- Crear un paquete en el que agrupar los ficheros fuente

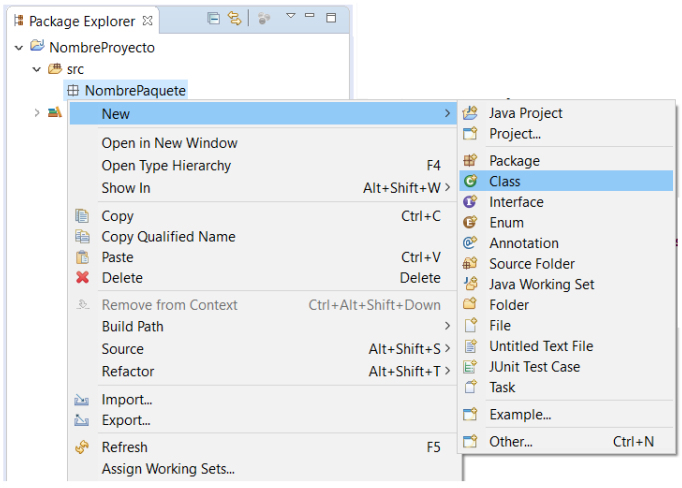
Package Explorer → boton derecho sobre la carpeta source "src" *→ New → Package.*

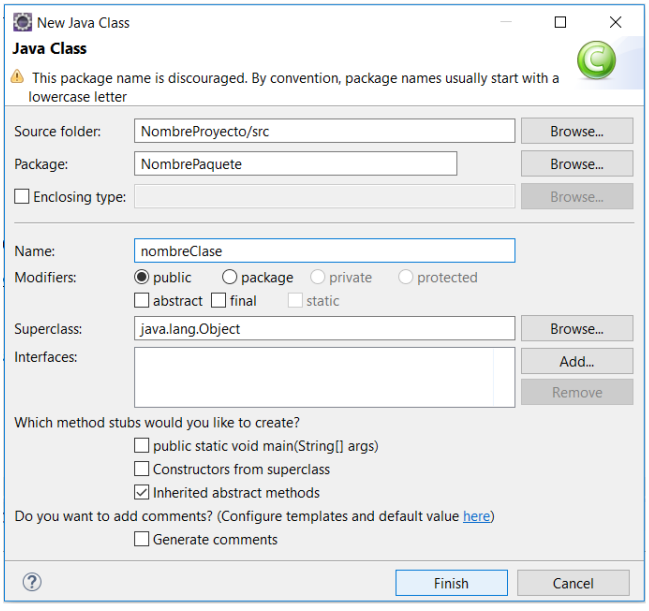




- Crear una clase

Package Explorer *→ boton derecho sobre el paquete → New → Class.*



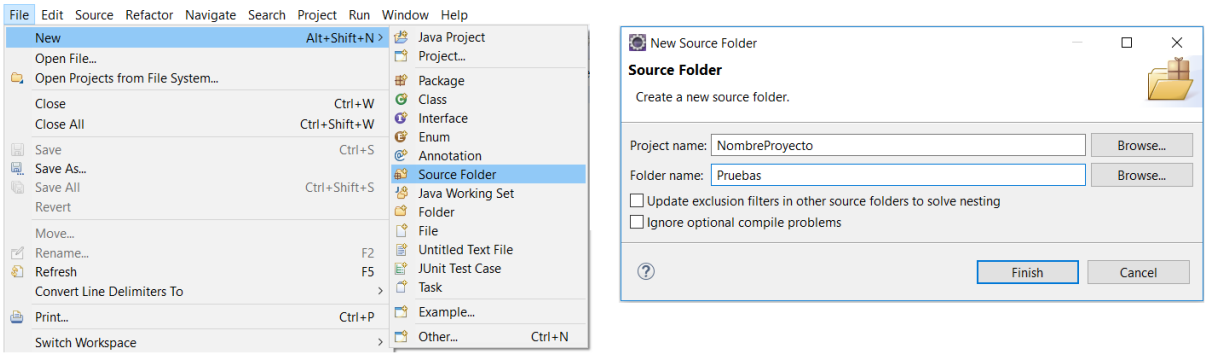


JUnits:

Consiste en un conjunto de clases que facilitan la evaluación del funcionamiento de los métodos implementados mediante determinadas entradas a los métodos comparan los resultados devueltos que se quieren analizar con los valores esperados.

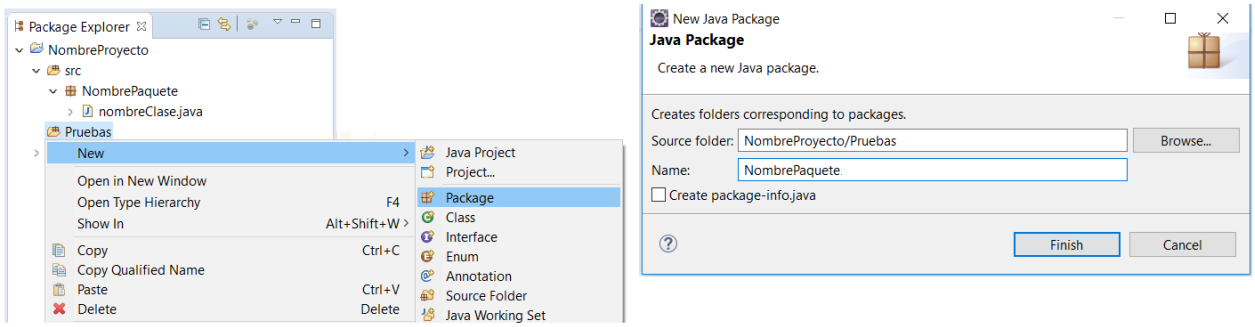
- Deberemos crear una nueva carpeta

$$File \rightarrow New \rightarrow Source\ Folder$$



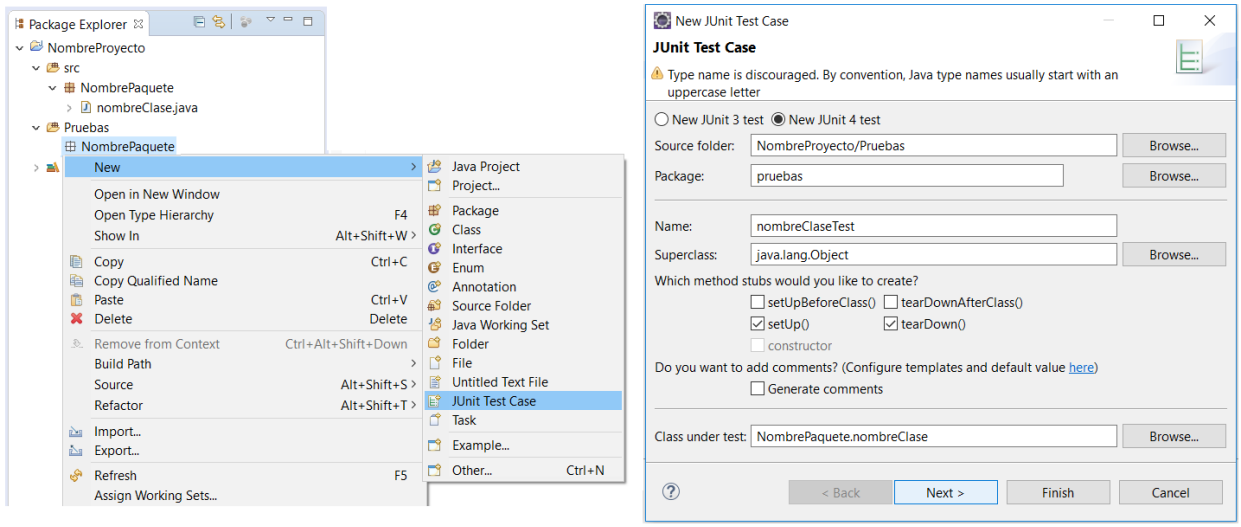
- Debe formar parte del mismo paquete en el que se encuentra la clase que se pretende evaluar
  - llevará el mismo nombre que el paquete al que simula

Package Explorer → botón derecho sobre la carpeta "*Pruebas*" → New → Package

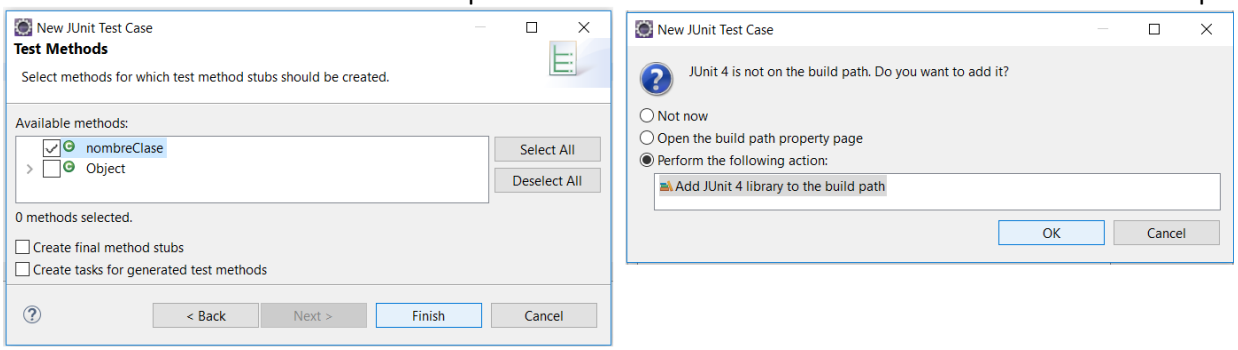


- Creamos el caso de prueba

Package Explorer → *boton derecho sobre el paquete* → New → JUnit Test Case



- Activamos los métodos de la clase a implementar e incluimos la ruta de acceso a las librerías de compilación



Orientación a objetos

Uno de los principios básicos de la programación orientada a objetos es el principio de mínimo conocimiento, implica que una clase tan sólo debe conocer aquellas otras con las que tenga que relacionarse de forma directa con la finalidad de que cuando alguno de los elementos del sistema cambie menor será el impacto en el resto del programa.

Clase ⟶ abstracción de objetos concretos

Atributos ⟶ *implementan su forma o estado*

Estaticos ⟶ Parametro consante que pertenece a la clase y es comun a todos los objetos del mismo tipo

- Para acceder a dicho valor no se necesita crear un objeto, sino que se hace a través de la clase

```
public class Producto
{
  //atributos
  private int idProducto;
  private double precioUnitario;
  private static double iva = 0.18;
```

```
//constructora
public Producto (int pId, double pPrUn)
{
   this.idProducto = pId;
   this.precioUnitario = pPrUn;
}
```

```
//otros métodos
public double obtenerPVP()
 {
   return this.precioUnitario*(1 + Producto.iva);
 }
}
```

Métodos ⟶ *implementan su comportamiento*

Constructora ⟶ Método especial que construye e inicializa los nuevos objetos

- Puede haber varias siempre y cuando empleen diferentes parámetros
- Si no se implementa ninguna, Java inicializa los atributos a valores por defecto

Estaticos ⟶ Utilizan exclusivamnete atributos estaticos

- Su ejecución no está ligada a un objeto concreto
- Devolverán el mismo resultado sin importar desde dónde se les ha invocado

Objeto ⟶ elemento real con identidad propia

Variables de instancia ⟶ instancias de los atributos pertenecientes a una clase

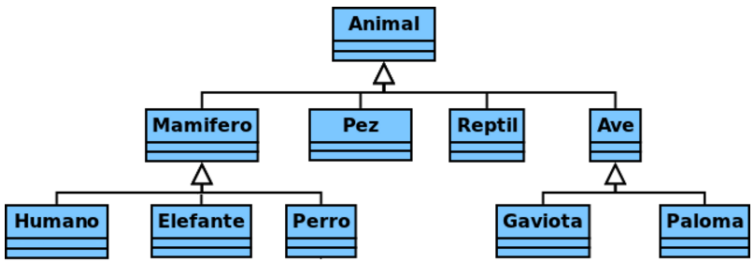
- Son punteros la instrucción new genera un objeto invocando los métodos constructores

Herencia

Se trata de un tipo de relación en la que las subclases heredan atributos y métodos de una superclase.

Todas las clases son subclases de la clase Objet

- algunos métodos de interés:



nombreObjeto. equals( ) ⟶ evalúa si dos objetos son exactamente la misma instancia en memoria

nombreObjeto. toString( ) ⟶ *convierte la dirección de memoria del objeto a String*

Los métodos heredados se pueden sobrescribir para que realicen otra función

- Cuando se ejecuta un método heredado, el compilador siempre llamará a la implementación más reciente.

Para implementar la herencia se utiliza la palabra reservada extends No usarla equivale a hacer *extends Object*

La constructora de una subclase debe llamar a la constructora de la superclase mediante el comando

equals(*parametros_SuperClase*)

De lo contrario se estaría llamando a la constructora de Object

```
public  class ProductoPerecedero extends Producto
{
 // atributos
 private static double iva = 0.11;
 private int diaCad;
 private int mesCad;
 private int añoCad;
 // constructora
 public ProductoPerecedero(double pPrec, String pNombre,
        Proveedor pProv, int pDiaC, int pMesC, int pAñoC)
 {
     super (pPrec, pNombre, pProv);
    this.diaCad =pDiaC;
    this.mesCad =pMesC;
    this.añoCad =pAñoC;
 }
```

Visibilidad

Los atributos siempre se definen privados

Esto significa que los atributos heredados no son directamente accesibles desde las subclases

Lo mismo ocurre con los métodos privados de una superclase. Aunque sus subclases los hereden, no los pueden utilizar porque no son visibles para ellas

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	**private**	*no*	*no*	*no*	*no*
#	**protected**	*sí*	*sí*	*sí*	*no*
+	**public**	*sí*	*sí*	*sí*	*sí*
~	*package*	*sí*	*sí*	*no*	*no*

JUnits:

```
package nombrePaquete;

import static org.junit.Assert.*;

public class nombreClaseTest {

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }
```

El comando **Package** permite incluir la clase que se pretende analizar.

El comando **import static** incluye de la librería externa las clases necesarias para realizar las llamadas a las aserciones

El método publico **setUp()** inicializa el contexto de las pruebas asignando valores a los parámetros de las constructoras. Este método se ejecuta justo antes de proceder con cada uno de los métodos de prueba

El método publico **tearDown()** se encarga de eliminar el contexto de las pruebas y realizar la limpieza necesaria antes de comenzar con otro test. Este método se ejecuta justo después de terminar la ejecución de cada uno de los métodos de prueba

Los métodos **test()** son los métodos que permiten comprobar la corrección del método que se está poniendo a prueba.

Método	Descripción
assertTrue()	la prueba solo se supera si la condición es verdadera.
assertFalse()	la prueba solo se supera si la condición es falsa.
assertEquals()	la prueba solo se supera si los dos objetos son iguales. Si son valores *double*, assertEquals pide un tercer parámetro (delta) que representa el margen de error aceptable para asumir que ambos números son iguales.
assertArrayEquals()	la prueba solo se supera si los dos arrays son iguales.
assertNotNull()	la prueba solo se supera si el objeto existe (no es null)
assertNull()	la prueba solo se supera si el objeto no existe (es null).
assertSame()	la prueba solo se supera si ambos objetos son el mismo.
assertNotSame()	la prueba solo se supera si los dos objetos no son el mismo.
assertThat()	la prueba solo se supera si el objeto satisface la condición.
fail()	la prueba falla.

Para ejecutar la simulacion

$$Run \rightarrow Run\ As \rightarrow JUnit\ Test$$

Interfaz Java:

Consiste en una colección de constantes y métodos abstractos no implementados que servirán como especificación de las clases que lo implementen

La relación entre clases es similar a la herencia, con la diferencia de que una clase puede implementar múltiples interfaces al mismo tiempo.

Esta estructura ha sido previamente implementada en Java de modo que se proporcionan un conjunto de métodos básicos para su facilitar el manejo:

public interface nombreInterfaz ⟶ permite definir un objeto tipo interfaz

public nombreClase implements nombreInterfaz ⟶ permite reconocer a una clase como implementadora de una interfaz

Tipos de clases:

Maquina Abstracta de estado MAE

Se trata de una clase cuya constructora asegura que solo va a existir una instancia de dicho objeto

- La instancia al objeto se realiza dentro de la constructora
- La constructora
  - es privada por lo que no se podrá crear objetos desde fuera de dicha clase
  - estará vacía si no hay más atributos además del static

```
public class MAE
{
  //generar instancia de manera estática
  private static MAE miMAE = new MAE();
  // la constructora es privada
  private MAE() { }
  // método de acceso a la instancia
  public static MAE getMAE()
                { return miElvis; }
}
```

```
public class MAE
{
  //atributos
  private ListaCanciones repertorio; ...
  private static MAE miMAE = new MAE();
  //la constructora ya no estaría vacía
  private MAE() { this.repertorio= new ListaCanciones(); }
  // método de acceso a la instancia
  public static MAE getMAE()
                { return miElvis; }
}
```

Tipo abstracto de datos TAD

Se trata de una clase que es capaz de realizar un conjunto de operaciones sobre una estructura de datos, de tal forma que pueda obviarse su implementación.

- Mediante interfaces se ofrecen un conjunto de métodos capaces de operar sobre los datos
- La precondición y poscondición establecen los límites de dichas operaciones
- Permite un cambio en la implementación de dichas estructuras sin afectar a los programas que las usan
- Permite una verificación independiente de los programas que utilizan o implementan dichas estructuras

Listas:

Se trata de una agrupación de elementos de una misma clase en posiciones de memoria consecutivas, de tal modo que se puede acceder directamente a cada elemento a través de un índice que indica su posición relativa en la estructura.

Esta estructura ha sido previamente implementada en Java de modo que se proporcionan un conjunto de métodos básicos para su facilitar el manejo:

private ArrayList < TipoElementos > NombreLista ⟶ permite definir un objeto tipo lista

TipoElementos[Tamaño] $NomnreLista$ ⟶ permite definir un objeto tipo lista de un tamaño determinado

nombreLista[0] ⟶ accede a la primera posicion de la lista

nombreLista[NombreLista. lenght() − 1] ⟶ accede a la ultima posicion de la lista

nombreLista. add( ) ⟶ añade un elembento

nombreLista. remove( ) ⟶ elimina la 1ª aparición de un elemento y desplaza todos los demas

nombreLista. size() ⟶ dice cuántos elementos tiene el contenedor

nombreLista. contains() ⟶ dice si un elemento está en el contenedor

nombreLista. iterator( ) ⟶ devuelve un objeto detipo iterador que permite recorrer los elementos

iterador. next( ) ⟶ devuelve elemento actual y apunta al siguiente

iterador. hasNext( ) ⟶ dice si el elemento actual es o no es null

```
import java.util.ArrayList;
import java.util.Iterator;

public class ListaPista
{
    private ArrayList<Pista> lista;

    public ListaPista() { this.lista = new ArrayList<Pista>(); }

    private Iterator<Pista> getIterador() { return this.lista.iterator(); }

    public void añadirPista(Pista pPista){ this.lista.add(pPista); }

    public int getNumeroPistas(){ return this.lista.size(); }

    public int getDuracionDisco(){
        Iterator<Pista> itr = this.getIterador();
        int total = 0;
        while(itr.hasNext())
        {
            Pista pista = itr.next();
            total += pista.getDuracion();
        }
        return total;
    }
}
```

Las ventajas de las listas frente a otro tipo de estructura de datos son:

- El coste de acceso a un elemento cuya posiciones conocida es constante
-

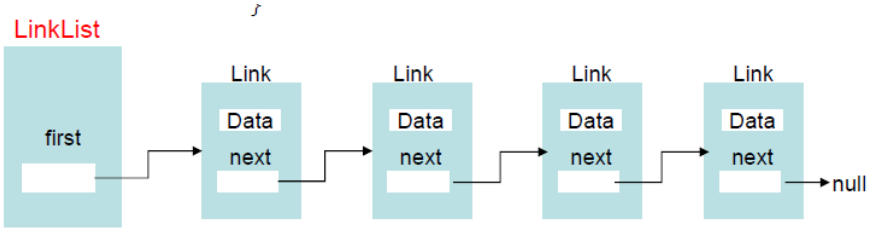
Las desventajas de las listas frente a otro tipo de estructura de datos son:

- Al tener un tamaño fijo, es necesario conocer de antemano el número de elementos que formaran parte de la estructura

Estructuras enlazadas

Se trata de una estructura de datos que utiliza variables de referencia a objetos para crear enlaces entre los mismos. Esta estructura se implementa mediante un puntero que indica la posición de memoria del primer elemento de la lista, y un conjunto de nodos que constituirán los elementos de la lista y estarán formados por:

- un conjunto de campos de datos donde se almacenara la información relativa a la lista
- un conjunto de campos de referencias de la clase del nodo.




```java
public class LinkList
{
    // Atributos
    private Link first;
    // Constructora
    public LinkList() { first = null; }
}
```

```java
public class Link
{
    // Atributos
    private TipoDato data;
    private Link next;
    // Constructora
    public Link( TipoDato pDato ) { next = null; }
}
```

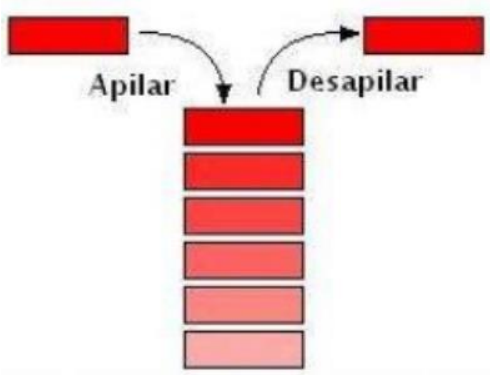
Pilas de datos

Se trata de una colección lineal de elementos en la que sólo se pueden añadir y retirar elementos por uno de sus extremos.

import java. util. Stack;

private Stack < Tipo > nombrePila = new Stack < Tipo > ( ); ⟶ permite definir un objeto tipo pila

Operación	Descripción
void push(T elem)	Añade un elemento en la cima de la pila
T pop()	Elimina y devuelve el elemento de la cima de la pila (presupone que la pila no está vacía)
T peek()	Da acceso al elemento situado en la cima de la pila (presupone que la pila no está vacía)
boolean isEmpty()	Determina si la pila está vacía
int size()	Determina el número de elementos de la pila



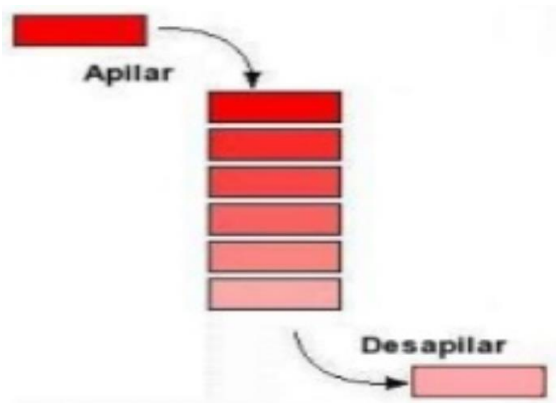
Colas de datos

Se trata de una colección lineal de elementos en la que se puede añadir elementos por un extremo pero se retiran por el otro.

import java. util. Queue;

private Queue < Tipo > nombreCola = new ArrayDeque < Tipo > ( ); ⟶ permite definir un objeto tipo pila

Operación	Descripción
void insert(T elem)	Añade un elemento al final de la cola
T remove()	Elimina y devuelve el elemento del principio de la cola (supone que la cola no está vacía)
T first()	Da acceso al elemento del principio de la cola (supone que la cola no está vacía)
boolean isEmpty()	Determina si la cola está vacía
int size()	Determina el número de elementos de la cola



Listas de Pilas o Colas

Para agrupar Pilas, Colas o cualquier otra estructura de datos en una lista es necesario:

- Inicializar la lista:
  ```
  private Stack < Tipo > [] nombreListadePilas = new Stack [];
  private Queue < Tipo > [] nombreListadeColas = new ArrayDeque [];
  ```
- Inicializar la pila o cola de cada posición de la lista:
  ```
  for (int i = 0 , i < nombreListadePilas . size( ) , i + +) { nombreListadePilas[ i ] = new Stack < Tipo > ( ); }
  for (int i = 0 , i < nombreListadeColas . size( ) , i + +) { nombreListadeColas[ i ] = new ArrayDeque < Tipo > ( ); }
  ```

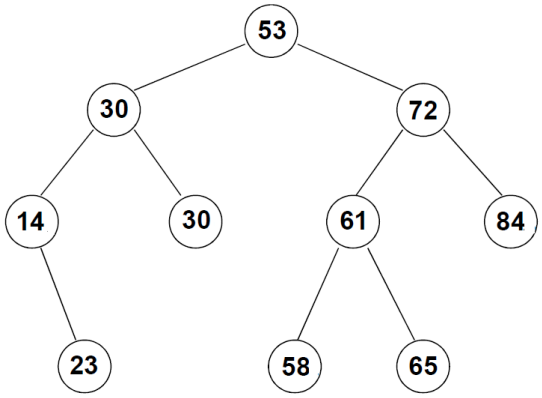
Arboles binarios

Se trata de una estructura enlazada en la que cada nodo tiene dos referencias con otros nodos del mismo tipo, de tal forma que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo son menores que el elemento almacenado en el nodo padre del mismo modo que los elementos almacenados en el subárbol derecho es mayor o igual que el elemento almacenado en el nodo padre.

Los arboles binarios se utilizan principalmente para almacenar un conjunto de datos ordenables en un orden determinado.

```
import java. util. TreeSet;
private TreeSet < Tipo > nombreArbol = new TreeSet < Tipo > ( );
```

Operación	Descripción
add	Añade el elemento especificado al árbol
remove	Elimina el elemento especificado del árbol
removeMin	Elimina el elemento de valor mínimo del árbol
removeMax	Elimina el elemento de valor máximo del árbol
findMin	Devuelve el elemento de valor mínimo del árbol
findMax	Devuelve el elemento de valor máximo del árbol



Tablas Hash

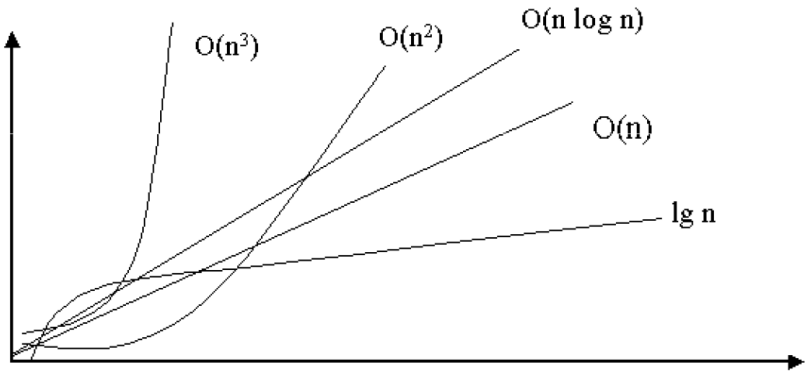
Se trata de una matriz asociativa en la que mediante un sistema de codificación se le asigna un valor numérico a cada palabra clave. De esta forma es posible almacenar un conjunto de valores en una posición única de una lista

```
private HashMap < TipoKey , TipoDato > nombreTablaHash = new HashMap < TipoKey , TipoDato > ( );
```

Métodos	***Significado***
V **put** (K key, V value)	Asocia el valor V a la clave K en la aplicación. Si la clave ya existe modifica el valor viejo por el valor especificado. En este caso devuelve el valor viejo.
V **get** (K key)	Devuelve el valor asociado a la clave. Si la clave no existe devuelve *null*
V **remove** (K key)	Elimina la clave y su valor asociado. Devuelve el valor asociado a la clave o *null* si no existe la clave.
void **clear** ()	Elimina todos los elementos de la tabla
boolean **isEmpty**()	True si está vacía, false e.o.c.
boolean **containsKey** (K key)	True si la clave está en la aplicación, false e.o.c.
int **size** ()	Devuelve el número de elementos de la tabla.

Analisis de costes:

Consiste en calcular el número de operaciones elementales que realiza el programa en función del tamaño de los parámetros de entrada haciendo una abstracción del tiempo real de realización de una operación elemental



- O(1) *constante*
- O(log n) *logarítmica*
- O(n) *lineal*
- O(n.log n) *quasilineal*
- $O(n^2)$ *cuadrática*
- $O(n^3)$ *cúbica*
- $O(n^k)$ *polinómica*
- $O(2^n)$ *exponencial*

Algoritmos de ordenacion

Burbuja: $O(n^2)$

Compara todos los elementos de una lista dos a dos intercambiando sus valores si no están en el orden adecuado y descartando el último elemento tras cada iteración.
Este proceso se repite (n – 1) veces siendo n el número de elementos de la lista.

Seleccion: $O(n^2)$

Compara todos los elementos de la lista buscando el que pertenezca a la primera posición, cuando lo encuentra intercambia sus valores y descarta esa posición para repetir el proceso en las próximas iteraciones.
Este proceso se repite (n – 1) veces siendo n el número de elementos de la lista.
Este método es ligeramente más eficiente que la burbuja, porque a pesar de que hace las mismas comparaciones, no hace tantos intercambios.

Insercion: $O(n * \log_2(n))$

Se construye una lista auxiliar en la que se insertan los elementos de la lista original directamente en su posición correspondiente. A pesar de que la lista original solo se recorre una vez, hay que buscar la posición correcta de cada elemento en la lista auxiliar que ya se encuentra ordenada
Este método será más eficiente con listas que estén casi ordenadas debido a que solo compara los elementos de la lista hasta localizar su posición correspondiente.

Fusion o MergeSort : $O(n * \log_2(n))$

Divide la lista original en pares de valores y compara cada par intercambiando sus valores si no están en el orden correcto. A continuación combina los pares dos a dos y sabiendo que estos están ordenados entre si y los combina obteniendo un conjunto ordenado. El proceso sigue hasta que el único par restante es la lista completa ya ordenada
Sabiendo que dos pares están ordenados solo es necesario comparar sus primeros valores y el menor de ellos será el menor del par, siguiendo con la condición se ordena todo el conjunto.

Rapida o QuickSort : $O(n * \log_2(n))$

Selecciona un elemento aleatorio de la lista (generalmente el primer elemento) y lo compara con los siguientes de tal modo que todos los valores menores que dicho elemento quedan a su derecha y los mayores a su izquierda. A continuación este elemento queda fijado y se repite el proceso para cada una de las sub listas resultantes. El proceso sigue hasta que todos los elementos han quedado fijados.
Obsérvese que tras cada iteración las listas a ordenar tendrán la mitad del tamaño original
Este método tiene el inconveniente de que si la lista ya está ordenada solo se creara una sub lista con el tamaño de la original menos un elemento, forzando a realizar todas las comparaciones posibles entre los elementos de la lista.

Patrones

Son un conjunto de técnicas empleadas con la finalidad de maximizar la reutilización de un código

Podemos encontrar tres tipos de patrones:

- **Generadores:** relacionados con la creación de objetos.
- **Estructurales:** se encargan de la composición de clases y objetos.
- **Comportamiento:** describen cómo se deben repartir las interacciones y responsabilidades de clases y objetos.

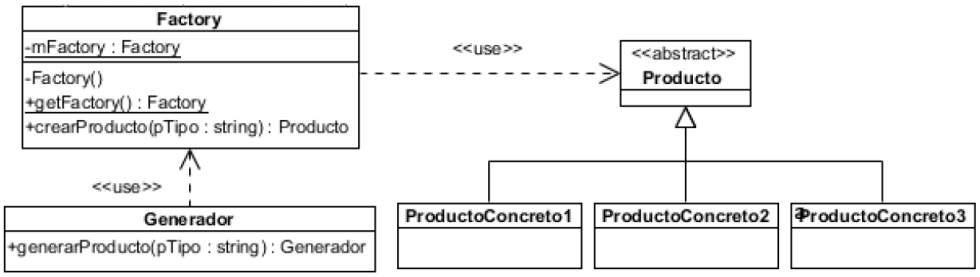
Singleton Iterator

```
import java.util.ArrayList;
import java.util.Iterator;

public class SingletonLista
{
 // Atributos
    private ArrayList<Objeto> lista;
    private static SingletonLista miSingletonLista = new SingletonLista();
 // Constructora
    private SingletonLista() { this.lista = new ArrayList<Objeto>(); }
 // Getters
    public static SingletonLista getSingletonLista() { return miSingletonLista; }
    private Iterator<Trabajador> getIterador() { return this.lista.iterator(); }
 // Funciones
    public int metodoSobreLista()
    {
        Iterator<Objeto> itr = this.getIterador();
        int resultado = 0;
        Objeto pObjeto;
        while ( itr.hasNext() )
        {
            pObjeto = itr.next();
            resultado = pObjeto.calcular(resultado);
        }
        return resultado;
    }
}

public class ClaseMAE
{
 // Atributos
    private int numero;
    private String texto;
    private static ClaseMAE miClaseMAE = null;
 // Constructora
    private ClaseMAE( int pAtributo , String pTexto )
    {
        this.atributo = pAtributo;
        this.texto = pTexto;
    }
 // Getters
    public static ClaseMAE getClaseMAE( int pAtributo , String pTexto )
    {
        if ( miClaseMAE == null ) { miClaseMAE = new ClaseMAE( int pAtributo , String pTexto ); }
        return miClaseMAE;
    }
}
```

Factory

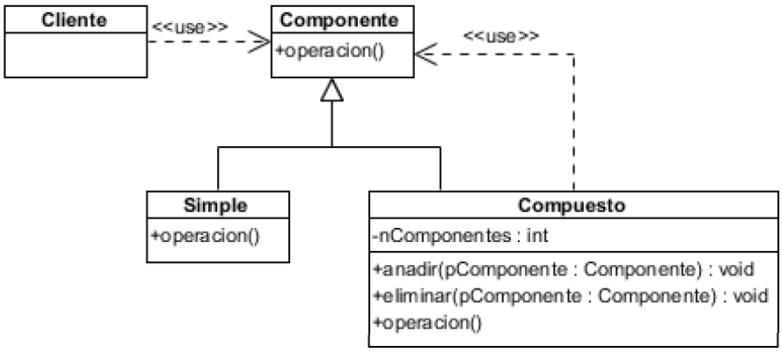
Se trata de una clase que conoce las constructoras de todas las clases hijas de otra clase abstracta de modo que:

- Permite a clases de terceros crear objetos de instancia sin conocer sus constructoras.
- Para Ampliar la clase abstracta, añadiéndole una nueva clase hija, es suficiente con modificar la Factory.
- El encapsulado de constructoras permite controlar las instancias de una determinada clase



Composite

Se trata de un patrón que permite definir una superclase común para un conjunto de objetos simples y compuestos con un método común. De este modo se simplifica definición y gestión de nuevos objetos minimizando la complejidad del objeto compuesto al crear de forma recursiva una jerarquía de clases que permite tratar a los objetos compuestos de la misma manera que a los simples



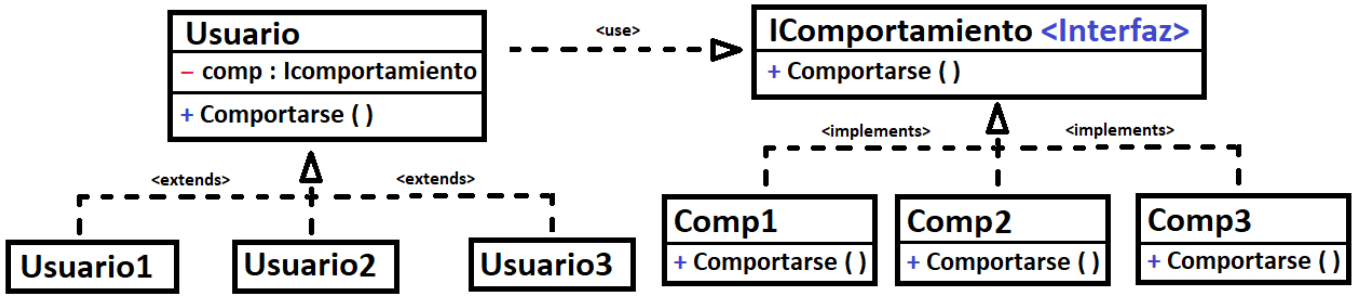
Facade

Se trata de una clase que permite ejecutar un conjunto de métodos de otras clases en un orden predeterminado. Este patrón permite minimizar el acoplamiento entre las distintas clases de un programa ya que

- Se aísla al cliente de los subsistemas: Los clientes pueden usar los subsistemas en caso de que sea necesario.
- Se separa el sistema en distintas capas: Si cambian los subsistemas, se cambia la Facade.
- Cambiar la Facade no implica cambiar los Clientes.
- Si cada Cliente usa pocos métodos de la Facade, se pueden definir distintas Facades.

Strategy

Se trata de un patrón que permite que cuando una clase ofrezca un servicio que puede hacerse de diferentes maneras, el cliente pueda decidir dinámicamente la forma concreta de realizar dicho servicio.
Para ello se usa una interfaz que puede ser implementada por diferentes clases



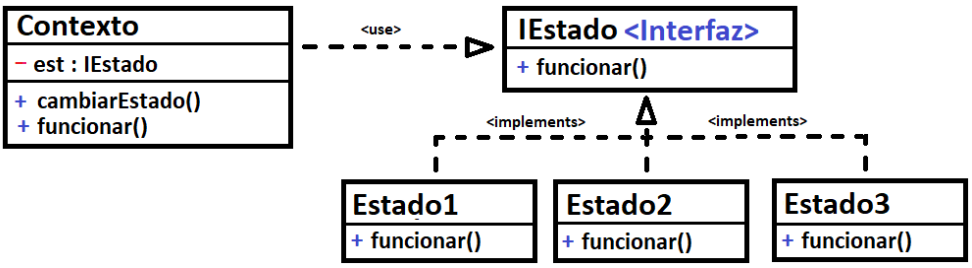

```
public class Usuario1 extends Usuario
{
 // Atributos
    private Icomportamiento com;
 // Constructora
    public Usuario1() { com = new comp1(); }
 // Funciones
    public void cambiarComportamiento ( IComportamiento pComp ) { this.com = pComp }
    public void comportarse(){ this.com.comportarse(); }
}

public class comp1 implements Icomportamiento
{
 // Constructora
    public comp1( ... ){  ...  }
 // Funciones
    public void comportarse() { ... }
}
```

State

Se trata de un patrón que permite que un objeto pueda modificar su comportamiento en función de su estado:

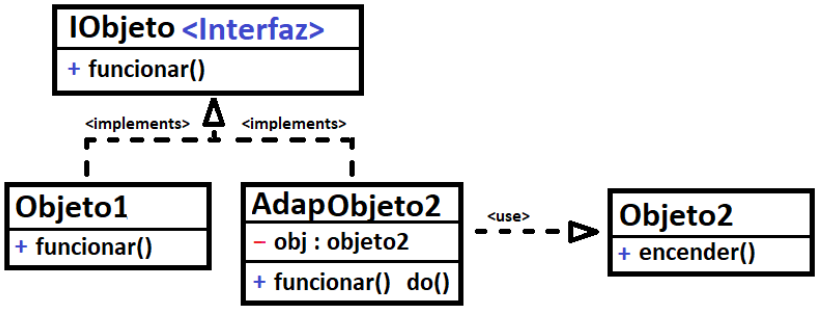
- El estado de un objeto es el conjunto de los valores de sus atributos en un momento concreto.
- Se permite un comportamiento distinto dependiendo de cada situación
- El comportamiento en cada estado está encapsulado
- Las transiciones entre estados son explícitas
- Es extensible

```
public class Contexto extends Usuario
{
 // Atributos
    private IEstado est;
 // Constructora
    public Contexto() { est = new comp1(); }
 // Funciones
    public void cambiarEstado ( IEstado pEst ) { this.est = pEst;}
    public void funcionar(){ this.est.funcionar(); }
}

public class Estado1 implements IEstado
{
 // Constructora
    public estado1( ... ){  ...  }
 // Funciones
    public void funcionar() { ... }
}
```



Adapter

Se trata de un patrón que permite que dos clases implementen una misma interfaz a pesar de tener nombres distintos para sus métodos sin modificar dichos nombres.




```
public interface Objeto { public void funcionar(); }

public class Objeto1 implements Guitarra
{
 // Constructora
    public Objeto1(){}
 // Funciones
    public void funcionar(){...} }

public class Objeto2
{
 // Constructora
    public Objeto1(){}
 // Funciones
    public void encender(){...} }

public class AdaptObjeto2 implements Objeto
{
 // Atributos
    private Obj Objeto2
 // Constructora
    public AdaptObjeto2(){ this.obj = new Objeto2(); }
 // Funciones
    public void encender(){ Obj.encender(); }
}
```

Observer

Como los objetos dependen unos de otros, el cambio de estado de un objeto provoca cambios en otros objetos.

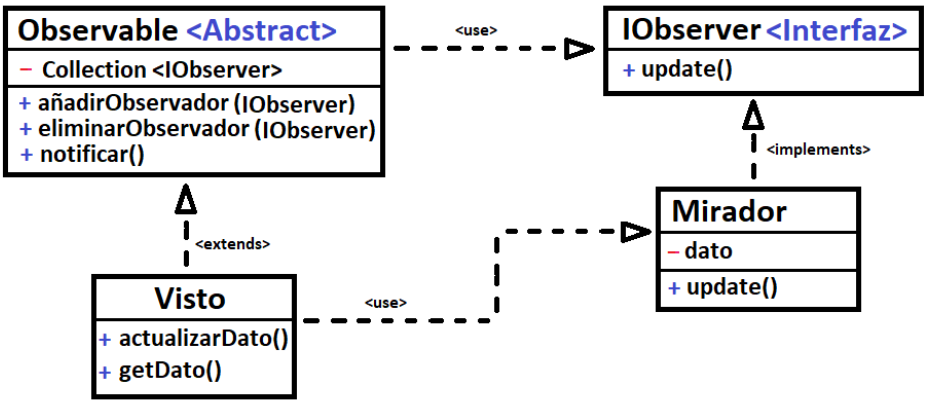
- No deseamos que sea necesario conocer el número de objetos que deben ser modificados en cada cambio de estado particular.
- Queremos que un objeto sea capaz de notificar a otros objetos que ha cambiado, sin hacer ninguna suposición acerca de los objetos notificados.

Mediante este patrón conseguimos

- Evitar el acoplamiento entre el sujeto y el observador.
- El sujeto no necesita especificar a los observadores afectados por un cambio.

Se basa en que cada vez que una clase observada realice una modificación en sus datos envía una notificación a todos los suscriptores que la están observando para que actualicen su información.

- con la aplicación de este patrón se desconocen las consecuencias de una actualización.

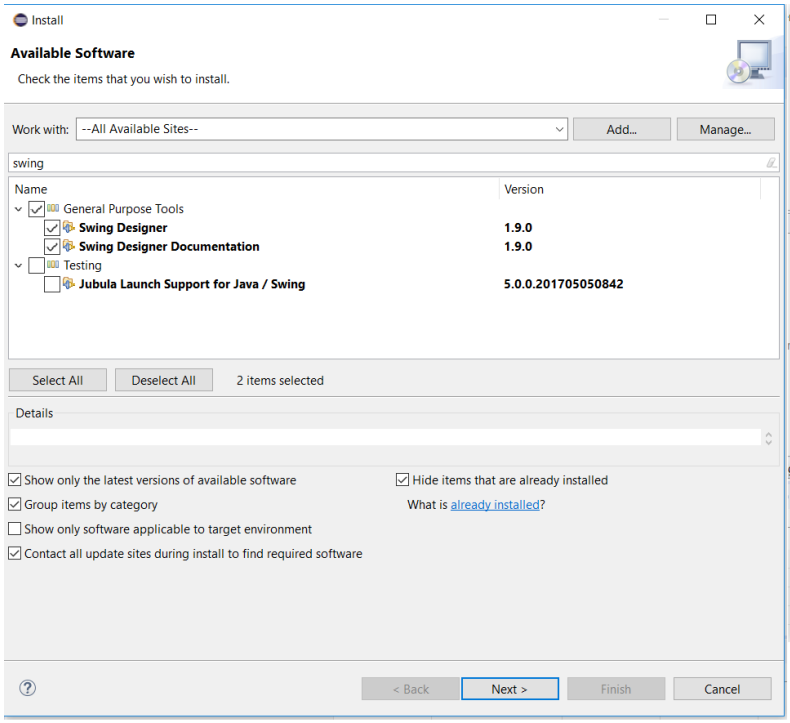


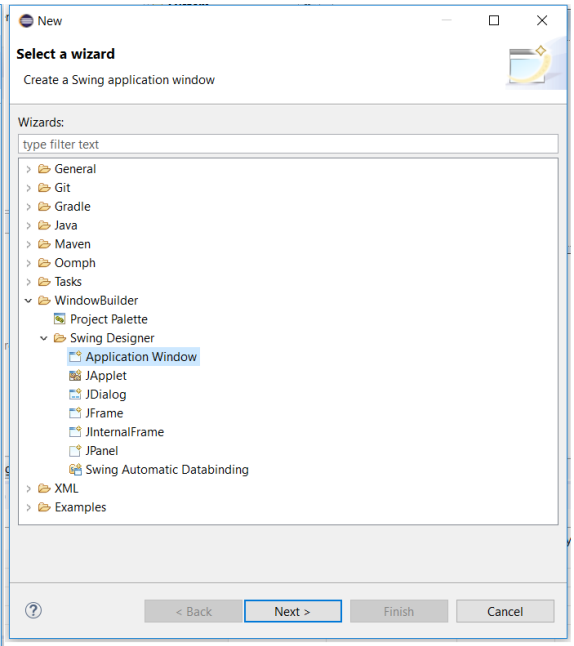

```
public abstract class Observable
{
 // Atributos
    private List<IObserver> suscriptores;
 // Constructora
    public Observable () { suscriptores=new ArrayList<IObserver>(); }
 // Funciones
    public void registrarObservador(IObserver o) { if (!suscriptores.contains(o)){ suscriptores.add(o);    } }
    public void eliminarObservador (IObserver o) { if ( suscriptores.contains(o)){ suscriptores.remove(o); } }
    public void notificar()
    {
        IObserver observador;
        Iterator it = suscriptores.iterator();
        while (it.hasNext())
        {
            observador = (IObserver)it.next();
            observador.update();
        }
    }
}
public interface IObserver { public void update(); }

public class Mirador implements IObserver
{
 // Atributos
    private int dato;
    private Mirado subject;
 // Constructora
    public Mirador(Mirado pSubject)
    {
        subject = pSubject;
        subject.registrarObservador(this);
    }
 // Funciones
    public void setDato(int pDato){ this.dato = pDato; }
    public void mostrarDato(){ System.out.println("el dato es:" + this.dato); }
    public void update(){ this.dato = subject.getDato();}
}

public class Visto extends Observable
{
 // Atributos
    private int dato;
    private List<IObserver> suscriptores;
 // Constructora
    public Visto (){ super(); setDato(0); }
    public int setDato (int pDato){ this.dato = pDato; }
    public int getDato(){ return this.dato; }
    public void actualizarDato(/*parámetros*/)
    {
     // Funciones que modifiquen el dato;
        super.notificar();
    }
}
```

Interfaces graficas

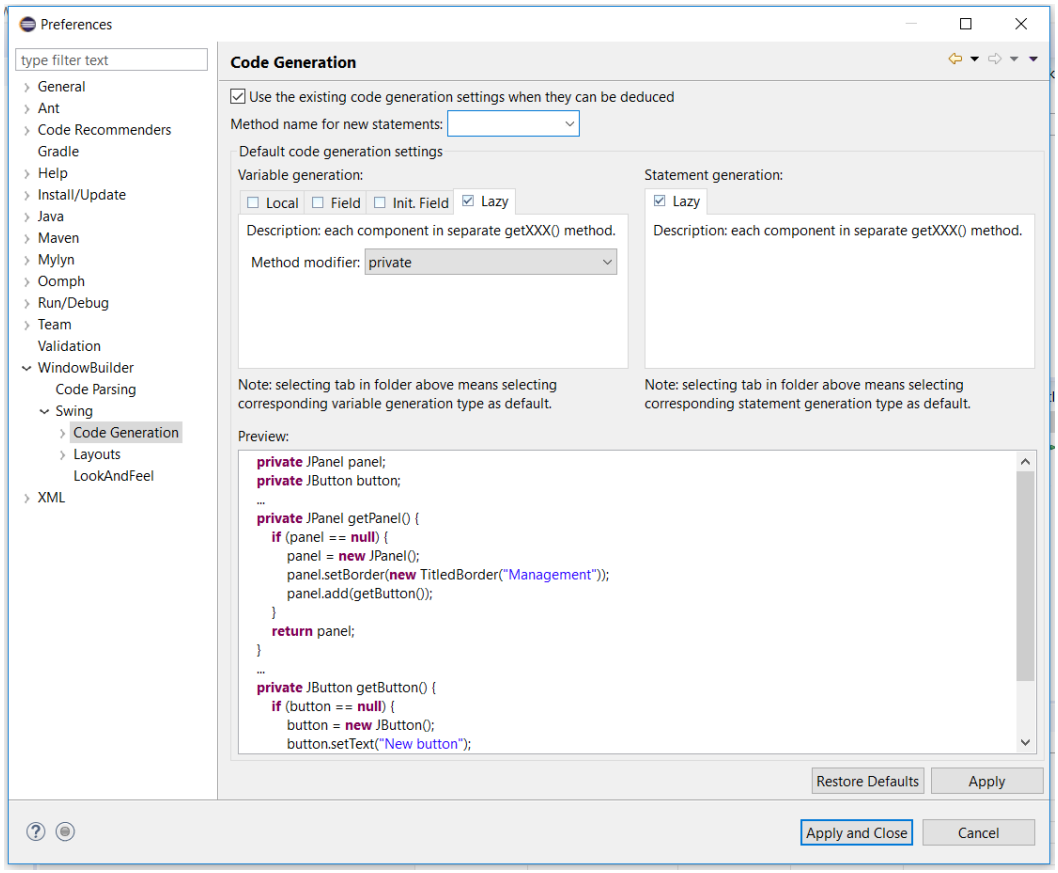
Instalación

Help --> Install New Software





Window --> Preferences



Librerías

Java dispone de las siguientes librerías para el diseño de interfaces graficas

```
import javax.swing.*; ————————→ Librería que permite importar todos los componentes gráficos
import javax.swing.event.*; ——→ Librería que permite manejar los eventos de swing
import java.awt.*; ——————————→ Librería que permite importar funcionalidades sobre el color
import java.awt.event.*; ————→ Librería que permite manjar los eventos de awt
```

Declaración de clase

```
public class ClaseVentana
    extends JFrame ————————————————→ Permite heredar todos los componentes predefinidos
    implements ActionListener, ItemListener, ChangeListener
```

- Permite habilitar las funciones de captura de eventos
  - ActionListener captura de eventos para botones
  - ItemListener captura de eventos para los comboBoxs
  - changeListener captura de eventos para los checkBoxs

Componentes

Menú

Declaración de variables:

```
private JMenuBar miMenuBar;
```

- Los menús se almacenan una barra de menús
- Restringida a un máximo de una por ventana
- Puede almacenar muchos menús
- No necesita coordenadas porque siempre se ubica en la parte superior de la ventana

```
private JMenu menu_nombre_i;
```

- Cada menú puede almacenar otros menús o funcionalidades
- Se puede añadir el menú a otro menú o a la barra de menús

```
private JMenuItem menuItem_nombre_1;
```

- Las funcionalidades implementan la clase ActionListener que permite disparar un evento

Inicialización

```
miMenuBar = new JMenuBar();
setJMenuBar( miMenuBar);
menu_nombre_i = new JMenu(" Nombre Grafico del Menu ");
nombreMenuAñadir.add( menu_nombre_i );
menuItem_nombre_1 = new JMenuItem("Nombre de la funcionalidad");
nombreMenuAñadir.add(menuItem_nombre_1);
menuItem_nombre_1.addActionListener(this);
```

Etiquetas

Se trata de un componente que puede almacenar texto o imágenes

Declaración de variables:

```
private JLabel etiq_nombre_i;
```

Inicialización

```
etiq_nombre_i = new JLabel();
getContentPane().add( etiq_nombre_i );
```

Cuadros de texto

Se trata de un componente que permite al usuario introducir texto

Declaración de variables:

```
private JTextField textField_nombre_i;
```

- Cuadro de texto simple de una sola línea

```
private JTextArea textArea_nombre_i;
```

- Cuadro de texto de múltiples líneas

```
private JScrollPane scrollPanel_nombre_i;
```

- Componente grafico que permite añadir una barra deslizable al cuadro de texto

Inicialización

```
textField_nombre_i = new JTextField();
getContentPane().add( textField_nombre_i );
```

Boton

Se trata de un componente que permite lanzar un evento ActionListener cuando el usuario lo pulsa

Declaración de variables:

```
private JButton boton_nombre_i;
```

Inicialización

```
boton_nombre_i = new JButton( "Nombre de la funcionalidad" );
getContentPane().add( boton_nombre_i );
boton_nombre_i.addActionListener( this );
```

Listas Desplegables

Se trata de un componente que permite al usuario seleccionar una opción de entre las previamente determinadas por el programador. Además permite lanzar un evento del tipo ItemListener

Declaración de variables:

```
private JComboBox combo_nombre_i;
```

Inicialización

```
combo_nombre_i = new JComboBox();
getContentPane().add( combo_nombre_i );
combo_nombre_i.addItemListener(this);
```

Añadir elementos a la lista:

```
combo_nombre_i .addItem( "Nombre de la opcion desplegable");
for( int i = 0 ; i <= 255 ; i + + ) { combo_nombre_i .addItem( i int ); }
```

Casillas Check Múltiple

Se trata de un componente que permite al usuario seleccionar múltiples opciones

Constructora

```
private JCheckBox check_nombre_i;
```

Inicialización

```
check_nombre_i = new JCheckBox("Nombre de la funcionalidad");
getContentPane().add( check_3 );
check_3.addChangeListener(this);
```

Casillas Check Simple

Se trata de un componente que permite al usuario seleccionar una única opción (se deselecciona la anterior al pulsar otra)

Constructora

```
private ButtonGroup botonGrup_nombre_i;
```

- Todos las casillas de una misma selección se deben encerrar en un mismo grupo

```
private JRadioButton radio_nombre_i;
```

Inicialización

```
botonGrup_nombre_i = new ButtonGroup();
radio_nombre_i = new JRadioButton("Nombre de la funcionalidad");
getContentPane().add( radio_nombre_i );
botonGrup_nombre_i.add( radio_nombre_i );
radio_nombre_i.addChangeListener( this );
```

Ventana

Constructora

```
getContentPane().setLayout(null);
```

- Permite que la constructora no asigne a la ventana los parámetros por defecto

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

- Cuando se cierra la ventana se termina la ejecucion

```
setTitle("Bar Bestial");
```

- Permite modificar el título de la ventana

```
getContentPane().setBackground( color Color );
```

- Cambia el color del fondo de la ventana

```
setIconImage(new ImageIcon(getClass().getResource( imagen String )).getImage());
```

- Cambia la imagen de icono de la aplicación

Inicialización

```
ClaseVentana nombre_Ventana_i = new MenuPartida();
nombre_Ventana_i.setBounds( X int , Y int , Ancho int , Alto int );
nombre_Ventana_i.setVisible(true);
nombre_Ventana_i.setLocationRelativeTo(null);
```

- Arranca la ventana siempre desde el centro

```
nombre_Ventana_i.setResizable(false);
```

- permitir/impedir que el usuario cambie el tamaño de la ventana

Funciones comunes a todos los objetos gráficos

Permite establecer la una cadena de texto dentro del objeto grafico

```
nombreObjetoGrafico . setText( String );
```

Permite extraer el texto de un objeto grafico

```
String texto = etiq_nombre_i . getText( );
```

Permite establecer una imagen dentro del objeto grafico

```
nombreObjetoGrafico . setIcon( imagen imageIcon );
```

Permite cambiar la visibilidad de un objeto grafico

```
nombreObjetoGrafico . setVisible( estado boolean );
```

- false ⟶ el objeto grafico se oculta
- true ⟶ el objeto grafico se muestra

Permite cambiar el estado de un objeto grafico

```
nombreObjetoGrafico . setEnabled( estado boolean );
```

- false ⟶ no permite ninguna interacción con el objeto grafico se oculta
- true ⟶ permite al usuario interactuar con el objeto grafico

Permite modificar posición y tamaño del objeto grafico

```
nombreObjetoGrafico. setBounds( X int , Y int , Ancho int , Alto int );
```

- x ⟶ Coordenadas derecha – izquierda
- y ⟶ Coordenadas arriba – abajo
- Ancho ⟶ Anchura del objeto grafico
- Alto ⟶ Altura del objeto grafico

Permite modificar el tipo de letra y tamaño del texto de un objeto grafico

```
nombreObjetoGrafico. setFont( new Font( fuenteLetra string , formato int , tamaño int ) );
```

- fuenteLetra ⟶ “Andele Mono”
- formato ⟶ 1 : Negrita 2 : cursiva 3 : negrita y cursiva
- tamaño ⟶

Permite modificar color de fondo del objeto grafico

```
nombreObjetoGrafico. setForeground( new Color(R int G int , B int ));
```

- R G B ⟶ [0 − 255] intensidad del color

Colocar una imagen en un objeto

```
private String im_ruta = "packImagenes/cocodrilo. png";
ImageIcon im_1 = new ImageIcon(getClass(). getResource( im_ruta ));
Image im_2 = im_1. getImage(). getScaledInstance(Ancho int , Alto int , java. awt. Image. SCALE_SMOOTH);
ImageIcon imagenTamañoCorrecto = new ImageIcon( im_2 );
```

Generalidad en las Listas

Interfaces funcionales

Son interfaces que implementan un único método abstracto, se caracterizan porque:

- No se consideran los métodos heredados de Object
- Representan funciones o condiciones
- Se pueden pasar como parámetro listaDeClaseListada(comparacion(ClaseListada :: funcionAplicada));

Comparator <T>

Es posible crear diferentes criterios de ordenación implementando la interfaz Comparator:

- Añadida por la librería: java.util.Comparator
- Su método abstracto es compare(Object pClase1 , Object pClase2)
- Se incluye el método short(Comparator < T > pComparacion) que permite ordenar una lista en función del criterio de comparación que reciba como parámetro

```
public class Comparacion implements Comparator<ClaseListada>
{
    public int compare(ClaseListada pComp1, ClaseListada pComp2)
    {
        return  pComp1.compareTo( pComp2 );
        // si devuelve un numero positivo pComp1 va delante
        // si devuelve un numero negativo pComp2 va delante
        // si devuelve cero no importa quien va delante
    }
}
public class Principal
{
    public void metodoOrdenar( Comparator<ClaseListada> pComp )
    {
        listaDeClaseListada.sort(pComp);
    }
}
```

Predicate <T>

Es posible crear diferentes criterios de selección implementando la interfaz Predicate:

- Su método abstracto es test (Object pClase)

```
// Con Expresiones Landa
public class ExpresionLanda implements Predicate<ClaseListada>
{
    public boolean test(ClaseListada pEntrada)
    {
        return pEntrada.metodoDeLaClaseListadaQueDevuelvaBoolean();
    }
}
```

Function <T, R>

Es posible aplicar funciones a todos los elementos de una lista implementando la interfaz Function

- **T** representa el tipo del parámetro de entrada
- **R** es el tipo del resultado de la función
- Su método abstracto es apply (Object pClase)
- Se incluye el método nap (Function < T , R > pFuncion) que permite ejecutar la función que recibe como parámetro a cada elemento de la lista que lo llame

Existen dos tipos de Function:

- **UnaryOperator<T>** Especialización de Function para casos en los que:
  - El parámetro de entrada y el resultado son del mismo tipo.
  - Solo tienen un parámetro de entrada

  Su método abstracto es excec (Object pClase)
- **BinaryOperator<T>** Especialización de Function para casos en los que:
  - El parámetro de entrada y el resultado son del mismo tipo.
  - Solo tienen un parámetro de entrada

  Su método abstracto es excec (Object pClase1 , Object pClase2)

Expresiones Landa

Proporcionan implementación a las interfaces funcionales sin tener que definir una nueva clase

Sintaxis (parametros) –> cuerpo

Los parámetros son la lista de parámetros formales del método abstracto de la interfaz funcional

- Se puede omitir los paréntesis cuando sólo hay un parámetro
- Se puede omitir el tipo del parámetro formal
- Se puede omitir la instrucción de return puesto que es implícita

El cuerpo consiste en un bloque de instrucciones o una expresión.

- Los bloques deben ir entre llaves

Utilización

Es posible construir expresiones landa combinando varias expresiones simples mediante:

- Una operación AND u OR entre dos o más predicados.
- Componiendo funciones de tal manera que el resultado de una función se convierta en la entrada de otra.

Stream

Es una secuencia de elementos permite manipular colecciones de datos de forma declarativa. Sólo se expresa lo que se espera que haga la función, sin indicar específicamente cómo se recorren internamente los elementos de la colección.

Añadida por la librería: java. util. stream. Stream

Se caracteriza porque:

- Soporta el procesamiento de operaciones con datos
- Posibilidad de encadenar varias operaciones como bloques de construcción
- Simplifica la comprensión y modificación de código.
- Permite procesar en paralelo de forma transparente: Las operaciones de filtrado, clasificaciones, mapeos y recolección están disponibles como bloques de alto nivel independientes del modelo de hilos específico que se use, por lo que no es necesario escribir ningún código multiproceso ni preocuparse de los threads y locks para indicar la manera de paralelizar las tareas de procesamiento de datos (se encarga la API Stream)

Funciones de la clase Stream:

Convierte una Colección en un Stream

nombreCollection. stream ( )

Convierte un Stream en una Colección

nombreStream. collect (toList())

Recorre cada elemento p del Stream y devuelve otro Stream incluyendo solo aquellos elementos que cumplían la condición

nombreStream. filter (p –> exprComparativa)

Recorre cada elemento del Stream y lo ordena en función del atributo de la clase que se halla elegido

nombreStream. sorted (ClaseListadaEnElStream :: getAtributoComparador)

Recorre cada elemento del Stream y devuelve otro Stream que contiene el atributo seleccionado

nombreStream. map (ClaseListadaEnElStream :: getAtributoSeleccionado)

Recorre cada elemento p del Stream y ejecuta la función indicada

nombreStream. forEach (p –> funcionAplicada)

Recorre cada elemento del Stream y devuelve otro Stream eliminando los elementos que estén repetidos

nombreStream. distinct ( )

Devuelve un Stream en el que solo se incluyen los n primeros elementos

nombreStream. limit (Integer n)

Devuelve un Stream del que se han eliminado los n primeros elementos

nombreStream. skip (Integer n)

Recorre cada elemento de un Stream formado por elementos numéricos y devuelve el valor medio

nombreStream. average ( )

Recorre cada elemento de un Stream formado por elementos numéricos y devuelve

nombreStream. average ( ) ⟶ El valor medio

nombreStream. sum ( ) ⟶ El sumatorio

nombreStream. max ( ) ⟶ El máximo

nombreStream. min ( ) ⟶ El mínimo

Ejemplo:

```
public static ListaSalida<TipoSalida> funcionQueAltera( ListaEntrada<ClaseEntrada> pLista )
{
    return pLista.stream()                                          // convertimos la colección en un Stream
                 .filter( p -> p.getPrecio() < 6 )                  // filtramos algunos elementos
                 .sorted( comparing(ClaseEntrada::getPrecio) )      // ordenamos por precio
                 .map(ClaseEntrada::getNombre)                      // extraemos los nombres
                 .collect(toList());                                // recoge el resultado en una colección
}
```