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:

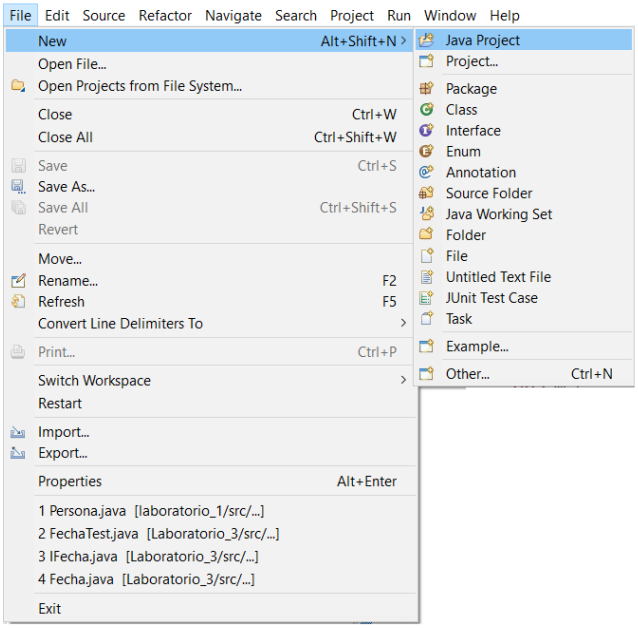
public static void main(String[] args) {

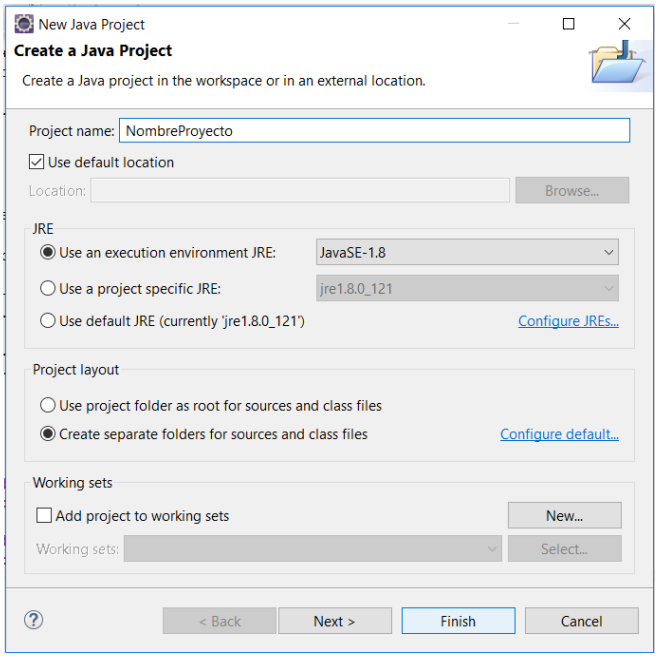
NombreClase InstanciaClase = new NombreClase (Parametros_Inicializacion);

Proyectos:

- Crear un proyecto

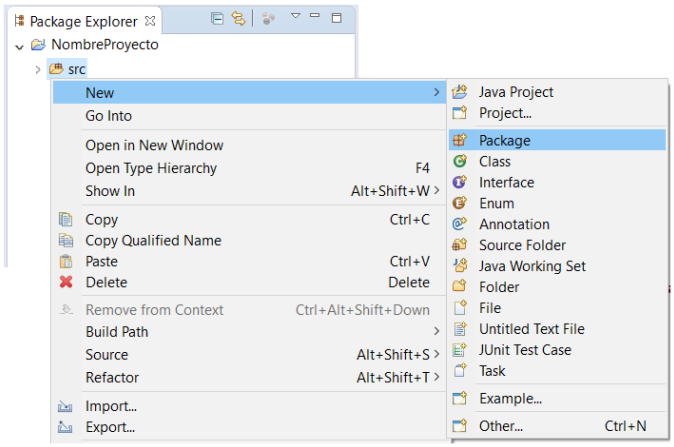
File → New → Project → Java → Java Project

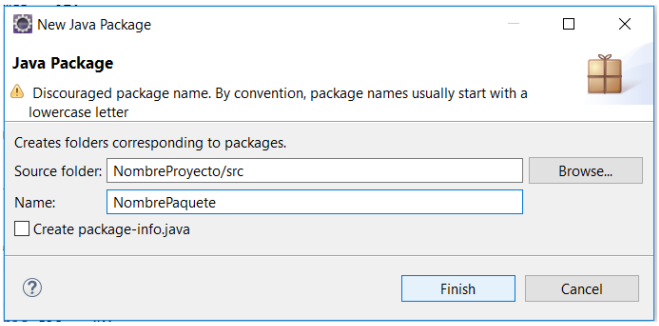




- Crear un paquete en el que agrupar los ficheros fuente

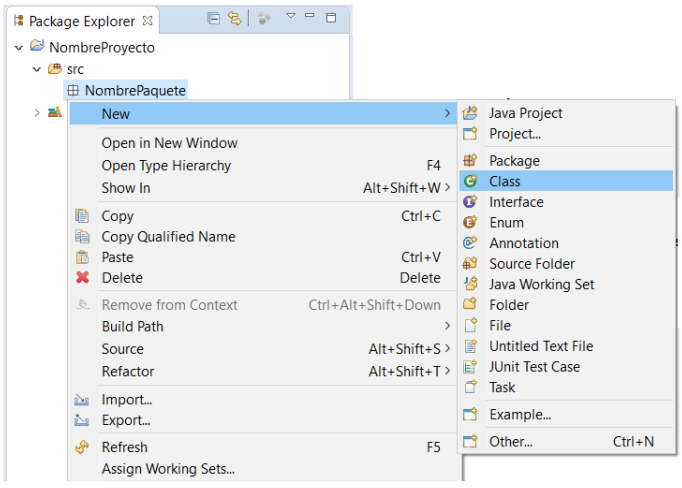
Package Explorer → boton derecho sobre la carpeta source "src" *→ New → Package.*

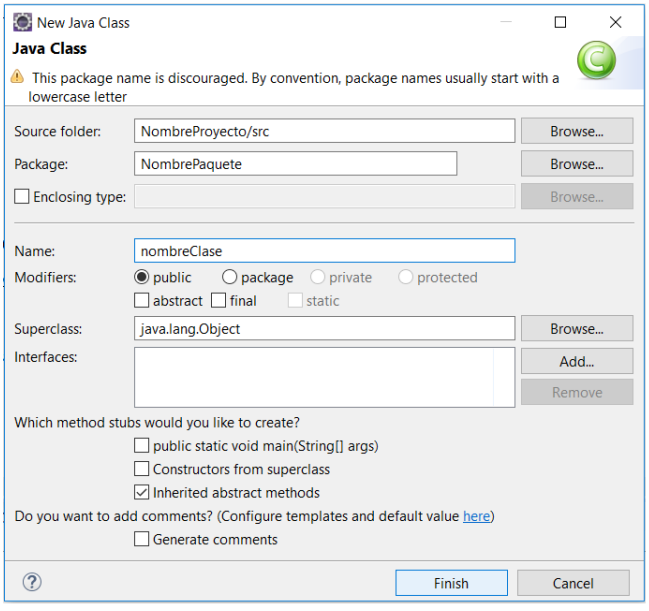




- Crear una clase

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



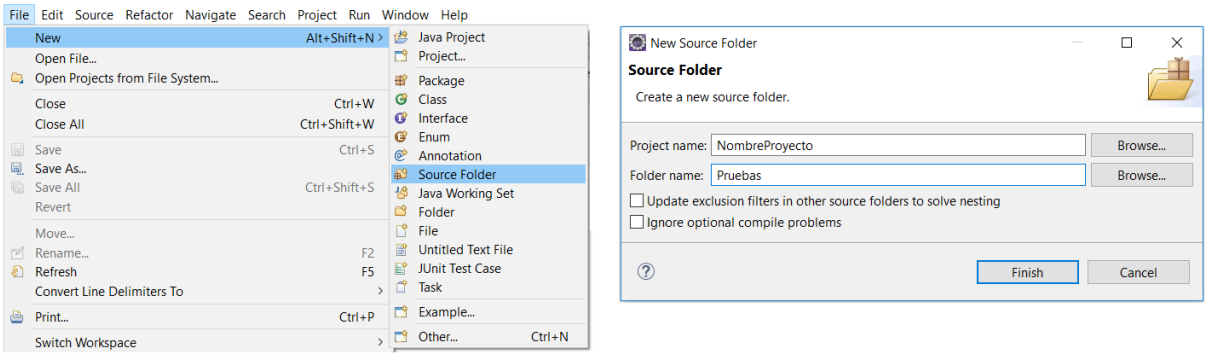


JUnits:

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.

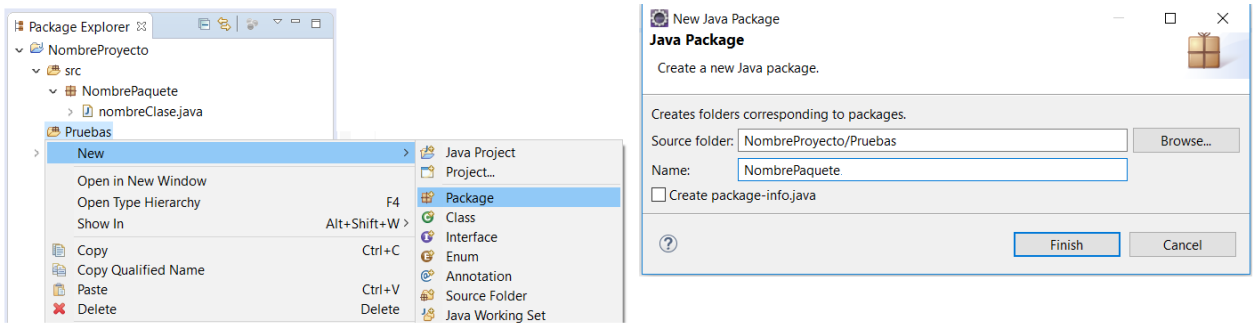
- Deberemos crear una nueva carpeta

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



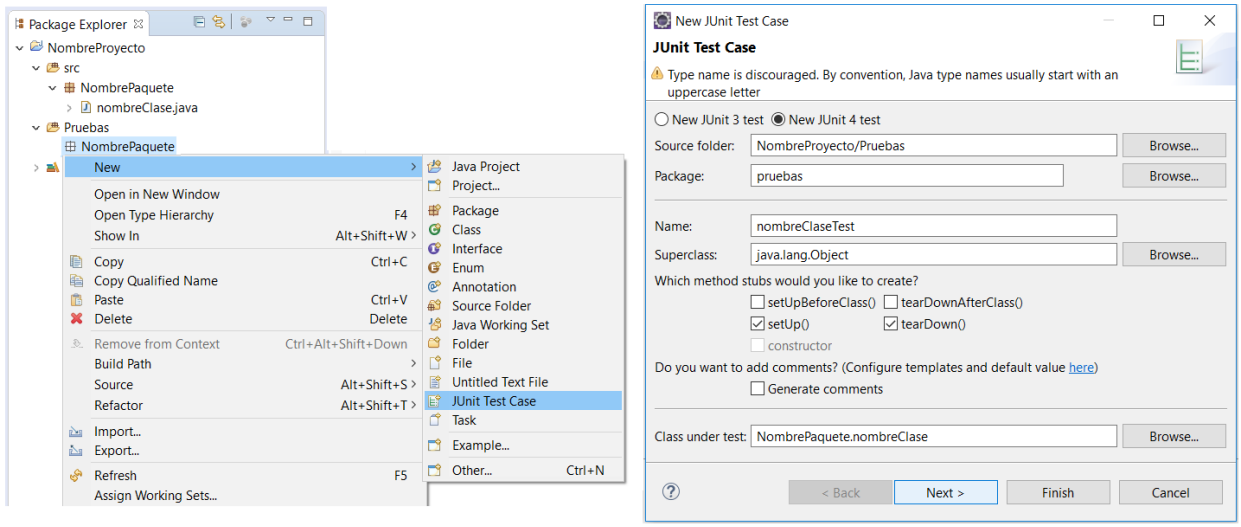
- Debe formar parte del mismo paquete en el que se encuentra la clase que se pretende evaluar
  - llevará el mismo nombre que el paquete al que simula

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

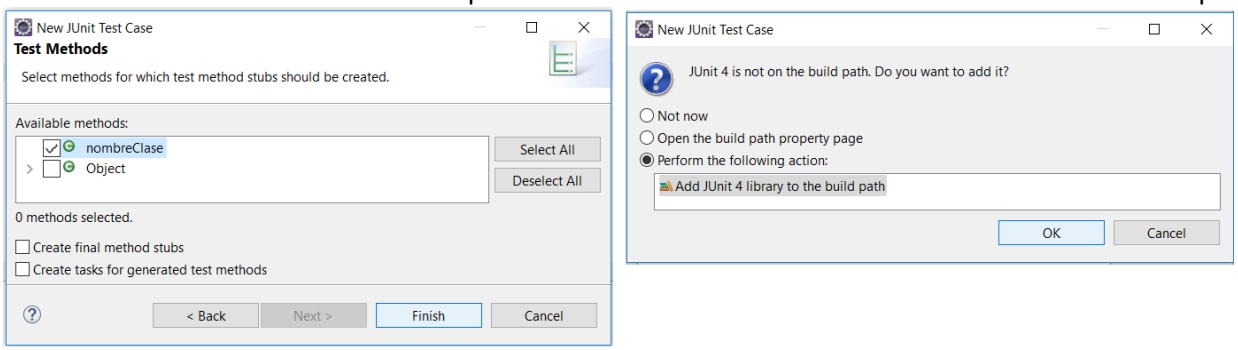


- Creamos el caso de prueba

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

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

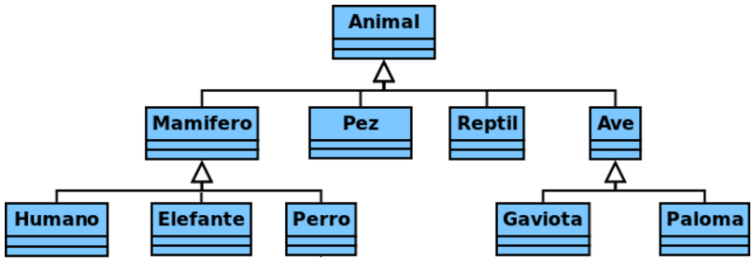
Objeto ⟶ elemento real con identidad propia

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

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

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
-

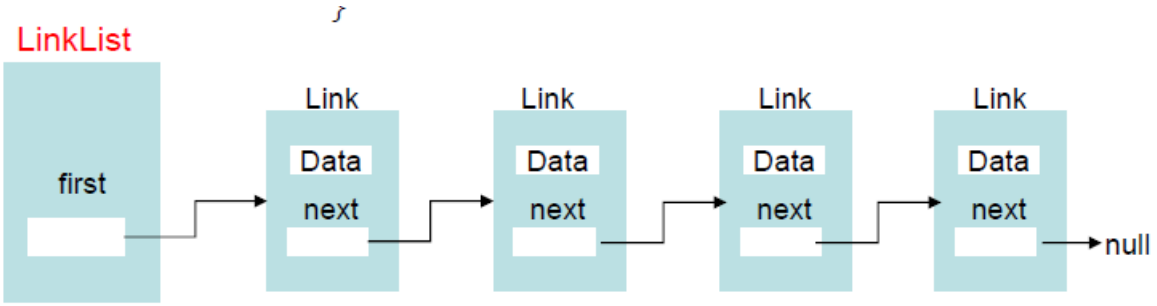
Las desventajas de las listas frente a otro tipo de estructura de datos son:

- 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.




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

```
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.

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

Interfaz:

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

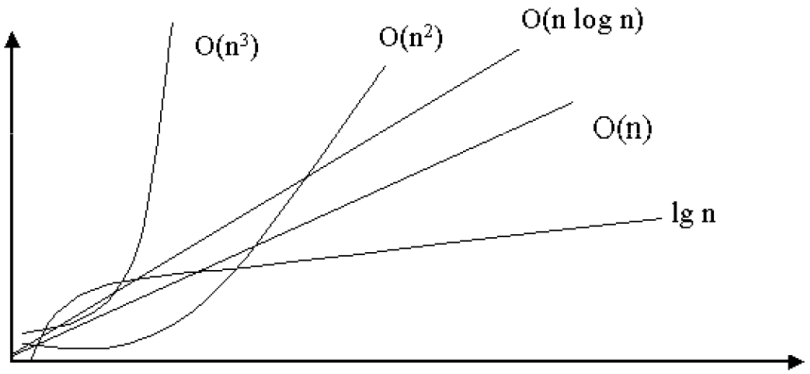
Esta estructura ha sido previamente implementada en Java de modo que se proporcionan un conjunto de métodos básicos para su facilitar el manejo:

Public Interface NombreInterfaz ⟶ permite definir un objeto tipo interfaz

Public NombreClase implements NombreInterfaz ⟶ permite reconocer a una clase como implementadora de una interfaz

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.