# Laboratorio A.E.D. Laboratorio 1

**Guillermo Román**
guillermo.roman@upm.es

**Julio García**
juliomanuel.garcia@upm.es

**Lars-Åke Fredlund**
larsake.fredlund@upm.es

**Tonghong Li**
tonghong.li@upm.es

**Manuel Carro**
manuel.carro@upm.es

**Sergio Paraíso**
sergio.paraiso@upm.es

**Marina Álvarez**
marina.alvarez@upm.es

**Juan José Moreno**
juanjose.moreno@upm.es

# Normas

- Fechas de entrega y penalización:

| | |
|---|---|
| Hasta el Martes 28 de Septiembre, 12:00 horas | 0 % |
| Hasta el Miércoles 29 de Septiembre, 12:00 horas | 20 % |
| Hasta el Jueves 30 de Septiembre, 12:00 horas | 40 % |
| Hasta el Viernes 01 de Octubre, 12:00 horas | 60 % |

  Después la puntuación máxima será 0
- Se comprobará plagio y se actuará sobre los detectados.
- Usad las horas de tutoría para preguntar sobre programación – son oportunidades excelentes para aprender.

# Entrega

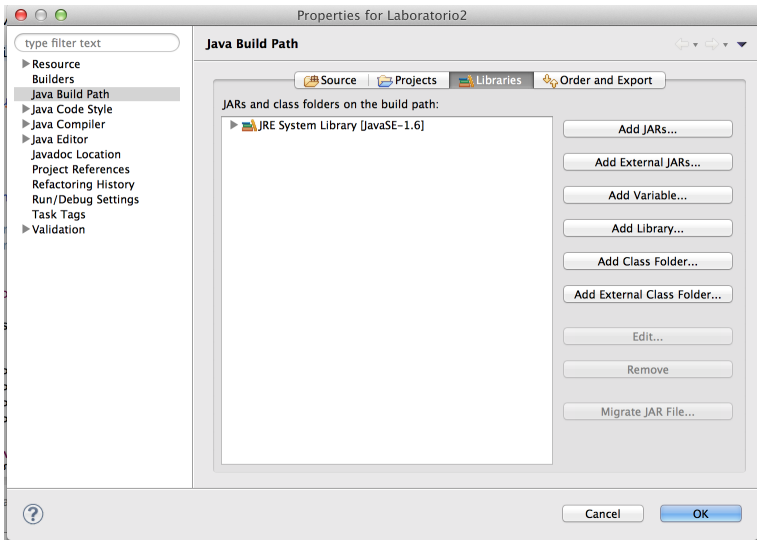
- Todos los ejercicios de laboratorio se deben entregar a través de
  `http://vps142.cesvima.upm.es`
- El fichero que hay que subir es `BancoFiel.java`.

# Configuración previa

- Arrancad Eclipse
- Si trabajáis en portátil, podéis utilizar cualquier versión reciente de Eclipse. Es suficiente con que instaléis la *Eclipse IDE for Java Developers*.
- Cambiad a "Java Perspective".
- Debéis tener instalado al menos Java JDK 8.
- Cread un proyecto Java llamado `aed`:
  - Seleccionad separación de directorios de fuentes y binarios.
  - **No debéis elegir la opción de crear el fichero** `module-info.java`
- Cread un *package* `aed.bancofiel` en el proyecto `aed`, dentro de `src`
- Aula Virtual → AED → Laboratorios → Laboratorio 1 → `Laboratorio1.zip`; descomprimidlo
- Contenido de Laboratorio1.zip:
  - `BancoFiel`, `ClienteBanco`, `ComparadorSaldo`, `Cuenta`, `CuentaNoExisteExc`, `CuentaNoVaciaExc`, `GestorBanco`, `InsuficienteSaldoExc`, `TesterLab1`

# Configuración previa

- Importad al paquete `aed.bancofiel` los fuentes que habéis descargado ( `BancoFiel`, `ClienteBanco`, `ComparadorSaldo`, `Cuenta`, `CuentaNoExisteExc`, `CuentaNoVaciaExc`, `GestorBanco`, `InsuficienteSaldoExc`, `TesterLab1` )
- Añadid al proyecto `aed` la librería `aedlib.jar` que tenéis en Moodle (en Laboratorios).

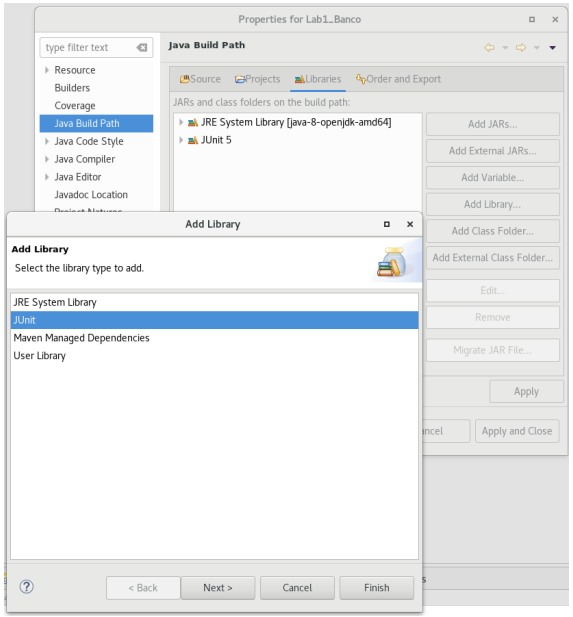



Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda
- Usad la opción “Add External JARs...”.
- Si vuestra instalacion distingue `ModulePath` y `ClassPath`, instalad en `ClassPath`

# Configuración previa

- Añadid al proyecto `aed` la librería `JUnit 5`




Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda;
- Usad la opción "Add Library..." → Seleccionad "Junit" → Seleccionad "JUnit 5"
- Si vuestra instalacion distingue `ModulePath` y `ClassPath`, instalad en `ClassPath`

- En la clase `TesterLab1` tenéis las pruebas, para ejecutarlas, abrid el fichero `TesterLab1`, pulsando el botón derecho sobre el editor, seleccionar "Run as..." → "JUnit Test"
- NOTA: Si al ejecutar, no aparece la vista "JUnit", podéis incluirla en "Window" → "Show View" → "Java" → "JUnit"

# Documentación de la librería aedlib.jar

- La documentación de la API de `aedlib.jar` esta disponible en `http://costa.ls.fi.upm.es/~entrega/aed/docs/aedlib/`
- Tambien se puede añadir la documentación de la librería a Eclipse (*no es obligatorio*):
  - En el "Package Explorer": "Referenced Libraries" → aedlib.jar y elige la opción "Properties". Se abre una ventana donde se puede elegir "Javadoc Location" y ahí se pone como "javadoc location path:"

  `http://costa.ls.fi.upm.es/~entrega/aed/docs/aedlib/`

  y presionar el buton "Apply and Close"

# Tarea: Implementar las interfaces `ClienteBanco` y `GestorBanco`

- Se pretende desarrollar un sistema para gestionar las cuentas bancarias de `BancoFiel`
- Para ello se han definido dos interfaces: `ClienteBanco` y `GestorBanco`
- La interfaz `ClienteBanco` dispone de métodos para:
  - Crear y eliminar una cuenta
  - Ingresar y retirar dinero de una cuenta, así como consultar su saldo
  - Hacer transferencias entre dos cuentas
  - Obtener una lista de cuentas de un cliente, y el saldo total de las cuentas de un cliente
- La interfaz `GestorBanco` dispone un método que devuelve las cuentas de todos los clientes ordenadas según un cierto criterio:
  - Para ello se podrían utilizar diferentes `Comparator<Cuenta>` (que no tendréis que implementar)

# Tarea 1: Implementar la interfaz ClienteBanco

- Se pide implementar la interfaz ClienteBanco:

```
public interface ClienteBanco {
 public String crearCuenta(String dni, int saldoInicial);
 public void borrarCuenta(String id)
    throws CuentaNoExisteExc, CuentaNoVaciaExc;
 public int ingresarDinero(String id, int cantidad)
    throws CuentaNoExisteExc;
 public int retirarDinero(String id, int cantidad)
    throws CuentaNoExisteExc, InsuficienteSaldoExc;
 public int consultarSaldo(String id)
    throws CuentaNoExisteExc;
 public void hacerTransferencia(String idFrom, String idTo, int cantidad)
    throws CuentaNoExisteExc, InsuficienteSaldoExc;
 public IndexedList<String> getIdCuentas(String dni);
 public int getSaldoCuentas(String dni);
}
```

- Documentación detallada en ClienteBanco.java o también en:
  http://costa.ls.fi.upm.es/entrega/aed/docs/bancofiel/

# Tarea 2: Implementar la interfaz GestorBanco

- Se pide implementar la interfaz GestorBanco:

```
public interface GestorBanco {
  // Ordena las cuentas segun el comparador cmp
  public IndexedList<Cuenta> getCuentasOrdenadas(Comparator<Cuenta> cmp);
}
```

- Documentación detallada en GestorBanco.java o también en:
  http://costa.ls.fi.upm.es/entrega/aed/docs/bancofiel/

# La clase BancoFiel

- La clase  debe implementar las interfaces `ClienteBanco` y `GestorBanco`

```
public class BancoFiel implements ClienteBanco, GestorBanco {
   ...
}
```

- Para implementarlo debéis usar el atributo `cuentas` de tipo `IndexedList<Cuenta>` que ya tenéis en la clase `Banco`
- Podéis usar lo métodos privados que consideréis necesarios
- La clase dispone de un único constructor sin parámetros para inicializar el atributo `cuentas`, que debe almacenar las cuentas bancarias creadas (ya incluido)
- Revisad los consejos antes de tomar decisiones

# Ejemplo

```
BancoFiel b = new BancoFiel ();

b.crearCuenta ("5248",10);     // [ Cuenta{"5248","5248/0",10}] -> "5248/0"
b.crearCuenta ("3489",20);     // [ Cuenta{"3489","3489/1",20},
                               //   Cuenta{"5248","5248/0",10}] -> "3489/1"
b.crearCuenta ("5248",30);     // [ Cuenta{"3489","3489/1",20},
                               //   Cuenta{"5248","5248/0",10},
                               //   Cuenta{"5248","5248/2",30}] -> "5248/2"

b.ingresarDinero("5248/2",15);  // [ Cuenta{"3489","3489/1",20},
                               //   Cuenta{"5248","5248/0",10},
                               //   Cuenta{"5248","5248/2",45}]

b.retirarDinero("5248/0",5);    // [ Cuenta{"3489","3489/1",20},
                               //   Cuenta{"5248","5248/0",5},
                               //   Cuenta{"5248","5248/2",45}]

b.ingresarDinero("65445",15);  //  CuentaNoExisteExc
b.retirarDinero("65445",15);  //  CuentaNoExisteExc
b.retirarDinero("5248/0",200); // InsuficienteSaldoExc
```

# Ejemplo (cont.)

```
                                // [ Cuenta{"3489","3489/1",20},
                                //   Cuenta{"5248","5248/0",5},
                                //   Cuenta{"5248","5248/2",45}]
b.consultarSaldo("5248/2");      //   -> 45

b.hacerTransferencia("5248/2","3489/1",22);
                                // [ Cuenta{"3489","3489/1",42},
                                //   Cuenta{"5248","5248/0",5},
                                //   Cuenta{"5248","5248/2",23}]

b.hacerTransferencia("5248/2","3489/1",100);  // InsuficienteSaldoExc
b.hacerTransferencia("31111","3489/1",10);   // CuentaNoExisteExc
b.hacerTransferencia("5248/2","55555",10);   // CuentaNoExisteExc

b.getIdCuentas("5248") -> ["5248/0","5248/2"]
b.getIdCuentas("1111") -> []

b.getSaldoCuentas("5248") -> 28
b.getSaldoCuentas("1212") -> 0
```

# Ejemplo (cont.)

```
                                    // [ Cuenta{"3489","3489/1",42},
                                    //   Cuenta{"5248","5248/0",5},
                                    //   Cuenta{"5248","5248/2",23}]


b.getCuentasOrdenadas(new ComparadorSaldo()); // NO CAMBIAR "cuentas"
                                    // -> [ Cuenta{"5248","5248/0",5}
                                    //      Cuenta{"5248","5248/2",23},
                                    //      Cuenta{"3489","3489/1",42}]
```

# Consejos

**Seguir estos consejos os permitirá conseguir mejores resultados!**

- Para construir objetos de tipo `Cuenta`, usad el constructor `Cuenta(DNI,saldo)`, que genera automáticamente el identificador de la cuenta
  - Para consultar su `id` podéis usar el método `getId()`
  - También dispone de los método `ingresar(x)` y `retirar(x)`
- Los métodos que realizan operaciones con una cuenta dado un identificador tienen todos una estructura muy similar:
  - Usad un método privado que devuelva la posición que ocupa la cuenta con identificador `id`, o $-1$ si la cuenta con ese `id` no existe:

```
private int buscarCuenta (String id)
```

- Mantened la lista de cuentas *ordenada* usando el identificador de cuenta (recordad que `String` implementa la interfaz `Comparable`)
- Si la estructura está ordenada por el identificador de las cuentas, `buscarCuenta` puede hacer una búsqueda binaria $\Rightarrow$ $O(log(n))$ en lugar de $O(n)$

# Consejos

**Seguir estos consejos os permitirá conseguir mejores resultados!**

- El método `getCuentas` debe devolver una **nueva lista** con todas las cuentas almacenadas en el banco, ordenadas por el criterio que establece `Comparator<Cuenta> cmp`
- **No debe** modificar el orden ni los elementos de la estructura interna que almacena las cuentas
- Recordad que la clase `ArrayIndexedList<E>` implementa la interfaz `IndexedList<E>`
- Para crear la nueva lista podéis recorrer los elementos almacenados en el `cuentas` e ir insertándolos en el orden adecuado en la lista resultado (usando para ello `cmp`)

# Evaluación

**Seguir estos consejos os permitirá conseguir mejores resultados!**

- Corrección
- Ausencia de código repetido con la misma funcionalidad (podéis usar métodos auxiliares para evitarlo)
- Concisión del código
- Legibilidad, incluida selección de nombres descriptivos para variables y métodos
- El código debe estar correctamente indentado y con comentarios *útiles* cuando lo veáis necesario
- Eficiencia:
  - Se valorará la complejidad computacional del código
  - Se valorará no iterar innecesariamente en los recorridos de las estructuras de datos

# Notas

- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar
- Debe pasar todos los test `TesterLab1` correctamente sin mensajes de error
- **Nota:** una ejecución sin mensajes de error y que pase todas las pruebas **no** significa que el método sea correcto (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se corrigen manualmente antes de dar la nota final