

# Anotaciones - 1er Parcial Laboratorio de Software

---

## Tipos de acceso

- `public` → Cualquiera puede acceder.
- `protected` → Solamente pueden acceder las subclases o las clases que están en el mismo paquete.
- `package` → Solamente pueden acceder las clases que están en el mismo paquete.
- `private` → Solamente puede acceder la propia clase.

## Herencia - Polimorfismo - Clases Abstractas - Interfaces

- Los constructores se encadenan.
- Las interfaces proveen un mecanismo de herencia de comportamiento y NO de implementación como las clases.
- Las variables de una interface son *public static* y *final* por defecto.
- Los métodos de una interface son *public* y *abstract* por defecto.
  - No tendría sentido que sean *final* porque si no no se podrían sobrescribir.
  - Desde Java 8 se pueden tener métodos *static* que se pueden llamar sin instanciar a la interfaz.
- Si una clase hereda de sus interfaces un atributo con igual nombre, puede haber conflicto y un problema en compilación → se soluciona usando su nombre completo.
- Desde Java 8 existen para las interfaces métodos de default que permiten agregar funcionalidad nueva a la interfaz asegurando compatibilidad binaria con el código escrito para versiones previas de la misma interface.

## Clases anidadas

- Las clases anidadas pueden declararse como *public*, *private*, *package* o *protected*.
- Una instancia de una clase anidada está siempre asociada con una instancia de la clase contenedora.
- Una clase interna (estática) no esta ligada a una instancia de la clase que la contiene, solo puede acceder a los miembros estáticos de la clase que la contiene y solo existe para la clase que la contiene.
- Una clase local es visible solamente dentro del bloque de código donde se definió (por eso no se le debe declarar ningún especificador de acceso).
  - Los objetos son upcasteados y pueden seguir siendo accedidos una vez que termina de ejecutarse el bloque de código donde se definió la clase local.
- Las clases anónimas son clases locales sin nombre.
  - Son simultáneamente declaradas e instanciadas en el punto en que se van a usar.
  - Solamente pueden extender a una clase o implementar a una sola interfaz.
  - Tanto las locales como las anónimas tiene acceso a los parámetros de los métodos (y las variables) que las contienen, estos son *efectivamente finales* por defecto para preservar su estado.
  - Para inicializar una variable en instanciación se usa un bloque de inicialización:

```
{  
    // este es el bloque de inicialización  
    //variables inicializadas...  
};
```

- Las clases anidadas se usan para proveer ocultamiento de detalles de implementación. Lo único que se obtiene es una referencia a través del upcasting a una clase base o interface publica.
- Las clases anidadas privadas que implementan interfaces son completamente *invisibles* e *inaccesibles* y de esta manera se *oculta la implementación*. Se evitan dependencias de tipos. Desde afuera de la clase se obtiene una referencia al tipo de la interface pública.
- Se usan para implementar iteradores:

```
private class StackIterator implements Iterator {  
  
    private int i = 0;  
  
    @Override  
    public boolean hasNext() {  
        return i < items.size();  
    }  
  
    @Override  
    public Object next() {  
        if (!hasNext()) {  
            throw new java.util.NoSuchElementException();  
        }  
        return items.get(i++);  
    }  
}
```

- Se usan para implementar adaptadores:

```
private class IteratorStringAdapter implements Iterator{  
  
    private int i = 0;  
  
    @Override  
    public boolean hasNext() {  
        return i < size();  
    }  
  
    @Override  
    public String next() {  
        if (!hasNext()) {  
            throw new java.util.NoSuchElementException();  
        }  
        return set.get(i++).toString();  
    }  
}
```

- Para nombrar al objeto de la clase contenedora es:

```
NombreDeLaClaseContenedora.this.objetoDeLaClaseContenedora
```

- Para instanciar una clase anidada desde una clase contenedora se hace:

```
ClaseContenedora.new ClaseAnidada();
```

- Una clase anidada puede instanciarse desde otra clase con:
  - *Nota: la clase anidada debe ser publica*

```
Interfaz_Clase_anidada i = Clase_que_contiene_a_la_anidada.new Clase_anidada();
```

---

Esto es un singleton y ademas se oculta la implementación de la clase anidada.

```
public class Host {  
  
    private static final java.util.Comparator<String> STRING_LENGTH_COMPARATOR =  
    new StrLenCmp();  
  
    public Comparator<String> stringLengthComparator() {  
        return STRING_LENGTH_COMPARATOR;  
    }  
  
    private static class StrLenCmp implements java.util.Comparator<String> {  
        public int compare(String s1, String s2) {  
            return s1.length() - s2.length();  
        }  
    }  
}
```

## Enumerativos

- Los valores de un tipo enumerativo son constantes públicas de clase *public static final*.
- El tipo enumerativo es una clase y sus valores son instancias de dicha clase.
- No se pueden extender.
- Los tipos enumerativos pueden tener métodos y propiedades.
  - Estos no son public static final por defecto como los valores enumerativos.

```
public enum Prefijo{  
    MM("m", 0.001),
```

```

    CM("c", 0.01),
    DM("d", 0.1),
    M("m", 1),
    DAM("dam", 10),
    HM("hm", 100),
    KM("km", 1000);

    private String abreviatura; // no es final, para ello debería agregar private
    final String abreviatura;
    private double factor;

    Prefijo(String abreviatura, double factor){ // el constructor es privado por
    defecto, lo usa solo el compilador. Se pasan los valores declarados para las
    propiedades.
        this.abreviatura = abreviatura;
        this.factor = factor;
    }

    public String getAbreviatura(){ // recupero el valor de la abreviatura
        return abreviatura;
    }

    public double getFactor(){ // recupero el valor del factor
        return factor;
    }
}

```

- Se pueden usar en un switch:

```

public class Main {
    public static void main(String[] args) {
        Prefijo prefijo = Prefijo.KM;
        switch (prefijo){
            case MM:
                System.out.println("Milimetro");
                break;
            case CM:
                System.out.println("Centimetro");
                break;
            case DM:
                System.out.println("Decimetro");
                break;
            case M:
                System.out.println("Metro");
                break;
            case DAM:
                System.out.println("Decametro");
                break;
            case HM:
                System.out.println("Hectometro");
                break;
            case KM:

```

```

        System.out.println("Kilometro");
        break;
    default:
        throw new IllegalStateException("Unexpected value: " + prefijo);
    }
}
}

```

- values() → Devuelve un array con los valores del enumerativo en el orden en que fueron declarados.
- Se le puede agregar comportamiento distinto a cada valor del enumerativo:

```

public enum Operations {
    SUM {
        @Override
        public double apply(double x, double y) {
            return x + y;
        }
    },
    SUBSTRACT {
        @Override
        public double apply(double x, double y) {
            return x - y;
        }
    },
    MULTIPLY {
        @Override
        public double apply(double x, double y) {
            return x * y;
        }
    },
    DIVIDE {
        @Override
        public double apply(double x, double y) {
            return x / y;
        }
    };

    public abstract double apply(double x, double y);
}

```

- Pueden implementar interfaces.

---

Hacer lo siguiente es equivalente a hacer un singleton:

```

public enum FitoPaez {
    INSTANCE(new Piano());
}

```

```
private final piano;  
  
FitoPaez(Piano piano) {  
    this.piano = piano;  
}
```

y ademas se puede hacer de la siguiente manera:

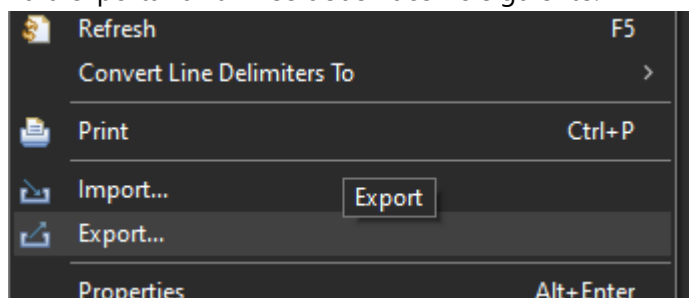
```
public enum FitoPaez {  
    INSTANCE;  
  
    private final Piano piano = new Piano();  
}
```

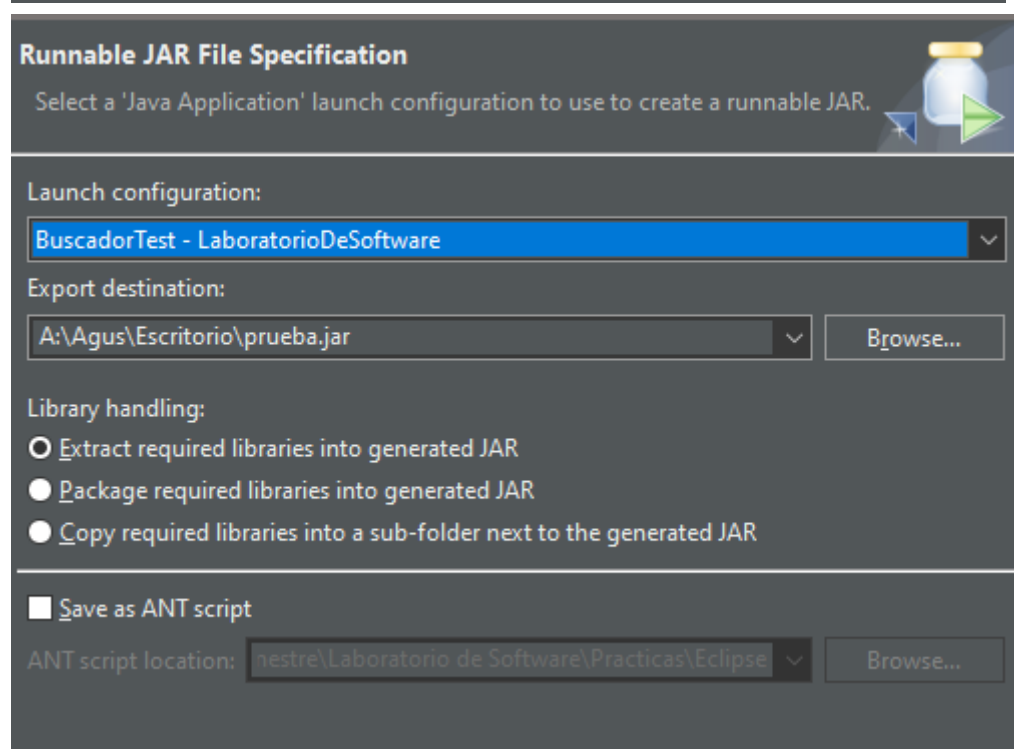
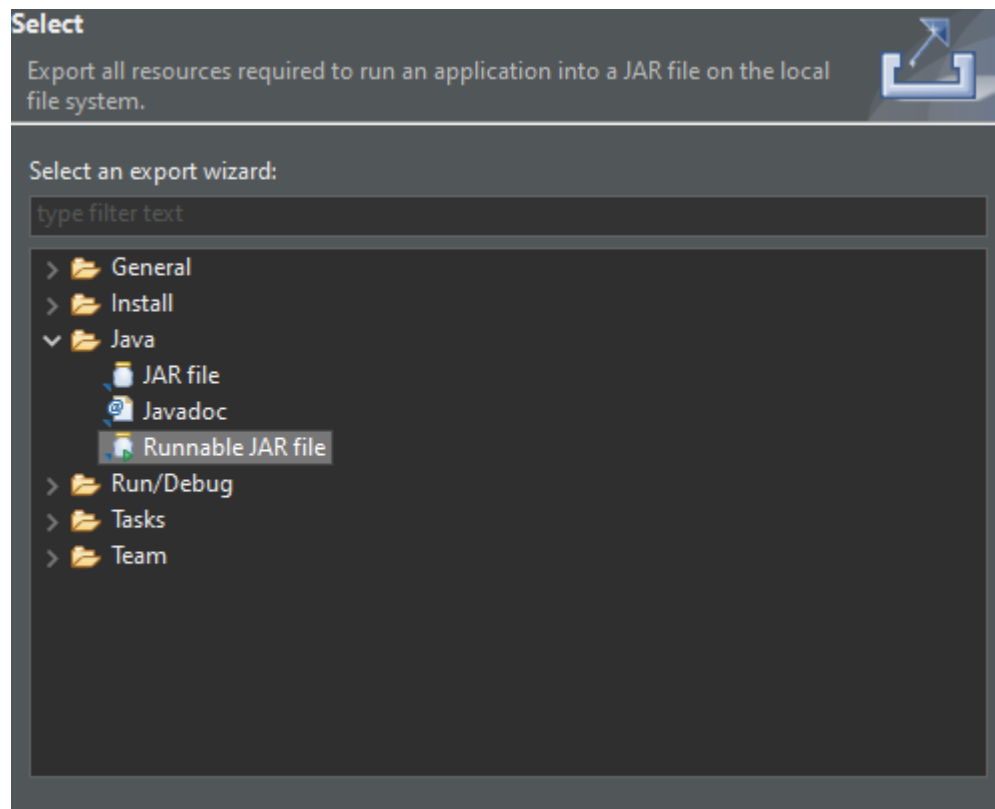
o asi:

```
public enum FitoPaez {  
    INSTANCE;  
  
    private final piano;  
  
    FitoPaez() {  
        this.piano = new Piano();  
    }  
}
```

## Otras anotaciones

- Una clase *static* tiene a todos sus miembros como estáticos, eso quiere decir que pueden utilizarse sin instanciar dicha clase.
  - Un método *static* puede ser llamado sin instanciar a la clase.
  - Una variable *static* puede usarse sin instanciar a la clase.
  - Para importar a un miembro estático hay que hacer *import static*
- Una clase *final* no se puede extender, no puede reemplazarse y tiene a todos sus métodos como final.
  - Un método *final* no puede ser sobrescrito.
  - Una variable *final* no puede ser sobrescrita (es como una constante).
- Para exportar un JAR se debe hacer lo siguiente:





```
agusr@DESKTOP-E5THSHN MINGW64 /a/Agus/Escritorio
$ java -jar prueba.jar |
```