Laboratorio 2 – Programación Orientada a Objetos, *equals, Genéricos*

Los objetivos de la práctica son trabajar los elementos siguientes:

- herencia
- definición de equals
- genéricos
- comodines
- interfaz Comparable<E>

La práctica está compuesta por los ejercicios descritos a continuación.

Ejercicio 1: Jerarquía de clases y equals

El objetivo de este primer apartado es, por un lado, analizar el código que genera IntelliJ para el método equals y las diferentes propiedades que equals ha de cumplir (nos centraremos en simetría y transitividad).

Apartado 1a

Las clases de este apartado estarán en el paquete **apartado1a** (recordad que el paquete estará en el subdirectorio `apartado1a` del directorio `src`). En el código de producción partiremos de estas dos clases simples:

```
package apartado1a;

public class Person {

    protected final String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

Y de una subclase `Employee` que añade el salario al nombre:

```java
package base;

public class Employee extends Person {

    private final int salary;

    public Employee(String name, int salary) {
        super(name);
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "salary=" + salary +
                ", name='" + name + '\'' +
                '}';
    }
}
```

En primer lugar, generaremos en ambas clases el método `equals` (también nos generará el método `hashCode` que ya veremos para qué existe más adelante en la asignatura y que en este laboratorio no analizaremos).

Para hacerlo

- seleccionaremos el menú: `Code > Generate ...` (o usaremos el atajo de teclado que en el Mac es Cmd-N y creo que Alt+Ins en Win/Linux)
- seleccionaremos `equals() and hashCode()`
- como template usaremos el que indica `... (Java 7+)`
    - no aceptaremos subclases como parámetro
    - no usaremos getters
- seleccionaremos `name` como atributo a usar en `equals()`
- seleccionaremos `name` como atributo a usar en `hashCode()`
- no seleccionaremos ningún atributo para que sea no nulo.

Después de esta selección, el método `equals` debería quedar como:

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(name, person.name);
}
```

Para entender dicha implementación será necesario estudiar la documentación de los métodos:

- método `getClass()` definido en la clase `Object`
- método estático `equals` de la clase `Objects`

En el informe deberéis explicar el porqué de cada una de las líneas de este equals.

En segundo lugar, realizaréis los mismos pasos para generar el método `equals` de la subclase `Employee`, obviamente seleccionando `salary` como atributo.

En el informe deberéis explicar el porqué de cada una de las líneas de este `equals`, obviamente centrándoos en las diferencias con el `equals` de la clase `Person`.

Para finalizar este apartado, realizaremos una clase de prueba, que también estará en el paquete apartado1a (pero en el subdirectorio test, que como siempre, habréis marcado como directorio de pruebas) y tendréis estos métodos de prueba:

```
package apartado1a;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class Properties {

    @Test
    void reflexivity() {
        Person p = new Person("John Doe");
        assertEquals(p, p);
    }

    @Test
    void symmetry1() {
        Person p1 = new Person("John Doe");
        Person p2 = new Person("John Doe");
        assertEquals(p1, p2);
        assertEquals(p2, p1);
    }

    @Test
    void symmetry2() {
        Person p1 = new Person("John Doe");
        Person p2 = new Person("Jane Doe");
        assertNotEquals(p1, p2);
        assertNotEquals(p2, p1);
    }

    @Test
    void transitivity() {
        Person p1 = new Person("John Doe");
```

```
        Person p2 = new Person("John Doe");
        Person p3 = new Person("John Doe");
        assertEquals(p1, p2);
        assertEquals(p2, p3);
        assertEquals(p1, p3);
    }

    @Test
    void nullParameter() {
        Person p = new Person("John Doe");
        assertNotEquals(p, null);
    }

    @Test
    void notInteroperable() {
        Person p = new Person("John Doe");
        Employee e = new Employee("John Doe", 25_000);
        assertNotEquals(p, e);
        assertNotEquals(e, p);
    }
}
```

Como comentario, `assertEquals(a, b)` comprueba que `a.equals(b)` devuelve true.

En el informe deberéis explicar qué hace que pase la prueba, marcada en negrita, notInteroperable.

Apartado 1b

En este apartado modificaremos la generación de `equals`, de manera que **sí aceptaremos que se pueda comparar una instancia una clase con la de una de sus subclases**. Para ello:

- crearemos un paquete con nombre **apartado1b**
- en él definiremos de nuevo las clases `Person` y `Employee`
- generaremos los métodos `equals`, pero ahora, seleccionaremos la opción de `"Accept subclases as parameter to equals() method"`

En el informe deberéis explicar también cómo funciona este `equals` y por qué es diferente del que tenemos en el apartado anterior.

Si copiáis la clase de pruebas, en el paquete `apartado1b` (en un subdirectorio de `test`), y los ejecutáis veréis que todo funciona excepto el `notInteroperable` (ya que hemos construido un equals que permita comparar una instancia de Person con una de Employee).

Modificad la prueba, obviamente la colocaréis en el paquete apartado1b y como subdirectorio de test, de manera que se compruebe la semi-interoperabilidad: se puede comparar un Person con un Employee, pero la comparación entre Employee y Person siempre retorna falso.

Fijaos en que al hacer interoperable `Person` y `Employee`, la prueba que hemos definido demuestra que ¡**hemos dejado de cumplir la simetría!**

Apartado1c

En este aparado, volveréis a copiar las clases en un nuevo paquete (`apartado1c`) y:

- modificaréis el método `equals` de la clase `Employee` del aparatado anterior para que se cumpla la simetría; es decir, que la siguiente prueba funcione:

```
@Test
void interoperable() {
    apartado1a.Person p = new Person("John Doe");
    apartado1a.Employee e = new Employee("John Doe", 25_000);
    assertEquals(p, e);
    assertEquals(e, p);
}
```

- crearéis una prueba, ahora en el paquere apartado1c, que muestre que, al permitir la simetría entre `Person` y `Employee`, **¡perdéis la transitividad!**

En el informe deberéis explicar cómo funciona y cómo habéis definido el método equals de la clase Employee (y, si es el caso, las cosas que habéis intentado y que han fallado).

Podéis, si queréis aprender más cosas, analizar la/las otras plantillas de generación de equals y analizar el código generado.

Ejercicio 2: Iteradores e interfaces

En este apartado diseñaremos métodos estáticos para trabajar con los elementos de una lista genérica usando iteradores.

En primer lugar, implementaremos un método estático (en una clase del paquete `apartado2`), para contar el número de elementos de la lista (de hecho, de la estructura) que cumplen una determinada propiedad (que también querremos que sea parametrizable).

¿Cómo parametrizamos la propiedad, de manera que podamos invocar a la función si queremos saber el número de elementos pares de una lista de enteros, o el número de palabras que empiezan por la letra 'a' en una lista de Strings?

Usando una interfaz que contenga el método de comprobación, es decir:

```
package apartado2;

public interface Check<E> {
    boolean test(E e);
}
```

Si, por ejemplo, deseamos saber si un número es par, podemos implementar:

```
public class CheckEven implements Check<Integer> {
    @Override
    public boolean test(Integer n) {
        return n % 2 == 0;
    }
}
```

NOTA: Desde Java 5 las clases envolventes (wrapper classes) que sirven, entre otras cosas, para poder usar tipos primitivos en contextos que requieran de tipos referencia, interoperan de forma "mágica" con los tipos primitivos y no es necesario transformarlos manualmente haciendo, en este caso:

```
return n.intValue() % 2 == 0;
```

Podéis leer sobre ello en The Java Tutorials: Autoboxing and Unboxing.

Lo que se pide es que implementéis el método:

```
public static <E> int countIf(
        Iterator<E> it,
        Check<E> test)
    { ¿? }
```

que calcula el número de elementos obtenibles desde el iterador cumple la propiedad.

Unos posibles tests para este método serían:

```
package apartado2;

import org.junit.jupiter.api.Test;

import java.util.List;
```

```
import static org.junit.jupiter.api.Assertions.assertEquals;

class ChecksTest {

    @Test
    void countEvenLocalClass() {
        class CheckEven implements Check<Integer> {
            @Override
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        }
        var it = List.of(1, 2, 3, 4, 5, 6, 7).iterator();
        assertEquals(3, Checks.countIf(it, new CheckEven()));
    }

    @Test
    void countAnonymousClass() {
        var it = List.of(1, 2, 3, 4, 5, 6, 7).iterator();
        assertEquals(3, Checks.countIf(it, new Check<Integer>() {
            @Override
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        }));
    }

    @Test
    void countLambda() {
        var it = List.of(1, 2, 3, 4, 5, 6, 7).iterator();
        assertEquals(3, Checks.countIf(it, n -> n % 2 == 0));
    }
}
```

En ellos podemos observar los siguientes elementos de sintaxis:

- Java, en algunos casos, es capaz de deducir el tipo que tienen las variables locales con lo que no es necesario definir el tipo (de ahí el uso de var)

Esto lo podéis encontrar en Local Variable Type Inference y una guía de cuando es apropiado y cuando no Local Variable Type Inference: Style Guidelines

- es posible definir una clase dentro de un método que, obviamente, solamente se podrá usar en dicho método (estas clases se denominan locales)
- es posible definir una clase que implemente una interfaz (o que sea subclase de otra) y, a la vez, crear un objeto de ésta, sin darle un nombre a la clase (estas clases se denominan anónimas)
- si la interfaz solamente tiene un método, a partir de Java 8, se puede usar en vez de una clase que implemente la interfaz, una expresión lambda,

que contiene solamente los elementos esenciales de lo que se quiere expresar (en este caso que, dado un n, éste ha de ser par)

Todo esto (y mucho más) lo podéis encontrar en: The Java Tutorials: Classes and Objects.

En este apartado deberéis implementar el método countIf y explicar su diseño en el informe.

Ejercicio 3: Comparadores y comodines

En este apartado, partiremos de las clases del `apartado1a` (es decir, las copiaremos en el paquete apartado3).

Haremos que la clase `Person` implemente la interfaz `Comparable<Person>` de manera que se comparen los nombres, pero sin tener en cuenta si están en mayúscula y minúscula.

https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Comparable.html

Haced una clase de tests para comprobar que la personas se comparan correctamente bajo ese criterio. Probad también comparaciones "mixtas" entre personas y empleados.

Cread una clase con un método tal que:

- dado un iterador y un elemento de referencia, cuente el número de elementos que se acceden a través del iterador que sean estrictamente menores que el elemento dado.

Definid la signatura del método de manera que el siguiente código compile sin problemas:

```
package apartado3;

import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class ComparisonsTest {

    @Test
    void mixed1() {
        List<Employee> employees = List.of(
```

```
                new Employee("John Doe", 150_000),
                new Employee("Peter Parker", 12_000),
                new Employee("Bruce Wayne", 999_999),
                new Employee("Clark Kent", 45_000)
        );
        assertEquals(3,
                Comparisons.countLower(
                        employees.iterator(),
                        new Person("Noname")));
    }


    @Test
    void mixed2() {
        List<Person> employees = List.of(
                new Employee("John Doe", 150_000),
                new Employee("Peter Parker", 12_000),
                new Employee("Bruce Wayne", 999_999),
                new Employee("Clark Kent", 45_000)
        );
        assertEquals(3,
                Comparisons.countLower(
                        employees.iterator(),
                        new Employee("Noname", 0)));
    }

    @Test
    void employees() {
        List<Employee> employees = List.of(
                new Employee("John Doe", 150_000),
                new Employee("Peter Parker", 12_000),
                new Employee("Bruce Wayne", 999_999),
                new Employee("Clark Kent", 45_000)
        );
        assertEquals(3,
                Comparisons.countLower(
                        employees.iterator(),
                        new Employee("Noname", 0)));
    }
}
```

Explicad porqué la generalización usando comodines que habéis hecho funciona en cada uno de los casos.

Como siempre la práctica está abierta a ampliaciones, la más evidente de las cuales es:

- ¿qué pasa si hago que la subclase Employee también implemente Comparable, en este caso, de Employee?
- ¿o que defina de manera diferente el compareTo que hereda de Person?

Entrega

El resultado obtenido debe ser entregado por medio de un único fichero comprimido por cada grupo con el nombre "Lab2_NombreIntegrantes". Dicho fichero contendrá el proyecto Intellij (con las clases en los paquetes indicados por el enunciado) y el PDF con el informe.

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- Solución a los ejercicios planteados.
- Calidad y limpieza del código.
- Razonamientos y justificaciones.