UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias


# Introducción a las Ciencias de la Computación

*Práctica 13: Interfaces.*

Profesora: Amparo López Gaona
Ayudante: Ramsés Antonio López Soto
Ayudantes de Laboratorio:
Adrián Aguilera Moreno
Kevin Jair Torres Valencia

# Objetivos

A continuación se presentan los objetivos de esta práctica de laboratorio:

1. Que el alumno se ejercite en el manejo de excepciones como un mecanismo para escribir programas robustos. Estas excepciones pueden ser de las proporcionadas por Java o bien desarrolladas para un programa particular. El manejo de las excepciones incluye su lanzamiento, atrapado y recuperación.

2. Que el alumno ejercite sus conocimientos acerca de la utilidad y programación de interfaces durante el desarrollo de sus programas. Las prácticas incluyen la implementación de interfaces proporcionadas por Java.

3. Que el alumno ponga en práctica sus habilidades con el manejo de arreglos de objetos y clases genéricas.

# Introducción

Una excepción es un evento que ocurre en cualquier momento de ejecución de un programa y que modifica el flujo normal de éste. Las excepciones son objetos de la clase `Exception` que almacenan información que se regresa en caso de que ocurra una anormalidad. Todos los métodos que hayan llamado al método en que se produce la excepción pueden darse por enterados y alguno de ellos o todos tomar las medidas apropiadas.

La clase `Exception`, que se encuentra en el paquete `java.lang`, es la raız de una jerarquía de clases para los errores más comunes. En esta jerarquıa se tiene la clase `RuntimeException` de la que se derivan varias clases de uso frecuente, por ejemplo, `NullPointerException`. Una excepción se activa (dispara) para indicar que ocurrió una falla durante la ejecución de un método. La excepción se propaga hasta encontrar un método en el cual se indica (atrapa) qué se debe hacer en circunstancias anómalas.

Para tratar con las excepciones es necesario escribir un manejador de excepciones utilizando la instrucción try que tiene la siguiente sintaxis:

```
try {
   instrucciones
}

catch (Excepción e) {
   instrucciones
}
...
catch (Excepción e) {
   instrucciones
}
finally {
   instrucciones
}
```

Cada cláusula de la instrucción `try` es un bloque que incluye las instrucciones que pueden disparar la(s) excepción(es). Las cláusulas `catch` tienen como parámetro un objeto de alguna clase de excepción. En una instrucción try puede haber varias cláusulas catch; en el bloque de cada una se coloca el código que implementa la acción a realizar en caso de que ocurra una excepción del tipo de su parámetro. Por último, la cláusula opcional `finally` contiene el código para establecer un estado adecuado para continuar la ejecución del método donde aparece la instrucción `try`.

En ocasiones puede suceder que las clases de excepciones existentes no describan la naturaleza del error que se tiene; en ese caso y para dar mayor claridad a los programas es posible crear excepciones propias, esto se logra extendiendo la clase `Exception`. Todas las nuevas clases requieren que se proporcione una cadena de diagnóstico al constructor.

Por otro lado una `interface` puede verse como el caso extremo de una clase abstracta, pues en la `interface` todos los métodos son abstractos, es decir, sólo se define el comportamiento de las clases que las implementan, mediante los métodos sin especificar la implementación de ninguno.

La definición sintáctica de una interfaz se muestra a continuación:

visibilidad **interface** nombre_de_la_interfaz {
  declaración de constantes y métodos
}

La `interface` puede definirse pública o sin modificador de acceso, y tiene el mismo significado que para las clases. En el cuerpo de la interfaz aparecen constantes públicas y firmas de métodos públicos. Debido a que no tiene sentido utilizar cualquier otro calificador de visibilidad para los atributos y métodos de una `interface`, pueden omitirse y se asume que todo es público. Lo común es que las interfaces tengan un nombre con el sufijo `able`.

Para usar una `interface` se debe especificar en una clase que ésta la implementa mediante la palabra reservada `implements`. La clase que implementa la `interface` debe implementar al menos todos los métodos de ella, es decir, puede contener otros métodos. Resulta indispensable que la firma de cada método definido en la interfaz coincida con la firma del método en la clase que los implementa. De otra forma se tiene sobrecarga de métodos y el compilador envıa un mensaje de error indicando que falta de implementarse un método.

Así mismo, en `Java`, los Genéricos permiten definir clases, interfaces y métodos con tipos parametrizados. Esto proporciona flexibilidad y seguridad de tipo en tiempo de compilación, evitando la necesidad de conversiones explícitas y reduciendo la probabilidad de errores en tiempo de ejecución.

El "operador diamante" es una característica introducida en `Java 7` que simplifica la creación de instancias de clases genéricas al permitir omitir el tipo genérico explícito en el lado derecho

de una declaración de variable. Se representa con el símbolo < >.

Cuando se usa el operador diamante, el compilador puede inferir automáticamente el tipo genérico a partir del contexto, evitando la redundancia y haciendo el código más conciso y fácil de leer. A continuación se muestra un ejemplo:

```
public class MyGenericClass<T> {
    /* Aqui los constructores, y metodos que utilicen el parametro
    generico 'T'*/
}
```

# Desarrollo

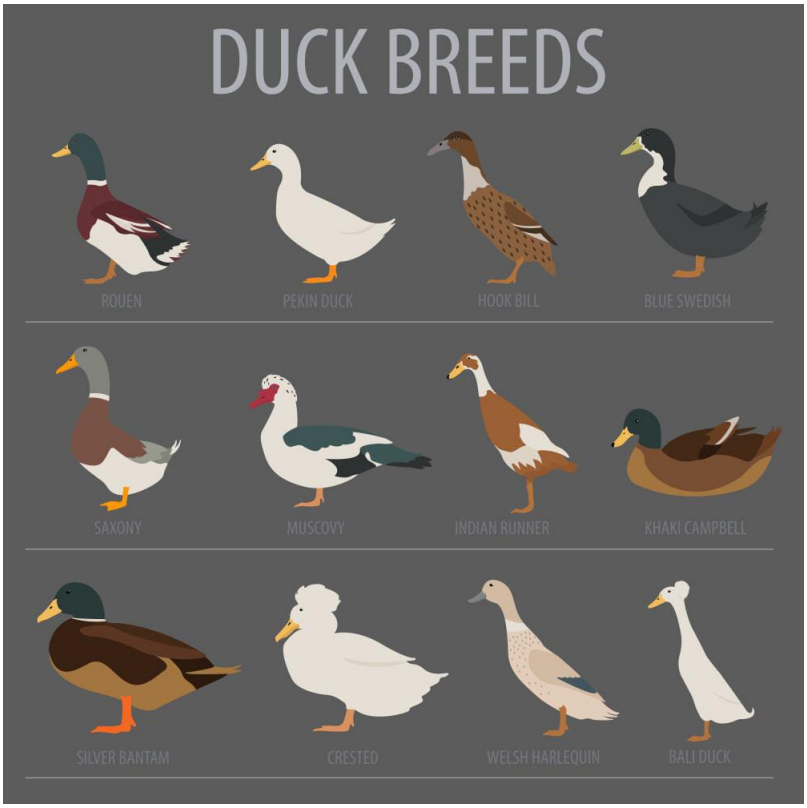
**1.** Implementa las clases que heredan de `Anatida` considerando los siguientes requisitos:

1. Guarda tus clases e implementaciones en la dirección: `src/fciencias/Anatidas`. Usa atributos con acceso `protected`.

2. Implementa la interface `Comparator` del paquete `java.util` en la clase `Pato.java` y `Oca.java`

   Así, debes comparar respecto a:

   (a) Para las clases que heredan de `Pato` debes comparar respecto al peso de cada `Pato`.




   Deben garantizar que el peso dependa del sexo del ejemplar. Como requisito deben verificar que no existan pesos mayores a 10 kg y menores que 0.45kg, para esto se debe crear una excepción personalizada, son libres de dar el nombre y el paquete que deseen.

(b) Para las clases que heredan de `Oca` deben comparar respecto al número de huevos que producen anualmente. Es decir, una oca que tiene una postura menor debe tener menor prioridad en un orden.


El rango de postura debe encontrarse entre 4 y 70 huevos al año, para esto crea una excepción personalizada que maneje datos fuera del rango de manera robusta.

3. Las excepciones personalizadas deben incluirse en un mismo paquete dentro de `src/fciencias/`.

**2.** Para este punto debes utilizar la clase `src/fciencias/Punto/Punto.java` e importar el respectivo paquete. Realiza las siguientes modificaciones a esta clase:

1. Implementa la interface `Comparator` del paquete `java.util` en la clase `Punto.java` y compara respecto la distancia que tiene un punto respecto al origen.

2. Para esta práctica solo debes admitir puntos en el primer cuadrante. Para esto implementa una excepción personalizada. Incluye la excepción en el mismo paquete que las anteriores.

**3.** Crea una clase `ordenamientoGenerico.java` genérica que reciba un arreglo de objetos de alguno de los tipos: `Anatida` y `Punto`.

El único método público, distinto a los *posibles constructores*, debe ser `ordenar<T> (T arreglo)` y que regrese un arreglo con los elementos ordenados respecto al método `comparable` implementado en su respectiva clase.

Incluye algunas pruebas en la clase `Main.java`, imprime los datos sin ordenar y ordenados con su respectivo método.

**4.** Por último, genera una clase de prueba que realice el ordenamiento de puntos, libros, discos, y películas por separado. Debes visualizar los elementos sin ordenas y después ordenados.

**Nota:** Crea los puntos por copia y luego modifica sus respectivos atributos. Hint. Usa el operador punto.

# Formato de Entrega

1. Las prácticas se entregarán en parejas.
2. Cada práctica (sus archivos y directorios) deberá estar contenida en un directorio llamado `EquipoX_pY`, donde:
   (a) X es el número de equipo correspondiente.
   (b) Y es el número de la práctica.

   Por ejemplo: `Equipo10_p13`
3. NO incluir los archivos .class dentro de la carpeta.
4. Los archivos de código fuente deben estar documentados.
5. Se pueden discutir y resolver dudas entre los integrantes del grupo. Pero cualquier práctica plagiada total o parcialmente será penalizada con cero para los involucrados.
6. La práctica se debe subir al Github Classroom correspondiente.
7. La entrega en classroom debe contener el link HTTPS y SSH de su repositorio y es lo único que se debe entregar.
8. El horario y día de entrega se acordará en la clase de laboratorio y no deberá sobrepasar 2 clases de laboratorio.