Licenciatura en Sistemas - Orientación a Objetos I – 2018

Prof. Titular:	Lic. María Alejandra Vranić	alejandravranic@gmail.com
Prof. Ayudantes:	Lic. Romina Mansilla	romina.e.mansilla@gmail.com
	Lic. Leandro Ríos	leandro.rios.unla@gmail.com
	Lic Gustavo Siciliano	gussiciliano@gmail.com

Métodos estáticos - Excepciones - try catch - Escenarios

Métodos estáticos:

Utilizamos métodos estáticos cuando no es necesario crear objetos de esa clase, ejemplo para llamar distintos métodos reutilizables en todos los proyectos.

```
package modelo;

import java.util.Calendar;
import java.util.GregorianCalendar;

public  class Funciones {

	public static int traerAnio(GregorianCalendar f){
		return f.get(Calendar.YEAR);   }

	public static String traerFechaCorta(GregorianCalendar f){
		String fechaCorta=""; // definir fecha corta dd/mm/aaaa de
longitud=10
		return fechaCorta;
	}

}


package test;

import java.util.GregorianCalendar;

import modelo.Funciones;

public class TestFunciones {

	public static void main(String[] args) {
		GregorianCalendar fecha=new GregorianCalendar();//ahora
		//no se crea instancia de Funciones por ser método estático
		System.out.println(Funciones.traerAnio(fecha));  }

}
```

Crear los siguientes métodos estáticos en la clase Funciones, realizar los test de todos los métodos según el análisis de los posibles escenarios.

Crear la clase Funciones en modelo con los siguientes métodos estáticos
+ esBisiesto (int anio) : boolean
+ traerAnio (GregorianCalendar fecha) : int
+ traerMes (GregorianCalendar fecha) : int //devuelve int entre 1 y 12
+ traerDia (GregorianCalendar fecha) : int

Serán bisiestos los años divisibles por 4, exceptuando los que son divisibles por 100 y **no** divisibles por 400. Ejemplos: son bisiestos 1996, 2004, 2000, 1600; No son bisiestos 1700, 1800, 1900, 2100

+ esFechaValida (int anio, int mes, int dia) : boolean
El mes es entero entre 1 y 12. En el caso de ser bisiesto es válido el día 29/02
+ traerFecha (int anio, int mes, int dia) : GregorianCalendar
Mes de 1 a 12
+ traerFecha (String fecha) : GregorianCalendar
Parámetro de la forma "dd/mm/aaaa"
+ traerFechaCorta (GregorianCalendar fecha) : String
Retorna "dd/mm/aaaa"
+ traerFechaCortaHora (GregorianCalendar fecha) : String
Retorna "dd/mm/aaaa hh:mm:ss"
+ traerFechaProximo (GregorianCalendar fecha, int cantDias) :GregorianCalendar
Ejemplo: traerFechaProximo (new GregorianCalendar(2016,7,21) , 7) ----> new GregorianCalendar(2016,7,28)
+ esDiaHabil (GregorianCalendar fecha) : boolean
Consideramos hábil de lunes a viernes.
+ traerDiaDeLaSemana (GregorianCalendar fecha) : String
+ traerMesEnLetras (GregorianCalendar fecha) : String
+ traerFechaLarga (GregorianCalendar fecha) : String
Ejemplo: "Sábado 20 de Agosto del 2016"
+ sonFechasIguales(GregorianCalendar fecha , GregorianCalendar fecha1) : boolean
+ sonFechasHorasIguales(GregorianCalendar fecha , GregorianCalendar fecha1) : boolean
+ traerCantDiasDeUnMes (int anio, int mes) : int
+ aproximar2Decimal (double valor) : double
Si el tercer decimal es mayor o igual 5, suma 1 al segundo decimal
+ esNumero(char c) : boolean
+ esLetra(char c) : boolean
+ esCadenaNros(String cadena) : boolean
+ esCandenaLetras(String cadena) : boolean

Trabajo Práctico

modelo:
Implementar la clase Numero con el atributo int n.

+ esPrimo ():boolean (1)

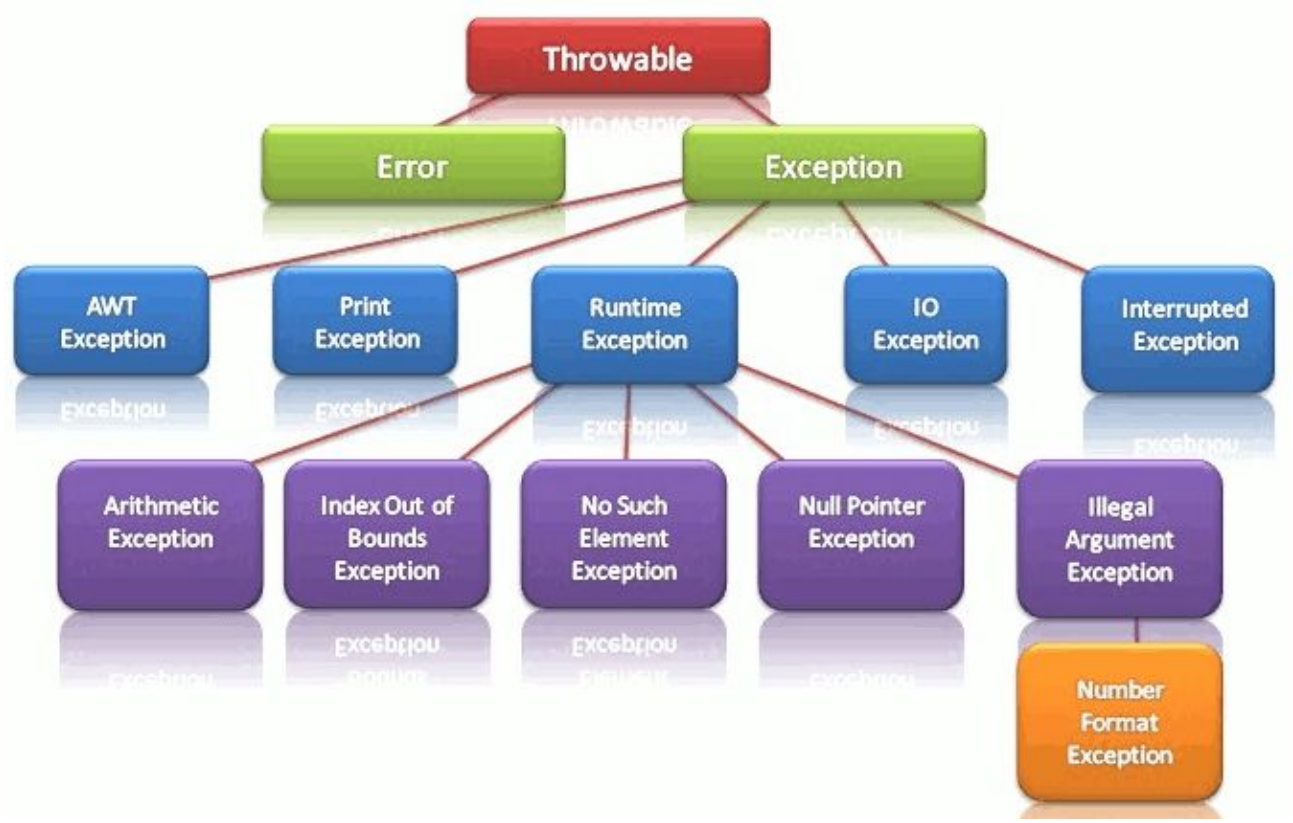
> ✔ *Un número natural p se dice que es primo si p>1 y si sus únicos divisores positivos son 1 y p*

+ esPrimoMellizo ():boolean (2)

Dos números primos se dice que son mellizos si su diferencia es 2, por ejemplo 17 y 19 son primos mellizos

test
test1.java: Generar los 2 escenarios para (1)
test2.java: Generar los 2 escenarios para (2)
test3.java: Utilizando for e imprimir los números primos hasta n
test4.java: Utilizando un while imprimir los números primos mellizos hasta n

Excepciones



En términos generales, una excepción es un evento que ocurre durante la ejecución de un programa y que provoca la imposibilidad de continuar con el flujo de ejecución normal del mismo. Si necesitamos acceder a un archivo y no existe, si queremos dividir por cero, o si intentamos acceder a una posición de un arreglo más allá de sus límites, todos son ejemplos de excepciones, de cosas que no deberían estar ocurriendo y que no están previstas en el programa.

Esta interrupción debe representarse de algún modo en el lenguaje de programación, y en general se representa como un salto en la ejecución de nuestro programa. En lugar de seguir el flujo de acciones previstas, al producirse una excepción, se redirige el mismo hacia algún lugar donde esté previsto como manejar esa excepción. Puede ser que esto sea dentro del mismo método o en otros métodos de otras clases que hayan invocado a aquel en el que se produjo la excepción. Para ello, cuando se encuentra una excepción la JVM instancia un objeto de la clase Exception o de alguna de sus subclases (las distintas subclases de Exception representan distintos tipos de errores) y comienza a buscar algún bloque de código que pueda manejarla. Manejar una excepción es ofrecer una solución al problema que la causa (informamos al operador que el archivo no existe y le pedimos que nos indique otro, por ejemplo) para poder resumir la ejecución del programa o realizar una salida ordenada en el peor de los casos.

Esta búsqueda de un manejador se realiza en primer lugar en el método en el que ocurrió la excepción. Si el mismo no la maneja, se procede hacia abajo por la *pila de llamadas* (call stack) hasta que se encuentre uno. Si no se encuentra un manejador para la instrucción, el programa termina con un mensaje de error.

La pila de llamadas es la forma en que el runtime de Java sabe a que método debe retornar el control cuando termina la ejecución normal de un método. Así, si como en el ejemplo anterior nuestro método Test.main invoca a un método Funciones.traerAnio, y éste invoca a GregorianCalendar.get, la búsqueda de un manejador ocurrirá en orden inverso al que

ocurrieron las llamadas: Primero en GregorianCalendar.get, luego en Funciones.traerAnio y así sucesivamente.

Decimos que el método en el que ocurre la excepción la arroja (*throws*) y que el manejador la atrapa (*catch*) o la maneja. Cuando ejecutamos código en el que puede ocurrir una excepción, éste debe estar rodeado por un bloque *try* (intentar), seguido de una cláusula *catch*, que es la que recibe la excepción y que contiene el código para manejarla. La idea es: intentamos ejecutar un bloque de código, si no hay excepciones, bien, si se encuentra una y es del tipo que declara la cláusula *catch*, se le pasa el control para manejarla. En seguida veremos ejemplos de cómo hacerlo.

Luego del bloque *catch* puede seguir un bloque *finally* (finalmente o por último) en el que se definen las acciones que deben ejecutarse haya ocurrido una excepción o no. La JVM nos garantiza que este bloque de código se ejecutará siempre y por lo general se utiliza para liberar recursos y dejar el sistema en el estado más ordenado posible, ya sea para continuar la ejecución o para hacer una salida prolija.

Tipos de excepciones principales

Todo programa java válido debe cumplir con un requisito: cualquier bloque de código que pueda arrojar ciertas excepciones debe estar dentro de un bloque *try/catch* que las maneje o el método que lo contiene debe especificar que las arroja a través de una cláusula *throws* en su definición. Este requerimiento se denomina *catch or specify*. Todo código que no lo cumpla arrojará un error al compilar.

Hay dos tipos de excepciones principales: Las denominadas *checked exceptions* (una traducción podría ser comprobadas, pero utilizaremos el término inglés), y son condiciones de las que cualquier programa bien construido debería poder recuperarse. El ejemplo anterior del archivo no existente es una de ellas.

Los otros dos tipos de excepciones son conocidas colectivamente como *unchecked exceptions* (no comprobadas). Una de ellas son los errores (*error*), y son condiciones que ocurren fuera del alcance del programa y no pueden preverse pero afectan su ejecución. Las fallas de hardware y de conectividad son ejemplos de las mismas. La aplicación podría intentar recuperarse pero también podría simplemente terminar y dejar una traza de llamadas (es la representación gráfica de la pila de llamadas, *stack trace* en inglés). El otro tipo son las excepciones en tiempo de ejecución (*runtime exception*), generalmente causadas por errores de lógica en el programa (bugs) y tampoco pueden preverse. La aplicación podría manejarlas y recuperarse, pero es preferible tener la información necesaria para corregir el bug que la causa.

Las únicas excepciones que deben cumplir con el requisito de *catch or specify* son las *checked exceptions*. Son *checked exceptions* todas menos las que pertenecen a las clases Error, RuntimeException y sus subclases.

En Funciones agregar:

```
public static double convertirADouble(int n){
	return ((double) n);
}
```

En el paquete modelo:

```
package modelo;
public class Fraccion {
```

```
	private int numerador, denominador;

	public Fraccion(int numerador, int denominador) throws Exception {
		this.numerador = numerador; //en el constructor sólo se asigna por
	el this.atributo
		this.setDenominador(denominador); //o por this.setAtributo, NO se
	debe implementar funcionalidad en el constructor

	}

	public int getNumerador() {
		return numerador;
	}

	public void setNumerador(int numerador) {
		this.numerador = numerador;
	}

	public int getDenominador() {
		return denominador;
	}

	public void setDenominador(int denominador) throws Exception {
		if (denominador ==0) throw new Exception("Error: Objeto
				Fracción inválido, el denominador NO puede ser cero");
		this.denominador = denominador;
	}

	public String toString(){
		return "("+numerador+"/"+denominador+")";
	}

	public Fraccion dividir (Fraccion f) throws Exception{
		if (f.convertirAReal()==0) throw new Exception("Error:
						División por cero: "+f);

		return new Fraccion(numerador*f.getDenominador(),
					denominador*f.getNumerador());
	}

	public double convertirAReal() throws Exception{
		return (Funciones.convertirADouble(numerador) /
			 Funciones.convertirADouble(denominador));
	}


	public double raizCuadrada() throws Exception{
		if (numerador *denominador <0) throw new Exception("Error:
						el radicando es negativo"
		return Math.pow(this.convertirAReal(), 0.5);

	}

}
```

En el Test agregar:

```
package vista;
import modelo.Fraccion;
```

```
public class TestInstanciarFraccion {
	public static void main(String[] args) {
		try{  int n=2, d=0;
			System.out.println("-->Escenario 1: new
						Fraccion("+n+","+d+")");
			Fraccion f1=new Fraccion(n,d);
			System.out.println("Objeto Fraccion: "+f1);
		}
		catch ( Exception e ){
		    System.out.println("Excepcion: " + e.getMessage());
		}

		try{
			int n=3, d=5;
			System.out.println("\n-->Escenario 2: new
						Fraccion("+n+","+d+")");
			Fraccion f1=new Fraccion(n,d);
			System.out.println("Objeto Fraccion: "+f1);
		}
		catch ( Exception e ){
			System.out.println( "Excepcion: " + e.getMessage() );
		}
	}
}
```

.....

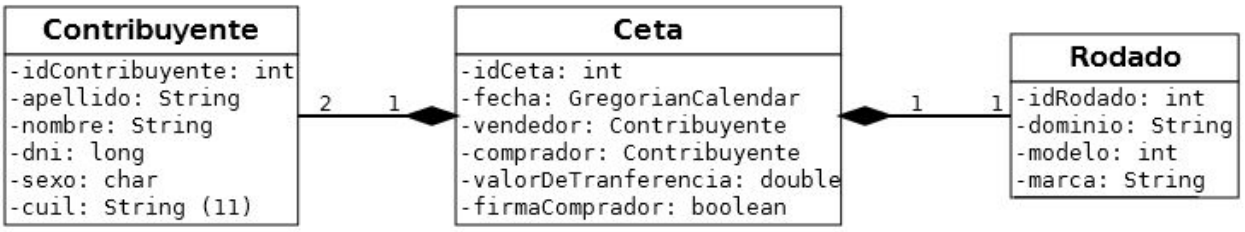
```
package vista;
import modelo.Fraccion;
public class TestDividirFraccion {
	public static void main(String[] args) {
		try{
			Fraccion f2=new Fraccion(2,5);
			Fraccion f3=new Fraccion(0,3);
			System.out.println("-->Escenario 1: Dividir "+f2+
								" por "+f3);
			 System.out.println(f2+"/"+f3+"="+f2.dividir(f3));

		}
		catch ( Exception e ){
			System.out.println("Excepcion: " + e.getMessage());
		}
		try{
			Fraccion f2=new Fraccion(2,5);
			Fraccion f3=new Fraccion(7,3);
			System.out.println("\n-->Escenario 2: Dividir "+f2+
						" por "+f3);
			System.out.println(f2+"/"+f3+"="+f2.dividir(f3));

		}
		catch ( Exception e ){
			System.out.println("Excepcion: " + e.getMessage());
		}
	}
}


public class TestRaizCuadradaFraccion {
	public static void main(String[] args) {
```

```
            try{
                  Fraccion f4=new Fraccion (-3,5);
                  System.out.println("-->Escenario 1:
                                    Raiz cuadrada de "+f4);
                  System.out.println("La raiz cuadrada de"+
                                    f4+"="+f4.raizCuadrada());
            }
            catch ( Exception e ){
                  System.out.println( "Excepcion: " + e.getMessage() );
            }
            try{
                  Fraccion f4=new Fraccion (9,25);
                  System.out.println("\n-->Escenario 2: Raiz cuadrada
                                    de"+f4);
                  System.out.println("La raiz cuadrada de"+
                                    f4+"="+f4.raizCuadrada());
            }
            catch ( Exception e ){
            System.out.println( "Excepcion: " + e.getMessage() );
            }

      }

}
```

TP: Trámite del formulario Ceta

Está basado en el desarrollo Certificado de Transferencia de Automotores CETA que es un tramite que debemos realizar por en el afip desde la web AFIP CETA

Modelo:



\+ validarSexo (char sexo) : boolean
\+ validarCuil (String cuil) : boolean
\+ validarDominio (String dominio) : boolean

Como se verifica un CUIT o CUIL (genérico)

El atributo sexo char con valores posibles, F o M, en el método que valida el cuil los primeros dos dígitos que corresponden 27 y 20 respectivamente.

El CUIL consta de 11 números. Los 10 primeros (2 + 8) constituyen el código de identificación y el último, el dígito de verificación. Para obtener esta verificación se procede de la siguiente forma: A cada dígito del código, se lo multiplica por los siguientes números (respectivamente) 5, 4, 3, 2, 7, 6, 5, 4, 3, 2 y cada valor obtenido, se suma para obtener una expresión (que llamaremos "valor 1". A este "valor 1", se le saca el resto de la división entera a 11. Se obtiene de esta forma un número (del 0 al 10) (que llamamos "valor 2"). Sacamos la diferencia entre 11 y el "valor 2", y obtenemos un valor comprendido entre 1 y 11 (llamémosle "valor 3"). Si "valor 3"=11, el código verificador es cero. Si "valor 3"=10, el código verificador es 9. En cualquier otro caso, el código verificador es "valor 3".

Ejemplo numérico con un número de CUIT, que es 20-17254359-7.

```
  2    0    1    7    2    5    4    3    5    9
x
  5    4    3    2    7    6    5    4    3    2
-------------------------------------------------
 10 + 00 + 03 + 14 + 14 + 30 + 20 + 12 + 15 + 18 = 136

v1 = 136
136 mod 11 = 4
v2 = 4
11 - 4 = 7
v3 = 7 => Código de verificación es siete.
```

Excepciones mínimas a implementar: ERROR: CUIL inválido; ERROR: Dominio inválido; ERROR: El vendedor y el comprador son los mismos; ERROR: la fecha no puede ser posterior al día de hoy

En los casos que se implementan las excepciones, el constructor asigna el valor al "this.atributo" utilizando el "setAtributo" y a su vez el set llama al método "validarAtributo" que de retornar un true realizará la asignación.

Test: implementar los Test necesarios para los escenario de excepciones.

Análisis de casteo:

La cuestión ocurre al castear por ejemplo entre long y double, porque un long utiliza 63 bits para el valor y 1 para el signo, mientras que el double usa 52 para la mantisa y 1 para el signo. el resto es el exponente. Si el long que queremos castear ocupa más de 52 bits va a ocurrir pérdida de precisión, porque el double sólo puede almacenar 52 bits significativos. Puede (no vamos a tener error) almacenarse porque el double usa exponente, pero se perderán los dígitos menos significativos.

El int ocupa 32 bits en java, así que no debería haber problema al castearlo a double, ya que entra, el problema aparecería si lo queremos castear a float (usa 23 bits para la mantisa), por ejemplo.

Algunos sugieren verificar que la longitud en bits del número no supere la de la mantisa y levantar una excepción, si así ocurre. Si hacemos el casteo de un int a un float directamente, java pierde los dígitos menos significativos y hace la conversión sin decir nada.

De todos modos, la conversión como la estamos haciendo es ineficiente, ya que pasa el int a un string y después ese string a un double y nos nos evita el problema, ya que tampoco funciona:

```
int i=2147483638; //(2^31-10)
float f=(float)i;
System.out.println(i);
System.out.println(f);
System.out.println((int)f);
System.out.println((int)Float.parseFloat(String.valueOf(i)));
```

esto devuelve:
2147483638
2.14748365E9
2147483647
2147483647

En ambos casos se pierde precisión, y la conversión a String y float es más cara.