**CONCEPTOS Y PARADIGMAS DE LENGUAJA DE PROGRAMACIÓN**

**TRABAJO INTEGRADOR 2018**

**Segunda Parte**

**INTEGRANTES:**

* **SALGADO IVAN, Legajo 11823/6**
* **ALONSO DIAZ MARIANO HUGO IGNACIO, Legajo 11046/7**
* **VARELA JUAN MANUEL, Legajo 11997/9**

**GRUPO: 43**

**LENGUAJES: Processing – Python**

**AYUDANTE: Viviana**

**TURNO: Viernes (M)**

**BIBLIOGRAFIA:**

<http://docs.python.org.ar/tutorial/3/errors.html>

<http://www.python.org.ar>

<http://python.org.ar/pyar/Tutorial>

<https://javadesdecero.es/intermedio/manejo-de-excepciones/#1_Manejo_de_Excepciones>

**Correcciones de la entrega #1:**

A. Desarrolle en EBNF el bloque “for” donde se puedan utilizar expresiones numéricas en los delimitadores. Estas expresiones pueden tener paréntesis.

- La expresión está mal definida

- Falta definir <bloque>

G = (N, T, S, P)

N = {

<sentencia\_for>, <variable>, <inicializada>, <var>, <letra>, <alfanumérica>, <digito>, <real>, <declarada\_inicializada>, <tipo\_dato>, <comparación>, <signo\_comparacion>, <calculo>, <incremento>, <decremento>, <expresión>, <fact>, <term>

}

T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, .. , z, A, .. , Z, for, ;, (, ), =, <, >, >=, <=, ==, !=, +, -, \*, /, ++, - -}

S = { <sentencia\_for> }

P = {

**<sentencia\_for>** ::= ‘for’ ‘(‘ <variable> ‘;’ <comparación> ‘;’ <calculo> ‘)’ ‘{‘

<bloque> ‘}’

**<variable>** ::= (<declarada\_inicializada> | <inicializada>)

**<inicializada>** ::= <var> ‘=’ <real>

**<var>** ::= <letra> {<alfanumérico>}\*

**<letra>** ::= (‘a’ | .. | ‘z’ | ‘A’ | .. | ‘Z’ |)

**<alfanumérica>** ::= (<letra> | <digito>)

**<digito>** ::= (‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’)

**<real>** ::= [(+ | -)] {<digito>} + [‘.’{<digito>}+]

**<declarada\_inicializada>** ::= <tipo\_dato> <inicializada>

**<tipo\_dato>** ::= (‘int’ | ‘float’ | ‘byte’)

**<comparación>** ::= (<var> | <real> ) <signo\_comparacion> (<var> | <real>)

**<signo\_comparación>** ::= (‘<’ | ‘>’ | ‘>=’ | ‘<=’ | ‘==’ | ‘!=’)

**<calculo>** ::= (<incremento> | <decremento> | <expresión>)

**<incremento>** ::= <var> ‘++’

**<decremento>** ::= <var> ‘- -‘

**<expresión>** ::= <fact> { ( ‘+’ | ‘-‘ ) <fact> }\*

**<fact>** ::= <term> { (‘\*’ | ‘/’) <term> }\*

**<term>** ::= ( <var> | <real> | ‘(’ <expresión> ‘)’ )

**<bloque>** ::= (<sentencia\_if> | <sentencia\_switch> | <sentencia\_while> | <sentencia\_do\_while> | … todas las sentencias, declaraciones, llamadas a métodos, etc )

}

**2da Parte (Fecha Final de entrega: 29/06):**

**D. Suponiendo que Ud. está definiendo un nuevo lenguaje de programación, defina y enuncie las características más importantes del manejo de los parámetros incorporando las características que Ud. considere necesarias de los lenguajes asignados a su grupo.**

Los parámetros o argumentos son una forma de intercambiar información con el método. Pueden servir para introducir datos para ejecutar el método (entrada) o para obtener o modificar datos tras su ejecución (salida).

**E. Realice una tabla comparativa permitiendo visualizar las diferencias más relevantes respecto del sistema de tipos de los lenguajes asignados. Justifique las características mencionadas con ejemplos de código (al menos para 3 de las características).**

|  |  |
| --- | --- |
| **Processing** | **Python** |
| Fuertemente Tipado | Fuertemente Tipado |
| Según tipo de ligadura: Tipado Estático, pero también cuenta con tipado dinámico (casting). | Según tipo de ligadura: Tipado Dinámico |
| Tipos Primitivos: byte, short, int, long, float, doublé, char, boolean. | Tipos Primitivos: byte, char, boolean, float. |
| Tipos definidos por el usuario: clases (el programador puede crear sus propias clases e instanciar). | Tipos definidos por el usuario: clases. |
| Tipos Compuestos: arreglos. | Tipos Compuestos: listas |
|  | Estos tipos de datos pueden ser mutables (ej. listas) o inmutables (ej. string). |
|  |  |

**F. Suponiendo que Ud. está definiendo un nuevo lenguaje de programación, defina y enuncie las características más importantes del manejo de excepciones incorporando las características que usted considere necesarias de los lenguajes asignados a su grupo. Ejemplifique con código.**

Una excepción es la indicación de que se produjo un error en el programa. Las excepciones, como su nombre lo indica, se producen cuando la ejecución de una instrucción no termina correctamente, sino que termina de manera excepcional como consecuencia de una situación no esperada.

Cuando se produce una situación anormal durante la ejecución de un programa (por ejemplo se accede a un objeto que no ha sido inicializado o tratamos de acceder a una posición inválida en un vector), si no manejamos de manera adecuada el error que se produce, el programa va a terminar abruptamente su ejecución.

Existen dos tipos de modelo de manejo de excepciones:

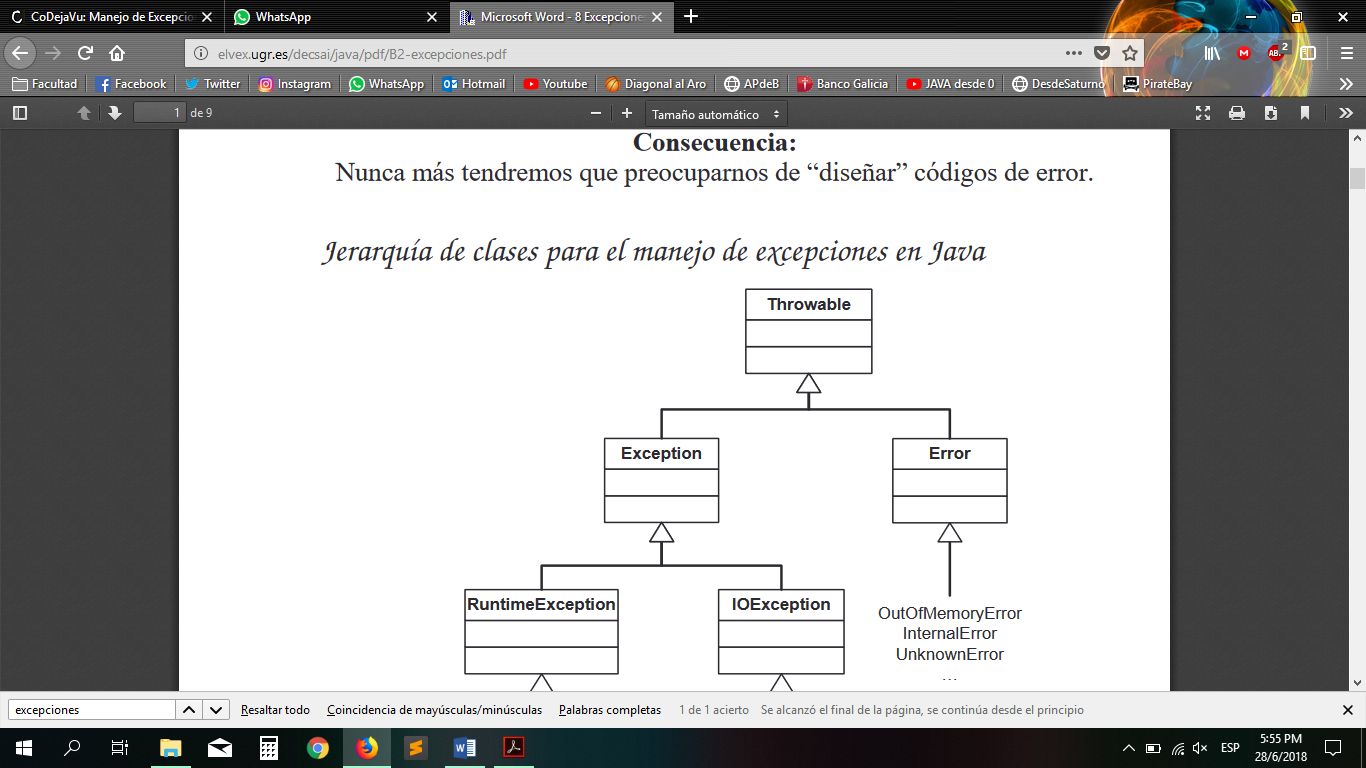
* Reasunción: se maneja la excepción y se devuelve el control al punto siguiente donde se invocó la excepción, permitiendo la continuación de ejecución de la unidad. El lenguaje que lo utiliza es PL/1.
* Terminación: se termina la ejecución de la unidad que alcanza la excepción y se transfiere el control al manejador. Los lenguajes que lo utilizan son ADA, CLU, C++, JAVA.

**Processing:**

El modelo de ejecución es por terminación.

Cuando se produce un error en un método, “se lanza” un objeto *Throwable*. Cualquier método que haya llamado al método puede “capturar la excepción” y tomar las medidas que estime oportunas. Tras capturar la excepción, el control no vuelve al método en el que se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se haya capturado la excepción.

Jerarquía de clases:



* Throwable: Clase base que representa todo lo que se puede “lanzar” en Java.
  + Contiene una instantánea del estado de la pila en el momento en el que se creó el objeto (“stack trace” o “call chain").
  + Almacena un mensaje (variable de instancia de tipo String) que podemos utilizar para detallar qué error se produjo.
  + Puede tener una causa, también de tipo Throwable que permite representar el error que causó este error.
* Error: Subclase de Throwable que indica problemas graves que una aplicación no debería intentar solucionar (documentación de Java). Este tipo de excepciones escapan a su control y, por lo general, tu programa no se ocupará de ellas.
* Exception: Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.
  + Los dos tipos principales de excepciones son:
    - *RuntimeException* (errores del programador, como una división por cero o el acceso fuera de los límites de un array)
    - *IOException* (errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa)

Manejo:

El manejo de excepciones Java se gestiona a través de cinco palabras clave: *try*, *catch*, *throw*, *throws*,y *finally*. Forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro.

Las declaraciones de programa que desea supervisar para excepciones están contenidas dentro de un bloque **try**. Si se produce una excepción dentro del bloque try, se lanza. El código puede atrapar esta excepción usando **catch** y manejarlo de una manera racional. Las excepciones generadas por el sistema son lanzadas automáticamente por el sistema de tiempo de ejecución de Java. Para lanzar manualmente una excepción, use la palabra clave **throw**. En algunos casos, una excepción arrojada por un método debe ser especificada como tal por una cláusula **throws**. Cualquier código que debe ejecutarse al salir de un bloque try se coloca en un bloque **finally**.

Uso del *Try y Catch:*

En el centro del manejo de excepciones están try y catch. Estas palabras clave trabajan juntas; no puedes atrapar (catch) sin intentarlo (try).

Ejemplo:

try{

//bloque de código para monitorear errores

}

catch (TipoExcepcion1 exOb){

//Manejador para TipoExepción1

}

catch (TipoExcepcion2 exOb){

//Manejador para TipoExepción2

}

Acá, *TipoExcepcion* es el tipo de excepción que ocurrió. Cuando se lanza una excepción, es atrapada por su instrucción catch correspondiente, que luego procesa la excepción. Como muestra la forma general, puede haber más de una declaración catch asociada con un try. El tipo de la excepción determina qué declaración de captura se ejecuta. Es decir, si el tipo de excepción especificado por una instrucción catch coincide con el de la excepción, entonces se ejecuta esa instrucción de catch (y todos los demás se anulan). Cuando se detecta una excepción, exOb recibirá su valor.

Lanzar una excepción:

Es posible lanzar manualmente una excepción utilizando la instrucción throw. Su forma general se muestra como “throw excepcOb;”. excepcOb debe ser un objeto de una clase de excepción derivada de Throwable. Ejemplo que ilustra la instrucción throw arrojando manualmente una ArithmeticException:

public class ThrowDemo {

public static void main(String[] args) {

try{

System.out.println("Antes de lanzar excepción.");

throw new ArithmeticException(); //Lanzar una excepción

}catch (ArithmeticException exc){

//Capturando la excepción

System.out.println("Excepción capturada.");

}

System.out.println("Después del bloque try/catch");

}

}

Uso de finally:

Cuando se quiere definir un bloque de código que se ejecutará cuando quede un bloque try/catch. Por ejemplo, una excepción puede causar un error que finaliza el método actual, causando su devolución prematura. Sin embargo, ese método puede haber abierto un archivo o una conexión de red que debe cerrarse.

Tales tipos de circunstancias son comunes en la programación, y hay una forma conveniente de manejarlos: *finally*. Para especificar un bloque de código a ejecutar cuando se sale de un bloque try/catch, incluya un bloque finally al final de una secuencia try/catch. Aquí se muestra la forma general de un try/catch que incluye finally. El bloque finally se ejecutará siempre que la ejecución abandone un bloque try/catch, sin importar las condiciones que lo causen. Es decir, si el bloque try finaliza normalmente, o debido a una excepción, el último código ejecutado es el definido por finally. El bloque finally también se ejecuta si algún código dentro del bloque try o cualquiera de sus declaraciones catch devuelve del método.

Ejemplo:

public class UsoFinally {

public static void genExcepecion(int rec) {

int t;

int nums[]=new int[2];

System.out.println("Recibiendo "+rec);

try {

switch (rec){

case 0:

t=10 /rec;

break;

case 1:

nums[4]=4; //Genera un error de indexación

break;

case 2:

return; //Retorna desde el blorec try

}

}

catch (ArithmeticException exc){

//Capturando la excepción

System.out.println("No se puede dividir por cero!");

return; //retorna desde catch

}

catch (ArrayIndexOutOfBoundsException exc){

//Capturando la excepción

System.out.println("Elemento no encontrado");

}

finally {

//esto se ejecuta al salir de los blorecs try/catch

System.out.println("Saliendo de try.");

}

}

}

Uso de throws:

En algunos casos, si un método genera una excepción que no maneja, debe declarar esa excepción en una cláusula throws. Aquí está la forma general de un método que incluye una cláusula throws:

tipo-retorno nombreMetodo(lista-param) throws lista-excepc {

// Cuerpo

}

**Python:**

El modelo de ejecución es por terminación.

Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas). Por ejemplo, *ZeroDivisionError*, *NameError* y *TypeError*.

Ejemplo:

While true:

Try:

X = int (input (“Dividendo:”))

Y = int (input (“Divisor:”))

If (X < 0) or (Y < 0):

myError = ValueError (‘solo se aceptan valores positivos’)

Raise myError

Print X/Y

Except ZeroDivisionError:

Print “Error: Division por 0”

Except ValueError:

Print myError

Else: #solo si no se produjo una excepción

Print “dividiendo”, X, “-divisor:”, Y , “Resultado:” , X/Y

Finally:

Print “esto se ejecuta siempre”

Manejo de excepciones en Python:

La declaración **try** funciona de la siguiente manera:

* Primero, se ejecuta el bloque try (el código entre las declaraciones try y except).
* Si no ocurre ninguna excepción, el bloque except se saltea y termina la ejecución de la declaración try.
* Si ocurre una excepción durante la ejecución del bloque try, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada except, se ejecuta el bloque except, y la ejecución continúa luego de la declaración try.
* Si ocurre una excepción que no coincide con la excepción nombrada en el except, esta se pasa a declaraciones try de más afuera; si no se encuentra nada que la maneje, es una excepción no manejada, y la ejecución se frena con un mensaje como los mostrados arriba.

Una declaración **try** puede tener más de un **except**, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente try, no en otros manejadores del mismo try. Un except puede nombrar múltiples excepciones usando paréntesis

Las declaraciones **try... except** tienen un bloque **else** opcional, el cual, cuando está presente, debe seguir a los except. Es útil para aquel código que debe ejecutarse si el bloque try no genera una excepción.

Definiendo acciones de limpieza:

La declaración **try** tiene otra cláusula opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Una cláusula **finally** siempre es ejecutada antes de salir de la declaración try, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la cláusula try y no fue manejada por una cláusula except (u ocurrió en una cláusula except o else), es relanzada luego de que se ejecuta la cláusula finally. El Finally es también ejecutado “a la salida” cuando cualquier otra cláusula de la declaración try es dejada via break, continue or return.

En aplicaciones reales, la cláusula finally es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

Levantando excepciones:  
La declaración **raise** permite al programador forzar a que ocurra una excepción específica.

Excepciones definidas por el usuario

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción. Las excepciones, típicamente, deberán derivar de la clase *Exception*, directa o indirectamente.

Las clases de Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores de la excepción. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error.

La mayoría de las excepciones son definidas con nombres que terminan en "Error", similares a los nombres de las excepciones estándar. Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias.

**G. Conclusión sobre los lenguajes: Realice una conclusión exponiendo las diferencias, características, virtudes y defectos de los lenguajes asignados. Esta conclusión no debe superar una carilla**

Características:

* **Processing**:
  + Orientado a objetos: toda la programación en processing en su mayoría está orientada a objeto, ya que al estar agrupados en estructuras en estructuras encapsuladas es más fácil su manipulación.
  + Portable: por ser indiferente a la arquitectura sobre la cual está trabajando, esto hace que su portabilidad sea muy eficiente, sus programas son iguales en cualquiera de las plataformas, ya que processing especifica tamaños básicos.
  + Seguro: la seguridad es una característica muy importante en processing ya que se han implementado barreras de seguridad en el lenguaje y en el sistema de ejecución de tiempo real.
  + Simple: se lo conoce como lenguaje simple porque viene de la misma estructura de c y c++; ya que c++ fue un referente para la creación de java y processing está basado en java.
  + Robusto: es altamente fiable en comparación con c, se han eliminado muchas características con la aritmética de punteros, proporciona numerosas comprobaciones en compilación y en tiempo de ejecución.
  + Dinámico: el lenguaje java es muy dinámico en la fase de enlazado, sus clases solamente actuarán en medida en que sean requeridas o necesitadas con esto permitirá que los enlaces se puedan incluir incluso desde fuentes muy variadas o desde la red.
  + **PREGUNTAR????** Compilado: es un lenguaje que es compilado, generando ficheros de clases compilados, pero estas clases compiladas son en realidad interpretadas por la máquina virtual java. Siendo la máquina virtual de java la que mantiene el control sobre las clases que se estén ejecutando.
* **Python:**

Ventajas y desventajas:

* **Processing:**
  + Ventajas:
    - La principal característica es que es independiente de la plataforma. Esto significa que cuando estás programando en Processing, no necesitas conocer a priori el tipo de ordenador o el sistema operativo para el que estás programando. Puedes ejecutar EL MISMO programa en un PC con Windows, otro con Linux, en un Servidor SUN con sistema operativo Solaris, o en un teléfono móvil de última generación.
    - El lenguaje Processing es orientado a objetos. El paradigma de programación orientada a objetos supuso un gran avance en el desarrollo de aplicaciones, ya que es capaz de acercar la forma de programar a la forma de pensar del ser humano.
    - En Processing no existen problemas con la liberación de memoria en el sistema: Decidieron romper con el sistema tradicional de liberación de memoria, haciendo que el programador ya no fuese el responsable de esa tarea. Así, lo único que necesita hacer el programador es solicitar la memoria al sistema.
    - El lenguaje Processing es relativamente fácil de aprender comparado con otros.
    - Librerías Estándar: Una de las características que más potencia aporta al lenguaje Processing es que viene acompañado de una serie de librerías estándar para realizar multitud de operaciones comunes a la hora de programar.
    - Hoy en día existen excelentes editores (IDEs) que aportan multitud de ayudas a la programación, haciendo que el desarrollo sea más fluido y cómodo.
    - Una de las soluciones más elegantes propuestas por el lenguaje Processing a uno de los problemas recurrentes en otros lenguajes de programación es la gestión de errores a través de excepciones. en C o C++ no existe un mecanismo específico para la gestión de los errores que puedan producirse en el código.
  + Desventajas:
    - Menos Eficiente, comparado a C/C++.
    - Requiere un intérprete.
    - Algunas implementaciones y librerías pueden tener código rebuscado.
    - Una mala implementación de un programa en processing, puede resultar en algo muy lento.
    - Algunas herramientas tienen un costo adicional.
* **Python:**

**H. Conclusión sobre el trabajo: Realice una conclusión mencionando los aportes que le generó la realización del trabajo como grupo.**

**Nuestra conclusión sobre el trabajo:**

La cursada es bastante diferente a las materias que venimos cursando años anteriores, notamos que hay que realizar muchas investigaciones bibliográficas, ya sea por libros o a través internet. Nos resultó interesante ver diferentes tipos de lenguajes de programación, y no solamente uno como surge mayormente en las demás materias. También esta bueno conocer las distintas características que posee cada uno, y de algunos pudimos ver más en profundidad cómo es su sintaxis y semántica, como por ejemplo los lenguajes de Python y Processing que fueron los que nos designaron para el trabajo.

Con respecto al trabajo, nos costó entender algunas consignas ya que no especificaba muy claro lo que teníamos que realizar o nosotros no pudimos entenderlo correctamente, pero pudimos consultarlas en cualquier horario de consulta con los ayudantes y eso nos ayudo mucho en aclarar conceptos, como así también dudas sobre lo que realizamos.