Dokumentáció

Magic Poloska a Logo nyelv egy implementációja az Imagine logo mintájára.

A dokumentációban az osztályok működését egy minta bemenet alapján fogom végig vezetni, mely jól megmutatja miként változik a bemenet alakja, ahogy kiszűrjük belőle majd elvégezzük a kiadott parancsokat.

A példa parancs a következő:

b 90 loop 7{loop $l1+3{e 100 j 360/($l1+3)}}

Tokenizer:

Az első szűrő réteg amin áthalad a parancsunk az a Tokenizer. A Tokenizerek egyetlen feladata beazonosítani a különböző részek fajtáit és azokat egy Token egységbe szedni.

Token:

Ez egy tároló osztály mely egy TokenType-ot tárol mely a Token típusát azonosítja, és egy string értéket abban az esetben ha a Token típusa nem írja le eléggé az azonosításhoz.

pl.: [NUMBER, 14.56] v. [OPERATOR, +] v. [LPARAM]

Egy parancsot a Tokenizernek a konstruktorán keresztül adunk meg. A karaktereken iterátor segítségével haladunk át. A tokenizer nagy előnye, hogy így bizonyos karaktereket figyelmen kívül is hagyhatunk, mint például a space és sortörés vagy tabulátor karaktereket.

Parancs jelenlegi állapota:

(STRING b), (NUMBER 90), (STRING loop), (NUMBER 7), (CURLYLP), (STRING loop), (VARIABLE l1), (OPERATOR +), (NUMBER 3), (CURLYLP), (STRING e), (NUMBER 100), (STRING j), (NUMBER 360), (OPERATOR /), (LPARAM), (VARIABLE l1), (OPERATOR +), (NUMBER 3), (RPARAM), (CURLYRP), (CURLYRP)

Parser:

Miután lebontottuk a parancsot Tokenekre ideje azt parancsokká formálni. Ezt valósítja meg a Parser osztály. A parse metódusa egy Command Listát ad vissza ami tárolja a létrehozott utasításokat.

Command:

Egy interfész amiből származnak le az utasítások. Minden leszármazottjának kell lennie egy execute(Poloska p) metódusának. Melybe a különböző utasítások megvalósítása található.

A parse függvény úgy működik, hogy végig olvassa a Tokeneket egyessével és, ha talál egy STRING tokent akkor tudja, hogy az egy utasításnak kell lennie, de még nem hozza létre az utasítást, hanem addig olvas amíg nem jut a végére vagy nem talál egy újabb STRING tokent.

Addig amíg ez nem következik be egy ideiglenes tömbe pakolja a Tokeneket mert, hogy azok az előző utasításhoz tartoznak. Ha talált egy új STRINGET akkor létrehozásra várakozó utasítást megkreálja és az összegyűjtött Tokeneket pedig paraméterként hozzáadja. Majd a createExpresion() függvény eldönti, hogy milyen metódust hoz létre a STRING token értéke alapján, majd ezt hozzáadjuk a végső listához

A Command implementációi és a hozzátartozó parancsok:

Utasítás	Osztály	Leírás
„e”	MoveExpression	Előre mozgás.
„j”	RotateExpression	Jobbra fordul a fokokban megadott értéket.
„b”	RotateExpression	Ballra fordul a fokokban megadott értéket.
„h”	MoveExpression	Hátra mozgás.
„loop”	LoopExpression	Ciklus megvalósító osztály. **részletesebben később**
„pw”	PenWidthExpression	A Pen vastagságát állítja be.
„prgb”	PenColorChangeRGB	A Pen színét állítja be RGB értékek alapján.
„phex”	PenColorChangeHEX	A Pen színét állítja be kex érték alapján pl.:(#ffffff)
„clr”	ClearScreen	Törli az eddig rajzolt dolgokat és a poloskát alaphelyzetbe állítja
„pu”	PenUp	Felemeli a tollat, így minden további művelet rajzolás nélkül hajtódik végre.
„pd”	PenDown	A tollat leteszi, így minden mozgás nyomot hagy maga után.
„eljárás neve”	FunctionExpression	Eljárásokat reprezentál. **részletesebben később**

LoopExpression:

Ez az osztály a fentiek közül az egyik izgalmasabb ezért bővebb kifejtésre szorulhat.

Egy ciklus a következő formában található: **loop 4{e 100 j 90}**

Token formában a következőt kapja:
(NUMBER 4), (CURLYLP), (STRING e), (NUMBER 100), (STRING j), (NUMBER 90), (CURLYRP)

Először is az első CURLYLP tokenig található tokeneket számként értelmezi, ez fogja reprezentálni, hogy hányszor ismételjük a {} közötti utasítás sort. Ezután a maradék tokenekre a {} kivéve, meghívja a Parser parse metódusát. Az executeban pedig csupán az ebből kapott utasításokat hajtjuk végre annyiszor amennyi a kiértékelt számunk volt.

A cikluson belül lehetőségünk van elérni, a ciklus változónkat is. A változók a követlező nevet kapják automatikusan. „l”+ a legkisebb szám amire még nincs változónk 1-től kezdve. Kivétel ha eljárásban használjuk akkor „az eljárás neve_l”-t használ sima „l” helyett erre azért van szükség, hogy amikor az eljárást írjuk akkor

figyelmen kívül hagyhassuk, hogy amikor majd írjuk a parancsot a bemenetre akkor éppen milyen változó nevet kap az adott ciklus. Ez ezenkívül lehetővé teszi az eljárásokon belüli eljárás hívásokat is, kivétel a rekurzív hívásokat.

A változónevek kiosztásáért a VariableStorage osztály felelős. A változókat minden execute elején rakom bele a változótáblába, és amint vége van az utasításnak azt ki is törlöm, így biztosítva, hogy a ciklus változók csakis a ciklus belsejében érhetőek el, és így újra feltudjuk használni a változó neveket, így könnyen következtető, hogy éppen melyik tartozik az adott ciklushoz.

FunctionExpression:

A program másik komplexebb parancs osztálya a FunctionExpression. Ez az osztály felelős az eljárások értelmezésért. A konstruktorába a tokeneken kívül megkapja még az eljárás nevét is. Ezután megpróbálja megtalálni a névhez tartozó függvényt a FunctionStorage osztályban. Ezután az eljáráshoz tartozó előre definiált utasítás sztringet áttolja az előzőekben leírt értelmező rétegeken. (Tokenizer->Parser->Execute) Az eljáráshoz tartozhatnak változók is melyeket az utasítás sorozatban használhatunk, és a eljáráshívásnál kell meghatároznunk. Ezeket az execute elején felvesszük a VariableStorrageba a megfelelő értékekkel majd az execute végén eltávolítjuk őket. Fontos, hogy az eljárások változóinak különböző nevet kell adni abban az esetben ha azokat egymásba szeretnénk ágyazni.

Változók:

Amennyiben változókat szeretnénk használni a változó neve ’$’ jelet kell helyezni. Innen tudja a program, hogy ez egy változó és nem egy parancs. A változók nevei Tartalmazhatnak betűket, számokat és ’_’ karaktereket.

ShuntingYardParser:

A programban a számokat Abstract Syntax Treeben(AST) tárolom ez lehetővé teszi bonyolultabb matematikai számítások elvégzését is. Szövegből a Shunting Yard algoritmust felhasználva alakítom a számokat AST formába. A számokat double változók tárolják a programban.

ASTNode:

Egy absztrakt osztály melyből leszármaznak az AST fa elemei. Minden Nodenak van egy sztring értéke, és 2 szomszédos nodeja. A tényleges értéket az evaluate függvény hívásával kapjuk. Ezek általában meghívják a szomszédai evaluate függvényeit is egész addig amíg nem érünk el egy olyan Nodeig melynek tényleges értéke van.

Az általam implementált algoritmus a következő 5 matematikai műveletet ismeri: összeadás(+), kivonás(-), szorzás(*), osztás(/), hatványozás(^). Az egyváltozós műveleteket nem támogatja. Az algoritmus képes a zárójelek alkalmazására is. A tényleges konverziót a

convertTokenToAST(List<Token> tokens) függvény végzi, mely a beérkező Tokenekből kialakít egy AST-t.

Operator:

Egy segéd osztály mely a Shunting Yard algoritmus megvalósítását egyszerűsíti.

Poloska:

A rajzoló poloskáért felelős osztály, tárolja a jelenlegi pozícióját és forgását is. Itt történik a rajzfelületre való rajzolás. A rajzolást befolyásoló változókat a Pen osztály öleli körbe.

Pen:

A rajzoláshoz alkalmazott „tollat” reprezentálja, ebben található a rajzolás során használt toll vastagság, szín és az, hogy a toll éppen fel van-e emelve. Ahhoz hogy használhassuk, a drawWithPen() metódusát kell meghívni, ami a rajzfelület GraphicsContext-jét várja paraméterként majd miután annak a tulajdonságait beállította, visszadja visszatérési értékben.

VariableStorage:

A programban található különböző változókat tároló singleton osztály. Az osztály azért lett singleton, mivel alapból csak egy tárolót szeretnénk, és egyébként pedig a az öröklődések és egymásba ágyazottság miatt rengetek helyen kellene paraméterként átadogatni, még olyan helyeken is ahol abszolút nem használjuk. A változókat egy Map-ben tárolom, név és double értékpárok alapján. A tároló feladata még kiosztani a megfelelő neveket a ciklusváltozókhoz. Az osztály szintén tud róla, hogy ha egy eljárásban lévő kód részletet futtat, mivel akkor az eljárásnak megfelelő ciklusváltozót kell adnia.

FunctionStorage:

A VariabeStorage-hoz nagyon hasonlóan ez is egy singleton tároló osztály, mely hasonló indokok miatt lett singleton. Mivel a program teljesen különböző részein hozom létre őket és másik részén kérdezem le, így a összekapcsoltság miatt szinte minden függvényben szerepeltetni kellett volna paraméterként. Tárolásnak egy ObservableList-et használ, amibe Function objektumokat tárol. Azért használ ObservableList-et mivel így a JavaFx felület ListView-ével közvetlen összeköthető. Lehetőséget ad újabb elemek felvételére törlésére és a mentés és betöltés is itt van megvalósítva.

JavaFx kinézet:

A kinézet megvalósításához JavaFx környezetet alkalmaztam, FXML fájlok felhasználásával.

PoloskaController:

A főablak vezérlője a sample.fxml fájlt használja alapjául.

FunctionList:

Az eljárások menedzselésére használt ablak vezérlője, a FunctionList.fxml-t nyitja meg.

FunctionPanel:

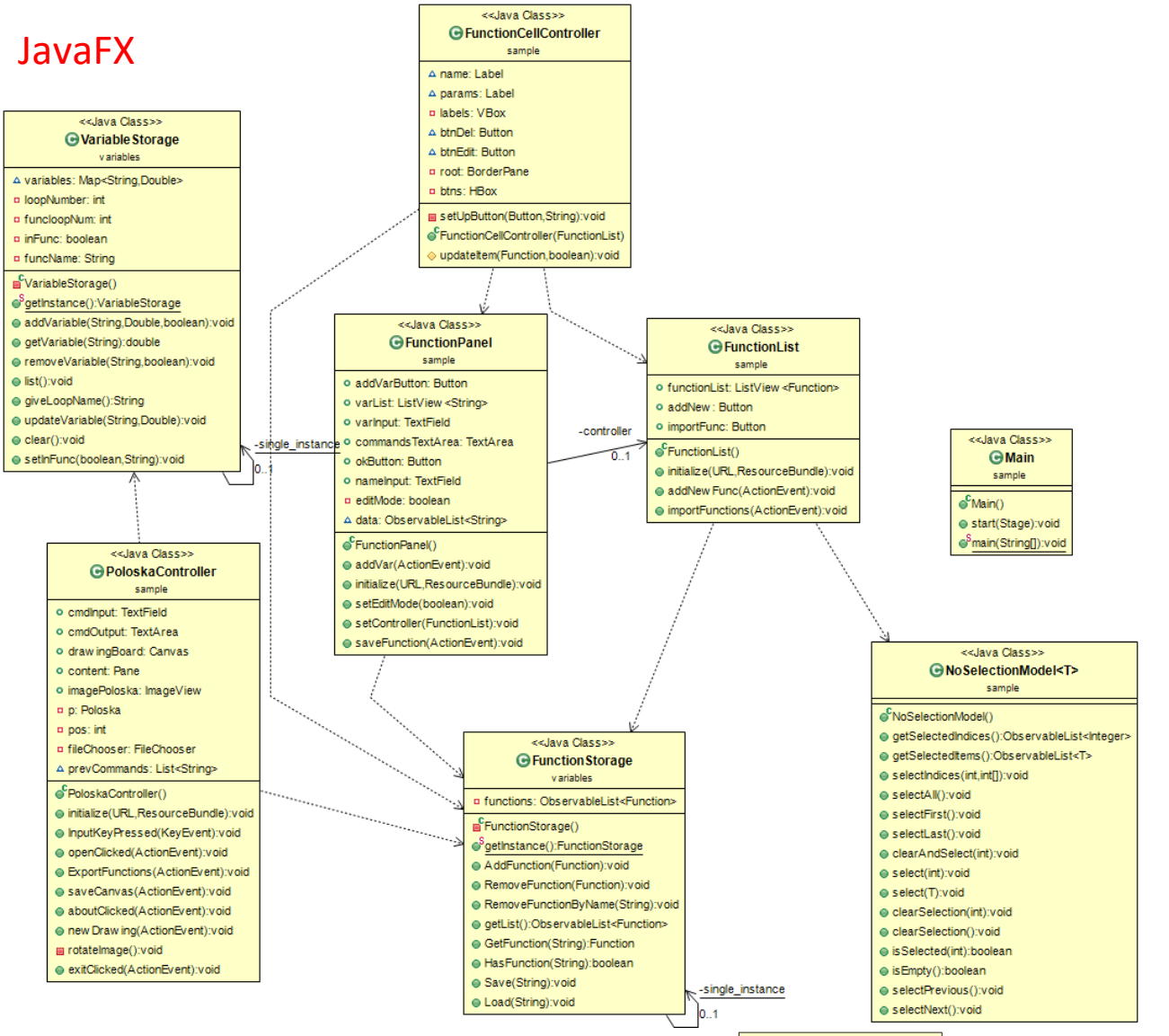
Azon ablak vezérlője ahol egy új eljárást tudunk felvenni, vagy egy már meglévőt szerkeszteni. A functionPanel.fxml-t használja alapjául.

FunctionCellController:

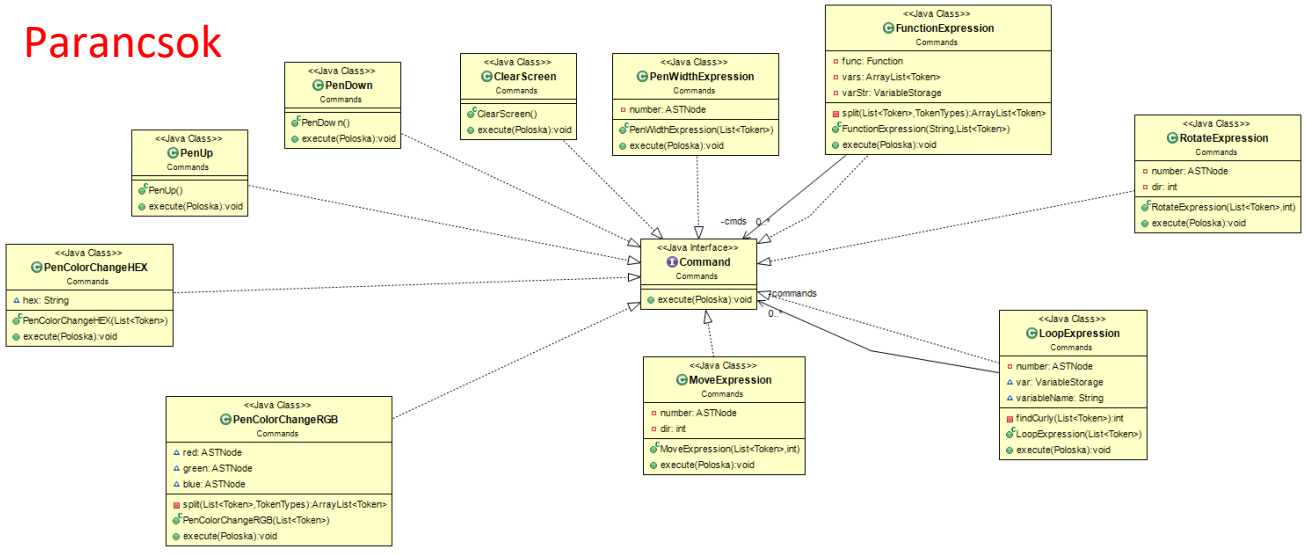
A FunctionListen megjelenő egyedi listaelemek vezérlője.

Osztálydiagrammok:

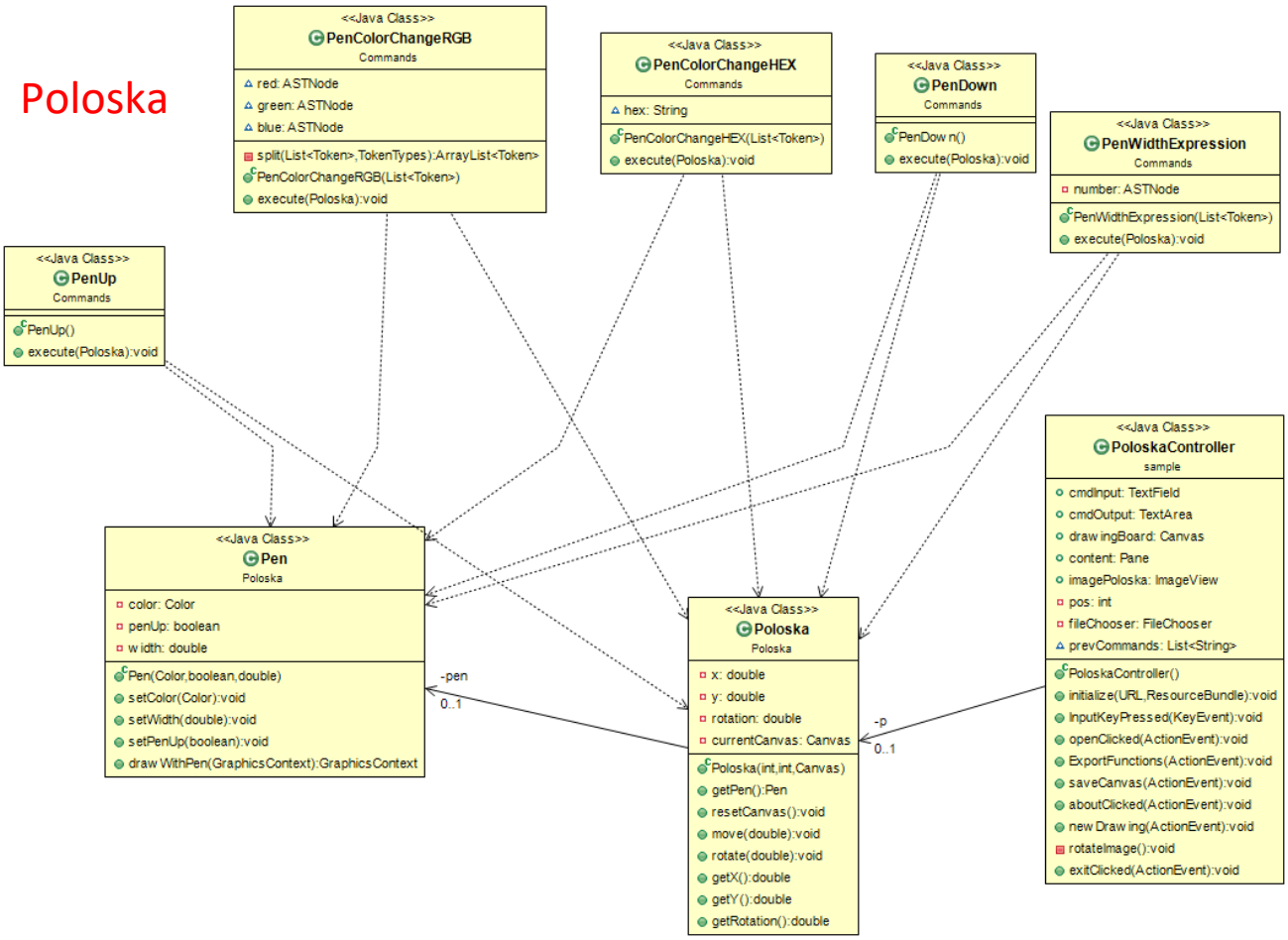
JavaFX

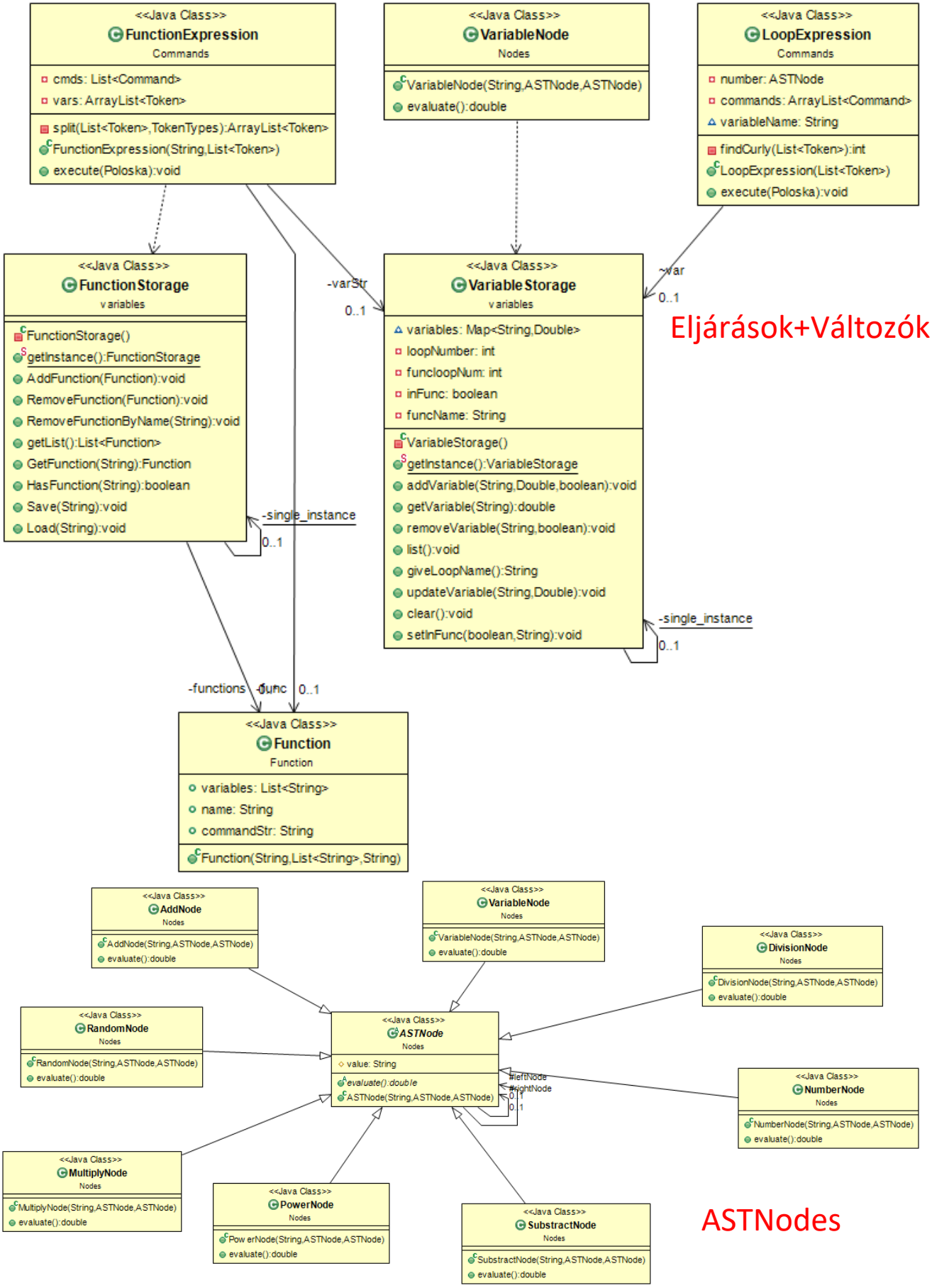


Parancsok



Poloska


<<Java Class>>
PenColorChangeRGB
Commands
red: ASTNode
green: ASTNode
blue: ASTNode
split(List<Token>,TokenTypes):ArrayList<Token>
PenColorChangeRGB(List<Token>)
execute(Poloska):void
<<Java Class>>
PenColorChangeHEX
Commands
hex: String
PenColorChangeHEX(List<Token>)
execute(Poloska):void
<<Java Class>>
PenDown
Commands
PenDown()
execute(Poloska):void
<<Java Class>>
PenWidthExpression
Commands
number: ASTNode
PenWidthExpression(List<Token>)
execute(Poloska):void
<<Java Class>>
PenUp
Commands
PenUp()
execute(Poloska):void
<<Java Class>>
Pen
Poloska
color: Color
penUp: boolean
width: double
Pen(Color,boolean,double)
setColor(Color):void
setWidth(double):void
setPenUp(boolean):void
drawWithPen(GraphicsContext):GraphicsContext
-pen
0..1
<<Java Class>>
Poloska
Poloska
x: double
y: double
rotation: double
currentCanvas: Canvas
Poloska(int,int,Canvas)
getPen():Pen
resetCanvas():void
move(double):void
rotate(double):void
getX():double
getY():double
getRotation():double
-p
0..1
<<Java Class>>
PoloskaController
sample
cmdInput: TextField
cmdOutput: TextArea
drawingBoard: Canvas
content: Pane
imagePoloska: ImageView
pos: int
fileChooser: FileChooser
prevCommands: List<String>
PoloskaController()
initialize(URL,ResourceBundle):void
inputKeyPressed(KeyEvent):void
openClicked(ActionEvent):void
ExportFunctions(ActionEvent):void
saveCanvas(ActionEvent):void
aboutClicked(ActionEvent):void
newDrawing(ActionEvent):void
rotateImage():void
exitClicked(ActionEvent):void


<<Java Class>>
FunctionExpression
Commands
cmds: List<Command>
vars: ArrayList<Token>
split(List<Token>,TokenTypes):ArrayList<Token>
FunctionExpression(String,List<Token>)
execute(Poloska):void
<<Java Class>>
VariableNode
Nodes
VariableNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
LoopExpression
Commands
number: ASTNode
commands: ArrayList<Command>
variableName: String
findCurly(List<Token>):int
LoopExpression(List<Token>)
execute(Poloska):void
<<Java Class>>
FunctionStorage
variables
FunctionStorage()
getInstance():FunctionStorage
AddFunction(Function):void
RemoveFunction(Function):void
RemoveFunctionByName(String):void
getList():List<Function>
GetFunction(String):Function
HasFunction(String):boolean
Save(String):void
Load(String):void
-varStr
0..1
<<Java Class>>
VariableStorage
variables
variables: Map<String,Double>
loopNumber: int
funcloopNum: int
inFunc: boolean
funcName: String
VariableStorage()
getInstance():VariableStorage
addVariable(String,Double,boolean):void
getVariable(String):double
removeVariable(String,boolean):void
list():void
giveLoopName():String
updateVariable(String,Double):void
clear():void
setInFunc(boolean,String):void
~var
0..1
Eljárások+Változók
-single_instance
0..1
-single_instance
0..1
-functions
-func
0..1
<<Java Class>>
Function
Function
variables: List<String>
name: String
commandStr: String
Function(String,List<String>,String)
<<Java Class>>
AddNode
Nodes
AddNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
VariableNode
Nodes
VariableNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
DivisionNode
Nodes
DivisionNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
RandomNode
Nodes
RandomNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
ASTNode
Nodes
value: String
evaluate():double
ASTNode(String,ASTNode,ASTNode)
#leftNode
#rightNode
0..1
0..1
<<Java Class>>
NumberNode
Nodes
NumberNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
MultiplyNode
Nodes
MultiplyNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
PowerNode
Nodes
PowerNode(String,ASTNode,ASTNode)
evaluate():double
<<Java Class>>
SubstractNode
Nodes
SubstractNode(String,ASTNode,ASTNode)
evaluate():double
ASTNodes

Szöveg értelmezése