



## داکیومنت پروژه ی کامپایلر

گروه ۴

اعضای گروه:

شفایق مبشر ۹۷۵۲۲۰۰۴، پریسا علانی ۹۷۵۲۲۱۷۵، ستایش کولوبندی ۹۷۵۲۲۲۰۲

## کد کلی برای تمام قسمت‌ها (main.py)

در این قسمت ابتدا دیتابیس پایه ساخته می‌شود و سپس روی تمام فایل‌های جاوای درون آدرس فولدر داده شده (path) برای هر کدام از رفرنس‌های پروژه یک حلقه می‌زند. به ازای هر کدام از فایل‌ها در این حلقه ابتدا entity فایل برگردانده می‌شود (getFileEntity in Project class) و سپس listener مربوط به رفرنس ساخته می‌شود و در متد ParseAndWalk آن listener و آدرس فایل برای ساخت FileStream داده می‌شود. این متد درخت را walk کرده و listener با لیستی از خصوصیات رفرنس‌های پیدا شده در فایل پر می‌شود. سپس متد مخصوص به آن رفرنس برای پر کردن دیتابیس صدا زده می‌شود. همچنین در اینجا error handling صورت گرفته تا اگر یک فایل به مشکل خورد بتواند به اجرا ادامه دهد و دیگر فایل‌ها را پردازش کند.

## یافتن فیلدهای مربوط به یک کلاس یا اینترفیس (class\_properties.py)

```
44     def enterClassDeclaration(self, ctx:JavaParserLabeled.ClassDeclarationContext):
45         if self.class_properties: # already found the class
46             return
47         if self.class_longname[-1] == ctx.IDENTIFIER().getText():
48             if self.checkParents(ctx):
49                 # this is the exact class we wanted.
50                 self.class_properties = {}
51                 self.class_properties["name"] = self.class_longname[-1]
52                 self.class_properties["longname"] = ".".join(self.class_longname)
53
54                 if len(self.class_longname) == 1:
55                     self.class_properties["parent"] = None
56                 else:
57                     self.class_properties["parent"] = self.class_longname[-2]
58                 self.class_properties["modifiers"] = ClassPropertiesListener.findClassOrInterfaceModifiers(ctx)
59                 self.class_properties["contents"] = ctx.getText()
60
```

```

62 class InterfacePropertiesListener(JavaParserLabeledListener):
63     interface_longname = []
64     interface_properties = None
65
66     def checkParents(self, c):
67         return set(ClassPropertiesListener.findParents(c)) & set(list(reversed(self.interface_longname)))
68
69     def enterInterfaceDeclaration(self, ctx:JavaParserLabeled.InterfaceDeclarationContext):
70         if self.interface_properties: # already found the interface
71             return
72         if self.interface_longname[-1] == ctx.IDENTIFIER().getText():
73             if self.checkParents(ctx):
74                 # this is the exact class we wanted.
75                 self.interface_properties = {}
76                 self.interface_properties["name"] = self.interface_longname[-1]
77                 self.interface_properties["longname"] = ".".join(self.interface_longname)
78
79                 if len(self.interface_longname) == 1:
80                     self.interface_properties["parent"] = None
81                 else:
82                     self.interface_properties["parent"] = self.interface_longname[-2]
83                 self.interface_properties["modifiers"] = ClassPropertiesListener.findClassOrInterfaceModifiers(ctx)
84                 self.interface_properties["contents"] = ctx.getText()
85
86

```

برای یافتن فیلدهای یک کلاس یا اینترفیس که در listener مربوط به آن تنها long name آن در دسترس بود از این دو listener استفاده می‌شود. (برای رفرنس‌های Create/CreateBy و Implement/ImplementBy) این دو listener تفاوت زیادی باهم ندارند، تنها به دلیل متفاوت بودن rule‌های مورد نیازشان (که ClassDeclaration باشد یا InterfaceDeclaration) این دو کلاس ClassPropertiesListener و InterfacePropertiesListener جدا ساخته شده‌اند. به همین دلیل تنها به شرح ClassPropertiesListener پرداخته شده است.

برای استفاده از listener ابتدا longname کلاس به صورت یک لیست که از استرینگ نام بالاترین parent تا نام خود کلاس را دارد داده می‌شود. همچنین یک class\_properties نیز تعریف شده است که در صورتی که کلاس مورد نظر پیدا شود آن متغیر با یک دیکشنری از فیلدهای موردنیاز انتی کلاس پر خواهد شد. در این کلاس تنها یک rule بررسی شده است: classDeclaration.

توضیح enterClassDeclaration: با رسیدن به هر class declaration در درخت پارسر ابتدا چک می‌شود که در صورت پیدا شدن کلاس درست تا آن زمان ادامه متد اجرا نشود. اما اگر class\_properties هنوز None باشد ابتدا نام کلاسی که به آن رسیده است (IDENTIFIER) با نام کلاس موردنظر که آخرین المان class\_longname است مقایسه می‌شود. اگر نامشان برابر باشد باید چک شود که آیا parent‌های این کلاس با کلاس موردنظر یکی است یا نه، به این دلیل که چند کلاس ممکن است با نام‌های یکسان در فایل موجود باشند، پس لازم است کلاسی که پرنتهایش با پرنتهای کلاس موردنظر یکسان باشد را بیابیم. پرنتهای کلاس را به کمک متد استاتیک findParents پیدا می‌کنیم که در قسمتی جدا توضیح داده شده است.

در صورتی که هم نام هم parent‌های کلاس یکی باشد باید فیلدهای مورد نیاز برای ساخت انتیتی را پیدا کند. name, longname و parent از longname داده شده به دست می‌آیند. برای گرفتن modifierهای کلاس (مثلا

public یا static بودن) از متد استاتیک findClassOrInterfaceModifiers استفاده کردم. این modifier برای پیدا کردن kind مرتبط با کلاس استفاده می‌شود. همچنین با گرفتن کل متن class declaration قسمت contents انتی را به دست آوردیم.

## یافتن parentهای یک entity به کمک متد findParents

```
17 @staticmethod
18 def findParents(c): # includes the ctx identifier
19     parents = []
20     current = c
21     while current is not None:
22         if type(current).__name__ == "ClassDeclarationContext" or type(current).__name__ == "MethodDeclarationContext" \
23             or type(current).__name__ == "EnumDeclarationContext" \
24             or type(current).__name__ == "InterfaceDeclarationContext" \
25             or type(current).__name__ == "AnnotationTypeDeclarationContext":
26             parents.append(current.IDENTIFIER().getText())
27             current = current.parentCtx
28     return list(reversed(parents))
```

این متد context رول مربوط به آن انتی را می‌گیرد و روی contextهای parent آن حلقه می‌زند. در این حلقه اگر contextهای پیدا شده مربوط به رولهای EnumDeclaration، ClassDeclaration، InterfaceDeclaration و یا AnnotationTypeDeclaration باشد که همه درون خود یک IDENTIFIER (نام آن کلاس / annotation type / interface / enum است) را دارند، متن IDENTIFIER به عنوان parent به لیست اضافه می‌شود. در پایان چون parentها از خود کلاس شروع شده و از پایین به بالاست، این لیست reverse شده است.

## یافتن modifierهای یک entity کلاس یا interface یا enum به کمک متد findClassOrInterfaceModifiers

```
30 @staticmethod
31 def findClassOrInterfaceModifiers(c):
32     m = ""
33     modifiers=[]
34     current = c
35     while current is not None:
36         if "typeDeclaration" in type(current.parentCtx).__name__:
37             m=(current.parentCtx.classOrInterfaceModifier())
38             break
39         current = current.parentCtx
40     for x in m:
41         modifiers.append(x.getText())
42     return modifiers
```

با دقت در رول type declaration می‌یابیم که تمام modifierهای انواع انتی‌های مورد نیاز ما برای پروژه می‌تواند با رسیدن به این رول و گرفتن classOrInterfaceModifier به دست آید. به همین دلیل مانند قبل parentهای این context را بالا رفته تا به یک type declaration برسد. سپس modifierهای درون آن به لیست modifiers اضافه شده است.

## یافتن یا ایجاد مدل انتی‌ها در کلاس Project (فایل main.py):

### انتی فایل (getFileEntity):

```
42     def getFileEntity(self, path):
43         # kind id: 1
44         path = path.replace("/", "\\")
45         name = path.split("\\")[-1]
46         file = open(path, mode='r')
47         file_ent = EntityModel.get_or_create(_kind=1, _name=name, _longname=path, _contents=file.read())[0]
48         file.close()
49         print("processing file:",file_ent)
50         return file_ent
```

این متد فیلدهای نام و longname انتی فایل را با استفاده از آدرس آن به دست می‌آورد و همچنین با open کردن آن contents درونش را خوانده و تمام این فیلدها را علاوه kind id آن (که در دیتابیس ۱ است) برای گرفتن یا ساخت انتی فایل به کار می‌برد.

### انتی پکیج (getPackageEntity):

```
102     def getPackageEntity(self, file_ent, name, longname):
103         # package kind id: 72
104         ent = EntityModel.get_or_create(_kind= 72, _name=name, _parent=file_ent,
105                                         _longname=longname, _contents="")
106         return ent[0]
```

فیلدهای پکیج عبارت است از kind آن، نام و parent و longname. همچنین contents آن استرینگ خالی در نظر گرفته می‌شود.

### انتی پکیج بدون نام (getUnnamedPackageEntity):

```
108     def getUnnamedPackageEntity(self, file_ent):
109         # unnamed package kind id: 73
110         ent = EntityModel.get_or_create(_kind= 73, _name="(Unnamed_Package)", _parent=file_ent,
111                                         _longname="(Unnamed_Package)", _contents="")
112         return ent[0]
```

فرق این متد با متد بالا در id kind و نام و longname است که دو مورد آخر (Unnamed Package) در نظر گرفته می‌شوند.

## اننتی کلاس (getClassEntity):

```
def getClassEntity(self, class_longname, file_address):
    props = p.getClassProperties(class_longname, file_address)
    if not props: # This class is unknown, unknown class id: 84
        ent = EntityModel.get_or_create(_kind=84, _name=class_longname.split(".")[1],
                                         _longname=class_longname, _contents="")
    else:
        if len(props["modifiers"]) == 0:
            props["modifiers"].append("default")
        kind = self.findKindWithKeywords("Class", props["modifiers"])
        ent = EntityModel.get_or_create(_kind=kind, _name=props["name"],
                                         _longname=props["longname"],
                                         _parent= props["parent"] if props["parent"] is not None else file_ent,
                                         _contents=props["contents"])

    return ent[0]
```

برای گرفتن اننتی یک کلاس به کمک آدرس فایل و نام کامل (longname) کلاس از این متد استفاده می‌شود. ابتدا به کمک این دو ClassPropertiesListener تلاش می‌کند این کلاس را در فایل پیدا کند. اگر نتواند این کار را بکند یعنی این کلاس در فایل تعریف نشده و لازم است اننتی از نوع unknown class باشد. در غیر این صورت باید نوع دقیق کلاس پیدا شود. برای این کار از متد findKindWithKeywords استفاده شده است که نوع اننتی ("Class") و modifierهای آن را میگیرد و مناسبترین kind را بازمیگرداند. سپس اننتی کلاس به کمک get\_or\_create به دست می‌آید. اگر در دیکشنری parent نداشته باشد یعنی parent آن خود فایل است.

## اننتی اینترفیس (getInterfaceEntity):

```
150 def getInterfaceEntity(self, interface_longname, file_address): # can't be of unknown kind!
151     props = p.getInterfaceProperties(interface_longname, file_address)
152     if not props:
153         return None
154     else:
155         kind = self.findKindWithKeywords("Interface", props["modifiers"])
156         ent = EntityModel.get_or_create(_kind=kind, _name=props["name"],
157                                         _longname=props["longname"],
158                                         _parent= props["parent"] if props["parent"] is not None else file_ent,
159                                         _contents=props["contents"])
160     return ent[0]
```

برای گرفتن اننتی اینترفیس به کمک آدرس فایل و نام کامل (longname) اینترفیس از این متد استفاده می‌شود. ابتدا به کمک این دو InterfacePropertiesListener تلاش می‌کند آن را در فایل پیدا کند. اگر نتواند این کار را بکند از آنجا که اینترفیس از نوع unknown نداریم None برمیگرداند تا متدی که این متد را صدا زده به جای اینترفیس آن را کلاس در نظر بگیرد و اگر جایی تعریف نشده بود unknown class برگرداند. در غیر این صورت باید نوع دقیق اینترفیس پیدا شود. برای این کار از متد findKindWithKeywords استفاده شده است که نوع اننتی ("Interface") و modifierهای آن را میگیرد و مناسبترین kind را بازمیگرداند. سپس اننتی اینترفیس

به کمک `get_or_create` به دست می‌آید. اگر در دیکشنری `parent` نداشته باشد یعنی `parent` آن خود فایل است.

## متد کمکی `findKindWithKeywords` (فایل `main.py`):

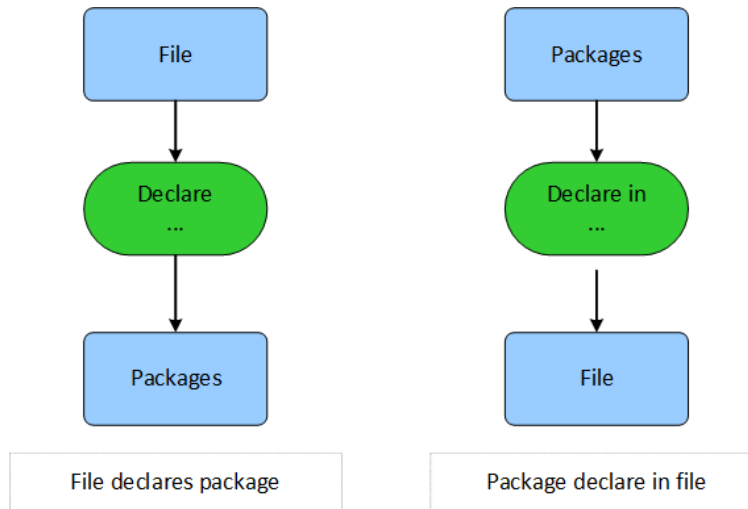
```
168     def findKindWithKeywords(self, type, modifiers):
169         if len(modifiers) == 0:
170             modifiers.append("default")
171         leastspecific_kind_selected = None
172         for kind in KindModel.select().where(KindModel._name.contains(type)):
173             if self.checkModifiersInKind(modifiers, kind):
174                 if not leastspecific_kind_selected \
175                     or len(leastspecific_kind_selected._name) > len(kind._name):
176                     leastspecific_kind_selected = kind
177         return leastspecific_kind_selected
178
179
180     def checkModifiersInKind(self, modifiers, kind):
181         for modifier in modifiers:
182             if modifier.lower() not in kind._name.lower():
183                 return False
184         return True
```

در این متد `type` منظور نوع اصلی (مثلا "Class" یا "Interface" یا "Enum" بودن نوع) و `modifiers` خصوصیات فرعی نوع (مثلا پابلیک بودن یا استاتیک بودن) را دارند. اگر `modifiers` خالی باشد یک `default` به آن اضافه میکنیم که نوع دیفالت آن پیدا شود.

سپس بین تمام `kind`هایی که درونشان `type` را دارند چک میکنیم و بین آن‌هایی که `modifier`ها را هم در نامشان دارند `kind` با کوتاهترین نام را برمی‌گردانیم که احتمالاً خصوصیات اضافی که در `modifiers` نیامده را ندارد. به طور مثال اگر `modifiers` تنها `public` داشته باشد و به دنبال یک کلاس باشیم، بین دو نوع `Java Abstract Class Type Public Member` و `Java Class Type Public Member` آن که کوتاه تر است را انتخاب می‌کند که نوع مورد نظر ماست.

## رفرنس‌های `declare` و `declarein` (فایل `declare_declarein.py`)

رفرنس‌های `declare` با تعریف پکیج‌ها به وجود می‌آیند. تمام فایل‌ها یک پکیج را `declare` می‌کنند، و اگر نام پکیج در ابتدای فایل ذکر نشود این پکیج به صورت `unnamed package` به انتی‌ها اضافه می‌شود. همچنین هر پکیج پکیجی که در ادامه نام آن آمده و با نقطه جدا شده است را `declare` می‌کند.



↓ F

```
DependencyActivityBindingImpl.java X
{} ... v
1  ▾  /*
2      * Created on 24.10.2004
3      */
4  package net.sourceforge.ganttproject.task.dependency.constraint; ← P
5
6  import java.util.Date;
7
8  import net.sourceforge.ganttproject.task.TaskActivity;
9  import net.sourceforge.ganttproject.task.dependency.TaskDependency;
10 import net.sourceforge.ganttproject.task.dependency.TaskDependencyConstraint;
11
12 ▾  /**
13     * @author bard
14     */
15 ▾  class DependencyActivityBindingImpl implements TaskDependency.ActivityBinding{
16
```

R



```

23 class DeclareAndDeclareinListener(JavaParserLabeledListener):
24     """
25     #Todo: Implementing the ANTLR listener pass for Java Call and Java Callby reference kind
26
27     """
28     declare = []
29
30     def enterCompilationUnit(self, ctx:JavaParserLabeled.CompilationUnitContext):
31         if not ctx.packageDeclaration(): # unnamed package
32             self.declare.append({
33                 "scope": None, "ent": None,
34                 "line": 1, "col": 0
35             })
36
37     def enterPackageDeclaration(self, ctx:JavaParserLabeled.PackageDeclarationContext):
38         all_declared = ctx.qualifiedName().IDENTIFIER()
39         longname = ""
40         for i in range(len(all_declared)):
41             ent_name = all_declared[i].getText()
42             ent_longname = longname + ( "." if longname != "" else "" ) + ent_name
43             self.declare.append({
44                 "scope": all_declared[i-1].getText() if i != 0 else None, "ent": ent_name,
45                 "scope_longname": longname, "ent_longname": ent_longname,
46                 "line": all_declared[i].symbol.line, "col": all_declared[i].symbol.column
47             })
48             longname = ent_longname

```

به این ترتیب دو rule در listener این فایل استفاده شده است. یکی CompilationUnit که چک شود اگر در این فایل نام پکیجی آورده نشده، تنها یک رفرنس با خط و ستون ۱ و ۰ اضافه کند. (در اینجا که اسکوپ فایل است None تعریف شده اما در main خود انتی فایل به جای آن قرار می‌گیرد. همچنین ent که منظور انتی رفرنس است در main کامل می‌شود)

اگر package declaration وجود داشته باشد، IDENTIFIERهای درون qualifiedName آن تک تک پکیج‌ها را دارند. پس با حلقه زدن روی آن باید ۱. اگر اولین پکیج باشد یک رفرنس بین آن و فایل ایجاد کنیم و ۲. اگر پکیج‌های بعدی باشد بین آن و پکیج قبلی رفرنس را بسازیم. پس scope فایل یا پکیج قبلی (که اگر فایل باشد در این دیکشنری رفرنس None گذاشته می‌شود) و entity رفرنس، آن پکیج است. برای گرفتن خط و ستون رفرنس از خط و ستون خود پکیج استفاده شده است. برای longname از استرینگ پکیج اول تا آن پکیج استفاده می‌شود.

## استفاده از Declare and Declarein Listener در فایل main.py:

```
52 def addDeclareRefs(self, ref_dicts, file_ent):
53     for ref_dict in ref_dicts:
54         if ref_dict["scope"] is None: # the scope is the file
55             scope = file_ent
56         else: # a normal package
57             scope = self.getPackageEntity(file_ent, ref_dict["scope"], ref_dict["scope_longname"])
58
59         if ref_dict["ent"] is None: # the ent package is unnamed
60             ent = self.getUnnamedPackageEntity(file_ent)
61         else: # a normal package
62             ent = self.getPackageEntity(file_ent, ref_dict["ent"], ref_dict["ent_longname"])
63
64         # Declare: kind id 192
65         declare_ref = ReferenceModel.get_or_create(_kind=192, _file=file_ent, _line=ref_dict["line"],
66                                                    _column=ref_dict["col"], _ent=ent, _scope=scope)
67
68         # Declarein: kind id 193
69         declarein_ref = ReferenceModel.get_or_create(_kind=193, _file=file_ent, _line=ref_dict["line"],
70                                                    _column=ref_dict["col"], _scope=ent, _ent=scope)
```

برای اضافه کردن رفرنس‌های پیدا شده که به صورت دیکشنری هستند به دیتابیس از addDeclareRefs استفاده شده است. این متد برای هر دیکشنری از declareهای پیدا شده به صورت زیر عمل می‌کند:

اگر scope از نوع NoneType باشد آن را به عنوان فایل تلقی کرده و اگر نه به کمک اطلاعات داده شده در دیکشنری پکیج انتیتی آن را می‌سازد / می‌گیرد.

در صورتی که انتیتی خود رفرنس None باشد آن را به عنوان پکیج unnamed تلقی می‌کند و در غیر این صورت آن را یک پکیج عادی در نظر می‌گیرد و انتیتی آن را می‌سازد / می‌گیرد.

سپس برای ساخت رفرنس‌ها در دیتابیس، ابتدا آی دی kind آن را به صورت دستی (که با استفاده از برنامه db browser به آن دست پیدا کرده ایم) وارد می‌کنیم و سپس دیگر اطلاعاتی که تا اینجا جمع‌آوری کرده ایم را به آن اضافه می‌کنیم.

رفرنس declareIn تنها scope و entity برعکس است و آی دی kind آن یکی بعد از declare است.

## رفرنس‌های implementCouple و implementbyCoupleby (implementCouple\_implementbyCoupleby.py)

این رفرنس زمانی اتفاق می‌افتد که یک class یا enum می‌آید از implement استفاده می‌کند و یک اینترفیس یا کلاس را پیاده‌سازی می‌کند.

در این قسمت نیاز است که در ابتدا JavaParserLabeled و JavaParserLabeledListener را از فایل‌های گرامر جاوا در این متد ایمپورت کنیم.

در این فایل ما نیاز داریم که IMPLEMENTS ها را بدست بیاوریم. وقتی این استرینگ را در فایل های متعلق به جاوا گشتیم و در ClassDeclaration و EnumDeclaration آن را یافتیم. پس نیاز است که برای هردو آن ها IMPLEMENTS را چک کنیم و موارد خواسته شده را بیابیم.

```
class ImplementCoupleAndImplementByCoupleBy(JavaParserLabeledListener):
    """
    #Todo: Implementing the ANTLR listener pass for Java Call and Java C
    """

    implement = []
```

در این قسمت یک لیست تعریف کردیم که همه ی موارد خواسته شده را در implement میریزیم.

```
def enterClassDeclaration(self, ctx:JavaParserLabeled.ClassDeclarationContext):
    if ctx.IMPLEMENTS():
        scope_parents = class_properties.ClassPropertiesListener.findParents(ctx)
        if len(scope_parents) == 1:
            scope_longname = scope_parents[0]
        else:
            scope_longname = ".".join(scope_parents)

        print("scope_longname list nabash",scope_longname)
        [line, col] = str(ctx.start).split(",")[3].split(":")
        for myType in ctx.typeList().typeType():
            if myType.classOrInterfaceType():
                myType_longname = ".".join([x.getText() for x in myType.classOrInterfaceType().IDENTIFIER()])
                self.implement.append({"scope_kind": "Class", "scope_name": ctx.IDENTIFIER().__str__(),
                                      "scope_longname": scope_longname,
                                      "scope_parent": scope_parents[-2] if len(scope_parents) > 2 else None,
                                      "scope_contents": ctx.getText(),
                                      "scope_modifiers":
                                          class_properties.ClassPropertiesListener.findClassOrInterfaceModifiers(ctx),
                                      "line": line,
                                      "col": col[:-1],
                                      "type_ent_longname": myType_longname})
```

در این قسمت یک تابع صدا زدیم که وارد classDeclaration می شود و موارد مربوط به آن را بررسی می کند. در ابتدا چک میکنیم که اگر در ctx کلمه ی implement باشد، کارهای لازم را انجام دهد. در خط بعد برای scope ها نیاز داریم که parents آن ها را بیابیم. با کمک تابعی که در فایل class\_properties و در کلاس ClassPropertiesListener و در متد static ای به نام findParents، همه ی parent های مربوط به این scope در scope\_parents ذخیره می شود.

در ادامه نیاز است که برای `scope longname` ها را نیز بدست آوریم که این کار در متد `findParents`، به ما برمی گردد. اگر تعداد خانه های `scope_parents` فقط یک باشد، `Scope longname` همان یک خانه میشود و در غیر این صورت همه ی خانه ها را یک '.' بهم وصل می کنیم و آن را به `scope longname` می دهیم. برای گرفتن خط و ستون از `ctx.start` استفاده کردیم و جوابی را که می داد `split` و خانه هایی که نیاز داشتیم را برداشتیم.

چیزی که `ClassDeclaration` به می دهد، همه ی `implement` ها است که با کما از هم جدا شده اند پس نیاز داریم که یک `loop` بنویسیم که در `typelist` هایی که دارد و `typetype` های آن ها بچرخد و `type` هایی که مربوط به `classOrInterfaceType` را بدست آوریم. `Loop` زده شده و شرط اولیه آن برای همین توضیحات داده شده است.

در ادامه برای `entity` هایی که داریم در این `loop` باید `longname` ها را بیابیم که میشود همه ی موجودیت هایی که در `myType.classOrInterfaceType` است که با `join` با "." به هم وصل شده اند.

در آخر همه ی مواردی را که پیدا کرده ایم را باید در لیستی که در بالا زده بودیم، اضافه کنیم. در اینجا `scope kind` برابر `class` است چون ما در `ClassDeclaration` هستیم و اسم `scope` نیز میشود استرینگ `ctx.IDENTIFIER()` و `scopelongname` که در بالا بدست آوردیم و ذخیره کردیم را برای `scope_longname` می دهیم و برای `parent` این `scope` نیاز است که خانه ی ۲- لیست `scope_parents` را بدهیم چون خانه ی آخر خود `scope` است، این زمانی درست است که طول این لیست بیشتر از ۲ باشد در غیر این صورت برای این `scope parent` آن `none` است. برای پیدا کردن `modifiers` های این `scope` از فایل `class_properties` و از کلاس `ClassPropertiesListener` و از متد `static` ای به نام `findClassOrInterfaceModifiers` استفاده کرده ایم که در بالا توضیح آن داده شده است. در ادامه برای `line` مقدار بدست آورده شده در بالا را داده ایم و برای `col` تا خانه ی یکی مانده به آخر چیزی که بدست آوردیم زیرا یک "]" بیشتر داشت و ما به آن نیازی نداشتیم. و در آخر برای `type_ent_longname`، مقدار ذخیره شده ی `myType_longname` را داده ایم.

و به پایان این متد می رسیم.

برای متد `enterEnumDeclaration` درست مثل بالا عمل کرده ایم با دو تفاوت در کد :

۱. از `ctx:JavaParserLabeled.EnumDeclarationContext` که مربوط به `enum` است استفاده شده است.

۲. `scope_kind": "Enum"` است چون در قسمت `enum` هستیم.

```

def enterEnumDeclaration(self, ctx:JavaParserLabeled.EnumDeclarationContext):
    if ctx.IMPLEMENTS():
        scope_parents = class_properties.ClassPropertiesListener.findParents(ctx)
        if len(scope_parents) == 1:
            scope_longname = scope_parents[0]
        else:
            scope_longname = ".".join(scope_parents)

    [line, col] = str(ctx.start).split(",")[3].split(":") # line, column
    for myType in ctx.typeList().typeType():
        if myType.classOrInterfaceType():
            myType_longname = ".".join([x.getText() for x in myType.classOrInterfaceType().IDENTIFIER()])
            self.implement.append({"scope_kind": "Enum", "scope_name": ctx.IDENTIFIER().__str__(),
                                   "scope_longname": scope_longname,
                                   "scope_parent": scope_parents[-2] if len(scope_parents) > 2 else None,
                                   "scope_contents": ctx.getText(),
                                   "scope_modifiers":
                                       class_properties.ClassPropertiesListener.findClassOrInterfaceModifiers(
                                           ctx),
                                   "line": line,
                                   "col": col[:-1],
                                   "type_ent_longname": myType_longname})

```

## استفاده از Listener implement and implementBy در فایل :main.py

```

# implement:
for file_address in files:
    file_ent = p.getFileEntity(file_address)
    listener = ImplementCoupleAndImplementByCoupleBy()
    listener.implement = []
    p.ParseAndWalk(listener, file_address)
    p.addImplementOrImplementByRefs(listener.implement, file_ent, file_address)

```

در این جا با توجه به توضیحات بالا و موارد توضیح داده شده است، عمل می کنیم و در آخر متد addImplementOrImplementByRefs را صدا می زنیم.

```

def addImplementOrImplementByRefs(self, ref_dicts, file_ent, file_address):
    for ref_dict in ref_dicts:

        scope = EntityModel.get_or_create(_kind=self.findKindWithKeywords(ref_dict["scope_kind"],
                                                                           ref_dict["scope_modifiers"]),
                                          _name=ref_dict["scope_name"],
                                          _parent= ref_dict["scope_parent"] if ref_dict["scope_parent"] is not None else file_ent,
                                          _longname=ref_dict["scope_longname"],
                                          _contents=ref_dict["scope_contents"])[0]
        ent = self.getImplementEntity(ref_dict["type_ent_longname"], file_address)

        implement_ref = ReferenceModel.get_or_create(_kind=188, _file=file_ent, _line=ref_dict["line"],
                                                    _column=ref_dict["col"], _ent=ent, _scope=scope)
        implementBy_ref = ReferenceModel.get_or_create(_kind=189, _file=file_ent, _line=ref_dict["line"],
                                                    _column=ref_dict["col"], _ent=scope, _scope=ent)

```

در اینجا برای رفرنس هایی که داریم یک scope، entity، یک implement reference و implementBy می سازیم.

در ابتدا برای ساختن scope، از EntityModel با فیلدهایی که دارد یک مدل می سازیم. نحوه ی پرکردن فیلد : برای kind از متد findKindWithKeywords استفاده کردیم که در بالا توضیح آن داده شده است. برای name، longname، contents از موارد ذکر شده در بالا که در یک لیست ذخیره کرده بودیم استفاده کردیم. برای قسمت parents نیز همینطور با این تفاوت که یک شرط برای آن قرار داده ایم که اگر این scope هیچ parent ای نداشت، parent آن اسم فایل اصلی جاوا می شود. در آخر متد get\_or\_create به ما یک آرایه برمی گرداند که ما خانه ی صفر آن را نیاز داریم.

برای ساختن entity متد getImplementEntity را صدا زده ایم که در آن طبق عکس کد پایین، دو حالت برای entity داریم ، یا class است یا interface است. در ابتدا ent را برابر خروجی متد getInterfaceEntity میگذاریم که در صورتی که entity آن interface باشد مقدار آن را به ent می دهیم ولی اگر از نوع interface نباشد مقدار ent برابر none است و وارد ایف میشود و با کمک متد getClassEntity مقدار آن داده می شود و در آخر مقدار مقدار ent را برمی گردانیم. (دو متد getClassEntity و getInterfaceEntity در بالا توضیح داده شده اند.)

```

def getImplementEntity(self, longname, file_address):
    ent = self.getInterfaceEntity(longname, file_address)
    if not ent:
        ent = self.getClassEntity(longname, file_address)
    return ent

```

در ادامه باید refrence هایی که برای implement، implementBy هستند را بسازیم . نحوه ی مقدار دهی پارامترها برای implement reference :

برای مقدار kind از عدد ثابت ۱۸۸ استفاده کردیم که این عدد را بررسی دیتابیس با برنامه DB Browser پیدا کردیم و مقدار file را مقدار پاس داده شده و بدست آمده که در بالا توضیح داده ایم می گذاریم برای line و

column از دو مقداری که در ابتدا توضیح داده ام استفاده میکنیم و در آخر برای مقدار scope, ent از دو مقدار بالایی که بدست آوردیم، استفاده می کنیم.

برای ساختن refrence برای implementBy مثل بالا عمل می کنیم با سه تفاوت :

برای عدد kind عدد ثابت ۱۸۹ را داده ایم و مقادیر scope و ent را باهم عوض می کنیم .

## رفرنس های Create , Createby ( فایل create\_createby.py )

create , created by تنها داخل یک متد زمانی که یک object از یک کلاس ساخته می شود اتفاق می افتد. برای مثال : `a = new box ()`

داخل این فایل listener خود انتل را extend میکنیم که بتوانیم از متد هایش استفاده کنیم. آرایه create را برای ذخیره ی همه ی اطلاعات ساختیم .

زمانی که داخل فایل tests.py تست کردیم که ببینیم وقتی create اتفاق می افتد understand چه چیزهایی را به ما نشان می دهد دیدیم که داخل understand برای create به modifier (public private) نیاز داریم که این را با استفاده از متد find method access به دست آوردیم (برای پیدا کردن modifier باید به بالا برویم تا به rule ای برسیم که class body Declaration باشد و سپس از روی آن modifier ها را به دست بیاوریم)

```
def findmethodreturntype(self, c):
    parents = ""
    context = ""
    current = c
    while current is not None:
        if type(current.parentCtx).__name__ == "MethodDeclarationContext":
            parents=(current.parentCtx.typeTypeOrVoid().getText())
            context=current.parentCtx.getText()
            break
        current = current.parentCtx
    return parents_context
```

از طرفی به تاییپی که متدی که داخلش new اتفاق افتاده نیاز داریم یعنی void یا int .. برای به دست آوردن این موضوع متد find method return types را پیاده سازی کردیم که البته این متد contents را هم به عنوان خرجی برمیگرداند. ( برای به دست آوردن تاییپی که یک متد برمیگرداند باید بین rule ها بالا می رفتیم تا به rule ای برسیم که \_\_name\_\_ آن methoddeclarationcontext باشد چون داخل rule method declaration context در واقع داخل rule methoddeclaration (وقتی از \_\_name\_\_ استفاده میکنیم کلمه ی context به آخر نام rule اضافه می شود). و سپس typeTypeorVoid. تاییپی که آن متد ریترن میکند را برمی گرداند.

```
def findmethodaccess(self, c):
    parents = ""
    modifiers=[]
    current = c
    while current is not None:
        if "ClassBodyDeclaration" in type(current.parentCtx).__name__:
            parents=(current.parentCtx.modifier())
            break
        current = current.parentCtx
    for x in parents:
        if x.classOrInterfaceModifier():
            modifiers.append(x.classOrInterfaceModifier().getText())
    return modifiers
```

شرطی که بعد از آن گذاشتیم یعنی شرط `ctx.creator().classCreatorRest` برای این است که `new` فقط مربوط به کلاس باشد نه یک آرایه چون `understand` فقط زمانی `create` , `createby` را تشخیص میدهد که `new` فقط برای `class` اتفاق بیفتد.

سپس متد `find parents` را از داخل کلاس `class properties` صدا زدیم و `refent` را آخرین خانه ی آرایه ای که از متد `find parents` گرفتیم قرار دادیم ( یعنی `scope name` ) برای به دست آوردن `longname` روی آرایه ای که از متد گرفتیم که در واقع همه ی `parent` ها داخل آن قرار دارد `join` زدیم و دات گذاشتیم که فرمتمان شبیه فرمت `understand` شود و در آخر لیست `create` را مطابق با `Understand` پر کردیم .

```
create = []

def enterExpression4(self, ctx:JavaParserLabeled.Expression4Context):
    modifiers=self.findmethodaccess(ctx)
    methodedreturn,methodcontext=self.findmethodreturntype(ctx)

    if ctx.creator().classCreatorRest():
        allrefs= class_properties.ClassPropertiesListener.findParents(ctx) #self.findParents(ctx)
        refent=allrefs[-1]
        entlongname=".".join(allrefs)
        [line, col] = str(ctx.start).split(",")[3].split(":")

        self.create.append({"scopeparent":refent,"scopelongname":entlongname,"scopemodifiers":modifiers,
                           "scopereturntype":methodedreturn,"scopecontent":methodcontext,
                           "line":line,"col":col[:1],"refent":ctx.creator().createdName().getText(),
                           "scope_parent": allrefs[-2] if len(allrefs) > 2 else None,
                           "potential_refent":".".join(allrefs[:-1]) + "." + ctx.creator().createdName().getT
```

## استفاده از Listener Create Createby در فایل main.py :

ابتدا با توجه به توضیحات قسمت کد کلی برای تمام قسمتها (main.py) عمل می کنیم.



```
# create
for file_address in files:
    file_ent = p.getFileEntity(file_address)
    listener = CreateAndCreateBy()
    listener.create = []
    p.ParseAndWalk(listener, file_address)
    p.addCreateRefs(listener.create, file_ent, file_address)

# declare
```

سپس برای اضافه کردن رفرنس‌های پیدا شده که به صورت دیکشنری هستند به دیتابیس از addCreateRefs استفاده شده است. این متد برای هر دیکشنری از create های پیدا شده به صورت زیر عمل می‌کند: ابتدا برای scope, ent مدلی در دیتابیس می‌سازیم و آنها را با مقادیر به دست آمده پر می‌کنیم. برای ساخت مدل ent از متد getCreatedClassEntity استفاده می‌کنیم که این متد به صورت زیر عمل می‌کند: ابتدا با استفاده از class\_potential\_longname چک می‌کنیم که ببینیم خصوصیات آن از داخل متد getClassproperties به دست می‌آید یا نه اگر به دست آمد به متد getclassentity خود class\_potential\_longname را می‌دهیم و entity ای را می‌گیریم که با potential long name به دست آمده (در صورتی که class ای که از آن object ای ساخته شده است لوکال باشد) در غیر اینصورت class\_longname را به متد ذکر شده می‌دهیم. (در صورتی که class ای که object ای از آن ساخته شده لوکال نباشد)

```
def getCreatedClassEntity(self, class_longname, class_potential_longname, file_address):
    props = p.getClassProperties(class_potential_longname, file_address)
    if not props:
        return self.getClassEntity(class_longname, file_address)
    else:
        return self.getClassEntity(class_potential_longname, file_address)
```

سپس برای ساخت رفرنس‌ها در دیتابیس، ابتدا آی دی kind آن را به صورت دستی (که با استفاده از برنامه db browser به آن دست پیدا کرده ایم) وارد می‌کنیم و سپس دیگر اطلاعاتی که تا اینجا جمع‌آوری کرده ایم را به آن اضافه می‌کنیم.

رفرنس createby تنها scope و entity اش برعکس است و آی دی kind آن یکی بعد از create است.

```
def addCreateRefs(self, ref_dicts, file_ent, file_address):
    for ref_dict in ref_dicts:
        scope = EntityModel.get_or_create(_kind=self.findKindWithKeywords("Method", ref_dict["scopemodifiers"]),
                                          _name=ref_dict["scopename"],
                                          _type=ref_dict["scopereturntype"],
                                          _parent=ref_dict["scope_parent"] if ref_dict["scope_parent"] is not None else file_ent,
                                          _longname=ref_dict["scope_longname"],
                                          _contents=[ref_dict["scopecontent"]][0])
        ent = self.getCreatedClassEntity(ref_dict["refent"], ref_dict["potential_refent"], file_address)
        Create = ReferenceModel.get_or_create(_kind=190, _file=file_ent, _line=ref_dict["line"],
                                              _column=ref_dict["col"], _scope=scope, _ent=ent)
        Createby = ReferenceModel.get_or_create(_kind=191, _file=file_ent, _line=ref_dict["line"],
                                              _column=ref_dict["col"], _scope=ent, _ent=scope)
```

پایان