Obsah

Okruhy otázok (2017 – 2018)

1. Metaprogramovanie — definícia, použitie, typy metaprogramovania
2. Reflexia — definícia, prostriedky realizácie reflexie, použitie v jazyku Java
3. Zásobník volaní a jeho introspekcia
4. Realizácia tried v jazyku Java a ich načítavanie (ClassLoader)
5. Dynamické proxy v jazyku Java
6. Anotácie — význam a typy metadát v programovom kóde, ich použitie
7. Anotácie v jazyku Java — definícia a použitie anotačných typov pomocou reflexie
8. Spracovanie anotácii počas prekladu — anotačný procesor
9. Princíp inverzie závislosti v architektúre softvéru a vzor Dependency Injection
10. Generovanie kódu — typy generátorov a techniky generovania kódu
11. Vývoj softvéru na základe modelu (MDSD), význam modelu pri generovaní
12. Makrá a ich použitie v programovacích jazykoch
13. Prostriedky metaprogramovania v jazyku Python
14. Systémy typov a ich využitie
15. Aspektovo orientované programovanie — význam, základné vlastností, realizácia v AspectJ
16. AspectJ — body spájania a súvisiace bodové prierezy (kinded pointcuts)
17. AspectJ — non-kinded pointcuts
18. AspectJ — odporučenia
19. AspectJ — statické pretínanie
20. AspectJ — vytváranie inštancií aspektov

1. Metaprogramovanie - intro

1.1. Metaprogramovanie — definícia, použitie, typy metaprogramovania

- **Metaprogramming** is a programming technique in which computer programs have the ability to treat other programs as their data.
- Metaprogramming can be used to move computations from run-time to compile-time, to generate code using compile time computations, and to enable self-modifying code.

Technika pri ktorej programy manipuluju s inymi programami (aj so sebou samymi) ako s datami:

- citanie
- analyzovanie
- generovanie
- transformovanie
-

Metaprogramovanie sa pouziva pre generovanie kodu pocas kompilacie, sebaupravu programov, zlepsenie designu (AOP). Typy metaprogramovania:

- reflexia (skumanie a modifikovanie seba sameho za behu)
- makra (nahrada kodu)
- generovanie kodu pocas kompilacie
- AOP

1.2. Reflexia — definícia, prostriedky realizácie reflexie, použitie v jazyku Java

Schopnost programu za behu skumat, pozorovat a modifikovat svoju strukturu a spravanie.
V Jave sa reflexia pouziva na:

- prehliadanie tried, rozhrani alebo metod
- vytvaranie instancii
- volanie metod
- zmenu pristupnosti metod a premennych

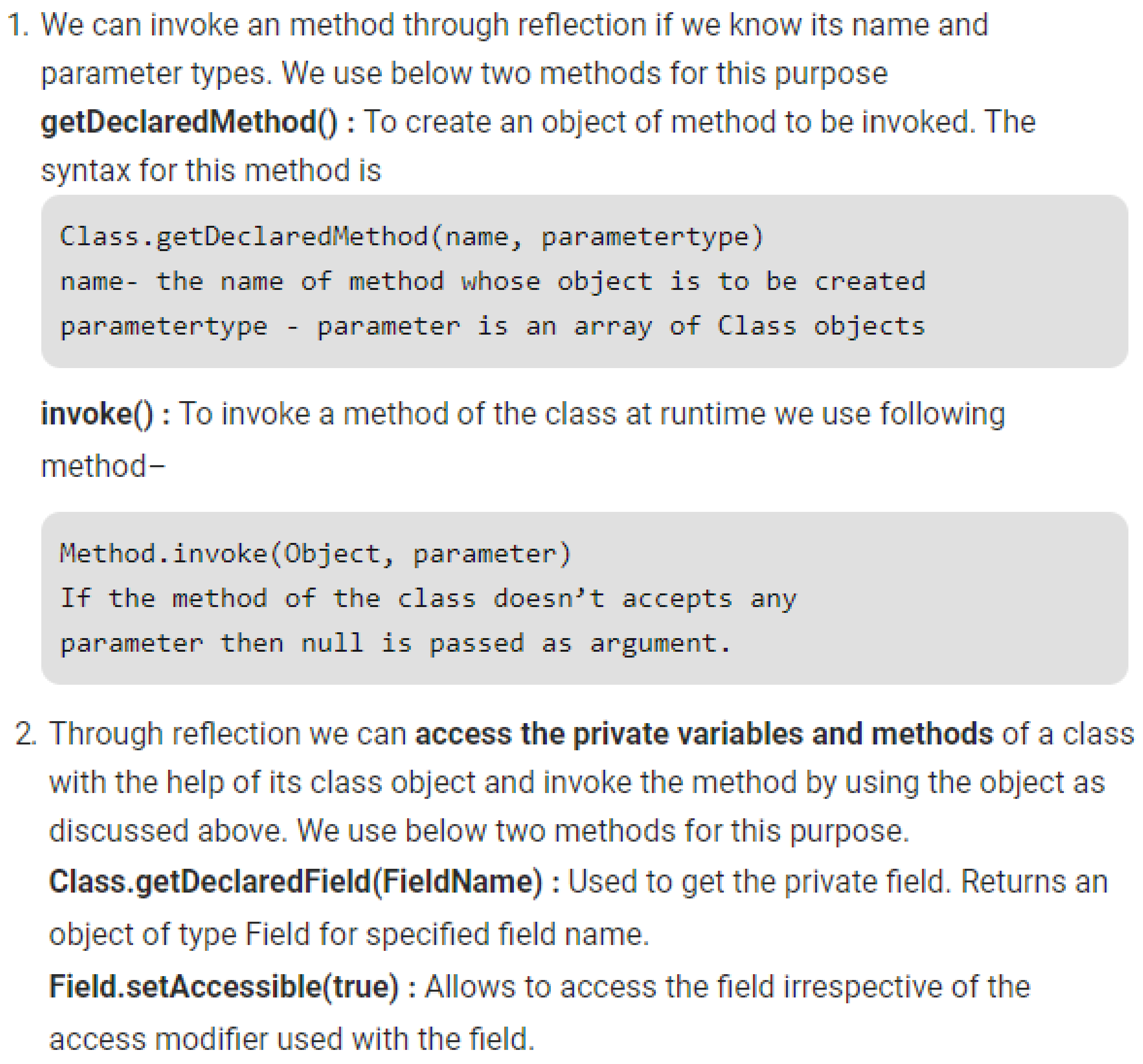
Metody:

- getClass(), getInterfaces(), getDeclaredMethods(), getDeclaredFields(),getConstructors(),getMethods()
- newInstance(),
- invoke()
- setAccessible()

- metatriedy a analyzatory tried ?
Metatriedy:
Metatrieda je trieda, ktorej inštancie sú triedy. Rovnako ako bežná trieda definuje správanie určitých objektov, metatrieda definuje správanie určitých tried a ich inštancií.

Reflexia je API, ktoré sa používa na skúmanie alebo modifikáciu správania metód, tried, rozhraní pri behu. Požadované triedy pre reflexiu sú poskytované v balíku **java.lang.reflect.**
Reflexia nám poskytuje informácie o triede, do ktorej objekt patrí, ako aj o metódach tejto triedy,

ktoré možno vykonať pomocou objektu. Prostredníctvom odrazu môžeme vyvolať metódy za behu bez ohľadu na špecifikátor prístupu, ktorý sa s nimi používa.



1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

   ```
   Class.getDeclaredMethod(name, parametertype)
   name- the name of method whose object is to be created
   parametertype - parameter is an array of Class objects
   ```

   invoke() : To invoke a method of the class at runtime we use following method–

   ```
   Method.invoke(Object, parameter)
   If the method of the class doesn’t accepts any
   parameter then null is passed as argument.
   ```

2. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
   Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
   Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

1.3. Zásobník volaní a jeho introspekcia

Dynamic data structure maintained inside the computers ram by the OS. Its purpose is to control the way procedures and functions call each other and to control the way they pass parameters to each other. Maintained for each task and indeed for each thread

Zasobnik volani (**call stack**) je LIFO zasobnik, ktory obsahuje aktualne vykonavane metody a podmetody. Pri vnoreni sa do metody je metoda pridana do zasobnika, pri odchode z nej je odstranena. Kazda invokacia metody je reprezentovana **stack frame**-om. Na call stack sa mozeme pozerat priamo pri debugovani.

Stack trace je report call stacku z isteho bodu vykonavania programu – typicky zobrazeny pri zlyhani programu. Daju sa z neho citat metody volane pred zlyhanim programu aj konkretny bod (riadok kodu) v programe, kde nastala chyba.

Kazdy frame v nom je reprezentovany objektom **StackTraceElement**. Kazdy frame, okrem toho navrchu stacku reprezentuje volanie metody. Frame navrchu reprezentuje bod vykonavania, kedy bol stack trace vygenerovany. Typicky ide o bod, kedy bola vyhodena vynimka spojena so stack traceom.

1.4. Realizácia tried v jazyku Java a ich načítavanie (ClassLoader)

java.lang.ClassLoader je sucast JRE, ktora dynamicky nacitava triedy do JVM. They are part of JRE. ClassLoader je zodpovedny za najdenie kniznic, precitanie ich obsahu a nacitanie tried v nich.

Hence, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Also, these Java classes aren't loaded into memory all at once, but when required by an application. This is where class loaders come into the picture. They are responsible for loading classes into memory

Trieda je nacitavana bud ked sa vykona new klucove slovo (… = new Class()), alebo ked sa narazi na staticky odkaz na triedu (System.out).

Ked JVM potrebuje triedu, zavola loadClass() metodu ClassLoadera. Metoda potom vola findLoadedClass() metodu, ktora skontroluje ci trieda uz bola nacitana alebo nie. Ak nebola, metoda deleguje ulohu rodicovskemu ClassLoaderovi, ktory proces zopakuje. Ak nebola nacitana v ziadnom ClassLoaderi, vrchny ClassLoader zavola metodu findClass(), ktora sa pokusi triedu najst. Ak sa to nepodari, vyhodi sa ClassNotFoundException, ktoru odchyti nizsi ClassLoader a on sa pokusi triedu najst.

```
protected synchronized Class<?> loadClass (String name, boolean resolve)
    throws ClassNotFoundException{

    // First check if the class is already loaded
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClass0(name);
            }
        } catch (ClassNotFoundException e) {
            // If still not found, then invoke
            // findClass to find the class.
```

```
                c = findClass(name);
            }
        }
        if (resolve) {
                resolveClass(c);
        }
        return c;
}
```

1.4.1. Typy ClassLoaderov:

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader**: This is the first classloader which is the superclass of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes, etc.
2. **Extension ClassLoader**: This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside *$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader**: This is the child classloader of Extension classloader. It loads the class files from the classpath. By default, the classpath is set to the current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

Trieda je identifikovana ClassLoaderom a plnym menom.

1.5. Dynamické proxy v jazyku Java

- Proxy constructed at runtime as opposed to compile time. You take an existing object and you make a wrapper around it in order to intercept every single call to every single one of its methods.

Dynamicke proxy je prikladom proxy patternu. Proxy je ako prostrednik medzi klientom a realnym objektom. To nam umoznuje kontrolovat pristup k realnemu objektu a delegovat poziadavky nanho podla potreby.

Dynamicke vs staticke proxy:
Ak pouzijeme staticke proxy, pre kazdu triedu musime predom napisat proxy triedu. To znamena ze ak mame 1000 tried, potrebujeme aj 1000 proxy tried. Tomu sa nevyhneme ak sa chceme ku kazdej triede spravat inac. Ak vsak chceme vykonat len nejake genericke spravanie, mozeme pouzit dynamicke proxy. V dynamickom proxy je narozdiel od proxy trieda vytvorena pri runtime a pouzivame InvocationHandler pre definovanie jej spravania. IH je trieda, ktora riesi volanie metod definovanych v rozhrani.

2. Anotácie

2.1. Anotácie — význam a typy metadát v programovom kóde, ich použitie

Metadata su data poskytujuce informacie o inych datach. Anotacie su strukturovane metadata o zdrojovom kode.

Klasifikacia metadat:

- standardne / pouzivatelske
- strukturovane / nestrukturovane
- zabudovane / externe

Metadata pouzivame pre typove definicie, doplnujuce informacie, vytvorenie pomocnych nastrojov, generovanie kodu, runtime konfiguraciu, zaznamenavanie navrhovych rozhodnuti.

Prikladmi metadat su:

-	typy	standardne / pouzivatelske	strukturovane	zabudovane
-	komentare	pouzivatelske	nestrukturovane	zabudovane
-	XML	pouzivatelske	strukturovane	externe
-	anotacie	pouzivatelske	strukturovane	zabudovane

2.2. Anotácie v jazyku Java — definícia a použitie anotačných typov pomocou reflexie

Anotacie vytvarame pouzitim @interface. Anotacie mozu mat vlastne elementy – vyzeraju ako metody bez implementacie. Anotacie nemozu byt rozsirene. Pomocou anotacii mozme oznacit triedy, rozhrania, premenne, alebo parametre.

Classes, methods, variables, parameters and Java packages may be annotated.

Zabudovane anotacie v jave:

- `@Override` - Checks that the method is an override. Causes a compilation error if the method is not found in one of the parent classes or implemented interfaces.
- `@Deprecated` - Marks the method as obsolete. Causes a compile warning if the method is used.
- `@SuppressWarnings` - Instructs the compiler to suppress the compile time warnings specified in the annotation parameters.

```
// DEFINICIA
@interface MyAnnotation {
    String   value();
    String   name();
    int      age();
    String[] newNames();
}

// POUZITIE
@MyAnnotation(
    value="123",
    name="Jakob",
    age=37,
    newNames={"Jenkov", "Peterson"}
)
public class MyClass { }
```

```
  @Retention(RetentionPolicy.RUNTIME) // Make this annotation accessible
at runtime via reflection.
  @Target({ElementType.METHOD})       // This annotation can only be
applied to class methods.
```

2.3. Spracovanie anotácii počas prekladu — anotačný processor

Anotacny procesor sluzi na spracovanie anotacii pocas prekladu. Anotacny procesor moze generovat zdrojove subory. Ak generovany kod obsahuje anotacie, prebieha dalsie kolo generovanie. Vystup jedneho kola je vstupom nasledujuceho.

Je potrebne implementovat javax.annotation.processing.Processor alebo rozsirit javax.annotation.processing.AbstractProcessor a registrovat META-INF/services/javax.annotation.processing.Processor

Implementujeme:
void init(ProcessingEnvironment processingEnv);
boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv);

Vdaka anotacnemu procesoru vieme zistit vsetky elementy, ktore boli oznacene vybranou anotaciu. Na zaklade toho dokazeme generovat kod alebo inak vyuzit tento poznatok.

When Java source code is compiled, annotations can be processed by compiler plug-ins called annotation processors. Processors can produce informational messages or create additional Java source files or resources, which in turn may be compiled and processed. The Java compiler conditionally stores annotation metadata in the class files, if the annotation has a RetentionPolicy of CLASS or RUNTIME. Later, the JVM or other programs can look for the metadata to determine how to interact with the program elements or change their behavior.

In addition to processing an annotation using an annotation processor, a Java programmer can write their own code that uses reflections to process the annotation. the java.lang.reflect package the interface called AnnotatedElement that is implemented by the Java reflection classes including Class, Constructor, Field, Method, and Package. The implementations of this interface are used to represent an annotated element of the program currently running in the Java Virtual Machine. This interface allows annotations to be read reflectively.

The AnnotatedElement interface provides access to annotations having RUNTIME retention. This access is provided by the getAnnotation, getAnnotations, and isAnnotationPresent methods. Because annotation types are compiled and stored in byte code files just like classes, the annotations returned by these methods can be queried just like any regular Java object

3. Random veci

3.1. Princíp inverzie závislosti v architektúre softvéru a vzor Dependency Injection

SOLID:

Single responsibility principle[6]
A class should have only a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open–closed principle[7]
"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle[8]
"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle[9]
"Many client-specific interfaces are better than one general-purpose interface."[4]

Dependency inversion principle[10]
One should "depend upon abstractions, [not] concretions."[4]

Princip inverzie zavislosti je specificky sposob udrzavania nezavislosti modulov. V tomto principe ide o taky navrh softveru, kde su high-level moduly nezavisle od detailov implementacie low-level modulov.
Princip:

- high-level moduly maju byt nezavisle od low-level modulov
- obidva maju byt zavisle na abstrakciach
- abstrakcie by nemali byt zavisle na detailoch ale detaily na abstrakciach.

V prvom rade by sa malo mysliet na abstraktnu interakciu medzi modulami. Moduly by sa mali navrhnut s ohladom na tuto interakciu. V podstate to znamena, ze objektu predame jeho instance variables.

Dependency injection je designovy navrh, pri ktorom nenechavame objekty aby si vytvarali svoje dependencie (instance fields), ale predavame im ich externe. Znizuje zavislost tried. Zvysuje znovupouzitelnost tried a udrzatelnost. Ulahcuje testovanie.

Druhy dependency injection:

- constructor injection
- setter injection
- interface-based injection
- service locator injection

3.2. Generovanie kódu — typy generátorov a techniky generovania kódu

Generatory delime na:

- pasivne – jednorazove, kod je dalej rucne upravovany (generovanie kostry noveho modulu)
- aktivne – opakovane, kod nema byt upravovany rucne, pri potrebe zmeny sa vygeneruje znovu, vstupom je konfiguracia / domenovo specificky jazyk

Podla semantickeho modelu delime generovanie na:

- bez modelu
- s modelom

Podla sposobu generovania delime generovanie na:
- pomocou sablony
- transformacne generovanie

Generovanie bez pouzitia modelu:
Generovany kod obsahuje logiku programu zakodovanu priamo v riadiacich strukturach. Neobsahuje explicitne semanticky model. Vysledny kod nie je upravovany rucne.

Generovanie s modelom:
Obsahuje semanticky model. Je potrebne vygenerovat len kod zabezpecujuci naplnenie semantickeho modelu. Je zachovane rozdelenie medzi vseobecnym / specifickym kodom.

Transformacne generovanie:
Vytvorenie programu, ktory bude na zaklade semantickeho modelu generovat vystupny kod. Dva druhy:
- riadene vystupom – vychadza sa z pozadovaneho vystupneho kodu, z modelu sa ziskavaju udaje potrebne na jeho generovanie
- riadene vstupom - prechadzaju sa vstupne datove struktury a na ich zaklade sa generuje vystup

Generovanie pomocou sablony:
Pouziva sa pri vyvoji webovych aplikacii na generovanie HTML kodu. Vhodne pouzivat, ked je velka cast vystupneho kodu staticka. Pri tomto generovani sa pouzivaju tri hlavne komponenty:
- sablona - vystupny kod, v ktorom su premenlive casti nahradene znackami
- kontext – zdroj, z ktoreho sa cerpaju data pre vyplnenie sablony
- sablonovy system – vyplna sablonu na zaklade kontextu

Generovany kod nie je mozne priamo upravovat, lebo pri opatovnom generovani budu upravy prepisane. Pri OOP jazykoch je mozne oddelit generovany a manualne pisany kod pomocou dedicnosti. Tato technika sa vola generation gap.

Jednotlive sposoby mozu byt aj kombinovane. Typickymi problemami generovania su specialne znaky – escaping a konflikt klucovych slov a konfilktov mien.

3.3. Vývoj softvéru na základe modelu (MDSD), význam modelu pri generovaní

MDSD (Model Driven Software Development) je technika generovania kodu na zaklade vysokourovnoveho modelu. Ten pridava vyssiu uroven abstrakcie ako programovaci jazyk. Povazuje sa za alternativu ku klasickemu programovaniu. Najprv sa navrhne model specifikujuci system, potom sa vyuzije modelovaci nastroj, ktory vygeneruje kod v pozadovanom programovacom jazyku.

MDA (Model Driven Architecture) je standardny pristup k pouzivaniu modelov pri vyvoji softveru. Predpisuje urcite druhy modelov, ako maju byt pripravene a vztahy medzi roznymi druhmi modelov. Ako modelovaci jazyk sa pouziva UML. Zakladnou myslienkou je pri vyvoji separovat operacie systemu od detailov implementacie na konkretnej platforme. Vdaka MDA su systemy specifikovane nezavisle od platformy.

model systemu -> generator -> zostaveny system (generovany kod + univerzalny kod (frameworky) + specificky kod (rucne pisany)

3.4. Makrá a ich použitie v programovacích jazykoch

Makra (macroinstructions) su programovaci pattern, ktory premiena dany input na preddefinovany output. Mozu nahradzovat sekvenciu stlaceni klaves, pohybov mysi, prikazov a podobne.

Pri programovani su makra nastrojom, ktory umoznuje developerovi jednoducho znovupouzit kod. Ide o znacky, ktore su neskor nahradene skutocnym kusom kodu (typicky omnoho dlhsim). Funguju ako search and replace funkcia. Substitucia makra na realny kus kodu sa nazyva makro expanzia a je vykonavana kompilatorom pred kompilaciou skutocneho kodu.

```
#define SWAP(a, b, type)  type t = a; a = b; b = t
SWAP(x, y, int);
```

Syntakticke makra pracuju na urovni syntaktickeho stromu a nie textu. Makro je vtedy funkcia produkujuca fragment syntaktickeho stromu.

```
(define (fib n)
    (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                (fib (- n 2))))))
```

3.4.1. So what are macros?

Macros are a way to make language transformation using a part of the language itself.

```
1 #define MAX(a,b) ((a)>(b) ? (a) : (b))
2 #define SWAP(t,a,b) { t temp; temp = a; a=b; b=temp; }
3 …
4 int a_function(int a, int b) {
5     int mab = MAX(a,b);
6     int maxx = MAX(a+5, b-a);
7     SWAP(int, a, b);
8 }
```

We define here two macros, namely `MAX` (line 1) and `SWAP` (line 2). They actually look like functions: they have a name, they have parameters, they have a body but indeed there is a significant difference.

Functions are called, so all of these elements are used only at run time.

Macros do not exist a run time: all of what they state lives only during the compilation.

line 5: the compiler finds that `MAX(a,b)` is a macro call so it finds inside its symbol table what the content of the `MAX` macro is and transforms, at compile time, what follows with the content of the macro using a symbolic transformation.

In this case the line is transformed into

```
int mab = ((a)>(b) ? (a) : (b));
```

Even if you do not understand the C language, what happens here is a lexical transformation: the compiler analyzes the content of the macro and whenever it finds a parameter, changes it with the content of the actual parameters available when the macro is called.

What happens in the following line?

Same stuff so:

```
int maxx = ((a+5)>(b-a) ? (a+5) : (b-a));
```

You easily get the point that the macro concept is not tied to any particular language: it is a kind of meta-language that enables textual transformation of a text document.

3.5. Prostriedky metaprogramovania v jazyku Python

Python je dynamicky interpretovany jazyk.

Dekoratory (@decorator) umoznuju modifikovat funkciu alebo triedu. @decorator je syntakticka znacka, ktora reprezentuje, ze some_func je zaobalena inou funkciou. To znamena, ze moze urobit nieco pred a po vykonani funkcie.

```
@decorator

def some_func(*args, **kwargs)…
```

Metatriedy su triedy, ktorej instancie su triedy (defaultne type). Vytvorenie metatriedy pozostava z napisania podtriedy inej metatriedy a vlozenie novej metatriedy do procesu vytvarania tried pomocou hooku.

Specialne atributy:

__dict__ - pouzivane pre citanie a zmenu atributov oznaceneho objektu. Basically it contains all the attributes which describe the object under question. It can be used to alter or read the attributes.

```
def func():
    pass
func.temp = 1

print func.__dict__

class TempClass(object):
    a = 1
    def tempFunction(self):
        pass

print TempClass.__dict__
```

Output

```
{'temp': 1}
{'a': 1, '__module__': '__main__', 'tempFunction': <function tempFunction at 0x7f77951a95f0>,
```

__class__ - odkazuje na typ oznaceneho objektu

Specialne metody:

__getattr__	- zachyti pristup k neexistujucim atributom a umozni vykonat akciu
__setattr__	- overriduje defaultne spravanie Pythonu pre priradenie atributov
__call__	- implementovanie sposobuje, ze instancie triedy sa budu spravat ako funkcie
__getitem__	- implementovanie umoznuje pouzivat [] operator

-

3.5.1. *args and **kwargs

It is not necessary to write *args or **kwargs. Only the `*` (asterisk) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention.

*args and **kwargs are mostly used in function definitions. *args and **kwargs allow you to pass a variable number of arguments to a function. What variable means here is that you do not know beforehand how many arguments can be passed to your function by the user so in this case you use these two keywords. *args is used to send a **non-keyworded** variable length argument list to the function

kwargs allows you to pass **keyworded variable length of arguments to a function. You should use **kwargs if you want to handle **named arguments** in a function. Here is an example to get you going with it:

3.6. Systémy typov a ich využitie

- Python – dynamically typed language
- Javascript – untyped language (personal)
- C# - strongly typed

Wiki definicia:
V programovacích jazykoch je typový systém súborom pravidiel, ktoré prideľujú vlastnosti nazývané typ rôznym konštruktom počítačového programu, ako sú premenné, výrazy, funkcie alebo moduly.

Set of rules – napr. String -> Rules (Print, Split, Concat… co mozese s nim robit. Python ho uz pozna) Compiler needs to understand types.

Systemy typov pozostavaju z
- mien typov
- polymorfizmu – nadtyp – podtyp
- generickych typov

Genericke typy umoznuju typu alebo metode operovat na objektoch roznych typov a zaroven poskytuju typovu kontrolu pri kompilacii.

Covariance = narrowing conversion - a feature which allows to substitute a subtype with supertype.
Contravariance = widening conversion - a feature which allows to substitute a supertype with subtype.
Invariance = not convertible – a feature that does not allow any of the above substitution.

Nominalne typovanie – typova kontrola na zaklade mena
Strukturalne typovanie – typova kontrola na zaklade struktury

Typove systemy sa vyuzivaju na typovu kontrolu (odhalenie chyb), zvysenie vykonu a pri dokumentacii.

4. Aspektovo orientovane programovanie

4.1. Aspektovo orientované programovanie — význam, základné vlastností, realizácia v AspectJ

Zvysuje modularitu kodu eliminaciou cross cutting concerns (opakujuce sa casti kodu v roznych moduloch). Riesi tangling (prepletanie) a scattering (rozptylenie) kodu. Nemodifikuje povodny kod ale separatne specifikuje ktory kod ma byt modifikovany pomocov pointcutov. To umoznuje pridat spravanie, ktore nie je klucove pre biznis logiku do kodu bez zneprehladnovania povodneho kodu.

Dolezite vyrazy AOP:

- aspekt – zapuzdrenie pretinajucich sa zaujmov
- joinpoint – dobre definovane miesto v programe / bod v procese vykonavania
- pointcut – mnozina joinpointov
- advice - funkcionalita vykonana vo vybratych pointcutoch

Pri kompilacii dochadza k pretkavaniu (weavingu) kodu z adviecov do kodu povodneho programu. Za pretkavanie je zodpovedny ajc – AspectJ Compiler. Pretkavanie moze byt staticke aj dynamicke.

AspectJ je AOP pre Javu. Ide o jazyk pre definiciu pravidiel pretkavania:

- aspect namiesto class
- .aj namiesto .java

Vyuzitie: logging, transactions, frameworks.

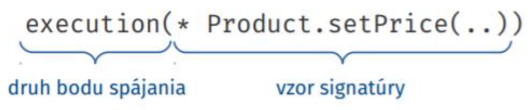
4.2. AspectJ — body spájania a súvisiace bodové prierezy (kinded pointcuts)

Body spajania su definovane miesta v programe, kde ma AspectJ modifikovat kod.
- join point - dobre definovane miesto v programe kde chcem vykonat nejaku akciu
- pointcut – mnozina joinpointov

Body spajania:

- metoda, konstruktor	- execution, call
- clenska premenna	- get, set
- zachytenie vynimky	- handler
- staticka premenna	- static initialization
- inicializacia objektu	- initialization, preinitialization

Kinded pointcuts su prierezy podla druhu bodu spajania (joinpointu). V priereze moze byt iba jeden druh bodu spajania.



```
execution(* Product.setPrice(..))
```

Deklaracia pintcutu:

```
pointcut deliverMessage() : call(* Messanger.deliver(..));
```

4.3. AspectJ — non-kinded pointcuts

Prierezy ktore vyberaju joinpointy na zaklade inych kriterii ako je signature

Body spajania:

- zalozene na control flow	- cflow, cflowbelow
- zalozene na zdrojom kode	- within, withincode
- zalozete na cielovom objekte	- this, target, args

4.4. AspectJ — odporučenia

Odporucenia su zdrojovy kod, ktory sa ma vykonat pri pointcute. Pozname tri druhy odporuceni podla toho kde pri pointcute sa maju vykonat:

- before
- after (/ returning / throwing)
- around:
    - obklopuje bod spajania
    - ma navratovu hodnotu
    - vykonanie obklopeneho kodu pomocou proceed()

```
after() throwing(Exception ex) : loggedOperation() {
        // zaznamenanie chyby
}
```

4.5. AspectJ — statické pretínanie

Doplnenie struktury triedy. Je mozne pridat clenov do existujucej triedy, alebo interfaceu. Mozme pridat field, metodu, alebo konstruktor. Toto sa nazyva medzitypove deklaracie (ITD – Intertype Declarations). Iba 1 typ alebo rozhranie.

```
public aspect FooAspect {

public float Foo._newField = 9.0;


public void Foo.newMethod() {

System.out.println("Crosscutting - The static way !");

System.out.println("Value of field : " + _newField);

}

public void FooInterface.blue(){

System.out.println("In new method from Aspect");

}
public aspect FooInterfaceAspect {

declare parents : Foo implements FooInterface;

}

}
```

4.6. AspectJ — vytváranie inštancií aspektov

Instancie aspektov sa vytvaraju automaticky systemom. Pouzivaju bezparametricky konstruktor a aspekt je zaroven singletonom. Aspekt moze mat clenske premenne a metody, byt abstraktnym, implementovat rozhrania a dedit od inych aspektov.

Je mozne vytvorit aspekt pre objekt osobitne a podobne:

- perthis, pertarget
- percflow, percflowbelow
- pertypewithin

```
public abstract aspect TracingAspect pertypewithin(ajia.services.*) {
…
}
```

Ziskanie instancie:

- aspectOf()
- hasAspect()

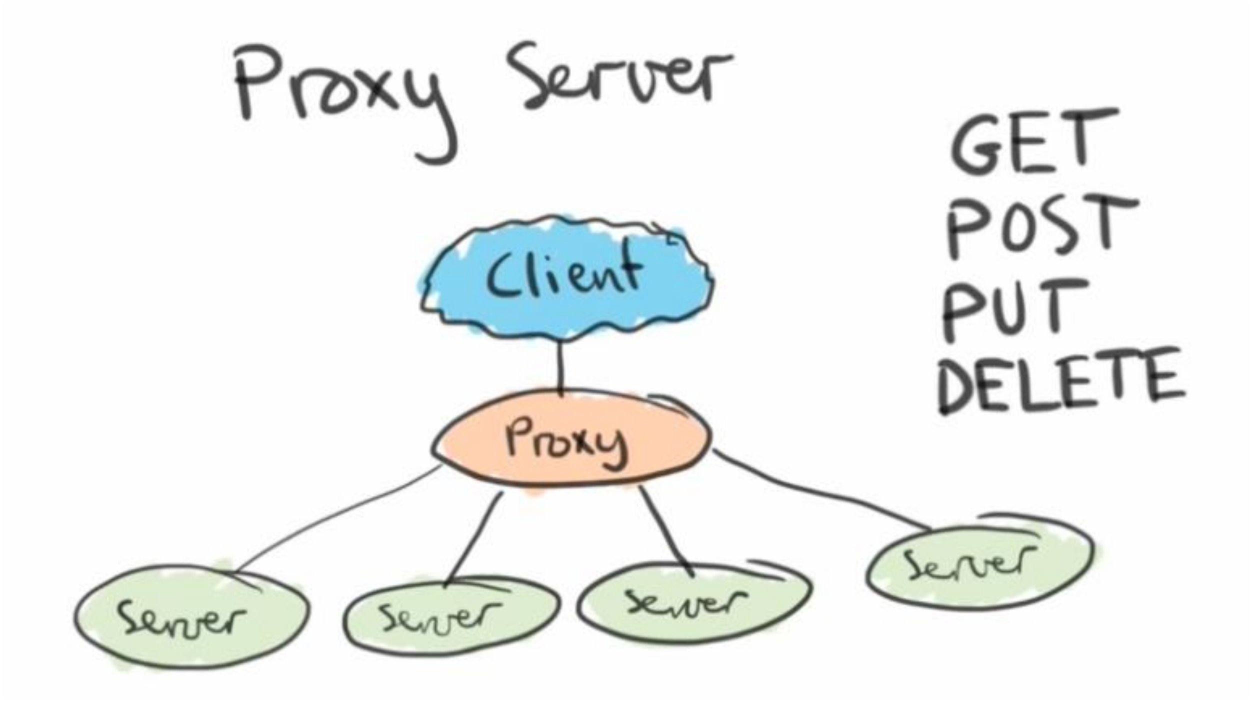
5. Moje veci

- Happens during compile time – or at least before your code runs
- Reflexion (runtime)
    - Introspection (code inspects itself)
    - Self-modification (runtime)
    - Intercession

In some cases this allows programmers to minimize the number of lines of code to express a solution, and thus reducing the development time.

Expressivity to model – elegant and friendly interfaces.

Proxies:



- In Java the code is in named blocks
- 2 memory area reserved when you hit the run button:
    - Stack
    - Heap – local reference variable Car myCar; declaration: will be a reservation in stack. myCar = new Car(); - not a primitive data type. It refers where the actual object locates.

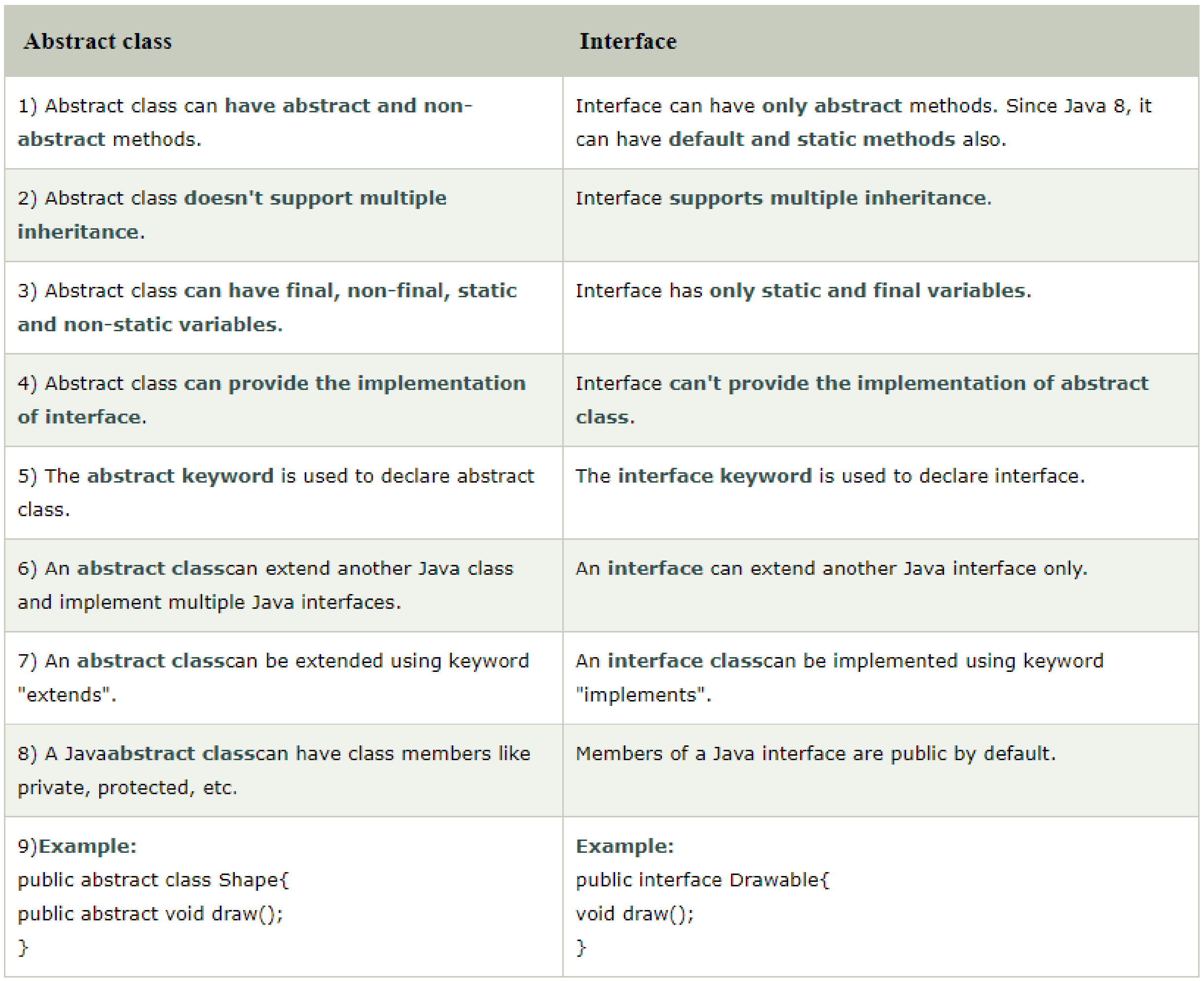
It is in heap ☺ So from the Stack what is created will point to the Heap. And there will be the actual value of the variable.

Garbage collection is a process that runs in the heap and looks for rouge objects, lost objects, objects without reference.

Java Data Objects (**JDO**) is a specification of Java object persistence. One of its features is a transparency of the persistence services to the domain model. JDO persistent objects are ordinary Java programming language classes (POJOs); there is no requirement for them to implement certain interfaces or extend from special classes

6. Pohovor

6.1. Abstract class vs Interface



Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class**can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class**can be extended using keyword "extends".	An **interface class**can be implemented using keyword "implements".
8) A Java**abstract class**can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

- Abstract methods means there is no default implementation for it and an implementing class will provide the details.

6.2. Difference between JavaEE and JavaSE

Java SE

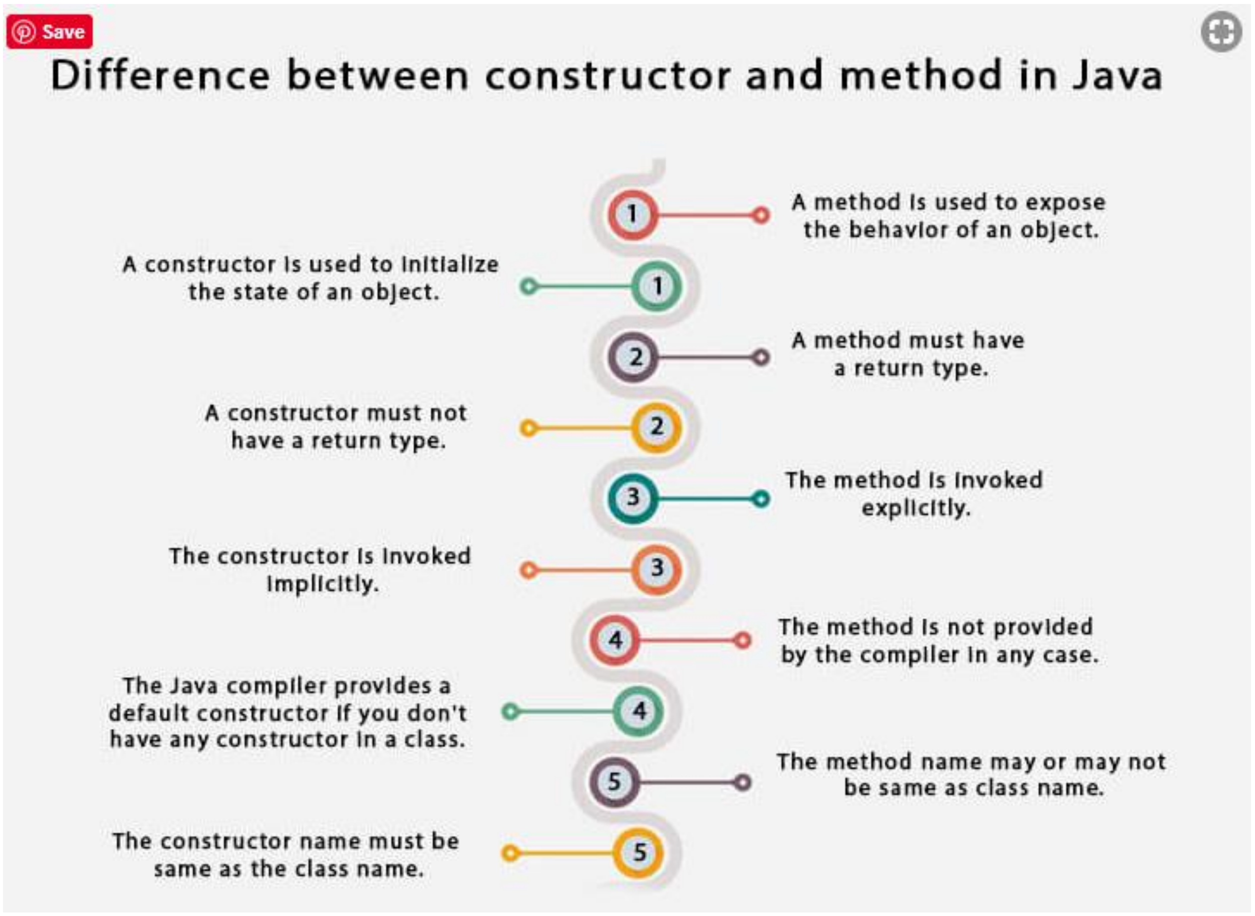
When most people think of the Java programming language, they think of the Java SE API. Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

In addition to the core API, the Java SE platform consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits commonly used in Java technology applications.

Java EE

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications

6.3. Method vs Contructor


Save
Difference between constructor and method in Java
1 A method is used to expose the behavior of an object.
1 A constructor is used to initialize the state of an object.
2 A method must have a return type.
2 A constructor must not have a return type.
3 The method is invoked explicitly.
3 The constructor is invoked implicitly.
4 The method is not provided by the compiler in any case.
4 The Java compiler provides a default constructor if you don't have any constructor in a class.
5 The method name may or may not be same as class name.
5 The constructor name must be same as the class name.

6.4. Random

1. Portable: Java supports read-once-write-anywhere approach. We can execute the Java program on every machine. Java program (.java) is converted to bytecode (.class) which can be easily run on every machine.