Contents

[Contents 1](#_Toc118882069)

[Java 3](#_Toc118882070)

[Syntax jazyka Java 6](#_Toc118882071)

[Syntax metód 8](#_Toc118882072)

[Rozsah premenných 10](#_Toc118882073)

[Primitívne dátové typy: 11](#_Toc118882074)

[int 11](#_Toc118882075)

[char 12](#_Toc118882076)

[boolean 12](#_Toc118882077)

[double 12](#_Toc118882078)

[float 12](#_Toc118882079)

[long 12](#_Toc118882080)

[OverFlow: 12](#_Toc118882081)

[UnderFlow 13](#_Toc118882082)

[Type casting 14](#_Toc118882083)

[String 14](#_Toc118882084)

[StringBuilder class. 16](#_Toc118882085)

[Primitivne dátové typy vs referenčné typy: 17](#_Toc118882086)

[Operátory a ich priorita : 18](#_Toc118882087)

[Aritmetické operátory: 19](#_Toc118882088)

[Logické operatory 21](#_Toc118882089)

[|| (OR) 21](#_Toc118882090)

[&& a & (AND) 21](#_Toc118882091)

[!(NEGACIA) 22](#_Toc118882092)

[Ternárny oprátor 23](#_Toc118882093)

[Komentáre: 23](#_Toc118882094)

[If / Else 24](#_Toc118882095)

[Switch statement 25](#_Toc118882096)

[Cykly 29](#_Toc118882097)

[For 29](#_Toc118882098)

[FOR EACH 30](#_Toc118882099)

[While 31](#_Toc118882100)

[DO WHILE 31](#_Toc118882101)

[Príkazy break a continue 32](#_Toc118882102)

[Array 33](#_Toc118882103)

[ArrayList 35](#_Toc118882104)

[Primitive Wrapper Types 36](#_Toc118882105)

[Triedy 37](#_Toc118882106)

[Staticke datove cleny a metody 39](#_Toc118882107)

[Abstraktná trieda 39](#_Toc118882108)

[Definícia abstrakcie 40](#_Toc118882109)

[Rozhranie – Interface 42](#_Toc118882110)

[Rozdiel medzi abstraktnou triedou a rozhraním v jazyku Java 42](#_Toc118882111)

[definícia 42](#_Toc118882112)

[premenné 42](#_Toc118882113)

[metódy 43](#_Toc118882114)

[Viacnásobné dedičstvo 43](#_Toc118882115)

[Kľúčové slovo triedy 43](#_Toc118882116)

[Rozšírenie a implementácia 43](#_Toc118882117)

[uskutočnenie 43](#_Toc118882118)

[záver 43](#_Toc118882119)

[Zapúzdrenie - encapsulation 43](#_Toc118882120)

[Dedičnosť - inheritance: 44](#_Toc118882121)

[Polymorfizmus: 46](#_Toc118882122)

[Vynimky: 46](#_Toc118882123)

[IO Java 53](#_Toc118882124)

[Čítanie súborov pomocou Java IO 55](#_Toc118882125)

[Trieda File 57](#_Toc118882126)

[Writing File via Java IO 58](#_Toc118882127)

[Java Networking 59](#_Toc118882128)

[JDBC 63](#_Toc118882129)

[Import JavaFx pri java 9 a vyššie 67](#_Toc118882130)

[Maven 70](#_Toc118882131)

[Git a Github 72](#_Toc118882132)

[Trieda Scanner 73](#_Toc118882133)

[Poznámky 74](#_Toc118882134)

[Úlohy: 74](#_Toc118882135)

[1 Mačka: 75](#_Toc118882136)

[2. Prevodník Libry na kilogramy 75](#_Toc118882137)

[3. Priestupný rok 75](#_Toc118882138)

[Písomky 78](#_Toc118882139)

[Prva písomka 78](#_Toc118882140)

# Java

Java je Objektovo orientovaný programovací jazyk(OOP), pretože jeden z hlavných cieľov Javy technológie programovania je vytvoriť objekty alebo časti samostatnejších kódov, ktoré môžu vzájomne pôsobiť s inými predmetmi, na vyriešenie problému.

Program pozostáva z množiny objektov, ktoré sú schopné uchovávať a spracovávať dáta a komunikovať s ostatnými objektami. Základné časti jazyku Java sú **trieda** a **objekt.**

-Cierna skrinka dopisat

**Trieda**:

-je to štrukturovaný dátový typ charakterizovaný vlastnosťami (dáta, atribúty) a schopnosťami (metódy)

Príklad: trieda Človek

***Vlastnosti***: meno, vek, výška, váha, farba vlasov...

***Schopnosti***: predstaviť sa, povedať svoj vek, miery atď.

**Objekt:**

-konkrétny prvok triedy, s jednoznačne danými vlastnosťami

Príklad:

Janko Kováč, 45 rokov, svetlohnedé vlasy, 180 cm

Majka Nováková, 35 rokov, hnedé vlasy, 165 cm

Tri základné princípy:

**Zapúzdrenie** (encapsulation)

**Mnohotvárnosť** (polymorphism)

**Dedičnosť** (inheritance)

**Zapúzdrenie:**

Základou jednotkou zapúzdrenia je *trieda*. Zapúzdrenie je mechanizmus, ktorý zväzuje dohromady dáta a kód do *objektu*. V úplne objektovo orientovanom programe patria všetky dáta a funkcie nejakej triede (okrem hlavného programu) . Zapúzdrenie umožňuje lepšiu prehľadnosť programu a najmä môže chrániť dáta pred nežiadúcimi zásahmi zvonku. Vo vnútri triedy môžu byť všetky dáta alebo metódy definované ako súkromné - prístupné len pre triedu samotnú verejné – prístupné aj pre ostatné triedy.

**Polymorfizmus:**

-Ide o mnohotvárnosť, resp. viacúčelové využitie metód ν Metóda s jedným názvom môže byť použitá pre rôzne typy dát alebo rôzny počet vstupov, čo uľahčuje orientáciu v programe

**Dedičnosť:**

-Každá trieda môže mať svoje “dieťa”, teda triedu, ktorá je od nej odvodená, preberá všetky jej dáta a metódy (okrem konštruktora a deštruktora)

-Odvodená trieda môže byť potomkom ľubovoľného počtu tried, ale má len jedného priameho rodiča. A každá trieda môže mať ľubovoľný počet potomkov

-Pre rodičovskú triedu je možné sprístupniť svojim potomkom svoje súkromné dáta, takéto dáta sa nazývajú chránené a okrem samotnej triedy a jej podtried nie sú inak zvonku prístupné

**Konštruktor a deštruktor:**

- Sú to funkcie, ktoré sa automaticky spustia pri vytvorení (konštruktor) a pri zániku (deštruktor) objektu danej triedy

***Konštruktor*** - používa sa najmä na počiatočné nastavenie hodnoty dát daného objektu a na alokáciu potrebnej pamäte

***Deštruktor*** (garbage collection)- používa sa na “upratanie”, teda hlavne na dealokáciu vyhradenej pamäte

**Rozdiel medzi OOP a procedurálnym programovaním**

Objektovo orientované programovanie (OOP) a procedurálne programovanie sú dve programovacie paradigmy. Programovacie paradigma je základný štýl počítačového programovania a líši sa spôsobom, akým sú zastúpené rôzne prvky programu a ako sú definované kroky na riešenie problémov. Ako už názov napovedá, OOP sa zameriava na ***reprezentáciu problémov pomocou objektov v reálnom svete a ich správania***, zatiaľ čo procedurálne programovanie sa ***zaoberá reprezentáciou riešení problémov pomocou procedúr, ktoré sú kolekciami kódu, ktoré bežia v špecifickom poradí*.**

Kľúčovým rozdielom medzi OOP a procedurálnym programovaním je to, že cieľom procedurálneho programovania je ***rozdeliť programovaciu úlohu do súboru premenných a podprogramov***, zatiaľ čo OOP sa zameriava na ***rozdelenie programovacej úlohy na objekty, ktoré zapuzdrujú údaje a metódy***. Najvýznamnejší rozdiel by mohol byť v tom, že zatiaľ čo procedurálne programovanie ***používa postupy na priame fungovanie na dátových štruktúrach,*** OOP spojí údaje a metódy dohromady, takže ***objekt bude pracovať na svojich vlastných údajoch***. Pokiaľ ide o ***nomenklatúru, postup, modul, volanie procedúry a premenná*** v procedurálnom programovaní sa v OOP často označujú ako ***metóda, objekt, správa a atribút***.

**História**

OOP vzniklo v období, keď bežné programy začali presahovať určitú dĺžku a štrukturálne programy sa stali neprehľadnými. Bežne sa začalo využívať začiatkom 90. rokov

**Simula(1967)** – prvý OO jazyk, používaný na simulácie pri konštrukcii integrovaných obvodov, na počítačovú grafiku, vzdelávanie a pod.

**Smalltalk(1970)** – OO reflektívny jazyk, vyvinutý pôvodne na vzdelávacie účely, ale neskôr našiel aj bohaté komerčné uplatnenie

**Python(1980)** – jazyk podporujúci rôzne programátorské techniky, s dôrazom na ľahko čitateľný kód, mnohé softvéry ho používajú ako svoj skriptovací jazyk

**C++(1983)** – jeden z najrozšírenejších komerčne používaných jazykov, s množstvom aplikácií, umožňuje OO aj procedurálne programovanie ν Common Lisp – jazyk umožňujúci veľmi všeobecné operácie, s veľkým množstvom dátových typov, využíva sa na programovanie internetových aplikácií, dizajnérskych softvérov, hier atď.

**Perl(1987)** – dynamický jazyk pôvodne vyvinutý pre spracovanie textov, dnes má využitie pri programovaní internetových aplikácií, pri správe systémov a pod., je efektívny pri manipulácii s reťazcami a správe pamäte

**PHP(1994)** – reflektívny programovací jazyk (program môže meniť svoju štruktúru počas behu), používaný najmä pri vývoji web stránok

**Java(1995)** – jazyk vyvinutý tak, aby umožňoval spúšťanie programov na rôznych platformách a zo vzdialených zdrojov, so zabudovanou podporou siete

**JavaScript(1995)** – skriptovací jazyk (inštrukcie sa vykonávajú priamo z kódu, bez kompilácie) vyvinutý na tvorbu internetových aplikácií

-Java Virtual Machine

JVM je ***emulácia počítača, ktorá umožňuje Jave bežať na všetkých zariadeniach***, ktoré si viete predstaviť. Počítače, android telefóny, automaty na listy v MHD, to všetko je Java.

V princípe, JVM zoberie skompilovaný program a ten ďalej spracuje. Program [Java](https://www.learn2code.sk/kurzy/java-pre-zaciatocnikov" \t "_blank)kompilátor skompiluje do tzv. bytecode a tento bytecode dokáže JVM verifikovať a interpretovať.

Nevyhnutnosť JVM spočíva práve v tom, že na rozdielnych prístrojoch s rozdielnymi operačnými systémami majú počítačové programy k dispozícii rozdielne funkcie, ktoré môžu využívať – takzvané API (Application Programming interface). JVM je súčasť JRE (Java Runtime Environment) a ďalšou súčasťou JRE sú aj java API. Takže fakt, že pri inštalácii javy na prístroj, kde budú bežať naše programy (povedzme smart televízor), si nainštalujeme aj virtuálny počítač (naša JVM), umožní využívať všetky funkcie, ktoré sú k chodu programu potrebné (API). A keby sme sa rozhodli tento program nainštalovať namiesto Smart TV napríklad na smartfón – nie je problém, pretože na ňom beží rovnaká JRE s našimi API a JVM.

**-JVM je *vlastne súbor programov, ktoré sa načítajú do pamäte a vytvoria virtuálny počítač.*** Patria sem registre, pamäť (STACK a HEAP), execution environment a ďalšie.

**Výhody a nevýhody Java jazyka**

**-** Jednou z najväčších výhod ovládania programovacieho jazyka Java je vysoká zamestnanosť. Na trhu práce je veľký dopyt pre ľudí so znalosťami s týmto programovacím jazykom. Java sa používa vo veľkom množstve firiem a s dopytom rastie aj platové ohodnotenie.

**-** Vďaka automatickej správe pamäti (garbage collector) sa prográmator nemusí starať o manuálne čistenie pamäti po nepotrebných objektoch.

**-** Veľká komunita, ktorá vznikla za samotnou Javou = open-source jazyk (otvorený softvér je počítačový softvér s otvoreným zdrojovým kódom). Vďaka veľkej komunite si ľahšie vyhľadáte pomoc pri problémoch s Javou. (napr. fórum na to určené)

**-** Kód napísaný v Jave sa dá spustiť všade, nehľadiac na platformu. Keďže ide o interpretovaný jazyk a spušťaný v JVM.

- Bezpečnosť: java aplikacie su spúšťané v Java prostredí (JVM) a tak izolujeme applikaciu od priameho spojenia s prostredim OS.

Metoda main:

Metoda main je prvým kódom, ktorý prekladač v aplikácií vykoná. Metoda main je špeciálna metoda, ktorú Java Virtual Machine rozoznáva ako štartovací bod pre všetky Java technologické programy, ktoré bežia z príkazového riadku alebo riadku.

Metoda main obsahuje všetky programové príkazy, deklarácie premenných, príkazy cyklov atd. Každý program, ktorý chcete rozbehnúť z príkazového riadku alebo riadku musí mať metódu main.

public static void main(String[] args) {  
   
}

***Všetok kód píšeme do {}. Ak nedáme kód do zložených závoriek tak java vyhlási chybu.***

# Syntax jazyka Java

<https://www.tutorialspoint.com/java/java_basic_syntax.htm>

package lubos.sukup.main.generator;  
  
import java.util.Random;  
  
public class Generator {  
  
  
 char a = 'a';  
 float er = 5.4f;  
  
 public static void main(String[] args) {  
 Random random = new Random();  
 System.*out*.println(random.nextInt(10)+1);  
  
 int y =5;  
  
 if(false){  
 int a =0;  
 }  
 else {  
 int x = 10;  
 }  
  
 }  
  
}

Aplikácia Java, z hladiska fyzickej architektúry obsahuje dva komponenty. Triedy ( Class) a balíčky (Package). Pre predstavu analogicky baličky a triedy predstavujú to čo priečiny a subory v operačnom systéme.

Názvy tried vždy začínajú veľkým začiatočným písmenom. Pokiaľ názov jezložený z viac slov tak použijeme pomenovanie nazvané *CamelCase.*

Príklad názvu triedy: ***MojaPrváTrieda.class***

Názvy balíčkov píšeme len malými písmenami.

Základ tvorí trieda. Je to súbor, do ktorého zapisujeme java kód. Trieda obsahuje adresu umiestnenia tiredy v aplikacii a nazýva sa package. Package je vždy na začiatku. Zapis cesty je rozdelenz bodkou a konci je bodko-čiarka ***;***.

Potom nasledujú importy. Keďže trieda može použivat rôzne objekty umiestnené v rôznych častiach aplikácie tak pomocou importov je trieda schopná náisť miesto dannej triedy, z ktorej je objekt použitý. Tak ako package, import udáva umiestnenie.

Nasleduje nazov triedy. *public class Nazov {}*. Konvencia je, že názov píšeme s velkým začiatočným písmenom. Telo triedy je ohranicene {}. Do tela triedy sa píše všetok kód.

Metódy sa možu vykonávat len na nejakých objektoch. Napriklad toLowercase(). Je to metóda, ktorá sa používa s objektom String. Bez objektu java nie je schopná vykonať metódu. Vždy je treba nejaký objekt na vykonanie metód. “Meno”.toLowerCase(). (existuje výnimka a tým sú statické metódy, ale o nich sa budeme rozpravať neskôr 😊 ).

Príklad deklarácie a inicializácie premennej:

**typ premennej názov priradenie hodnota ukončenie príkazu**

int x = 10 ;

deklarácia long a float.   
 Treba pri inicalizácii typu long na konci dat písmeno L inak defaultne sa priradi int a nemude možné použiť velké čísla

long l = 34344445556; // java vyvolá chybové hlasenie

long l = 34344445556L;

Pri inicializíácii float reba dat pismeno f inak defaultne priradi hodnotu typu double.

float f = 453.664f;

Všetok kód sa v Jave píše do Tried. Každá trieda musí obsahovať:

* package : je to umistnenie triedy v adresárovej štruktúre aplikácie
* importy: to sú importy častí, ktoré Trieda použie

Poradie package a import **je nemenné a musí byť** na začiatku súbrou triedzy

package lubos.sukup.main.MujString;  
  
import java.lang.\*;  
  
public class NazovTriedy {  
   
   
 // !!!!!!!Tu píšeme všetok kód!!!!!  
   
   
}

Každý blok kódu je umiestnený do { }. A budeme chcieť napísať obligátne System.out.println(); do tela triedy bez toho, aby sme to napísali do { }, kompilátor Java vyhlási chybu. Každú činnosť čo má program vykonnať musí byt v zložených závorkách.

Pozor deklarácia premenných nieje nijaká činnosť a preto pri deklarácii premenných netreba uzatvárať ich do {}. Prakticky sú už v jednych závorkách a to v závorkách triedy.

public class NazovTriedy {  
  
 int x = 10;  
  
 StringBuilder stringBuilder = new StringBuilder("test");  
  
 //je tu chyba nie je v { }  
 System.*out*.println(stringBuilder);  
  
 {  
 //vsetko je v poriadku   
 System.*out*.println(stringBuilder);  
 }  
   
  
  
}

Obyčajné závorky sa v jazyku java používajú pri zadávaní parametrov. Napríklad pri metódach, ak metóda vyžaduje stupné dáta. Vid. System.out.println(„Nejaky text“);

## Syntax metód

Ako sme si povedali tak každý blok kodu sa uzatvára zloženými závorkami. A ak chceme použit ten samý blok kódu možeme donekonečna kopírovať blok kódu tam kde potrebujeme. Bohužial takto vzniká duplicitný kód a vzrastá neprehľadnosť kódu. Aby sme zamedzili týmto veciam zaviedli sa metódy.

Metóda je len pomenovaný blok kódu. Spolu s menom treba zadať vstupne parametre. Respektíve vstupné dáta. Tie sa dávajú do obyčajných závoriek za názov. A typ objektu/premennej aký metóda po vykonaní kód vnútri { } navráti. Do metódy svtupujú dáta a vystupujú upravené dáta. Samozrejme nie je podmienka. Metóda nemusí mať žiadne vstupy, výstupy alebo ani jedno.

public class Trieda {  
  
  
 {  
 System.*out*.println("vysup do konzoly");  
  
 }  
}

Ak chceme nami vytvoreneú procedúru použit viacej krat a neduplikovať kód tak možme pomenovat nami vytvorený blok kódu.

public class Trieda {  
  
  
 void mojaMetoda(){  
  
 System.*out*.println("vysup do konzoly");  
  
 }  
}

Do metód zapisujeme úkony, procedúry čo má program vykonať. Na vytvorenie metódy treba názov. Názov píšeme v malým začiatočným písmenom.

Pri vytváraní metódy je treba zadať navratomú hodnotu. V hore uvedennom prípade je to *void*. Je to špecialny typ návratovej hodnoty kedy metóda nevracia žiadnu hodnotu. Ako návratový typ môže byť akýkoľvek objekt alebo primitívny typ.

**public static final void nazovMetody( int x, int y ){**

**return;**

**}**

public, static , final - nepovinné parametre metódy. Jazyk java môže prijať aj metódy bez týchto parametrov.

Napr: *void nazovMetody(){}*

Parameter prístupu: public, private, protected. Metóde bez parametra prístupu java pridá zakladný tzv. default.

Povinné parametre metódy:

**Návratový typ** – void, alebo akýkolvek primitívny dátový typ , alebo akíkolvek objekt.

Ak je definovaný navratový typ tak telo metódy musí obsahovať

***return***.



Hore je metóda test a navratový typ je definovaný ako int.

**Názov Metódy** – akýkoľvek názov, ktorý je povolený v jazyku Java.

**()** – obyčajné zátvorky obsahujú vstupné premenné, s ktorými nami vztvorená metóda pracuje.

Zátvorky môžu byť prázdne.

**{} –** telo metódy.

Dalšie nepovinné parametre: static final synchronized

## Rozsah premenných

Premenné definujeme a použivame v rôznych rozsahoch(scope). A to v rozshu metod. Kedy platnosť premenných je ohraničenaá dannou metódou. Alebo môžme definovať premenné v rámci celej triedy.

public class Trieda {  
  
 static int *premennaTriedy* = 1000;  
  
 public static void main(String[] args) {  
   
 int premenaVmain = 60;  
   
 //Je to v poriadku   
 premenaVmain = *premennaTriedy*;  
   
 //Nie je to v poriadku. premennaMetody existuje len v   
 // rozsahu zlozenych zavoriek metody mojaMetoda  
 premenaVmain = premennaMetody;  
   
 }  
   
   
 void mojaMetoda(){  
  
 int premennaMetody = 80;  
 //Je to v poriadku premennaTriedy je viditelna v   
 //celej triede teda aj v metode mojaMetoda()  
 premennaMetody = *premennaTriedy*;  
 System.*out*.println("vysup do konzoly");  
  
 }  
}

Zápis premennej s modifikátorom **final** zabraňuje ďalším zmnám hodnoty premennej:

final int x =100;

Hodnota x ostane konštantná po celý čas behu programu.

# Primitívne dátové typy:

int long, char, boolean, float, double. String

char:

Slúži na uchovávanie znakov(characters). Zápis char premennej je následovný:

char znak ='x';

V premennej char si ukladáme znaky v Unocode formáte. To znamená, že môžeme zapisať ten isty znak rôznymi spôsobmi.

char znak = 'x';  
  
//zapis x v hexadecimalnom cisle  
char uni = '\u0078';

byte 16b

byte b = 5;

## int

int 32B

int x = 15;

## char

char 16b

char ch = ‘f’;

char sa používa na ukladanie znaku do pamäti. Aj keď ukladáme znak ako písmeno tak java ukladá znak v číslelnom tvare. To číslo reprezentuje záznam v UNICODE znakovej sade. Ak chceme uložit písmeno **J** pamäte tak vskutočnosti ukladáme jeho čiselnú preprezentáciu v UNICODE a tým je čislo **74.** Len ešte hovorime Jave že 74 musí interpretovať ako **J**. A hovoríme to jave tak, že číslo uložíme do premennej char.

## boolean

boolean true false.

boolean ano = true;

## double

double 64b

double = 12.5;

## float

float 32b

float f = 12.545f;

## long

long 64b

long l = 15165L;

Číselné dátové typy môžeme zapísať v

desiatkovej sústave 123

osmičkovej 012 10 v decimal

dvojkovej sústave 0b0100201

šestnástkovej sústave 0xAB12

## OverFlow:

K overflow dôjde, keď takúto hodnotu priradíme premennej, ktorá je vyššia ako maximálna povolená hodnota

Príklad overflow:

Zoberme si prípad premennej int, ktorá má 32 bitov a akoukoľvek hodnotou, ktorá je vyššia ako Integer.MAX\_VALUE (2147483647), sa prevráti. Napríklad Integer.MAX\_VALUE + 1

vráti -2147483648 (Integer.MIN\_VALUE).

Pretože typ údajov int je v jazyku Java 32 bitov, prevráti sa akákoľvek hodnota, ktorá presahuje 32 bitov. Z číselného hľadiska to znamená, že po zvýšení 1 na Integer.MAX\_VALUE (2147483647) bude vrátená hodnota -2147483648. V skutočnosti si tieto hodnoty nemusíte pamätať a je možné použiť konštanty Integer.MIN\_VALUE a Integer.MAX\_VALUE.

normalne by mal vbyt vysledok 2147483648 ale kedze int ma maximalnu hodnotu dannout velkosi pamate teda 32b  
// tak nie je mozne zapisat do int väčšiu hodnotu ako 2147483647. z tohto dovodu pamät pretetie na zaciatok  
// je to ako tachometer: ked prejdete maximalnu hodnotu na tachometri tak sa vynuluje a ide sa od zaciatku.

public static void main(String[] args) {  
  
  
 int maxValue = Integer.*MAX\_VALUE*;  
  
 int result = maxValue + 1;  
  
 System.*out*.println(maxValue); // 2147483647   
   
 System.*out*.println(result); // -2147483648  
  
   
}

## UnderFlow

K underflow dôjde, keď takúto hodnotu priradíme premennej, ktorá je menšia ako minimálna povolená hodnota

Príklad underflow

Underflow je opakom overflow. Keď v prípade overflow dosiahneme hornú hranicu, v prípade underflow dosiahneme dolnú hranicu. Po dekrementácii 1 z Integer.MIN\_VALUE teda dosiahneme Integer.MAX\_VALUE. Tu sme prevrátili z najnižšej hodnoty int na maximálnu hodnotu.

V prípade typov údajov, ktoré nie sú celočíselnými, vedie overflow a underflow k hodnotám INFINITY a ZERO.

int a = Integer.MAXVALUE+1

int b = Integer. MINVALUE-1

## Type casting

Pri matematických operáciach sa výsledok zmení na vyšší typ premennej. Teda ak sčítame byte + byte, short + short , automaticky sa výsledok pretypuje na int.

byte b1 = 4;  
byte b2 = 5;  
  
int x = b1 + b2;  
byte y = (byte)(b1 + b2);

ak chceme zachovat stavajúci dátový typ treba pretypovať. Pretypovanie sa robi tak, že pred dátový typ dáme požadovaný typ do závorky. Vid. Príklad.

Pozor ak pretypujeme veľmy veľké čisla na menší dátový typ. Môžeme stratit informáciu. Teda bude to úlne iné číslo. 32B čislo nevojde do 16B číselného typu.

rozširujúce konverzie (do nadtypu):

byte->short->int->long->float->double

zužujúce konverzie (do podtypu):

double->float->long->int->short->byte

## String

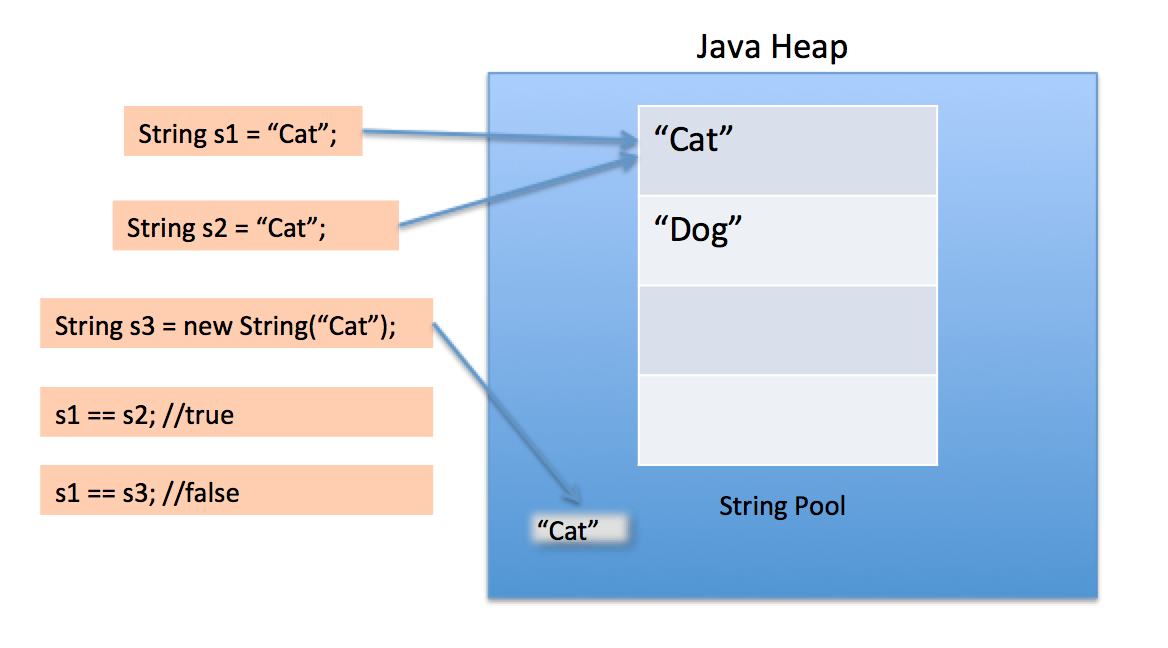
String pool.

String pool v jave je miesto v Java Heap pamäti. String je špecialna trieda ktotej objekty môžeme vytvárať pomocou peratoru new:

String s1 = new String("Macka");

Alebo pomocou priadenia =:

String s2 ="Macka";



// poznamka – pripomenúť aj string + string . a porovnam s necim uz vytvorenym tak sa to nerovna ==

Obrázok názorne ukazuje ako sú objekty String usporiadané v pamäti. Kvôli String Poolu je String nastavený tak, že je String je nemenný. (immutable). Dovod vyniku string pool je to, že String je pamäťovo náročné a tak keď je String rovnaký tak keď si vytvoríme novú premmenú string s rovnakým obsahom tak nová premenná String bude odkazovať na to isté miesto v pamäti.

String s = „Jano“;

String b = „Jano“`

String c = new String(„Jano“);

Equals vs ==

Pri Stringu je treba byť na pozore ako porovnávame premenné a absah premenných v jazyku Java. Tu je možné použiť == alebo metódu equals(String string).

public static void main(String[] args) {  
  
 String s1 = "jano";  
 String s2 = "jano";  
  
 String s3 = new String("jano");  
  
 // Vysledok porovnavania je true  
 boolean rovnajuSa = s1 == s2;  
 System.*out*.println(" s1 a s2 boolean je: " + rovnajuSa);  
  
 // Vysledok porovnavania je false  
 boolean neRovnajuSa = s1 == s3;  
 System.*out*.println(" s1 a s3 boolean je: " + neRovnajuSa);  
  
 // Vysledok porovnavania je true  
 boolean rovnajuSaEquals = s1.equals(s3);  
 System.*out*.println(" s1 a s3 boolean je: " + rovnajuSaEquals);  
}

a výstup z konzoly:

*s1 a s2 boolean je: true*

*s1 a s3 boolean je: false*

*s1 a s3 boolean je: true*

ak porovnávame s1 == s2 vysledok je true. Dôvod:

1. String „jano“ je rovnaký a zároveň

2. s1 a s2 odkazujú na to rovnaké miesto v pamäti.

Objekty sú identické.

Ak porovnáme s1 == s3 výsledok je false. Dôvod:

Sice String „jano“ je rovnaký ale s1 a s3 ukazujú( odkazujú) na rôzne miesta v pamäti.

Z toho dôvodu pre porovnávanie obsahu String ( ale aj pri porovnávani iných objektov , nie primitívnych dátových typov) je lepšie použiť metódu *equals()*.

**Uloha: zsitit a popísať co robia tieto metódy:**

s1.toLowerCase() // ahoj

s1.toUpperCase() // AHOJ

s1 + s2 // ahojahoi

s1.concat(s2) // ahojahoi

s1.replace('h', ‘H') // aHoj

s1.substring(2) // oj

s1.substring(2,3) // o

s1.charAt(2) // o

s1.indexOf(‘o') // 2

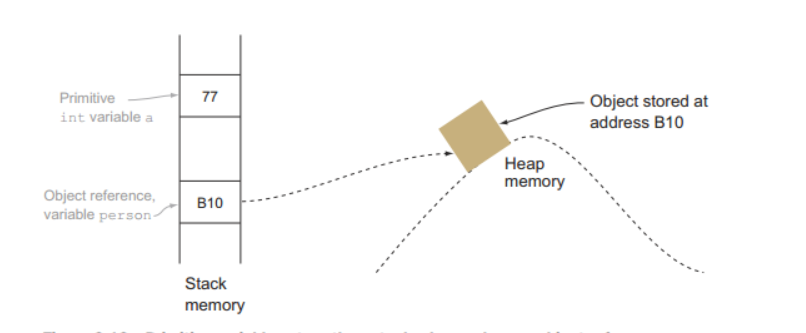
ukázať ako sa robí zo string int a opačne.

## StringBuilder class.

String objekty sú nemeniteľné. To znamená, že ak robime nejakú operaciu na objekte String napríklad zmeníme malé písmená na veľké, Java vytvorí nový objekt String.

Teda Java si alokuje nové miesto v pamäti a tam vloží nový String s velkými písmenami. Nemení String s malými písmenani. Alebo ak sa robí operácia taká, že sa stále s pôvodnému String pridáva nová časť. Tieto operácie sú náročné na zdroje keďže sa zakaždým musí vytvoriť nový String(alokuje sa pamäť). Preto pri operáciach s častými zmenami String je lepšie použit triedu StringBuilder.

StringBuilder stringBuilder = new StringBuilder("jano");

Primitivne dátové typy vs referenčné typy:   


Na obrázku je ukázaný rozdiel medzi primitívnym dátovým typom-premennou a referenčnou premennou. JVM má pamäť ktorá je rozdelená na niekoľko častí. Jedna čast sa vola stack pamäť. Toto je rýchla pamäť. Do tejto pamäte sa ukladajú primitívne dátové typy priamo. Teda v stack pamäti je uložená hodnota int 77. Naproti tomu String v stack pamati má uloženú len adresu do Heap pamäte. Dôvod je ten že pri int máme pevne dannú hodnotu, 32b, a tá sa lahko dá alokovať v stack pamäti. Naproti tomu, String môže mať jeden znak alebo aj pol strany textu. Z toho dôvodu je pol strany textu uložena v Heap pamäti a do stack pamati je uložená len adresa do Heap. A rovnakým spúsobom sa v JVM pamäti ukladajú aj ostatné objekty. Lebo nevieme ako ten objekt môže byť veľký.

# Operátory a ich priorita :

Operátory sú symboly,ktoré sa používajú na vykonávanie matematických alebo logických operácaii.

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Type** | **Associativity** |
| 15 | () | Parentheses | Left to Right |
| 14 | ++ -- | Unary post-increment Unary post-decrement | Right to left |
| 13 | ++ -- + - ! ( *type* ) | Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary type cast | Right to left |
| 12 | \*  /  % | Multiplication Division Modulus | Left to right |
| 11 | + - | Addition Subtraction | Left to right |
| 9 | < <= > >= | Relational less than Relational less than or equal Relational greater than Relational greater than or equal | Left to right |
| 8 | == != | Relational is equal to Relational is not equal to | Left to right |
| 4 | && | Logical AND | Left to right |
| 3 | || | Logical OR | Left to right |
| 2 | ? : | Ternary conditional | Right to left |
| 1 | = += -= \*= /= %= | Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment | Right to l |

Operátory sa používajú na vykonávanie operácii s premennými a ich hodnotami.

Operátory v Jave rozdelujeme na tieto skupiny:

Aritmetické operátory

Operátory priradenia

Operátory pororvania

Logické operátory

Bitové operátory

### Aritmetické operátory:

Operátory Binárne

|  |  |  |
| --- | --- | --- |
| Oper. | Použitie | Popis |
| + | op1 + op2 | súčet op1 a op2 |
| - | op1 - op2 | rozdiel op1 a op2 |
| \* | op1 \* op2 | súčin op1 a op2 |
| / | op1 / op2 | podiel op1 a op2 |
| % | op1 % op2 | zostatok po delení op1 operandom op2 |

Operátory unárne

Unárne operatory sa vzkonávajú skôr ako binárne operatory.

|  |  |  |
| --- | --- | --- |
| Oper. | Použití | Popis |
| + | +op | indikace kladné hodnoty |
| - | -op | aritmetická negace operandu |
| ++ | op++ | inkrementace op o 1 po jeho vyhodnocení |
| ++ | ++op | inkrementace op o 1 před jeho vyhodnocením |
| -- | op-- | dekrementace op o 1 po jeho vyhodnocení |
| -- | --op | dekrementace op o 1 před jeho vyhodnocením |

používajú sa pri vyšovaní alebo zmenšovaní o jedna. Môžeme dvojznamienko použiť pred premennou alebo za premennou.

int x = 1;  
int y = 2;

int z = x++ + y;

int x = 1;

int u = ++x + y;

Pri vyššie uvedenom príklade z = 3, a u =4. V prípade hodnoty z sa najprv sčítajú x a y a potom sa pripočíta k x jedna. Teda na konci prikladu hodnoty premených budú:

x = 2 ; y = 2 ; z = 3;

U premennej z sa jedná o príklad **post**-increment.

U premennej u sa jedná o príklad pre-increment kedy sa jedna prípočíta pred samotným sšítaním x + y. Výsledné hodnoty x a y v obidvoch prípadoch je rovnaký.

Relačné operátory:   
Relačné operátory porovnávajú dva výrazy ekvivalentných typov a rozhodnú o platnosti vzťahu medzi nimi. Výsledkom relačného operátoru je hodnota boolean ( true, false).

|  |  |  |
| --- | --- | --- |
| Oper. | Použití | Výsledek je true jestliže |
| > | op1 > op2 | op1 je väčší než op2 |
| >= | op1 >= op2 | op1 je väčší než alebo rovný op2 |
| < | op1 < op2 | op1 je menší než op2 |
| <= | op1 <= op2 | op1 je menší než alebo rovný op2 |
| == | op1 == op2 | op1 a op2 sú si rovnaké |
| != | op1 != op2 | op1 a op2 si nie sú rovnaké |

## Logické operatory

Na kombináciu relačných operátorov je nutné použiť logické operátory. Napríklad výraz op1 < op2 < op3 je potrebné zapístať ako logickú kombináciu dvoch relačných operátrov (op1 < op2) && (op2 < op3).

|  |  |  |
| --- | --- | --- |
| Oper. | Použití | Výsledok je true ak: |
| && | op1 && op2 | op1 a op2 nabývajú hodnotu true |
| || | op1 || op2 | aspoň jeden z op je true |
| ! | !op | op nadobúda hodnotu false (negácia) |

V Jave sú tri logické operátory: ||(OR), &&(AND), !(NEGACIA).

Hoci sa nazývajú „logické“, môžu sa použiť na hodnoty akéhokoľvek typu, nielen na booleovské hodnoty. Ich výsledok môže byť tiež akéhokoľvek typu.

## [|| (OR)](https://javascript.info/logical-operators" \l "or)

| - overí obidva výsledky logickej operácie

**| |**- prestane vyhodnocovať, ak prvý výsledok logickej operácie vyhodnotí hodnotu TRUE, tak výsledok bude pravdivý

result = a || b;

V klasickom programovaní má logický OR slúžiť iba na manipuláciu s booleovskými hodnotami.

Existujú štyri možné logické kombinácie:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Ako vidíme, výsledok je vždy TRUE okrem prípadu, keď sú obidva FALSE.

## [&& a & (AND)](https://javascript.info/logical-operators#and)

& - overí obidva výsledky logickej operácie

**&&**- prestane vyhodnocovať, ak prvý výsledok logickej operácie vyhodnotí hodnotu FALSE, tak výsledok bude nepravdivý

result = a && b;

V klasickom programovaní má logický OR slúžiť iba na manipuláciu s booleovskými hodnotami.

Existujú štyri možné logické kombinácie:

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

## 

## !(NEGACIA)

Boolovský operátor NOT je vyjadrený výkričníkom !.

result = !value;

Postup:

1. **Prevedie operand boolean: true/false.**
2. **Vráti inverznú hodnotu.**

Napríklad:

alert( !true ); // false

alert( !0 ); // true

**^(XOR) (nepovinné)**

=bitové exkluzívne OR

Existujú štyri možné logické kombinácie:

alert( true || true ); // false

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Operátor priradenia:

Pre operátor priradenia sa používa znak =. Tento operátor je možné použiť aj pre inicializáciu premenných pri deklarácii lokálnej premennej.

Pri priradení hodnoty do referenčnej premennej sa objekt alebo pole nekopíruje , ale priradí sa len referencia. Vlastný objekt alebo pole zostáva v jedom exemplári. Teda Samotný exemplár je len jeden v Heap Pamäti ale vznikli dve premenné v Stack pamäti, ktoré obsahujú rovnakú adresu do Heap pamäte. Referenčnej premennej môžme priradiť neplatnú referenciu *null* ( teda nelatnú adresu do heap pamäte).

|  |  |  |
| --- | --- | --- |
| Oper. | Použití | Ekvivalent |
| += | op1 += op2 | op1 = op1 + op2 |
| -= | op1 -= op2 | op1 = op1 - op2 |
| \*= | op1 \*= op2 | op1 = op1 \* op2 |
| /= | op1 /= op2 | op1 = op1 / op2 |
| %= | op1 %= op2 | op1 = op1 % op2 |

Ternárny oprátor:

Ternárny operátor je takou obodobou príkazu if / else.

**výraz1 ? výraz2 : výraz3**

**Ak výraz1 nadobúda hodnoty true, vyhodnotú sa výraz2 v opačnom prípade sa vyhodnotí výraz3.**

String vysledok1 = (i==0) ? " i je rovne 0" : " i je nerovne 0";  
  
System.*out*.println(vysledok);

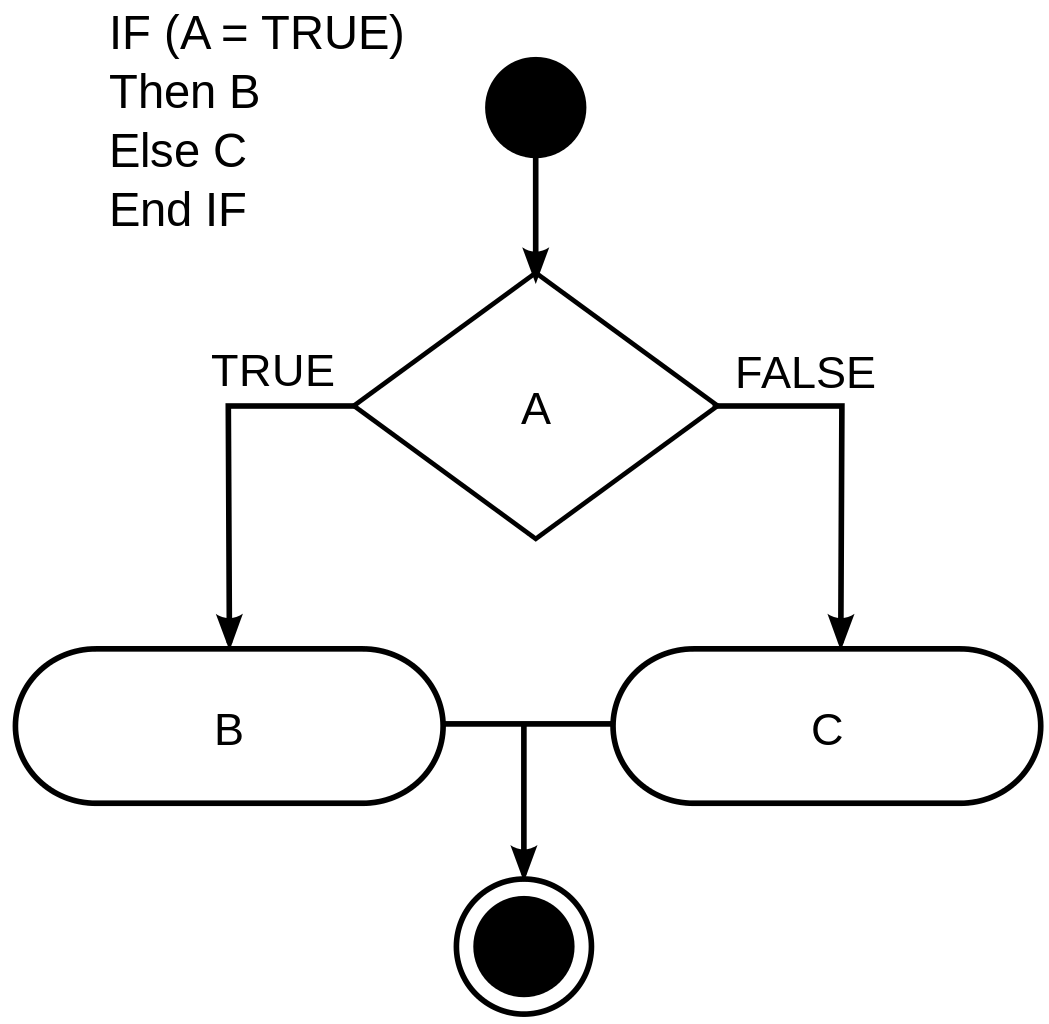
**b != 0 ? c = a / b : c = 0 ak je b nenulové urobí sa a/b. Ak je b =0 tak c sa priradí : c=0;.**

# Komentáre:

Komentáre sa sa píšu:



# If / Else



Syntax:

if( boolean )

{ nejaký kód }

else

{nejaký kód}

Ak je potrebné vložiť viacero podmienok je možné použiť konštrukt if - else if – else.

else if je možné pridat teoreticky nekonečno 😊

if( bollean )

{ nejaký kód }

else if (boolean)

{nejaký kód}

else if (boolean)

{nejaký kód}

else{

nejaký kód

}

Treba zachovať logickú štruktúru:

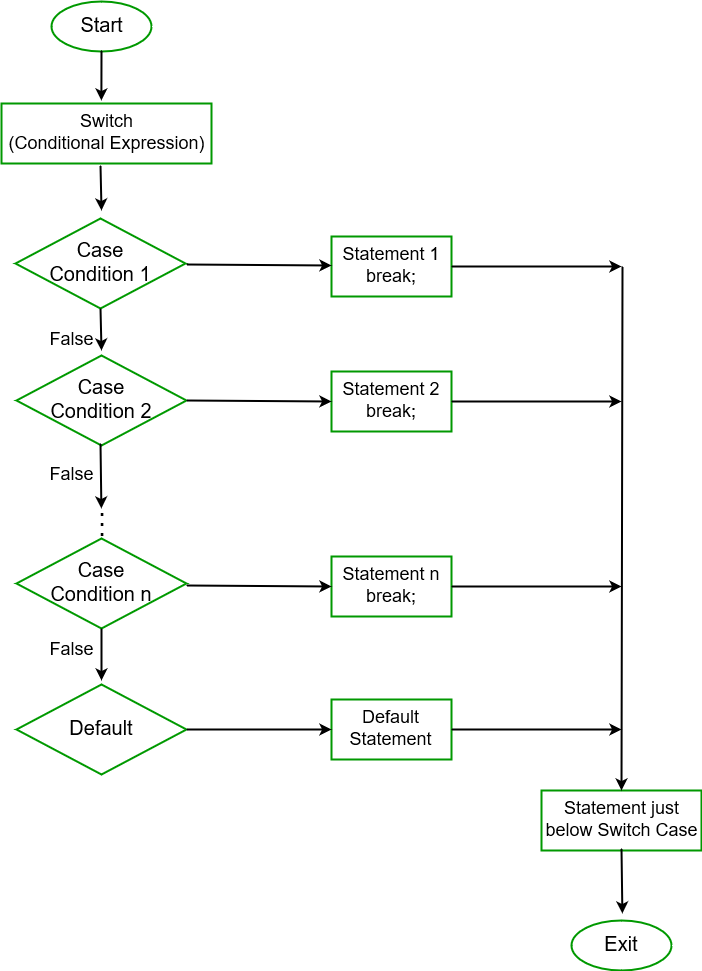
Príklad zlej logiky:



Vyššie uvedený príklad ukazuje, že je potrebné zachovať logicku postupnost. Aby kód pracoval správne podmienka ( hodina < 12 ) má ísť ako prvá. Inak pri hodina=10 výsledok bude popoludní lebo ide prvá kontrola a to 10 < 15 a výsledok je “popoludní”.

## Switch statement

Switch pracuje s : byte, short, char, int a String. Pre zvedavých: Je možné použiť aj triedu Enum. Je podobný k if else.





Môžeme to prepísať:



Switch (expression){

case value1:

block of code;

**break;**

case value2:

block of code;

**break;**

case value3:

block of code;

**break;**

default;

block of code;

}

Break; je dôležitý lebo ak sa tam nedá break tak sa ignoruje dalši case value a pokračuje v behu programu.



Uprava:



Switch expresisons : char, byte, short, int

String, enum

Switch sa rebi sekvenčne, tj postupne.

Else if sa používa pri testoch rozahov alebo pri testovani podmienok

Swich sa použiva pri testovani kde je porovnvanie zalozené na jednom prvku.

# Cykly

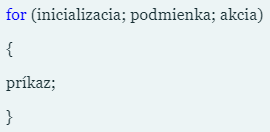
Pokial potrebujeme, aby sa určitá čast programu opakovala použijeme určitú formu cyklu. V jazyku java môžme zapísať cyklus pomocou while, do/while a rôznymi variaciami for cyklu. Pri cykloch je potrebné zaistiť začiatok cyklu a potom treba zaistiť koniec cyklu. Pokiaľ nebude zaistený koniec cyklu tak sa nám program ocitne v nekonečnej slučke a program nám teoreticky nikdy neskončí.

Koniec cyklu musíme zaistit tak že pri každom začiatku cyklu sa bude kontrolovať nejaká nami zadaná podmienka, ktorá rozhodne či danný cyklus bude znova spustený alebo program už preskočí danný cyklus a bude pokračovať v behu porgramu. Beh cyklu je možné ovplyvňovať aj iným spôsobom a to výrazmi *break;* a *continue;*, ktoré sa použijú v tele cyklu.

## For

Prvý typ cyklu si predstavíme for cyklus. Tento typ cyklu, ako uvidíme v jeho syntaxe nám už ukazuje aké parameter by sme mali nastaviť tak, aby for cyklus fungoval štandartne.

Jeho definicia vyzerá:



**inicializácia** – tu si vytvoríme premennú, ktorú sme schopní testovať v druhej časti parametrov cyklu.

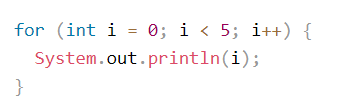
**podmienka** – vyjadruje podmienku od ktorej záleží či sa nám daný cyklus (opakovanie) vykoná

**akcia** – tu je možné zmeniť inicializačnú premennú tak aby sme boli schopní meniť výsledok podmienky.

Program beží postupne a keď narazí na cyklus jeho beh sa preruší a sústredí na cyklus. Začne sa vykonávať telo cyklu a to toľkokrát kým platí podmienka cyklu.

Obyčajne do inicializácie sa vytvorí typ premennej int. Ľahko sa mení jej hodnota v časti akcia, a aj vytvaranie podmienky je jednoduché. Vtedy vlastne porovnávame dve čísla. Vid. príklad.

Príklad:



## FOR EACH

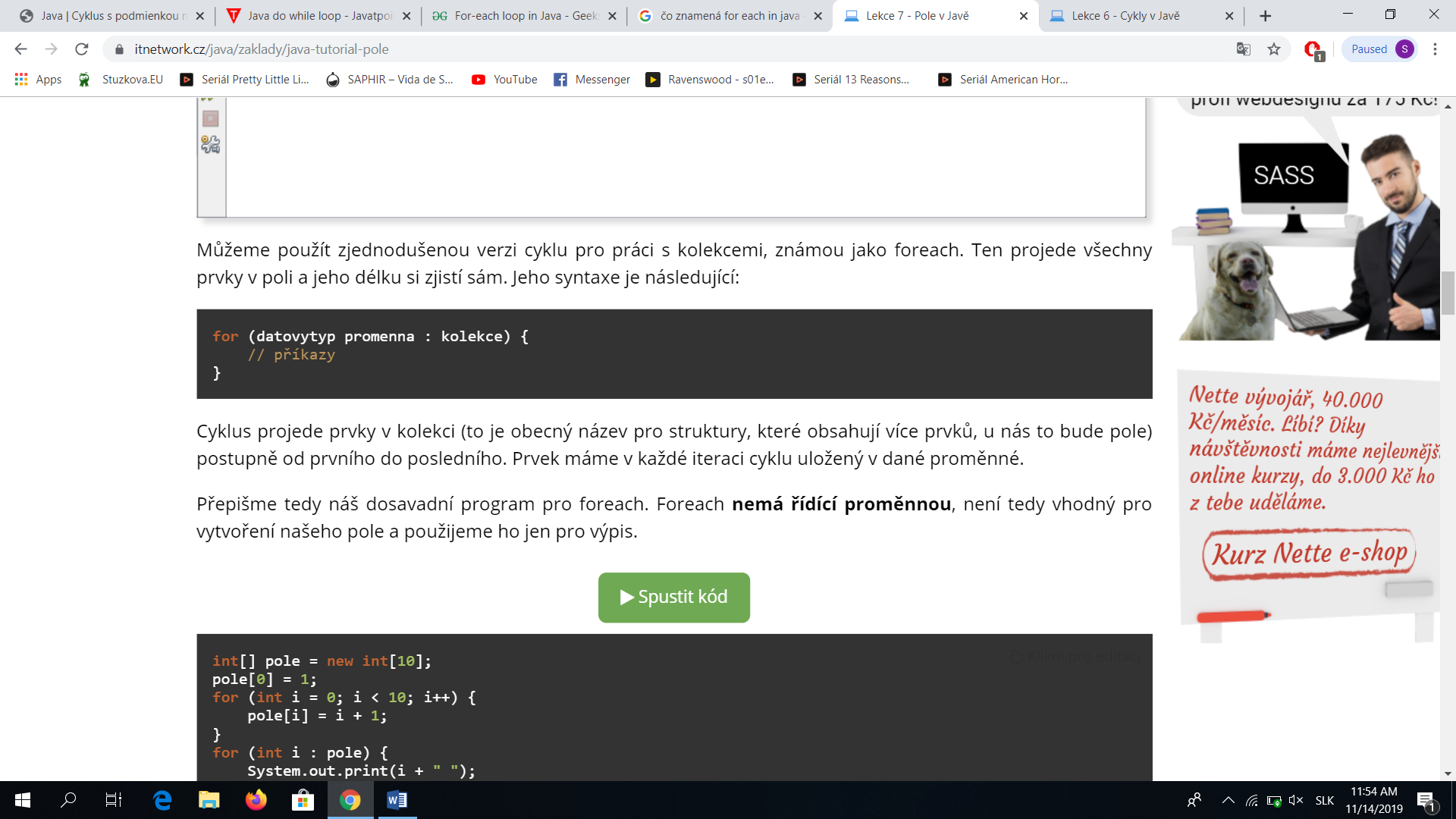
Je špeciálny typ for cyklu, ktorý sa využíva u polí.

Rozdiel medzi foe a for each:

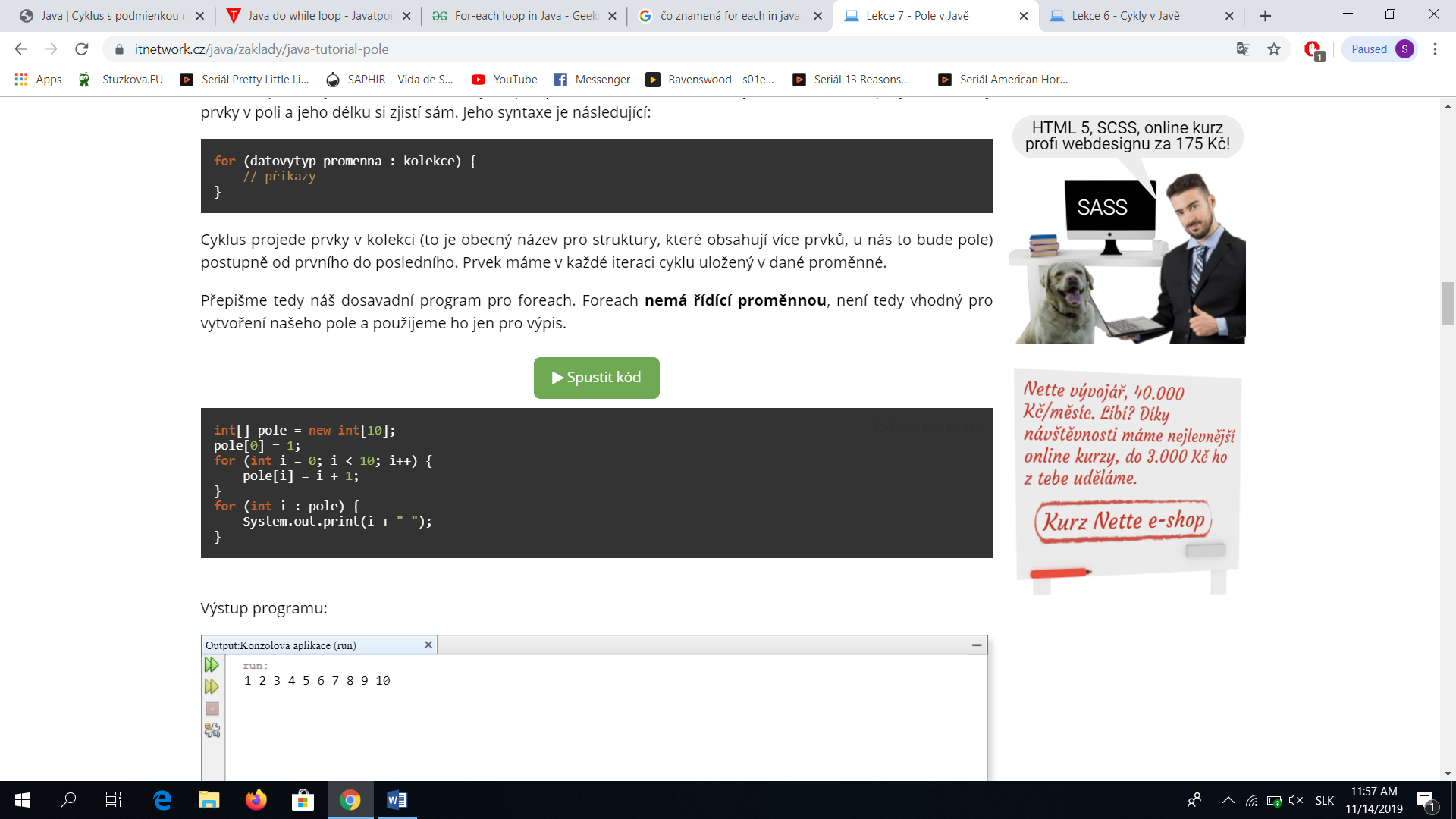
For- JA si musím zadať počet prvkov a teda celú dĺžku pola(length).

For-each je zjednodušený for. Ten si prejde všetky prvky v poli a jeho dĺžku si zistí SÁM.

Syntax:



Rozdiel v použití medzi for a for-each:



## While

Testovanie podmienky sa vykonáva na začiatku cyklu. Pred každou iteráciu je podmienka testovaná, či bola splnená. Už pred prvou spustením dôjde k otestovaniu podmienky, použitie tohto cyklu je vynikajúci v situáciách kedy možno že ani raz nebude treba vykonať cyklus.

Možné použitie Cyklus s podmienkou na začiatku:

public static void main(String args[]){  
  
 int i =1;  
  
 while (i<10){  
 System.*out*.println(1);  
 i++;  
 }  
  
}

Diagram

Description automatically generated

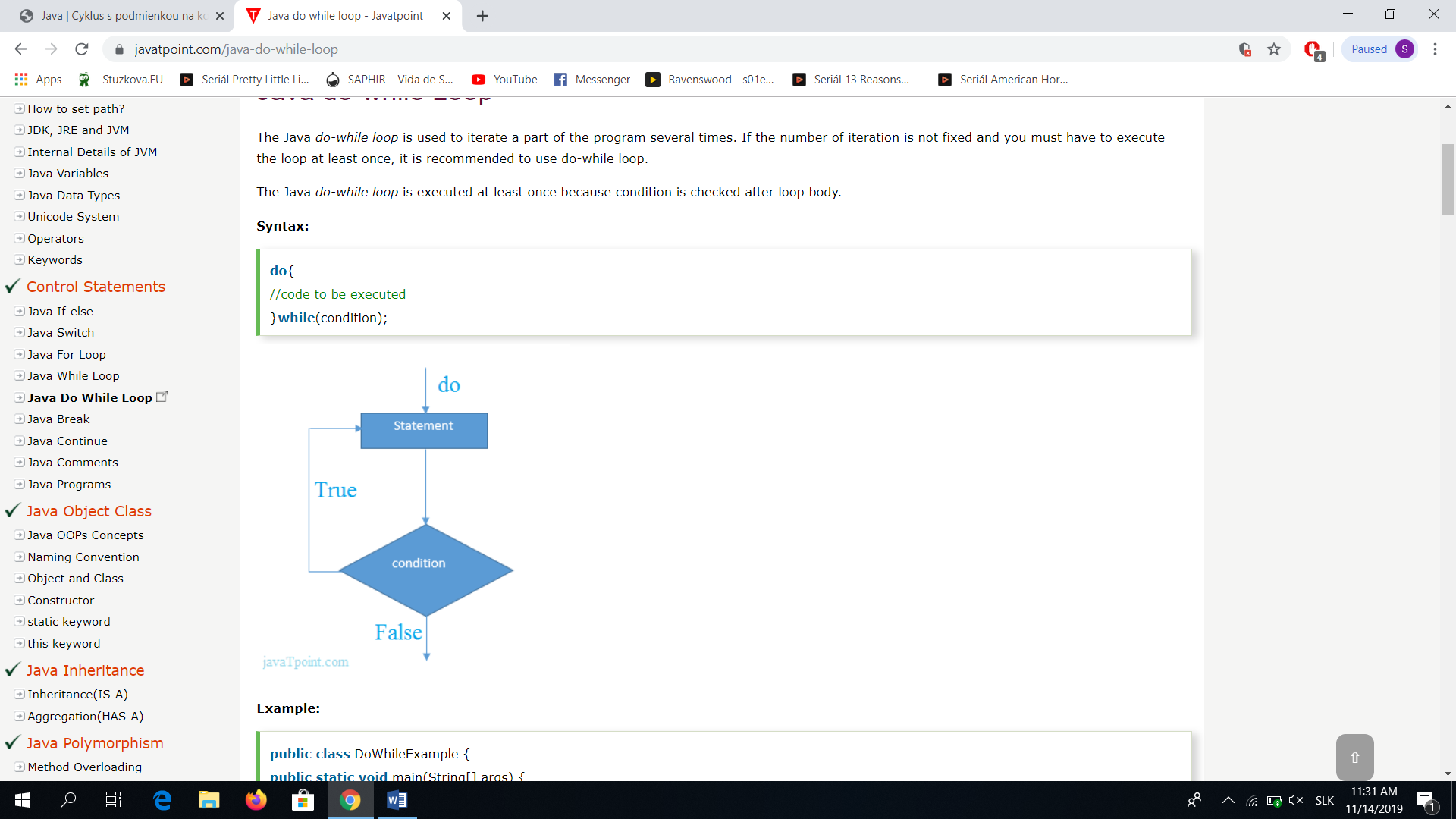
## DO WHILE

Pri do while cyklus prejde minimálne raz, a prebieha pokial je podmienka true.

Príklad:

public static void main(String args[]){  
   
 int i =1;  
   
 do{  
 System.*out*.println(i);  
 }while (i<10);  
  
}

Syntax



### Príkazy break a continue

Príkazy break a continue sú *skokovými* príkazmi, ktoré sa používajú na preskočenie niektorých príkazov vo vnútri cyklu alebo na okamžité ukončenie cyklu bez kontroly testovacieho výrazu. Tieto príkazy môžu byť použité vo vnútri akýchkoľvek slučiek, ako napríklad pre slučku for, while, do-while.

Break: Príkaz break v jave sa používa na okamžité ukončenie cyklu. Keď sa vo vnútri slučky vyskytne príkaz break, iterácia slučky sa zastaví a riadenie sa vráti zo slučky okamžite do prvého príkazu za slučkou. Príkazy break sa v podstate používajú v situáciách, keď si nie sme istí skutočným počtom iterácií cyklu, alebo ak chceme cyklus ukončiť na základe nejakej podmienky.

V Jave sa break hlavne používa:

* Ukončenie slučky behu programu.
* Ukončenie sekvencie v switch príkaze.
* Ukončenie behu programu/podprogramu

Pomocou break môžeme vynútiť okamžité ukončenie cyklu, obídenie podmieneného výrazu a zvyšného kódu v tele cyklu. Keď použijeme zlom vo vnútri vnorených slučiek, vylomí sa iba z najvnútornejšej slučky.

public static void main(String[] args)  
{  
 // cyklus ,ktory je nastaveny na 10 cyklov 0 - 9   
 for (int i = 0; i < 10; i++) {  
 // Ukoncenie celeho cyklu ked i je 5   
 if (i == 5)  
 break;  
 System.*out*.println("i: " + i);  
 }  
 System.*out*.println("Out of Loop");  
}

Príkaz continue v Jave sa používa na preskočenie aktuálnej iterácie slučky. Príkaz continue môžeme použiť vo všetkých typoch slučiek, ako sú cykly for, while a do-while. V podstate sa príkazy pokračovania používajú v situáciách, keď chceme pokračovať v cykle, ale nechceme zostávajúci príkaz po príkaze pokračovania.

Pomocou continue môžeme preskočiť aktuálnu iteráciu slučky a okamžite prejsť na ďalšiu iteráciu slučky.

public static void main(String args[])  
{  
 for (int i = 0; i < 10; i++) {  
 // Ak je cislo 2 tak sa ukonci akualna   
 // iteracia a pokracuje do konca teda do i = 9  
 if (i == 2)  
 continue;  
  
 System.*out*.print(i + " ");  
 }  
}

# Array

Vytvorenie pola typu int:

int[] mojePole1 = new int[4];

int[] mojePole2 = new int[]{4,6,7};

int[] mojePole3 = {6,7,8}

Lomené zátvorky [] definujú typ premennej ako pole. Pole môže uchovávať len jeden dátový typ a to je ten ktorý definujeme pri deklarácii. Teda: int[] môžeme uložiť len premenné int.

String[] len typ string atd.

Jednotlivé elementy pola sa indexujú od číslovky 0. Teda ak máme pole s prvkami {6,7, 8}.

tak prvok 6 má index 0, prvok 7 má index 1 atď.

Na prácu s poľami je Trieda Arrays. Tá má rôzne statické metódy.

Arrays.toString(mojePole); vypíše pole

Arrays.equals(pole1, pole2); porovnáva dve polia.

Arrays.sort(pole); zoradí pole.

K jednotlivým elementom poľa pristupujeme nasledovne:

int element = mojePole2[2] ;

priradenie elementu na index:

mojePole3[2] = 3; toto zmení hodnotu v poli na indexe 2 na číslo 3.

Veľkosť pola zistíme mojePole1.length. Tu pozor, voláme premennú a nie metódu tak na konci sa nedávajú zátvorky.

Prechádzanie elementov poľa môžeme klasickým for, alebo môžeme použiť foreach

for( dátovýtyp premenná : pole ){

}

for( int element: mojePole1 ){ }

Pri prechádzaní poľa s foreach nemôžeme meniť prvky poľa. Možeme to urobiť klasickým for

String môžeme premeniť na pole a to: nejakýString.toCharArray();

Vyhladavanie v poli sa robí metodou Arrays.binarySearch(pole, hladanaHodonota);

## ArrayList

ArrayList<Object> pole = new ArrayList<>();

List<Object> pole = new ArrayList<>();

List je interface a ArrayList je len implenetacia List Interface. Polymorfyzmus.

ArrayList môže uchovávať akýkoľvek objekt. Nemôže uchovávať primitívne dátové typy. ArrayLIst je dynamický Array. Teda môžeme meniť veľkosť poľa.

List list = new ArrayList(); tu sa ukladajú objekty ako typ Objekt.

Metódy: add(), remove(), set(), toString();

add() - overloaded metóda, remove() podla indexu alebo podľa objektu, set() mení objekt na dannom indexe.

Pozor na index out of bounds.

IsEmpty(), size(), clear(), contains(), equals()

Konverzia medzi poliami[] a Listami:

String poleString[]= new String[]{"janko","jozko","martinko"};  
  
List<String> premnenyZoString = Arrays.*asList*(poleString);

List<String> list = new ArrayList<>();  
list.add("janko");  
list.add("jozko");  
list.add("petko");  
list.add(1,"lubos");  
  
String[] premenenyZList = list.toArray(new String[4]);

Pri zmenie z List na pole[] treba pri použití metódy toArray() udať parameter metódy nový objekt pola. V našom prípade je to pole String[]. Treba mať na pamäti, že pri vztváraní poľa musíme zadať aj jeho veľkosť. A to takto **new String[** číslo, ktoré určuje veľkosť poľa **]**. Číslo získame tak, že si spočítame veľkosť listu. Najlepšie metodou list.size(). A do parametra metódy môžme zapísať :

**new String[list.size()]**

Ako už bolo spomenuté nemôžeme do Listu ukladať primitívne dátové typy. Preto sú v Jave takzvané Wrapper types. Sú to triedy podľa, ktorých vznikajú objekty, ktoré uchovávajú primitívne dátové typy.

## Primitive Wrapper Types



Pozor na to, že Wrapery môžu uchovávať null.

Konvertovanie z ArrayList do Array a opačne.

ArrayList.toArray().

String[] pole1 = polelist.toArray(new String[0]);

List<String> zasePole = Arrays.*asList*();

toto vracia List pevnej veľkosti. Add() a remove() nefunguje.

Sorting rovnaké ako u Array.

Collections.sort(), Collection.binarySearch(),

Iteracie v Listu.

Keď chceme meniť elementy v listu, napríklad zmazať nejaký element. Java vyhodí chybu.

For a forEach.

Riešenie: Trieda Iterator, ListIterator.

List<String> pole = new ArrayList<>();  
pole.add("jedna");  
pole.add("dva");  
  
for(Iterator<String> iter = pole.iterator(); iter.hasNext();){  
 String element = iter.next();  
 System.*out*.println(element);  
 iter.remove();  
}

# Triedy

**Trieda** je udajovy typ definovany uzivatelom. Reprezentuje skupinu objektov, ktore maju spolocne vlastnosti. Pri vzniku triedy sa nealokuje pamet (az na cleny static). Trieda je len taký plán pre samotný objekt, ktorý sa vytvorý na základe triedy-plánu. Trieda definuje vlastnosti a spravanie objektu, ktorý je na základe triedy vytvorený.

Možme si predstatiť triedu ako projektovú dokumentáciu pre dom. Dom ešte neexistuje ale jeho plány sú. To je ta trieda napríklad Dom.class. Je to plán pre nejaký objekt. Kľudne aj pre dom. Až keď sa projektová dokumentácia dostane do rúk murárovi tak až potom vznikne dom. V našej analógii, murár znamená ked použijeme: *Dom prvyDom = new Dom().* A na základe jedného projektu sme schopní vytvorit nekonečno domov. Tak isto ako na základe jednej triedy sme schopní vytvorit nekonečno objektov java typu Dom.

Trieda vyzera napriklad takto:

public class Kruh {  
 double polomer;  
  
 public Kruh(){}  
  
 public Kruh( double vstupnyAtribut){  
 if(vstupnyAtribut > 0){  
 polomer = vstupnyAtribut;  
 }else{  
 polomer = 0;  
 }  
  
  
 }  
 public double obvod(){  
 return 2\*polomer\*Math.*PI*;  
 }  
 public double obsah(){  
 return Math.*PI*\*polomer\*polomer;  
 }  
  
 @Override  
 public String toString() {  
 return "Kruh{" +  
 "polomer=" + polomer +  
 '}';  
 }  
  
}

**Inštancia triedy** je niečo ako premenna triedy. Na rozdiel od triedy ma alokovanu pamet. Niekedy instanciu triedy volame aj **objekt**. Pre nas budu tieto dva terminy uplne rovnocenne.

**Člen triedy**. Tento pojem som si zaviedol sam. Oznacuje datove cleny a metody triedy. Ako je vidko z príkladu triedy Kruh tak členovia triedy sú:

Atributy – Kruh má jeden a to double polomer.

Konštruktory – Tu sú dva defautný public Kruh() {} , a nami vyrvorený konštruktor

Metódy – obvod(), obsah() , toString()

**Konštruktor:** Je to metoda, ktoru ma kazda trieda. Od ostatnych metod sa lýši tym, ze ma take iste meno ako je meno triedy. Vola sa pri vytvoreni instancie triedy. V pripade, ze nevytvorime konstruktor my, bude vytvoreny implicitny konstruktor interpreterom jazyka. Takyto konstruktor je velmi primitivny a vacsinou nesplna nase poziadavky, preto je potrebne vytvorit iny, ktory by dokazal spracovat, napriklad, vstupne (inicializacne) argumenty.

Implicitný-defaultný konšruktor vytvorí objekt dannej triedy ale keď my si nezvolíme hodnoty aributov sami tak java pri vytváraní objektu dobplní defaultné hodnoty do jednotlivých atributov.

Pri primitivných datových typov to sú: 0 alebo 0.0 boolean je to false. Ak je atribut dalši objekt tak v prenemennej bude hodnota null.

**Getters Setters.**

Sú to metódy triedy, ktoré sú potrebné pri nastavení atributov (setters) a pri získaní hodnot (getters), keď používame zapúzdrenie (encapsulation). Pri nastaveni inej hondoty prístupoveho parametra ako public, je treba (teda ak si to trieda vyžaduje) vytvoriť metódy getters/setters.

*private String meno;*

*public String getMeno(){*

*return meno;*

*}*

*public void setMeno(String meno){*

*this.meno = meno;*

*}*

## Staticke datove cleny a metody

Kazdy staticky datovy clen triedy predchadza rezervovane slovicko static. Napriklad:

class TriedaA {

static int pocet\_instancii;

public TriedaA() { pocet\_instancii++ }

static int get\_pocet() { return pocet\_instancii }

}

**Staticke datove cleny** by sme mohli nazvat aj ako premenne triedy, pretoze *kazda staticka premenna je viazana k triede a nie k objektu!*

Cize staticka premenna vznika pri definovani triedy a nie pri vytvoreni instancie danej triedy.

(nová inštancia sa vytvorý slovíčkom **new**)

Znamena to, ze ak vytvorime niekolko objektov triedy, ktora obsahuje staticky datovy clen, potom vsetky tieto objekty zdielaju dany staticky clen ako spolocnu pamet, do ktorej mozu pristupovat pomocou metod.  
**Staticka metoda** je metoda, ktorej predchadza klucove slovo static. *Staticka metoda moze pristupovat len k statickym premennym!*  
Tieto vlastnosti sa daju vyuzit na predavanie informacii medzi objektami tej istej triedy (okrem ineho). Napriklad ak chceme kontrolovat pocet instancii danej triedy, tak jednoducho v konstruktore tejto triedy inkrementujeme staticku premennu, ktoru sme na to vyclenili. Tymto sposobom ziskame prehlad o tom, kolko instancii danej triedy sa prave nachadza v pameti. V praxi to vyzera takto:

class TriedaB {

// datovy clen (staticky)

static int pocet\_instancii = 0;

// konstruktor

public TriedaB() { pocet\_instancii++; }

// destruktor

protected void finalize() { pocet\_instancii--; }

// staticka metoda

static int get\_pocet() { return pocet\_instancii; }

}

## Abstraktná trieda

Abstraktné triedy sa využívajú na definovanie spoločných charakterových rysov podtried. Z abstraktnej triedy sa nedá vytvoriť inštancia (objekt). Môže použiť len ako generická trieda pre ostatné triedy, ktoré ju budú rozširovať.

### Definícia abstrakcie

Abstraktná trieda sa deklaruje pomocou kľúčového slova *abstract* Abstraktrné triedy sa využívajú ako šablóny pre konkrétne podtriedy. Tak ako Každá iná trieda, aj abstraktrná trieda môže obsahovať vlastnosti a metódy, ktoré definujú správanie triedy.

Abstraktná trieda môže taktiež obsahovať metódy bez implementácie. Takéto metódy sa nazývajú **abstraktné metódy**. Deklarácia abstraktnej metódy musí začínať kľúčovým slovom abstract a za názvom funkcie (a parametrami) nesmie byť telo, ale musí tam byť bodkočiarka.

Ak trieda obsahuje nejaké abstraktné metódy, buď deklarované v aktuálnej triede, alebo zdedené, musí byť deklarovaná ako abstraktná. Abstrakné metódy poskytujú určitú šablónu pre triedy, ktoré sú z pôvodnej (abstraktnej) triedy odvodené.

Z abstraktných tried sa nedajú vytvárať objekty. Abstaktné triedy musia byť použité ako triedy rodičov. Triedy, ktoré sú potomkami abstraktnej triedy buď musia abstraktné metódy implementovať, alebo ich zdedia ako abstatkné a samotná trieda potomka sa stáva abstraktnou[[1]](http://www.kiwiki.info/index.php/Java_-_abstraktn%C3%A9_triedy" \l "cite_note-1).

Príklad abstraktnej triedy:

abstract class GraphicObject {

int x, y;

...

void moveTo(int newX, int newY) {

...

}

abstract void draw();

abstract void resize();

}

Praktické použitie abstraktnej triedy: Zoberme si príklad so živočíšnymi druhmi: Zviera-Cicavec-Mačka-Plaz-Vtak. Keďže nemá zmysel vytvárať obejekt typu Zviera, triedu zviera budeme definovať ako abstraktnú:

public abstract class Zviera {

public String nazov;

public int vaha;

public Zviera()

{

this("Zviera",0);

}

public Zviera(String nazov, int vaha) {

this.nazov = nazov;

this.vaha = vaha;

}

// abstaktné metódy

public abstract void pohniSa(int k);

public abstract void prijmiPotravu(int mnozstvo);

public abstract void rozmnozujSa();

}

Teraz je trieda Zviera abstraktná a už nie je možné vytvárať objekt typu Zviera (Zviera z= new Zviera() ). Ale je možné použiť odkaz ma obejkt typu Zviera pri použití polymorfizmu ( Zviera p= new Plaz() ). Na to, aby celý náš zverinec fungoval treba doplniť triedy Cicavec, Pes, Plaz a Vtak o všetky abstraktné metódy, ktoré zdedili.

Trieda Cicavec po doplnení vyzerá nasledovne:

public class Cicavec extends Zviera{

public int dlzka\_gravidity;

public String typ\_srsti;

public Cicavec() {

super("Cicavec",1);

this.dlzka\_gravidity = 7;

this.typ\_srsti = "";

}

// ostatné metódy sú nezmenené

@Override

public void prijmiPotravu(int mnozstvo) {

System.out.println("Cicavec zozral "+mnozstvo+" kg potravy");

}

@Override

public void rozmnozujSa() {

System.out.println("Cicavec sa rozmnožuje");

}

}

Podobne triedy Vtak a Plaz budú vyzerať nasledovne:

public class Vtak extends Zviera {

public float rozpatie\_kridel;

public int pocet\_vajec;

public Vtak()

{

super("Vtak",3);

this.rozpatie\_kridel = 0;

this.pocet\_vajec = 8;

}

// ostatné metódy sú nezmenené

@Override

public void prijmiPotravu(int mnozstvo) {

System.out.println("Vtak vyzobal "+mnozstvo+" kg zrna");

}

@Override

public void rozmnozujSa() {

System.out.println("Vtak sa rozmnožuje kladenim vajec");

}

}

public class Plaz extends Zviera {

public float dlzka;

public String typ;

public Plaz(String nazov, int vaha, float dlzka, String typ) {

super(nazov, vaha);

this.dlzka = dlzka;

this.typ = typ;

}

// ostatné metódy sú nezmenené

@Override

public void prijmiPotravu(int mnozstvo) {

System.out.println("Plaz prehltol "+mnozstvo+" cicavcov");

}

@Override

public void rozmnozujSa() {

System.out.println("Plaz sa rozmnožuje");

}

}

## Rozhranie – Interface

Rozhranie v jazyku Java je podobné triede.Všetky metódy v rozhraní sú abstraktné metódy. Podobne ako abstraktná trieda, rozhranie nemôže byť inštancované. Rozhranie môže byť použité pre abstrakciu, ako aj pre implementáciu viacnásobného dedičstva. Rozhranie je deklarované pomocou kľúčového slova rozhrania. Trieda, ktorá implementuje rozhranie, by mala implementovať všetky metódy deklarované v rozhraní. Pozri nižšie uvedený príklad, ktorý ilustruje použitie rozhrania na dosiahnutie abstrakcie.

## Rozdiel medzi abstraktnou triedou a rozhraním v jazyku Java

### definícia

Abstraktná trieda je trieda deklarovaná abstraktným kľúčovým slovom, ktoré je súhrnom abstraktných a netradičných metód. Naopak, rozhranie v jazyku Java je referenčným typom, ktorý je podobný triede, ktorá je kolekciou abstraktných metód. Toto je základný rozdiel medzi abstraktnou triedou a rozhraním v jazyku Java.

### premenné

Použité premenné predstavujú jeden rozdiel medzi abstraktnou triedou a rozhraním v jazyku Java. Hoci abstraktná trieda môže mať konečné, ne-finálne, statické a nestatické premenné, rozhranie môže mať len statické a konečné premenné.

### metódy

Ďalším rozdielom medzi abstraktnou triedou a rozhraním v Jave je, že abstraktná trieda môže mať abstraktné metódy a ne-abstraktné metódy, zatiaľ čo rozhranie môže mať len abstraktné metódy.

### Viacnásobné dedičstvo

Okrem toho abstraktné triedy nemožno použiť na implementáciu viacnásobného dedičstva. Rozhrania však možno použiť na implementáciu viacerých rozhraní. Preto môžeme povedať, že použiteľnosť je ako rozdiel medzi abstraktnou triedou a rozhraním v jazyku Java.

### Kľúčové slovo triedy

*Abstract* kľúčové slovo sa tiež používa na deklarovanie abstraktnej triedy, ale kľúčové slovo rozhrania *interface* sa používa na deklaráciu rozhrania.

### Rozšírenie a implementácia

Okrem toho je možné rozšíriť abstraktnú triedu pomocou kľúčového slova „extend“. Trieda rozhrania(interface) môže byť implementovaná pomocou kľúčového slova „implements“.

### uskutočnenie

Ďalej, abstraktná trieda môže byť implementovaná pomocou kľúčového slova „implements“, zatiaľ čo rozhranie sa používa na implementáciu abstrakcie, ako aj viacnásobného dedičstva.

### záver

Rozdiel medzi abstraktnou triedou a rozhraním v jave je, že abstraktná trieda sa používa na implementáciu abstrakcie, zatiaľ čo rozhranie sa používa na implementáciu abstrakcie, ako aj viacnásobné dedičstvo.

# Zapúzdrenie - encapsulation

– schovananie parametrov a metod aby neboli verejne prístupné.

* private - pomocou tohto slovička programátor špecifikuje, že člen triedy može byť použitý len v triede, kde je definovaný.
* protected - k clenom triedy, ktorym predchadza toto slovicko, je mozne pristupovat len v triede, kde su definovane a z tried, ktore vznikly dedením od tychto tried (potomkovia).
* public - urcuje, že k takýmto členom triedy maju pristup všetky triedy.
* bez typu – je to takzvaný package private. To je ak parametru alebo metóde nepridáme žiaden prítupový typ. Tento typ prístupu obmedzuje použitie metód a parametrov len v rámci balíčku(package).

# Dedičnosť - inheritance:

Trieda, ktorá je odvodená od inej triedy, sa nazýva podtrieda (tiež odvodená trieda, rozšírená trieda alebodcérska trieda). Trieda, z ktorej je podtrieda odvodená sa nazýva nadtrieda (tiež nadradená trieda, základná trieda,rodičovská trieda alebo supertrieda).

Okrem triedy Object, ktorá nemá nadtriedu, má každá trieda iba jednu priamu rodičovskú triedu (jednoduché dedenie). Pri absencii explicitnej nadtriedy, je každá trieda implicitne podtriedou triedy Object.

Triedy môžu byť odvodené od tried, ktoré sú odvodené od tried, a tak ďalej, až po konečnú najvrchnejšiu triedu Object. Rodokmeň každej triedy vedie teda k rovnakému praprapra... rodičovi — triede Object

Idea dedičnosti je jednoduchá, no silná: Keď chcete vytvoriť novú triedu a už existuje trieda, ktorá obsahuje časť kódu, ktorý potrebujete, môžete novú triedu odvodiť od existujúcej triedy. Ak tak urobíte, môžete znova využiť vlastnosti a metódy existujúcej triedy bez toho, že by ste ich sami museli písať (a ladiť).

Podtrieda zdedí členov (vlastnosti, metódy a vnorené triedy) od nadtriedy. Konštruktory nie sú členmi, preto nie sú podtriedami dedené, ale konštruktor nadtriedy môže byť z podtriedy volaný.

Príklad dedičnosti

Tu je príklad kódu možnej implementácie triedy Bicykel,

public class Bicykel {  
 // trieda Bicykel má tri vlastnosti  
 public int tempo;  
 public int prevod;  
 public int rychlost;  
  
 // trieda Bicykel má jeden konštruktor  
 public Bicykel(int zacTempo, int zacRychlost, int zacPrevod) {  
 prevod = zacPrevod;  
 tempo = zacTempo;  
 rychlost = zacRychlost;  
 }  
  
 // trieda Bicykel má štyri metódy  
 public void nastavTempo(int novaHodnota) {  
 tempo = novaHodnota;  
 }  
  
 public void nastavPrevod(int novaHodnota) {  
 prevod = novaHodnota;  
 }  
  
 public void brzdi(int spomalenie) {  
 rychlost -= spomalenie;  
 }  
  
 public void pridaj(int zrychlenie) {  
 rychlost += zrychlenie;  
 }  
}

Deklarácia triedy HorskyBike, ktorá je podtriedou triedyBicykel, môže vyzerať napríklad takto:

public class HorskyBicykel extends Bicykel {  
 // podtrieda HorskyBike pridáva jednu vlastnosť  
 public int vyskaSedadla;  
 // podtrieda HorskyBike má jeden konštruktor  
 public HorskyBicykel(int zacVyska, int zacTempo, int zacRychlost, int  
 zacPrevod) {  
 super(zacTempo, zacRychlost, zacPrevod);  
 vyskaSedadla = zacVyska;  
 }  
 // podtrieda HorskyBike pridáva jednu metódu  
 public void nastavVysku(int novaHodnota) {  
 vyskaSedadla = novaHodnota;  
 }  
}

Trieda HorskyBike zdedila všetky vlastnosti a metódy od triedy Bicykel a pridala vlastnosť vyskaSedadla a metódu na jej nastavenie. S výnimkou konštruktora, je to ako by ste napísali novú triedu HorskyBike obsahujúcu úplné kódy štyroch vlastností a piatich metód. Ako vidíte, nemusíte sa namáhať s písaním. Toto môže byť zvlášť cenné, ak boli metódy triedy Bicykel napísané komplexne a boli dobre odladené.

Čo môžete robiť v podtriede Podtrieda zdedí všetky verejné (public) a chránené (protected) členy svojho rodiča, nezáleží na tom, v akom je podtrieda balíku. Ak je podtrieda v rovnakom balíku ako jej rodič, zdedí tiež package-privatečlenov rodiča. Zdedené členy môžete použiť, prekryť, skryť alebo ich doplniť novými členmi:

* Zdedené vlastnosti môžu byť priamo, tak ako iné normálne polia.
* V podtriede môžete deklarovať vlastnosť s rovnakým menom, aké má jedna z vlastností v nadtriede, teda ju skryjete(neodporúča sa).
* V podtriede môžete zadeklarovať nové vlastnosti, ktoré nie sú v nadtriede.
* Zdedené metódy môžu byť použité tak ako sú.
* V podtriede môžete vytvoriť metódu inštancie, ktorá má rovnaký charakter ako jedna z metód v nadtriede, teda ju prekryjete.
* V podtriede môžete napísať novú statickú metódu, ktorá má rovnaký charakter ako jedna zo statických metód nadtriedy, teda ju skryjete.
* V podtriede môžete zadeklarovať nové metódy, ktoré nie sú v nadtriede.
* Môžete vytvoriť konštruktor podtriedy, ktorý volá konštruktor nadtriedy, buď implicitne alebo pomocou kľúčového slova super

# Polymorfizmus:

https://paz1a.ics.upjs.sk/Material/Polymorfizmus

# Vynimky (Exceptions):

Pri behu programu môžu nastať nepredvídané udalosti, ktoré vyústia k nepredvídanému chovaniu programu a dosť často aj k náhlemu ukončeniu aplikácie. Tieto nepredvídané udalosti sú spôsobené chybami buď v programe urobené napríklad chybou programatora, alebo pri interakcii aplikácie sprostredím keď program dostane chybné vstupné údaje. Napríklad zlý subor, zle internetove pripojenie a pod. Na základe týchto a iných udalostí dôjde k CHYBE. A tieto chyby treba nejako spracovať a vyhodnotiť.

Výnimka (Exception, alebo mimoriadna udalosť) je problém, ktorý sa objaví počas vykonávania programu. Ak dôjde k výnimke, normálny tok programu sa preruší a program / aplikácia sa neočakávane ukončí, čo sa neodporúča, preto výnimkami riešime v kóde. Našou úlohou je zabezpečiť aby sa beh programu neočakavane neukončil. V OOP je všetko objekt. Preto aj výnimočná situacia (chyba, výnimka) je Objekt. Nazývame ju Exception. Jej ulohou je agregovať vsetky potrebné informácie o chybe. (miesto chyby, typ chyby atď. ). Takže keď nastane chyba v programe, program vytvori výnimku (objekt rodičovskej triedy Exception) a do tej výnimky zaznamená všetky potrebné údaje o chybe, tak aby programátor vedel čo spôsobilo chyby programu.

Výnimka môže nastať z mnohých rôznych dôvodov. Nasleduje niekoľko scenárov, v ktorých sa vyskytne výnimka.

*Používateľ zadal neplatné údaje.*

*Súbor, ktorý je potrebné otvoriť, sa nedá nájsť.*

*Uprostred komunikácie sa stratilo sieťové pripojenie alebo sa v JVM vyčerpala pamäť.*

Niektoré z týchto výnimiek sú spôsobené chybou používateľa, inou chybou programátora a inou fyzickými zdrojmi, ktoré nejakým spôsobom zlyhali.

Ako čítat chyby

Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 50

at java.lang.String.charAt(String.java:658)

at sk.sosholic.prvypolrok.vynimky.DomacaUloha.metodaKdeJeChyba(DomacaUloha.java:15)

at sk.sosholic.prvypolrok.vynimky.DomacaUloha.main(DomacaUloha.java:9)

V prvom riadku je napísane, že chyba vznikla vo vlákne "main" a že bola vyhodená chyba

StringIndexOutOfBoundsException: String index out of range: 50. Konkrétne: index Stringu je mimo rozsahu. A konkrétne index 50. Za oznámením o chybe nasleduje výpis zo zásobníku navratových adries (stack trace). Toto umožňuje vysledovať miesto kde nastala chyba.

at java.lang.String.charAt(String.java:658)

at sk.sosholic.prvypolrok.vynimky.DomacaUloha.metodaKdeJeChyba(DomacaUloha.java:15)

at sk.sosholic.prvypolrok.vynimky.DomacaUloha.main(DomacaUloha.java:9)

Tu je napísne že chyba nastala v metóde charAt v triede String, ale príčina chybe je niekde inde.

Dalej sa dozvieme na druhom riadku, že chyba bola spôsobená subore DomacaUloha v metóde metódaKdeJeChyba(). Na riadku 15.

Posledný riadok hovorí, že metóda metódaKdeJeChyba() bola volaná metódou main na riadku 9.

Na výnimky nemusíme čakať alebo zámerne vytvoriť chybný kód. Výnimku môžme vytvoriť pomocou výrazu *throw*. A za výrazom zadáme slovo *new*, keďže je to objekt, a názov dannej výnimky.

throw new RuntimeException();

throw new RuntimeException("toto je nejaka chyba !!!!");

Výniku možme zadať bez parametrov ale je lepčie zadať uprasňujúcu správu o chybe.

Keďže je Výnimka (Exception) objekt tak ma aj metódy, ktoré sú schopne mám vrátiť informacie o chybe, ktorá nastala.

Metódy tried v rodine Exeption:

*getMessage()*; vráti text správy, napísanej v konštruktore

*toString();* vráti názov triedy a ak ma aj správu tak ju pridá za názvom výnimky.

*printStackTrace();* na standartný chybový výstup vypíše text pomocou metódy toString() a celú cestu výnimky kde sa stala.

*printStackTrace(PrintStream)*; metóda umožnuje zapísať chybu aj niekam inam. Napríklad do suboru, na ine miesto v sieti …

Na základe toho máme tri kategórie výnimiek. Musíte im porozumieť, aby ste vedeli, ako funguje spracovanie výnimiek v Jave.

Nekontrolované výnimky (Unchecked Exception)- Nekontrolovaná výnimka je výnimka, ktorá sa vyskytne v čase vykonávania. Nazývajú sa tiež ako výnimky runtime. Patria sem programovacie chyby, ako sú logické chyby alebo nesprávne použitie API. Výnimky za behu programu sa pri kompilácii ignorujú. Tieto chyby z väčšej časti je prográmator vyriešiť úpravou kódu tak aby danná chyba nenastala.

Napríklad, ak ste vo svojom programe deklarovali pole veľkosti 5 a pokúšate sa volať šiesty prvok poľa, vyskytne sa ArrayIndexOutOfBoundsExceptionexception.

*public class Unchecked\_Demo {*

*public static void main(String args[]) {*

*int num[] = {1, 2, 3, 4};*

*System.out.println(num[5]);*

*}*

*}*

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5

at Exceptions.Unchecked\_Demo.main(Unchecked\_Demo.java:8)

Teda pokiaľ metóda vyvolá unchecked výnimku nie je potrebné ju ošetriť. Podtrieda RuntimeException.

Príklady unchecked exceptions:

ArithmeticException

ArrayIndexOutOfBoundsException

ClassCastException

IndexOutOfBoundsException

UnsupportedOperationException

StringIndexOutOfBounds

    Kontrolované výnimky (Checked Exception) - Kontrolovaná výnimka je výnimka, ktorú kompilátor skontroluje (oznámi) v čase kompilácie, tie sa tiež nazývajú ako výnimky pri kompilacii. Tieto výnimky nemožno jednoducho ignorovať, programátor by sa mal o tieto výnimky postarať (spracovať). Sú to chyby, ktoré programátor nie je schopný vyriešiť opravou v kóde ako napríklad NullPointerException alebo OutOfBoundexception.

Pri použitií metód, konštrukorov

Napríklad, ak vo svojom programe používate triedu FileReader na čítanie údajov zo súboru, ak súbor určený v jeho konštruktore neexistuje, dôjde k súboru FileNotFoundException a kompilátor vyzve programátora, aby vyriešil výnimku.

[Live Demo](http://tpcg.io/9u4a5O)

import java.io.File;

import java.io.FileReader;

public class FilenotFound\_Demo {

public static void main(String args[]) {

File file = new File("E://file.txt");

FileReader fr = new FileReader(file);

}

}

C:\>javac FilenotFound\_Demo.java

FilenotFound\_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

FileReader fr = new FileReader(file);

^

1 error

Poznámka - Pretože metódy read () a close () triedy FileReader vyvolávajú IOException, môžete si všimnúť, že kompilátor upozorní na spracovanie IOException spolu s FileNotFoundException. Teda pokiaľ metóda vyvolá checked výnimku je potrebné ju ošetriť.

Príklady checked exceptions:

* [Class­Not­Found­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ClassNotFoundException.html) ✔
* [Illegal­Access­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/IllegalAccessException.html) ✔
* [Instantiation­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/InstantiationException.html) ✔
* [No­Such­Field­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/NoSuchFieldException.html) ✔
* [No­Such­Method­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/NoSuchMethodException.html) ✔
* [File­Not­Found­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileNotFoundException.html) ✔
* [Invalid­Object­Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InvalidObjectException.html) ✔

Chyby (Errors)- Nejedná sa o výnimky, ale o problémy, ktoré vzniknú mimo kontroly používateľa alebo programátora. Chyby sa vo vašom kóde zvyčajne ignorujú, pretože s chybou môžete zriedka robiť čokoľvek. Napríklad, ak dôjde k pretečeniu zásobníka, vyskytne sa chyba. Pri kompilácii sa ignorujú.

Errors sú neobvyklé stavy, ktoré sa vyskytnú v prípade závažných porúch, ktoré nie sú riešené programami Java. Chyby sa generujú na označenie chýb generovaných runtime prostredím. Príklad: JVM nemá dostatok pamäte. Programy sa zvyčajne nemôžu zotaviť z chýb.

Hierarchia výnimiek

Všetky triedy výnimiek sú podtypmi triedy java.lang.Exception. Trieda výnimky je podtrieda triedy Throwable. Okrem triedy výnimiek existuje ďalšia podtrieda s názvom Chyba(Errors), ktorá je odvodená z triedy Throwable.

Diagram

Description automatically generated

Zachytávanie výnimiek

Metóda zachytáva výnimku pomocou kombinácie kľúčových slov **try , catch, finally**. V bloku try je umiestnený kód programu, ktorý môže vyvolať výnimku.

Ak nastane výnimka vykoná sa kód, ktorý je umestnený v časti catch.

K odchytávaní výnimiek je možné pridať aj časť finally. Blok finally sa vykná vždy. Či nastala výnika alebo nie.

Syntax:

**try {**

// Protected code

**} catch (ExceptionName e1) {**

// Catch block

**} finally{**

// kod, ktorý sa vykoná vždy

**}**

public static int cisloZPola(){  
 int[] pole = new int[]{1,3,4};  
 int cislo;  
 try {  
 cislo = pole[6];  
 }catch (Exception ex){  
 System.*out*.println(" zle cislo a tak zadame defaultne cislo 0");  
 cislo = 0 ;  
 }  
 return cislo;  
}

Blok catch treba deklarovať aj typ výnimky sa jedná. Napr. Ak robíme delenie tak na ošetrenie delenim nulou použijeme ArithmeticException:

Int x = 10;

Int y = 0;

try {

x / y;

} catch (ArithmeticException e1) {

System.out.println(“delenie nulou”);

}

Ak blok kódu môže vyvolať viac výnimiek, možeme použiť viac catch blokov:

try {

// Protected code

} catch (ExceptionName e1) {

// Catch block

} catch (ExceptionName e1) {

// Catch block

} catch (ExceptionName e1) {

// Catch block

}

Kľúčové slová Throw / Throws

Ak metóda nespracováva kontrolovanú výnimku (checked exception), musí ju táto metóda deklarovať pomocou kľúčového slova throws. Kľúčové slovo vyvolá sa zobrazí na konci podpisu metódy.

Pomocou kľúčového slova throw môžete vyhodiť výnimku, buď novo vytvorenú, alebo výnimku, ktorú ste práve chytili.

Pokúste sa porozumieť rozdielu medzi throws a throw kľúčových slov, throws sa používa na odloženie spracovania začiarknutej výnimky a throw sa používa na výslovné vyvolanie výnimky.

Nasledujúca metóda vyhlasuje, že vyvoláva RemoteException -

príklad

import java.io.\*;

public class className {

public void deposit(double amount) throws RemoteException {

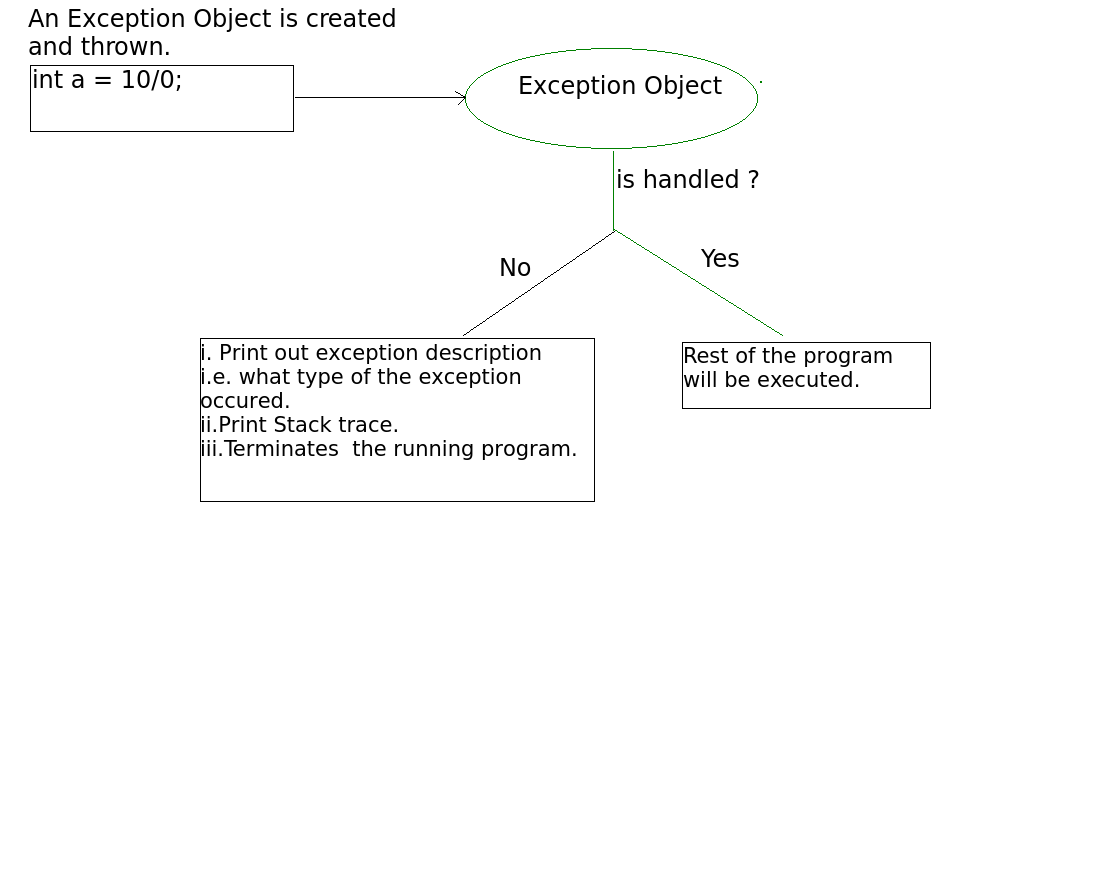
// Method implementation

throw new RemoteException();

}

// Remainder of class definition

}



# IO Java

Na prácu so vstupami a výstupami Java poskytuje triedy a metódy, ktoré sú uložené v balíčku java.io. Práca s IO je založená na mechanizmu vstupných a výstupných prúdov(streams). Stream je postupnosť

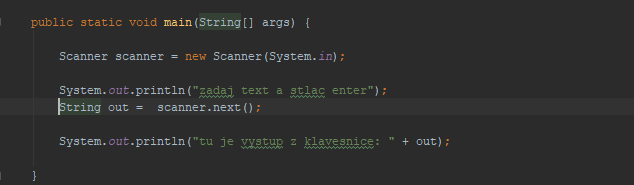
dát. V Jave stream pozostáva y bajtov. Doteraz sme sa stretli s tromi prúdmi. Tieto streamy sú prepojené s konzolou.

**1) System.out:** standard output stream

**2) System.in:** standard input stream

**3) System.err:** standard error stream

Priklad System.out a System.in:

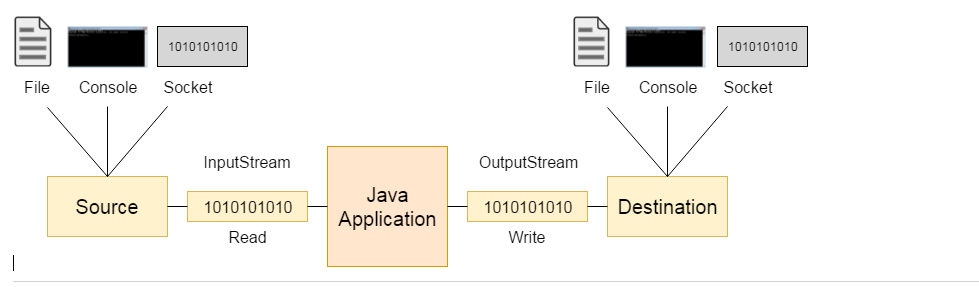


System.err vidime vždy ak sa nám niečo nepodarí. (kto ma viac System.err vyhrá 😊)

InputStream vs OutputStream

Pokiaľ chceme IO ( Input/Output, vstup/výstup) použit inde ako v interakciou s konzolou treba použit triedy Inputstream a OutputStream.   
Pre vysvetlenie pojmov: InputStream sa používa pri čítaní dát do Java aplikácie a OutputStream opačne. Treba názvoslovie brať z pohľadu Java aplikacie. Teda aplikácia číta dáta, ide do aplikácie ( InputStream ).

Ako zdroj/cieľ môžeme brať súbor, zariadenie (sieťová karta) atď.



ByteStreams vs character stream.

Java IO prúdy sa rozdelujú podľa toho aký je zdroj streamu. A to: bajtový alebo znakový zdroj. Streamy zaloťžené na bajtovom zdroji majú v názve tried slovo „stream“ InputStream alebo OutputStream. Tieto streamy zapisujú alebo čítajú neupravené bajty.

Streamy, ktoré pracujú so znakmi tak nazvz tied obsahujú Reader alebo Writer. Streamy ktorých zakladom sú znaky môžme tie znaky priamo čítať-zapisovať. Napr znaky v UNICODE tabuľke.

***Triedy na čítanie bajtových dát. Teda nie text.***

**InputStream**: abstaktná metóda, ktorá definuje základné metódy na čítanie po znakoch

**FileInputStream**: čítanie zo súboru, parametrom konštruktoru je String s menom súboru na objekt typu File

**BufferedInputStream**: vytvára buffer na čítanie, tým sa zlepši efektivita

**LineNumberInputStream**: pridáva metódu na číslovanie čítaných riadkov

***Triedy na čítanie textových dát.***

**Reader**: abstaktná metóda, ktorá definuje základné metódy na čítanie po znakoch

**FileReader**: čítanie zo súboru, parametrom konštruktoru je String s menom súboru na objekt typu File

**BufferedReader:** vytvára buffer na čítanie, tým sa zlepši efektivita

**LineNumberReader:** pridáva metódu na číslovanie čítaných riadkov

## Čítanie súborov pomocou Java IO

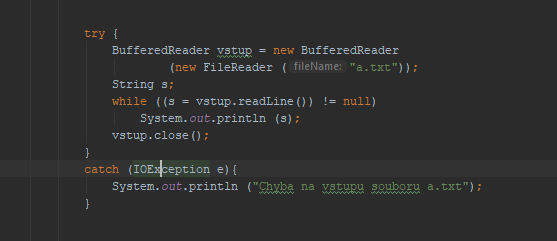
Pokiaľ chceme čítať súbor možme použit **[FileInputStream](http://tutorials.jenkov.com/java-io/fileinputstream.html)** alebo [**FileReader**](http://tutorials.jenkov.com/java-io/filereader.html) podľa toho či chceme čítať binárne alebo textové data. Tieto dve triedy nám umožňujú čítať jeden bajt alebo znak od začiatku suboru po koniec, alebo čítať zdrojové dáta po väčších častiach tak že sa jednotlivé znaky ukladajú do poľa typu char alebo byte. (priklad treba dat)

try (FileReader nacitajZnak = new FileReader("a.txt")) {  
 // a je mozne citat jednotlive znaky  
 int znak = nacitajZnak.read();  
 System.*out*.println((char)znak);  
}catch (IOException ex){  
 System.*out*.println(ex);  
}

Najčastejšie používaným vstupom je textový súbor. Ak chceme prečítať tento súbor po riadkoch, je nutné si pre to vytvoriť vstupný prúd. Pre čítanie po znakoch zo súboru A.TXT si môžeme pripraviť inštanciu triedy FileReader : *FileReader ctiZnak = new FileReader ("a.txt");* . A potom môžme čítať jednotlivé znaky z tohto súboru pomocou metódy read().

*znak = ctiZnak.read ( );*

Väčšinou chceme čítať súbor po riadkoch a pre toto čítanie musíme inštanciu triedy FileReader ešte zabaliť do triedy *BufferedReader.*



A potom môžme použiť metódu readLine().

*String s = radek.readLine();*

Obidve triedy poskytujú metódu close() na zatvorenie súboru. Je dôležité aby sa metóda close() použila vždy po skončení práce so súborom. Dôvod je ten, že ak JVM začne pracovať so súborom tak OS uzamkne súbor pre použitie pre ostatné aplikácie. A je potrebné, aby sa súbor uvoľnil po skončení práce s ním.

## Trieda File

Trieda File slúži na manipuláciu so súbormi a adresármi. Inštanciou triedy File môže byt adresár alebo súbor. Pri vytváraní inštancie nie je nutné aby súbor/adresár existoval. Trieda File slúži aj na vytvorenie súboru. Na popis cesty sa používa /.

|  |  |
| --- | --- |
| **príklad použitia** | **význam** |
| File mojSubor = new File ("a.txt") | inštancia mojSubor nastavená na a.txt v aktualnom adresári |
| File mojaPosta = new File("C:"+File.separator+ "posta") | inštancia mojaPosta nastavena na adresár posta na disku C |
| File mojList = new File ("C:/posta/list1.txt") | inštancia mojaPosta nastavena na súbor list1txt v adresári pošta na disku C |
| File mujList = new File ("C:\\posta","list1.txt") | inštancia mojList nastavená na súbor list1.txt a adresari posta |
| File mojList = new File (mojaPosta,"list1.txt") | instance mojList nastavená na súbor list1.txt v adresári posta na disku C, použili jsme instanciu mojaPošta vytvorenú vyššie |

Trieda File obsahuje metódy isFile() a is Directory(), ktoré zistia či danná inštancia triedy je súbor alebo adresár. Ak chceme zisti, či adresár alebo súbor existuje použijeme metódu exists(). Na vytvorenie adresára sa použie metóda mkdir() a na vytvorenie súboru sa požije metóda createNewFile(). Ďalšie metódy: delete(), renameTo(), list() – výpis obsahu adresára.

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

File myObj = new File("filename.txt");

if (myObj.exists()) {

System.out.println("File name: " + myObj.getName());

System.out.println("Absolute path: " + myObj.getAbsolutePath());

System.out.println("Writeable: " + myObj.canWrite());

System.out.println("Readable " + myObj.canRead());

System.out.println("File size in bytes " + myObj.length());

} else {

System.out.println("The file does not exist.");

}

}

}

## Writing File via Java IO

**OutputStream:** abstaktná metóda, ktorá definuje základné metódy na zápis po znakoch

**FileOutputStream:** zápis do súboru, parametrom konštruktoru je String s menom súboru na objekt typu File

**BufferedOutputStream:** vytvára buffer na efektívnejší zápis

**Writer:** abstaktná metóda, ktorá definuje základné metódy na zápis po znakoch

**FileWriter:** zápis do súboru, parametrom konštruktoru je String s menom súboru na objekt typu File

**BufferedWriter:** vytvára buffer na efektívnejší zápis

Ak je treba zapísať data do súboru môžme využit triedy FileOutputStream alebo FileWriter, podľa toho či potrebujeme zapísať binárne data alebo znakové data. Tak ako pri čítaní súboru si môžme vybrať zápis po znakoch alebo po väčších sekvenciach. Obyčajne po riadkoch.Dáta sú zapisované sekvenčne tak ako sú posielané streamom do súboru. Ak súbor na disku neexistuje, bude vytvorený nový, ale ak súbor existuje tak súbor bude prepísaný. Ak chceme zapisovať už do existujúceho súboru, (napríklad zápis do logu) treba použiť konštruktor triedy FileWriter s dvomi parametrami. Prvý parameter je súbor a druhý je parameter, logická hodnota, ktorá určuje či budeme zapisovať na koniec súboru alebo sa súbor prepíše.

void test() throws IOException {

// java zapíše na koniec súboru a nebude prepisovať súbor  
 FileWriter zapis = new FileWriter("a.txt", true);   
}

BufferedWriter writer = new BufferedWriter(new FileWriter("test.txt"));  
writer.write("vlozeny text do suboru text.txt");  
writer.close();

Poznamky pod čiarov:

Absolutna cesta suboru alebo pomocou

URL fileUrl = getClass().getResource("/subor/subor.txt");

<http://tutorials.jenkov.com/java-io/index.html>

<https://java.vse.cz/pdf/java-vstupy_vystupy.pdf>

https://javatutorialhq.com/java/io/bufferedreader-class-example/read-char-cbuf-int-off-int-len-method-example/

# Java Networking

URI – Uniform Resource Identifier.

URL - Uniform Resource Locator. Reprezentuje ukazateľa na konkrétny zdroj v sieti. Jeho format je *služba://uživatel:heslo@host:port/cesta#reference .*

URL možeme zadať absolútnou cestou alebo relatívnou.

URL – reprezentuje URL požadového sieťového zdroja. Obsahuje metódy na zistenie jednotlivých častí URL: getFile(), getHost(), getPort(), getProtocol() , getRef().

Metóda getContent() na ziskanie obsahu zdroja vo forme Objektu poskytovaného príslušným obladačom obsahu.

URLConnection

HttpURLConnection

Low Level Network Programming. Socket, ServerSocket, DatagramSocket.

URI, URL, URLConnection

URI je identifier, ktorý nemusí poskytnúť dosť informácií na ziskanie informácii zo droja.

URL je identifier, ktorý zahŕňa aj informácie ako sprístupniť zdroj. Protokol ako získať informaciu. (http, file, ftp)

možme konvertovať medzi URI a URL pomocou metod.

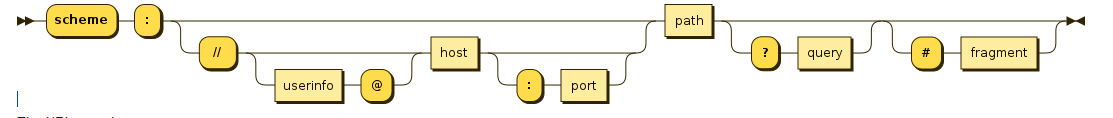
URI uri = new URI("file://username:password@myserver.com:5000/catalouge/phones?os=android#samsung");  
  
URL url = new URL("http://username:password@myserver.com:5000/catalouge/phones?os=android#samsung");  
  
URL uriToURL = uri.toURL();  
  
URI urlToURI = url.toURI();

URL môžme vždy skonvertovať na URI. Ale pri konvertovaní na URI na URL je možné, že metóda toURL(), môže vykázať chybu.

Schema je čast URI alebo URL pred dvojbodkou.

URI obsahuje deväť častí:

<https://en.wikipedia.org/wiki/Uniform_Resource_Identifier>



file://, ftp://, http://, https://, mailto:, news:, telnet://

URLConnection a HttpURLConnection

Tieto triedy sa využívajú ku komunikácii so serverom pomocou URL.

URLConnection je rodičovská trieda pre všetky triedy, ktoré sa používajú ku komunikácii pomocou URL.

HttpURLConnection je podtrieda, ktorá sa špecifikuje na komunikaciu cey HTTP. Obidve triedy su abstraknté a teda nemôžeme vytvoriť priamo novú inštanciu pomocou ***new.*** Namiesto toho sa objekt vytvára otvorením spojenia s URL objektom.

URLConnection urlConnection = url.openConnection();

Obyčajne postup vytvorenia spojenia klient/server:

1. Vytvorí sa URL objekt
2. Získa sa objekt URLConnection pomocou URL objektu
3. Konfigurácia URLConnection
4. Načítanie hlavičky
5. Ziskanie inputSteam na čitanie dát
6. Získanie outputStream na zápis dát ( ak je potreba na zápis)
7. Zastvorenie spojenia

Body 3-6 sú dobrovoľné a vyberáme si aký ukon potrebujeme.

1.Vytvorí sa URL objekt

*URL url = new URL("http://www.google.com");*

2.Získa sa objekt URLConnection pomocou URL objektu

*URLConnection urlCon = url.openConnection();*

Alebo ak chceme použiť HttpURLConnection:

*HttpURLConnection httpCon = (HttpURLConnection) url.openConnection();*

Treba si pamätať že metóda openConnection() nevytvára spojenie s URL ale len vracia inštanciu objektu URLConnection.

Na vytvorenie stpojenia treba použiť metódu connect().

3.Konfigurácia URLConnection

Pred spojením s URL cieľom je možné nastaviť rôzne parametre spojenia. Upraviť metadáta, tj parametre v hlavičke. Napríklad:

* **setConnectTimeout**(int timeout):
* **setReadTimeout**(int timeout):

4. Čítanie hlavičky

Keď je nastavné spojenie, server spracuje URL a pošle naspäť metadáta a aktuálny obsah. Metadáta znamenajú kolekciu kľúč=hodnota.

5.Ziskanie inputSteam na čitanie dát

Aktuálny obsah sa potom načítava pomocou ImputStream-u

*InputStream inputStream = urlCon.getInputStream();*

*byte[] data = new byte[1024];*

*inputStream.read(data);*

*BufferedReader reader = new BufferedReader(new InputStreamReader(urlCon.getInputStream()));*

*String line = reader.readLine(); // reads a line*

6. Vytvorenie outputStream-u na zápis dát

*OutputStream outputStream = urlCon.getOutputStream();*

*byte[] data = new byte[1024];*

*outputStream.write(data);*

7.Zatvorenie spojenia

Na zatvorenie spojenia treba použit close() metódu.

# JDBC

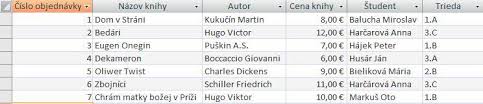
Dáta sú informácie. napríklad vaše krstné meno. Databáza je organizovaný zber dát. V skutočnom svete je kartotéka typom databázy. Má priečinky na súbory, z ktorých každý obsahuje kúsky papiera. Priečinky súborov sú usporiadané nejakým spôsobom, často abecedne. Každý kúsok papiera je ako kúsok údajov. Podobne sú priečinky vo vašom počítači podobné ako databáza. Priečinky poskytujú organizáciu a každý súbor je údajom.

Relačná databáza je databáza usporiadaná do tabuliek, ktoré pozostávajú z riadkov a stĺpcov. Existujú dva hlavné spôsoby prístupu k relačnej databáze z jazyka Java:

■ Jazyk Java Database Connectivity Language (JDBC): Pristupuje k údajom ako riadky a stĺpce.

■ Java Persistence API (JPA): Pristupuje k údajom prostredníctvom objektov Java pomocou konceptu nazývaného objektovo-relačné mapovanie (ORM). Ide o to, že nemusíte napísať toľko kódu a svoje údaje získate v objektoch Java.

Prístup do relačnej databázy sa uskutočňuje pomocou jazyka SQL. Structure Query Language. Ako už bolo spomenuté data v relačnej database sú usporiadané do tabuliek s určitým počtom riadkov a stĺpcov.



Kde vždy jeden stĺpec obashuje unikatne čislo **id**. (čislo riadku takzvaný primárny kľúč)

V našom priklade na obrázku tomu zodpovedá stĺpec ***Číslo objednávky.***

Základné príkazy SQL na komunikáciu s databázou sú:

INSERT: INSERT INTO hnihy VALUES (8, '1984', ‘George Orwell’, ’10.00 ’, ‘Janko Hrasko’, ‘3A’);

SELECT: SELECT \* FROM KNIHY WHERE ‘cislo objednavky’ = 3;

UPDATE:

DELETE: DELETE FROM KNIHY WHERE ‘Nazov knihy’ = '1984';

Dolezite interfaces

What do these four interfaces do? On a very high level, we have the following:

Balicek java.sql.\*;

**Driver**: Vie ako získať konektivitu k databze

**Connection:** Vie ako komunikovať s databázou

**Statement:** Vie ako spustiť SQL dotaz

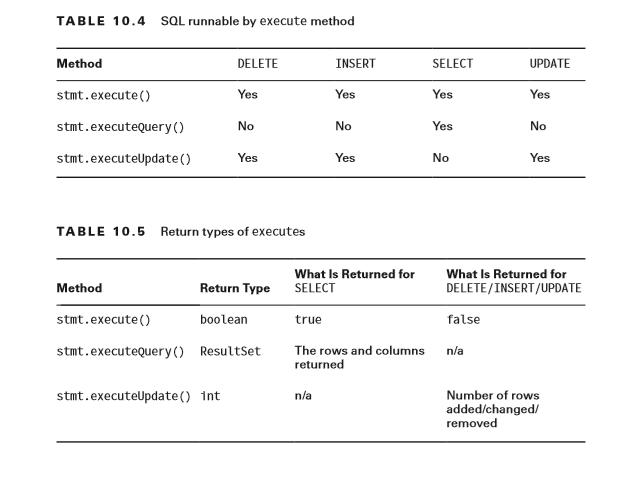
**ResultSet:** Vie spracovať odpoved z databáze

Rozhrania (Interfaces) nám len definujú čo môžme robit s databátami. Ale skutočná implelentácia, teda ako sa napríklad pripojíme k databáze, je v baličkoch prislúcahucej k dannej databaze. Napriklad ak použivame oracle dababazu musime si improtovať ovladače k Oracle databaze. A tam Je skutočná Implementácia Rozhrani.

Priklad pripojena sa k database:

public static void main(String[] args) throws MalformedURLException, SQLException, ClassNotFoundException {  
 String DB\_URL = "jdbc:h2:tcp://localhost/C:/Users/lubossukup/JAVA/skola/DB/db.db";  
 String USER = "sa";  
 String PASS = "sa";  
 Statement stmt;  
  
 Connection conn = DriverManager.*getConnection*(DB\_URL, USER, PASS);  
  
  
}

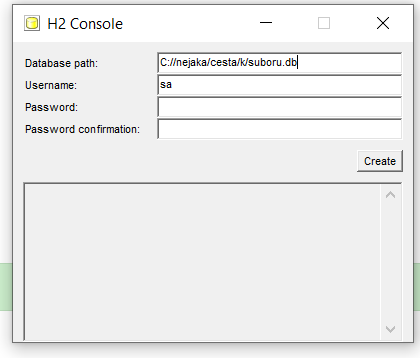
Method DELETE INSERT SELECT UPDATE



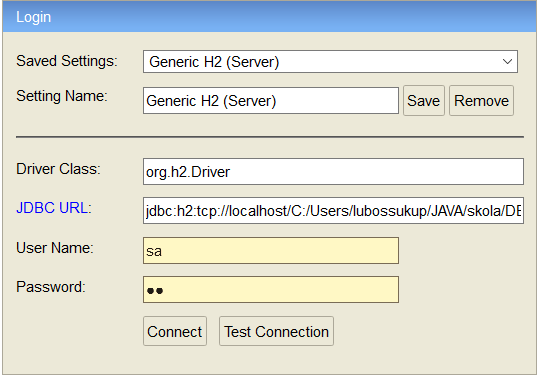
Instalacia H2  
<http://www.h2database.com/html/download.html>

Po inštalácii spustime h2 Console. Na pravej strane v lište windows, sa ukáže ikona H2. Pravým kliknutim myši si zobrazime menu a vyberieme create new database.

Zadame cestu k suboru db, ktory H2 vytvorí. Zadame meno a heslo.



Po vytvorení databazy, spustíme si H2 konzolu. . Na pravej strane v lište windows ikona H2. Pravým kliknutim myši si zobrazime menu a vyberieme h2 Console.



JDBC URL bude: jdbc:h2:tcp://localhost/C://nejaka/cesta/k/suboru.db

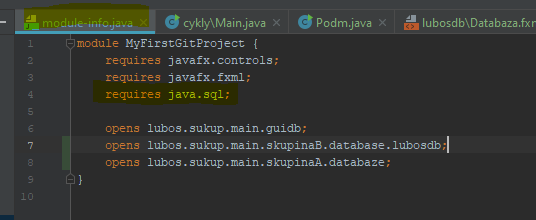
Treba si stiahnut H2 driver: <https://repo1.maven.org/maven2/com/h2database/h2/1.4.200/h2-1.4.200.jar>

A importovat do projektu. (tak ako javafx)

H2 databáza tutorial: <https://www.tutorialspoint.com/h2_database/h2_database_insert.htm>

V prípade ak JDBC je súčasťou javafx a balíčky pre GUI sú importované pomocou module-info.java tak treba do tohto súboru je nutné pridať riadok:

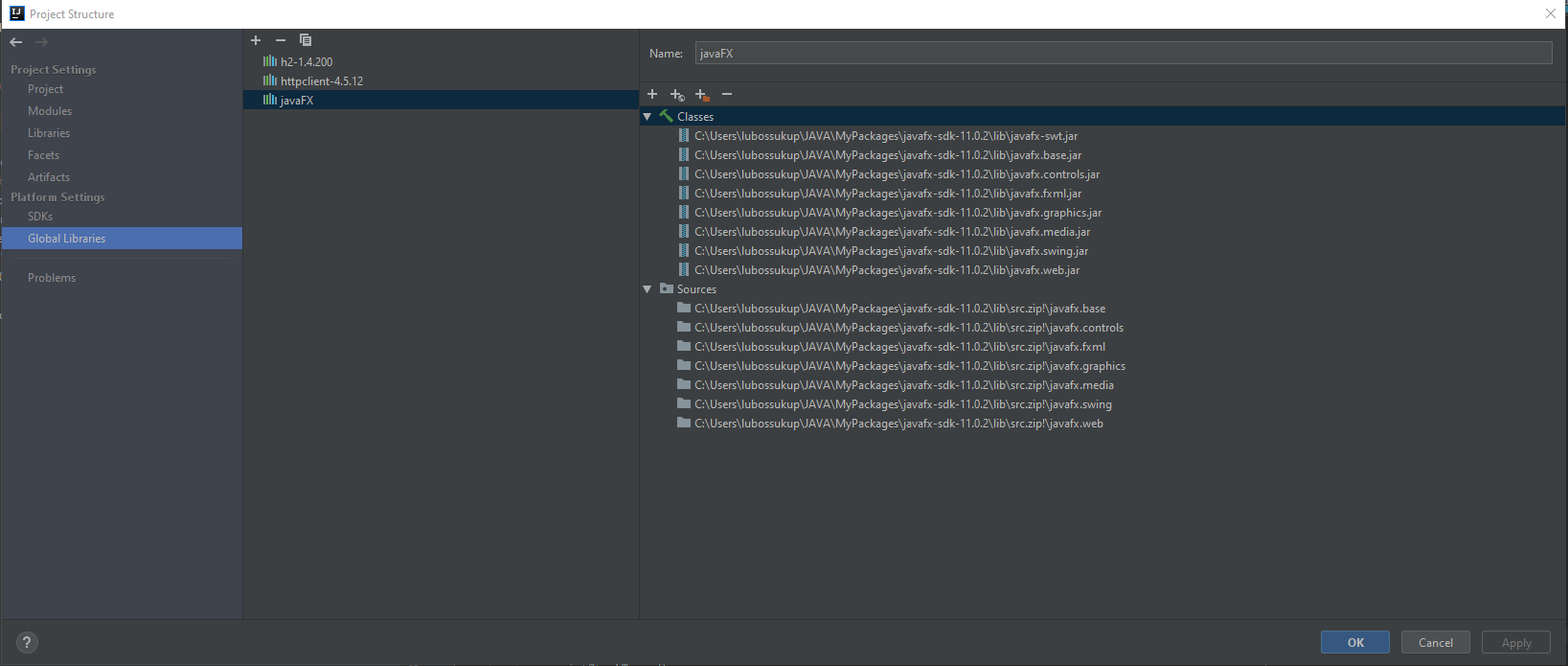
*requires java.sql;*



# Import JavaFx pri java 9 a vyššie

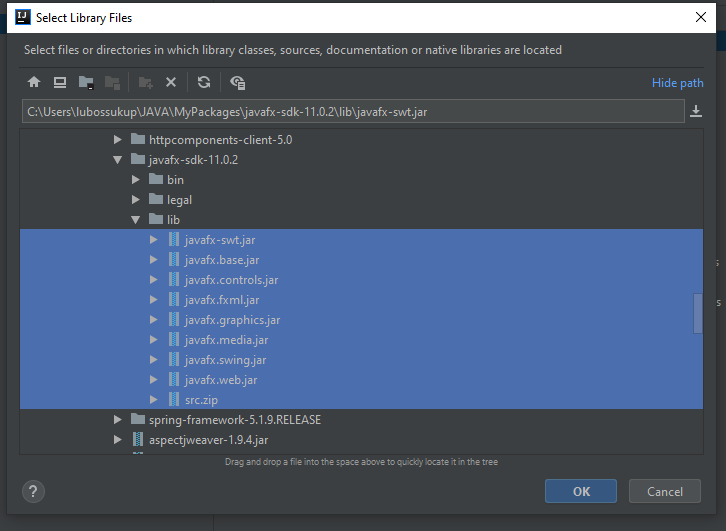
File -> Project structure -> Global Libraries

Treba pridať javafx tak aby importované jar subory javafx bolo ako na obrazku.



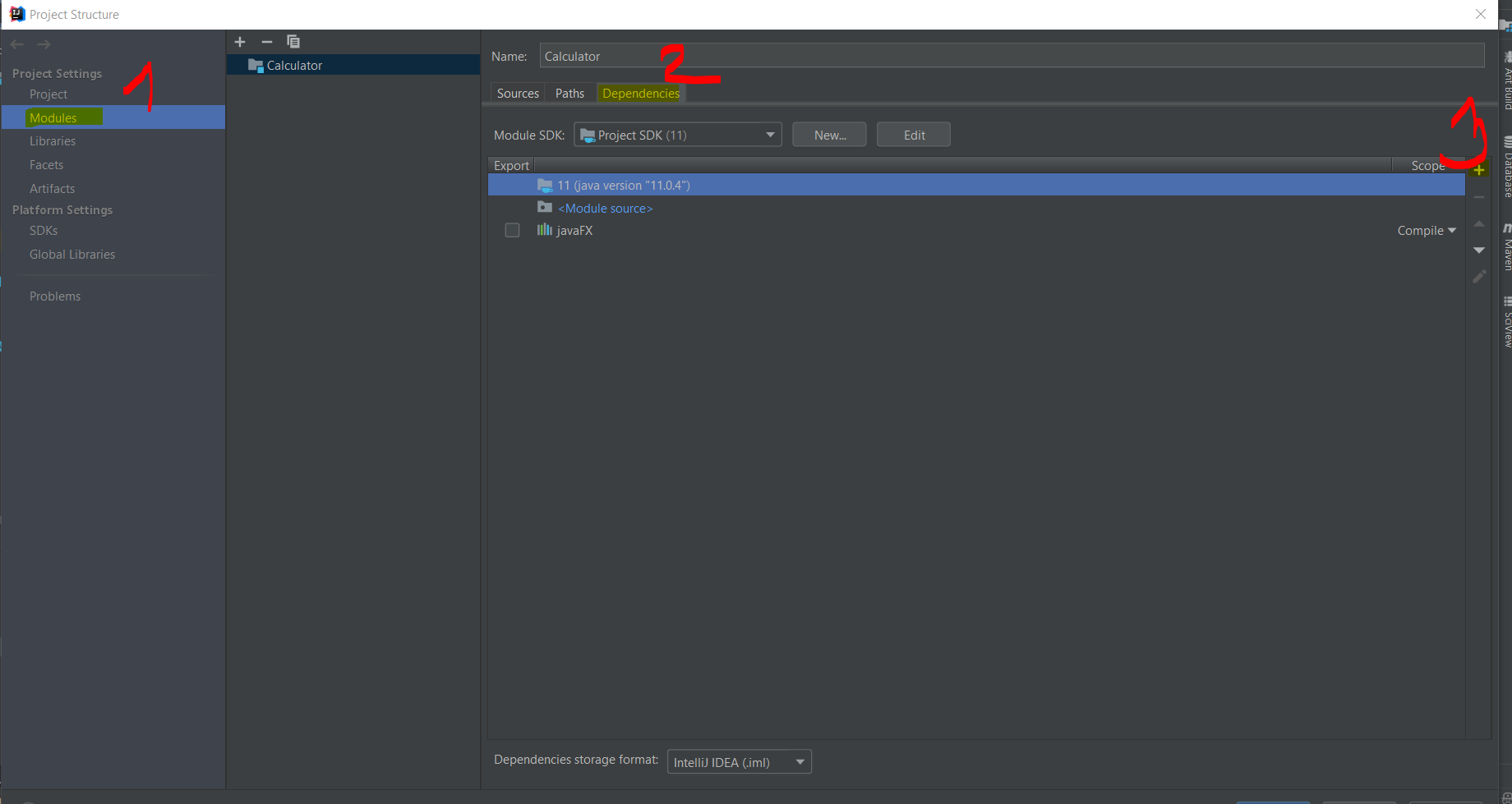
Tlacidlo + pridanie.

V okne si najdeme lib adresar v javafx a pridame si jednotlive jar subory



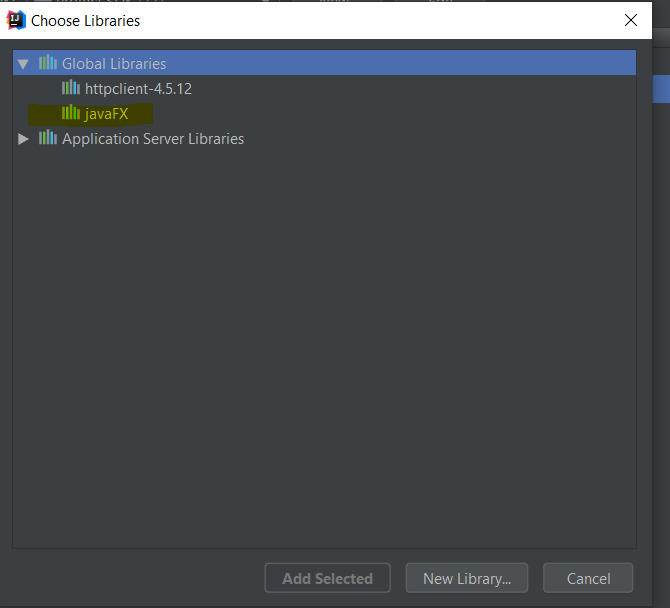
Dame ok, apply.

Potom si skontrolujeme či importovaná javafx sa nacházda v module dependencies:



Ak nie je tak si javafx pridáme manuálne.

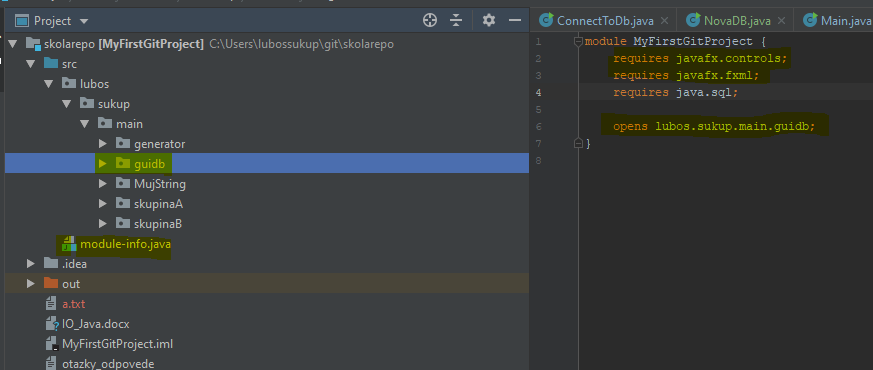
Ked kliknete na + vyberte z menu Library



Z menu vyberte vasu javafx kniznicu. Ja ju mam nazvanu javafx. U vas sa bude volat inac. A Add selected.

Potom date apply a je to. Javafx naimportovana.

Za dalšie je treba si vytvoriť súbor module-info.java.



Treba vložiť do suboru: requires javafx.controls;

requires javafx.fxml;

opens „cestu k baličku kde je umiestnená trieda co využiva fx kniznicu“

Na obrazku to je“ lubos.sukup.main.guidb;

# Maven

Je to nástroj ktorý nám automatizuje vytváranie java-projektov. Pri použití mavenu ako nástroja na vytváranie java programu sa nám stará o:

1. Vytvorí nám adresárovú štruktúru projektu pre:

* zdrojové kódy programu
* zdrojové kódy testu
* adresár manifest (v tomto adresári je umiestnený subor Manifest.mf potrebný pri kompilácci a nasledom spusteni aplikácie v JVM)
* adresár resource (slúži nám na uloženie dodatočných zdrojov pre aplikácie a napr. obrázky potrebné k našej aplikácií, dokumenty atp.)

2. Maven sa stará o správu externých balíčkov ktoré by mohli byť použité v danej aplikácií.

3. Po pridaní pluginu sa Maven stará o kompiláciu programu a zabalenie do spustiteľného balikča. Naznámejšie balíčky sú JAR, WAR, EAR.

Na konfigurovanie projektu Maven sa používa súbor s názvom pom.xml, každý Maven projekt musí obsahovať tento súbor. Dôležité konfiguračné parametre pom súboru sú : groupid, artifact, version.

<project>

<artifactid> </artifactid>

<groupid> </groupid>

<version> </version>

<dependencies>

<dependency>

<artifactid> </artifactid>

<groupid> </groupid>

<version> </version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

</plugin>

</plugins>

</build>

</project>

Ukázka štrukovania pom.xml

Samotné vytvorenie balíčku našej aplikácie v Mavene je definované v tkz. Life-cycle, je to súbor krokov ktoré sa vykonávajú pri vytváraní balíčku našej aplikácie a následnej distribúcie do repozitára

Life-cycle obsahuje :

* clean
* validate
* compile
* test
* package
* verify
* install
* site
* deploy

Clean – Zmaže adresár v ktorom sú vytvorené skompilované triedy a balíčky, adresár ktorý sa maže sa volá target

Validate – Kontroluje správnosť

Compile – Zkompliuje zdrojové kódy aplikácie

Test – Vykoná unit testy ak sú prítomné v aplikácií

Package – Vytvorí daný balíček aplikácie jar, war atp.

Verify – kontorluje vysledok integračnýcch testov

Install – nahrá skompilovanú applikaciu vo forme balíčka na lokálny repozitár

Deploy – ak je nastavený vzdialený repozitár, maven naň nahrá balíček.

Site – maven vytvorí stránku pre daný projekt.

Lokálny repozitár je miesto v počítači kde sú uložené všetky balíčky, ktoré sme doposial použili vo projktoch, na ktoré sme použili Maven. Postup vyhľadávania balíčka Maven alikácie, ktorý je potrebný pri vývoji daného projektu. Maven najprv vyhľadá balíček v lokálnom repozitrari ak tam nie je pokusí sa ho najisť s stiahnuť vo vzdialenom repozitari. Ten sa nachadza na url adrese *mvnrepository.com*

Ak je baliček nájdený vo vzdialenom repozitari, maven si ho stiahne do lokálne ho repozitára. Lokálny repozitár sa nachádza v adresari s názvom *m2*.

O tom aké baličky sú potrebné v dannom porjekte je vedený záznam v pom súbore s tagami *dependencies* a *dependency*.

# Git a Github

Git je verziovací systém, ktorý je open source DVCS (distribuovaný systém riadenia verzií). Je možné ho ovládať cez príkazový riadok alebo si nainštalovať nejaké GUI pre Git. Git v podstate ukladá celú históriu projektu na všetkých zariadeniach, kde je repozitár stiahnutý. Git sleduje zmeny v dannom projekte a tieto jednotlove zmeny je schopný uloziť (commit). Ak je to potrebné je možné sa k týmto znenám vratiť.

Prikazy:

**git init** – tento príkaz vytvorí adresár .git kde bude Git ukladať všetky informácie o verzovanom repozitari. Je to prvý príkaz, ktorý je nutné spustiť pri zakladaní nového projektu.

**git config --global user.name "Vase Jmeno" – Tímto sa nastavia informácie o sebe, ktoré budú použité pri každom commite.**

**git add nazevSouboru.pripona --** vymeníte za názov súboru, ktorý chcete verzovať. Tento súbor sa tak pridá do tzv. „staging area“. Zmeny v tomto súbore bude následne možné commitovať.

**git add . -- P**okiaľ ste v priečinku, ktorý chcete pridať celú a nepridávať jednotlivo súbory pomocou predchádzajúceho príkazu, môžete využiť bodku. Vďaka nej pridáte všetky súbory v aktuálnom adresári.

**git commit -m "Popis commitu" --** Uloží zmeny zo staging area, do popisu commitu sa dáva krátka správa ako napríklad fix určitého bugu alebo nová feature.

Git je umiestnený na vašom počitači a zaznamenáva zmeny len lokalne. Ale ak na projekte pracuje viac členov tak je tak isto potrebné aby zmeny na projekte boli synchornizované. K tomuto služia vzdialene repozitáre. Jedným z nich je Github. Na Githube je uloženy projekt v podobe branche (vetva). Kazdý projekt ma hlavnú vetvu ktorá uchováva projekt v produkčnej verzii. Z hlavnej branche je mozné vytvoriť vedľajšie branches (vetvy). My budeme pracovat len s jednou hlavnou branche. Prepojenie lokalneho repozitára so vzdialeným repozitárom sa deje pomocou týchto príkazov:

git branch -M main -- vytvorí sa vetva s názvom main. Zatiaľ len lokálne.

git remote add origin <https://github.com/lubbeno/test.git> -- tento prikaz prepojí lokalny repozitár so vzdialeným repozitárom. Tento vzdialený repozitár je najprv nutne vytvrorit v konte na github-e. (zatiaľ na githube nič nie je uložené). Po tomto príkaze je potrebne zadat prihlasovacie udaje ku github kontu kde bol repozitár vytorený.

git push -u origin main – Nahrá main branche na vzdialený repozitár.

# Trieda Scanner

Objekt Scanner môže analyzovať vstup používateľa zadaný na konzole alebo zo súboru. Zadané hodnoty analyzované triedou Scanner sú nami známe primitívne dátove typy a String. Skener rozdeli svoj vstup

do samostatných tokenov (ktoré sú zvyčajne oddelené medzerou alebo novym riadkom- enter) a potom ich vráti po jednom. Skener poskytuje metódy na konverziu tokenov na hodnoty rôznych typov. java.util.Scanner je časťou Java API, a preto je súčasťou každej java inštalacie.

Príklad použitia triedy Scanner.

Tento kód načíta dve čísla z konzoly a vytlačí ich súčet:

*Scanner in = new Scanner(System.in);*

*int i = in.nextInt(); i*

*nt j = in.nextInt();*

*System.out.println(i+j);*

Konštruktory:

Scanner(InputStream source)

Scanner(File source)

Scanner(String source)

Vytvorí Scanner objekt a skenuje hodnoty zo zdroja, ktorý je špecifikovaný ako parameter v konštruktore.

Scanner console = new Scanner(System.in);

Vybraté metódy:

String nextLine() Prečíta a vráti ďalší riadok vstupu ako reťazec. (Konkrétne prečíta a vráti všetky zostávajúce znaky v riadku ako reťazec znakov a potom sa presunie na ďalší riadok.)

String next() Prečíta a vráti reťazec znakov kým nenarazí na špeciálny znak(medzera, tabulator, novy riadok).

double nextDouble() Prečíta a vráti double hodnotu. Ak sa ďalší znak nedá preložiť na double, hádže InputMismatchException.

int nextInt() Prečíta a vráti int hodnotu. Ak sa ďalší znak nedá preložiť na int, hádže InputMismatchException.

boolean hasNextLine() Vráti hodnotu true, ak existuje ďalší riadok vstupu. Nečíta za riadkom.

boolean hasNext() Ak existuje ďalší token, vráti hodnotu true. Nečíta za token. boolovská hodnota

hasNextDouble() Vráti hodnotu true, ak existuje ďalší token, ktorý možno preložiť ako double. Nečíta za token.

boolean hasNextInt() Vráti hodnotu true, ak existuje ďalší token, ktorý možno preložiť ako int. Nečíta za token.

# Poznámky

Základné príkazy MS Win

cd = Change directory

cd .. = vrátime sa o stupeň vyšši

dir = Zobrazí nám adresár v ktorom sa aktuálne nachádzame

# Úlohy:

Zaklady:

Napíšte metódu hasEqualSum s 3 parametrami typu int.

Metóda by mala vrátiť boolean a musí sa vrátiť true, ak sa súčet prvého a druhého parametra rovná tretiemu parametru. V opačnom prípade vráťte hodnotu false.

## 1 Mačka:

The cats spend most of the day playing. In particular, they play if the temperature is between 25 and 35 (inclusive). Unless it is summer, then the upper limit is 45 (inclusive) instead of 35.

Write a method isCatPlaying that has 2 parameters. Method needs to return true if the cat is playing, otherwise return false

1st parameter should be of type boolean and be named summer it represents if it is summer.

2nd parameter represents the temperature and is of type int with the name temperature.

EXAMPLES OF INPUT/OUTPUT:

\* isCatPlaying(true, 10); should return false since temperature is not in range 25 - 45

\* isCatPlaying(false, 36); should return false since temperature is not in range 25 - 35 (summer parameter is false)

\* isCatPlaying(false, 35); should return true since temperature is in range 25 - 35

NOTES

\* The isCatPlaying method needs to be defined as public static like we have been do

## 2. Prevodník Libry na kilogramy

1 libra je 0.45359237kg.

vytor prmenn na ukadanie Libier

a potom vytvor premenu na ulozenie vysledku kilogramu

## 3. Priestupný rok

Každý rok deliteľný štyrmi je priestupný. Každý rok deliteľný 100 nie je priestupný. Každý rok deliteľný 400 je priestupný. Teda, 1700, 1800, 1900, 2100 a 2200 nie sú priestupné roky. Avšak 1600, 2000 a 2400 sú priestupné.

Urob metodu ktora berie parameter od 1 – 9999

a vypočitaj priestupny rok.

Update. Dva parametere ( mesiac , rok )

vystup poced dni v mesiaci

4. vytvor metódu ktorá má jeden parameter int vypíše či je číslo párne alebo nepárne.

5. napíš metódu ktorá vypíše študentovu známku pri zadaní počtu bodov.

90 – 100 A

80 – 89 B

70 - 79 C

60 – 69 D

50 – 59 E

menej ako 50 F

5. uloha vytvor program kamen nôžky papier.

6 . Palindrom String

class ChkPalindrome { public static void main(String args[]) { String str, rev = ""; Scanner sc = new Scanner(System.in); System.out.println("Enter a string:"); str = sc.nextLine(); int length = str.length(); for ( int i = length - 1; i >= 0; i-- ) rev = rev + str.charAt(i); if (str.equals(rev)) System.out.println(str+" is a palindrome"); else System.out.println(str+" is not a palindrome"); } }

7. Práca so string: Urob si dve premenné string: meno a priezvisko.

a. zmeň na velké pismená a vypíš.

b. vytvor novú premennú spojením mena a priezviska.

c. vytvor program, ktorý napíše string odzadu.

d. vypíš každé druhé písmeno z celého mena.

e. vymeň každé tretie písmeno za X v priezvisku.

8. Arrays.

a. vytvor si pole ktoré bude uchovávať reťazce, veľkosti 5.

b. vytvor si pole ktoré bude uchovávať double, veľkosti 4.

c. vypíš všetky elementy do konzole Pomocou for a foreach, a pomocou Arrays.toString

d. metodou zisti velkost poli

e. v poli bodu a. zmen element na indexe 3

f. v poli bodu b. zmen element na indexe 4

g. Write a Java program to sum values of an array and average

https://www.w3resource.com/java-exercises/array/index.php

9. arrayList

<https://www.w3resource.com/java-exercises/collection/index.php>

Triedy:   
uloha 1:

// Vytvorenie novej triedy pre bankový účet

        // Vytvorte polia pre číslo účtu, zostatok, meno zákazníka, e-mail a telefónne číslo.

        //

        // Vytvorte getery a setters pre každé pole

        // Vytvorte ďalšie dve metódy

        // 1. Umožniť zákazníkovi vkladať prostriedky (tým by sa malo zvýšiť pole rovnováhy).

        // 2. Umožniť zákazníkovi vyberať prostriedky. Toto by sa malo odpočítať z rovnovážneho poľa,

        // ale neumožňujú dokončenie výberu, ak nemajú dostatok finančných prostriedkov.

        // Budete chcieť vytvoriť rôzne kódy v hlavnej triede (kód vytvorený spoločnosťou IntelliJ)

        // potvrďte, že váš kód funguje.

        // Do dvoch vyššie uvedených metód pridajte aj nejaké System.out.println's.

Uloha2:

Napíšte triedu s názvom SimpleCalculator. Trieda potrebuje dve polia (premenné inštancie) s menom firstNumber a secondNumber obidvoch typov double.

Napíšte nasledujúce metódy (inštančné metódy):

\* Metóda s názvom getFirstNumber bez akýchkoľvek parametrov, musí vrátiť hodnotu poľa firstNumber.

\* Metóda s názvom getSecondNumber bez akýchkoľvek parametrov, musí vrátiť hodnotu poľa secondNumber.

\* Metóda s názvom setFirstNumber s jedným parametrom typu double, je potrebné nastaviť hodnotu poľa firstNumber.

\* Metóda s názvom setSecondNumber s jedným parametrom typu double, je potrebné nastaviť hodnotu secondNumberfield.

\* Metóda s názvom getAdditionResult bez akýchkoľvek parametrov, musí vrátiť výsledok pridania hodnôt polí firstNumber a secondNumber.

\* Metóda s názvom getSubtractionResult bez akýchkoľvek parametrov, musí vrátiť výsledok odčítania hodnôt poľa secondNumber od firstNumber.

\* Metóda s názvom getMultiplicationResult bez akýchkoľvek parametrov, musí vrátiť výsledok vynásobenia hodnôt polí firstNumber a secondNumber.

\* Metóda s názvom getDivisionResult bez akýchkoľvek parametrov je potrebné vrátiť výsledok delenia hodnôt poľa firstNumber druhým číslom. V prípade, že je hodnota secondNumber 0, vráti 0.

# Písomky

## Prva písomka

1. napis main metodu

na co sluzi.

2. syntax if else.

na co slúži.

3. deklaruj a inicializuj

vsetky prim dat typy a aj String

IsOdd

Napíšte metódu s názvom isOdd s parametrom int a zavolajte jej číslo. Metóda musí vrátiť booleovský výraz.

Skontrolujte, či je číslo> 0, ak nie je vrátené nepravdivé.

Ak je číslo nepárne, návrat je true, inak vráti false.

Napíšte druhú metódu nazývanú sumOdd, ktorá má 2 int parametre a začiatok a koniec, ktoré predstavujú rozsah čísel.

Metóda by mala používať slučku for na spočítanie všetkých nepárnych čísel v tomto rozsahu vrátane konca a vrátenia súčtu.

Mal by zavolať metódu isOdd a skontrolovať, či je každé číslo nepárne.

Koniec parametra musí byť väčší alebo rovný začiatku a počiatočné aj koncové parametre musia byť väčšie ako 0.

Ak nie sú tieto podmienky splnené, vrátite hodnotu -1 z metódy označujúcej neplatný vstup.

Príklad vstupu / výstupu:

\* sumOdd (1, 100); → mala by vrátiť 2 500

\* sumOdd (-1, 100); → mal by sa vrátiť -1

\* sumOdd (100, 100); → mal by sa vrátiť 0

\* sumOdd (13, 13); → mal by vrátiť 13 (Táto sada obsahuje jedno číslo, 13 a je nepárne)

\* sumOdd (100, -100); → mal by sa vrátiť -1

\* sumOdd (100, 1000); → malo by sa vrátiť 247500

FirstLastdigit

Napíšte metódu s názvom sumFirstAndLastDigit s jedným parametrom typu int nazvaného number.

Metóda musí nájsť prvú a poslednú číslicu čísla parametra odovzdaného do metódy pomocou slučky a vrátiť súčet prvej a poslednej číslice tohto čísla.

Ak je číslo záporné, metóda musí vrátiť -1, aby označila neplatnú hodnotu.

Príklad vstupu / výstupu

\* sumFirstAndLastDigit (252); → mala by sa vrátiť 4, prvá číslica je 2 a posledná je 2, čo nám dáva 2 + 2 a súčet 4.

\* sumFirstAndLastDigit (257); → ak sa vráti 9, prvá číslica je 2 a posledná je 7, čo nám dáva 2 + 7 a súčet 9.

\* sumFirstAndLastDigit (0); → by sa malo vrátiť 0, prvá číslica a posledná číslica je 0, pretože máme iba 1 číslicu, ktorá nám dáva 0 + 0 a súčet 0.

\* sumFirstAndLastDigit (5); → malo by sa vrátiť 10, prvá číslica a posledná číslica je 5, pretože máme iba 1 číslicu, čo nám dáva 5 + 5 a súčet 10.

\* sumFirstAndLastDigit (-10); → by sa malo vrátiť -1, pretože parameter je negatívny a musí byť pozitívny.

4. int x = 1;

int y = 1;

int z = 5\*4%3;

výsledok

5. int y = 3;

int z = 10;

(y <= 3 || ( z != y && !false)

vysledok

Písomka 2. skupina B

1. deklaruj a inicializuj (tj. Priraď k premenným hodnoty) 5 primitívnych dátových typov

2. napíš syntax cyklov *while* a *do while*. Aký je medzi nimi rozdiel?

3. napíš syntax výrazu *switch*. Popíš princíp.

4. Oprav danný kus kódu, aby Java mohla skompilovať kód. Sú tam syntaktické chyby.

int x = 1

if( x = 0 ) {

System.out.println(“ tu je nejaky text”);

}

else(

System.out.println( tu je zasa iny else text );

)

else if (x > 0){

System.out.println(“ tu je else if text”)

}

5. aká je syntax metód a ktoré parametre signatúry metódy sú povinné.

Písomka 2 skupina A

1. deklaruj a inicializuj (tj. Priraď k premenným hodnoty) 5 primitívnych dátových typov

2. napíš syntax výrazu *switch*. Popíš princíp.

1. napíš syntax výrazu if / else a popíš princíp.

Pisomka 3.

Vytvor triedu:

Zdroje:   
<http://www.kiwiki.info/index.php/Java_-_abstraktn%C3%A9_triedy>

Trieda Scanner

Objekt Scanner môže analyzovať vstup používateľa zadaný na konzole alebo zo súboru. Zadané hodnoty analyzované triedou Scanner sú nami známe primitívne dátove typy a String. Skener rozdeli svoj vstup

do samostatných tokenov (ktoré sú zvyčajne oddelené medzerou alebo novym riadkom- enter) a potom ich vráti po jednom. Skener poskytuje metódy na konverziu tokenov na hodnoty rôznych typov. java.util.Scanner je časťou Java API, a preto je súčasťou každej java inštalacie.

Príklad použitia triedy Scanner.

Tento kód načíta dve čísla z konzoly a vytlačí ich súčet:

*Scanner in = new Scanner(System.in);*

*int i = in.nextInt(); i*

*nt j = in.nextInt();*

*System.out.println(i+j);*

**Constructors:**

Scanner(InputStream source)

Scanner(File source)

Scanner(String source)

Creates a Scanner object to scan (read) values from the specified source. E.g.,

Scanner console = new Scanner(System.in);

Selected Methods:

String nextLine() Reads and returns the next line of input as a string. (Specifically, it reads and returns all the remaining characters on the line as a character string, and then moves to the next line.)

String next() Reads and returns the next input token as a character string.

double nextDouble() Reads and returns a double value. If the next token cannot be translated to a double, throws InputMismatchException.

int nextInt() Reads and returns an int value. If the next token cannot be translated to an int, throws InputMismatchException.

boolean hasNextLine() Returns true if there is another line of input. Does not read past the line.

boolean hasNext() Returns true if there is another token. Does not read past the token. boolean

hasNextDouble() Returns true if there is another token that can be translated as a double. Does not read past the token.

boolean hasNextInt() Returns true if there is another token that can be translated as a int. Does not read past the token.