01. Programovací jazyk JAVA (historie, zařazení, průběh překladu)

Historie

Green Team

Na počátku zrodu Javy byl vznik Green Teamu vedený Jamesem Goslingem, který měl za úkol vytvořit pro společnost Sun Microsystems jazyk, který bude stručný a efektivní. Pro programování systémů domácích spotřebičů (embedded systémy).

Oak

Green Team se zprvu pokoušel systém pro spotřebiče založit na jazyce C++, poté i na jazyku Pascal. Po neúspěších Green Team navrhl nový jazyk, jenž pochází z C++. Jazyk pojmenovali dle stromu, co rostl před kanceláří Jamese Goslinga, Oak (Dub). Oak se zprvu nemohl prosadit, po neúspěších v PDA telefonech a set-top boxů k televizorům použit pro webové aplikace – applety. Oak oživil v té době statický web a začal se proto používat na psaní webových appletů (1991).

Java

V roce 1995 se zjistilo, že jazyk se jménem Oak už existuje a tak dostal název Java. Java byla poprvé představena v roce 1995 na konference SunWord. Technologie Java byla licencována společnostem IBM, Microsoft, Symantec, Silicon Graphics, Inprise (Borland), Oracle, Toshiba či Novell. V roce 1996 bylo vydáno první Java Develepement Kit – JDK 1.0, které obsahovalo vše potřebné pro tvorbu appletů.

C++

Java pochází z jazyka C++, byly odstraněny konstrukce, které se nepoužívaly, způsobovaly těžké běhové chyby a pointery. Někteří programátoři co přešli z jazyka C++ k Javě tvrdí, že v Javě jejich produkce programů dvojnásobně stoupla.

JVM

Java běží na Java Virtual Machine, která zprostředkovává veškeré propojení s hardwarem, což zaručuje nezávislost na OS a hardwaru počítače. Ovšem i to má svou nevýhodu, operační systémy musí mít JVM nainstalovanou. Na konci 90. let se totiž Sun dostal do sporu s Microsoftem, který si chtěl Javu upravovat podle sebe. Vše to vyvrcholilo soudním sporem a nakonec odebráním licence Microsoftu.

Dnes

Dnes je Java jeden z nejpoužívanějších programovacích jazyků. Programy v Javě se píší pro mobily, osobní počítače, servery a internetové aplikace. Největší zastoupení má Java na serverových aplikacích.

Překlad a spuštění

JVM

Java Virtual Machine obstarává vazbu na hardware a na JVM běží všechny java programy. JVM je součástí balíčků jak JDK (Java Develepement Kit) tak i JRE (Java Runtime Environment).

JDK

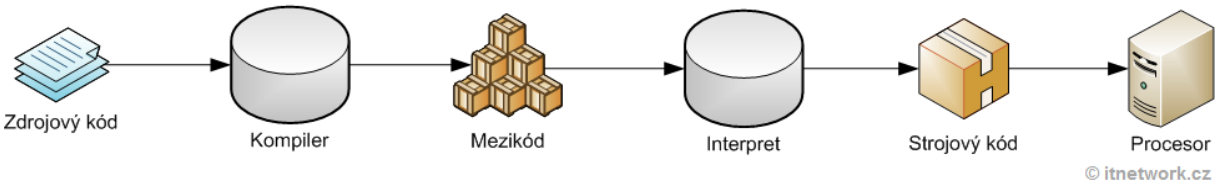
Java Develepement Kit obsahuje JVM a také spoustu předepsaných tříd uspořádaných v balíčcích v jedné velké knihovně. JDK je potřeba pro překlad programu.

JRE

Java Runtime Environment obsahuje JVM a knihovny stejně jako JDK, hlavní rozdíl je, že JRE je určeno jen pro spouštění javovských aplikací, nikoliv pro překlad.

Překlad

Zdrojový kód je nejprve přeložen (**kompilerem**) do tzv. mezikódu (**bajtkód**). Jedná se v podstatě o binární kód, který má jednodušší instrukční sadu a přímo podporuje objektové programování. Tento mezikód je potom díky jednoduchosti relativně rychle interpretovatelný virtuálním strojem (**interpretem - JVM**). Výsledkem je strojový kód pro procesor.



JIT Compiler

V době zavedení programu z disku do paměti počítače (Po ověření správnosti bajtkódu) jej přeloží do strojového jazyka konkrétního počítače (prakticky vytvoří v paměti EXE soubor). Ten pak běží stejnou rychlostí jako kterýkoliv jiný zkompilovaný program napsaný třeba v C.

Paměť:

Paměť javy je rozdělena do 4 segmentů:

Data Segment

Obsahuje globální a statické údaje (data), které jsou přímo inicializována uživatelem. Další část Data Segmentu je BSS (Block Start with Symbol), kde OS inicializuje paměťové bloky na 0, tato oblast je fixována a má statickou velikost.

Code Segment

Segment, kde se nachází aktuální zkompilovaný Java code (bajtkód)

Stack Segment

Obsahuje veškeré vytvořené objekty v běhu.

Heap

Zde se nacházejí deklarované proměnné.

Výhody:

- **Odhalení chyb ve zdrojovém kódu** – Díky kompilaci do bytekódu se jednoduše odhalí chyby ve zdrojovém kódu.
- **Stabilita** – Díky tomu, že interpret kódu rozumí, zastaví před vykonáním nebezpečné operace a na chybu upozorní
- **Jednoduchý vývoj** – Jsou k dispozici hitech datové struktury a knihovny, správu paměti za nás provádí garbage collector.
- **Rychlost** – Rychlost se u virtuálního stroje pohybuje mezi interpretem a kompilerem. Virtuální stroj již výsledky své práce po použití nezahazuje, ale dokáže je **cachovat**, sám se tedy optimalizuje při četnějších výpočtech a může dosahovat až rychlosti kompileru. Start programu bývá pomalejší, protože stroj překládá společné využívané knihovny.
- **Málo zranitelný kód** – Aplikace se šíří jako zdrojový kód v bytekódu, není tedy úplně jednoduše lidsky čitelná.
- **Přenositelnost** – Hotový program poběží na každém HW, kde je virtuální stroj.

Další výhodou Javy je, že je zcela zdarma a tedy dostupná všem vývojářům. Aplikace v Javě lze také spouštět přímo ve webovém prohlížeči pomocí Java Web Start.

Jazyky s virtuálním strojem ctí objektově orientované programování a jedná se o současný vrchol vývoje v této oblasti.

Zařazení

Jednoduchý, Objektové orientovaný, distribuovaný, interpretovaný, bezpečný, nezávislý, přenositelný, procedurální

- **Jednoduchý** – Jeho syntaxe je zjednodušenou (a drobně upravenou) verzí syntaxe jazyka **C** a **C++**. Odpadla většina konstrukcí, které způsobovaly programátorům problémy, na druhou stranu přibyla řada užitečných rozšíření.
- **Objektově orientovaný** – s výjimkou osmi primitivních datových typů jsou všechny ostatní datové typy **objektové.**
- **Distribuovaný** – Je navržen pro podporu aplikací v síti (podporuje různé úrovně síťového spojení, práce se vzdálenými soubory, umožňuje vytvářet distribuované klientské aplikace a servery.
- **Interpretovaný** – místo skutečného strojového kódu se vytváří pouze tzv. **mezikód** (bytekód). Tento formát je nezávislý na **architektuře počítače** nebo zařízení. Program pak může pracovat na libovolném počítači nebo zařízení, které má k dispozici interpret Javy (JVM). V současných verzích Javy se využívá **JIT compiler**. Tato vlastnost zásadním způsobem zrychlila provádění programů v Javě, ale výrazně zpomalila start programů. V současnosti se převážně používají technologie zvané HotSpot compiler, které mezikód zpočátku interpretují a na základě statistik získaných z této interpretace později provedou překlad často používaných částí do strojového kódu včetně dalších dynamických optimalizací (jako je např. inlining krátkých metod atp.).
- **Robustní** – Je určen pro psaní vysoce spolehlivého **softwaru** – z tohoto důvodu neumožňuje některé programátorské konstrukce, které bývají častou příčinou chyb (např. správa paměti, příkaz goto, používání ukazatelů). Používá tzv. silnou **typovou kontrolu** – veškeré používané proměnné musí mít definovaný svůj datový typ.

- **Generační správa paměti** – Správa paměti je realizována pomocí automatického **Garbage collectoru,** který automaticky vyhledává již nepoužívané části paměti a uvolňuje je pro další použití. To bylo v prvních verzích opět příčinou pomalejšího běhu programů. V posledních verzích běhových prostředí je díky novým algoritmům pro garbage collection a tzv. generační správě paměti (paměť je rozdělena na více částí, v každé se používá jiný algoritmus pro garbage collection a objekty jsou mezi těmito částmi přesunovány podle délky svého života) tento problém ze značné části eliminován.
- **Bezpečný** – Vlastnosti, které chrání počítač v síťovém prostředí, na kterém je program zpracováván, před nebezpečnými operacemi nebo napadením vlastního operačního systému nepřátelským kódem.
- **Nezávislý na architektuře** – Vytvořená aplikace běží na libovolném operačním systému nebo libovolné architektuře. Ke spuštění programu je potřeba pouze to, aby byl na dané platformě instalován správný virtuální stroj. Podle konkrétní platformy se může přizpůsobit vzhled a chování aplikace.
- **Přenositelný** – Vedle zmíněné nezávislosti na architektuře je jazyk nezávislý i co se týká vlastnosti základních datových typů (je například explicitně určena vlastnost a velikost každého z primitivních datových typů). Přenositelností se však myslí pouze přenášení v rámci jedné platformy Javy (např. J2SE). Při přenášení mezi platformami Javy je třeba dát pozor na to, že platforma určená pro jednodušší zařízení nemusí podporovat všechny funkce dostupné na platformě pro složitější zařízení a kromě toho může definovat některé vlastní třídy doplňující nějakou speciální funkčnost nebo nahrazující třídy vyšší platformy, které jsou pro nižší platformu příliš komplikované.
- **Výkonný** – Přestože se jedná o jazyk interpretovaný, není ztráta výkonu významná, neboť překladače pracují v režimu **JIT** a do strojového kódu se překládá jen ten kód, který je opravdu zapotřebí.
- **Více úlohový** – Podporuje zpracování více vláknových aplikací
- **Dynamický** – Java byla navržena pro nasazení ve vyvíjejícím se prostředí. Knihovna může být dynamicky za chodu rozšiřována o nové třídy a funkce, a to jak z externích zdrojů, tak vlastním programem.
- **Elegantní** – Velice pěkně se v něm pracuje, je snadno čitelný (např. i pro publikaci algoritmů), přímo vyžaduje ošetření výjimek a typovou kontrolu.

02. Komentáře a dokumentace v jazyce JAVA (javadoc)

Komentáře

Další velice důležitou věcí, která zpřehledňuje kód, jsou komentáře. **JVM** si komentářů nevšímá a slouží jen pro programátora jako pomůcka. Java definuje celkem 3 druhy komentářů.

Jednořádkové komentáře

Prvním typem jsou jednořádkové komentáře, které jsou uvozeny dvojicí lomítek //. Komentáře jsou platné do odřádkování, poté se zas vše vrací do starých kolejí. Tyto komentáře se nejčastěji používají k vysvětlení použití nějakého složitějšího vzorečku, nebo nějaké neobvyklé konstrukce.

```
//konstuktor
public Constructor(int value) {
      this.value = value;
      this.test = null;
      this.right = null;
}
```

Víceřádkové komentáře

Druhým typem jsou víceřádkové komentáře, které jsou uvozeny lomítkem a hvězdičkou /*. Komentář se ukončuje hvězdičkou a lomítkem */. Vše co je mezi těmito značkami, je v kódu považováno za komentář. Komentář se dá dobře využít jako vysvětlení nějakého nastávajícího bloku příkazů.

```
/* První komentář k programu
 * a ještě jeden!
 */
 public Constructor(int value) {
      this.value = value;
      this.test = null;
      this.right = null;
 }
```

Dokumentační komentáře

Posledním typem jsou dokumentační komentáře, které jsou uvozeny lomítkem a dvojicí hvězdiček /**. Komentář se ukončuje hvězdičkou a lomítkem */. Dokumentační komentáře se používají při deklaraci datových složek a vytváření metod. Obsah komentářů zpracovává program **javadoc.exe**, který z nich vytvoří dokumentaci třídy.

```
/**
 * Testovací metoda MAIN
 * <ul>
 *
 * </ul>
 * @param args parametry z příkazové řádky
 */
```

Javadoc.exe

Javadoc je speciální program, který z napsané dokumentace vygeneruje dokumentační html stránku, která se nachází ve složce **dist => javadoc** a zde je to html dokument jménem index.html.

Formátování

V dokumentačních komentářích jsou povoleny html tagy pro formátování textu. Javadoc je zpracuje a fungují úplně stejně jako v normálním html dokumentu.

```
/**
* <b>Testovací metoda MAIN</b>
* <ul>
*
* </ul>
* @param args parametry z <i>příkazové řádky</i>
*/
```

Speciální dokumentační značky

- V dokumentačních komentářích se používají speciální dokumentační značky. Všechny začínají znakem **@** a následuje speciální slovo, které má svůj význam.
- Globálně se dá používat **{@code}**, která značí, že text mezi složenými závorkami, bude napsán neproporcionálním písmem jako zdrojový kód.
- U deklarace třídy se pro jméno autora používá spojení **@author**, za kterým je uveden autor popřípadě autoři. Dále se používá **@version**, za kterým je uvedena verze kódu.
- U deklarace metod se pro parametry používá značka **@param** jméno_parametru důvod_parametru. Dále se používá **@return**, za který se uvede, co metoda vrací. Pokud metoda nemá žádné parametry, **@param** se prostě neuvede, to samé platí pro **@return**.

Dokumentace tříd

Na začátku každé třídy by měla být dokumentace o třídě, co třída představuje, co dělají její instance. Dále autor a verze projektu. Dokumentace by měla být co nejvýstižnější. V javadoc se pak připojí na začátek dokumentace o třídě.

```
/**
 * Třída, která kontroluje, zda je výraz správně uzávorkován.
 * @author „Jakub Čábera“
 * @version 1.0
 * Zadání:
 * Naprogramujte a odlaďte v jazyce JAVA program, který v řetězci
převezme příklad obsahující závorky.
 * Všechny typy () [] {}
 * Program zkontroluje, zda je výraz správně uzávorkován.
 */
```

Dokumentace metod

Dokumentace metody by měla být hned nad deklarací metody. Dokumentace metody se v **javadoc** připojí k seznamu metod. Na začátku dokumentačního komentáře by mělo být, co metoda dělá. Pokud metoda přebírá parametry nebo nějaké vrací, měly by následovat **@param** a **@return.**

```
/**
 * Šifrovací metoda
 *
 * @return zahashovaný řetězec
 * @param args parametry z příkazové řádky
 */
```

03. Objekty (třída, instance, konstruktor, garbage collector)

OOP

Object-oriented programming – objektově orientované programování.

Třída

Obecný předpis objektu (šablona pro instance). Obsahuje formální proměnné a metody pro instance → Určuje, jak bude objekt vypadat a jak se bude chovat.

Skupina s podobnými vlastnostmi, která nese informace o instancích. Data mohou být buď proměnné primitivního datového typu, nebo odkazy na další objekty.

Program obsahuje vždy alespoň jednu třídu označenou klíčovým slovem **class** a ohraničenou složenými závorkami *{ }*.

Instance

Konkrétní objekt (instance; klon třídy), který je vytvořen podle vzoru třídy se skutečnými hodnotami. Termíny objekt a instance se často zaměňují.

Konstruktor

- Složí k inicializaci proměnných objektu.
- Žádný návratový typ se neuvádí, i když konstruktor vždy vrátí vytvořený objekt.
- Metoda, která vrací referenci na objekt, může mít libovolný počet parametrů různých typů.
- Má vždy stejné jméno, jako je jméno třídy.
- Každá třída, kde není uveden konstruktor, má svůj vlastní bezparametrický (init()). Pokud je napsán jiný, tento překrývá.

Pomocí tříd se vytvářejí jejich instance, které mají stejné metody, ale jiné vlastnosti.

Vytvoření objektu

Třída Auto může mít vlastnosti barva, motor a metodu pohyb(). Instancí této třídy může být MojeAuto s vlastnostmi: barva – Červená, motor – 2.0 TDI 125kW.

Zavolá se konstruktor s 2 parametry:

```
Auto MojeAuto = new Auto("Červená", "2.0 TDI 125kW");
```

Připravená proměnná MojeAuto nyní v sobě uchovává referenci na skutečný objekt, který je pomocí klíčového slova *new* vytvořen v paměti, s parametry uvedenými v závorce.

```
Scanner sc = new Scanner(System.in);
```

Datové složky

Třídy i objekty si uchovávají své informace v datových složkách (proměnné a konstanty). Datové složky tříd a instancí se zapisují v těle třídy, nikoliv v těle metod, pokud by se deklarovali v těle metody, jednalo by se o lokální proměnné. Datové složky třídy jsou jedinečné pro třídu a pro její objekty, ovšem datové složky objektů jsou jedinečné pro každý objekt zvlášť.

Datové složky třídy jsou statické datové složky, a jak je z názvu zřejmé jsou uvedeny klíčovým slovem static. Tyto složky patří třídě a jsou jedinečné a nezávislý na objektech.

Datově složky objektů (instancí) jsou bez klíčového slova static. Při vytvoření každého objektu se pro něj vytvoří zvláštní datové složky, které uchovávají stav objektu.

V oop platí: co není nutné, aby bylo vidět, vidět být nemusí → *private*. Přístup k nim je potom řešen pomocí **getrů** a **setrů**.

Garbage collector (čistič paměti)

Je součástí Java Virtual Machine a stará se o odstraňování nepotřebných objektů (nepotřebný objekt pozná podle toho, že na něj už neukazuje žádná reference). Lze ho vyvolat pomocí příkazu:

```
System.gc();
```

Finalizér

Metoda objektu, která se spustí, když je objekt rušen garbage collectorem.

04. Atributy tříd (statické vs. instanční, veřejné vs. privátní)

Statické

```
public static void cosi(int a, int b){}
```

Statická metoda (či proměnná) se netýká žádné instance, ale třídy jako celku. Statické metody lze volat, i když neexistuje instance dané třídy. Statické metody využívají statické proměnné.

```
public class Osoba {
    public static int pocetOsob;
}
```

Pomocí statických proměnných se často obchází nepřítomnost globálních proměnných a konstant v Javě.

```
public static final double PI = 3.141592653589793;
```

Instanční

Instanční metody (či proměnné) se vztahují na konkrétní objekt (instanci dané třídy). Instanční metoda může pracovat se statickými proměnnými. Statická metoda nemůže pracovat s instančními proměnnými (lze přes objekt).

Fully qualified name

```
public class TridaSKonstantou {
    public static final int MAX = 10;
}
```

Pak může být kdykoliv použita ve své třídě pod jménem MAX a mimo třídu pod takzvaným plně kvalifikovaným jménem TridaSKonstantou.MAX.

Tím je zabráněno konfliktu jmen konstant, což se v jazyce C často stávalo při použití konstant vzniklých pomocí #define.

05. Atributy tříd (lokální vs. nelokální, viditelnost proměnných – stínění)

Lokální proměnné

Existují pouze v místě svého vzniku až k zániku (obvykle začátek a konec metody)

Nelokální proměnné

Existují mimo metody a jsou všem metodám v celé třídě přístupné.

V Javě neexistují globální proměnné, tedy proměnné, které patří celému programu. Každá proměnná patří buď třídě, instanci nebo metodě.

Lokální třídy

Vnitřní třídy nebo vnitřní rozhraní, které se nadefinují v rámci metod (Třída v rámci metody). Její použitelnost je omezena jen na tuto metodu nebo blok.

Stínění

Situace, kdy se lokální a statické / instanční proměnné stejného jména překrývají (nelokální proměnná xyz X lokální proměnná xyz) → lokální proměnná zastíní instanční. Poté se buď používá klíčové slovo **this** (statické jméno třídy), které označuje, že se pracuje se zastíněnou proměnnou nebo **full qualify name**.

```
public class cosi {
    private int test;

    public cosi(int test) {
        this.test = test;
    }
}
```

06. Metody tříd (deklarace, přetěžování)

Metoda; Podprogram

Část kódu konající nějakou specializovanou funkci, která je zpravidla použita vícekrát. Metody jsou umístěny mimo hlavní program, mohou být spouštěny z main() nebo z jakékoliv jiné metody. Díky tomu, že se na tuto operaci vytvoří metoda, nemusí se pokaždé opakovat tentýž kus kódu, ale stačí pouze zavolat tuto metodu.

Metody v programu vystupují svojí signaturou (jméno a seznam parametrů), dle signatury se v programu metody rozeznávají. V třídě nesmí být dvě metody se stejnou signaturou.

Skládání programu

Všechny netriviální programy v Javě se skládají z tříd, třídy se dále skládají z metod a datových složek (proměnných). Podprogramům se v Javě říká metody, v jazyce C/PHP jsou to zase funkce.

Statické metody

Statické metody jsou metody, které patří třídě. Pokud se má statická metoda zavolat, nemusí se vytvářet instance:

```
Modifikátor návratový_typ jmeno(seznam_parametrů) {}
```

Deklarace metod

Modifikátor přístupu: Existují 3 modifikátory přístupu (Public, Private, Protected). Modifikátory přístupu jsou jedna z hlavních vlastností samotného objektově orientovaného programování.

Návratový typ: typ hodnoty, kterou vrátí podprogram volajícímu. Metody mohou vracet všechny datové typy a navíc i typ **void** (prázdno) → metoda nic nevrací. Taková metoda se například využívá k výpisům na obrazovku.

Jméno: libovolný název metody. Název metody by měl začínat malým písmenem a vystihovat chování metody. Špatné pojmenování má za následek znepřehlednění kódu. Dále se doporučuje metody pojmenovávat slovesem, které lépe zdůrazní, co daná metoda provádí.

Seznam_parametrů: seznam hodnot, které metoda dostane od svého volajícího.

Metoda může převzít libovolný počet parametrů, dokonce to nemusí být ani určeno, stejně tak, ale nemusí převzít žádné parametry, pak se uvedou prázdné kulaté závorky.

Parametry metody: deklarují se v kulaté závorce za jménem metody, uvede se nejdříve typ proměnné a poté její jméno. Pokud metoda přebírá více hodnot, mezi proměnnými se udělá čárka.

Klíčové slovo return zařizuje vrácení hodnoty, metoda vždy může vracet jen jednu proměnnou a to stejného datového typu jako je uveden v deklaraci metody. Pokud metoda nic nevrací, return se neuvádí. Pokud ovšem metodu návratového typu void a chcete ji ukončit předčasně lze použít **return;**

```
public class SoucetMetodou {
	/** metoda převezme dvě hodnoty a vrátí jejich součet */
	public static int secti(int a, int b) {
		return a + b;
	}
	/**
	public static int sectiNeomez(int... a){
		int vysl = 0;
		for(int b : a) {
			vysl += b;
		}
		return vysl;
	}
	public static void main(String[] args) {
		int a = 10;
		int b = 5;
		/** volání metody, metoda vrátí celé číslo */
		System.out.println(secti(a, b));
		System.out.println(sectiNeomez(5, 6, 8, 12));
	}
}

public class VypisMetodou {
	/** metoda nic nepřevezme, a vypíše text na obrazovku */
	public static void pis() {
		System.out.println("Tento text vypíše metoda," + " může se
	volat kolikrát chcete");
	}
	public static void main(String[] args) {
	/** volání metody, metoda nic nevrací */
		pis();
	}
}
```

Přetěžování metody

Pokud je zapotřebí sčítat dvě čísla, tak mohou být buď celá, nebo reálná pak je vhodné využít přetěžování metod.

- Přetížená metoda má stejné jméno, ovšem různý seznam parametrů.
- Pokud se program spustí, vyhledá se metoda, která svým seznamem parametrů odpovídá a ta se zavolá.
- Nedoporučuje se používat přetěžování metod, ovšem pokud potřebujete zavolat metodu pro různé parametry je to nejlepší možný způsob.
- Nejvíce přetěžovaná metoda je **toString()**.

```
public class PretizeneMetody {
	/** metoda převezme dvě celá čísla a vypíše součet */
	public static void secti(int a, int b) {
		System.out.println("Toto vypsala metoda celých čísel\n" +
		"součet je: " + (a + b));
	}

	/** metoda převezme dvě reálná čísla a vypíše součet */
	public static void secti(double a, double b) {
		System.out.println("Toto vypsala metoda reálných čísel\n" +
		"součet je: " + (a + b));
	}

	public static void main(String[] args) {
		int a = 10;
		int b = 5;
		double c = 10.5;
		double d = 20.5;
		/** volání metod */
		secti(a, b);
		secti(c, d);
	}
}
```

07. Parametry metod (předávané hodnotou vs. odkazem, jak je to v jazyce JAVA)

Volání metody

Existují sice i metody, které proběhnou pokaždé stejným způsobem, a tudíž jí při zavolání žádná data nepotřebují (bez parametrické). Nicméně častěji se vyskytuje, že metoda potřebuje pro svůj běh nějaká zdrojová data, podle kterých se bude při svém průběhu řídit. Tato data se jí předají právě prostřednictvím parametru při jejím zavolání.

Příkladem metody bez parametrů je například typický getter- závorky pro parametry jsou prázdné.

```
public class cosi {
    private int test;

    public cosi(int test) {
        this.test = test;
    }
    public int getTest() {
        return test;
    }
}
```

Parametry

Formální parametry

Parametry, které jsou deklarovány v metodě. Tyto parametry zastupují skutečné hodnoty, které se převezmou při volání metody.

Skutečné parametry

Hodnoty, které jsou předány metodě při volání. Skutečné protože při volání je jasná jejich hodnota.

Parametry předávané hodnotou

```
public static int sectiCisla(int a, int b){
    a = 2 * a;
    int soucet = a + b;
    return soucet;
}
```

Volání předchozí metody

```
int paraA = 12;
int paraB = 13;
int vysledek = sectiCisla(paraA, paraB);
```

Při volání metody **sectiCisla** se v paměti alokuje místo pro tuto metodu a při tom současně místo pro formální parametry (int a, int b). Následně se do formálních parametrů zkopíruje obsah skutečných parametrů parA, parB.

I když se v metodě měnila hodnota parametru a, v proměnné parA se nic nezmění.

Parametry předávané referencí

Parametrem ovšem nemusí být jen proměnná primitivního datového typu, ale také pole, nebo přímo objekt. V takovém případě už nedochází ke kopírování skutečných parametrů do formálních, ale je zde využívána jen reference na konkrétní pole nebo objekt.

Při předávání parametrů odkazem (referencí) se předává adresa místa skutečného parametru. To znamená, že hodnota skutečného parametru se při volání metody může změnit.

Odkazem se předávají parametry objektového typu. Výhodou je vyšší rychlost práce s parametrem, protože se nepořizuje kopie skutečných parametrů, ale pouze jejich adresa.

08. Rekurze (princip, příklady)

Metoda, která uvnitř svého těla volá sama sebe. Obvykle na jednodušší zadání.

Obecná definice

- Sebeopakování, které je realizováno bez použití cyklu.
- Rekurzivní algoritmus musí obsahovat podmínku, při které dojde k jejímu ukončení, aby nepokračovala do nekonečna.
- Po každém kroku musí dojít ke zjednodušení problému, jinak je postup neefektivní.
- Důležité je nalezení vztahu mezi řešenými problémy.
- Každý rekurzivní algoritmus lze přepsat na iterační (pomoci cyklů, podmínek, zásobníku).
- Lze ji použít, pokud počet rekurzivních volání roste lineárně.

Nevýhody

- Rekurzivní algoritmy jsou neefektivní pro velké programy, při každém novém volání funkce dochází k vytvoření nové sady lokálních proměnných (HW náročnost).
- Pro rozsáhlé datové soubory může dojít paměť dříve, než je nalezeno řešení.
- Stručný, ale nepříliš přehledný kód.
- Nedoporučuje se pro Fibonacciho posloupnost (roste exponenciálně)

Použití

- Faktoriál
- Hanojské věže
- Binární vyhledávání
- Převod z desítkové do dvojkové soustavy
- Fibonacciho posloupnost
- Quick sort

```
public class test {
     /**
      * Vypočítá rekurzivně faktoriál zadaného čísla
      * @param number cislo  >=0
      * @return faktoriál čísla, -1 v případě neplatného vstupu
      */
     public static int faktorial(int number){
         if(number < 0) return - 1;
         if(number == 0 || number == 1) return 1;
         return number*faktorial(number - 1);
     }
 }
```

09. Principy OOP (dědičnost, zapouzdření, polymorfismus)

Dědičnost

Dědičnost se v Javě provádí pomocí klíčového slova extends v deklaraci třídy.

```
public class Potomek extends Rodič {}
```

Potomek je třída, jenž dědí od jiné třídy (rodiče či předka). Dědí se všechny metody a proměnné, jenž jsou v daný okamžik viditelné.

Pokud se neuvede předek, tak je třída potomek **java.lang.Object**, dědí jeho defaultní metody (**equals(), toString(), finalize()…)**

Pokud se má v konstruktoru potomka zavolat konstruktor předka, použije se klíčové slovo **super(parametry)**. Volání konstruktoru předka musí být vždy na začátku konstruktoru.

```
public class Potomek extends Rodič {
    public Potomek(params) {
        super(params_predka);    //konstruktor předka
        super.toString();        //lze volat metodu předka
        inicializace_vlastnich_dat;
    }
}
```

Při přepisování metody, která se dědí od předka, metoda se normálně přepíše s anotací **@Override** → upozornění překladače, že se daná metoda přepisuje.

```
@Override
public String toString {
    return "Ahoj";
}
```

Nelze dědit (přepisovat) metody, které mají v deklaraci třídy **final.**

Specializace

Jedním z nejčastějších důvodů pro použití dědičnosti je **specializace** existujících tříd a objektů. Při inicializaci získává třída nové datové atributy a chování proti původní třídě → Vytvářejí se potomci z rodičovské třídy.

Generalizace

Opak specializace, při generalizaci jsou potomci (určité objekty), pro které se vytváří společný rodič.

Zapouzdření

Princip zapouzdření (**Encapsulation**) chrání data proti přístupu zvenčí.

Ostatní třídy mohou z objektu získávat informace pomocí **getterů** a některé i měnit pomocí pomocí **setterů**. Vše funguje tak, že datové složky nejsou viditelné pro ostatní třídy. Této neviditelnosti se dosáhne pomocí **private** modifikátorů přístupu. Třídy a objekty nemají k datovým složkám přístup a nemohou je měnit.

Měnit se dají jen pomocí setterů, a to tak že si setter novou hodnotu nejprve zkontroluje a až poté hodnotu přiřadí. V případě že na hodnotě závisí nějaká jiná vlastnost, tak vyvolá metodu dané vlastnosti a pozmění ji.

Polymorfismus

Zastoupení. Vlastnost, která umožňuje zastoupit objekt, kde je očekáván rodič, potomkem. (Tam, kde se očekává rodič, lze použít potomka → umí vše co rodič).

Podstatou polymorfismu je metoda (metody), kterou mají všichni potomci definovanou se stejnou hlavičkou, ale jiným tělem. V Javě nikdy nelze dědit od více tříd zároveň (na rozdíl od C++). K tomuto účelu slouží rozhraní (interface).

Modifikátory přístupu

Znepřístupnění datových složek se provádí pomocí modifikátoru přístupu. Celkem existují čtyři modifikátory. Modifikátory přístupu se umísťují hned na začátek deklarace statických a instančních datových složek, na začátek deklarace tříd i metod. Prakticky vše co má něco společného s OOP, má modifikátor přístupu.

Modifikátor	Třída	Balíček	Podtřídy	Neomezeně
Public	Ano	Ano	Ano	Ano
Private	Ano	Ne	Ne	Ne
Protected	Ano	Ano	Ano	Ne
Bez modifikátoru	Ano	Ano	Ne	Ne

Sloupec třída znamená, jestli je metoda (proměnná) vidět z jiné třídy.

Sloupec balíček určuje, zda je třída vidět z jiného balíčku.

Sloupec podtřída uvádí, zda je metoda (proměnná) vidět svými podtřídami, které jsou v jiném balíčku.

Poslední sloupec se vztahuje pro ostatní balíčky.

10. Principy OOP (dědičnost, rozhraní, abstraktní třída)

Interface

Interface (rozhraní), je speciální druh třídy, která obsahuje soupis metod.

```
public interface DataType {
    public String getValue();
    public String getDataType();
    public void nullValue();
    public void reset();
}
```

Interface je předpis pro třídy, který jim říká, jaké metody musí implementovat. Když poté třída implementuje dané rozhraní, tak se zavazuje, že tyto metody implementuje (doplní do nich zdrojový kód a tím jim dá funkčnost).

V rozhraní se nesmí deklarovat proměnné a žádná deklarovaná metoda nesmí obsahovat zdrojový kód. Pokud bude nějaký zdrojový kód obsahovat, kompilátor vyhodí chybu.

```
public class Trida implements DataType {
    public String getValue() {
        source code;
    }
    public String getDataType() {
        source code;
    }
    public void nullValue() {
        source code;
    }
    public void reset() {
        source code;
    }
}
```

Jedna třída může implementovat i více rozhraní, což je velká výhoda oproti dědičnosti.

V programu nelze vytvářet instance rozhraní.

Abstract class

Třída je abstraktní třídou právě tehdy, když obsahuje alespoň jednu abstraktní metodu. Metodu, která neobsahuje žádnou implementaci podobně jako u rozhraní.

Hlavní rozdíl mezi abstraktní třídou a rozhraním je ten, že v abstraktní třídě se můžou deklarovat proměnné a některé metody mohou obsahovat implementaci.

```
public abstract class DataType {
	private int i;
	public abstract String getValue();
	public abstract String getDataType();
	public abstract void nullValue();
	public void reset(){
		source code;
	}
}
```

Abstraktní metody musí po modifikátoru přístupu obsahovat klíčové slovo **abstract.** Pokud metoda toto slovo neobsahuje, musí mít nějakou implementaci.

Pokud se má podědit abstraktní třída, provádí se to úplně stejně jako, když se dědí obyčejná třída s tím rozdílem, že se musí implementovat všechny abstraktní metody. Třída může dědit pouze jednu abstraktní třídu.

```
public class Trida extends DataType {
	public String getValue() {
		source code;
	}
	public String getDataType() {
		source code;
	}
	public void nullValue() {
		source code;
	}
	/*
	není třeba implementovat
	public void reset() {
		source code;
	}
	*/
}
```

Instanci abstraktní třídy nelze vytvořit.

11. Standardní datové typy jazyka JAVA (rozdělení, rozsahy)

Datový Typ

V programování definuje druh nebo význam hodnot, kterých smí nabývat proměnná (konstanta). Datový typ je určen oborem hodnot a zároveň výpočetními operacemi, které lze s hodnotami tohoto typu provádět

Proměnné

Místo v paměti dle typu, který říká, jakou bude mít velikost a jaké operace s proměnou se dají provádět. Vytvoření proměnné se skládá z názvu typu a jména proměnné. V proměnných se můžou uchovávat celá čísla, znaky, nebo ukazatele na objekt. Proměnná na rozdíl od konstant umožňuje měnit svojí hodnotu.

Deklarace proměnné je vytvoření místa v paměti dle určitého typu. Následně je doporučená inicializace (přiřazení) počáteční hodnoty proměnné, není nutná, ale doporučuje se provádět vždy, kdy je to možné.

Konstanty

Konstanty v jazyce Java jsou stejné jako proměnné s tím rozdílem že se před název typu napíše klíčové slovo **final**. Konstanta nemůže během svého života měnit svou hodnotu.

Deklarace konstanty je stejně jako u proměnných vytvoření místa v paměti určitého typu. Pokud je známo, jaká hodnota se bude do konstanty přiřazovat, pak se může inicializovat ihned. Jinak se konstanta nechá neinicializovaná, a hodnota se jí přiřadí, až bude známa.

Pojmenování

Názvy proměnných a konstant by měli odpovídat jejich použití.

Názvy proměnných musí začínat malým písmenem, dále pokračují také malými písmeny, pokud se název skládá z několika slov, použije se velbloudíNotace (camelCase).

Názvy konstant se udávají velkými písmeny. Pokud se skládá z více slov, používá se podtržítko.

```
public static final double KONSTANTA_PI = 3.14159265359;
```

Primitivní datové typy

Java má 8 primitivních datových typů. Procesor s primitivními datovými typy může pracovat přímo → rychlá práce s těmito typy. Každý primitivní datový typ může nabývat různých hodnot a hodí se pro jinou práci. Jedná se o čtyři celočíselné typy (**byte, short, int, long**), dva typy pro práci s plovoucí desetinou čárkou (**float, double**), znakový typ (**char**), který nabývá hodnot z ASCII tabulky. A poslední je pravdivostní typ (**boolean**).

Celočíselné primitivní datové typy

Primitivní typ	Velikost	Minimum	Maximum	Wrapper Class
Byte	1B	-2^7	$-2^7 -1$	Byte
Short	2B	-2^{15}	$-2^{15} -1$	Short
Int	4B	-2^{31}	$-2^{31} -1$	Integer
Long	8B	-2^{63}	$-2^{63} -1$	Long

Reálné primitivní datové typy

Primitivní typ	Velikost	Minimum	Maximum	Wrapper Class
Float	4B	ANSI IEEE.754	ANSI IEEE.754	Float
Double	8B	ANSI IEEE.754	ANSI IEEE.754	Double

Pravdivostní primitivní datový typ

Primitivní typ	Velikost	Minimum	Maximum	Wrapper Class
boolean	Není přesně definována	false	true	Boolean

Znakový primitivní datový typ

Primitivní typ	Velikost	Minimum	Maximum	Wrapper Class
Char	2B	Unicode 0	Unicode 2^{16}-1	Character

```
char znak = 65; // ASCCI hodnota písmene A
char znak = `A`;
char znak = `u00ff`;
```

Datový typ void

Nelze vytvořit proměnnou datového typu void. Datový typ void pouze udává, že metoda nemá návratovou hodnotu.

Wrapper Class

Obalová třída, každý z 8 primitivních datových typů má „Wrapper Class“, která určuje, co lze provádět s daným datovým typem.

Objektové datové typy

„Referenční datové typy“. Jsou datové typy objektu. Hodnota referenční proměnné (proměnná s objektem) je odkaz na místo v paměti, kde je daný objekt (pole) uložen. Deklarace je téměř totožná s vytvořením normální proměnné, jen místo názvu se uvádí název třídy objektu.

```
String retezec = “Cosi”;
```

```java
public class cosi {
    public static void main(String[] args) {
        float a = 5;
        float b = 0;
        System.out.println(a / b); //Infinity
        float aa = 0;
        float bb = 0;
        System.out.println(aa / bb); //Not a Number (NaN)
    }
}
```

12. Datová struktura pole v jazyce JAVA (deklarace, definice, práce s ním)

Typ pole

Neprimitivní datový typ (dá se dále dělit, obsahuje prvky). V Javě existuje pouze homogenní pole (v poli může být pouze 1 datový typ).

Pole v Javě je skupina stejných datových typů, na které ukazuje proměnná typu pole. Pole je rozděleno na části velké jako daný datový typ, tyto části jsou indexovány od **0** do **pole.lenght-1**. Pokud je pole vytvořeno a neinicializováno prvky pole se automaticky inicializují na nulu.

Pole o velikost 10.

0	1	2	3	4	5	6	7	8	9
56	78	4226	452	32	26	265	659	321	25

Deklarace a konstruktor

```
Datovy_typ[] jmeno;
```

Tento příkaz vytvoří proměnou pole, ale ještě nealokuje místo v paměti.

Místo v paměti se vytvoří konstruktorem.

```
jmeno = new datovy_typ[velikost_pole];
```

Tímto příkazem se v paměti alokuje místo pro proměnou určitého typu tolikrát kolik je velikost pole.

```
Datovy_typ jmeno = {hodnota, hodnota, hodnota};
Datovy_typ[] jmeno = new Datovy_typ[velikost_pole];
```

Přístup k prvkům

Prvky v poli jsou přístupné pomocí jejich indexu. Pro přístup k prvku pole slouží **jmeno[index].** U pole se smí přistoupit jen na prvky, které do pole doopravdy patří, pokud se přistoupí mimo, program vyhodí výjimku (**java.lang.ArrayIndexOutOfBoundsException**).

K přístupu ke všem prvkům pole se používá **for each** nebo **for** cyklus. Každé pole má proměnnou jménem **length**, pro zjištění velikosti pole

```
public class PristupPole {
	public static void main(String[] args) {
		int[] pole = {10, 55, 66, 15, 73};
		for(int i = 0; i < pole.length; i++) {
			if(pole[i] < 20) {
				pole[i] *= pole[i];
			}
			System.out.println(pole[i]);
		}
	}
```

Vícerozměrné pole

Vícerozměrná pole v Javě jsou ve skutečnosti pole plné polí. Z více rozměrných polí se nejčastěji používá dvourozměrné pole.

```
int[][] jmeno = new int[počet_prvků][počet_prvků];
```

Pole se může vytvářet i více než dvou rozměrové, pak se jedná o pole polí, jež má v sobě pole.

```
import java.util.Random;
public class Matice {
	public static void main(String[] args) {
		final int n = 5;
		int[][] matice = new int[n][n + 1];
		Random rd = new Random();
		for(int i = 0; i < matice.length; i++) {
			for(int j=0;j<matice.length + 1; j++) {
				matice[i][j]= random.nextInt(10);
			}
		}
		for(int[] i : matice) {
			for(int j : i) {
				System.out.print(j + "\t");
			}
		System.out.println();
		}
	}
}
```

13. Datový typ řetězec v jazyce JAVA (deklarace, práce s ním, metoda toString)

Nejde o primitivní datový typ!

Třída String

V Javě jsou řetězce prezentovány v objektu typu **String,** který v sobě uchovává řetězec charů.

```
String muj_retezec = "Hello Word";
String muj_retezec = new String("Hello Word");
```

Operace s řetězci

Objekty typu String lze sčítat pomocí **+**.

```
String novy = "Hello Word" + "Gutten Tag";
```

Nebo pomocí metody **concat(String s):**

```
String novy = "Hello word".concat("Guten Tag");
```

Pokud se řetězce sčítají, nemění se původní instance, ale pouze se vytváří nová instance (**imutabilita**). Což je celkem náročná operace, a pokud se provádí častěji, je lepší využít **StringBuffer**, který je pro tento případ vylepšený.

Pokud se dva řetězce porovnávají, nelze použít **==**, ale musí se použít metoda **.equals(String s)**, jelikož == by vyzkoušela, zda se jedná o tentýž objekt.

```
String Hello = "Hello Word";
String Gutten = "Gutten Tag";
boolean stejny = Hello.equals(Gutten);
```

Pokud je potřeba jenom část řetězce, je třeba použít metodu **subString(int start, int end).**

```
String ahoj = "Zdravíčko".substring(0, 4);
```

Pohyb v řetězci

Pomocí metody **charAt(int i)** lze přistupovat k jednotlivým znakům řetězce. První znak má index 0, poslední **length()– 1**.

Metody **indexOf(char)** a **lastIndexOf(char)** vrací pozici prvku v řetězci, první vrátí první výskyt znaku, druhá zase poslední.

Metoda **compareTo(String s)** porovnává řetězce **lexikograficky** (znak po znaku)

StringBuffer

Třída pro řetězce, která uchovává větší pole. Také si pamatuje operace s řetězcem a pomocí speciální metody určí, když je příliš velký nebo malý a určí novou velikost pole.

14. Výčtový datový typ v jazyce JAVA (deklarace, práce s ním)

Enum

Enum, výčtový datový typ, je speciální druh třídy, kde se definuje nový datový typ a uvnitř enumu se definují přípustné hodnoty. Místo klíčového slova class se použije klíčové slovo **enum**. Hlavní důvody použití jsou přehlednost kódu, a ochrana proti neplatným vstupům od méně inteligentních uživatelů.

```
public enum SvetoveStrany {
    Sever,
    Jih,
    Vychod,
    Zapad;
}
```

Jednotlivé hodnoty se od sebe oddělují pomocí čárky, za poslední je středník.

```
SvetoveStrany strana = SvetoveStrany.Sever;
```

Takto vytvořený datový typ, lze porovnávat pomocí operátoru (==), jelikož enum implementuje **Comparable**. Lze ho uložit i do binárního souboru, jelikož implementuje **Serializable**. Enum lze použít i ve switchi.

Enum s konstruktorem a metodami

V Javě na rozdíl od C, lze v enumu používat metody, definovat konstanty, proměnné a používat konstruktor, kde každé hodnotě lze nastavit hodnoty, které se mají předat konstruktoru. Konstruktor musí být privátní.

Důležité metody při práci s Enumem

.ordinal()

Vrátí všechny hodnoty enumu, kterých může enum nabývat.

.values()

Vrátí pořadové číslo pro konkrétní hodnotu (číslováno od nuly)

Použití enumu

Error logy, Měsíce, Světové Strany…

```
public enum Alkohol {
	Rum(38), //instance může nést více atributů
	Vodka(40),
	Pivo(5);
	private int procent;

	private Alkohol (int procent) {
		this.procent = procent;
	}
	public int getProcent() {
		return procent;
	}
}
```

15. ADT - Zásobník (princip, realizace, základní operace)

Element

Jak fronta, tak zásobník jsou architektury, které lze realizovat pomocí lineárních spojových seznamů. Oba mají strukturu tvořenou **elementy**, jenž na sebe ukazuji, proto se nejdříve vytvoří element, který je pro oba lineární seznamy stejný.

```
public class Element {
    private int value;
    private Element next;

    public Element(int value) {
        this.value = value;
        this.next = null;
    }
    public int getValue() {
        return this.value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public Element getNext() {
        return this.next;
    }
    public void setNext(Element next) {
        this.next = next;
    }
}
```

Element udržuje hodnotu daného elementu (int **value**), která se nastaví při vytvoření elementu. Proměnná typu Element (Element **next**) je **ukazatel** na další Element.

Zásobník

LIFO (Last In First Out). Prvek, který byl do fronty přidán jako poslední, je z fronty odebrán jako první.
Základní metody pro práci se zásobníkem:

Konstruktor

Vytvoření nového zásobníku

isEmpty()

Ověření prázdnosti zásobníku

top()

Ukazatel na element na vrcholu.

push()

Přidání elementu na začátek zásobníku. Elementu, který se přidává, se nastaví další prvek na top. Současný top se nastaví na přidávaný element.

pop()

Odebrání elementu ze začátku zásobníku. Odebírání nelze provádět, pokud je zásobník prázdný!! Vytvoří se nový dočasný element, do kterého se uloží aktuální top. Do aktuálního topu se uloží další element (tmp.getNext()). Poté se vrátí dočasný element.

Výhody

Nekonečný (oproti poli, které má pevně definovanou velikost).

Nevýhody

Zabere delší čas na vyhledávání, oproti poli, které má konstantní rychlost.

Využití

DFS

Zásobník v poli

Vytvoří se nový zásobník s parametry:
private int array[]; – pole, které slouží jako zásobník
private int top; – počítadlo prvků
private int max; – maximální velikost zásobníku

Při přidání hodnoty, se přidá hodnota na **array[this.top]** a zvýší se top o 1.
Při odebírání se sníží top o 1, a vrátí se **hodnota array[this.top]**.
Poté se hodí metody pro zvětšení a zmenšení zásobníku.

```
public class Stack {
    private Element top;

    public Stack() {
        this.top = null;
    }
    public boolean isEmpty() {
        if (this.top == null) {
            return true;
        } else {
            return false;
        }
    }
    public Element top() {
        return this.top;
    }
    public void push(Element e) {
        e.setNext(this.top);
        this.top = e;
    }
    public Element pop() {
        if (this.top != null) {
            Element tmp;
            tmp = this.top;
            this.top = tmp.getNext();
            return tmp;
        } else {
            return null;
        }
    }
}
```

16. ADT – Fronta (princip, realizace, základní operace)

Architektura, kterou je možno realizovat pomocí lineárního spojového seznamu typu **FIFO** (First In First Out). První prvek, který se do fronty vloží se z ní i jako první odebere.

Nejprve se zvolí, odkud se budou odebírat prvky a kam se budou ukládat. Ideální stav je aby vršek fronty sloužil k odebírání prvků z fronty. Spodek fronty bude tedy sloužit k přidávání prvků do pole

Základní metody pro práci s frontou:

Konstuktor

Vytvoří prázdnou frontu (first, last ukazují na null)

isEmpty()

Test na prázdnost; pokud se top rovná null → fronta je prázdná

first()

Vrátí referenci na první prvek fronty.

last()

Vrátí referenci na poslední prvek fronty.

add()

Přidá další prvek na konec fronty.

Pokud je fronta prázdná, tak při přidání nového prvku je first právě ten nový prvek, last je nový prvek taktéž. Pokud fronta není prázdná, tak poslednímu prvku se nastaví (**setNext()**) přidávaný prvek a opět se nastaví last jako přidávaný prvek.

remove()

Odebere první prvek z fronty.

Pokud je fronta prázdná, tak nelze nic odebírat → vyhodí se výjimku. Poté se vytvoří nový dočasný Element, do kterého se uloží první prvek fronty. Do prvního prvku se uloží **first.getNext()**, Poté se provede kontrola, zda se první prvek nerovná null → jestli ano, tak se poslední prvek taky nastaví na null. Následně se vrátí dočasný element, který byl na začátku vytvořen.

Výhody

Nekonečný

Nevýhody

Složitější realizace

Použití

BFS

```
public class Queue {
     private Element first; private Element last;
     public Queue() {
        this.first = null;
        this.last = null;
     }
     public boolean isEmpty() {
        if((this.first == null) && (this.last == null)) {
            return true;
        } else {
            return false;
        }
     }
     public Element first() {
          if(this.isEmpty()) {
               throw new NullPointerException("prázdno");
          } else {
               return this.first;
          }
     }
     public Element last() {
          if(this.isEmpty()) {
               throw new NullPointerException("prázdno");
          } else {
               return this.last;
          }
     }
     public void add(Element e) {
          if(this.isEmpty()) {
               this.first = e;
          } else {
               this.last.setNext(e);
          }
          this.last = e;
    }
```

```
    public Element remove() {
        if(this.isEmpty()) {
            throw new NullPointerException("prázdno");
        } else {
            Element tmp = this.first;
            this.first = this.first.getNext();
            if(this.first == null) this.last = null;
            return tmp;
        }
    }
}
```

17. Práce se soubory v jazyce JAVA (textové soubory)

Soubor

Množina bytů, uložená na přenosném médiu.

Existují dva typy souborů. **Binární** (osmibitové), a **Textové** (znakové šestnáctibitové, tvořené pomocí Unicode).

Práce se soubory

Aby bylo možné třídy spojené se soubory používat, musí se naimportovat patřičné knihovny.

Pro práci se soubory a též adresáři, slouží třída **File**. Ta ale nemá žádné metody pro čtení, nebo zápis. Slouží pouze jako „**manažer**“. Je zde proto, aby vytvořila jakýsi interface mezi OS a javou.

```
import java.io.File;
File cosi = new File („./cosi.txt“);
```

Pro editaci souboru se musí použít datové proudy. Tvoří je třídy FILEREADER a FILEWRITER. Tyto metody jsou poděděné od předků **inputStream()** a **outputStream()** a ořezané o binární komunikaci.

```
import java.io.FileWriter;
import java.io.FileReader;
FileReader fr = new FileReader(cosi);
FileWriter fw = new FileWriter(cosi);
```

Nejpoužívanější metody:

cosi.delete() – Smaže soubor, složku (složka musí být prázdná)

cosi.canRead() – Otestuje, zda do souboru lze zapisovat. Vrací **boolean.**

cosi.canWrite() – Otestuje, zda ze souboru lze číst. Vrací **boolean.**

cosi.listFiles() – Vypíše pole všech souborů.

cosi.exists() – Otestuje, zda soubor existuje. Vrací **boolean**.

cosi.length() – Vrátí délku souboru (**long**).

cosi.isDirectory() – Test, zda cesta končí adresářem. Vrací **boolean**.

cosi.isFile() - Test, zda cesta končí souborem. Vrací **boolean**.

cosi.ready() - Testuje, zda je proud připraven pro čtení. Vrací **boolean**.

cosi.read() – Přečte jeden znak. Vrací ho jako **int**.

cosi.read(char[] pole) – Přečte znaky do pole.

cosi.write(int c) – Zápis jednoho znaku.

cosi.write(char[] pole) – Zapíše pole znaků.

cosi.close() – Uzavře a uloží datový proud. Kdyby se proud neuzavřel, soubor by se neuložil. Povinností každého programátora je tuto metodu vždy, po jakékoli práci se souborem, zavolat!

K těmto třídám lze použít vyrovnávací paměti (buffer) pro urychlení vstupu nebo výstupu. Tzv. **BufferedReader** nebo **BufferedWriter**.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
BufferedReader br = new BufferedReader(fr);
BufferedWriter bw = new BufferedWriter(fw);
```

A pomocí **metod**:

cosi.readLine() – Slouží ke čtení souboru po řádcích. Vrací **String**.

cosi.newLine() – Zapisuje řádky.

Pokud nastane chyba při práci se soubory, Java vyhodí výjimky: **NullPointerException, FileNotFoundException, IOException**

18. Chyby v programování a mechanismus výjimek v jazyce JAVA (zachycení a ošetření)

Chyby v programování

V programování může programátor udělat tři druhy chyb.

Syntax Error

Syntaktická chyba je chyba v syntaxi programovacího jazyka, například když se programátor přepíše → překladač neporozumí danému příkazu, protože je blbě napsán a tudíž neexistuje. Tyto chyby vždy odhalí **compiler**.

Semantic Error

Sémantická chyba je chyba v logice programu. Například prohození příkazu, nebo ještě hůř, špatný návrh algoritmu daného řešení. Chybu neodhalí překladač, je to jen a jen na programátorovi. V těchto případech pomůže **Debugger**.

Runtime Error

Run-time error je chyba za běhu programu, run-time chyba může nastat jen za nějakých podmínek. Run-time error pomůže odhalit **Java Virtual Machine**, jelikož vyhazuje výjimky, které říkají, co se stalo špatně a na jakém řádku kódu program spadl.

Výjimky v Javě

Java má velice propracovaný systém výjimek a chyb, které mohou nastat při běhu programu. Třídy, jenž vyznačují nějakou chybu nebo výjimku musí dědit od třech základních tříd nebo od jejich potomků.

Error – Vyznačují chybu, nelze zachytit a program vždy končí.

Exception – Vyznačují výjimku jenž je během programování potřeba ošetřit.

RuntimeException – Jsou výjimky, jenž nepotřebují nijak ošetřit, ale pak program padá.

Ošetření výjimek

V Javě jsou dva způsoby ošetření výjimek.

Propuštění výjimky výš

Při propouštění výjimek výš, se jednoduše za signaturu metody napíše příkaz **throws**. Propuštění (**delegování**) výjimky výš může sloužit k propuštění do vyšší metody, ale nikdy by to nemělo být skrz metodu **main**, jelikož pak program padá s výjimkou. Výjimka by se vždy měla zachytit.

```
public static void metoda() throws FileNotFoundException {
     FileReader fr = new FileReader("file.txt");
}
```

Zachycení výjimky

K zachycení slouží 3 bloky pro práci s výjimkami:

Try-catch: Blok try slouží pro kód, v němž může nastat výjimka. V bloku catch(**ExceptionName ex**) se výjimka zachytí. Blok catch má jako parametr výjimku, která může nastat v bloku try. Bloků catch může být i více, ale je důležité zvolit pořadí výjimek, tak aby byli v kaskádě (od potomka k dědicovi).

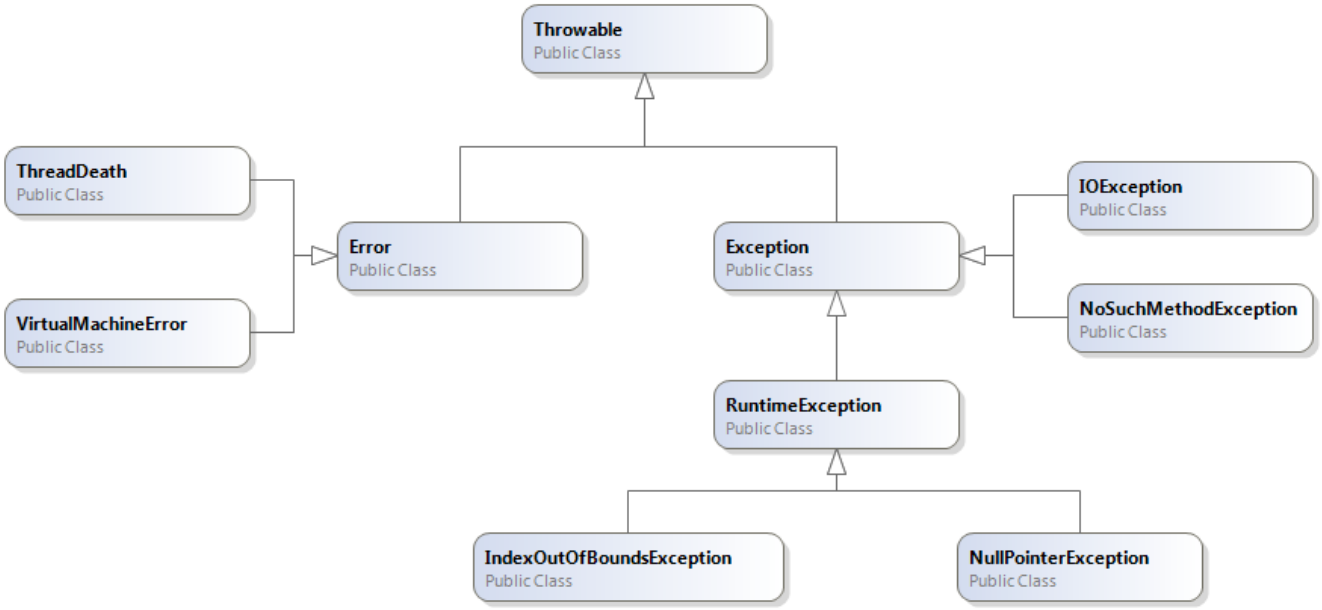
Try-finally: V bloku finally je kód, který se provede v každém případě, ať už skončí blok try vyhozením výjimky, nebo zda proběhne dobře.

Pokud se má výjimka zachytit, tak se musí použit vždy minimálně blok try a jeden z bloků catch nebo finally.

Try-with resources: blok dat, který dovoluje deklarovat jeden či více zdrojů pro daný blok. Zdroj musí být třída, která implementuje **AutoCloseable**. Výhodou tohoto zápisu je to, že se zdroj sám uzavře a uvolní, takže se nemusí použít finally blok. Při klasickém try-catchi s finally uzavřením zdroje je problém s **Výjimkou**, kterou může vyhodit metoda **close()** uvnitř finally metody.

Takže z try-catche vyleze tato Výjimka, i když uvnitř try byla jiná. Try-with resources řeší tento problém, pokud nastane chyba při uzavření zdroje, tak se ven tato chyba nepromítne, pokud byla před ni vytvořena jiná výjimka.

```
public static void metoda() {
	FileReader f = null;
	try {
		f = new FileReader("file.txt");
	} catch (FileNotFoundException ex) {
		System.err.println(ex.getMessage());
	} finally {
		if(f != null) {
			try {
				f.close();
			} catch (IOException ex) {
				System.err.println(ex.getMessage());
			}
		}
	}
}
```


Throwable
Public Class
ThreadDeath
Public Class
VirtualMachineError
Public Class
Error
Public Class
Exception
Public Class
IOException
Public Class
NoSuchMethodException
Public Class
RuntimeException
Public Class
IndexOutOfBoundsException
Public Class
NullPointerException
Public Class

19. Složitost algoritmu - konečný automat vs. Touringův stroj

Složitost algoritmu

Jedna z možností jak porovnat algoritmy v určitých ohledech. Počet kroků nebo operací, které musí procesor vykonat pro různý počet vstupních hodnot. Vyjadřuje, jak se bude měnit chování algoritmu v závislosti na změně vstupních dat. O tom, zda je algoritmus realizovatelný, rozhodují tzv. **matematické modely**, které vyhodnocují složitost pomocí **vyčíslitelnosti** (teorii vyčíslitelnosti)

Matematické Modely:

Konečný automat – deterministický

Deterministic Finite Automaton (**DFA**) je matematický model, pomocí kterého lze určit, zda je daný algoritmus vyčíslitelný.

Skládá se z:

- „Paměti“ (matematický model paměť nemá; pouze při programování)
  - Proměnná
  - Uchovává aktuální stav konečného automatu
  - „Množinu vnitřních stavů“ ve kterých se může nacházet
- Vstupu
  - Čte jednotlivá data, podle kterých se rozhoduje pro další polohu konečného automatu
- Funkce
  - Metoda nebo tabulka, podle které rozhoduje o dalším kroku

Pokud lze na řešení problému použít konečný automat, je algoritmus vyčíslitelný.

Konečný automat musí mít minimálně **2 stavy**, stav pro začátek a stav pro konec.

Z **matematického** pohledu je Konečný automat Bez paměti, ale při simulování tohoto automatu je potřeba ukládat do nějaké proměnné aktuální stav. Při programování se místo 2 stavů používají 3 (4 – chyba): **bez přenosu, s přenosem a koncový**.

Výhoda

- Snadno realizovatelný

Nevýhoda

- Neumí moc věcí

Použití

- Binární sčítačka
- Lexikální analyzátory (v překladačích)

Nedeterministický konečný automat

- Pro stejný vstup existují 2 stavy
- Matematicky mocnější než **DKA**
- Umí simulovat více věcí než DKA
- Jednodušší na kreslení
- Nenaprogramovatelný

Touringův stroj

Matematický model definovaný **Alanem Touringem**. Jedná se o nejsilnější matematický model.

Skládá se z:

- CPU
  - Uvnitř CPU se nachází konečný automat
- Čtecí a zapisovací hlavy
  - Mohou se posouvat na obě strany
- Paměti
  - Nekonečná páska
  - Zápis mezivýsledků

Nevýhoda

- Programátorsky nerealizovatelný
- Pouze teoretický návrh

Použití

- Násobení binárních čísel

Church-Turingova Teze

Podle této teze má každý algoritmus ekvivalentní Touringův stroj.

Halting Problem – Problém zastavení

Znáte-li zdrojový kód programu a jeho vstup, rozhodněte, zda program zastaví, nebo zda poběží navždy bez zastavení.

Problém vymyšlený Alane Touringem. Tento problém nedokáže vyřešit Touringův stroj. Alan Touring dokázal, že obecný algoritmus, který by řešil problém zastavení pro všechny vstupy všech programů, neexistuje.

20. Morseova abeceda (algoritmy převodu)

V Morseově abecedě jsou příslušné hodnoty krátký a dlouhý signál. V zápisu se nechá zapsat jako pomlčka a tečka. Dále na oddělení písmene /, na slovo // a na větu ///. Morseova abeceda vyplývá ze statistiky a pravděpodobnosti.

A	.—	M	— —	Y	—.— —	6	—....
B	—...	N	—.	Z	— —..	7	— —...
C	—.—.	O	— — —	Ä	.—.—	8	— — —..
D	—..	P	.— —.	Ö	— — —.	9	— — — —.
E	.	Q	— —.—	Ü	..— —	.	.—.—.—
F	..—.	R	.—.	Ch	— — — —	,	— —..— —
G	— —.	S	...	0	— — — — —	?	..— —..
H	....	T	—	1	.— — — —	!	..— —.
I	..	U	..—	2	..— — —	:	— — —...
J	.— — —	V	...—	3	...— —	“	.—..—.
K	—.—	W	.— —	4	....—	‘	.— — — —.
L	.—..	X	—..—	5	.....	=	—...—

Implementace převodu textu na Morseovu abecedu

Metodě, která převede text na Morseovu abecedu, se předá String s textem, jenž se má převést. Poté se do nově vytvořeného řetězce přidává vždy kód daného znaku.

Implementace převodu Morseovi abecedy na text

Metodě, která převádí Morseovu abecedu na obyčejný text, se předá hodnota jednoho Morseova písmene a ona vrátí jeho hodnotu v normální abecedě.

Implementace pomocí enumu

Vytvoří se enum, kde budou všechny znaky (v normální a Morseově abecedě). Poté se při překladu vyhledávají příslušné znaky a zapisují se.

Výhoda

- Primitivní
- Snadno realizovatelný

Nevýhoda

- Pomalý (Vyhledávání)

```
public enum Abeceda {
	A('a', ".-"),
	…
	NULA('0', "-----"),
	…

private char c;
private String s;
Abeceda(char c, String s) {
	this.c = c;
	this.s = s;
}

public char getC() {
	return c;
}

public String getS() {
	return s;
}
```

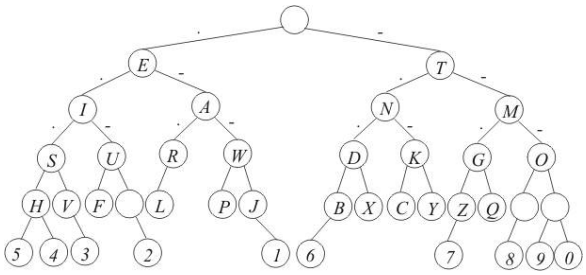
Implementace pomocí pole

Vytvoří se pole Stringů → pozice odpovídají znakům v ASCII tabulce (0 = ‘a‘…). Poté pomocí vzorce $((načtený\_znak) - 'a')$ → A je v ASCII tabulce na pozici 65. Když se od načteného znaku odečte 65, tak vyjde číslo mezi 0 a 25 → index v daném poli.

```
public class TextToMorse{
public static final String morse[] = {".-", "-...", "-.-.", "-..",".",
"..-.", "--.", "....", "..", ".---", "-.-", ".-..", "--", "-.", "---",
".--.", "--.-", ".-.", "...", "-", "..-", "...-", ".--", "-..-",
"-.--", "--..", ".----", "..---", "...--", "....-", ".....", "-....",
"--...", "---..", "----.", "-----", "|"};

public static String textToTranslate = "Eis Was Here";
public static String transleText(String text, String[] array){
          String finalSt = "";
          for(int i = 0; i < text.length(); i++) {
               char znak = text.charAt(i);
               //Malá písmena
               if((znak >= 'a') && (znak <= 'z')) {
                    finalSt += (morse[znak - 'a']);
               }
               if((znak >= 'A') && (znak <= 'Z')) {
                    //Velká písmena
                    finalSt += (morse[znak - 'A']);
               }
               finalSt += '/';
          }
          finalSt += '/';
          return finalSt;
     }
}
```

Při převodu morseovky na text se využívá **binární strom**. Nejvyšší prvek je mezera, která se dělí na ‘.‘ a ‘-‘. Při převodu se začíná na pozici 1 a postupně se čte znak po znaku, když je znak ‘.‘ vynásobí se aktuální pozice * 2, když je znak ‘-‘ vynásobí se pozice * 2 a přičte 1.



```
public class MorseToText{
     public static final char   chars[]   = {'?', ' ', 'E', 'T', 'I',
     'A', 'N', 'M', 'S', 'U', 'R', 'W', 'D', 'K', 'G', 'O','H', 'V',
     'F', '?', 'L', '?', 'P', 'J', 'B', 'X', 'C', 'Y', 'Z', 'Q'};
     public static String morseText =
          "./../...//.--/.-/...//...././.-././";
     public static String morseToText(String morse, char[] array){
          String finalStr = "";
          int    position = 1;
          for(int i = 0; i < morse.length(); i++) {
               char character = morse.charAt(i);
               switch(character) {
                    case '.':
                         position = position * 2;
                         break;
                    case '-':
                         position = position * 2 + 1;
                         break;
                    case '/':
                         finalStr += chars[position];
                         position = 1;
                         break;
               }
          }
          finalStr += chars[position];
          return finalStr;
     }
}
```

21. Jednoduché šifrovací algoritmy (Ceasarova šifra, VIC)

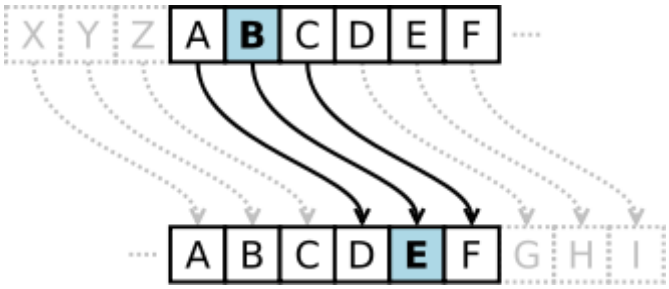
Šifra

Jakýkoli převod otevřeného textu na uzavřený (nečitelný pro běžného uživatele). Přečíst šifru lze na základě znalosti nějaké zvláštní informace, typicky klíče.

CAESAROVA ŠIFRA

Tuto šifru použil poprvé César. Zmínil se o ní ve svých Zápiscích o galské válce.

Šifrování spočívá v tom, že se abeceda posune o zadaný počet písmen.



Tabulka má tedy 2 řádky

A	B	C	D	E	F	…	…	Y	Z
D	E	F	G	H	I	…	…	B	C

POSUN = 3

Zašifrování:

```
((ZNAK – 'A' + ROTACE)%26) + 'A'
```

První znak

Posun

Počet písmen abecedy

Rozšifrování:

```
((ZNAK – 'A' + (26-ROTACE))%26) + 'A'
```

Proto, aby nebylo možné se dostat do záporných čísel

Co se týká bezpečnosti, je tento algoritmus na velmi nízké úrovni, protože nehledě na to jaký je posun, pomocí hrubé síly nejhůře na 25 kroků lze získat správný výsledek.

VIC- ŠIFRA

- Ruská šifra z 50. Let 20. Století (jedna z nekomplikovanějších ručních šifer)
- Mechanismus na dešifrování nebyl nikdy objeven
- Způsob dešifrování by vyzrazen

Důležité

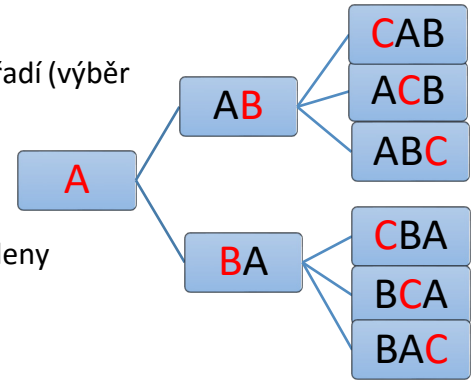
- Permutace čísel 0 – 9
- Šifrovací klíč
  - Tři slova oddělená mezerami (každé písmeno právě jednou)
  - Včetně mezer

	0	1	2	3	4	5	6	7	8	9
	J	D	U		N	A		S	E	X
3	B	C	F	G	H	I	K	L	M	O
6	P	Q	R	T	V	W	Y	Z	_	α

ŠIFROVAT se začíná a končí znakem α tj. 96, mezi nímž je kódovaný text.
EIS WAS HERE – 8357 6557 348628

22. Algoritmy pro generování permutací a řetězců

Permutace n prvků



Skupina všech prvků, které jsou uspořádány v jakémkoliv možném pořadí (výběr prvků závisí na pořadí).

Pokud se prvky ve výběru nemohou opakovat, pak počet všech možných výběrů je určen vztahem $P_{(n)} = n!$

Pokud se hovoří o permutacích prvků, jsou tím obvykle myšleny permutace bez opakování.

Brutal Force

- Prochází již vygenerované číslo
- Pokud najde vygenerované číslo, generuje dál
- + ze začátku rychlé
- – ke konci dost pomalé

Memory Force

Používá pole boleanů, kde jsou všechny prvky **false**, pokud je zadán prvek → **true,** když se zadá další prvek a už má nastaveno **true**, vygeneruje další

- + neprohledává pořád dokola celé pole
- – ke konci pomalé

Dvouprůchodový

- + pořád stejně rychlý → nejrychlejší
- – 2 průchody
  - Vyplní A–C
  - Prohází prvky (složitější)

Generování řetězců

- Rekurzivní algoritmus
- Čísla jsou klíčové znaky… lze jen od 0-9
- Středníky jsou klíčové znaky (nejde je rozšiřovat)
- Musí být také platný vstup
- Vnořování je realizováno rekurzí
- 2a; → aa
- 32ab;c; → 2x ab + 3x c → ab ab c ab ab c ab ab c

Průběh generování

- Používají se dva jezdce
  - Pravý se zvyšuje a hledá klíčový znak (0 - 9)
    - Pokud nenajde a je na úrovni nula (není vnořen) → znak jde do výstupu
    - Pokud najde a není to středník (je to číslo), a je na nulté úrovni tak posune levý jezdec na stejnou pozici jako pravý
- Zvedne úroveň
- Pokud najde středník
  - Sníží úroveň
  - Zkontroluje, jestli není zrovna na úrovni nula
    - Pokud ano tak se použije for s počtem opakování jako hodnota znaku na pozici levého jezdce (bude zákonitě na čísle)
    - Do výstupu potom uloží rekurzivní volání s parametrem substringu od levého jezdce + 1 (tedy hned za číslem) až po pravý jezdec (tedy ke středníku)
- Úroveň se mezi rekurzí nepřenáší! Ta je jen pomocná

Formatter

Při použití formatter je klíčový znak 3 a některé další znaky.

Například „Test %03d“ je decimální číslo na tři místa a volná místa vyplní nulou. To znamená

```
System.out.format(„Test %03d“, 46);
```

Vypíše Test 046

```
public static String generate(String s){
     int level = 0;
     int l = 0, r = 0;
     String finalStr = "";
     for(r = 0; r < s.length(); r++) {
     Boolean cond1= "0123456789;".contains(String.valueOf(s.charAt(r)));
          if(cond1) {
               if(s.charAt(r) == ';') {
                    level--;
                    if(level == 0) {
                    int                              cond                              =
                    Integer.valueOf(String.valueOf(s.charAt(l)));
                         for(int i = 0; i < cond; i++) {
                              finalStr += generate(s.substring(l + 1, r));
                         }
                    } else {
                         if(level == 0) {
                              l = r;
                         }
                         level++;
                    }
               } else {
                    if(level == 0) {
                         finalStr += s.charAt(r);
                    }
               }
          }
     finalStr += " ";
     return finalStr;
}
```

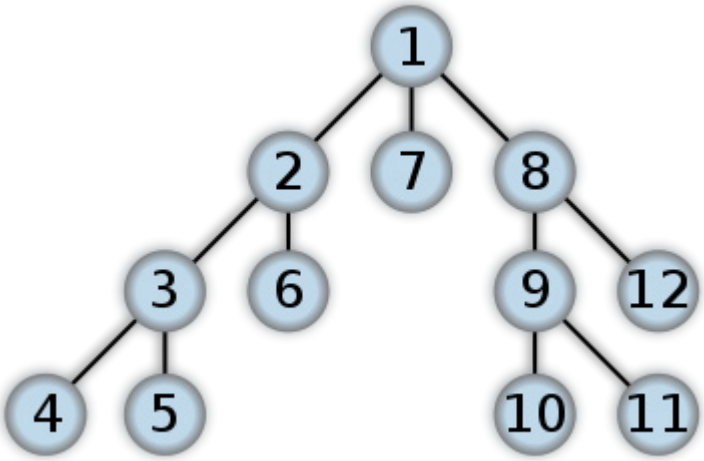
23. Back-track algoritmy (DFS, BFS)

Back-track algoritmus

Zpětné vyhledávání; V množině hledají správně řešení → lze vygenerovat všechna řešení. Organizovaná hrubá síla. Opak hladového algoritmu → heuristika (zkušenost).

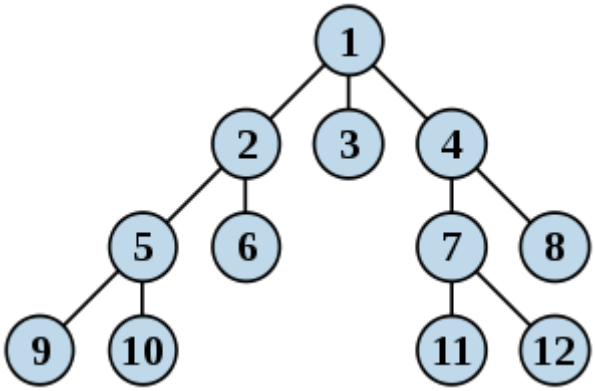
DFS (Depth-first search)

- Prohledávání do hloubky
- Rekurzivní algoritmus
- Pokud existuje řešení, vždy ho najde, ovšem ne to nejkratší
- ***Princip***:
  - Postupuje se stále dál od počátečního uzlu dosud neprozkoumaným směrem.
  - Když už to dál nejde, vrátí se pomocí backtrackingu a postupuje zase co nejdál.
  - Používá pro vrcholy v grafu následující stavy:
    - FRESH (Ještě nebyl objeven)
    - OPEN (Právě objeven)
    - CLOSE (Už byl prozkoumán)
- Př.: šachovnice; 8dam; eternity hra; cesta jezdce
- Lze implementovat pomocí zásobníku



BFS (Breadth-first search)

- Prohledávání do šířky
- Postupuje systematicky (ve vlnách)
- Př.: navigace; nejrychlejší poskládání šachovnice
- Pokud existuje řešení, vždy ho najde (Nejkratší cestu)
- ***Princip***:
  - Algoritmus začne v libovolném počátečním vrcholu.
  - Nejprve projde všechny sousedy startovního vrcholu, poté sousedy sousedů atd. až projde všechny dostupné vrcholy.
  - Pro vrcholy v grafu používá následující stavy:
    - FRESH (Ještě nebyl objeven)
    - OPEN (Právě objeven)
    - CLOSE (Už byl prozkoumán)
  - Vrcholy se při průchodu grafem ukládají na FIFO frontu a z ní jsou posléze odebírány.
  - Každý krok si pamatuje předchozí
- Lze implementovat pomocí front

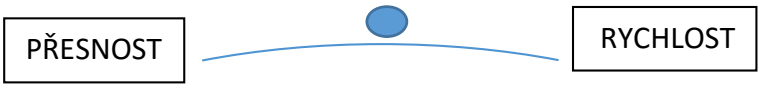


```
void DFS (Graph G) {
//Všechny uzly nastav jako FREH
     for (Node u in U(G)){
          stav[u] = FRESH;
     }
     for (Node u in U(G)){
      if (stav[u] == FRESH) DFS-Projdi(u);
               //Pro každý fresh zavolej DFS-projdi
  }
void DFS-Projdi(Node u) {
     stav[u] = OPEN;          //Pokud je stav OPEN
     for (Node v in Adj[u]){  //Vyhledej sousedy
          if (stav[v] == FRESH) DFS-Projdi(v); //Pokud jsou FRESH;
     zavolej DFS-projdi
     stav[u] = CLOSED;   //Nastav CLOSE
}
void BFS (Graph G, Node s) {
     //Všechny uzly nastav jako FREH (kromě počátečního)
     for (Node u in U(G)-s){
          stav[u] = FRESH;
     }
     stav[s] = OPEN;          //Označ počáteční uzel jako fresh
     Queue.Push(s);           //Ulož ho do fronty
     while (!Queue.Empty()) { //Dokud fronta není prázná
          u = Queue.Pop();    //Vyzvedni uzel
          for (v in Adj[u]) { //Najdi všechny sousedy
               if (stav[v] == FRESH) {  //Pokud mají fresh
                    stav[v] = OPEN;     //Změň na open
                    Queue.Push(v);      //Ulož do fronty
               }
          }
     stav[u]=CLOSED;     //Označ jako CLOSED
     }
}
```

24. Numerické metody (Hornerovo schéma, metoda půlení intervalu, metoda sečen)

Velmi malé procento reálných funkcí na světě, je popsáno tak, že se dají spočítat z hlavy. Většinou mají tak složitě popsané průběhy, že běžnými nástroji v matematice není možné je spočítat. Proto existuje **numerická matematika**, kde se využívá neúnavnosti počítače (opakování výpočtů). Jsou to postupy jak vypočítat výsledky polynomů.

Celá numerická matematika je o poměru přesnosti proti době průběhu. Vždy se musí volit mezi těmito stranami.



Hornerovo schéma

Je zadán polynom a je potřeba zjistit jeho hodnotu v konkrétním bodě (vyhodnocení polynomu).

$$y = 2x^3 - 6x^2 + 2x - 1 \quad X_0 = 3$$

V technice je ale mocnění náročné na výpočty. Proto William George Horner vymyslel jak tuto operaci převést na posloupnost násobení a sčítání.

Řádek		x^3	x^2	x^1	x^0
1.	3	2	-6	2	-1
2.		X	6	0	6
3.		2	0	2	5

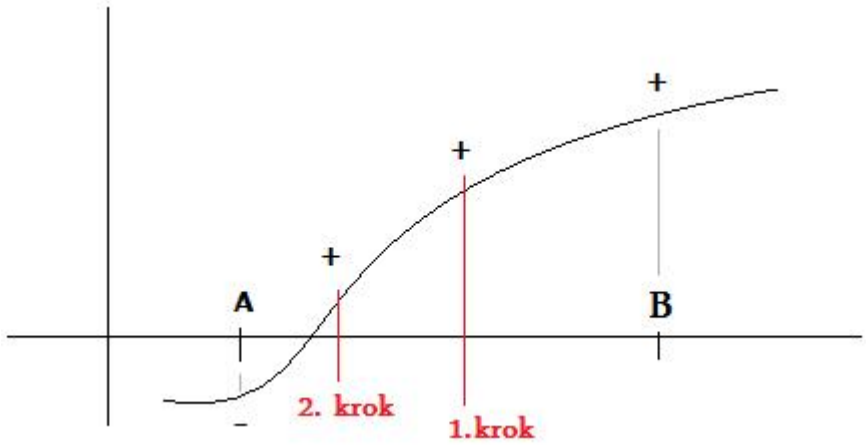
- První řádek se opíše
- Číslo ve třetím řádku je součet dvou čísel nad sebou
- Každé číslo v druhém řádku je součin čísla ve třetím řádku a X_0
- Hodnota v bodě $X_0 = 3$ je 5
- Tato metoda je přesná a je rychlejší než mocnění

Další metody jsou pro řešení kořenu funkcí.

Metoda půlení intervalu (BISEKCE)

Zvolí se interval funkce, ve kterém musí být spojitá (musí mít řešení v daném okamžiku). V tomto intervalu musí procházet bodem X = 0 a musí ve vybraném intervalu řešení existovat.

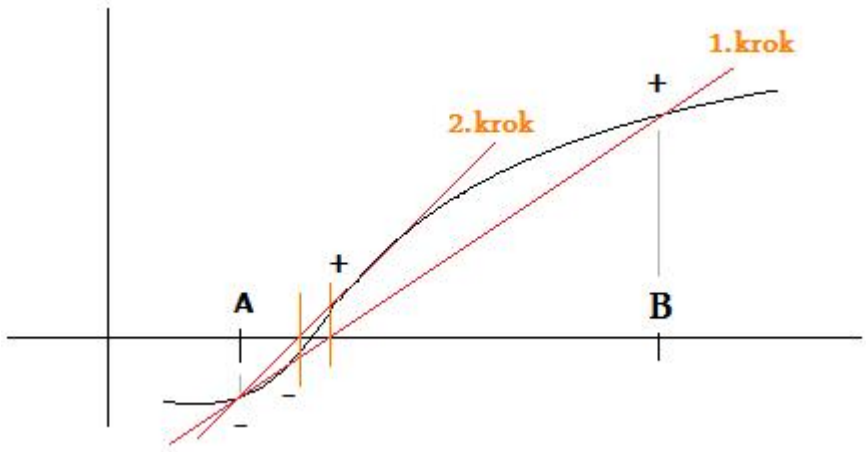
Metoda funguje tak, že rozpůlí interval řešení uprostřed, a zjistí znaménko hodnoty v tomto bodě. Tam, kde se znaménko mění, bude mít funkce řešení. Pokud ještě nedošlo k nějaké chybě, která je v toleranci, dělí se interval dále.



Metoda sečen (REGULA FALSI)

Podmínky jsou stejné. V intervalu musí protínat bod X = 0 a musí být spojitá.

Dvěma hranicemi se proloží sečna (přímka; spočítá se snadněji než daná funkce). Určí se průsečík s osou X a spočítá se bod funkce. Tímto bodem se opět proloží sečna. Tento krok se opakuje, dokud nedojde k chybě, která je v toleranci.



25. SW inženýrství (přístupy k vývoji SW)

Softwarové Inženýrství

Standart IEEE 1993 (softwarové inženýrství) je systematický, disciplinovaný a kvalifikovaný přístup k vývoji, tvorbě a údržbě software.

"Kód, kterému rozumí počítač, umí napsat každý trouba. Dobří programátoři píší kód, kterému rozumí lidé." – Martin Fowler

"Vždy pište kód tak, jako by ten chlapík, co ho po vás bude udržovat, měl být násilnický psychopat, který bude vědět, kde bydlíte." – Martin Golding

Důvody vzniku SW inženýrství

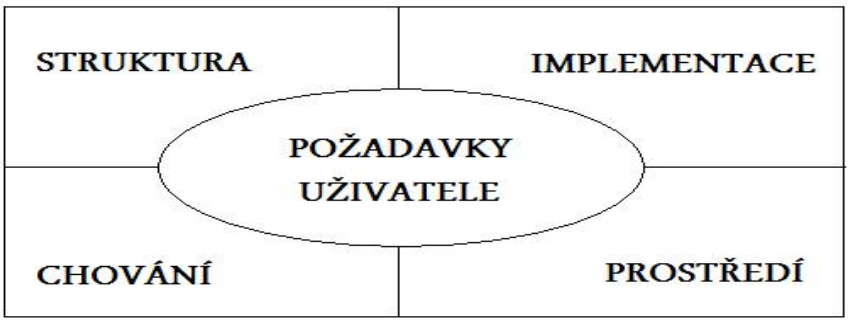
Softwarová krize měla dva hlavní problémy:

- **Rozsáhlost a komplexnost softwaru**
  - Počítače se rozmohly a měli k nim přístup i normální lidé
  - Software se začal zvětšovat a přestalo být možné, aby jeden člověk naprogramoval rozsáhlý software sám → Týmová spolupráce.
- **Opakování chyb**
  - Problémy se pořád opakují
  - Mění se pouze doba, jazyky a hardware
  - Pokud tedy někdo v historii vymyslel řešení daného problému a byl ochoten se o svoji myšlenku podělit, není důvod proč ji nevyužít → Návrhové, Architektonické vzory

UML

- 1994
- Unifikovaný modelovací jazyk
- Množina diagramů, které slouží pro vývojáře, návrháře i údržbu, k domluvě na projektu
- Nezávislé na programovacím jazyce
- Existuje software, ve kterém lze vytvořit UML diagram a zároveň vygenerovat kód v daném jazyce
- UML definuje své diagramy na určité pohledy

4 + 1 Pohled / 4 + 1 View



Požadavky uživatele / zákazníka

Nezajímá ho jak / v čem zadaný program bude vytvořen

Diagram případu užití (**Use case view**) → Z pohledu běžného koncového uživatele → Počítá, že daný software bude něco umět (přihlásit, odhlásit apod.)

Struktura

Statický pohled na systém (Structural View)

Diagram tříd (**Class diagram**) → Z jakých komponent (tříd) se systém bude skládat

Zachycuje z jakých částí, se daný systém bude skládat → Popis jednotlivých komponent

Chování

Dynamický pohled (Behavioral view) – jak se má systém chovat, než úspěšně dokončí určitou akci

Diagram aktivit (**Activity diagram**) → Pohled z dynamického pohledu → co všechno se musí udělat, než bude akce dokončena

Implementace

Pohled na implementaci / ze strany programátora (**Implementation view**) → Pohled programátora. Popis v konkrétním programovacím jazyce. Komponenty, které se musí programovat.

Prostředí / nasazení

Zajišťuje uživatelům, že ten systém bude použitelný pro uživatele

Diagram nasazení (**Deployment diagram**) → Pohledu správce systému → Na jaké technologii software poběží (požadavky)

PRO KAŽDÝ Z POHLEDŮ MÁ UML DEFINOVANÉ SVOJE DIAGRAMY.

Návrhové vzory

Popsané, ověřené a funkční řešení, nějakého problému.

- **Architektonické**
  - Zobrazují z jakých podsystémů, se daný systém bude skládat, a jak budou mezi sebou komunikovat (ISO/OSI, TABULE – komponenty řeší společný problém na tabuli)
  - **MVC**
  - **Pipeline**
  - **P2P**
  - **Server-Client**
  - **SOA – Service oriented architecture**
- **Návrhové**
  - Popis podsystému
  - **Vytvářecí**
    - **Factory**
      - Vyrábí instance podle seznamu
    - **Singleton / jedináček**
      - Od dané třídy existuje jediná instance

- **Strukturální**
  - **Fasade**
    - Vrstva mezi samostatným systémem a třídami
  - **Proxy**
    - Zástupný objekt za nějaký jiný → kontrola přístupu k danému objektu
- **Chování**
  - **Iterátor**
    - Slouží k vytvoření rozhraní, které sekvenčně prochází objekty uložené v kontejneru (složité architektuře)
  - **Observer**
    - Pozorovatel
    - Dohlíží nad systémem

- **Idiomy**
  - Platformě závislé
  - Konkrétní řešení (v daném jazyce) popsáno pomocí idiomů

26. GUI v jazyce JAVA (Swing)

Graphical User Interface

Grafické uživatelské rozhraní → Uživatel používá myš a klávesnici pro ovládání prvků (butony…)

AWT

- Abstract Windows Toolkit
- 1995
- Původní nástroj pro tvorbu grafického uživatelského rozhraní
- Závislý na platformě
- Předchůdce Swingu

Java FX

- Spíše pro grafiky než programátory
- Podpora pro používá multimediálních prvků
- Platformová přenositelnost
- Desktop, Browsery, Mobilní platforma
- Reakce na Adobe Flash, Microsoft Silverlight

SWING

Knihovna uživatelských prvků na platformě Java pro ovládání počítače pomocí grafického rozhraní.

Rozšiřuje a vyvinul se z **AWT**, který měl několik zásadních chyb. Swing tyto chyby opravuje a přidává další funkce. Především opravuje platformovou závislost, vykreslování prvků pomocí knihoven Javy (vzhled prvků je závislý na OS; vzhled lze změnit).

MVC

Softwarová architektura Model – Pohled – Řadič, na které je založen SWING, a která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní:

- Uživatel provede akci (kliknutí na tlačítko)
- Řadič zachytí tuto akci, převezme model a v případě potřeby jej zaktualizuje
- Řadič předá aktuální model pohledu, ten jej zobrazí a čeká na další akci

Komponenty / components

Každá komponenta je zodpovědná za svůj vzhled (nespoléhá se na OS) → definuje jak vypadá.

- JFrame
- JPanel
- JMenu
- JTextField
- JLabel
- JButton
- JToggleButton
- JRadioButton
- JCheckBox
- JTextArea
- JSlider
- JComboBox

Události / events

Událost, která nastane za běhu programu → Zachytí ji **listener** – Objekt, který zachytává určité události a má stanoveno, co má při té akci provést.

Vlastnosti / properties

Stanovují vlastnosti jednotlivých komponent.