Interreg
EVROPSKÁ UNIE
Rakousko-Česká republika
Evropský fond pro regionální rozvoj

INFORMATIKA

Objektově-orientované programování JAVA

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

VŠTE

EVROPSKÁ UNIE

Obsah

1. TŘÍDY

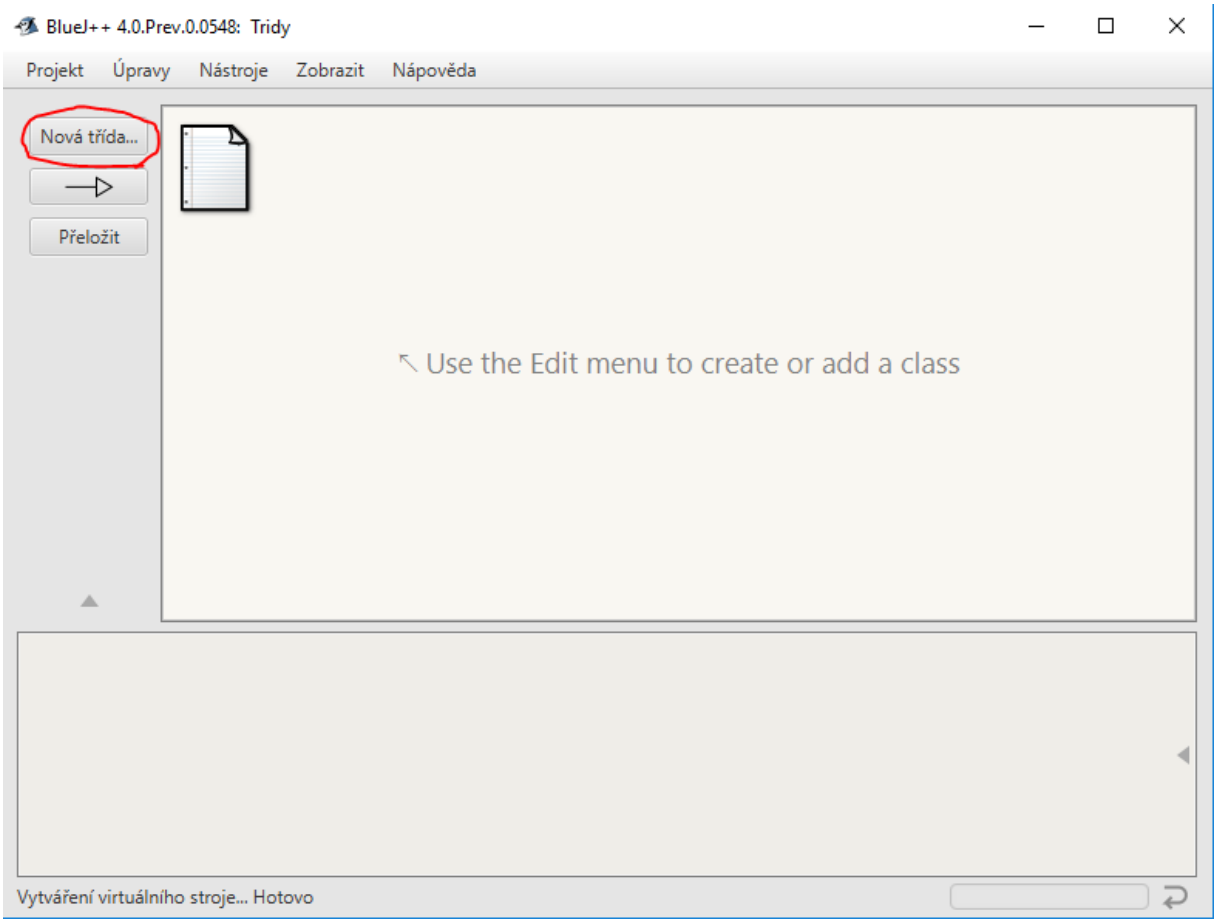
V reálném životě je například slovo židle názvem kusu nábytku, jehož funkcí je, že se dá na ní sedět. Je to tedy popis objektu, který je charakterizován pomocí různých vlastností, či funkcí. Analogií v programování je třída. Třída seskupuje objekty se nějakými společnými vlastnostmi. V reálném životě existuje velké množství rozdílných židlí, které se mohou lišit třeba v materiálu či barvě. V programování konkrétní židli odpovídá instance příslušné třídy, někdy se též nazývá objekt. Třídu tak můžeme přirovnat k formě, do které se odlévá konkrétní věc – instance. Běžně mohou mít třídy libovolný počet instancí.

Jednotlivé objekty mezi sebou mohou komunikovat, navzájem si posílat různé zprávy, ve kterých se mohou žádat o různé informace, nebo služby. Příkladem může být kalkulačka, kterou můžeme požádat o spoustu úkonů, například sečtení dvou čísel. Každá z funkcí této kalkulačky je odborně nazývána metoda. Žádost o využití konkrétní metody se nazývá volání metody. Metoda je tedy část programu, kterou instance pouští jako reakce na volání metody (obdržení zprávy). Tvůrce metody v metodě definuje, jak má objekt na příslušnou zprávu zareagovat.

Celý objektově orientovaný program Pecinovský (2004) popisuje jako v nějakém programovacím jazyce zapsaný popis použitých tříd, objektů a zpráv, které si tyto objekty posílají, doplněný u složitějších programů ještě o popis umístění programů na jednotlivých počítačích a jejich svěření do správy příslušných služebních programů. (Např. OS, nebo aplikačních serverů).

1.1. Tvorba tříd

Nyní již přikročíme k tvorbě samotných tříd. Tedy jakýchsi vzorů či forem, podle kterých vznikají konkrétní instance (objekty). V programu BlueJ, který budeme používat po celou dobu tohoto kurzu, zvolíme nový projekt. Následně zvolíme tlačítko na levé straně. Tak jak to ukazuje obrázek níže.



V následujícím okně zvolíme nejjednodušší možnou definici třídy emptyclass – prázdnou třídu. Dále zvolíme název třídy. V našem projektu jsme zvolili název PrvniTrida.

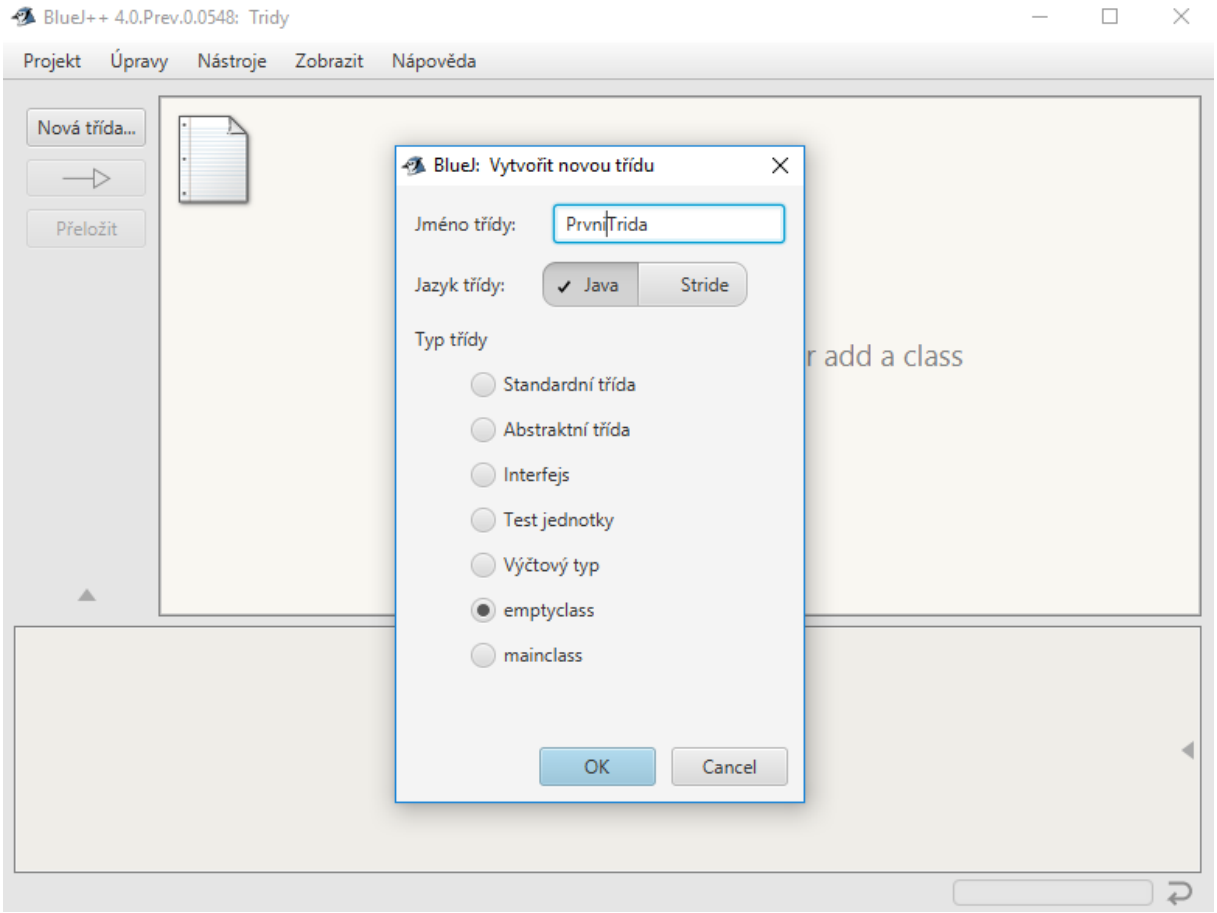
Název třídy se řídí několika pravidly:

Pravidla pro tvorbu názvu identifikátorů

Odkazy, třídy a další, musí být pro snadnou identifikaci pojmenovány. Těmto názvům říkáme identifikátory. Tyto identifikátory musí splňovat několik podmínek:

- Smí obsahovat libovolné znaky, které jsou obsaženy v sadě UNICODE
- Nesmí se používat mezery
- Je zvykem psát víceslovní názvy tříd bez mezer. Jednotlivá slova pak začínat velkými písmeny třeba: MojePrvniTrida
- Velká a malá písmena se považují za rozdílná
- Délka je neomezená
- Nesmí začínat číslicí

- Nesmějí být shodné s klíčovými slovy



Po vytvoření a přeložení máme první třídu. Pokud se podíváme do složky, ve které máme uložený celý projekt. Zjistíme, že ve složce přibilo několik souborů

PrvniTrida.java Tento soubor označujeme, jako zdrojový soubor. Do něj budeme zapisovat program, popisující chování třídy a tedy i jejích instancí. Tento soubor je pouze textovým souborem. Můžete jej tak editovat v libovolném textovém editoru.

PrvniTrida.class V tomto souboru je uložen přeložený bajtkód

PrvniTrida.ctxt Pomocný soubor prostředí BlueJ. Pokud tento soubor smažeme BlueJ si jej při příštím překladu vytvoří znovu

1.2. Práce s třídou

Po dvojkliku na třídu se nám otevře okno, ve kterém může psát, či editovat zdrojový kód třídy. Viz obrázek níže:

V textovém okně uvidíme text – který můžeme nazvat prázdná definice třídy. Tento text za nás automatiky vygeneroval BlueJ v momentě vzniku třídy.

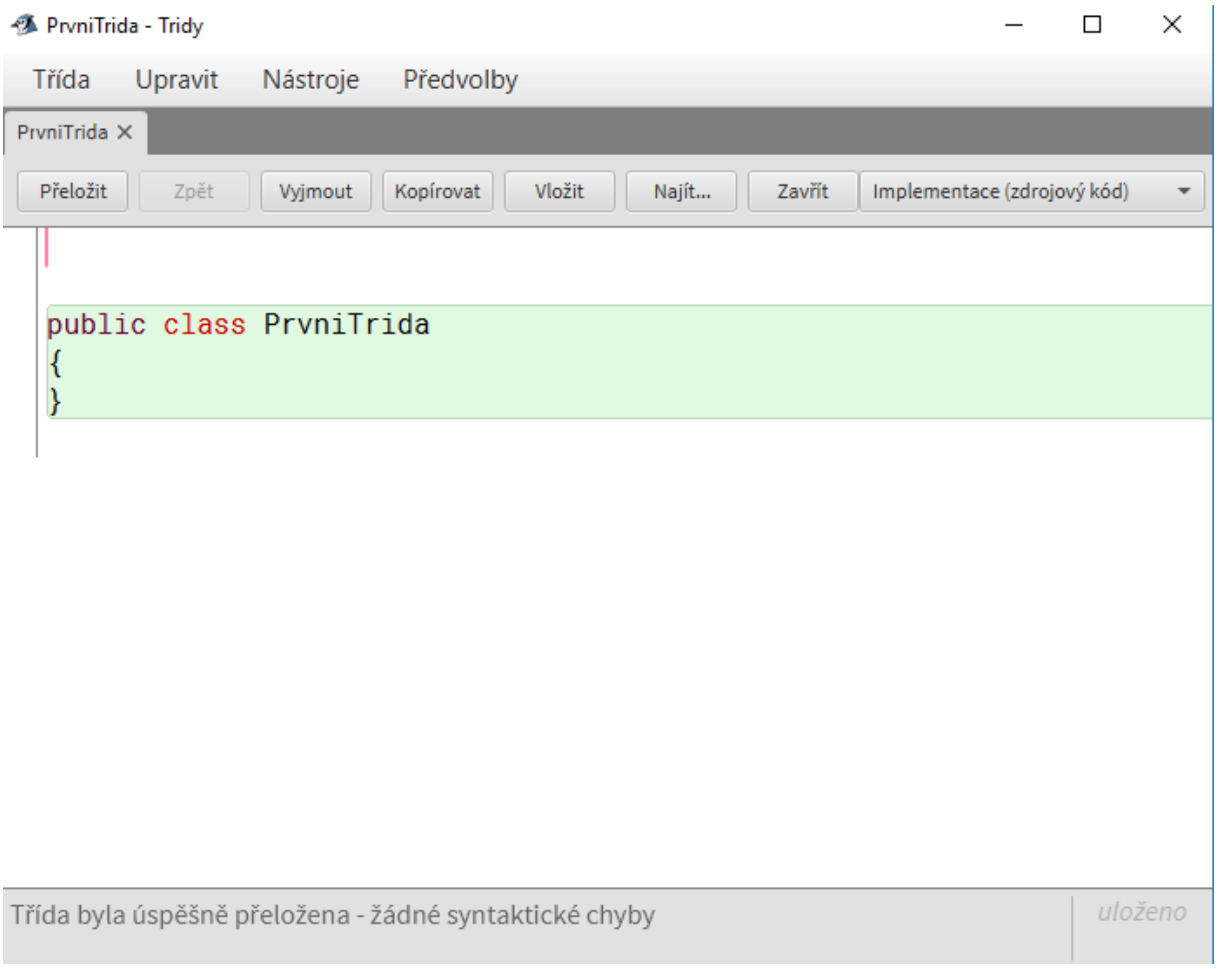
Tento text obsahuje slova

Public Klíčové slovo, které identifikuje, že s třídou může pracovat kdokoliv

Class Klíčové slovo, které oznamuje, že následně bude definice třídy

PrvniTrida Název třídy (její identifikátor)

Následuje samotné tělo třídy, které je uzavřeno v složených závorkách. V našem případě tělo třídy zatím nic neobsahuje.



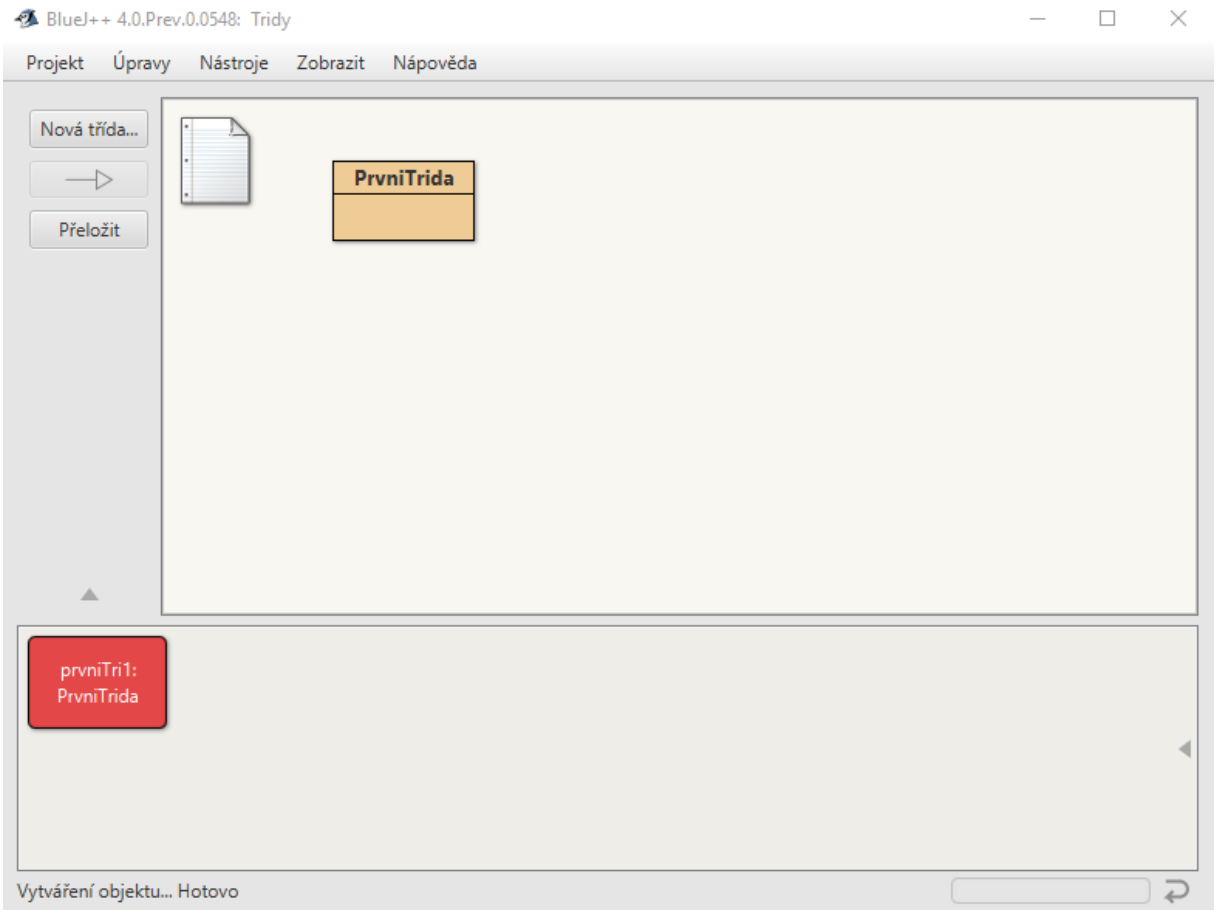
1.3. Vytvoření první instance

Zavřeme textové okno. Pokud klikneme pravým tlačítkem na naši první třídu. Pozor! Dotyčná třída musí být zkompilovaná! V rozbalovací nabídce vybereme příkaz new PrvniTrida(). BlueJ se nás následně zeptá na název právě vznikající instance. Zároveň nám nabídne jím před vybraný název. Který stačí pouze schválit. Čímž jsme vytvořili první třídu. Jak je vidět na obrázku níže.

Vybráním příkaz new PrvniTrida() vytváříme instanci třídy tak, že zasíláme zprávu sestavenou z klíčového slova new, následně uvedeme název třídy, ze které chceme instanci vytvořit dvojici kulatých zvorek. Tím voláme speciální metodu, která se jmenuje konstruktor. Konstruktor požadovanou instanci vytvoří a vrátí nám odkaz, přes který se budeme na vytvořenou instanci obracet.

V našem případě jsme žádnou metodu konstruktoru nevytvořili. Konstruktor musí mít ale každá třída! V případě, kdy nevytvoříme žádný konstruktor, překladač automaticky vytvoří nejjednodušší konstruktor, který označujeme – implicitní konstruktor. V momentě, kdy vytvoříme ve třídě první konstruktor, překladač přestává s tvorbou implicitního konstruktoru.

Vytvořená instance třídy je vidět na obrázku níže. Je v dolní části, v oblasti, kterou nazýváme zásobník odkazů. Instance třídy má červenou barvu.



1.4. Odstranění třídy

Velmi jednoduše. Klikneme pravým tlačítkem na danou třídu a zvolíme možnost odstranit. Po potvrzení je třída opravdu smazána. Samotná instance třídy ale zůstala. Žádost o smazání třídy je vlastně žádostí o smazání příslušných souborů z disku. Samotná instance třídy je uložená v operační paměti. Pokud chceme odstranit i instanci třídy, musíme restartovat virtuální stroj.

1.5. Restartování virtuálního stroje

Restartováním virtuálního stroje smažeme všechny odkazy v zásobníku odkazů. Smažeme tedy všechny instance. Dosáhneme toho buď klávesovou zkratkou Ctrl + shift + r anebo kliknutím pravého tlačítka myši na obdélníček vpravo dole a vybráním příkazu: Restartovat virtuální stroj.

2. KONSTRUKTORY

V předcházející kapitole za nás překladač vytvořil implicitní konstruktor. Nyní si ukážeme, jak tvořit vlastní konstruktory.

2.1. Bezparametrický konstruktor

Vytvořte si třídu student, podobné té, kterou jsme smazali. Nyní si vytvoříme bezparametrický konstruktor. Bezparametrický znamená, že pro svůj běh nevyžaduje žádné informace – parametry. Samotná deklarace třídy a konstruktoru by mohly vypadat na nějak takto. Samotný konstruktor je žlutý:

```
{
        int math =3;
              int english =3;
              int ICT =3;
        public Student ()
        {
              .........
        }
}
```

Nyní si projdeme význam jednotlivých částí textu.

Nejdříve jsme deklarovali třídu Student a proměnné matematiky, cestina a informatika.

Samotná hlavička konstruktoru vypadá dosti podobně, jako hlavička třídy. Jen je vynecháno klíčové slovo class. Název konstruktoru musí být stejný, jako název třídy, ve které je konstruktor uveden. Následují závorky, do kterých buď píší parametry, pak jde o parametrický konstruktor, pokud se do závorek nic nenapíše, půjde o bezparametrický konstruktor. Samotné tělo konstruktoru je v složených závorkách.

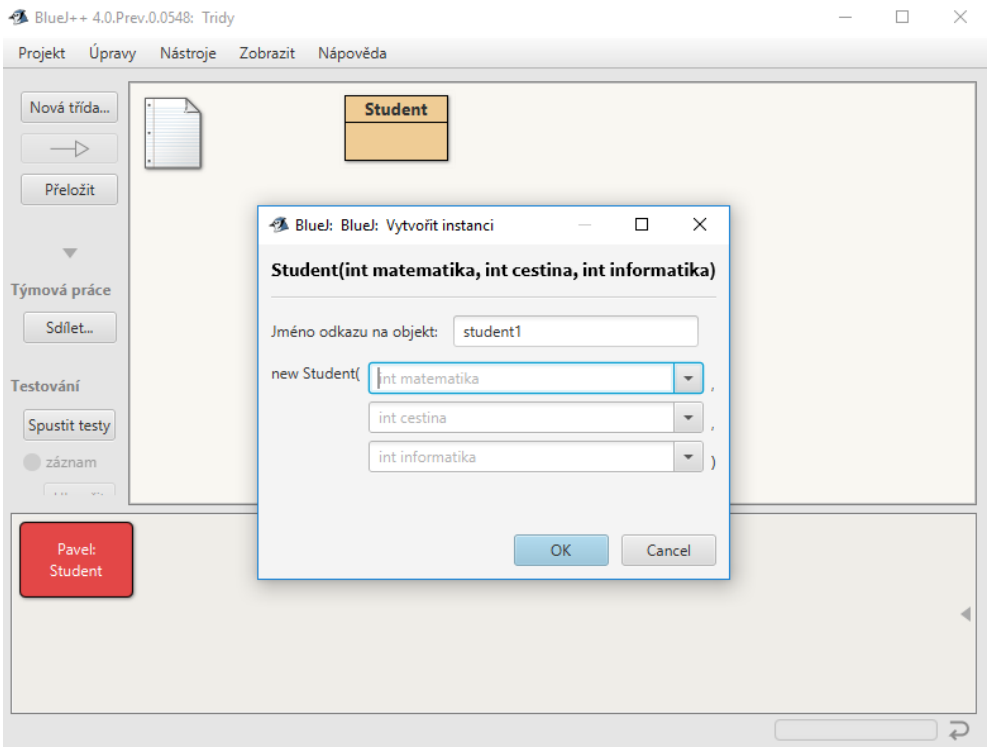
2.2. Konstruktor s parametry

Pokud bychom tvořili instance typu Student, každý ze studentů by měl ty samé známky. Vytvoříme proto tentokrát další konstruktor. Tentokrát již s parametry.

```
public class Student
{
      int math =3;
      int english =3;
      int ICT =3;
      public Student ()
      {
            .........
      }
      public Student (int math,int english,int ICT)
      {
            .........
      }
}
```

Nový parametrický konstruktor jsme žlutě zvýraznili. Všimněte si, že oba konstruktory mají stejný název. Překladač mezi nimi rozlišuje podle počtu a typu parametrů. Do parametrického konstruktoru, uvádíme nejdříve typ a následně identifikátor, jehož prostřednictvím se program na parametr v těle konstruktoru odvolává. Nemůžeme tedy vytvořit konstruktory se shodnými typy parametrů. Překladač dokonce nerozlišuje ani návratové hodnoty u jednotlivých konstruktorů. Využití více konstruktorů nazýváme přetěžování.

Nyní když budeme chtít vytvořit novou instanci – žáka pomocí parametrického konstruktoru, budeme dotázáni, na zadání parametrů typu integer tak, jak je tomu na obrázku níže.



2.3. This

Mnohé konstruktory jsou si navzájem podobné. Mnohdy se stane, že chceme využít v jednom konstruktoru jiný konstruktor. Tomuto opakovanému psaní těla konstruktoru se může vyhnout pomocí klíčového slova this, následovaného seznamem parametrů. Pozor volání jiného konstruktoru pomocí this musí být úplně prvním příkazem konstruktoru! Využití konstruktoru this je na dalším příkladu.

```
public class Student
{
      int math =3;
      int english =3;
      int ICT =3;
      public Student ()
      {
            this (0, 0, 0);
      }
      public Student (int math,int english,int ICT)
      {
            .........
      }
```

3. METODY

Metoda je specifický podprogram, který vykonává, nějakou specifickou funkci. A patří mezi nejčastěji používané nástroje (téměř) každého programovacího jazyka, tedy i jazyka Java. Metody bychom mohli přirovnat k nástrojům, kterými následně je vybavena každá instance třídy. Samotná metoda se skládá z několika částí:

- **Specifikátor přístupu** – který určuje, kdo všechno smí metodu volat. Nejčastěji se používá public a private. Specifikátor je nepovinný
- **Typ návratové hodnoty** – Povinný. Pokud metoda nic nevrací, uvádíme void
- **Název metody** – platí stejná pravidla, jaká byla uvedena u konstruktorů
- **Seznam parametrů metody** – také stejný princip jako u konstruktorů

Samotné tělo metody je uzavřeno do složených závorek. Kam můžeme psát jednotlivé příkazy. Pokud chceme, nemusíme do těla metody napsat nic.

```
public void Hello()
{
      System.out.println(”Hello”);
}
```

3.1. Metody vracející nějakou hodnotu

Pokud má metoda vracet nějakou hodnotu, musíme v hlavičce specifikovat její typ a v těle metody musíme uvést příkaz return, za kterým následují proměnná, kterou chceme vrátit. Po příkazu return bude metoda okamžitě ukončena. Nebude tedy vykonán kód, který v těle metody následuje po příkazu return. Příklad je uveden níže.

```
public double averageGrade ()
{
      double average= (math + english + ICT)/3;
      return average;
}
```

3.2. Volání metody

Samotným napsáním kódy se žádný kód neprovede. Provede se, až když jej zavoláme. Volání metody lze provést pouze z místa, ze kterého je metoda přístupná. Volání metody provádíme napsáním názvu metody a do závorek uvedeme případné parametry metody. Pokud je metoda uvedena v jiné třídě. Musíme nejdříve uvést jméno třídy, samotný název metody oddělujeme tečkou. Posíláme-li zprávu instanci, musíme nejdříve napsat odkaz na tuto instanci. U atributů je situace podobná.

3.3. Předávání parametrů metodám

Hodnoty primitivních typů jako jsou například znaky, logické hodnoty, nebo čísla se předávají tak, že hodnota se překopíruje do lokální proměnné metody

Hodnoty objektových typů se předávají odkazem. Tedy do lokální proměnné metody se pouze nakopíruje odkaz na objekt, ve kterém je skutečný parametr.

4. STATICKÉ ATRIBUTY

Statické prvky patří třídě, ne instanci. Statické atributy označujeme slovem **static**. Díky tomu že patří třídě, k němu mají přístup všechny metody dané třídy. Statické atributy můžeme číst i v případě, že neexistuje instance třídy. Deklarace statického atributu je uvedena níže a je zažlucena.

```
class Group {
      private static int number = 15;
      public void New(int count) {
            number = number + count;
      }
}
```

4.1. Statické třídy

Metody třídy se volají na třídě. Velmi často jde o pomocné metody, které často používáme, ale nechceme speciálně kvůli tomu tvořit instanci. Jako příklad může sloužit statická metoda, která testuje, zda je zadané číslo kladné.

```
public static boolean isPlus(int number) {
      if (number >= 0) {
            return true;
      }
      return false;
}
```

Pozor! Díky tomu, že statická metoda náleží třídě, nemůžeme v ní přistupovat k žádným instančním atributům. Tyto atributy totiž neexistují v rámci třídy, ale instance.

4.2. Lokální proměnné

Občas potřebujeme v rámci metody něco zapamatovat. K tomuto účelu slouží lokální proměnné. Deklarujeme je uvnitř metody. Mimo metodu se k nim nedá přistupovat. Díky čemuž můžeme v jiné metodě definovat jinou lokální proměnnou se stejným názvem. V jejich deklaraci nepoužíváme modifikátory přístupu (např. public a private) ani static. Při jejich deklaraci jim musíme přiřadit nějako konkrétní hodnotu. V momentě opuštění metody se lokální proměnná zruší. Nelze v nich tedy uchovávat nic, co bychom potřebovali mezi různými metodami. Častým důvodem využívání lokálních proměnných bývá zpřehlednění programu a snížení počtu chyb zavlečených v důsledku opakovaného opisování složitých výrazů. Deklarace lokální proměnné je na příkladu níže.

```
public void totalPrice (int pieces)
{
      int totalPrice = pieces *15;
}
```

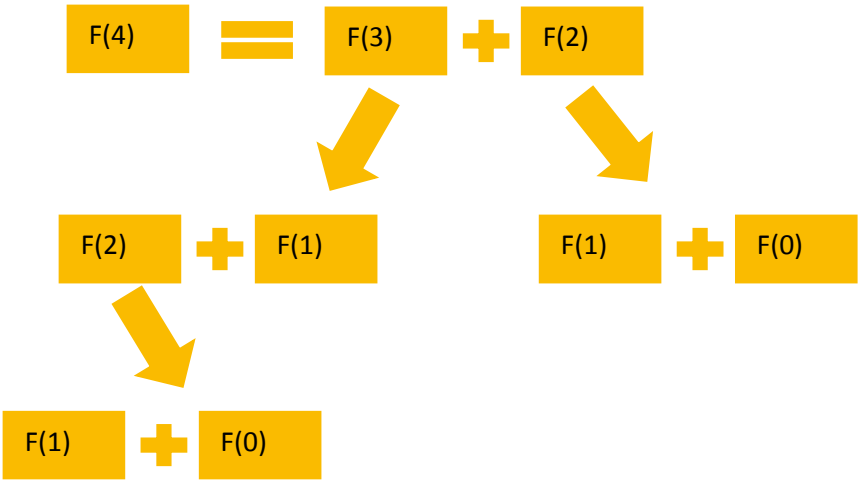
4.3. Rekurze

Rekurze je definování nějakého objektu (chápáno matematicky) pomocí sebe sama. Rekurzivní funkce musí obsahovat nějaký mechanismus, který ve vhodný moment rekurzi ukončí. Pokud by se

Rekurze je definování nějakého objektu (chápáno matematicky) pomocí sebe sama. Rekurzivní funkce musí obsahovat nějaký mechanismus, který ve vhodný moment rekurzi ukončí. Pokud by se tak nestalo, rekurze by probíhala až do „nekonečna“ Klasickým příkladem ukončení rekurze je vložení zarážek.

Klasickým příkladem využití rekurze je například výpočet Fibonacciho posloupnosti. U Fibonacciho posloupnosti je každý člen posloupnosti součtem jeho předchozích dvou prvků. A F(0) = 0 a F(1) = 1.

Příklad výpočtu čtvrtého členu je zobrazen na obrázku níže. Z Obrázku je patrno, že proto abychom mohli spočítat F(4), musíme nejdříve rekurzivně spočítat F(3) a F(2). Pro výpočet F(3) musíme nejdříve spočítat F(2) + F(1), přičemž pro výpočet F(2) musíme nejdříve rekurzivně spočítat F(1) + F(0).



Velkou část výpočtů tak rekurze dělá opakovaně, díky čemuž je výpočetně náročná. Z tohoto důvodu je často lepší využívat klasické cykly. V některých případech je celý algoritmus definován rekurzivně (například Fibonacciho posloupnost) případně nám rekurze může usnadnit práci s některými datovými strukturami.

Konkrétní příklad rekurzivního volání funkce je zobrazen níže. V našem příkladu máme 2 zarážky vyznačené žlutou barvou. Rekurzivní volání je pak vyznačeno zelenou barvou.

```
public static int fib (int n){
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else return fib (n - 1) + fib (n - 2);
}
```

5. ZAPOUZDŘENÍ

WIkipedia (ze dne 7. 5. 2018) mluví o zapouzdření:

Zapouzdření může být vysvětleno jako zabalení dat a metod do jedné komponenty. Funkce zapouzdření jsou dostupné skrze třídy ve většině objektově orientovaných programovacích jazyků. Zapouzdření rovněž umožňuje ukrytí atributů a metod v objektu pomocí stavby nepropustné zdi, která brání kód proti nechtěným změnám.

Což je velmi důležité. Pokud bychom měli složitý program. Mohli bychom omylem ovlivnit chod, nějaké jeho části. Zjištění a náprava takového problémy by nám zabralo zbytečně mnoho času. Zapouzdření je jeden ze základních konceptů objektového programování.

Zapouzdření v jazyce Java je mechanismus zabalení dat (proměnných) a kódu. V zapouzdření budou proměnné třídy skryté z jiných tříd a mohou být přístupné pouze metodami jejich současné třídy. Proto je také znám jako skrývání dat.

Dosáhneme ho tak, že části, ke kterým má mít přístup ostatní funkce označíme jako public. Tato veřejná část třídy. Je nazývána Rozhraní třídy. Do rozhraní třídy je vhodné zařadit pouze to, co o další části programu o dané třídě opravdu nutně musí vědět. U všeho ostatního, u kterého nechceme, aby mohli využívat ostatní části programu, nastavíme private.

Pokud budou nějaké části programu používat nějaké části veřejné třídy, které budou následně změněny. Může tím být ovlivněna jejich funkcionalita. Proto je lepší po zveřejnění třídy její veřejně přístupné části již neměnit.

Často se deklarují všechny atributy třídy jako soukromé. Podle našeho uvážení, pak můžeme zveřejnit přístupové metody (setter, getter) díky kterým si můžeme ohlídat, například, že dotyčný atribut půjde pouze číst a ne editovat, případně ho bude možné nastavovat s nějakými omezujícími podmínkami. (Třeba věk je pouze celé kladné číslo). Shoduje-li se název atributu a názvem parametru a potřebuji-li pracovat s oběma, použiji klíčové slovo this. Toto slovo je používáno jako název třídy, ve které je metoda obsažena.

6. LADĚNÍ PROGRAMU

Velmi často uděláme při psaní programu nějakou chybu. Tyto chyby se dají rozdělit do několika kategorií.

Častým problémem jsou chyby syntaktické. Kdy se prohřešíme proti syntaxi jazyka. Tedy proti pravidlům jak skládat jednotlivé části. Typicky jde například o zapomenutý středník, či závorku.

Tento problém nám překladač velmi často označí ještě před samotnou kompilací. Zároveň nám barevně vyznačí řádek, ve kterém předpokládá, problém. Jak je ukázáno na obrázku:

```
public static int vycetnasobku(int index) {
    if (index < 0)
    return 0
    return index + soucetRekuzivne(index - 1);

}

}
```

Aby mohly být nastaveny zarážky, musí být třída přeložena.

uloženo
Errors: 1

Další chyby se objeví při kompilaci. Pro další informace o chybě můžeme klinout vlevo dole do slova Errors. Po kliknutí se nám objeví další nápověda k chybě.

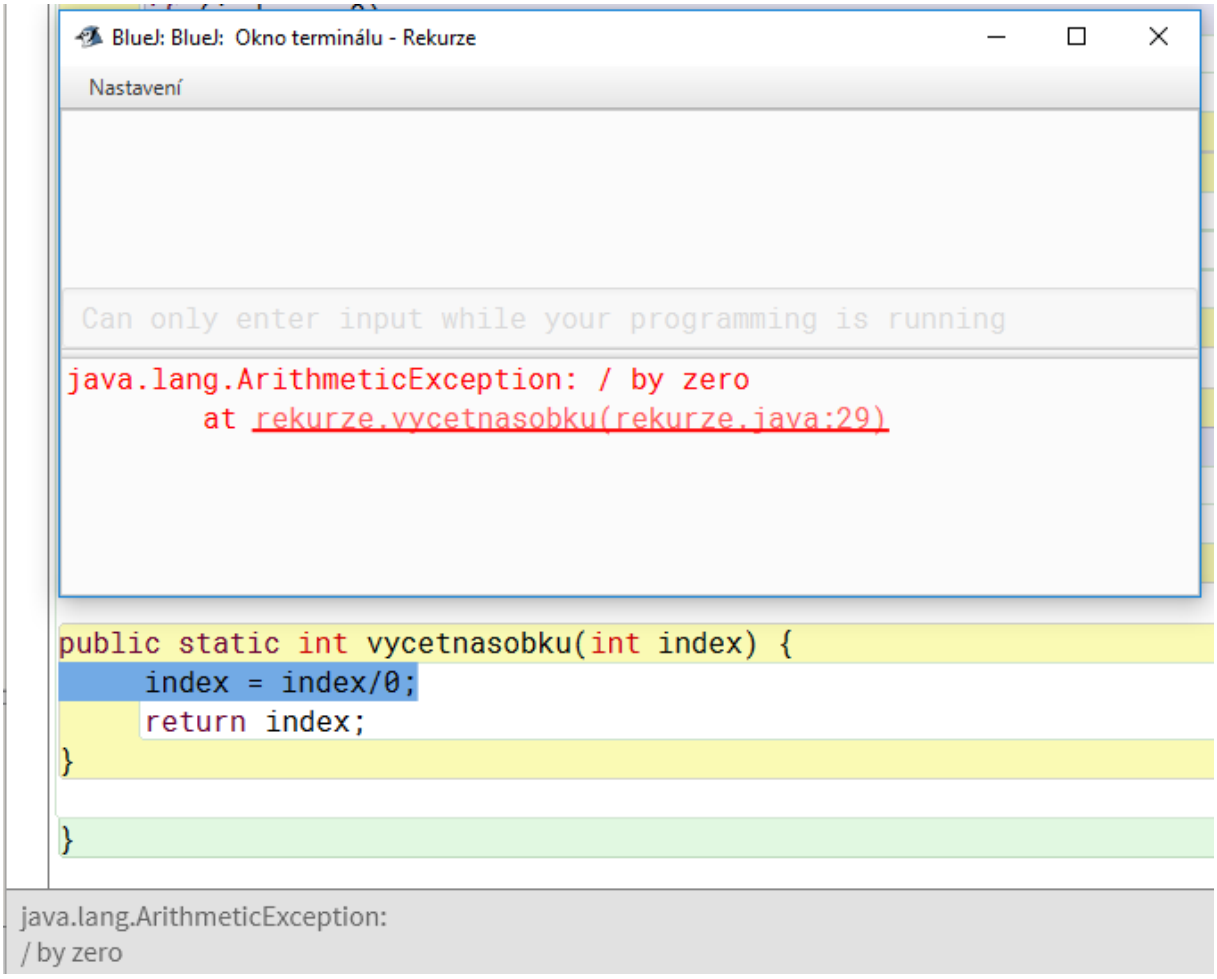
```
public static int vycetnasobku() {
    if (index < 0)
    return 0;
    return index + soucetRekuzivne(index - 1);
            cannot find symbol -   variable index
}

}
```

Error(s) found in class.
Press Ctrl+K or click link on right to go to next error.

uloženo
Errors: 3

Další chyby by se dali pojmenovat, jako běhové chyby. Tyto chyby překladač nenajde. Ale v některých fázích běhu programu mohou nastat. Typickým příkladem je dělení nulou. Kde v momentě, kdy dojde k dělení nulou se program zastaví otevře se terminálové okno, ve kterém se vypíše, o jako chybu se jedná. Zároveň BlueJ označí řádek, ve kterém došlo k problému. Tak, jak je vyobrazeno na obrázku níže. Pro odstranění takového druhu chyb. Je třeba nějakým způsobem ideálně vyzkoušet všechny možné potenciálně problémové

stavy programu. Tak aby se s podobnými problémy samotný uživatel nesetkal. Ideálně, když již dopředu připravíme sadu testových úloh.



Sémantické chyby jsou pak nejzáludnějším typem chyb. Kdy kompilátor nic neobjeví, program zdánlivě pracuje bez problémů. V některých neočekávaných momentech ale program vykazuje chybu. Což může být velký problém.

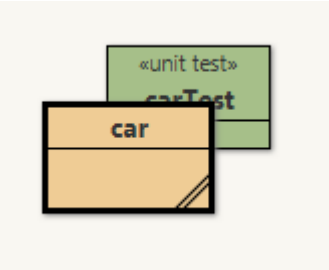
Příkladem může být: Havárie přistávacího modulu na Marsu (1999) Problém komunikace mezi komponentami – uživatel rozhraní očekával hodnotu v kilometrech, poskytovatel ji udával v mílích. Namísto plánovaných 140 až 150 kilometrů tedy zamířila pouze 57 kilometrů nad povrch. V té výšce je však atmosféra Marsu na sondu příliš hustá. Climate Orbiter shořel nejspíše ve výšce kolem 80 kilometrů. (Zdroj: Technet.cz)

Tento typ chyb řeší softwarové inženýrství.

6.1. Programování řízené testy

Krom testování hotového programu můžeme zvolit jinou filozofii tvorby software. Test Driven Development – tedy Programování řízené testy. V rámci tohoto přístupu si nejdříve definujeme sadu testů a teprve poté píšeme samotný program, při jehož tvorbě se pouze soustředíme na to, aby kód prošel těmito testy. (Neřešíme například efektivitu kódu) Následuje refaktorace. Odstraňují se duplicity v kódu a kód se celkově upravuje do co možná nejpřijatelnější podoby. Nové spuštění testů zajistí, že v průběhu refaktorace nedošlo k narušení funkcionality kódu.

BlueJ k tomuto účelu nabízí sadu nástrojů. V programu BlueJ vytvoříme **testovací třídu** dané třídy tak že klikneme na třídu, která má být testována pravým tlačítkem a vybereme možnost vytvořit testovací třídu. Daná třída následně dostane neoddělitelného partnera – testovací třídu. Testovací třídu barevně odliší. Jak je vidno na obrázku.



Pokud chceme testy provádět se stále stejnou sadou objektů. Můžeme si vytvořit **Testovací přípravek**. Přípravek vytvoříme tak, že si do zásobníku objektů vytvoříme pouze ty objekty, u kterých chceme, aby byly obsaženy v testovacím přípravku. Nyní klikneme pravým tlačítkem myši na testovací třídu a zvolíme možnost **Dosavadní činnost –> testovací přípravek.** Pozor do přípravku se nahrají i všechny zprávy, které jsme poslali od posledního restartu virtuálního stroje. Proto pokud chceme mít jistotu, co všechno se nahrálo, restartujeme nejdříve virtuální stroj a teprve následně tvoříme objekty. Pokud chceme přepsat již uložený přípravek jinými objekty, jednoduše tyto objekty vytvoříme a zase zvolíme **Dosavadní činnost –> testovací přípravek.** Přípravek se přepíše.

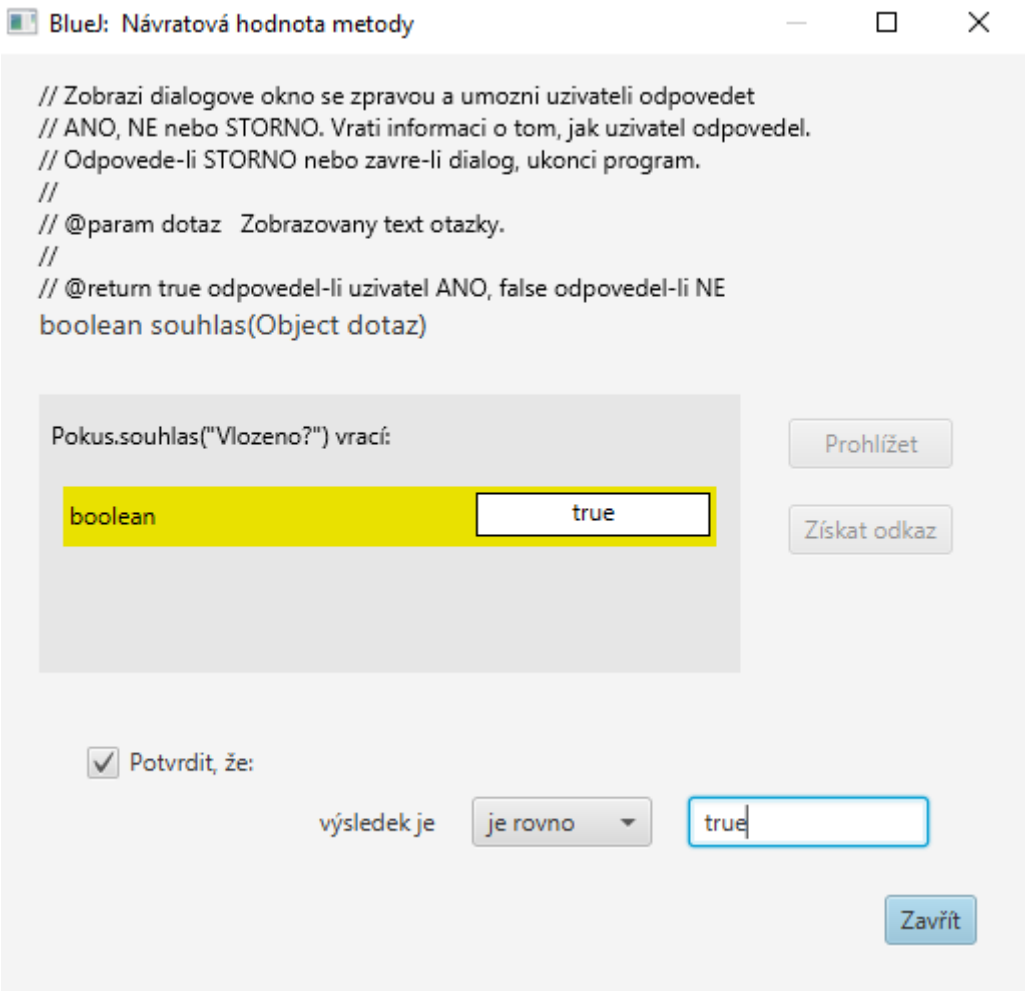
Případně můžeme ručně editovat danou testovací třídu.

Pro následné vyvolání objektů klineme pravým tlačítkem na testovací třídu a volíme **testovací přípravek –> Zásobník odkazů**.

6.2. Vytvoření testu

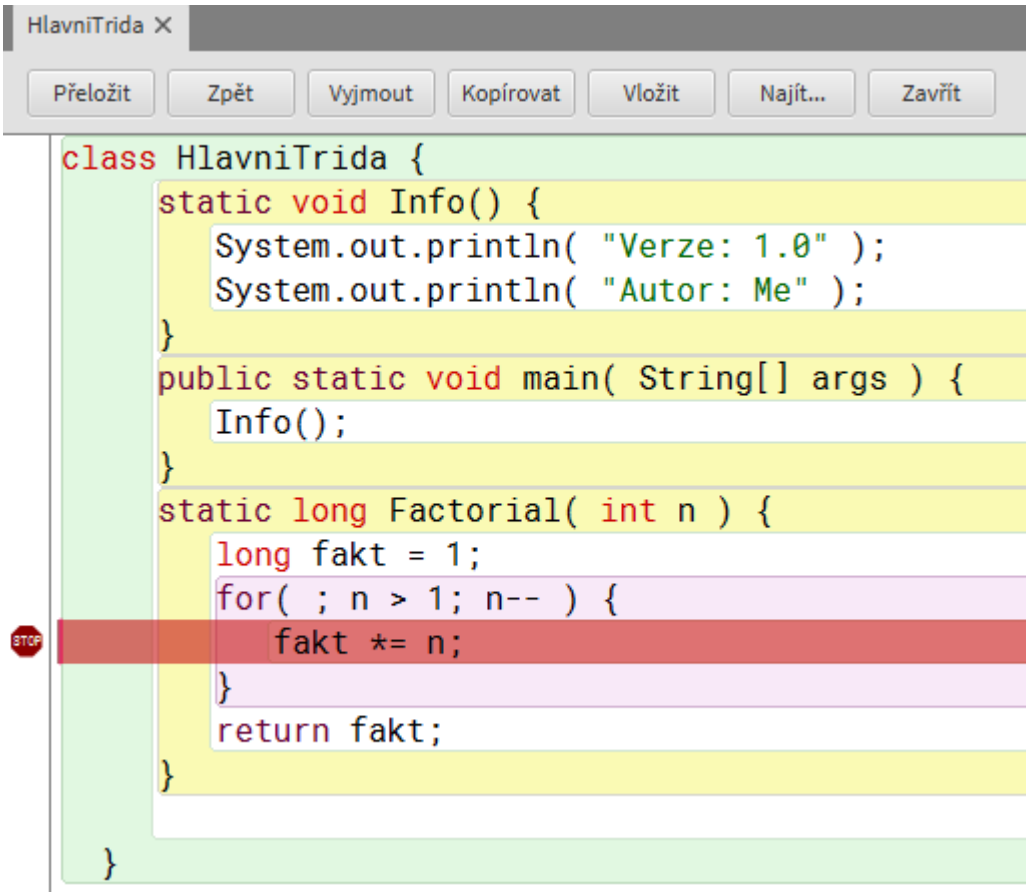
Ideálně nejdříve restartujeme virtuální stroj. Nyní klikneme pravým tlačítkem na testovací třídu a vybereme **Vytvořit testovací metodu.** Nejdříve zvolíme název samotné testovací metody**.** Nyní BlueJ nahrává náš postup, jak otestovat přípravek. Provedeme tedy požadované akce. V průběhu nahrávání se nás BlueJ bude ptát, zda u zvolených metod testovat návratovou hodnotu. Je vyobrazeno na obrázku níže. V průběhu nahrávání nám v levé části svítí červené světýlko. Pokud chceme ukončit nahrávání bez uložení, zvolíme: storno. Pro ukončení a uložení zvolíme ukončit. Obojí je hned pod červeným tlačítkem.

Pokud chceme právě vytvořený test vyzkoušet, klikneme pravým tlačítkem na testovací třídu, kde v menu vybereme test. Když v průběhu testu odpovíme jinak, než bylo nastaveno u testování návratové hodnoty. Test se přeruší. Otevře se okno s výsledky testů, kde se můžeme dozvědět, v jaké části kódu nastala chyba.



7. DEBUGGER

Je nástroj pomáhající k programátorovi odhalovat chyby v programu. Pro vyvolání debuggeru musíme nejdříve v kódu umístit zarážku. Což uděláme tak, že klikneme do levého sloupce v editoru kódu. Na příslušném řádku. Zobrazí se zde červená značka stop a řádek se zvýrazní červenou barvou. Jak je zobrazeno na obrázku níže. Při běhu programu se v místě zarážky běh programu zastaví a zobrazí se okno debuggeru.



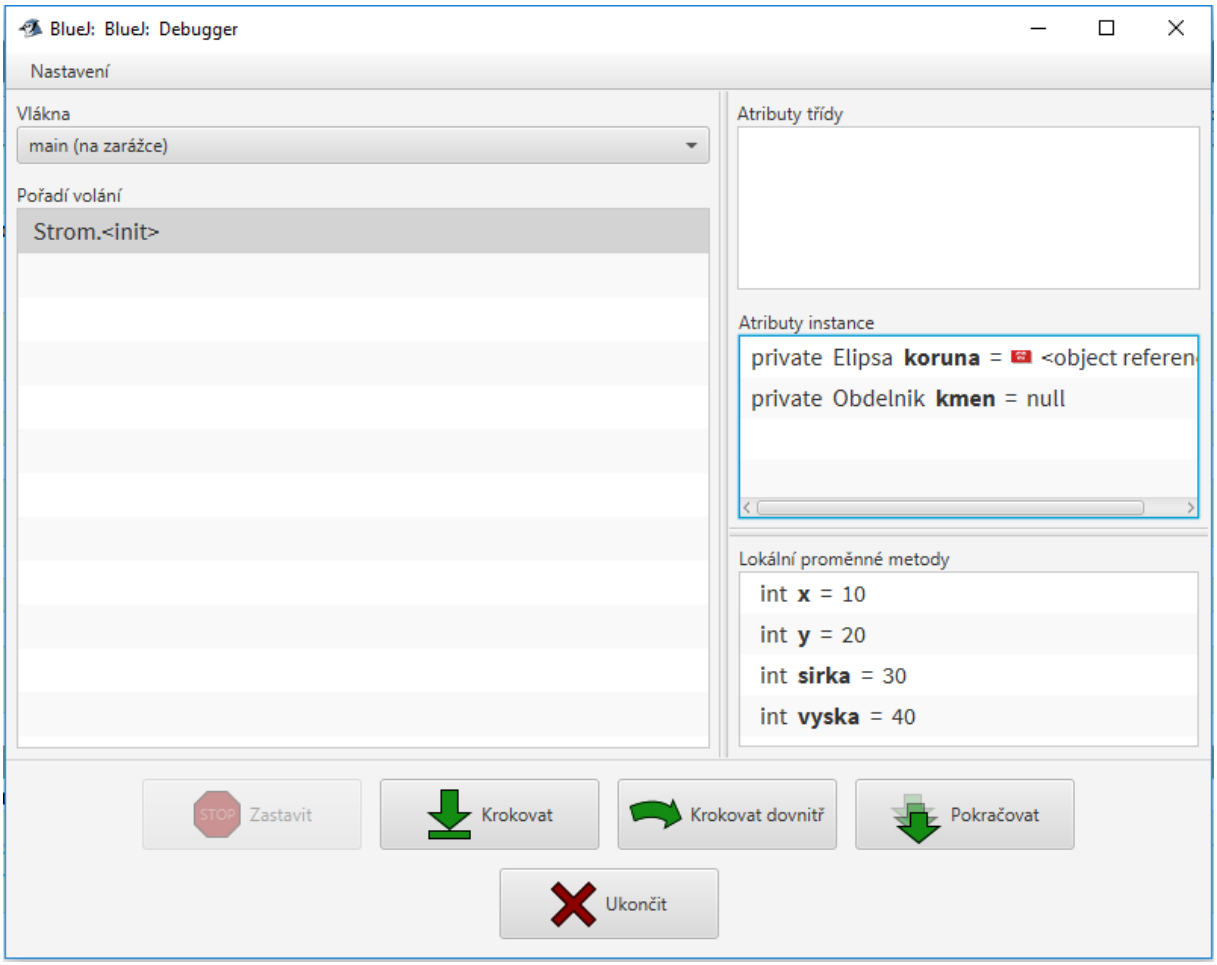
V dolní části okna debuggeru (obrázek níže) máme k dispozici celkem 5 tlačítek.

- **Zastavit** slouží k zastavení programu. Je například vhodné v případě, že se Vám zacyklí programu.
- **Krokovat** slouží k provedení dalšího kroku kódu. Příslušný krok je zároveň i graficky vyznačen zelenou barvou v editoru kódu.
- **Krokovat dovnitř** je dosti podobné, jako krokovat. Rozdíl je v tom, že v případě volání metody krokovat provede celou metodu, zatímco krokovat dovnitř, bude postupně krokovat volanou metodu.

- **Pokračovat** program bude normálně pokračovat až do svého konce, anebo do další zarážky**.**
- **Ukončit** ukončíme provádění programu. Případně program můžeme i ukončit restartováním virtuálního stroje.

V průběhu krokování můžeme také přidávat to kódu nové zarážky.

Samotné okno debuggeru je rozděleno do několika částí. V levo nahoře si můžeme volit konkrétní **vlákno**. Pod volbou vlákna je oblast věnující se **Pořadí volání**. Tato oblast by se dala nazvat, jako zásobník návratových adres. Vypisují se zde všechna volání, již realizovaná při běhu programu. Nová volání se zařazují na nejvyšší pozici. Klepneme-li na konkrétní volání, otevře se editor kódu s třídou, která obsahuje volanou metodu. Barevně se zvýrazní právě vykonávaný řádek.



V pravé části okna debuggeru se vypisují **Atributy třídy**. Jde o atributy třídy, které patří zrovna vybraná složka v Pořadí volání.

Níže jsou **Atributy instance**. Zobrazují se zde atributy instance, které náleží zrovna

položce v Pořadí volání. Pokud se podíváme blíže do atributů instancí v našem příkladu, zjistíme že, tato instance obsahuje dva atributy. Atribut koruna obsahuje odkaz na objekt. Pokud si na ni poklepem, zobrazí se nám okno s hodnotami všech atributů primitivních typů dané instance. Atribut kmen v daný moment ještě nemá odkaz nikam. (indikuje to hodnota null)

Pokud Ještě níže jsou zobrazeny **Lokální proměnné** také náležející právě vybrané položce v Pořadí volání. Debugger zde zobrazuje pouze proměnné, které již mají vyhrazenu paměť a přiřazenou počáteční hodnotu, tedy jsou definované.

8. VÝJIMKY

Až do této chvíle jsme předpokládali, že v momentě, kdy dojde nějaké nepředvídané události celý program „spadne“. Tomuto problému se dá ale předejít pomocí výjimek. Při takových situacích dojde k vyvolání výjimky, která okamžitě přeruší běh programu a přechod vlákna do místa, kde je situace ošetřena. Pokud takové místo není, vlákno je ukončeno a protože zatím používáme pouze jedno vlákno, dojde k pádu celé aplikace. Existuje několik druhů výjimek, které se dají rozdělit podle příčiny, která neočekávanou situaci zavinila:

- Error – kritická chyba způsobená například nedostatkem zdrojů pro práci virtuálního stroje -*OutOfMemoryError*, přetečení zásobníku - *StackOverflowErrora* podobně. Tyto výjimky se zpravidla neošetřují.
- RuntimeException – často vznikají chybou programátora, (třeba dělení nulou - *ArithmeticException*, neplatný index - *ArrayIndexOutOf BoundsException*). Pro vyvolání této podmínky nemusíme v hlavičce metody deklarovat možnost jejich vyvolávání.
- Exception uživatele (místo telefonního čísla zadává písmena), nebo neexistující soubor, nebo soubor jiného typu. Na takovéto typy výjimek můžeme často velmi jednoduše zareagovat. Například nechat vybrat uživatele soubor ještě jednou. U takovýchto výjimek musíme vždy uvést klíčové slovo throws a seznam tříd volaných výjimek.

8.1. Chráněný režim

Potenciálně problémovou část kódu můžeme umístit do „chráněného režimu pomocí try a složených závorek. Tento režim je o trošku pomalejší. Pokud u něj ale dojde k chybě - výjimce, vykoná se část obsažená v catch. Volitelný blok finally pak bude vykonán vždy. Finally se často využívá u výjimek na uklízecí práce, kdy se vracíme do stavu před výjimkou, například zavírání souborů, uvolňování paměti a podobně.

```
public static void exception () {
      try {
            int a = 5 / 0; //create exception
      }
      catch (Exception e)
      {
            System.out.println("An exception was taken ");
      }
      finally
      {
            System.out.println("The contents of the block
            will always be processed.");
      }
}
```

8.2. Throw

Případně můžeme přidat výjimku výše pomocí klíčového slova throws. Což se týká pouze kontrolovaných výjimek, v případě že chce ošetření výjimek předat do volající metody.

Jako je uvedeno na příkladu:

```
if (index == null) {
      throw new NullPointerException();
}
```

8.3. Throws

Pokud tvoříme metodu, u které může vzniknout výjimka, kterou neumíme, anebo nechceme ošetřit. Explicitně překladači sdělíme, že tuto výjimku předáváme k ošetření do nadřazené úrovně pomocí klíčového slova throws + třídy výjimek

```
public static void  read ( ) throws IOException {
      ...
}
```

9. PRÁCE SE SOUBORY

Veškeré informace, data, či objekty uložené v paměti se v případě vypnutí programu smažou. Pro jejich zachování a opětovné načtení je musíme uložit do nějakého souboru.

Pro bezproblémový zápis dat do soboru je vhodné ukládat data aplikace do složky appdata. Složka AppData či Application Data, případně Data aplikací obsahuje data vytvářená programy. V této složce si vytváří svou vlastní složku prakticky každý program, který je do počítače nainstalován, a do ní si následně ukládá různé údaje. Do této složky se nejjednodušeji dostanete, napíšete-li do průzkumníku souborů **%appdata%** ideální je vkládat soubory do podsložky roaming, data v této složce by měla následovat uživatele po různých počítačích v rámci domény.

Balík java.io obsahuje třídu File, která nám poskytne všechny důležité nástroje pro práci se soubory. Mnoho metod z třídy File vyžaduje jako svůj argument název souboru. Jako argument lze použít textový řetězec String, anebo instanci třídy File. Což je často optimálnější řešení, díky kterému můžeme předem o souboru zjistit nějaké informace, či s ním provést nějakou operaci.

Objekt file můžeme vytvořit několika způsoby

- Názvem souboru – vytváříme z absolutní anebo relativní cesty, která se převede na abstraktní cestu.
- Názvem souboru vzhledem k rodiči - abstraktní cesta bude vytvořena jako relativní vůči rodičovské cestě
- URI (Uniform Resource Identifier) – musí být splněny jisté požadavky. Např. cesta nesmí být prázdná

Samotný soubor lze například vytvořit takovýmto způsobem pomocí URI.

```
import java.io.File;
import java.io.IOException;
...
public void createFile() throws IOException{
      File soubor = new File("C:\\Temp\\hello.txt");
      soubor.createNewFile();
}
```

Souborový systém může implementovat omezení určitých operací na aktuálním objektu systému souborů, jako je čtení, psaní a spouštění, které nazýváme přístupová oprávnění.

Instance třídy File jsou neměnné; to znamená, že jakmile bude instance vytvořena, abstraktní cesta, kterou představuje objekt File, se nikdy nezmění.

Třída file nabízí různé metody určené k práci se soubory. Můžeme například pracovat s:

- **Cesta k souboru** - objekt File je abstraktní cestou k souboru. S touto cestou můžeme různě pracovat. Návratové metody vracející cestu k souboru mají obvykle návratovou hodnotu typu textový řetězec Například pomocí:
    - getPath() dostaneme abstraktní cestu k souboru
    - getName() zjistíme název souboru
- **Informace o souboru** – Existuje několik metod, které nám vrací informace o souboru, jako například: length(); canRead() nebo například lastModified()
- **Informace o adresářích** – Máme k dispozici několik metod týkajících se pouze adresářů, jako například list() - vracející seznam všech souborů v adresáři, nebo unixová listRoots() Vracející pole všech kořenů adresářových stromů
- **Práce se soubory** – k samotné práci se soubory máme k dispozici různé metody:
    - renameTo(File k) jako parametr zadává další instance objektu File.
    - delete()
    - mkdir() tvorba adresáře
    - setReadOnly()

10. GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ (GUI)

Je grafické prostředí, se kterým se běžný uživatel setkává a pracuje. Jednotlivé komponenty tohoto prostředí všichni velmi dobře známe. Patří mezi ně například tlačítka, roletky, posuvníky a podobně. V Javě jsou k dispozici rozdílné grafické knihovny jako například AWT (Abstract Windowing Toolkit), JFC (Java Foundation Classes). Pro BlueJ je možné nainstalovat Simple GUI Designer Extension. Které nám může zjednodušit tvorbu GUI. V tomto textu bude popsána práce s JFC, též známá jako Swing.

10.1. První aplikace

Při tvorbě aplikace s grafickým uživatelským prostředím, ve kterém aplikace poběží. Musíme nejdříve vytvořit „okno“ (rámec). Použijeme třídu JFrame. Možností jak vytvořit okno je ale vícero. Vytvořili jsme třídu KonstrukceGui, která dědí ze třídy JFrame. V této třídě jsme vytvořili bezparametrický konstruktor. Do třídy jsme vložili tlačítko a label.

Další běžně využívanou komponentou, kterou jsme ale v tomto příkladu nepoužili je pole pro zápis (JTextField). Deklarovalo by se takto:

```
JTextField someName = new JTextField(”Text”, 6)
```

V parametrech se zadává text, který se bude zobrazovat v poli. Dále je, pak číslo, které udává kolik znaků se de pole má vejít.

Rozložení komponent v okně jsme nastavili pomocí FlowLayout. Kvůli FlowLayout bylo třeba naimportovat i java.awt.

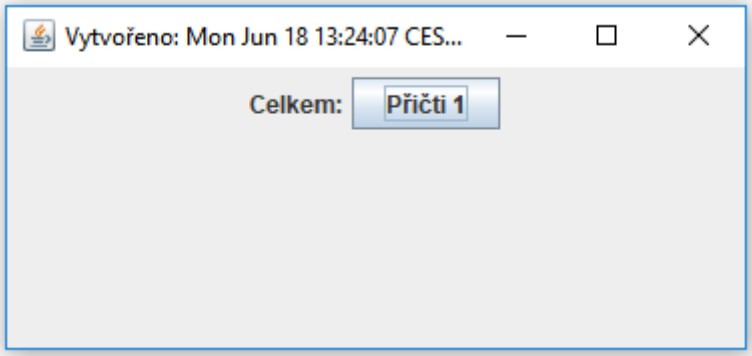
Jednotlivé komponenty bylo třeba do okna přidat. Což jsme udělali pomocí metody **add(),** u které jako parametr zadáváme název přidávaného objektu.

```
import java.awt.*;
import javax.swing.*;

public class ConstructionGui extends JFrame {
      private JButton button;
      private JLabel value;

public ConstructionGui()
{
      FlowLayout layout = new FlowLayout();
      setLayout(layout);
      value = new JLabel(”Total:”);
      add(value);
      button = new JButton(”Add 1”);
      add(button);
      }
}
```

Následně jsme vytvořili další třídu s názvem NavodGUI. S metodou main. V metodě main jsme vytvořili objekt ze třídy KonstrukceGui. Na tomto objektu jsme zavolali několik základních metod. Knihovnu date jsme importovali, abychom mohli použít funkci date(), která nám vypíše dnešní datum do titulku okna.

```
import java.awt.*;
import javax.swing.*;

public class ConstructionGui extends JFrame {
      private JButton button;
      private JLabel value;

public ConstructionGui()
{
      FlowLayout layout = new FlowLayout();
      setLayout(layout);
      value = new JLabel(”Total”);
      add(value);
      button = new JButton(”Add 1”);
      add(button);
}}
```

Samotný výsledek pak vypadá nějak takto:



Jednotlivé komponenty jsou vycentrované. Pokud budeme zmenšovat okno, budou se řadit pod sebe.

11. UDÁLOSTI

V současnosti již máme vytvořeno jednoduché okno s jedním labelem. Po máčknutí tlačítka se ale nic nestane. Využijeme proto události, abychom přidali našemu oknu funkcionalitu. Princip událostí je takový, že v GUI vyvoláme například na máčknutí na tlačítko nějakou událost. Na objektu, kterým jsme vytvořili událost je posluchač události (event listener) Na podnět posluchače události je vyvolána metoda, která něco udělá. V našem případě vytvoříme z Labelu a tlačítka čítač máčknutí tlačítka. Kompletní kód je na další straně.

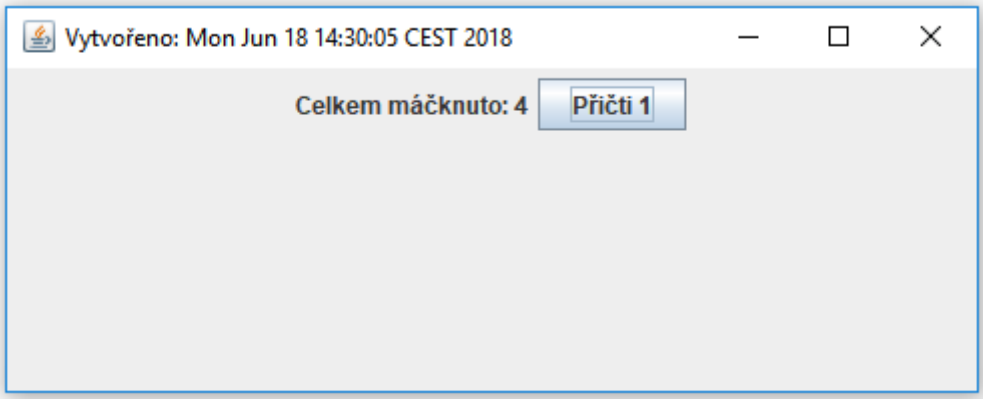
Do třídy KonstrukceGUI jsme si přidali další třídu s názvem UdálostPocet. Díky čemuž má přístup ke komponentám třídy KonstrukceGUI. Do třídy událostiPocet jsme implementovali ActionListener. Je nutné importovat java.awt.event.*;

Třída ActionListener obsahuje pouze jednu návratovou metodu actionPerformed(ActionEvent e) do které definujeme, co se má stát při vyvolání události. V našem případě jde o počítání zmačknutí tlačítka.

V konstruktoru jsme vytvořili posluchače události – objekt s názvem pocet. Tlačítku jsme přiřadili posluchače s parametrem pocet.

```
EventCost MyCount = new EventCost();
MyButton.addActionListener(MyCount);
```

Výsledek pak vypadá takto:



```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ConstructionGui extends JFrame {
      private JButton MyButton;
      private JLabel MyValue;
      int MyNumber = 0;

      public ConstructionGui()
      {
            FlowLayout layout = new FlowLayout();
            setLayout(layout);
            MyValue = new JLabel("Not squeezed ");
            add(MyValue);
            MyButton = new JButton("Add 1");
            add(MyButton);
            EventCost MyCount = new EventCost();
            MyButton.addActionListener(MyCount);
      }

      public class EventCost implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                  MyNumber = MyNumber + 1;
                  MyValue.setText("Totally    squeezed:    "    +
            MyNumber);
            }
      }
}
```