Lekce 2
(Teoreticko-praktická část)

Datové typy

Datové typy

- Php dokumentace: https://www.php.net/manual/en/language.types.intro.php
- C# dokumentace: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table

- = hodnoty s předem domluvenými pravidly chování, typizované hodnoty
- Datové typy se odlišují mezi programovacími jazyky, ale základní typy (například string nebo integer) bývají napříč jazyky shodné.

- Podstata datového typu:
    - V pamětích počítačů se vše ukládá jako číslo (bity, byty…).
    - Datové typy jsou určitými abstrakcemi nad touto základní vrstvou pro určení základních pravidel pro práci s danou ***hodnotou***. Například o integeru vím, že jej mohu sčítat a odčítat, zatímco u logického typu boolean taková operace nedává smysl. U stringu mohu zjišťovat jeho délku, zatímco u celočíselné hodnoty (integer) takový údaj nedává smysl.

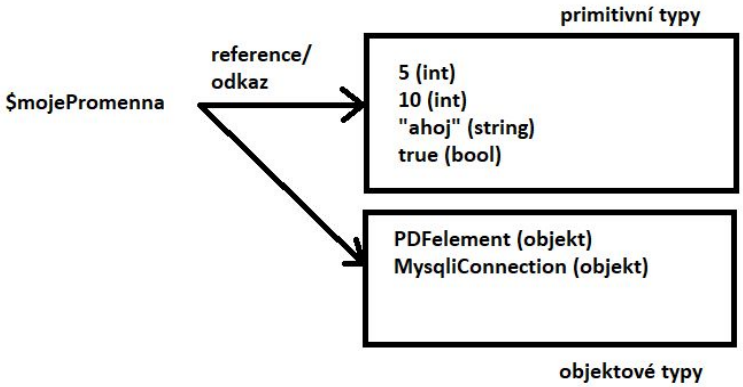
- Základní datové typy:
    - int (integer)
    - char, string
    - float
    - bool (boolean)
    - array
    - object
        - ⇒ datový typ objekt má výsadní postavení, protože je základním datovým typem pro veškeré další odvozování (dědění). Jakýkoli jiný objekt je v důsledku tímto datovým typem. Objekty se zpravidla “předávají referencí” (viz dále)

Proměnné, reference

- Proměnné jsou speciálním prvkem v rámci kódu programovacího jazyka, který má uživatelem stanovené jméno (indentifikátor/indentifier) a **odkazuje na hodnotu** či **je odkaz prázdný**
- Proměnné jsou každou syntaxí programovacích jazyků vyjádřené trochu odlišným způsobem. V jazyce PHP je obecně proměnnou vše uvozené znakem dolaru “$” (řekněme s výjimkou “$this”, což lze považovat za speciální identifikátor).
- **Proměnná je nejprve deklarována** (říká o sobě, že se s ní má vůbec počítat, případně k jakému datovému typu se bude odkazovat) a následně jí může být *přidělována nějaká hodnota*. Většina jazyků umožňuje oba kroky spojit v jeden.
    - PHP:
        - Deklarace: $mojePromenna;
        - Přiřazení hodnoty: $mojePromenna = 5;
        - Spojený zápis: $mojePromenna = 5;
    - C#
        - Deklarace: int mojePromenna;
        - Hodnota: mojePromenna = 5;
        - Spojené: int mojePromenna = 5;
    - PL/SQL:
        - Deklarace:
          Declare
          mojePromenna int;
        - Hodnota: mojePromenna := 5;
        - Spojené: neexistuje

Proměnné a reference

- Často se uvádí/říká, že proměnná “má nějakou hodnotu” nebo dokonce např: “$x je tento objekt”. Tato formulace je zavádějící a pramenní z ní některá častá zmatení.
- Správnější by bylo říkat, že proměnná **odkazuje** na hodnotu.



- Jak v php, tak v C# (a nejspíše dalších jazycích) je dobré rozlišovat mezi odkazy na řekněme “primitivní datové typy” a odkazy na “objekty”. Podle tohoto druhu hodnoty se totiž výchozím způsobem **předává (jiné proměnné či do nějaké funkce/metody) buď hodnota a nebo reference**.
- Význam tohoto bude zjevnější později, ale jde především o to, že předám-li někam (někam mimo mou kontrolu a můj zájem) hodnotu, pak ať se dále děje cokoli, má proměnná bude mít stále hodnotu nezměněnou. Předávám-li referenci, pak zůstává neměnná tato reference, ale objekt, na který odkazem odkazuji se již může proměňovat.
    - Snaha o analogii:
        - Předávání hodnoty:
            - Na kusu papíru mám napsané **číslo** 10. Kamarád se mě zeptá, jaké číslo mám a já mu ho sdělím (= předám). Kamarád pak s tímto číslem může udělat cokoliv (třeba o něj něco odečíst), ale tím se nijak nemění hodnota zapsaná na mém papírku.
        - Předávání reference:
            - Na kusu papírku mám napsanou **adresu** mého bytu. Kamarád se mě zeptá, jakou adresu mám napsanou a já mu ji sdělím. Kamarád s adresou může udělat cokoli (třeba tam zajít a byt vymalovat na růžovo), čímž se sice nijak nezmění adresa (hodnota) zapsaná na mém papírku, ale objekt, na který adresa odkazovala (byt), již ano.
        - ⇒ v případě příkladu s papírkem je nemyslitelné, že bychom adresu na papírku ztotožňovali s bytem samotným. V programování se však tohoto omylu můžeme lehko dopustit. $mujByt = new Home(address); Proměnnou $mujByt předám nějaké funkci a pokračuji dále a následně se divím, že $mujByt->stěna == “růžová”, když původně byla “bílá” a já s ní nic nedělal.
- Některé jazyky (C++) explicitně rozlišují mezi proměnnou, referencí a hodnotou a pracují s tzv. “Pointery” (ukazatel). Metodě následně předávám pointer a mělo by to být jasnější.

PRO JEDNODUCHOST DÁLE BUDEME NEPŘESNĚ ŘÍKAT, ŽE PROMĚNNÁ "MÁ" NĚJAKOU HODNOTU, NEBO ŽE JSME JÍ HODNOTU "PŘIŘADILI".

Imperativní programovací jazyky a změny hodnot proměnných

- Mohlo by být matoucí, když v programu říkáme, že $A se rovná 5 a následně řekneme, že $A se rovná 10, neboť to zní jako rozpor ⇒ čemu se tedy rovná $A?.
    - ⇒ o hodnotách proměnných musíme v imperativních programovacích jazycích uvažovat jako o odpovědi na otázku: "Na jakou hodnotu proměnná odkazuje **v danou chvíli, v daném kroku vyhodnocování kódu "krokovátkem"**".
        - ⇒ Platí tedy obě tvrzení. Např.: Při exekuci řádku 33 proměnná $A odkazuje na hodnotu 5 a od řádku 61 odkazuje na hodnotu 10.

Přiřazení hodnoty proměnné použitím jiné proměnné (#1 - primitivní typy)

- Při programování velice často přiřazujeme hodnotu nějaké proměnné použitím jiné proměnné. Je velice důležité si u toho zapamatovat, že:
    - **Proměnné se neřetězí, nejsou spolu svázané!**
- Příklad:

```
$a = 5;
$b = $a;
$a = 10;
// Kolik se teď rovná $b??
```

- Odpověď: $b se rovná 5.
- Vysvětlení (mohli bychom nesprávně očekávat, že se $b bude rovnat 10):
    - Ve chvíli, kdy provádíme příkaz $b = $a říkáme něco jako: "$b nyní odkazuje na stejnou hodnotu, na kterou **v tuto chvíli** odkazuje $a a od této chvíle mně již z pohledu $b proměnná $a vůbec nezajímá."
    - Proměnná $a nám tedy slouží jako pouhý prostředek k tomu, abychom se dobrali správné hodnoty v danou chvíli, ale samotná proměnná nás nezajímá. $b je na $a nezávislé a při zjišťování jeho hodnoty se na $a již dále nebudeme odkazovat.

Přiřazení hodnoty proměnné použitím jiné proměnné (#2 - reference na objekty)

- To stejné platí pro případ odkazu proměnných na objekty. Protože však máme tendenci proměnnou s objektem ztotožňovat, může to být o to více matoucí.
    - Příklad:

```
$a = muj_byt_s_bílými_stenami; // muj_byt je příklad objektu
$b = $a;
$a->zmalujStěny(červená); // stěny objektu se obarví na červeno
// 1a) Na co v tuto chvíli odkazuje $b?
// 1b) Pokud na objekt typu byt, pak jakou barvu mají stěny toho bytu?
```

```
$a = jiny_byt_s_modrymi_stenami;
// 2a) Na co v tuto chvíli odkazuje $b?
// 2b) Pokud na objekt typu byt, pak jakou barvu mají stěny toho bytu?
```

- Odpovědi:

```
1a) $b odkazuje na stejný objekt-byt jako $a
1b) Byt, na který odkazuje $b má nyní červené stěny.
2a) $b odkazuje stále na původní byt
2b) Byt, na který odkazuje $b má stále červené stěny.
```

- Vysvětlení
  - Ve chvíli příkazu $b = $a platí stejná věta jako výše. $b začne odkazovat na stejný objekt (byt) jako $a v **danou chvíli** a dále jej již proměnná $a vůbec nezajímá.
  - Přesto má následně byt, na který odkazuje $b červené stěny. Proč? **Protože příkazem zmalujStěny() nebarvíme proměnnou $a, ale onen byt.** Proměnná nemá žádné stěny, to je hloupost. Pokud se tedy zbarvily stěny daného bytu, pak je to ten stejný byt, na který odkazuje $b a tedy i stěny bytu, na který odkazuje $b, budou červené. **$a není tím bytem. $a pouze odkazuje na ten byt**.

Dobrá praxe: smysluplné názvy proměnných, zachování datových typů, žádná recyklace

- Proměnné je dobré pojmenovávat v souladu s tím, jaké hodnoty mají uchovávat. Například pokud chci počítat získané body v nějaké hře, pak se proměnná může jmenovat třeba $ziskaneBody; Proměnnou je sice možné nazvat třeba taky $x, ale pro programátora se následně stává kód těžko srozumitelným.
- Dále je dobré vždy do dané proměnné ukládat po celou dobu její existence stále ten stejný datový typ. V tzv. staticky typovaných jazycích jako je C# není ani možné udělat nic jiného, neboť je proměnná od počátku deklarována jako mající konkrétní datový typ. Avšak třeba v PHP nejsme ničím limitováni, tedy nejprve může být $x nějaké číslo a následně se může stát objektem a nebo logickou hodnotou true/false.
- To nepřímo souvisí s pojmem “recyklace proměnných”. To znamená jednoduše to, že stejnou proměnnou použiji dvakrát k jinému sémantickému učelu. Např. ji nazvu $pocet a nejprve do ní budu ukládat počet získaných bodů a následně počet hráčů a nakonec třeba zprávu “Game over”.
- ⇒ těmito pravidly se řídit nemusíme, nedojde k erroru, programovací jazyky nám tyto věci umožňují, ale obecně je radno se těmito pravidly řídit. Opak vede ke zmatkům a chybám.

Logické hodnoty true/false (datový typ: bool)

Logicka hodnota true/false a jednoduché logické výrazy

- Proměnná může mít hodnotu true/false, tzv. “logickou hodnotu”, či “boolean value” nebo prostě “bool”.
- Hodnota může být jednoduše stanovena (jako například: $a = true;) a nebo (což je běžnější) je důsledkem vyhodnocení nějakého *logického výrazu.*
- Logický výraz je možné chápat jako otázku, na kterou může být odpověď pouze a jenom: pravda či nepravda (zcela jasně vyhodnocená otázka, žádné “možná” nebo “pravděpodobně”).
- Příklady logických výrazů:
    - Početní příklady: 5 > 2 (= true), 5 < 2 (= false), 1 > 1 (= false), 1 == 1 (= true)
    - Řetězcové (string) příklady: “Ahoj” == “Ahoj” (= true), “ahoj” == “čau” (= false)
    - Logické příklady: true (= true), false (=false), true == false (= false), false == false (= true)
        - Pravda je vždy pravda, nepravda je vždy nepravda.
        - Tvrzení, že se pravda rovná pravdě je pravda.
        - Tvrzení, že se pravda rovná nepravdě není pravda.
    - Příklady logických metod/funkcí: $čr->jeHlavnímMěstem(“Ústí nad Labem”) (= false)

<u>Logické operátory AND a OR</u>

- Základem spojování logických výrazů jsou *operátory* AND (= a zároveň) a OR (= nebo)
- Logické A ZÁROVEŇ je celkem jasné: všechny skutečnosti musí platit zároveň, jinak se nejedná o pravdu.
    - Např.: “Petr má na sobě zelené tričko A ZÁROVEŇ zelenou čepici.” Pokud má na sobě Petr pouze zelené tričko nebo jen zelenou čepici a nebo dokonce vůbec nic zeleného, pak tvrzení není pravdivé.
- **Logické NEBO (OR) není VYLUČOVACÍM NEBO (označuje se jako XOR)**
    - To může být matoucí, protože v češtině máme většinou na mysli vylučovací NEBO, zatímco v programování se běžně používá jednoduché NEBO:
    - Vylučovací nebo:
        - Příklad 1 (VYLUČOVACÍ NEBO v češtině): “Petr pojede do Brna nebo do Prahy”, pak se jistě myslí “vylučující nebo”, protože Petr nemůže být na dvou místech zároveň. Musí si tedy vybrat: **buď** pojede do Brna **a nebo** pojede do Prahy.
        - Příklad 2 (VYLUČOVACÍ NEBO v češtině): “Vezmi si hrušku nebo jablko.” Očekává se, že si vyberu **buď** hrušku **a nebo** jablko. Rozhodně se neočekává, že bych si vzal obojí.
        - ⇒ platí, že **Právě jeden z členů musí být pravdivý. Ne méně, ne více.**
    - Logické NEBO funguje odlišně. Jedná se o pravdu, když **ALESPOŇ jeden z členů tvrzení je pravdivý. Pravdivých jich ale může být více.**
        - Na výzvu: “Vezmi si hrušku nebo jablko” bych mohl správně reagovat tak, že si vezmu hrušku a také jablko.

    - Oba dva druhy NEBO mají stejnou vlastnost: minimálně jeden z členů musí být pravdivý, aby se mohlo jednat o pravdu. VYLUČOVACÍ NEBO však umožňuje nejvýše jeden pravdivý člen.
- Logické AND se v php zapisuje znakem: &&. Logické OR se v php zapisuje: ||

<u>Složené logické výrazy a jejich vyhodnocování</u>

- Pomocí výše zmíněných logických operátorů můžeme logické výrazy skládat.
- Příklad:
    - (5 == 5) && (1==1) => to je pravda
    - (5==5) && (1==2) => to není pravda
    - (1==2) || (2==2) => to je pravda (jeden z členů je pravdivý)
    - (1==1) || (2==2) => to je také pravda (pravdivost členů se nevylučuje)
- Závorky není nutné psát, ale jedná se o dobrou praxi, neboť je pak jasné, co se vyhodnocuje dříve, než něco dalšího. Všechny logické celky je dobré do závorek uzavírat.
    - Např. v případě: 1==1 AND 1==2 OR 2==2 je situace dost nejasná. Daleko lepší bude zapsat: (1==1) AND ((1==2) OR (2==2)) a případně ještě lepší může být oddělení vizuální pomocí řádkování, třeba takto::

```
(
        (1==1) AND
        (
                (1==2) OR (2==2) OR (5 > 1000)
        )
)
```

- **Vyhodnocujeme (resp. program vyhodnocuje) vždy od nejzanořenějšího členu směrem nahoru k základní úrovni.**
    - 1) Nejzanořenější členy jsou zde:
        - (1==2) => to není pravda
        - (2==2) => to je pravda
        - (5 > 1000) => to není pravda
        - Situaci můžeme tedy přepsat takto:
        -

```
(
        (1==1) AND
        (
                (false) OR (true) OR (false)
        )
)
```

    - 2) Tyto členy jsou ve společné závorce a spolu svázány operátorem OR. Alespoň jeden člen tedy musí být pravdivý, aby celá závorka byla pravdivá. To splňujeme, tedy situaci můžeme přepsat takto:
        -

```
(
        (1==1) AND
```

(true)
)

- 3) Další člen je (1==1), co že je samozřejmě true, tedy máme:
- (
(true) AND
(true)
)
- 4) Máme už jen jednu závorku a oba členy jsou spojené vztahem AND a oba jsou pravdivé, tady je výsledek true. Získáváme tedy jednoduše:
    - (true), tedy prostě: true

- Takto provázaným logickým celkům se říká “logické stromy”.

Podmínka IF (či jiné), větvené zpracování kódu

- Možnost větvení zpracování kódu je jednou ze základních podmínek turingovsky kompletního programovacího (imperativního) jazyka (viz dříve).

- Kód je možné větvit právě na základě nějakého vyhodnocení logického výrazu (viz výše). Při zpracování kódu se stroj ptá, zda je něco pravda a podle toho začne dělat buď něco a nebo něco jiného.
- Nejzákladnějším konstruktem pro větvení zpracování kódu je konstrukt IF, resp. IF-ELSEIF-ELSE konstrukt. (části ELSEIF a ELSE nejsou nutnou součástí konstruktu, lze je vynechat),

- Podmínkové konstrukty lze do sebe lze vnořovat (jako většinu věcí v programování)

- Příklad:
    - ```
      $mámHlad = true;
      $mámŽízeň = false;
      If ($mámHlad == true || $mámŽízeň == true) {
          // sníst více jídla nebo vypít více pití
          If ($mámHlad == true) {
              $já->snístJídlo();
          }

          If ($mámŽízeň == true) {
              $já->pítPití();
          }
      } else {
          // zaplatit a odejít
      }
      ```
- Podmínky se vyhodnocují **ze základní úrovně k té nejvíce vnořené**. Neplést s vyhodnocením logických výrazů, kde je tomu naopak.
- Konstrukt IF se v závorce ptá, zda je něco pravda (viz logické výrazy výše) a pokud ano, tak začne vykonávat následný blok kódu uzavřený ve složených závorkách. Pokud nikoli, pak se podívá na další případné podmínky (ELSEIF), kde se zachová stejně. Takto může dojít až k ELSE, které rovnou vykoná a nebo pokud není přítomné, tak jde dále.
- **Pozor: nižší podmínka ELSEIF nebo také ELSE se vyhodnocuje pouze když nedošlo na zpracování vyšších částí. V opačném případě se vyhodnocení přeskočí.**
    - Podobný příklad jako výše:
      ```
      $můjHlad = true;
      $mojeŽízeň = true;
      ```
      ```

```
If ($můjHlad == true) {
        // sníst jídlo
}
elseif ($mojeŽízeň == true) {
        // vypít pití
}
```

- ⇒ přestože mám hlad a také žízeň, tak by došlo pouze na jezení jídla a následně je konstrukt opuštěn!

- Konstrukt IF není jediným, který takto podobně pracuje s podmínkami, ale je tím obecně nejpoužívanějším, a tedy nás nyní nejvíce zajímá. V jistém smyslu podobným konstruktem může být například cyklus while(podmínka { kód; } a nebo v SQL dotazování konstrukt CASE WHEN podmínka THEN výraz ELSE výraz END

Cykly, cyklus FOR a cyklus WHILE

- Cykly mají velmi blízko k větvenému provádění kódu, čehož bylo příkladem výše uvedené větvení konstruktem IF. Podstatou je také zde vyhodnocení nějaké podmínky a provedení určitého bloku kódu v případě jejího splnění. Odlišnost je zde pouze v tom, že v případě konstruktu IF se již k provedení stejného kódu nemáme možnost vrátit (postupujeme stále shora-dolů).
- Nemožnost vrácení zpracovávajícího “krokovátka” by byla omezením, které by daný programovací jazyk vyloučila ze sféry “turingovsky kompletního” jazyka.
- Analogický příklad použití #1:
    - Od trenéra dostanu pokyn, že mám provést 20 dřepů s výskokem. Provedu dřep s výskokem a v mysli si podržím číslo 1. Provedu další dřep s výskokem a pamatuji si číslo 2. Takto pokračuji stále dokola dokud číslo není požadovaných 20. V této chvíli skončím s dřepy a skákáním a pokračuji dle dalších pokynů.
- Analogický příklad použití #2:
    - Máme například polévku, která není osolená. My ji chceme osolit, ale zároveň ji nechceme přesolit. Existuje tedy jistá míra slanosti, které bychom chtěli dosáhnout. Dále máme slánku, která do polévky vždy vysype malé množství soli. Nakonec jsme tu my, kdo polévku můžeme ochutnávat.
        - ⇒ To, co budeme provádět, je shodné s tím, jak fungují cykly.
        - ⇒ Ochutnám polévku. Není dost slaná. Přidám sůl. Ochutnám polévku. Není dost slaná. Přidám sůl. Ochutnám polévku. Není dost slaná. Přidám sůl. Ochutnám polévku. Polévka je nyní dost slaná! Již žádnou sůl nepřidávám a pokračuji v dalších činnostech.
- ⇒ oba příklady jsou také obecné: Dřepů mám někdy vykonat 20, někdy 100. Můj postup se nebude nijak odlišovat, pouze provedu odlišný počet opakování. Polévka někdy může být z 1 litru vody, jindy ze 3.

- Případy výše lze vyjádřit zapsáním kódu. Řešit je budeme cykly FOR nebo WHILE. Oba cykly jsou záměnné, oba lze použít ve všech případech. Každý z nich je však zamýšlen pro odlišné použití, tedy sémanticky psaný kód využije správného cyklu dle situace. FOR v případě, kde potřebujeme počítat opakování či změnu nějaké hodnoty. WHILE v případě, kdy chceme kus kódu opakovat tak dlouho, dokud je pravdivá nějaká podmínka.

- Případ prvního typu (dřepy s výskokem) se nejčastěji řeší zápisem cyklu typu FOR. Jedinou nutnou podmínku zápisu cyklu FOR je sice podmínka a blok kódu pro vykonání, ale z 90% se setkáváme s plným zápisem a to většinou konkrétně v této podobě:
    - for($i = 0; $i < 20; $i++) {
        $já->dřep();
        $já->výskok();
      }
    - ⇒ nejdůležitější je prostřední část závorky (také jediná povinná), který určuje podmínku. Tato podmínka říká, že je-li pravdivá, pokračujeme v provádění opakování. Levá část definuje libovolnou proměnnou s počáteční hodnotou. Pravá část říká, co se má stát po vykonání každého opakování.
    - ⇒ exekuce tedy vypadá takto:
        $i se rovná 0.
        $i má menší hodnotu než 20 (= true).
        Pokračuji blokem kódu.
        Provedu dřep. Provedu výskok.
        Provede se příkaz $i++, což je zkratka pro $i = $i + 1;
        Ocitám se na konci cyklu a vracím se na jeho začátek.
        “$i = 0” je již ignorováno, to byla pouze úvodní definice.
        Vyhodnocuje se podmínka $i < 20. $i se rovná 1, tedy pravda.
        A takto stále dokola dokud se $i nebude rovnat 20
        Tehdy bude podmínka vyhodnocena jako false a cyklus je opuštěn.
- Případ druhého typu (solení polévky) vyřešíme cyklem WHILE úplně jednoduše:
    - while($polevka->jeOsolena() == false) {
        $polevka->přisolit();
      }
    - ⇒ exekuce vypadá takto:
        Program vyhodnotí podmínku v závorce.
        Je-li podmínka pravdivá, pak provede blok kódu.
        $polevka->jeOsolena() vrátí hodnotu false.
        false == false je ale true, tedy je podmínka splněna
        Provedeme kód, kterým je přisolení polévky.
        Vracíme se na začátek a opět ověřujeme slanost polévky.
        Je-li již polévka dost slaná, pak se z podmínky stává true == false.

True == false není pravda, tedy je podmínka nesplněna-
Jakmile je podmínka false, opuštíme cyklus.

Předčasné ukončení cyklu (libovolného)

- Cykly je možné ukončit předčasně použitím klíčového slova: break.
- Jakmile exekuce dorazí k příkazu break, opustí nejbližší nadřazený cyklus. (Cykly, stejně jako další věci, mohou být do sebe vnořené. Tímto se tedy dostaneme pryč z nynějšího cyklu, ale je možné, že budeme pokračovat v dalším nadřazeném cyklu.)
- Příklad použití:
    - while($polevka->jeOsolena() == false) {
        If ($polevka->bubláAPřetékáPřesHrnec() == true) {
            break;
        }
        $polevka->přisolit();
    }
    - ⇒ Při solícím cyklu vždy zkontroluji, zda polévka náhodou není mimo veškerou kontrolu a pokud snad ano, přestanu okamžitě solit a nejspíš se začnu zabývat mícháním a snižováním teploty.