

Datové typy	1
Použité jmenné prostory v XML	1
Registrace vlastního typu.....	1
Příklady použití	1
Základní datové typy.....	2
Binární (boolean).....	2
Výčtové (enumeration).....	2
Číselné (numeric)	3
Řetězcové (string)	4
Znakové (char)	4
Typ funkce (function)	5
Složené/odvozené datové typy.....	5
Povolení nedefinované hodnoty (null, undefined).....	5
Množinové (set)	6
Statické pole (array)	6
Asociativní pole (map)	6
Dynamické pole (kontejner) - seznamy, fronty, zásobníky.....	7
Struktura (struct)	8
Objektově orientované programování.....	8
Třída (class).....	8
Rozhraní (interface)	9
Operace v OOP	9
Volání funkcí a metod v xmlns="Code"	9
Úplný seznam elementů z xmlns="Type"	10

Datové typy

Použití jmenné prostory v XML

V následovném textu budou použity následovní jmenné prostory (XML namespaces):

- Type - pro definici datových typů
- Const - pro definici základních konstant
- Code - pro zápis kódu (není předmětem této kapitoly)

Registrace vlastního typu

Vlastní typ lze registrovat pomocí elementu pro definici typu následovně:

```
<typedef id="ThreeStateBoolean" xmlns="Type">
  <optional>
    <bool />
  </optional>
</typedef>
```

Zápis registruje typ ThreeStateBoolean jako složený typ:

```
<optional xmlns="Type">
  <bool />
</optional>
```

Což je volitelná binární hodnota - tj. může nabýt hodnot *true*, *false* nebo *undefined*.

Příklady použití

Definujeme si vlastní, uživatelský typ:

```
<!-- uživatelský typ -->
<typedef id="ThreeStateBoolean" xmlns="Type">
  <optional>
    <bool />
  </optional>
</typedef>
```

Z tohoto typu můžeme vytvořit jednoduše proměnnou:

```
<!-- přímá definice typu -->
<variable id="a" type="ThreeStateBoolean" xmlns="Code" />
```

Taky to ale můžeme zapsat takhle:

```
<!-- nepřímá definice typu, odkaz na uživatelský typ -->
<variable id="b" xmlns="Code">
  <type xmlns="Type">ThreeStateBoolean</type>
</variable>
```

Což ve své podstatě je zápis

```
<!-- in-line definice typu -->
<variable id="b" xmlns="Code">
  <optional xmlns="Type">
    <bool />
  </optional>
</variable>
```

Definice typu pro binární funkci (pozor na to, že atribut Type:type neexistuje)

```
<!-- uživatelský typ -->
<typedef id="3ST2" xmlns="Type">
  <function>
    <return>
      <type>ThreeStateBoolean</type>
    </return>
    <return>
      <parameter id="a">
        <type>ThreeStateBoolean</type>
      </parameter>
      <parameter id="b">
        <type>ThreeStateBoolean</type>
      </parameter>
    </return>
  </function>
</typedef>
```

Základní datové typy

Binární (*boolean*)

Nejjednodušší datový typ, reprezentuje pravdivostní hodnotu:

- pravda
- nepravda

Zápis v XML je následovní:

```
<bool xmlns="Type" />
```

Hodnota, kterou jde přiřadit do binárního typu může být binární konstanta, výsledek výrazu s relačními nebo logickými operátory nebo s operátorem identity. Konstanty jsou jenom dvě:

```
<true xmlns="Const" />
<false xmlns="Const" />
```

Relační operátory:

- menší než, větší než, rovný
- negace předchozích (větší nebo rovný, menší nebo rovný, nerovný)

Logické operátory:

- logický součet, logický součin
- operátor logické negace

Operátor identity se používá při objektech a zjišťuje, jestli jsou dva objekty jedna a ta samá instance. Taky existuje komplementární operátor (ne-identita).

Nepřímo, pomocí implicitních typových konverzí, je možno přiřadit do binárního typu taky zjednodušenou variantu jiných typů (pravdivá hodnota v závorce):

- číselné typy (nenulová hodnota)
- řetězcové typy (neprázdný řetězec)
- kontejnery (neprázdný)

Výčtové (*enumeration*)

Výčtový typ může nabýt jedné hodnoty z uvedeného seznamu a je interně očíslován v rozsahu 0 až N-1. Při generování kódu je možno vygenerovat funkce na převod výčtové hodnoty na řetězec a taky opačnou funkci, převod řetězce na výčtovou hodnotu. Implicitně se výčet chová jako celočíselný typ, takže je možno ho použít v přiřazení kde se požaduje číselný typ, u řetězcových operací se implicitně použije výše zmíněná funkce na převod. Přiřazení do binární hodnoty by ale nemělo být dovoleno, protože zde jsou nutné dvě implicitní konverze!

Zápis v XML:

```
<enum xmlns="Type">
  <option>DEFAULT</option>
  <option>MINIMIZED</option>
  <option>MAXIMIZED</option>
</enum>
```

Operace, které lze provádět s výčtovými typy jsou následovní:

- zjištění existence následovníka, předchůdce
- zjištění (změna na) následovníka, předchůdce, varianty jsou:
 - vrací tu samou hodnotu pokud je na pokraji
 - vrací nedefinovanou hodnotu (undefined)
 - vyvolá výjimku
- relační operátory, protože se to chová implicitně jako číselný typ
- zjištění pořadí (není až tak interní, souhlasí s uvedením v definici)
- převod čísla na výčtový typ, v případě hodnoty mimo rozsah
 - vrátí nedefinovanou hodnotu (undefined)
 - vyvolá výjimku

Číselné (numeric)

Číselné typy můžeme rozdělit na tyto tři kategorie:

- celočíselné (integer)
- s pohyblivou desetinnou tečkou (floating-point)
- komplexní číslo (complex)

Protože komplexní číslo jde reprezentovat strukturou, nebudeme ho definovat v této sekci, ale použijeme složený datový typ - strukturu. Zůstává teda celočíselný typ a typ s pohyblivou desetinnou tečkou. Oba tyto typy mohou mít různý rozsah (resp. přesnost), takže je nutno zavádět parametr do typu, který doplní informaci o přesnosti.

Příklady zápisu v XML bez udání přesnosti jsou:

```
<int xmlns="Type" />
<float xmlns="Type" />
```

Pokud je nutno, tak definujeme přesnost následovně:

```
<float size="32" xmlns="Type" />
<float size="48" xmlns="Type" />
<float size="64" xmlns="Type" />
<float size="80" xmlns="Type" />
```

V případě celých čísel můžeme definovat jak velikost, tak znaménkovost:

```
<!-- 8 bits unsigned -->
<int size="8" signed="0" xmlns="Type" />

<!-- 32 bits signed -->
<int size="32" signed="1" xmlns="Type" />
```

Speciálním případem celočíselného typu, připomínajícího v některých rysech výčet je typ rozsah, specifikován minimální a maximální hodnotou (rozsahem):

```
<!-- 8 bits unsigned (defined as range) -->
<range min="0" max="255" xmlns="Type" />
```

Pokud výsledek operace nelze reprezentovat pomocí daného rozsahu, tak se nastaví nedefinovaná hodnota, nebo se vyvolá výjimka (dle toho jestli je typ základní range nebo složený typ optional-range).

Pro zjednodušení konverze na řetězec, je možnost specifikovat výchozí formátovací řetězec pro daný číselný typ (platí pro int, float i range):

```
<!-- BYTE as 00.0000,00 to 00.00FF,00 -->
<int size="8" signed="0" xmlns="Type"
    width="10" decimals="2" fillzeros="1" cut="no"
    base="16" uppercase="1" intgroup="4" decsep="," intsep="."
/>
```

Formátovací parametry jsou následovní (v závorce jsou výchozí hodnoty pro int):

- width - minimální šířka formátovaného výstupu (1)
- decimals - počet desetinných míst (0)
- fillzeros - jestli doplnit zleva nuly na požadovanou šířku (0)
- cut - jestli oříznout výsledek na požadovaný počet míst („no“)
 - no - neořezávat, výsledek může mít více než width znaků
 - left - ořezávat zleva (tj. zarovnání vpravo)
 - right - ořezávat zprava (tj. zarovnání vlevo)
- base - základ číselné soustavy ve které se bude výsledek zobrazovat (10)
- uppercase - v případě že je základ větší než 10, tak značí velikost písmen (0)
- intgroup - počet míst v oddělovači „tisíců“ (3)
- decsep - oddělovač desetinného místa, tj. desetinná tečka („.“)
- intsep - oddělovač „tisíců“ („“)

Implicitní konverze:

- na binární - dle nulové nebo nenulové hodnoty

- na řetězec - dle formátovacího řetězce!

Operace proveditelné s číselnými typy:

- relační
- logické (implicitní konverze na binární hodnotu)
- bitové (pro floating point se provede implicitní konverze na int, tj. celočíselná část)
- aritmetické
- řetězcové (implicitní konverze na řetězec)

Definice konstantních hodnot je jednoduchá:

```
<int xmlns="Const">123</int>
<int xmlns="Const">1E3</int>
<float xmlns="Const">3.1415</float>
<float xmlns="Const">1E-06</float>
```

V našem systému můžeme taky zavést další konstanty, třeba:

```
<pi xmlns="Const" />
```

Řetězcové (string)

Řetězcové typy obsahují řetězec písmen a dělí se na několik druhů podle toho, jestli můžou, nebo nemůžou obsahovat nulový znak. Typ který nemůže (protože je implementačně sám ukončen nulou se značí v XML jako:

```
<ascii xmlns="Type" />
```

Druhý typ, který může obsahovat i nulový znak je pak:

```
<string xmlns="Type" />
```

Implementace tohoto typu je statická nebo dynamická, s tím, že se uchovává délka řetězce namísto ukončování nulovým znakem.

Varianty obou typů se tvoří upřesněním formátu znaků (8 nebo 16 bitů):

```
<string char="ansi" xmlns="Type" />
<string char="wide" xmlns="Type" />
```

resp. znakové sady (pro wide string je UNICODE výchozí formát):

```
<string char="ansi" charset="UTF-8" xmlns="Type" />
<string char="wide" charset="UNICODE" xmlns="Type" />
```

Operátory pro řetězec jsou konkatenace, která vrací spojené řetězce a indexace, která vrátí příslušný znak dle indexu (nebo nedefinovanou hodnotu anebo vyvolá výjimku pokud je index mimo rozsah).

Implicitní konverze existují na číslo (počáteční číslice, pro FP se povoluje tečka nebo čárka) nebo binární hodnotu (vlastně test na neprázdný řetězec).

Řetězcové konstanty:

```
<string xmlns="Const">Hello world!</string>
```

Znakové (char)

Znaky jsou obdobou řetězců o délce jedna:

```
<ansichar xmlns="Type" />
<widechar xmlns="Type" />
```

Můžou být tak upřesněny o znakovou sadu:

```
<ansichar charset="ISO-8859-2" xmlns="Type" />
<ansichar charset="WINDOWS-1250" xmlns="Type" />
```

A můžou tvořit svoje vlastní konstantní výrazy:

```
<ansichar xmlns="Const">A</ansichar>
<ansichar charset="ISO-8859-2" xmlns="Const">á</ansichar>
<widechar xmlns="Const">á</widechar>
```

Implicitní konverze na řetězec je možná, při konverzi na číslo se vrací pořadové číslo znaku v příslušející znakové sadě. Konverze na binární hodnotu je možná taky (false při nulovém znaku, true v ostatních případech).

Speciální znakové konstanty jsou:

```
<eol xmlns="Const" />
<eof xmlns="Const" />
```

A pro ASCII také existují tyto:

```
<cr xmlns="Const" />
<lf xmlns="Const" />
<bs xmlns="Const" />
<tab xmlns="Const" />
<esc xmlns="Const" />
```

Typ funkce (function)

K typu funkce nás vedou požadavky pro předání odkazu na funkci některým jiným funkcím (třeba časté je specifikování porovnávací funkce pro řadící algoritmus quicksort, nebo jiné funkce využívající zpětné volání - call-back).

Typ funkce je definován parametry včetně typu a typem návratové hodnoty. Příkladem budiž výše uvedený call-back pro quicksort:

```
<function xmlns="Type">
  <return>
    <enum>
      <option>OK</option>
      <option>DOESNT_MATTER</option>
      <option>REVERSED</option>
    </enum>
  </return>
  <parameter id="a"><int signed="0" /></parameter>
  <parameter id="b"><int signed="0" /></parameter>
</function>
```

Typ funkce lze použít kromě situací call-back taky pro definici rozhraní a abstraktních metod tříd v objektově orientovaném přístupu.

Složené/odvozené datové typy

Povolení nedefinované hodnoty (null, undefined)

Nejjednodušší odvozený typ. Doplnuje daný typ o možnost nedefinované hodnoty, implementačně je možné ho realizovat dvěma způsoby:

- jako odkaz na původní typ s tím, že pokud ukazatel nikam neukazuje, tak se jedná o nedefinovanou hodnotu
- složený typ, doplnění původního typu o příznak definovanosti / nedefinovanosti

Ukázkový zápis volitelného binárního typu v XML je pak:

```
<optional xmlns="Type">
  <bool />
</optional>
```

Do proměnné tohoto typu pak lze přiřadit kromě true a false konstant taky tuto konstantu:

```
<undefined xmlns="Const" />
```

Implementaci je možno upřesnit nápovědou:

```
<optional xmlns="Type" hint="pointer"><!-- implementace pres ukazatel -->
  <bool />
</optional>

<optional xmlns="Type" hint="flag"><!-- implementace pres priznak -->
  <bool />
</optional>
```

Generátor se nemusí řídit nápovědou a výchozí způsob implementace taky není definován.

Množinové (set)

Množinový typ je odvozenina od výčtového - množina může obsahovat žádnou až všechny hodnoty daného výčtu. Operace které lze provést na výčtu jsou:

- sjednocení, průnik, odečtení
- funkce pro:
 - vrácení plné a prázdní množiny
 - maximální počet prvků, počet aktuálně obsažených prvků

Zápis v XML:

```
<set xmlns="Type">
  <enum>
    <option>MAX_HORIZONTAL</option>
    <option>MAX_VERTICAL</option>
    <option>VISIBLE</option>
  </enum>
</set>
```

Množina by se měla implementovat třídou, která pak obsahuje příslušné metody jak pro operace tak pro výše popsané funkce.

Statické pole (array)

Statické pole je pole, o kterém dopředu víme kolik bude mít prvků. Příkladem je třeba IP adresa (IPv4) která v XML bude vypadat následovně:

```
<array size="4" xmlns="Type">
  <int size="8" signed="0" />
</array>
```

Pro definování rozsahu indexu do pole můžeme použít následovní způsoby zápisu:

```
<!-- 0..4 -->
<array size="4" xmlns="Type"><!-- ... --></array>
<array min="0" size="4" xmlns="Type"><!-- ... --></array>
<array size="4" max="3" xmlns="Type"><!-- ... --></array>
<array min="0" max="3" xmlns="Type"><!-- ... --></array>

<!-- 1..5 -->
<array min="1" size="5" xmlns="Type"><!-- ... --></array>
<array size="5" max="5" xmlns="Type"><!-- ... --></array>
<array min="1" max="5" xmlns="Type"><!-- ... --></array>
```

Pokud existuje minimální i maximální hodnota, a je definována velikost, tak se při generování (nebo validaci) musí zobrazit varování. V případě že uvedená velikost pak nesouhlasí s počtem podle minima a maxima, vyhlásí se chyba která musí být napravena. Operátor specifický pro pole je indexace - je to binární operátor, kde jedním operandem je pole a druhým je index, výsledný typ je typ prvku pole. Indexování mimo rozsah vrací nedefinovanou hodnotu nebo vyvolá výjimku (podle typu elementu, nebo spíše podle nápovědy k operátoru indexace).

Asociativní pole (map)

Asociativní pole je jistý druh šablony (mapa), vytváří typ kde jsou prvky jednoho typu asociovány s (resp. mapovány na) prvky jiného typu.

Příkladem - program pro spočtení počtu výskytů slov v textu bude využívat pole kde se asociuje slovo (řetězec) s příslušným počtem výskytů (kladné číslo):

```
<map xmlns="Type">
  <string char="ansi" />
  <int signed="0" />
</map>
```

Operátor indexace v tomto případě dovolí jako index hodnotu kterou lze převést na typ shodný s typem indexu v definici asociativního pole.

Pokud se asociativní pole implementuje jako objekt, může mít funkce pro zjištění počtu dvojic a taky metody pro uzamykání vůči zápisu nebo přepisu (co se stane když nastane zápis nebo přepis zamknutého pole - ignoruje se, povolí se, vyvolá se výjimka).

Dynamické pole (kontejner) - seznamy, fronty, zásobníky

Seznam, dynamické pole:

```
<list xmlns="Type">
  <string char="ansi" />
</list>
```

Fronta (FIFO):

```
<buffer xmlns="Type">
  <string char="ansi" />
</buffer>
```

Zásobník (LIFO):

```
<stack xmlns="Type">
  <string char="ansi" />
</stack>
```

Operace které se dají provést na všech typech:

- operace posuvů:
 - go_first, go_last
 - go_prev, go_next
- práce s hodnotami:
 - get, get_first, get_last
 - set, set_first, set_last
- úprava počtu prvků
 - delete, delete_first, delete_last
 - insert
 - pre
 - post
 - first
 - last
- ostatní funkce
 - is empty, item count
 - get current index, go to index, get by index, set by index
 - shift, unshift
 - pop, push
 - append

Protože je zde zřejmý objektový přístup, může se dynamické pole implementovat taky jako komponenta, které se určí minimální a maximální velikost - a pokud se meze překročí, zavolá se příslušná obslužná rutina (onEmpty, onFull) která může modifikovat obsah pole a říct co se má provést s prvkem (vložit znova, ignorovat, odmítnout s oznámením, odmítnout s výjimkou).

Příklad fronty pro tvorbu řádků o 64 znacích:

```
<typedef id="outBuffer" xmlns="Type">
  <buffer size="64">
    <ansichar />
    <onFull>
      :
      : this.first();
      : while( this.length > 0 )
      :   stdout.write( this.shift() );
      :   stdout.writeln();
      :
    </onFull>
  </buffer>
</typedef>
```

Metoda onFull se provede i v rámci destrukturu pokud je počet prvků větší než 0.

Struktura (struct)

Struktura sjednocuje více datových položek do společného typu, příklad v XML:

```
<typedef id="complex" xmlns="Type">
  <struct>
    <elem id="re"><float size="64" /></elem>
    <elem id="im"><float size="64" /></elem>
  </struct>
</typedef>
```

Objektově orientované programování

Třída (class)

Třída se skládá ze zapouzdřených dat (variable), vlastností (property) a metod (method) rozdělených do tří skupin podle přístupových práv (private, protected, public). Vlastnosti se chovají jako proměnné, ale při nastavování nebo zjištění hodnoty se provede příslušná funkce dle get/set atributů.

Příklad v XML (třídy mají vlastní identifikátory, takže nepoužívají typedef, elementy fcall a mcall jsou zapsány v zjednodušené formě):

```
<class id="math.Complex" xmlns="Type">
  <protected>
    <method id="getLength">
      <return><float /></return>
      <fcall id="math.sqrt" xmlns="Code">
        <add>
          <fcall id="math.pow2"><this>re</this></fcall>
          <fcall id="math.pow2"><this>im</this></fcall>
        </add>
      </fcall>
    </method>
  </protected>
  <public>
    <variable id="re"><float size="64" /></variable>
    <variable id="im"><float size="64" /></variable>
    <property id="length" get="getLength" />
    <constructor>
      <parameter id="re"><float /></parameter>
      <parameter id="im"><float /></parameter>
      <assign xmlns="Code"><this>re</this><par>re</par></assign>
      <assign xmlns="Code"><this>im</this><par>im</par></assign>
    </constructor>
    <method id="getNegative" type="cplxOperation">
      <new classname="math.Complex" xmlns="Code">
        <neg><this>re</this></neg>
        <neg><this>im</this></neg>
      </new>
    </method>
    <method id="add" type="cplxOperationOne">
      <inc xmlns="Code"><this>re</this>
        <elem id="re"><par>remote</par></elem></inc>
      <inc xmlns="Code"><this>im</this>
        <elem id="im"><par>remote</par></elem></inc>
    </method>
  </public>
</class>
```

Typ metody která vrací negaci komplexního čísla je (návratová hodnota - reference):

```
<typedef id="cplxOperation" xmlns="Code">
  <method xmlns="Type">
    <return><cref>math.Complex</cref></return>
  </method>
</typedef>
```

Pro vrácení vlastního objektu (self) a odkaz na typ objektu (current) se může použít pak:

```
<typedef id="cplxOperationOne" xmlns="Code">
  <method xmlns="Type">
    <return><self /></return>
  </method>
  <parameter id="remote"><current /></return>
</typedef>
```

Rozhraní (interface)

Rozhraní k výše uvedené třídě o komplexním číslu by bylo:

```
<interface id="math.Complex" xmlns="Type">
  <method id="getNegative" type="cplxOperation" />
  <method id="add" type="cplxOperationOne" />
</interface>
```

Použití rozhraní je pak následovní:

```
<class id="math.Complex" xmlns="Type">
  <implements>
    <if>math.Complex</if>
  </implements>
  :
  :
</class>
```

Operace v OOP

Definice proměnných s objektem a se skalární hodnotou:

```
<!-- var CN: Complex; -->
<variable id="CN" xmlns="Code">
  <cref xmlns="Type">math.Complex</cref>
</variable>

<!-- var CNL: Float; -->
<variable id="CNL" xmlns="Code">
  <float xmlns="Type" />
</variable>
```

Vytvoření instance dané třídy a přiřazení proměnné

```
<!-- CN = new Complex( 1.5 + 3.5j ); -->
<assign xmlns="Code">
  <var>CN</var>
  <new classname="math.Complex">
    <float xmlns="Const">1.5</float>
    <float xmlns="Const">3.5</float>
  </new>
</assign>
```

Přiřazení pomocí vlastnosti „length“:

```
<!-- CNL := CN.length; -->
<assign xmlns="Code">
  <var>CNL</var>
  <elem id="length">
    <var>CN</var>
  </elem>
</assign>
```

Přiřazení pomocí zavolání metody „getLength“ (tohle ale není validní pro naši třídu, protože uvedená metoda je privátní):

```
<!-- CNL := CN.getLength(); -->
<assign xmlns="Code">
  <var>CNL</var>
  <mcall id="getLength">
    <var>CN</var>
  </mcall>
</assign>
```

Volání funkcí a metod v xmlns="Code"

```
<fcall xmlns="Code">
  <func><!-- volana funkce --></func>
  <args><!-- parametry funkce --></args>
</fcall>

<mcall xmlns="Code">
  <obj><!-- objekt --></obj>
  <method><!-- nazev metody --></method>
  <args><!-- parametry metody --></args>
</mcall>
```

Úplný seznam elementů z xmlns="Type"

- typedef @id :TYPE
- type :NAME
- bool
- enum
 - option
- int @size? @signed?1
- float @size?
- range @min @max
- ascii @char?char/wide @charset?ascii
- string @char?wide/char @charset?unicode
- ansichar, widechar
- function
 - ? return :TYPE
 - * parameter @id :TYPE
- optional @hint? :TYPE
- set :enum / *option
- array @min?0 @max? @size?1 :TYPE
- map :TYPE+TYPE
- list, buffer, stack @size?
 - onFull
 - onEmpty
- struct
 - elem
- method @static?0
 - ? return :TYPE
 - * parameter @id :TYPE
- class @id @extends? @forceAbstract?0
 - ? implements
 - * if :NAME
 - ? private
 - ? protected
 - ? public
- class/private ~ class/protected ~ class/public
 - variable @id :TYPE
 - property @id @get? @set?
 - constructor
 - CODE
 - method @id @type
 - CODE
 - method @id @static?0
 - ? return :TYPE
 - * parameter @id :TYPE
 - CODE
- self, current
- iref, cref :NAME