3. Datové typy

Jak již bylo uvedeno, Java je přísně typový jazyk, proto je vždy nutno uvést datový typ datového atributu, formálního parametru metody, návratové hodnoty metody nebo pomocné proměnné v metodě. Datové typy v Javě lze rozdělit do těchto skupin:

- primitivní typy:
  - čísla (byte, short, int, long, float, double),
  - znaky (char),
  - logické hodnoty (boolean),
- referenční typy:
  - třídy (class),
  - výčtový typ (enum) – od verze Java 5,
  - rozhraní (interface),
  - pole (array).

Výčet všech dostupných primitivních datových typů je uveden v tabulce 3.1. Není možné vytvářet vlastní primitivní typy. Referenční datové typy nelze výčtem vypsat, vytvořením nové třídy nebo rozhraní vzniká nový datový typ. Standardní distribuce Javy obsahuje několik tisíc tříd a rozhraní. Nejdříve si popíšeme primitivní datové typy a operace s nimi spojené. Následovat bude popis rozdílů mezi referenčními a primitivními typy a operací s referenčními typy. Na konci kapitoly uvedeme možnosti převodu mezi primitivními datovými typy a odpovídajícími třídami.

3.1. Primitivní datové typy

Java používá pro celočíselné proměnné čtyři datové typy: **long**, **int**, **short** a **byte**, pro reálná čísla dva typy: **double** a **float**, pro znaky typ **char** a pro logické proměnné typ **boolean**. Velikost, jakou zabírají v paměti, a rozsah, který jsou schopny obsáhnout, vidíme v následující tabulce.

název typu	**velikost (v bytech)**	**rozsah**	**implicitní hodnota**
long	8	−9 223 372 036 854 775 808 až +9 223 372 036 854 775 807	0
int	4	−2 147 483 648 až +2 147 483 647	0
short	2	−32 768 až +32 767	0
byte	1	−128 až +127	0
double	8	±1.797 693 134 862 315 70 E+308	0.0
float	4	±3.402 823 47 E+38	0.0
char	2	65 536 různých znaků	\u0000
boolean	1 bit	`true` nebo `false`	`false`

Tabulka 3.1 Přehled primitivních datových typů

3.1.1. Deklarace a inicializace proměnné primitivního typu

Při deklaraci je třeba uvést identifikátor a datový typ, volitelně lze zadat i počáteční hodnotu. Počáteční hodnota může být uvedena nejen konstantou (hodnotou), ale i výrazem, např.:

```
    long delkaDne = 24 * 60 * 60;
```

Výraz, kterým se přiřazuje hodnota, se musí dát vyhodnotit v době překladu, popř. v okamžiku spuštění programu. Pokud programátor při deklaraci pomocné proměnné metody nepřiřadí počáteční hodnotu, bude obsah proměnné libovolný. Při deklaraci datového atributu přiřadí implicitní hodnotu kompilátor – implicitní hodnoty jednotlivých typů jsou v tabulce 3.1. Z důvodu dobrých programátorských návyků je vhodné explicitně uvádět počáteční hodnotu i u datových atributů. Není určeno, jak se mají překladače zachovat při přiřazení počáteční hodnoty mimo rozsah proměnné (např. `byte b = 200`), některé překladače přiřadí nesmyslnou hodnotu, některé ohlásí chybu.
Z celočíselných typů doporučujeme používat pouze typy **int** a **long**, typy `byte` a `short` by se měly využívat pouze při předávání hodnot s okolím (s databází, s komponentou v jiném programovacím jazyku, atd.). Při matematických operacích (např. při sčítání) se proměnné těchto typů převádějí na `int` a poté zpátky. Obdobně u reálných čísel se preferuje typ **double**.

3.1.2. Konstanty

Konstanty se používají nejen při deklaraci proměnných, ale i v rámci výrazů.
Celá čísla se zapisují obdobným způsobem jako v běžném životě, např. 9876; zápis nesmí obsahovat mezery či jiné oddělovače (např. tisíců). Je-li prvním znakem nula, znamená to, že číslo zapisujeme v osmičkové (oktalové) soustavě, začíná-li zápis čísla prefixem `0x`, jedná se o číslo v hexadecimální (šestnáctkové) soustavě. Následující tři proměnné obsahují stejnou počáteční hodnotu:

```
    int cisloDek = 10;
    int cisloOkt = 012;
    int cisloHex = 0xa;
```

Celočíselná konstanta je vždy typu `int`, pokud potřebujeme konstantu jiného celočíselného typu, je třeba uvést přetypování nebo literál (viz dále).
Reálná čísla se zapisují buď s desetinnou tečkou (např. `0.25`, `0.00002`) nebo se používá tzv. semilogaritmický zápis, kdy číslo `1.54 * 106` se zapíše ve tvaru `1.54e6`. Reálné konstanty jsou vždy typu `double`.
Každý **znak** je v jazyce Java uložen v kódování Unicode ve dvou bytech[3]. Znaky se píší v jednoduchých uvozovkách, a to buď v defaultním kódu operačního systému (v českých Windows je to kódová stránka CP1250), kdy překladač sám zajistí převod na příslušný znak Unicode, nebo použijeme zápis \uXXXX, ve kterém místo XXXX uvedeme hexadecimální kód příslušného znaku.

```
    char cSHackem = 'č';
    char nejakyZnak = '\u12ab';
```

Logické hodnoty – pro proměnné typu **boolean** jsou definovány dvě konstanty – **true** a **false.**
Programátory se znalostí jazyka C upozorňujeme, že jako logické hodnoty nelze v Javě používat čísla.

3.1.3. Pojmenované konstanty

V Javě lze vytvářet pojmenované konstanty primitivních datových typů – definují se stejně jako proměnné s počáteční hodnotou s tím, že se na začátku uvede klíčové slovo **final**:

```
    final long DELKA_DNE = 24 * 60 * 60;
```

Proměnnou definovanou s modifikátorem `final` nelze již dále změnit, hodnotu lze přiřadit pouze jednou (tj. jakýkoliv pokus přiřadit do proměnné `DELKA_DNE` novou hodnotu skončí chybou při překladu). Podle konvencí Javy se ve jménech konstant používají velká písmena a pro oddělení slov podtržítko.

[3] Od verze 5 Java podporuje i rozšířené znaky Unicode 4.0 – znaky, které jsou mimo rozsah dvou bytů typu `char`. Bližší informace o jejich používání je v dokumentaci Javy u firmy Sun.

3.1.4. Přetypování

U celočíselné konstanty zapsané v programu se předpokládá datový typ `int`, u reálných konstant datový typ `double`. Pokud chceme, aby byla konstanta jiného datového typu, je třeba zapsat ji s literálem. Literál pro `long` je L, pro `float` f a pro `double` d (lze používat malá i velká písmena, u typu `long` je vhodné používat jen velké L, aby nedošlo k záměně s jedničkou). Zapíšeme-li konstantu `100000L` bude uložena jako `long`, zápis `10000f` znamená, že číslo je uloženo jako reálné typu `float`.
Pokud se hodnota proměnné kratšího typu ukládá do proměnné delšího typu, provede Java převod automaticky (neexistuje nebezpečí ztráty informace, `byte`→`short` →`int` →`long` →`float` →`double`). Například:

```
int cislo1 = 12;
long cislo2;
cislo2 = cislo1;
```

U převodu z kratšího typu na delší nemusí být vždy jasné, kdy se převod provádí. Ve výrazu

```
long vysledek = 10000*10000;
```

se nejprve provede násobení konstant typu `int` a poté se výsledek typu `int` převede na typ `long` a uloží do proměnné `vysledek`[4].
Pokud potřebujeme opačné přiřazení, je třeba provést explicitní přetypování a oznámit tak překladači, že případná ztráta informace nevadí, např.:

```
float desetinneCislo1;
double desetinneCislo2 = 0.123456789;
desetinneCislo1 = (float) desetinneCislo2;
```

Při operacích s číselnými proměnnými typu `byte` a `short` se tyto proměnné automaticky převádějí na typ `int`, tj. výsledek operace s dvěma těmito proměnnými je typu `int`.
Při operaci s proměnnými/konstantami různě dlouhých typů je výsledek delšího typu, tj. výsledkem operace proměnných typů `int` a `long` je hodnota typu `long`, výsledkem operace typů `int` a `double` je proměnná typu `double` atd. Proměnná kratšího typu se převede na delší typ před provedením operace.
Pokud trváme na uložení do kratšího typu, je nutné explicitně přetypovat výsledek operace:

```
long cislo1 = 10000;
int cislo2 = 100;
int vysledek = (int) (cislo1 – cislo2);
```

3.1.5. Přetečení

V situaci, kdy informace ukládaná do proměnné má větší hodnotu než je maximální hodnota typu, mluvíme o tzv. **přetečení**. Java neohlásí žádnou chybu, ale usekne přesahující část a program tak vyprodukuje špatný výsledek. Např. výsledkem operace `100000 * 100000` je číslo `1 410 065 408` typu `int` místo správného `10 000 000 000`, neboť tento výsledek je mimo rozsah typu `int`. K získání správného výsledku je potřeba aspoň jeden z operátorů označit jako `long`, tj. operace by měla vypadat následovně: `100000L * 100000`.

3.2. Výrazy a operátory

3.2.1. Operátor přiřazení (=), přiřazovací příkaz

Operátor přiřazení = se v Javě používá pro přiřazování hodnot u primitivních datových typů.
S použitím operátoru přiřazení se vytváří přiřazovací příkaz. Překladač kontroluje typovou správnost

[4] V Javě se nejdříve vyhodnotí pravá strana operace přiřazení nezávisle na levé straně.

přiřazení – na obou stranách operátoru musí být buď stejné typy, nebo typy musí být v souladu s pravidly přetypování. Pokud přiřadíme hodnotu z jedné proměnné primitivního datového typu do druhé, vznikne kopie této hodnoty.

```
int cislo1 = 5;
int cislo2 = cislo1;
```

Použití operátoru přiřazení pro referenční typy si popíšeme později.

3.2.2. Aritmetické operátory

Se základními matematickými operátory jsme se už seznámili. Jejich přehled najdete v následující tabulce.

aritmetický operátor	význam
+	součet
−	rozdíl
*	násobení
/	dělení (celočíselné i desetinné)
%	zbytek po celočíselném dělení

Tabulka 3.2 Přehled aritmetických operátorů

U přetypování a přetečení jsme už hovořili o problémech, které mohou nastat při těchto operacích. S dalším problémem se setkáváme u dělení celých čísel. Pokud v programu použijeme tento výraz

```
double vysledek = 9/2;
```

bude v proměnné `vysledek` hodnota `4`. Výsledkem dělení dvou celých čísel je opět celé číslo bez ohledu na typ proměnné, do které výsledek ukládáme (tj. jedná se o celočíselné dělení). Proto je i zde třeba použít přetypování nebo literál u jednoho z operandů, aby se provedlo desetinné dělení, např.:

```
double vysledek = 9d/2;
```

V případě proměnných je potřeba jeden z operandů přetypovat, např. pokud `cislo1` a `cislo2` jsou typu `int`, bude výraz vypadat následovně[5]:

```
double vysledek = ((double) cislo1) / cislo2;
```

Java používá řadu zkrácených zápisů některých operací, jsou uvedeny v následující tabulce.

operátor	příklad	význam operátoru
+=	x += y	x = x + y
−=	x −= y	x = x − y
/=	x /= y	x = x / y
*=	x *= y	x = x * y
%=	x %= y	x = x % y

[5] Výraz lze zapsat i ve tvaru `double vysledek = (double) cislo1 / cislo2`, ale zde je nejasné, zda se přetypování týká prvního operandu či výsledku – závisí na prioritách přetypování a dělení. Zápis `double vysledek = (double) (cislo1 / cislo2)` je určitě chybně, neboť přetypovává až vlastní výsledek operace, která proběhne jako celočíselná.

operátor	příklad	význam operátoru
++	x++ ++x	x = x + 1
−−	x−− −−x	x = x − 1

Tabulka 3.3 Význam složených aritmetických operátorů

Pro často se vyskytující operace zvyšování nebo snižování hodnoty proměnné o jedničku lze v Javě použít i zkrácené zápisy `x++`, `++x`, `x--`, `--x`. Obvykle se tyto výrazy používají jako samostatné příkazy, lze je však (bohužel[6]) použít i na pravé straně výrazu. Zde se rozlišuje, zda použijeme `++` jako předponu (prefixový tvar) nebo příponu (postfixový tvar). Pokud se operátor `++` zapíše jako předpona, nejprve se zvýší hodnota proměnné a pak se použije. Pokud je `++` zapsán jako přípona, pracuje se ve výrazu s původní hodnotou a teprve poté je zvýšena. Vysvětlíme si to v následujících příkladech.

```
int puvodni = 10;
int nova = puvodni++;
```

Proměnná `puvodni` je nyní rovna `11` a proměnná `nova` je rovna `10`.

```
int puvodni = 10;
int nova = ++puvodni;
```

Proměnná `puvodni` je nyní rovna `11` a proměnná `nova` je rovna také `11`.
Pro operátor −− platí stejná pravidla. Stejné problémy jsou i při použití těchto operátorů na místě parametrů metod.
Operátor + se používá též pro spojování řetězců – blíže bude popsáno v kapitole 5.

3.2.3. Relační operátory

Relační operátory se používají pro porovnání hodnot dvou číselných proměnných (pokud nejsou stejného typu, tak se převedou na delší typ), proměnných typu `char` a `boolean`. Výsledkem je vždy hodnota typu `boolean`, tj. buď pravda (`true`) nebo nepravda (`false`). Obvykle se používají v příkazu `if` a v příkazech cyklu pro vytváření podmínky[7]. Přehled relačních operátorů je v následující tabulce (pozor, záleží na pořadí znaků v operátorech):

relační operátor	význam
==	rovná se
!=	nerovná se
<	menší než
>	větší než
<=	menší nebo rovno
>=	větší nebo rovno

Tabulka 3.4 Přehled relačních operátorů

Použití operátorů `==` a `!=` pro referenční typy si popíšeme později.

[6] U výrazů se předpokládá, že se při výpočtu nemění hodnoty operátorů použitých na pravé straně výrazu – operátory `++` a −− toto pravidlo porušují, což může vést k obtížně odhalitelným chybám.
[7] V textu se používá pojem podmínka místo správnějšího pojmu „výraz s výslednou hodnotou typu `boolean`“. Pojem podmínka je sice méně přesný, avšak často lépe pochopitelný.

3.2.4. Logické operátory

Logické operátory slouží pro vyjádření vztahu mezi dvěmi proměnnými/výrazy typu `boolean`, tj. obvykle k vytváření složených podmínek. Logické operátory používané v Javě jsou uvedeny v tabulce 3.5.

logický operátor	význam
&	logická spojka AND, vyhodnocují se oba operandy
|	logická spojka OR, vyhodnocují se oba operandy
&&	podmíněná logická spojka AND, pravý operátor se vyhodnocuje, pouze pokud je levý `true`
||	podmíněná logická spojka OR, pravý operand se vyhodnocuje, pouze pokud je levý `false`
!	negace NOT

Tabulka 3.5 Přehled logických operátorů

Chceme-li například otestovat, jestli je proměnná `i` větší nebo rovna 10 a současně menší než 50, zapíšeme podmínku takto:

```
(i >= 10) && (i < 50)
```

Výsledkem je hodnota `true` nebo `false`.

3.2.5. Bitové operátory

Java umožňuje pracovat i s jednotlivými bity celočíselných hodnot. Vzhledem k jejich výjimečnému použití zde nejsou popsány, v případě potřeby je nastudujte z on-line dokumentace.

3.2.6. Podmíněný výraz

V Javě existuje ternární[8] operátor, který slouží k vytvoření podmíněného výrazu, syntaxe je:

```
podmínka ? výraz1 : výraz2
```

Můžeme zapsat např. následující podmíněný příkaz:

```
cislo1 = cislo1 < 5 ? cislo1 + 1 : 0;
```

Pokud je hodnota proměnné `cislo1` menší než 5 (podmínka je splněna), zvýší se její hodnota o 1. Pokud je hodnota proměnné `cislo1` větší nebo rovna 5 (tedy podmínka není splněna), přiřadí se do této proměnné hodnota 0.
Zápis podmíněného výrazu je málo přehledný a proto se dává přednost podmíněnému příkazu **if** (viz kapitola 4).

3.2.7. Kulaté závorky

Kulaté závorky ( ) se v Javě používají na následujících místech:

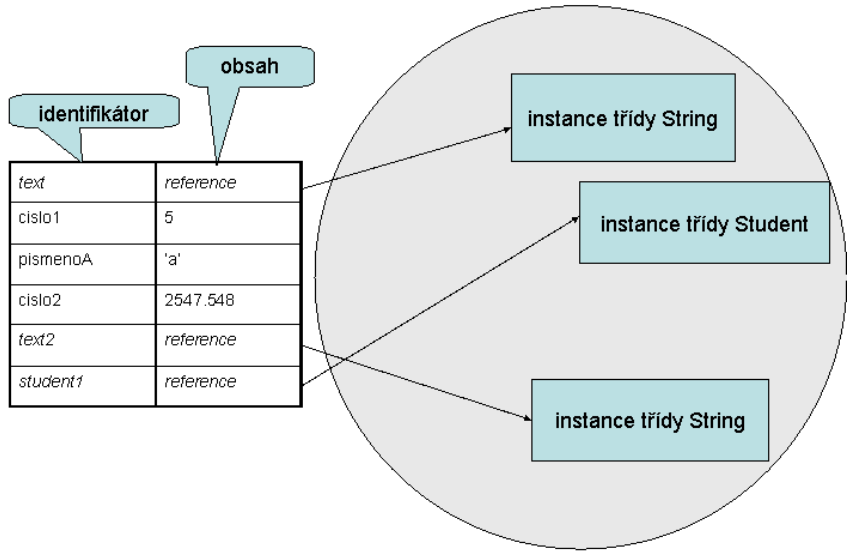
- ve složitějších výrazech pro vyjádření priority operací,
- jako operátor přetypování,
- v deklaraci metod pro uzavření seznamu formálních parametrů,
- při volání metod pro uvedení seznamu skutečných parametrů,
- v příkazech selekce a iterace pro uvedení podmínky,
- při odchytávání výjimek.

[8] Pojem ternární znamená, že operátor používá tři operandy. Obvykle se používají dva operandy (např. při sčítání) či jeden operand (např. logická negace či operátor ++) – používá se označení binární a unární operátory.

3.3. Referenční datové typy

3.3.1. Rozdíl mezi primitivními a referenčními datovými typy

Jak již bylo uvedeno v úvodní kapitole, na každou instanci, kterou budeme v našem programu využívat, si musíme uložit odkaz (referenci) do identifikátoru odpovídajícího typu. Samotný obsah jednotlivých datových atributů je poté dostupný přes tuto referenci. U primitivních datových typů se do identifikátoru neukládá odkaz, ale přímo hodnota. Zjednodušeně je to znázorněno na obrázku 3.1. Rozdíl mezi hodnotou a odkazem se projevuje v přiřazovacím příkazu či v předávání parametrů metodám – viz přiřazovací příkaz dále. Na jednu instanci referenčního typu může odkazovat více identifikátorů. U primitivních typů může jednu hodnotu obsahovat právě jeden identifikátor, druhý identifikátor může odkazovat pouze na kopii hodnoty.
Identifikátor referenčního typu nemusí odkazovat na žádnou instanci, identifikátor primitivního typu vždy obsahuje nějakou hodnotu, byť někdy může být náhodná (deklarace lokální proměnné).



Obrázek 3.1 Znázornění rozdílu mezi referenčními a primitivními datovými typy

3.3.2. Konstanta null

S referenčními typy souvisí speciální konstanta **null**, která popisuje situaci, kdy identifikátor referenčního typu neodkazuje na žádnou instanci. Konstantu `null` lze použít v přiřazovacích příkazech či při porovnávání identifikátorů referenčního typu.
Datové atributy referenčního typu nemají při deklaraci přiřazenu žádnou instanci – obsahují konstantu `null` (pokud není součástí deklarace i inicializace). Pokud identifikátor obsahuje hodnotu `null` a zavoláme nějakou metodu, vznikne výjimka `NullPointerException` (výjimky jsou popsány v kapitole 12). Příklad vzniku takovéto výjimky následuje:

```
String retezec;
retezec.toUpperCase();   // vznikne NullPointerException
Student student = null;
student.getSemestr();          // vznikne NullPointerException
```

3.3.3. Přetypování referenčních typů

Podobně jako primitivní typy lze přetypovávat referenční typy. Automatické přetypování probíhá směrem k předkovi v dědičné hierarchii, pokud se má přetypovávat v opačném směru, musí být

explicitně uvedeno. Při přetypování referenčních typů se nikdy nemění vlastní instance, což je rozdíl od primitivních číselných typů, u kterých může dojít ke ztrátě číslic. Přetypovávat lze též na implementovaná rozhraní.

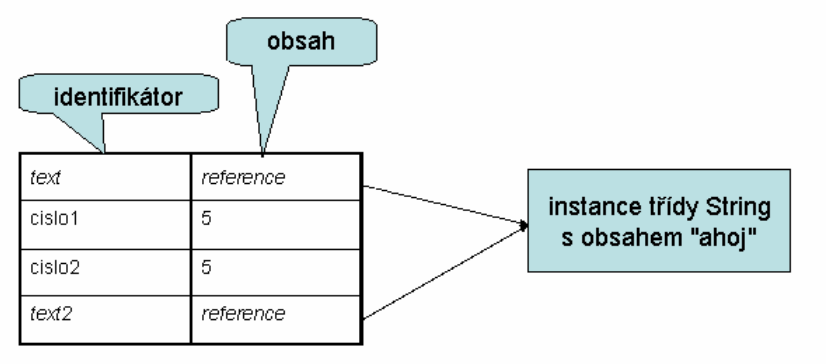
```
    String retezec = "řetězec";
    Object o = retezec;               // automatické přetypování
    String retezec2 = (String)o;      // explicitní přetypování
```

O přetypování referenčních typů bude uvedeno více podrobností v kapitole 11 věnované dědičnosti.

3.3.4. Operátor přiřazení (přiřazovací příkaz)

Operátorem přiřazení `=` se u referenčních typů přiřazují (kopírují) odkazy (reference). Při přiřazení referenčního typu vznikne kopie odkazu, instance zůstane v paměti pouze jednou. Překladač kontroluje typovou správnost přiřazení. Kopie instancí lze vytvářet pomocí metody `clone()` při splnění dalších podmínek.
Výsledek následujícího kódu je zobrazen na obrázku 3.2.

```
    String text = "ahoj";
    int cislo1 = 5;
    int cislo2 = cislo1;
    String text2 = text;
```



Obrázek 3.2 Znázornění rozdílu mezi přiřazováním primitivních a referenčních typů

Pro parametry metod platí stejná pravidla jako pro přiřazovací příkaz, neboť při volání metody se přiřadí (zkopíruje) skutečný parametr do formálního parametru. U primitivních typů se zkopíruje hodnota, u referenčních typů se zkopíruje odkaz. Zevnitř metody tudíž nelze změnit obsah původní primitivní proměnné, která byla použita jako parametr při volání metody. Nelze ani změnit odkaz na instanci, na kterou odkazuje identifikátor použitý při volání metody. U referenčních typů je však možné posílat zprávy příslušné instanci (volat metody instance). Tím se může změnit obsah datových atributů v instanci, na kterou ukazuje původní odkaz i kopie odkazu v parametru.

3.3.5. Relační operátory == a !=

Relační operátory `==` a `!=` je možné použít u referenčních datových typů pro porovnání odkazů. Lze pomocí nich zjistit, zda dva identifikátory ukazují na stejnou instanci, popř. zda identifikátor obsahuje hodnotu `null`. Tyto operátory nemohou sloužit pro logické porovnání (porovnání obsahu) dvou instancí (např. zda dva řetězce obsahují stejný text). Pro porovnání obsahu slouží v Javě metoda `equals()`, viz kapitola 6. V následujícím příkladu jsou použity instance třídy `Integer`, která bude popsána dále v kapitole.

```
Integer cislo1 = new Integer(10);         // vytvoření instance
Integer cislo2 = cislo1;            // zkopírování odkazu na instanci
Integer cislo3 = new Integer(10);   // vytvoření druhé instance
System.out.print(cislo1 == cislo2);             //výsledek je true
System.out.print(cislo1 == cislo3);             //výsledek je false
System.out.print(cislo1.equals(cislo3));  //výsledek je true
```

3.3.6. Relační operátor instanceof

Relační operátor **instanceof** se používá pro zjištění, zda lze instanci přetypovat na nějakou třídu či na nějaké rozhraní. Formální zápis vypadá následovně:

```
    identifikátor instanceof referenčníTyp
```

Operátor vrátí `true`, pokud je možné přetypovat na uvedený referenční typ instanci, na kterou odkazuje uvedený identifikátor. Operátor `instanceof` je použit např. v metodě `equals()` ve třídě `Mistnost` v projektu `Adventura` (v příkladu je vidět i přetypování referenčního typu):

```
    public boolean equals (Object o) {
        if (o instanceof Mistnost) {
            Mistnost druha = (Mistnost)o;
            return nazev.equals(druha.nazev);
        }
        else {
            return false;
        }
    }
```

3.3.7. Volání metod

Volání metod je základní způsob provádění operací s referenčními typy – metody jsme si popsali již v kapitole 2.7.

3.4. Obalové třídy pro primitivní typy

Ke každému primitivnímu datovému typu v Javě existuje třída zapouzdřující tento typ na referenční typ. Tyto třídy se označují jako **obalové typy** – představují obal (box) kolem primitivní hodnoty. V tabulce 3.6 jsou uvedeny obalové třídy pro jednotlivé primitivní typy
Obalové třídy mají následující význam:

- obsahují další metody pro práci s těmito typy (např. převod řetězce na číslo),
- obalové třídy číselných typů mají definovány konstanty pro určení maximální a minimální hodnoty daného typu např. `Integer.MAX_VALUE`,
- jsou v nich definovány další konstanty spojené s primitivními typy jako `Double.NEGATIV_INFINITY`, `Boolean.TRUE`, `Double.NaN`,
- v některých situacích nelze použít primitivní typ (např. při ukládání do seznamu), poté se použijí instance obalových tříd.

primitivní datový typ	obalová třída
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Tabulka 3.6 Přehled obalových tříd a jejich přiřazení k primitivním datovým typům

S hodnotami v obalovém typu (stejně jako u všech tříd) lze pracovat pouze pomocí metod. Dalším rysem těchto tříd je to, že jsou **read only**[9] – hodnotu, kterou obalují, již není možné změnit (lze samozřejmě vytvořit novou instanci s novou hodnotou).
V Javě 5.0 probíhají potřebné převody mezi primitivním datovým typem a odpovídající obalovou třídou automaticky (**autoboxing**), takže se může zdát, že mezi primitivním typem a jeho obalovou třídou není rozdíl. Lze např. napsat následující kód.

```
Integer cislo = 5;
cislo += 2;
```

Ve skutečnosti je však na prvním řádku vytvořena instance třídy `Integer` obalující hodnotu 5. Pro výpočet na dalším řádku je proveden převod na primitivní datový typ `int`, provedena operace sčítání a vytvořena nová instance třídy `Integer` obalující hodnotu 7. Ve starších verzích Javy musel tyto převody napsat programátor následovně:

```
Integer cislo = new Integer (5);
int pomocna = cislo.intValue(); // převod na primitivní typ
pomocna += 2;
cislo = new Integer(pomocna);
```

Je třeba si uvědomit, že takto napsaný kód je také mnohem pomalejší než kdybychom pro proměnnou `cislo` použili typ `int`. Pokud se pro uložení číselných hodnot použijí primitivní datové typy, jsou prováděné operace (matematické, porovnávání) 5x až 50x rychlejší.

Doporučení:
– dávejte přednost primitivním datovým typům,
– pokud je aplikace (část aplikace) pomalá, je vhodné zkusit optimalizovat obalové typy a jejich automatické konverze do primitivních datových typů.

Překladač se při automatických konverzích snaží optimalizovat kód – někdy to vede k nečekaným výsledkům. Např. při deklaraci dvou proměnných

```
Integer cislo1 = 5;
Integer cislo2 = 5;
```

vrací porovnání (`cislo1 == cislo2`) hodnotu `true` (porovnávají se hodnoty), při následující deklaraci však vrací hodnotu `false` (porovnávají se odkazy):

```
Integer cislo1 = new Integer(5);
Integer cislo2 = 5;
```

[9] Read-only třída je taková, která neposkytuje žádné metody, pomocí kterých by bylo možné měnit hodnoty datových atributů.

Využití obalové třídy pro **konverzi řetězce na číslo** si ukážeme na příkladě. Z grafiky dostaneme údaj od uživatele jako `String` (má to být např. počet kusů). Bude tedy třeba provést převod na číslo typu `int`. Kód využívající metodu `parseInt()` třídy `Integer` bude vypadat následovně:

```
    String text = vstupniPole.getText();
    int cislo = Integer.parseInt(text);
```

V této ukázce chybí ošetření chybného vstupu (např. uživatel zadá místo čísla písmena) – tento postup si ukážeme v kapitole 12 o výjimkách.