

# Více o Objective C

V minulém dílu jsme se seznámili se systémem objektů a ukázali jsme si všechny základní služby jazyka Objective C. Nyní se seznámíme se zbytkem konstrukcí, jež Objective C nabízí; ačkoli žádná z nich není pro programování bezpodmínečně nutná -- jednoduché testovací programky jste si snadno mohli vyzkoušet už s využitím služeb, popsaných minule -- dokáží programátorovi výrazně usnadnit život.

## Neobjektová rozšíření

Jazyk Objective C je určen především pro práci s objekty; neobjektových rozšíření v něm proto mnoho nenalezneme. Ta, jež zde jsou, jsou však velmi přijemná. Prvým z nich je možnost používat komentáře `///  
"///  
" stejní jako v C++ (to již nepřímo vyplynulo z příkladů v minulém dílu, kde byly takové komentáře používány). Druhým je standardizace typu a hodnot pro logické (boolovské) proměnné: aniž by byl narušen standardní přístup jazyka C, slouží jako logický typ BOOL a odpovídající hodnoty jsou YES a NO. Standardní headery prostě definují`

```
typedef int BOOL;  
#define YES 1  
#define NO 0
```

případně v jazyce C ekvivalentní `typedef enum {NO, YES} BOOL`, jehož výhodou je, že konstanty `YES` a `NO` jsou známy i na úrovni debuggeru.

Velmi šikovným rozšířením je direktiva `#import`. Ta funguje téměř stejně, jako klasická direktiva `#include`; překladač ale zajistí, že každý zdrojový soubor se bude překládat *nejvýše jednou*. V Objective C si proto můžeme ušetřit nepohodlné obkládání každého hlavičkového souboru direktivami typu

```
#ifndef _STDIO_H_  
#define _STDIO_H_  
...  
#endif
```

Je snad trochu sporné, zda mezi neobjektová rozšíření můžeme řadit *nové typy, hodnoty a identifikátory*: kromě typů `id` a `Class` a hodnot `nil` a `Nil`, jež již známe z minulého dílu, nabízí Objective C následující typy a hodnoty:

### Typy:

`SEL` vnitřní reprezentace zprávy  
`IMP` metoda (přímý ukazatel na metodu, používaný pro statický přístup)

### Identifikátory

`id self` v implementaci metody reprezentuje objekt, který metodu zpracovává  
`id super` ditto, ale jeho metody jsou vyhledávány v rodičovské třídě  
`SEL _cmd` v implementaci metody reprezentuje zprávu, jež metodu vyvolala

Typ `SEL` reprezentuje zprávu a je definován jako celočíselná hodnota, na kterou je zpráva interně přeložena. Spolu s direktivou `@selector`, jež zprávy převádí na tento typ, umožňuje zprávy ukládat do proměnných, vzájemně porovnávat a podobně. Typ `IMP` vlastně není ničím jiným, než ukazatelem na funkci, a využívá se v těch zcela výjimečných případech, kdy potřebujeme volat metodu rychleji, než prostřednictvím mechanismu zpráv. Ukázky praktického použití naleznete v příkladech na CD; totéž platí i pro všechny ostatní konstrukce.

Poznamenejme, že pro dosažení statické typové kontroly srovnatelné s C++ nabízí Objective C možnost používat na místě typu `id` konstrukci "ukazatel na třídu" s významem "objekt dané třídy nebo jejího dědice". Je vhodné zdůraznit, že jde pouze o statickou, překladačovou kontrolu -- na výsledný program to nemá vůbec žádný vliv, ten bude fungovat stejně dobře (nebo stejně špatně) jako kdybychom všude důsledně používali `id`.

Díky tomu že `self`, `super` a `_cmd` jsou *identifikátory* a ne klíčová slova (jako je tomu např. v nedomyšleném C++), můžeme je bez jakýchkoli problémů předefinovat; překladač Objective C proto bez problémů přeloží 'obyčejný céčkový' program, ve kterém je použita např. proměnná jménem `self`.

# Přístup k proměnným

Proměnné objektu mohou být k dispozici pouze jeho vlastním metodám, nebo i metodám všech jeho dědiců, nebo -- ve výjimečných případech, kdy z nějakého důvodu musíme rezignovat na objektové programování a využívat statické programátorské techniky -- mohou být proměnné přístupné z jakéhokoli úseku kódu. Možnosti přístupu k proměnným jsou určeny použitím jedné ze tří direktiv:

```
@private    proměnné jsou přístupné pouze metodám objektu samotného;
@protected proměnné jsou přístupné i dědicům (tento přístup je standardní nepoužijeme-li žádnou z
direktiv);
@public     proměnné jsou přístupné komukoli.
```

Jestliže z nějakého důvodu musíme rezignovat na objektový přístup, můžeme také získat neomezený přístup k proměnným kteréhokoli objektu pomocí direktivy `@defs`.

## Kategorie

Primární účel kategorií je umožnit rozložení implementace jedné složité třídy do několika zdrojových souborů. Kategorie má interface i implementaci obdobně standardním, avšak na místě nadřazené třídy je jméno kategorie v závorkách. Kategorie samozřejmě nemůže definovat vlastní proměnné; má však volný přístup k proměnným, definovaným v základním rozhraní třídy.

Dejme tomu, že máme následující třídu:

```
@interface Xxx:Yyy
-aaa;
-bbb;
-ccc;
@end
```

včetně odpovídající implementace

```
@implementation Xxx
-aaa { ... }
-bbb { ... }
-ccc { ... }
@end
```

Pokud by pro nás bylo z jakéhokoli důvodu výhodné oddělit od sebe implementace těchto tří metod do samostatných celků, mohli bychom stejně dobře použít základní třídy a dvou kategorií -- z hlediska práce s třídou `Xxx` a jejími objekty by se nezměnilo vůbec nic:

```
@interface Xxx:Yyy // základní třída
-aaa;
@end
@interface Xxx (KategorieProMetoduB)
-bbb;
@end
@interface Xxx (AProMetoduCcc)
-ccc;
@end
```

Obdobně by samozřejmě byla rozdělena i implementace.

Kategorie navíc umožňují *doplňovat nebo mnit již existující třídy*: dejme tomu, že bychom chtěli, aby *libovolný objekt* dokázal reagovat na zprávu `where` jménem počítače, na kterém běží proces, v rámci něhož objekt existuje. V Objective C není nic jednoduššího -- prostě implementujeme kategorii

```
@interface NSObject (ReportWhere)
-(NSString*)where;
@end

@implementation NSObject (ReportWhere)
-(NSString*)where
{
    return [[NSProcessInfo processInfo] hostName];
}
```

```
}  
@end
```

Jakmile máme kategorii hotovou, můžeme novou službu zcela volně používat u kteréhokoli objektu.

## Protokoly

Protokol v zásadě není ničím jiným, než seznamem metod; používá se jako společný prvek pro specifikaci tříd, které mají mít společné metody, ale nejsou strukturálně příbuzné (čímž nahrazuje implementačně i programátorsky obtížnou vícenásobnou dědičnost C++ v tom jediném případě, kdy má jakýsi smysl).

Protokol je definován obdobně jako interface, nemůže však samozřejmě obsahovat proměnné. Protokoly však mohou mít svou vlastní dědičnost. Namísto direktivy `@interface` je zde použita direktiva `@protocol`. Konkrétní příklad naleznete opět na CD.

## Ostatní

Objective C nabízí ještě dvě direktivy, `@class` a `@encode`. Prvá z nich prostě informuje překladač o existenci třídy daného jména, a slouží pro dopředné reference:

```
@class Xxx;  
@interface Yyy  
- (Xxx*) xxx;  
@end  
@interface Xxx  
- (Yyy*) yyy;  
@end
```

Direktiva `@encode` slouží pro dynamickou specifikaci typu, a v praxi se téměř vůbec nepoužívá (protože plně objektový systém dynamické typy vlastně nepotřebuje -- namísto nich se používají objekty, jež si typovou informaci nesou implicitně v sobě); její podrobný popis si proto můžeme odpustit.

## Příklady

Pro lepší ilustraci je na CD řada bohatě komentovaných příkladů jednoduchých programů:

Příklad 1: využití předdefinovaných tříd (knihovny tříd NeXTstepu);  
Příklad 2: tvorba vlastní třídy;  
Příklad 3: dědičnost a vkládání objektů;  
Příklad 4: dynamické rozpoznání třídy;  
Příklad 5: skládání objektů a dynamické rozpoznání zpráv;  
Příklad 6: skládání objektů a dynamické rozpoznání zpráv -- jiná varianta;  
Příklad 7: mechanismus klient/server;  
Příklad 8: statický přístup k objektům.

## Shrnutí

Dokončili jsme stručný popis jazyka Objective C; ti, kdo mají jeho překladač k dispozici (jako GNU C je k dispozici na libovolné platformě, od Mac OS X přes všechny varianty unixu až po DOS či Windows) v něm mohou psát libovolné testovací programy.

Příští se už začneme bavit o skutečných vlastnostech prostředí Cocoa: ukážeme si mechanismus tvorby a zániku objektů a podobně.

-oc-