JScript .NET – programowanie. Biblia

Autor: Essam Ahmed
ISBN: 83-7197-666-6
Tytuł oryginału: JScript .NET Programming
Format: B5, stron: 416
oprawa twarda
Przykłady na ftp: 1244 kB

JScript .NET to nowe, oparte na technologiach obiektowych, wszechstronne narzędzie do budowy aplikacji platformy Microsoft .NET. Wyczerpujące omówienie metod pracy z obiektami i liczne przykłady czynią z niniejszej książki jedyny w swoim rodzaju przewodnik, prowadzący Czytelnika od podstaw języka, architektury obiektowej i specyfikacji UML aż do kompletnych aplikacji ASP.NET i usług Windows, wykorzystujących bazy danych, ADO.NET i XML.·

- Poznaj, co różni JScript .NET od języków JavaScript, Visual Basic i C++.
- Rozpocznij pracę z typami danych, funkcjami, operatorami i innymi podstawowymi elementami JScript .NET.
- Zadbaj o wydajność, wprowadzając mechanizmy obiektowe.
- Poznaj tajniki tworzenia usług typu Web Service, Windows Forms i klientów ASP.NET.
- Połam zęby na analizowaniu kodu i obsłudze błędów.
- Zapoznaj się z tajnikami architektury obiektowej i translacji diagramów UML
- Wykorzystaj posiadaną wiedzę do analizowania aplikacji ASP.NET, przeglądającej bazy danych za pośrednictwem ADO.NET.
- Pójdź dalej – odkryj zasady pracy usługi Windows, wykorzystującej ADO.NET do przetwarzania plików XML.
- Dowiedz się, jak przenosić aplikacje ASP na platformę .NET.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Rzut oka na książkę

Spis treści

Rozdział 9.
Co nowego w JScript .NET?

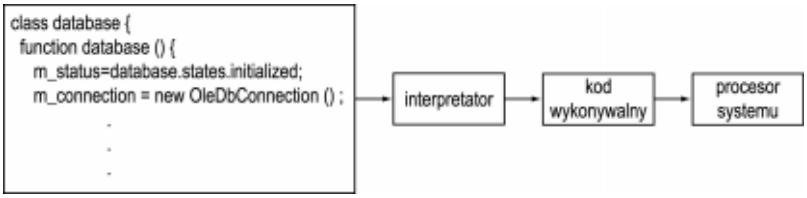
W tym rozdziale:

- JScript .NET i kompilowanie kodu
- Nowe typy danych
- Nowe instrukcje
- Dyrektywy kompilatora

JScript .NET wyposażony został w wiele nowych elementów — takich jak obsługa zróżnicowanej widoczności elementów klas, dziedziczenie i polimorfizm — które czynią z niego język obiektowy o ogromnych możliwościach. W tym rozdziale przedstawimy przegląd tych cech, opisując korzystanie z nowych typów danych, tworzenie klas i korzystanie z nowych dyrektyw kompilatora.

Kompilowanie kodu

Program w języku JScript uruchamiany jest w środowisku przetwarzania skryptów, którego rolę pełnią najczęściej: Internet Explorer, host skryptów systemu Windows (Windows Scripting Host) lub serwer internetowych usług informacyjnych (Internet Information Server). Środowisko takie przekształca kolejne wiersze kodu źródłowego do postaci kodu wykonywalnego, który zostaje przekazany głównej jednostce przetwarzania systemu. Taki sposób przetwarzania określa się terminem *interpretacja* — kod źródłowy odczytywany jest wiersz po wierszu, jednocześnie z jego wykonywaniem (patrz rysunek 9.1).



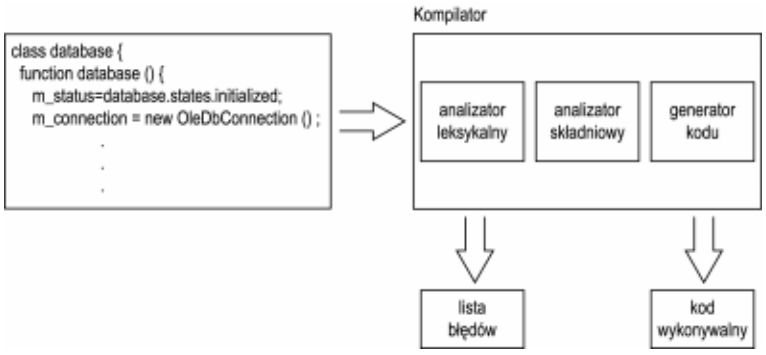
Rysunek 9.1. *Przetwarzanie programu JScript przez interpretator*

Interpretacyjny model wykonywania programów jest ułatwieniem — uruchomienie nie wymaga żadnych dodatkowych czynności. Wadą tego rozwiązania jest konieczność wykonywania programów w pewnym środowisku, nazywanym hostem lub środowiskiem hosta (host environment). Specyfikacja, której implementacją jest JScript (ECMAScript), nie precyzuje więc technik interakcji programów z użytkownikiem, wprowadzania i wyprowadzania danych, jak również interakcji z systemem. Owocem tego jest częste dostosowywanie kodu do pracy w pewnym szczególnym środowisku, np. przeglądarki Internet Explorer czy hosta skryptów Windows (Windows Scripting Host). Kolejną wadą modelu interpretacyjnego jest niższa wydajność programów interpretowanych niż programów „gotowych".

Ponieważ JScript .NET jest językiem kompilowanym, rozwiązuje wymienione wyżej problemy. Przed wykonaniem programu wymagana jest jego kompilacja. Tworzyć można samodzielne pliki wykonywalne (z rozszerzeniem *.exe*), pliki wykonywalne przeznaczone dla systemu Windows (pracujące w trybie konsoli bądź GUI) oraz biblioteki DLL.

Kompilator JScript .NET to program, który przekształca kod źródłowy na kod wykonywalny. Rysunek 9.2 ilustruje przebieg takiego przetwarzania. Podobnie jak interpretator, kompilator odczytuje kod źródłowy i dzieli na części (elementy składniowe): słowa kluczowe, operatory i znaki przestankowe. Wynikiem pracy kompilatora są dwa strumienie danych: lista błędów i skompilowany (wykonywalny) program. Lista błędów powstaje w wyniku analizy składniowej kodu, połączonej z wyszukiwaniem błędów i miejsc, w których niewłaściwie użyto zmiennych. Generowanie kodu i tworzenie pliku EXE lub DLL to ostatnie etapy pracy kompilatora.

Rysunek 9.2. *Przetwarzanie programu JScript przez kompilator*



W przeciwieństwie do tradycyjnych kompilatorów, które generują binarny kod wykonywalny, przeznaczony do przetwarzania przez procesory takie jak Intel Pentium, wszystkie kompilatory .NET tworzą kod w języku nazywanym Intermediate Language (IL), czyli języku pośrednim. Kod taki zostaje zapisany do pliku, a ostateczna konwersja na właściwy kod wykonywalny następuje dopiero po uruchomieniu programu. Element wykonujący ostateczną konwersję to kompilator Just-In-Time (JIT, „dokładnie w porę"). Ze względu na jego integrację z biblioteką CLR, jego praca pozostaje zazwyczaj niewidoczna.

Visual Studio .NET nie ma wbudowanej obsługi projektów JScript .NET ani kompilatora JScript .NET. Nowe środowisko programowania wykorzystywać można do tworzenia kodu źródłowego JScript .NET i usuwania błędów, ale samą kompilację inicjujemy z poziomu wiersza poleceń.

Kompilator JScript .NET nosi nazwę *jsc.exe*. Umieszczony zostaje w folderze *Microsoft.NET\Framework\version* (ostatni element ścieżki to numer zainstalowanej wersji infrastruktury .NET Framework). Przy instalowaniu .NET Framework SDK lub Visual Studio .NET odpowiednia nazwa katalogu dopisywana jest do zmiennej środowiskowej `path` systemu. Umożliwia to uruchamianie kompilatora JScript .NET z dowolnego katalogu systemu plików.

Jeżeli uruchomienie *jsc.exe* sprawia problemy, należy spróbować wywołać kompilator z wiersza poleceń Visual Studio .NET. Visual Studio .NET tworzy nowy skrót do wiersza poleceń, zapewniający dołączenie nazwy katalogu kompilatorów .NET do zmiennej środowiskowej `path`. Skrót ten znajdziemy w menu *Start/Microsoft Visual Studio.NET/Visual Studio.NET Tools/Visual Studio.NET Command Prompt*.

Kompilator pozwala korzystać ze znacznej liczby opcji, ułatwiających analizowanie programu, kompilację warunkową i optymalizowanie kodu wynikowego. Opcje te wprowadzamy w wierszu poleceń kompilacji programu. Najczęściej wykorzystuje się jedynie mały podzbiór dostępnych opcji. Gdy pojawia się potrzeba skorzystania z funkcji bardziej zaawansowanych, sięgamy po szczegółowe opisy do dokumentacji MSDN.

Nowe typy danych

Język JScript .NET, oparty na systemie CTS (Common Type System, wspólny system typów), wprowadza kilka nowych typów danych. Opisujemy je, wraz z mapowaniami do typów CTS, w tabeli 9.1.

Szerzej piszemy o CTS w rozdziale 2.

Precyzowanie typów danych

W języku JScript instrukcja `var` deklaruje zmienną beztypową. Zmiennym tego rodzaju można przypisywać dowolny typ danych. JScript konwertuje (przeprowadza koercję) zmienną do odpowiedniego typu na podstawie sposobu jej wykorzystania w programie (kontekst zmiennej).

JScript .NET wprowadza ideę zmiennych o określonych typach danych, gdzie programista deklaruje wraz z samą zmienną jej typ. Rozwiązanie takie charakteryzują następujące zalety:

- *Wyższa wydajność*. Biblioteka wykonawcza JScript .NET nie musi zmieniać typu zmiennej w zależności od jej zastosowania, co skraca czas jej pracy.

Tabela 9.1. *Nowe typy danych języka JScript .NET*

Typ JScript .NET	Typ CTS	Opis
`boolean`	`System.Boolean`	Wartość logiczna (`false` lub `true`). W odróżnieniu od języka JScript, typ wartościowy, a nie odniesieniowy.
`byte`	`System.Byte`	8-bitowa liczba całkowita bez znaku, z zakresu od 0 do +255.
`char`	`System.Char`	16-bitowa liczba całkowita bez znaku, z zakresu od 0 do +65 535.
`decimal`	`System.Decimal`	96-bitowa liczba zmiennoprzecinkowa ze znakiem, z zakresu od –79,2e27 do +79,2e27.
`double`	`System.Double`	64-bitowa liczba zmiennoprzecinkowa podwójnej precyzji, z zakresu od –1,7e308 do +1,7e308. Odpowiednik typu `Number` języka JScript.
`float`	`System.Single`	32-bitowa liczba zmiennoprzecinkowa pojedynczej precyzji, z zakresu od –3,4e38 do +3,4e38.
`int`	`System.Int32`	32-bitowa liczba całkowita ze znakiem, z zakresu od –2 147 483 648 do +2 147 483 647.
`long`	`System.Int64`	64-bitowa liczba całkowita ze znakiem, z zakresu od –9 223 372 036 854 755 808 do +9 223 372 036 854 755 807.
`sbyte`	`System.SByte`	8-bitowa liczba całkowita ze znakiem z zakresu od –128 do +127.
`short`	`System.Int16`	16-bitowa liczba całkowita ze znakiem, z zakresu od –32 768 do +32 767.
`String`	`System.String`	Ciąg znaków Unicode o długości do 2 miliardów znaków.
`uint`	`System.UInt32`	32-bitowa liczba całkowita bez znaku z zakresu od 0 do +4 294 967 295.
`ulong`	`System.UInt64`	64-bitowa liczba całkowita bez znaku z zakresu od 0 do +184 467 440 737 095 551 615.
`ushort`	`System.UInt16`	16-bitowa liczba całkowita bez znaku z zakresu od 0 do +65 535.

- *Sprawdzanie typów w czasie kompilacji*. Można wykryć i zlokalizować nieprawidłowe konwersje typów przed uruchomieniem aplikacji.
- *Większa przejrzystość kodu*. Inny programista łatwiej zrozumie działanie kodu, dysponując dodatkową wskazówką w postaci rodzaju danych, do których przechowywania można wykorzystać zmienne.

Wydruk 9.1 przedstawia kilka przykładów deklarowania zmiennych o określonym typie w języku JScript .NET. Składnia jest stosunkowo prosta i oparta na instrukcji `var` języka JScript. W deklaracji zmiennej, po dwukropku, podawana jest nazwa typu, decydująca o możliwościach wykorzystania identyfikatora.

Wydruk 9.1. *Deklarowanie zmiennych o określonym typie w języku JScript .NET*

```
var intValue : int;
var longValue : long;
var floatValue : float;
var dblValue : double;
var stringValue : String;
```

Ponieważ język JScript korzysta ze zmiennych beztypowych, kod przedstawiony na wydruku 9.2 jest jak najbardziej poprawny. Biblioteka wykonawcza JScript przekształca zmienną z typu ciąg na typ liczbowy w trakcie wykonywania programu, zapewniając możliwość wykonania instrukcji przypisania.

Wydruk 9.2. *Przykład niejawnych konwersji typów w języku JScript*

```
var j;
j = "ten";  // ok
j = 10;  // ok
```

JScript .NET nie pozwoli na skompilowanie takiego fragmentu. Kompilator generuje wówczas błąd `Type mismatch`. Nie oznacza to jednak, że język JScript .NET nie wykonuje konwersji niejawnych — przy wyprowadzaniu danych liczbowych do interfejsu użytkownika (na przykład) nie jest konieczne wprowadzanie dodatkowych instrukcji zmiany typu.

W jednym wierszu zadeklarować można wiele zmiennych tego samego typu. Ich nazwy rozdzielamy przecinkami, a po ostatniej podajemy nazwę typu. Jest to rozwiązanie proste i intuicyjne.

Zmienne o typach określonych i nieokreślonych muszą mieć różne nazwy. Innymi słowy, jeżeli zadeklarujemy zmienną `index` o określonym typie, nie można w innym miejscu tej samej funkcji umieścić deklaracji zmiennej o tej samej nazwie, ale bez podania typu. Nie można również zmienić typu zmiennej. W całym zakresie zmiennej jej typ musi pozostać taki sam.

Nowe instrukcje

W języku JScript .NET wprowadzono następujące nowe instrukcje:

- `class`,
- `interface`,
- `import`,
- `package`,
- `const`,
- `enum`.

Instrukcja class

Słowo kluczowe języka JScript .NET `class` umożliwia tworzenie obiektów pierwszego rzędu, w pełni korzystających z ułatwień oferowanych przez język i system operacyjny. Wydruk 9.3 przedstawia przykład definicji prostej klasy.

Wydruk 9.3. *Prosta klasa JScript .NET*

```
class demoClass
{
  // konstruktor
  function demoClass()
  {
    Console.WriteLine("Wywołano konstruktor");
  }

  function sampleMethod()
  {
    Console.WriteLine("Hello z demoClass.sampleMethod!");
  }
}
// użycie klasy
var dc = new demoClass;
dc.sampleMethod();
```

Widzimy tu implementację minimalną, zawierającą konstruktor i jedną metodę.

Konstruktor klasy to funkcja składowa nazwana tak samo, jak sama klasa. Na wydruku 9.3 konstruktorem jest funkcja `demoClass`. Konstruktor jest wywoływany automatycznie w chwili tworzenia nowej klasy przy użyciu operatora `new`. Może zostać wykorzystany do inicjowania zmiennych składowych i pozyskiwania wymaganych przez obiekt zasobów.

Operacje wykonywane przez klasę, nazywane *zachowaniami* (*behaviors*), realizowane są przy użyciu funkcji, określanych terminem *metody* (*methods*). Klasa `demoClass` ma jedną metodę — `sampleMethod`.

Klasę można traktować jako specyfikację obiektu. Klasa jest pewnym typem, a obiekt — egzemplarzem tego typu. Przy tworzeniu klasy opisujemy zachowania i dane tworzonych na jej podstawie obiektów. Zachowania reprezentowane są przez publiczne funkcje składowe klasy. Dane obiektu określane są przez właściwości klasy lub jej publiczne zmienne składowe. Całość składa się na strukturę obiektu.

Elementy klasy

Definiowane wewnątrz klasy funkcje decydują o zachowaniu tworzonych obiektów; określa się je terminem *funkcje składowe* (*member functions*). Zmienne definiowane wewnątrz klasy decydują o przechowywanych przez obiekty danych; są to *zmienne składowe* (*member variables*). Domyślnie elementy składowe klasy widoczne są dla wszystkich innych elementów aplikacji (klas i funkcji). Oznacza to, że wszelkie klasy i funkcje aplikacji mają nieograniczony dostęp do struktury obiektu. Wynika z tego, że programista może zaprojektować aplikację, której działanie zależy od implementacji obiektu, a nie jego metod. Elementy składowe klasy mogą mieć jednak zróżnicowane zakresy widoczności. Element może być publiczny, chroniony, prywatny lub wewnętrzny.

Funkcje i zmienne składowe różnią się od funkcji i zmiennych pozostających poza definicjami klas. Funkcje i zmienne nie należące do żadnej klasy mają zakres *globalny* (*global*), co oznacza, że pozostają widoczne dla wszystkich innych funkcji i klas aplikacji.

Elementy publiczne

Elementy *publiczne* (*public*) to odpowiedniki zmiennych i funkcji globalnych. Wszystkie składniki aplikacji mogą z nich bezpośrednio korzystać i wykonywać na nich operacje. U źródeł ogromnego znaczenia klas w programowaniu leży jednak możliwość oddzielenia ich implementacji (struktury wewnętrznej) od interfejsu (struktury widocznej publicznie). Oznacza to, że implementacja może w znacznym stopniu różnić się od interfejsu. W przykładowej klasie `demoClass` publiczna metoda `sampleMethod` mogłaby zwracać wartość zmiennej wewnętrznej lub wywoływać wewnętrzną funkcję składową, generującą zwracany przez metodę ciąg. Użytkownik obiektu nie musi wiedzieć, jak klasa została zaimplementowana (i często nie jest tym zainteresowany).

Elementy chronione i prywatne

Klasa oddziela interfejs od implementacji, stosując elementy składowe *chronione* (*protected*) lub *prywatne* (*private*) — niewidoczne lub widoczne dla innych obiektów jedynie częściowo. Tego rodzaju technikę określa się terminem *kapsułkowanie* (*encapsulation*). Klasa może zostać tak obudowana, że wszelkie jej klienty (użytkownicy klasy) polegać będą wyłącznie na udostępnianych przez nią metodach (zachowaniach). Zapewnia to projektantowi klasy pełen komfort modyfikowania jej implementacji.

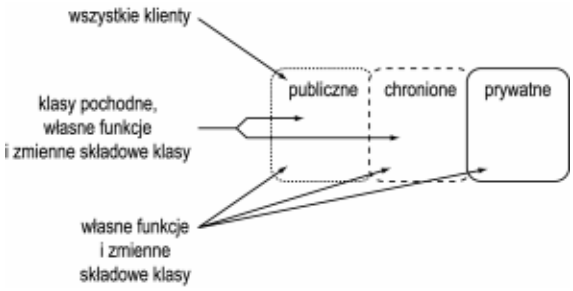
W przeciwieństwie do elementów prywatnych, których widoczność nie wykracza poza ramy jednej klasy, elementy chronione widoczne są dodatkowo dla klas *pochodnych* (*derived*). Wydruk 9.4 jest prostą ilustracją wykorzystania chronionej funkcji składowej klasy bazowej przez klasę pochodną.

Wydruk 9.4. *Użycie chronionego elementu składowego przez klasę pochodną*

```
class vehicle {
  /* inne elementy pomijamy */
  protected function _moveTo(p:globalPoint)
  { /* implementację pomijamy */ }
}

class scooter extends vehicle {
  function goToLocation(p:globalPoint) {
    _moveTo(p); // wywołanie metody _moveTo() klasy vehicle
  }
}
```

Element `_moveTo()` jest widoczny dla klasy `scooter`, ale pozostaje niedostępny dla innych klas i funkcji. Na rysunku 9.3 przedstawiamy zestawienie informacji o widoczności elementów klas.



Rysunek 9.3. *Widoczność elementów klas*

Dziedziczenie elementów innej klasy

Klasa (określana tu jako *klasa pochodna — deriving class*) może dziedziczyć elementy innej klasy (określanej jako *klasa bazowa — base class*), modyfikując jej metody i dodając nowe składowe funkcje publiczne lub (i) prywatne. Stosuje się w tym celu słowo kluczowe `extends`. Przykład tworzenia klasy pochodnej przedstawia wydruk 9.5. Słowo `extends` zostało wyróżnione pogrubionym drukiem.

Wydruk 9.5. *Tworzenie klasy pochodnej*

```
class vehicle {
  var name: String;
  var maxKgLoad: double;
  var latitude: double;
  var longitude: double;

  public function vehicle() { clear(); }

  public function clear() {
    name="pojazd";
    maxKgLoad=Number.POSITIVE_INFINITY;
    latitude = 0; longitude = 0; }
} // vehicle - koniec

class bicycle extends vehicle
{
  function bicycle() {
    // nazwa to publiczna zmienna składowa klasy vehicle
    name="rower";
    // maxKgLoad to publiczna zmienna składowa klasy vehicle
    maxKgLoad=100;
  }
  // redefiniowanie metody clear() klasy vehicle
  public function clear() {
    clear(); // wywołanie metody klasy vehicle
    name="rower";
    maxKgLoad=100;
  }

  // nowa funkcja składowa
  public function visitFriend() {
    // latitude i longitude to zmienne składowe klasy vehicle
    latitude=43.41;
    longitude=79.38;
  }
} // bicycle - koniec
```

W przedstawionym przykładzie klasa `bicycle` tworzona jest na podstawie klasy `vehicle`. Zmieniona została definicja metody `clear()` i dodana została metoda `visitFriend()`. Podobnie można modyfikować i uzupełniać metody dowolnej tworzonej samodzielnie klasy. Nie można jednak wykorzystywać jako klas bazowych klas systemowych, takich jak `System.String`.

Zabezpieczanie przed modyfikowaniem metod klasy

Słowo kluczowe `final` uniemożliwia redefiniowanie (zastępowanie) funkcji składowych w klasach pochodnych. W poniższym przykładzie definiujemy metodę `public final` o nazwie `driveTo`. Klasa pochodna może z niej korzystać, ale nie ma możliwości zastąpienia jej definicji:

```
class vehicle {
  public final function driveTo(place : Destinastion) {
    /* implementację pomijamy */
  }
  // inne elementy pomijamy
}
```

Wprowadzenie słowa kluczowego `final` uniemożliwi zmodyfikowanie funkcji składowej w klasach pochodnych.

Polimorfizm

Polimorfizm korzysta z podobieństw pomiędzy obiektami opartymi na wspólnej klasie bazowej. Wydruk 9.6 przedstawia przykładowe klasy wyprowadzone z klasy `vehicle`:

Wydruk 9.6. *Przykład hierarchii klas*

```
class truck extends vehicle { /* ... */ }
class van extends vehicle { /* ... */ }
class tricycle extends vehicle { /* ... */ }
class scooter extends vehicle { /* ... */ }
```

Pewna część aplikacji tworzy egzemplarze tych klas (innymi słowy, tworzy obiekty), a inna — wykonuje na nich operacje. Załóżmy, że dysponujemy funkcją, która wyświetla informacje o obiektach `scooter`, taką jak przedstawiona na wydruku 9.7.

Wydruk 9.7. *Funkcja printScooterDetails*

```
function printScooterDetails( s: scooter)
{
  Console.WriteLine("Typ\t{0}\n ładowność\t{1}",s.name,s.maxKgLoad);
}
// tworzymy obiekt scooter i wyświetlamy informacje o nim
var scooterObject=new scooter;
printScooterDetails(scooterObject);
```

Przedstawiona funkcja pobiera obiekt `scooter` jako parametr i wyprowadza na ekran informacje o nim. Funkcja działa poprawnie, ale dla każdego rodzaju pojazdu (`truck`, `van`, `tricycle`, `scooter`) wymagana będzie jej odrębna wersja. Poza oczywistym faktem, że

wszystkie one będą podobne, każda modyfikacja projektu o nowy typ pojazdu wymagać będzie pamiętania o tym, aby utworzyć kolejną funkcję tego rodzaju. W tym momencie skorzystać możemy z polimorfizmu. Ponieważ wszystkie klasy utworzone zostały na bazie klasy `vehicle`, każdy z obiektów może zostać potraktowany jako egzemplarz typu `vehicle` (patrz wydruk 9.8).

Wydruk 9.8. *Funkcja pobierająca typ vehicle*

```
function printDetails(v:vehicle)
{
  Console.WriteLine("Typ\t{0}\n Ładowność\t{1}",v.name,v.maxKgLoad);
}
// tworzymy pojazdy i wyświetlamy informacje o nich
var t : truck = new truck;
var b : bicycle = new bicycle;
var myVan : van = new van;
printDetails(t);
printDetails(b);
printDetails(myVan);
```

Rozwiązanie wykorzystane w funkcji `printDetails` jest znacznie prostsze niż w przypadku funkcji `printScooterDetails` i jej kolejnych odmian. Wszystkie obiekty traktowane są jako egzemplarze tej samej klasy. Na tym nie kończą się możliwości polimorfizmu. Możemy skorzystać z niego przy deklarowaniu obiektów. Ilustruje to wydruk 9.9.

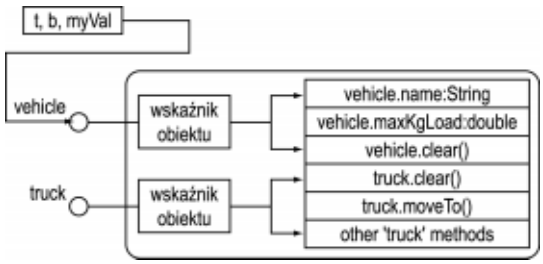
Wydruk 9.9. *Tworzenie egzemplarzy klas pochodnych, deklarowanych jako typ bazowy*

```
var t,b,myVan : vehicle;
// Uwaga! t, b i myVal zostały zadeklarowane jako obiekty typu vehicle
t = new truck;
b = new bicycle;
myVan = new van;
```

Zaprezentowane tu rozwiązanie opiera się na możliwości traktowania wszystkich obiektów utworzonych na bazie `vehicle` identycznie. Pozwala to ujednolicać kod programu i ułatwia jego ponowne wykorzystywanie.

Ponieważ nowe obiekty zadeklarowane zostały jako egzemplarze klasy `vehicle`, kompilator zezwoli na dostęp wyłącznie do metod klasy `vehicle`. Nie będzie można korzystać z metod żadnej z podklas. Rysunek 9.4 ilustruje sposób traktowania takiej sytuacji na przykładzie klas `vehicle` i `truck`.

Rysunek 9.4. *Obiekty vehicle i truck w pamięci*



Jeżeli podejmiemy próbę wywołania jednej z nie dziedziczonych funkcji składowych klasy `truck`, kompilator poinformuje nas, że element składowy nie istnieje. Mimo iż wiemy, że obiekt jest faktycznie typu `truck`, a jedynie zadeklarowany został jako `vehicle`, kompilator zachowuje się dość „biurokratycznie" — liczy się deklaracja. Można jednak przekonać kompilator, że jest inaczej. Wykonujemy w tym celu operację określaną terminem *rzutowanie* (*cast*), pokazaną na wydruku 9.10.

Wydruk 9.10. *Rzutowanie typu obiektu*

```
// uwaga: truckObject będzie zadeklarowany jako vehicle
var truckObject : vehicle = new truck;
// truckObject jest obiektem klasy truck, zadeklarowanym jako vehicle
truckObject.name = "ciężarówka1"; // OK - name jest elementem klasy vehicle
truckObject.moveTo(/*...*/); // błąd - kompilator nie rozpozna metody
// korekta: rzutujemy obiekt, żeby wykazać, że jest klasy truck
truck(truckObject).moveTo(/*...*/);
// OK - klasa truck ma element moveTo
```

Ponieważ rzutowanie nie prowadzi do trwałych zmian w obiekcie, przedstawiona składnia musi być stosowana przy każdym wywołaniu jego metod. Sposobem uniknięcia takiej komplikacji może być jedynie wprowadzenie dodatkowej zmiennej. Wskazuje ona ten sam egzemplarz obiektu, reprezentując go jednak jako podklasę `truck`, a nie klasę bazową `vehicle` (patrz wydruk 9.11).

Wydruk 9.11. *Zastąpienie rzutowania dodatkową zmienną*

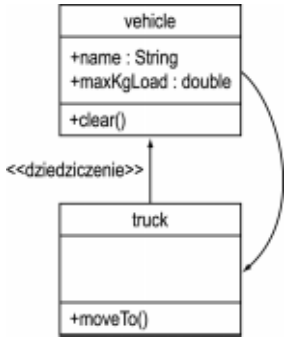
```
var truckObject : vehicle = new truck;
// truckObject jest obiektem klasy truck, zadeklarowanym jako vehicle
truckObject.name = "truck"; // OK - name jest elementem klasy vehicle
var truckNessOfObject : truck;
// w kolejnym wierszu przeprowadzamy rzutowanie truckObject do klasy truck
truckNessOfObject = truckObject;

truckNessOfObject.moveTo(/*...*/); // OK - klasa truck ma element moveTo
truckNessOfObject.name = "ciężarówka2";
// OK - name jest elementem klasy truck, utworzonej na bazie klasy vehicle
```

Zwróćmy uwagę na to, że elementy klasy `vehicle` pozostają dostępne poprzez zmienną zadeklarowaną jako `truck`. Ogólną zasadą jest bowiem dziedziczenie przez klasę pochodną wszystkich publicznych i chronionych elementów klasy bazowej.

Wykonywanie rzutowania jest wymagane tylko wtedy, gdy korzystamy z elementów stojących *niżej* w hierarchii niż kontekst bieżący (patrz rysunek 9.5).

Przedstawione rzutowanie nie zawsze jest bezpieczne, ponieważ nie dla każdego obiektu klasy bazowej może zostać wykonane. JScript .NET generuje wówczas błąd czasu wykonywania. Przyczyną tego rodzaju błędu jest rzutowanie obiektu klasy bazowej do klasy pochodnej, gdy dany obiekt nie należy do podklasy.



Rysunek 9.5.
Rzutowanie do niższych poziomów hierarchii klas

Elementy statyczne

Elementy statyczne (*static members*) to części klasy, ale nie obiektu. Oznacza to, że nie są powiązane z konkretnym egzemplarzem. Innymi słowy, niezależnie od liczby obiektów zawsze istnieje dokładnie jeden element składowy statyczny danej klasy, nawet wówczas, gdy żaden egzemplarz obiektu nie został utworzony (w jednej klasie może oczywiście istnieć wiele elementów statycznych).

Do czego to potrzebne? Istnieją aplikacje, wymagające informacji o klasie jeszcze przed utworzeniem jej egzemplarzy. Inne z kolei wymagają trwałych elementów danych, dostępnych niezależnie od wywołań klasy.

Słowo kluczowe static różni się działaniem, zależnie od tego, czy użyte zostanie w odniesieniu do zmiennych, czy funkcji składowych. W przypadku modyfikowania za jego pomocą deklaracji funkcji składowej funkcja może być wywoływana bez konieczności tworzenia egzemplarza klasy (tj. bezpośrednio, za pośrednictwem nazwy klasy). W przypadku gdy słowo `static` zostanie użyte w deklaracji zmiennej składowej, zmienna będzie współużytkowana przez wszystkie egzemplarze — wartość elementu będzie taka sama dla wszystkich obiektów klasy.

Współużytkowanie statycznej zmiennej składowej przez wszystkie egzemplarze klasy może prowadzić do pewnych problemów związanych z projektowaniem aplikacji, ponieważ każdy obiekt może przeprowadzać jej odczyt lub zapis w dowolnej chwili. Rozwiązaniem pozwalającym na uniknięcie potencjalnych konfliktów jest nieujawnianie statycznych zmiennych składowych klasy i udostępnienie wartości jako właściwości tylko-do-odczytu. Inną możliwością jest zapewnienie dostępu do zmiennej za pośrednictwem statycznych funkcji składowych.

Inicjatory statyczne

Inicjator statyczny (*static initializer*) to specjalny blok instrukcji, wykonywanych przy pierwszym odwołaniu do klasy lub jej elementu. Wewnątrz inicjatora statycznego używać można wyłącznie statycznych zmiennych i funkcji składowych (tylko elementy statyczne można wywołać bez odniesienia do egzemplarza). Wydruk 9.12 przedstawia sposób definiowania inicjatora statycznego.

Wydruk 9.12. *Korzystanie z inicjatora statycznego*

```
class staticDemo {
  static function funcOne() { /* ... */ }
  static function funcTwo() { /* ... */ }

  static {
    // Obie powyższe funkcje _muszą_ zostać wywołane,
    // aby skonfigurować klasę (ustalić stan początkowy)
    funcOne();
    funcTwo();
  }
}
```

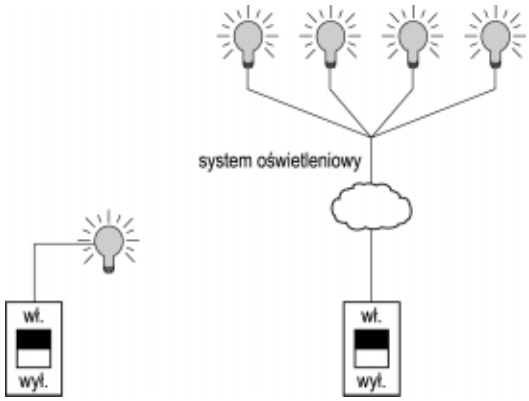
Funkcje `funcOne` i `funcTwo` zostają wywołane przy pierwszej próbie dostępu do dowolnego elementu statycznego klasy `staticDemo`. Jest to jedyne wywołanie instrukcji inicjatora statycznego w trakcie całej sesji pracy aplikacji.

Klasy abstrakcyjne

Przy pracy nad pewnymi rozwiązaniami pojawia się czasem potrzeba, aby klasy ujawniały pewien wspólny interfejs, różniąc się zarazem implementacjami. Znajduje to zastosowanie w przypadkach, gdy dążymy do zunifikowania interfejsu klas (jako uproszczenia dla ich użytkowników). Interfejs taki często uogólnia część metod klas jako metody wyższego poziomu (zachowania zagregowane), z których korzystanie jest bardziej intuicyjne i prostsze. Koncepcję tę można porównać do wyłącznika światła, jako interfejsu znacznie bardziej złożonego podsystemu oświetleniowego.

Wyłącznik światła może sterować pracą pojedynczej żarówki w jednym pokoju lub też setek żarówek w wielu pokojach. Niezależnie od ich liczby, interfejs pozostaje zawsze ten sam — wyłącznik ukrywa więc złożoność obsługiwanego systemu oświetleniowego, sprawiając, że korzystanie z niego jest intuicyjne i łatwe (patrz rysunek 9.6).

Rysunek 9.6. *Klasy ukrywające złożone systemy pod prostym interfejsem*



Oto przykład bardziej praktyczny: rozważmy klasy `vehicle`, `truck`, `car` i `scooter`. Klasa `vehicle` może reprezentować pewien ogólny pojazd, o standardowej nazwie i ładunku maksymalnym. Alternatywą dla takiego rozwiązania jest zdefiniowanie prostego interfejsu, którego implementacji nie ujmuje się w klasie bazowej, ale wymaga się od jej specjalizacji (patrz wydruk 9.13).

Wydruk 9.13. *Użycie klas abstrakcyjnych w celu utworzenia prostego, jednolitego interfejsu*

```
abstract class vehicle
{
  public abstract function getName() : String;
  public abstract function setName(newame : String) : void;
  public abstract function getMaxKgLoad() : int;
  public abstract function setMaxKgLoad(newame : int) : void;
}
```

Określenie zwracanego przez funkcję typu jako `void` oznacza, że funkcja faktycznie nie zwraca żadnej wartości.

Klasa abstrakcyjna ma jedną lub więcej funkcji składowych, które *nie mają* implementacji. Oznacza to, że *nie można bezpośrednio tworzyć egzemplarzy klasy abstrakcyjnej*. Funkcje składowe bez implementacji muszą zostać zrealizowane w klasach pochodnych. Pozostałe funkcje podlegają standardowemu dziedziczeniu i mogą być zastępowane. Klasa abstrakcyjna może więc zawierać zarówno funkcje z implementacją, jak i bez (w C++ nazywane elementami czysto wirtualnymi). Tworzenie funkcji bez implementacji jest proste — tworzymy funkcję, pomijając definicję jej zawartości. Wydruk 9.14 przedstawia tworzenie klasy pochodnej na bazie klasy abstrakcyjnej.

Wydruk 9.14. *Korzystanie z abstrakcyjnej klasy bazowej*

```
// klasa vehicle jest klasą abstrakcyjną
class scooter extends vehicle
{
  public function getName() : String { /*...*/ }
  public function setName(newName : String) : void { /*...*/ }
  public function getMaxKgLoad() : int { /*...*/ }
  public function setMaxKgLoad(newLoad : int) : void { /*...*/ }
  // inne funkcje i zmienne składowe...
}
```

Klasa `scooter` ujawnia jednolity interfejs klasy `vehicle`, zawierając zarazem wyspecjalizowane implementacje funkcji składowych. Użycie abstrakcyjnej klasy bazowej pozwala uniknąć niepotrzebnego dziedziczenia zastępowanych implementacji. Co więcej, wykluczona zostaje możliwość, że programista omyłkowo nie zdefiniuje wymaganych elementów i będą one przetwarzane w sposób określony w klasie ogólnej.

Słowo kluczowe `abstract` może również zostać wykorzystane do określenia jako abstrakcyjnej pojedynczej funkcji (w odróżnieniu od definicji abstrakcyjnej klasy). Należy wówczas zachować pewną ostrożność — klasa zawierająca co najmniej jedną funkcję abstrakcyjną nie może posłużyć do tworzenia egzemplarzy. Wybór pomiędzy zastosowaniem funkcji abstrakcyjnych i klas abstrakcyjnych należy do programisty. Funkcja abstrakcyjna *musi* zostać zaimplementowana w klasach pochodnych i może być wywoływana bezpośrednio poprzez typ bazowy, bez wykonywania operacji rzutowania (cast). Wydruk 9.15 przedstawia definiowanie abstrakcyjnej funkcji składowej.

Wydruk 9.15. *Tworzenie abstrakcyjnej funkcji składowej*

```
class vehicle {
  public abstract function getName() : String { }
  public function setName(newName:String) : void { /* ... */ }
  // dalsze elementy składowe klasy...
}
```

Klasy abstrakcyjne mogą zawierać funkcje składowe abstrakcyjne (choć nie muszą), jak również funkcje z implementacjami.

Tworzenie interfejsów

Inna technika pracy z klasami abstrakcyjnymi polega na wprowadzeniu interfejsów. Podobnie jak w przypadku klas abstrakcyjnych, nie można bezpośrednio utworzyć egzemplarza interfejsu — wszystkie jego metody są funkcjami abstrakcyjnymi (pozbawionymi implementacji). Elementy interfejsu implementowane są w klasach pochodnych. Ułatwia to wymuszenie implementacji wszystkich niezbędnych funkcji obiektów. Przy pracy z interfejsami wykorzystujemy słowa kluczowe `interface` i `implements`:

```
interface IErrorInfo {
  public function get Message() : String;
  public function get Number() : ulong;
}
class myError implements IErrorInfo {
  private var myErrorMessage : String;
  private var myErrorNumber : ulong;
  public function get Message() : String {
    return myErrorMessage; }
  public function get Number() : ulong {
    return myErrorNumber; }
}
```

Definiujemy tutaj interfejs `IErrorInfo` (powszechnie stosowana jest konwencja rozpoczynania nazw interfejsów od litery „I”), wykorzystywany następnie do utworzenia klasy `myError`.

Przy tworzeniu nowej klasy pochodnej przy użyciu słowa kluczowego `extends` użyta może zostać tylko jedna klasa bazowa. Gdy stosujemy słowo kluczowe `implements`, wskazać możemy dowolną liczbę interfejsów (które posłużą jako bazowe dla klasy tworzonej), o ile tylko zaimplementujemy wszystkie wymagane metody.

Zastępowanie i ukrywanie funkcji składowych klasy bazowej

Gdy wykorzystujemy klasę bazową, stosując słowo `extends`, klasa pochodna dziedziczy funkcje i zmienne składowe. Jak pisaliśmy już wcześniej, elementy klasy bazowej mogą być w jej klasach pochodnych definiowane ponownie. Domyślnie, jeżeli klasa pochodna implementuje funkcję składową o tej samej nazwie i argumentach (*sygnaturze funkcji*) co funkcja klasy bazowej, implementacja bazowa ulega *zastąpieniu* (*override*).

JScript .NET pozwala korzystać ze słowa kluczowego `overrides` w celu zwiększenia przejrzystości kodu (mimo że nie zmienia ono standardowego traktowania funkcji).

Może jednak pojawić się sytuacja, w której zastępowanie funkcji składowej klasy bazowej nie jest pożądane. Korzystamy wówczas ze słowa kluczowego `hide`, ukrywającego funkcję składową klasy pochodnej i powodującego, że wykorzystywana będzie funkcja klasy bazowej.

Właściwości klasy

Jedną z korzyści, wynikających z użycia klas, jest hermetyzacja (kapsułkowanie) struktury i zachowań. Może pojawić się potrzeba bezpośredniego ujawnienia zmiennej składowej klasy, jak w przykładzie:

```
class vehicle {
  public var name : String;
}
```

Mimo że jest to technika prosta w implementacji i użyciu, ma swoje wady. Ujawnienie zmiennej składowej klasy jest ujawnieniem części jej struktury. Gdy pojawi się potrzeba zmiany nazwy lub typu zmiennej, wpłynie to na funkcjonowanie użytkowników klasy — zmiana nazwy lub typu musi znaleźć swoje odzwierciedlenie w odpowiednich miejscach programów. Oznacza to, że hermetyzacja funkcji klasy nie jest pełna.

Problem ujawniania struktury klasy pojawił się już wcześniej w językach C++ i Visual Basic. Jako rozwiązanie wprowadzona została koncepcja właściwości klasy. *Właściwość* to funkcja składowa pełniąca rolę zmiennej, która izoluje użytkownika klasy od zmian w nazwach i typach zmiennych składowych. W programowaniu w językach C++ i Visual Basic przyjęła się dodatkowo konwencja nadawania właściwościom klas nazw znaczących i poprzedzania ich słowem `set` lub `get`. Jest to oczywiście odniesienie do modyfikowania i odczytywania wartości. We wcześniejszych przykładach deklarowane były w ten sposób funkcje składowe, służące od odczytywania i zapisywania właściwości `name` i `maxKgLoad`.

Takie użycie funkcji składowych do ukrycia zmiennych składowych jest rozwiązaniem efektywnym, prowadzącym do korzystania z wartości właściwości w sposób podobny jak przy wywoływaniu metod.

JScript .NET zapewnia bezpośrednią obsługę właściwości klas. Zachowana zostaje składnia identyczna, jak przy wywołaniach zmiennych składowych. Oto przykład implementacji właściwości klasy w języku JScript .NET:

```
class vehicle {
  private var m_name : String;

  function vehicle() {
    m_name="pojazd";
  }

  function get Name() : String { return m_name; }
  function set Name(newName:String) { m_name=newName; }
}
```

```
// korzystanie z właściwości Name...
var v : vehicle = new vehicle;
// wywołanie funkcji ustawiania właściwości
v.Name = "Ferrari";
// wywołanie funkcji odczytu właściwości
Console.WriteLine("Chcę " + v.Name);
```

Właściwość deklarujemy tak samo, jak każdy innych element funkcji, natomiast przy deklarowaniu funkcji odczytującej i funkcji zapisującej jej wartość używamy odpowiednio słowa kluczowego `get` lub `set`.

Rozwiązanie takie ma następujące zalety:

- Właściwości mogą mieć status tylko-do-odczytu, jeżeli zaimplementowana zostanie wyłącznie funkcja `get`.
- Właściwości mogą mieć status tylko-do-zapisu, jeżeli zaimplementowana zostanie wyłącznie funkcja `set`.
- Właściwości mogą być ustawiane i odczytywane tylko wtedy, gdy wymaga tego klient. Jest to wygodne, gdy wykonanie operacji `get` lub `set` wymaga operacji znacznie obciążających system (związanych z obliczeniami lub korzystaniem z bazy danych). Operacje te mogą być wówczas wykonywane wyłącznie wtedy, gdy są wymagane, co zapewnia wysoką wydajność i ogranicza wykorzystanie pamięci.

Przeciążanie funkcji

Przeciążanie funkcji (*function overloading*) pozwala definiować funkcje wykonujące te same zadania na obiektach różnego typu. Przykładem może być klasa `System.Console`, wyposażona w 17 przeciążonych wersji metody `WriteLine`. Metody przeciążone wykonują analogiczne, a koncepcyjnie nie różniące się, czynności zapisywania danych wyjściowych do systemowego strumienia wyjściowego dla obiektów: `String`, `int`, `double`, `float`, `char`, `Object` i `Boolean`.

Przeciążać można wyłącznie funkcje składowe klas. Technika ta nie może być stosowana w odniesieniu do funkcji definiowanych poza klasami. Aby przeciążyć funkcję, definiujemy element o tej samej nazwie, zmieniając liczbę lub (i) typy parametrów. Na podstawie liczby i typów parametrów wywołania funkcji wybierana jest odpowiednia wersja faktycznie wykonywanej funkcji przeciążonej.

Przy wybieraniu wywoływanej wersji funkcji przeciążonej kompilator nie bierze pod uwagę typu wartości zwracanej. Rozpatrywane są wyłącznie parametry przekazywane do funkcji.

Dynamiczne rozszerzanie zbioru właściwości klasy

Zbiór właściwości klasy określamy na etapie jej projektowania i implementacji. Atrybut `expando` umożliwia jednak dalsze rozszerzanie tego zbioru. Na przykład możemy utworzyć obiekt `Person` i dołączyć do niego nowe atrybuty w czasie wykonywania programu (słowo `expando` zostało wyróżnione pogrubionym drukiem):

```
expando class Person {
  var name,age;
  function Person(theName, theAge) {
    name = theName;
    age = theAge;
  }
}
var child = new Person("Siddharth",4);
var someAttribute = "Wzrost";
print(child.name);   // wyświetla: Siddharth
print(child.age);    // wyświetla: 4
child[someAttribute] = 48;
print( typeof child["Wzrost"] ); // wyświetla: number
print( child["Wzrost"] );        // wyświetla: 48
print( child[someAttribute] );   // wyświetla: 48
```

W przedstawionym przykładzie najpierw korzystamy z atrybutu klasy `expando`, następnie dołączamy nową właściwość w czasie wykonywania. Zmienna `someAttribute` wykorzystana zostaje do określenia nazwy nowej właściwości `height`, która zostaje dołączona do obiektu typu `Person` wraz z zainicjowaniem wartości (`child[someAttribute]=48`).

Instrukcja import

Instrukcja `import` umożliwia korzystanie w programie z dodatkowych przestrzeni nazw. Przestrzeń nazw jest rodzajem logicznego zgrupowania. Pisany w języku JScript .NET program korzysta standardowo z globalnej przestrzeni nazw. Wszelkie obiekty tworzone w tej samej przestrzeni nazw są dla siebie widoczne. Zamknięcie obiektów w przestrzeni nazw wiąże je i pozwala wprowadzać pewien rodzaj uporządkowania. Skojarzenie z przestrzenią nazw następuje w wyniku wprowadzenia części kodu w deklaracji przestrzeni, rozpoczynanej się słowem kluczowym `package`. Instrukcja `import` umożliwia włączenie do przestrzeni programu dodatkowej przestrzeni nazw.

Instrukcja pobiera jeden parametr — nazwę importowanej przestrzeni nazw. Przykładami przestrzeni nazw w infrastrukturze klas .NET mogą być `System.Web` i `System.Data`.

Instrukcja `import` działa jak odniesienie do przestrzeni nazw — odpowiedni kod nie jest faktycznie ładowany. Kompilator podejmuje jedynie próbę zlokalizowania implementacji (zazwyczaj pliku DLL) już w czasie kompilowania aplikacji.

Po przeprowadzeniu importowania przestrzeni nazw jej elementy dostępne są za pośrednictwem nazw w pełni kwalifikowanych, jak również nazw skróconych. Rozważmy przykład klasy `Stack`, należącej do przestrzeni nazw `System.Collections` biblioteki klas .NET. Po użyciu odpowiedniej instrukcji `import` jej elementy wywoływać można na dwa sposoby:

```
import System.Collection;
// pełna nazwa kwalifikowana
var m_stack : System.Collections.Stack;

// nazwa skrócona
m_stack = new Stack();
```

Nazwy obu rodzajów można w programie dowolnie łączyć. Zalecaną konwencją jest jednak podawanie w deklaracjach nazw kwalifikowanych, co ułatwia ustalenie lokalizacji klasy w trakcie analizowania kodu.

Instrukcja package

Instrukcja `package` służy do tworzenia nowej przestrzeni nazw. Oto przykład jej użycia:

```
package samplePackage {
  class myClass { /* ... */ }
  class anotherClass { /* ... */ }
}
// użycie pakietu
// instrukcja import pozostaje opcjonalna, o ile pakiet zdefiniowany został
// w tym samym pliku, w którym jest wykorzystywany
import samplePackage;
// pełna nazwa kwalifikowana...
var mc : samplePackage.myClass = new samplePackage.myClass();
// forma krótka
var ac : anotherClass = new anotherClass;
```

Aby umieścić implementację przestrzeni nazw w osobnym pliku (DLL lub bibliotece), należy przeprowadzić jej kompilację jako biblioteki DLL. Gdy pojawi się potrzeba odwołania do niej, używamy instrukcji `import` i kompilujemy program w zwykły sposób.

Instrukcja const

Termin *stała* (*constant*) oznacza wartość, która w trakcie wykonywania programu nie ulega zmianie. JScript .NET pozwala wybrać pomiędzy stałymi o typie określonym i stałymi beztypowymi. Określenie typu pozwala uniknąć wątpliwości przy konwersji i innych operacjach na stałej, natomiast stałe beztypowe są wygodniejsze w użyciu. W różny sposób traktowane są stałe o charakterze wartościowym i odniesieniowym. Typ stałej wartościowej jest prosty (`int`, `double` itp.), stała odniesieniowa jest rodzajem obiektu. Wydruk 9.16 przedstawia przykład użycia instrukcji deklaracji stałych, `const`.

Wydruk 9.16. *Przykład użycia instrukcji const*

```
// stała beztypowa
const i = 100;
// stała o typie określonym
const canadianCity : String = "Toronto";
// stała typu odniesieniowego (tablica obiektów String)
const s : String[] = new String[10];

// poprawne w JScript .NET!
s[0] = "element 0";
s[1] = "element 1";

s[0] = "element A tablicy";
s[1] = "element B tablicy";
```

W przykładzie zilustrowana została szczególna cecha stałych typu odniesieniowego — wartość docelowa odwołania może być modyfikowana. Stała `s` została zadeklarowana jako dziesięcioelementowa tablica ciągów. Kolejne instrukcje modyfikują wartości początkowych elementów tablicy. Jest to poprawne, ponieważ słowo `const`, sygnalizujące niezmienność wartości, odnosi się do odniesienia do obiektu, a nie do zawartości tego obiektu.

Instrukcja enum

Instrukcja `enum` kojarzy nazwę symboliczną z pewną wartością. Wydruk 9.17 ilustruje sposób deklarowania i korzystania z wyliczeń (enumeracji).

Wydruk 9.17.

```
enum trafficLight : int {
  Czerwone,
  Zolte,
  Zielone
}

var aLight : trafficLight = trafficLight.Czerwone;
// równoważne aLight = trafficLight.Zielone...
aLight = "Zielone";
// równoważne aLight = trafficLight.Zolte...
aLight = 1;

// można również przypisywać elementom wyliczenia wartości...
enum relativePriceOfAnItem : String {
  cheap = "Świetny interes",
  notBad = "W porządku",
  tooHigh = "Pomyślę o tym",
  notWorthIt = "Poza moim zasięgiem, dzięki"
}
```

Nowe dyrektywy kompilacji

JScript .NET wprowadza trzy dyrektywy kompilacji, pomocne przy wyszukiwaniu błędów kodu i jego optymalizacji:

- `@option(fast)`,
- `@debug`,
- `@position`.

Wszystkie rozpoczynają się od symbolu `@`. Opiszemy teraz ich przeznaczenie.

Dyrektywa @option(fast)

Dyrektywa włączająca ściślejsze zasady kompilacji kodu, których zastosowanie pozwala uzyskać wydajniejszą pracę aplikacji. Klauzulę `@option(fast)` wprowadzamy na początku programu:

```
@set @option(fast)
```

Powyższy wiersz wyłącza funkcje prowadzące do wydłużenia czasu wykonywania programu. Po jego wprowadzeniu:

- wymagane jest deklarowanie każdej zmiennej,
- obiekt `arguments` nie jest dostępny,
- wywołania funkcji muszą mieć poprawną liczbę argumentów,
- nie można modyfikować, usuwać i wyliczać predefiniowanych właściwości obiektów wewnętrznych. Przykładem takiej właściwości może być `Math.PI` — nie może być zmieniona, usunięta lub wykryta (wyliczona),
- nie można redefiniować funkcji ani przypisywać im wartości (pozwala ograniczyć liczbę przeprowadzanych w trakcie wykonywania operacji kontrolnych),
- nie można tworzyć właściwości dynamicznych (`expando`) obiektów wewnętrznych.

Dyrektywa `@option(fast)` jest standardowo wyłączona dla kodu pozostającego poza deklaracjami `package`, ale włączona dla zawartości takich deklaracji. Opcja może zostać włączona w wierszu poleceń kompilatora JScript .NET przełącznikiem `/fast`:

```
jsc /fast+ ...
```

Opcję `/fast` można łączyć w wierszu poleceń z innymi opcjami uruchomieniowymi kompilatora.

Dyrektywa @debug

Dyrektywa `@debug` decyduje o włączaniu do kompilacji informacji wspomagających analizowanie pracy programu. W niektórych przypadkach, na przykład przy pracy z fragmentami kodu, których nie pisaliśmy samodzielnie, informacje związane z usuwaniem błędów nie są przy kompilacji potrzebne:

```
@set @debug(off)
var Button button1;
@set debug(on)
var v:vehicle = new vehicle;
```

Dane do analizy kodu związanego z przyciskiem `button1` nie będą uwzględniane. Ich generowanie zostanie jednak włączone przed deklaracją zmiennej `v`. Przełączenie takie może być wykonane wielokrotnie.

Dyrektywa @position

Dyrektywa `@position` służy do zerowania licznika wierszy kompilatora. Numery wierszy wykorzystywane są przy generowaniu komunikatów błędów i ostrzeżeń. Podstawowym zastosowaniem dyrektywy jest oznaczanie pierwszego wiersza kodu wprowadzanego przez programistę samodzielnie, w odróżnieniu od „nagłówka", wstawianego automatycznie przy generowaniu pewnego modułu. Oto przykład:

```
/*
  20 wierszy kodu wstawionego przez wspomagający tworzenie formularza generator...
*/
@set @position(line = 1)
va myVariable = 5; // powinno być var myVariable = 5;
// kompilator poinformuje o błędzie w wierszu nr 1
```