

## » Idź do

- Spis treści
- Przykładowy rozdział

## » Katalog książek

- Katalog online
- Zamów drukowany katalog

## » Twój koszyk

- Dodaj do koszyka

## » Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

## » Czytelnia

- Fragmenty książek online

## » Kontakt

Helion SA  
ul. Kościuszki 1c  
44-100 Gliwice  
tel. 032 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
© Helion 1991-2008

## Myślenie obiektowe w programowaniu. Wydanie III

Autor: Matt Weisfeld  
Tłumaczenie: Łukasz Piwko  
ISBN: 978-83-246-2189-7  
Tytuł oryginału: [The Object-Oriented Thought Process \(3rd Edition\)](#)  
Format: 170x230, stron: 328



### Przestaw się na myślenie obiektowe i twórz oprogramowanie wysokiej jakości

- Jak zaprojektować mechanizm obsługi błędów w klasie?
- Jak tworzyć obiekty, aby nadawały się do kooperacji?
- Jak wykorzystywać dziedziczenie i kompozycję?

Obiekty to twory łączące w sobie zarówno dane (atrybuty), jak i procedury (czyli zachowania). Dzięki projektowaniu obiektowemu możesz w pełni wykorzystać funkcjonalność wszystkich obiektów, zamiast traktować dane i procedury jako odrębne jednostki, wymagające osobnej implementacji. Programowanie obiektowe ułatwia zatem wielokrotne wykorzystywanie kodu źródłowego, ale zrozumienie tych technik wymaga zmiany w sposobie myślenia – na myślenie w kategoriach obiektowych.

W książce „Myślenie obiektowe w programowaniu. Wydanie III” pokazano, w jaki sposób można nauczyć się myśleć obiektowo, aby zyskać wprawę w takim programowaniu. Dzięki temu podręcznikowi zrozumiesz, jak wykorzystywać dziedziczenie i kompozycję, agregację i asocjację. Dowiesz się, na czym polega różnica między interfejsem a implementacją. W trzecim wydaniu szeroko opisano także współpracę różnych technologii (możliwą dzięki pomocy języka XML) oraz zamieszczono informacje dotyczące działania obiektów biznesowych w sieciach. Omówiono tu także architekturę klient-serwer oraz usługi sieciowe.

- Programowanie obiektowe a proceduralne
- Hermetyzacja i ukrywanie danych
- Implementacja
- Konstruktory i klasy
- Obsługa błędów
- Wielokrotne użycie kodu
- Serializacja i szeregowanie obiektów
- Projektowanie z wykorzystaniem obiektów
- Dziedziczenie i kompozycja
- Tworzenie modeli obiektów przy użyciu języka UML
- Obiekty i dane przenośne – XML
- Wzorce projektowe

**Myśl obiektowo, programuj profesjonalnie!**

# Spis treści

<b>O autorze .....</b>	<b>13</b>
<b>Wstęp .....</b>	<b>15</b>
<b>Rozdział 1 Wstęp do obiektowości .....</b>	<b>21</b>
Programowanie obiektowe a proceduralne .....	22
Zamiana podejścia proceduralnego na obiektowe .....	25
Programowanie proceduralne .....	26
Programowanie obiektowe .....	26
Definicja obiektu .....	26
Dane obiektu .....	27
Zachowania obiektu .....	27
Definicja klasy .....	31
Klasy są szablonami do tworzenia obiektów .....	31
Atrybuty .....	33
Metody .....	33
Komunikaty .....	33
Modelowanie diagramów UML klas .....	34
Hermetyzacja i ukrywanie danych .....	34
Interfejsy .....	35
Implementacje .....	36
Realistyczna ilustracja paradygmatu interfejsu i implementacji .....	36
Model paradygmatu interfejs – implementacja .....	37
Dziedziczenie .....	38
Nadklasy i podklasy .....	39
Abstrakcja .....	39
Związek typu „jest” .....	40
Polimorfizm .....	41
Kompozycja .....	44
Abstrakcja .....	44
Związek typu „ma” .....	45
Podsumowanie .....	45

Listingi .....	45
TestPerson: C# .NET .....	45
TestPerson: VB .NET .....	46
TestShape: C# .NET .....	47
TestShape: VB .NET .....	48
<b>Rozdział 2 Myślenie w kategoriach obiektowych .....</b>	<b>51</b>
Różnica między interfejsem a implementacją .....	52
Interfejs .....	54
Implementacja .....	54
Przykład implementacji i interfejsu .....	55
Zastosowanie myślenia abstrakcyjnego w projektowaniu interfejsów .....	59
Minimalizowanie interfejsu .....	61
Określanie grupy docelowej .....	62
Zachowania obiektu .....	63
Ograniczenia środowiska .....	63
Identyfikowanie publicznych interfejsów .....	63
Identyfikowanie implementacji .....	64
Podsumowanie .....	65
Źródła .....	65
<b>Rozdział 3 Zaawansowane pojęcia z zakresu obiektowości .....</b>	<b>67</b>
Konstruktory .....	67
Kiedy wywoływany jest konstruktor .....	68
Zawartość konstruktora .....	68
Konstruktor domyślny .....	69
Zastosowanie wielu konstruktorów .....	69
Projektowanie konstruktorów .....	73
Obsługa błędów .....	74
Ignorowanie problemu .....	74
Szukanie błędów i kończenie działania programu .....	75
Szukanie błędów w celu ich naprawienia .....	75
Zgłaszanie wyjątków .....	76
Pojęcie zakresu .....	78
Atrybuty lokalne .....	78
Atrybuty obiektowe .....	79
Atrybuty klasowe .....	81
Przeciążanie operatorów .....	82
Wielokrotne dziedziczenie .....	83
Operacje obiektów .....	84
Podsumowanie .....	85
Źródła .....	85

Listingi .....	86
TestNumber: C# .NET .....	86
TestNumber: VB .NET .....	86
<b>Rozdział 4 Anatomia klasy .....</b>	<b>89</b>
Nazwa klasy .....	89
Komentarze .....	91
Atrybuty .....	91
Konstruktory .....	93
Metody dostępne .....	94
Metody interfejsu publicznego .....	97
Prywatne metody implementacyjne .....	97
Podsumowanie .....	98
Źródła .....	98
Listingi .....	98
TestCabbie: C# .NET .....	98
TestCabbie: VB .NET .....	99
<b>Rozdział 5 Wytyczne dotyczące tworzenia klas .....</b>	<b>101</b>
Modelowanie systemów świata rzeczywistego .....	101
Identyfikowanie interfejsów publicznych .....	102
Minimalizacja interfejsu publicznego .....	102
Ukrywanie implementacji .....	103
Projektowanie niezawodnych konstruktorów i destruktorów .....	104
Projektowanie mechanizmu obsługi błędów w klasie .....	105
Pisanie dokumentacji i stosowanie komentarzy .....	105
Tworzenie obiektów nadających się do kooperacji .....	105
Wielokrotne użycie kodu .....	106
Rozszerzalność .....	106
Tworzenie opisowych nazw .....	107
Wydzielanie nieprzenośnego kodu .....	107
Umożliwianie kopiowania i porównywania obiektów .....	108
Ograniczanie zakresu .....	108
Klasa powinna odpowiadać sama za siebie .....	109
Konservacja kodu .....	111
Iteracja .....	111
Testowanie interfejsu .....	112
Wykorzystanie trwałości obiektów .....	113
Serializacja i szeregowanie obiektów .....	114
Podsumowanie .....	115
Źródła .....	115
Listingi .....	115
TestMath: C# .NET .....	116
TestMath: VB. NET .....	116

<b>Rozdział 6</b>	<b>Projektowanie z wykorzystaniem obiektów .....</b>	<b>119</b>
	Wytyczne dotyczące projektowania .....	119
	Wykonanie odpowiedniej analizy .....	123
	Określanie zakresu planowanych prac .....	123
	Gromadzenie wymagań .....	124
	Opracowywanie prototypu interfejsu użytkownika .....	124
	Identyfikowanie klas .....	124
	Definiowanie wymagań wobec każdej z klas .....	125
	Określenie warunków współpracy między klasami .....	125
	Tworzenie modelu klas opisującego system .....	125
	Studium przypadku — gra w blackjacka .....	125
	Metoda z użyciem kart CRC .....	127
	Identyfikacja klas gry w blackjacka .....	128
	Identyfikowanie zadań klasy .....	131
	Przypadki użycia UML — identyfikowanie kolaboracji .....	136
	Pierwsza analiza kart CRC .....	139
	Diagramy klas UML — model obiektowy .....	139
	Tworzenie prototypu interfejsu użytkownika .....	142
	Podsumowanie .....	142
	Źródła .....	143
<b>Rozdział 7</b>	<b>Dziedziczenie i kompozycja .....</b>	<b>145</b>
	Wielokrotne wykorzystywanie obiektów .....	145
	Dziedziczenie .....	146
	Generalizacja i specjalizacja .....	149
	Decyzje projektowe .....	150
	Kompozycja .....	152
	Reprezentowanie kompozycji na diagramach UML .....	152
	Czemu hermetyzacja jest podstawą technologii obiektowej .....	154
	Jak dziedziczenie osłabia hermetyzację .....	155
	Szczegółowy przykład wykorzystania polimorfizmu .....	157
	Odpowiedzialność obiektów .....	158
	Podsumowanie .....	161
	Źródła .....	161
	Listingi .....	162
	TestShape: C# .NET .....	162
	TestShape: VB .NET .....	163
<b>Rozdział 8</b>	<b>Wielokrotne wykorzystanie kodu</b>	
	<b>— interfejsy i klasy abstrakcyjne .....</b>	<b>165</b>
	Wielokrotne wykorzystanie kodu .....	165
	Infrastruktura programistyczna .....	166
	Co to jest kontrakt .....	168
	Klasy abstrakcyjne .....	169
	Interfejsy .....	172

Wnioski .....	173
Dowód kompilatora .....	176
Zawieranie kontraktu .....	176
Punkty dostępowe do systemu .....	179
Przykład biznesu elektronicznego .....	179
Biznes elektroniczny .....	179
Podejście niezakładające wielokrotnego wykorzystania kodu .....	180
Rozwiązanie dla aplikacji biznesu elektronicznego .....	182
Model obiektowy UML .....	182
Podsumowanie .....	186
Źródła .....	186
Listingi .....	186
TestShop: C# .NET .....	187
TestShop: VB .NET .....	189
<b>Rozdział 9 Tworzenie obiektów .....</b>	<b>193</b>
Relacje kompozycji .....	193
Podział procesu budowy na etapy .....	195
Rodzaje kompozycji .....	197
Agregacja .....	197
Asocjacja .....	198
Łączne wykorzystanie asocjacji i agregacji .....	199
Unikanie zależności .....	200
Liczność .....	201
Kilka asocjacji .....	202
Asocjacje opcjonalne .....	204
Praktyczny przykład .....	204
Podsumowanie .....	205
Źródła .....	206
<b>Rozdział 10 Tworzenie modeli obiektów przy użyciu języka UML .....</b>	<b>207</b>
Co to jest UML .....	207
Struktura diagramu klasy .....	208
Atrybuty i metody .....	210
Atrybuty .....	210
Metody .....	210
Określanie dostępności .....	211
Dziedziczenie .....	212
Interfejsy .....	213
Kompozycja .....	213
Agregacja .....	214
Asocjacja .....	215
Liczność .....	216
Podsumowanie .....	218
Źródła .....	218

<b>Rozdział 11</b>	<b>Obiekty i dane przenośne — XML</b>	<b>219</b>
	Przenośność danych	219
	Rozszerzalny język znaczników — XML	221
	XML a HTML	221
	XML a języki obiektowe	222
	Wymiana danych między firmami	224
	Sprawdzanie poprawności dokumentu względem DTD	224
	Integrowanie DTD z dokumentem XML	226
	Kaskadowe arkusze stylów	231
	Podsumowanie	233
	Źródła	234
<b>Rozdział 12</b>	<b>Obiekty trwałe — serializacja i relacyjne bazy danych</b>	<b>235</b>
	Podstawy trwałości obiektów	235
	Zapisywanie obiektu w pliku płaskim	236
	Serializacja pliku	237
	Jeszcze raz o implementacji i interfejsach	239
	Serializacja metod	241
	Serializacja przy użyciu języka XML	241
	Zapisywanie danych w relacyjnej bazie danych	243
	Dostęp do relacyjnej bazy danych	245
	Ładowanie sterownika	247
	Nawiązywanie połączenia	248
	Kwerendy SQL	248
	Podsumowanie	251
	Źródła	251
	Listingi	252
	Klasa Person: C# .NET	252
	Klasa Person: VB .NET	254
<b>Rozdział 13</b>	<b>Obiekty a internet</b>	<b>257</b>
	Ewolucja technik przetwarzania rozproszonego	257
	Obiektowe skryptowe języki programowania	258
	Weryfikacja danych za pomocą języka JavaScript	260
	Obiekty na stronach internetowych	263
	Obiekty JavaScript	263
	Kontrolki na stronach internetowych	265
	Odtwarzacze dźwięku	265
	Odtwarzacze filmów	266
	Animacje Flash	267
	Obiekty rozproszone i systemy przedsiębiorstw	267
	CORBA	268
	Definicja usługi sieciowej	272
	Kod usług sieciowych	276

Invoice.cs .....	276
Invoice.vb .....	277
Podsumowanie .....	278
Źródła .....	278
<b>Rozdział 14 Obiekty w aplikacjach typu klient-serwer .....</b>	<b>281</b>
Model klient-serwer .....	281
Rozwiązanie własnościowe .....	282
Kod obiektu do serializacji .....	282
Kod klienta .....	283
Kod serwera .....	285
Uruchamianie aplikacji .....	286
Technika z wykorzystaniem XML .....	286
Definicja obiektu .....	287
Kod klienta .....	288
Kod serwera .....	290
Uruchamianie programu .....	291
Podsumowanie .....	292
Źródła .....	292
Listingi .....	292
Definicja obiektu w języku VB .NET .....	292
Kod klienta w języku VB .NET .....	293
Kod serwera w języku VB .NET .....	294
<b>Rozdział 15 Wzorce projektowe .....</b>	<b>297</b>
Historia wzorców projektowych .....	298
Wzorzec MVC języka Smalltalk .....	298
Rodzaje wzorców projektowych .....	300
Wzorce konstrukcyjne .....	301
Wzorce strukturalne .....	305
Wzorce czynnościowe .....	307
Antywzorce .....	309
Podsumowanie .....	310
Źródła .....	310
Listingi .....	310
C# .NET .....	310
VB .NET .....	314
<b>Skorowidz .....</b>	<b>317</b>

# Anatomia klasy

**W** trzech początkowych rozdziałach zostały opisane fundamentalne pojęcia z zakresu obiektowości oraz różnica między interfejsem a implementacją. Bez względu na to, jak starannie zostanie zaplanowany interfejs i implementacja klasy, najważniejsze zawsze jest to, czy dana klasa jest przydatna i czy dobrze współpracuje z innymi klasami. Żadna klasa nie powinna wisieć w próżni, ponieważ, że tak powiem, żadna klasa nie jest samotną wyspą. Obiekty są prawie zawsze tworzone po to, aby współpracować z innymi obiektami. Ponadto obiekt może być częścią innego obiektu lub zajmować miejsce w hierarchii dziedziczenia.

W rozdziale tym zostanie przedstawiona prosta klasa. Na podstawie jej szczegółowej analizy opiszę, co należy wziąć pod uwagę podczas projektowania własnych klas. Rozwinę klasę `Cabbie`, która po raz pierwszy pojawiła się w rozdziale 2. „Myślenie w kategoriach obiektowych”.

W każdym podrozdziale opisuję jeden specyficzny aspekt klasy. Mimo że nie każda klasa musi mieć wszystkie opisane tu składniki, ważne jest, aby wiedzieć o możliwości ich użycia podczas projektowania.

## Uwaga

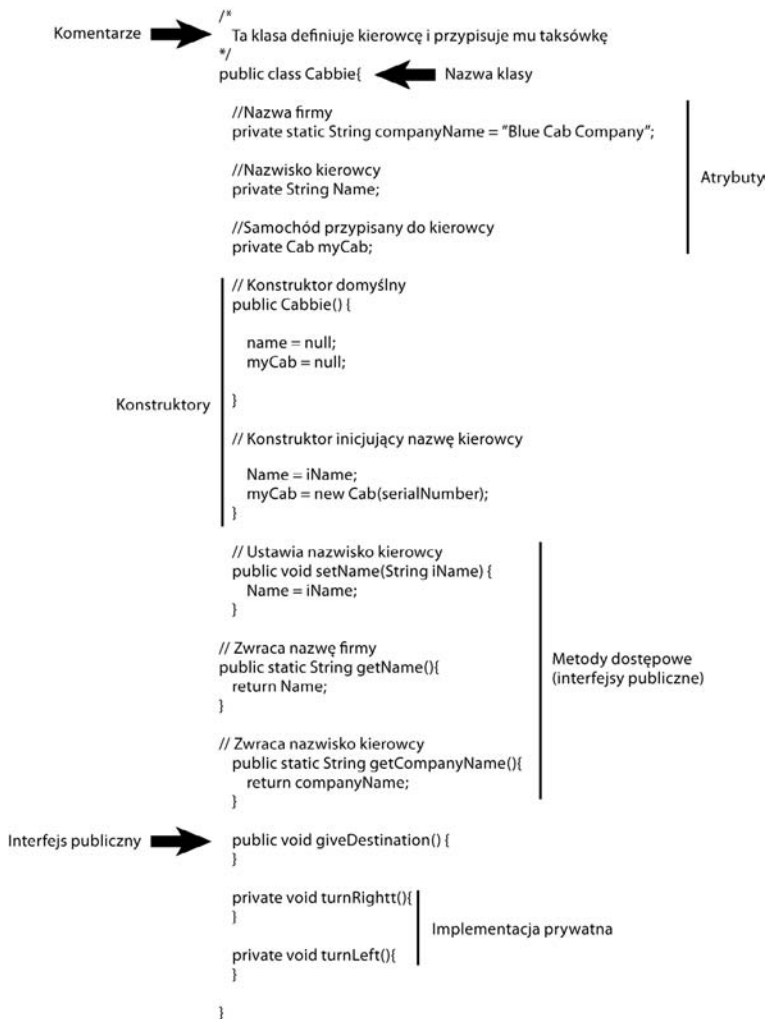
Opisana w tym rozdziale klasa jest przeznaczona tylko dla celów demonstracyjnych. Niektóre jej metody nie zostały zaimplementowane, a ich zadaniem jest tylko prezentować interfejs, który jest najważniejszą rzeczą we wstępnych pracach projektowych.

## Nazwa klasy

Nazwa klasy jest bardzo ważna z kilku powodów. Najbardziej oczywista jest jej funkcja identyfikacyjna. Aby ją dobrze pełniła, nazwa musi być adekwatna do roli klasy. Wybór ten jest tym bardziej ważny, że nazwa stanowi źródło informacji o przeznaczeniu klasy i jej sposobie interakcji z innymi większymi systemami.

Przy dobieraniu nazwy należy także wziąć pod uwagę pewne ograniczenia związane z językiem programowania. Na przykład w Javie nazwa klasy publicznej musi odpowiadać nazwie pliku, w którym się znajduje. Jeśli nazwy te będą różne, program nie zadziała.

Klasa, którą opiszę, została przedstawiona na rysunku 4.1. Jej nazwa, `Cabbie`, znajduje się za słowem kluczowym `class`.



Rysunek 4.1. Przykładowa klasa

```

public class Cabbie {
}

```

## Składnia języka Java

Należy zaznaczyć, że w książce tej do ilustrowania przykładów używam składni języka Java. Jest ona podobna jak w językach C# .NET, VB .NET i C++. Natomiast z innymi obiektowymi językami programowania, jak Smalltalk, ma niewiele wspólnego.

Nazwa klasy `Cabbie` będzie używana wszędzie tam, gdzie tworzy się jej egzemplarz.

## Komentarze

Bez względu na to, za pomocą jakiej składni wstawia się komentarze, są one niezbędne do zrozumienia działania klasy. W językach Java, C# .NET i C++ wyróżnia się dwa rodzaje komentarzy.

### Dodatkowy rodzaj komentarza w Javie i C#

W językach Java i C# w istocie są trzy rodzaje komentarzy. W tym pierwszym języku trzeci rodzaj (`/** */`) służy do generowania dokumentacji. Nie będę go opisywał w tej książce. W C# podobna składnia służy do tworzenia dokumentów XML.

Pierwszy rodzaj komentarza został zaczerpnięty z języka C. Otwiera go ukośnik z gwiazdką (`/*`), a zamyka gwiazdka z ukośnikiem (`*/`). Ten rodzaj komentarza może obejmować wiele wierszy kodu. Należy tylko pamiętać, aby go odpowiednio zamknąć. Jeśli pominię symbol zamykający, część właściwego kodu programu może zostać uznana za komentarz i pominięta przez kompilator. Poniżej znajduje się przykład takiego komentarza, który został użyty w klasie `Cabbie`:

```
/*
   Ta klasa definiuje kierowcę i przypisuje mu taksówkę.
*/
```

Drugi rodzaj komentarza zaczyna się od znaków `//` (dwa ukośniki). Wszystko, co się za nimi znajduje aż do końca wiersza, jest traktowane jako komentarz. Taki komentarz obejmuje tylko jeden wiersz kodu, a więc nie trzeba pamiętać o jego zamknięciu. Z drugiej strony, należy pamiętać, aby nie umieścić na końcu takiego wiersza rzeczywistego kodu aplikacji. Poniżej znajduje się przykład takiego komentarza, który został użyty w klasie `Cabbie`:

```
// Nazwisko kierowcy.
```

## Atrybuty

Atrybuty przechowujące informacje o obiekcie są źródłem informacji o jego stanie. W klasie `Cabbie` są atrybuty przechowujące nazwę firmy taksówkarskiej, nazwisko kierowcy oraz dane przydzielonej mu taksówki. Poniżej znajduje się pierwszy z wymienionych atrybutów, który określa nazwę firmy:

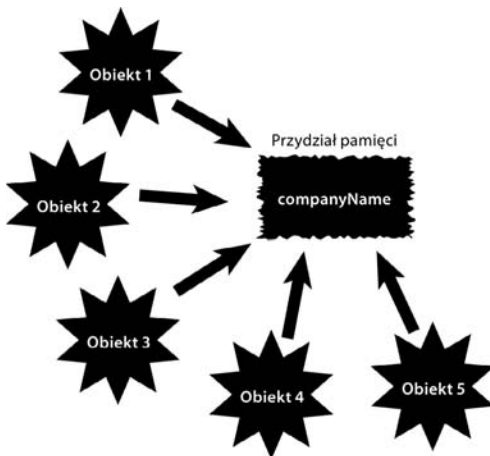
```
private static String companyName = "Blue Cab Company";
```

Należy zwrócić uwagę na użycie słów kluczowych `private` i `static`. Pierwsze oznacza, że metoda lub zmienna jest dostępna tylko wewnątrz obiektu.

### Ukrywanie jak największej ilości danych

Wszystkie atrybuty w prezentowanej tu klasie są prywatne. Jest to zgodne z zasadą, aby utrzymywać tak ubogie interfejsy, jak to możliwe. Jedyнным sposobem na uzyskanie do nich dostępu jest użycie metod interfejsowych (o których piszę nieco dalej w tym rozdziale).

Słowo kluczowe `static` oznacza, że dla wszystkich egzemplarzy tej klasy będzie tylko jedna kopia tego atrybutu. Innymi słowy, jest to atrybut klasy (więcej na temat atrybutów klas napisano w rozdziale 3. „Zaawansowane pojęcia z zakresu obiektowości”). Zatem, nawet jeśli zostanie utworzonych 500 obiektów klasy `Cabbie`, w pamięci powstanie tylko jedna kopia atrybutu `companyName` (rysunek 4.2).



Rysunek 4.2. Przydzielanie pamięci obiektom

Drugi atrybut o nazwie `name` służy do przechowywania nazwiska kierowcy:

```
private String name;
```

Ten atrybut także jest prywatny, przez co inne obiekty nie mogą uzyskać do niego bezpośredniego dostępu.

Atrybut `myCab` zawiera referencję do innego obiektu. Informacje o taksówce, np. numer seryjny i książka napraw, są przechowywane w klasie o nazwie `Cab`.

```
private Cab myCab;
```

### Przekazywanie referencji

Jest bardzo prawdopodobne, że obiekt klasy Cab byłby tworzony przez jakiś inny obiekt i referencja do niego byłaby przekazywana do obiektu klasy Cabbie.

Dla uproszczenia jednak obiekt ten będzie tworzony w obiekcie klasy Cabbie.

Także zawartość obiektu klasy Cab w tym rozdziale nie jest ważna.

Należy zauważyć, że w tym momencie tworzona jest tylko referencja do obiektu klasy Cab. Instrukcja ta nie powoduje przydzielenia mu obszaru pamięci.

## Konstruktory

Klasa Cabbie ma dwa konstruktory. Są to metody, których nazwa jest identyczna z nazwą klasy, do której należą. Pierwszy to konstruktor domyślny:

```
public Cabbie() {
    name = null;
    myCab = null;
}
```

Z technicznego punktu widzenia nie jest to konstruktor domyślny. Kompilator tworzy taki tylko wówczas, gdy programista nie utworzy żadnego. Nazywam go „domyślnym”, ponieważ nie przyjmuje żadnych argumentów. A jak wiadomo, jeśli programista utworzy konstruktor przyjmujący argumenty, system nie dostarczy konstruktora domyślnego, który nic nie przyjmuje. Zasada jest taka, że domyślny konstruktor jest tworzony przez kompilator tylko wówczas, gdy programista nie utworzy *żadnego* konstruktora.

W tym konstruktorze atrybuty name i myCab są ustawiane na null:

```
name = null;
myCab = null;
```

### Wartość null

W wielu językach programowania dostępna jest specjalna wartość null, która reprezentuje brak wartości. Mimo iż wydaje się to osobliwe, ustawienie atrybutu na tę wartość jest bardzo ważną techniką programistyczną. Sprawdzając, czy zmienna ma wartość null, można określić, czy została ona poprawnie zainicjowana. Na przykład czasami konieczne jest zadeklarowanie atrybutu, który będzie później służył do odbierania danych od użytkownika. Można go ustawić na null, zanim to nastąpi. Dzięki takiemu ustawieniu (które jest poprawnym stanem zmiennej) można później sprawdzić, czy atrybut ma odpowiednio ustawioną wartość.

Wiadomo już, że dobrze jest zawsze inicjować atrybuty w konstruktorze. Analogicznie, do dobrych zwyczajów należy sprawdzanie, czy wartość atrybutu wynosi null. W ten sposób można zaoszczędzić sobie wiele kłopotów, jeśli atrybut lub obiekt nie został właściwie zainicjowany. Jeśli na przykład programista spróbuje użyć referencji myCab przed przypisaniem do niej obiektu, najprawdopodobniej nie uda mu się ta sztuka.

Jeśli `myCab` zostanie ustawiona na `null`, będzie można sprawdzić, czy nadal ma wartość `null`, czy jakąś inną, której można użyć. Próba użycia niezainicjowanej referencji tak, jakby była zainicjowana, może zakończyć się zgłoszeniem przez system wyjątku.

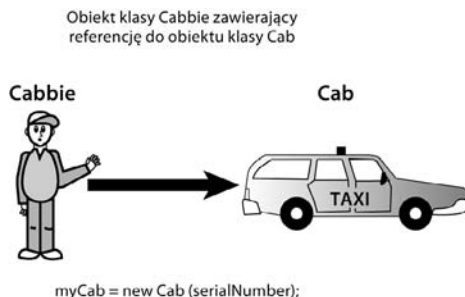
Drugi konstruktor umożliwia zainicjowanie atrybutów `name` i `myCab`:

```
public Cabbie(String iName, String serialNumber) {
    name = iName;
    myCab = new Cab(serialNumber);
}
```

Użytkownik tego konstruktora chcący poprawnie zainicjować atrybuty musiałby podać w jego wywołaniu dwa argumenty łańcuchowe. Należy zauważyć, że w tym konstruktorze inicjowana obiektem jest referencja `myCab`:

```
myCab = new Cab(serialNumber);
```

W wyniku wykonania tej instrukcji programu w pamięci zostanie przydzielony obszar dla obiektu klasy `Cab`. Na rysunku 4.3 została zilustrowana referencja `myCab` do obiektu klasy `Cab`. Użycie dwóch konstruktorów w tej klasie stanowi doskonały przykład techniki przeciążania metod. Należy zauważyć, że wszystkie konstruktory są zawsze publiczne. Jest to bardzo rozsądne, ponieważ konstruktory są oczywistymi kandydatami do składu interfejsu publicznego klasy. Gdyby były prywatne, inne obiekty — na przykład te, które chcą utworzyć egzemplarz klasy `Cab` — nie miałyby do nich dostępu.



Rysunek 4.3. Obiekt `Cabbie` zawierający referencję do obiektu klasy `Cab`

## Metody dostępne

Większość (jeśli nie wszystkie) atrybutów w tej książce jest zadeklarowana jako prywatna, dzięki czemu inne obiekty nie mają do nich bezpośredniego dostępu. Oczywiście tworzenie odizolowanego od reszty świata obiektu, który w żaden sposób nie komunikuje się z innymi obiektami, byłoby nierozsądne — tworzy się je przecież po to, aby udzielały odpowiednich informacji. Czy czasami nie jest konieczne sprawdzenie i zmodyfikowanie wartości atrybutu

innej klasy? Odpowiedź brzmi — tak. Zdarzają się sytuacje, w których jeden obiekt musi zmienić wartość atrybutu innego obiektu, ale nie musi tego robić bezpośrednio.

Klasa powinna chronić swoje atrybuty. Nie powinno być tak, że obiekt A może sprawdzać lub zmieniać atrybuty obiektu B bez żadnej kontroli ze strony obiektu B. Jest to ważne z kilku powodów, z których najważniejsze to chęć utrzymania integralności danych oraz ułatwienie debugowania kodu.

Założmy, że w klasie `Cab` jest błąd. Udało się ustalić, że jest on związany z atrybutem `Name`. Coś zmienia jego wartość, przez co niektóre instrukcje zwracają nieprzydatne dane. Gdyby atrybut `Name` był publiczny i dostęp do niego miałyby wszystkie klasy, trzeba by było sprawdzić wszystkie fragmenty kodu, które się do niego odnoszą i mogą go modyfikować. Gdyby jednak uprawnienia do modyfikacji atrybutu `Name` miały tylko obiekty klasy `Cabbie`, źródła problemu trzeba by było szukać tylko w klasie `Cabbie`. Tego rodzaju kontrolę dostępu umożliwiają specjalne metody zwane *dostępowymi* (ang. *accessor*). Czasami nazywane są metodami pobierającymi (ang. *getter*) i ustawiającymi (ang. *setter*), a czasami po prostu metodami `get()` i `set()`. W tej książce będę korzystał z konwencji polegającej na poprzedzaniu nazwy danego rodzaju metody odpowiednim przedrostkiem `get` lub `set`:

```
// Ustawia nazwisko kierowcy.
public void setName(String iName) {
    name = iName;
}

// Zwraca nazwisko kierowcy.
public String getName() {
    return name;
}
```

W tym przypadku musi być dodatkowy obiekt, np. `Supervisor`, który poprosi obiekt `Cabbie`, aby ujawnił swoje nazwisko (rysunek 4.4). Ważne jest to, że obiekt `Supervisor` nie może tej wartości pobrać samodzielnie — musi o tę informację poprosić obiekt `Cabbie`. Ten sposób działania jest ważny w wielu sytuacjach. Można na przykład napisać metodę o nazwie `setAge()`, która będzie sprawdzała, czy wprowadzona liczba jest równa 0 lub mniejsza. Jeśli tak się stanie, metoda ta może odmówić zaakceptowania tej niepoprawnej wartości. Mówiąc ogólnie, jednym z zadań metod ustawiających jest zapewnianie integralności danych.

Jest to także kwestia bezpieczeństwa. W obiekcie mogą być przechowywane takie poufne dane, jak hasła czy lista płac pracowników, do których dostęp powinien być ograniczony. Zatem metody dostępne umożliwiają wykorzystanie takich mechanizmów, jak funkcje sprawdzające hasła, i innych technik sprawdzania poprawności danych. To znacznie zwiększa poziom integralności danych.



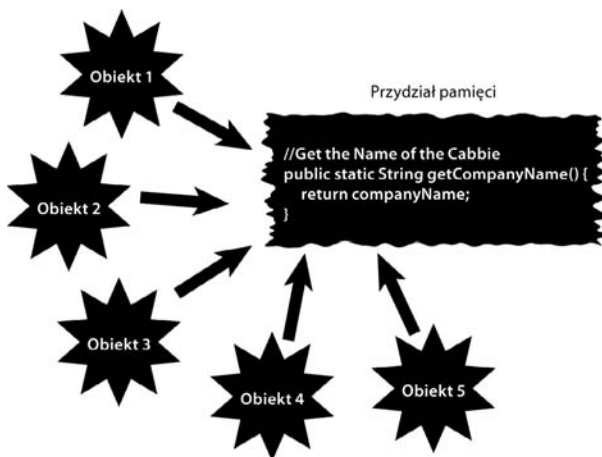
Rysunek 4.4. Prośba o informację

Należy zauważyć, że metoda `getCompanyName` została zadeklarowana jako statyczna (`static`), a więc metoda klasy. Więcej o tego rodzaju metodach zostało napisane w rozdziale 3. Przypomnę, że także atrybut `companyName` jest zdefiniowany jako statyczny. Metodę, podobnie jak atrybut, można zadeklarować jako statyczną, aby zapewnić utworzenie tylko jednej jej kopii dla całej klasy.

### Obiekty

W rzeczywistości nie jest tworzona osobna fizyczna kopia każdej niestatycznej metody dla każdego obiektu. Każdy obiekt odwołuje się do tego samego fragmentu kodu. Z koncepcyjnego punktu widzenia można obiekty traktować jako zupełnie niezależne byty, które mają własne metody i atrybuty.

Poniższy fragment kodu ilustruje definicję metody statycznej, a na rysunku 4.5 przedstawiono, jak kilka obiektów może odwoływać się do tego samego kodu.



Rysunek 4.5. Przydzielanie pamięci metodom

## Statyczne atrybuty

Jeśli atrybut jest statyczny i zostanie napisana metoda służąca do zmiany jego wartości, to każdy obiekt, który ją wywoła, zmieni wartość tej konkretnej kopii atrybutu. Oznacza to, że zmiana jego wartości będzie widoczna we wszystkich obiektach.

```
// Zwraca nazwisko kierowcy.
public static String getCompanyName() {
    return companyName;
}
```

## Metody interfejsu publicznego

Zarówno metody pobierające, jak i ustawiające deklaruje się jako publiczne, a więc wchodzi one w skład interfejsu publicznego klasy. Są wyodrębniane, ponieważ mają szczególne znaczenie dla konstrukcji klasy. Jednak znaczna część *prawdziwej* pracy jest wykonywana w innych metodach. Przypominając z rozdziału 2.: metody interfejsu publicznego są zazwyczaj bardzo abstrakcyjne. Konkretna jest natomiast kryjąca się za nimi implementacja. Dla tej klasy utworzę metodę o nazwie `giveDestination()`, która będzie stanowiła interfejs publiczny, za pomocą którego użytkownik będzie mógł poinformować, gdzie chce dotrzeć:

```
public void giveDestination (){
}
}
```

Treść tej metody nie jest tu ważna. Ważne jest to, że jest to publiczna metoda, która stanowi część interfejsu publicznego klasy.

## Prywatne metody implementacyjne

Mimo że w rozdziale tym były do tej pory opisywane tylko publiczne metody, nie wszystkie metody w klasie wchodzi w skład interfejsu publicznego. **Niektóre** metody mogą zostać ukryte przed innymi klasami. W ich deklaracji należy użyć słowa kluczowego `private`:

```
private void turnRight(){
}

private void turnLeft() {
}
```

Zadaniem metod prywatnych jest realizacja funkcji implementacyjnych, a nie wzbogacanie interfejsu publicznego. Skoro żadna inna klasa nie może ich wywołać, to co to robi? Odpowiedź brzmi — inne metody tej samej klasy. Może je na przykład wywoływać poniższa metoda o nazwie `giveDestination`:

```
public void giveDestination (){
    .. kod źródłowy
}
```

```

turnRight();
turnLeft();

.. kod źródłowy

}

```

Innym przykładem może być wewnętrzna metoda szyfrująca, która jest używana tylko wewnątrz klasy. Mówiąc krótko, metody tej nie można wywołać poza obiektem, do którego należy.

Chodzi o to, że prywatne metody są częścią implementacji, która jest niedostępna innym klasom.

## Podsumowanie

W tym rozdziale zostały opisane podstawowe pojęcia, których znajomość jest niezbędna do zrozumienia budowy klasy. Podczas gdy w tym rozdziale opisano praktyczne aspekty tworzenia klas, w rozdziale 5. „Wytyczne dotyczące tworzenia klas” prezentuję bardziej teoretyczne stanowisko.

## Źródła

Martin Flower, *UML Distilled*, 3rd ed., Addison-Wesley Professional, Boston 2003.

Stephen Gilbert i Bill McCarty, *Object-Oriented Design in Java*, The Waite Group Press, Berkeley 1998.

Paul Tyma, Gabriel Torok i Troy Downing, *Java Primer Plus*, The Waite Group Press, Berkeley 1996.

## Listingi

Poniżej przedstawiam kod źródłowy w językach C# .NET i VB .NET. Listingi te są odpowiednikami przykładów kodu w języku Java, które przedstawiono w tekście rozdziału.

### TestCabbie: C# .NET

```

using System;

namespace ConsoleApplication1
{
    class TestPerson
    {
        public static void Main()
        {
            Cabbie joe = new Cabbie("Joe", "1234");
        }
    }
}

```

```

        Console.WriteLine(joe.Name);
        Console.ReadLine();
    }
}

public class Cabbie
{
    private string name;
    private Cab myCab;

    public Cabbie() {
        name = null;
        myCab = null;
    }

    public Cabbie(string iName, string serialNumber) {
        name = iName;
        myCab = new Cab(serialNumber);
    }

    // Metody
    public String Name
    {
        get { return name; }
        set { name = value; }
    }
}

public class Cab
{
    private string serialNumber;

    public Cab (string sn) {
        serialNumber = sn;
    }
}
}
}

```

## TestCabbie: VB .NET

Module TestCabbie

Sub Main()

Dim joe As New Cabbie("Joe", 1234)

```
        joe.Name = "joe"

        Console.WriteLine(joe.Name)

        Console.ReadLine()

    End Sub

End Module
Public Class Cabbie

    Dim strName As String

    Sub New()

        strName = " "

    End Sub

    Sub New(ByVal iName As String, ByVal serialNumber As String)

        strName = iName
        Dim myCab As New Cab(serialNumber)

    End Sub

    Public Property Name() As String
        Get
            Return strName
        End Get
        Set(ByVal value As String)
            strName = value
        End Set
    End Property

End Class
Public Class Cab

    Dim serialNumber As String

    Sub New(ByVal val As String)

        serialNumber = val

    End Sub

End Class
```