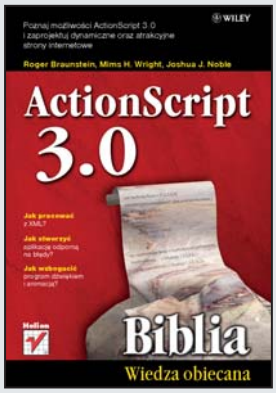
Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

ActionScript 3.0. Biblia



Autor: Roger Braunstein, Mims H. Wright, Joshua J. Noble
Tłumaczenie: Sebastian Rataj
ISBN: 978-83-246-1939-9
Tytuł oryginału: ActionScript 3.0 Bible
Format: 172x245, stron: 752

Poznaj możliwości ActionScript 3.0 i zaprojektuj dynamiczne oraz atrakcyjne strony internetowe

- Jak pracować z XML?
- Jak stworzyć aplikację odporną na błędy?
- Jak wzbogacić program dźwiękiem i animacją?

Język programowania ActionScript wykorzystywany jest do tworzenia treści dla programu Flash Player. Jest on niezbędny wszędzie tam, gdzie trzeba tworzyć dynamiczne aplikacje Flash, reagujące na działanie użytkownika, nadające się do ponownego użycia i łatwe w dostosowaniu do nowych okoliczności. Wersja ActionScript 3.0 została stworzona od podstaw z wykorzystaniem najnowszej specyfikacji ECMA. W aktualnej wersji tego języka wprowadzono wiele nowych funkcji, m.in. nowy sposób zarządzania typami wyświetlania, obsługę błędów czasu wykonania, zakończenia metod i wewnętrzny model zdarzeń.

Książka „ActionScript 3.0. Biblia" obszernie i wyczerpująco opisuje ten obiektowy język programowania, służący do budowania nowoczesnych aplikacji internetowych. Znajdziesz tu zarówno niezbędną wiedzę teoretyczną, jak i praktyczne przykłady, ilustrujące chociażby programowanie z wykorzystaniem klas, używanie tablic i obiektów czy obsługę błędów. Z tym podręcznikiem nauczysz się tworzyć interaktywną zawartość Flasha oraz aplikacje Flex, a także pracować z dźwiękiem, animacją i danymi zewnętrznymi. Zdobędziesz wiedzę, która pozwoli Ci na zrealizowanie wszelkich pomysłów dotyczących Twojego programu.

- Rozpoczęcie pracy z ActionScript 3.0
- Stosowanie zmiennych
- Praca z operatorami
- Programowanie z wykorzystaniem klas
- Metody i funkcje
- Konwersje liczbowe
- Praca z XML
- Wyrażenia regularne
- Pola tekstowe i formatowanie tekstu
- Korzystanie z timerów
- Obsługa błędów
- Praca z danymi zewnętrznymi
- Wzbogacanie programu dźwiękiem i wideo
- Wdrażanie programu

Poznaj ActionScript i wykorzystaj pełnię jego możliwości!

Spis treści

Rozdział 4.

Praca z metodami i funkcjami

W tym rozdziale:

- Ponowne wykorzystanie bloków kodu poprzez definiowanie własnych funkcji
- Wywoływanie funkcji i przekazywanie argumentów za pomocą operatora wywołania
- Otrzymywanie wyników poprzez zwracanie wartości z funkcji
- Praca z funkcjami jako obiektami
- Tworzenie funkcji rekurencyjnych

Teraz, gdy wiesz już wszystko na temat zmiennych, zapewne będziesz chciał coś z nimi zrobić. W tym miejscu do akcji wkraczają metody i funkcje. Funkcje są blokami kodu wielokrotnego zastosowania, które mogą być definiowane w Twoich własnych klasach i często są stosowane w całym API ActionScript 3.0.

Funkcje umożliwiają organizowanie kodu w niezależne fragmenty. Mogą być używane do zwracania różnych wyników w zależności od dostarczonych im danych. Być może najważniejsze jest jednak to, że funkcje mogą enkapsulować wewnątrz klasy zachowania i funkcjonalność oraz oferować publiczny interfejs dla tej funkcjonalności. W niniejszym rozdziale opisano sposoby wykorzystywania funkcji i tworzenia od podstaw własnych funkcji.

Jedną z rzeczy, jakie należy zapamiętać w związku z ActionScript, jest to, że *każda zmienna i część klasy jest obiektem*. Funkcje nie są tutaj wyjątkiem. Chociaż może być to trudne do wyobrażenia, funkcje są instancjami klasy `Function` i zawierają własne metody oraz właściwości. W tym rozdziale opisujemy, jak używać funkcji jako obiektów.

Funkcje

Mówiąc najprościej, funkcja jest fragmentem kodu zachowanym w postaci zapisanej procedury, która może być uruchomiona w razie potrzeby poprzez wpisanie nazwy funkcji lub wywołanie funkcji. Każdy napisany przez Ciebie program będzie w dużym stopniu bazował na funkcjach.

Różnice pomiędzy metodami i funkcjami

Będziesz spotkać się ze słowami „metoda" oraz „funkcja" (a czasami nawet z określeniem „domknięcie funkcji") oznaczającymi blok kodu nadający się do wielokrotnego zastosowania. Pomimo tego iż często nazw tych używa się zamiennie, występuje pomiędzy nimi niewielka różnica. Metoda jest dowolną funkcją zdefiniowaną wewnątrz klasy. Metody powinny być logicznie powiązane z instancją klasy i dążyć do spójności z obiektem, w którym są zdefiniowane. Np. klasa `Kitty` może definiować funkcję o nazwie `meow()`. W tym przypadku `meow()` będzie metodą klasy `Kitty`.

Ogólnie pojęcia „metoda" oraz „funkcja" różnią się więc tylko nazwami — innymi słowy, metody nie posiadają żadnych dodatkowych możliwości, nie występuje też słowo kluczowe „method". Oto przykład wywołania wymyślonej funkcji:

```
addElementToGroup(myGroup, myElement);
```

A teraz wywołanie tej samej funkcji, gdy napisana będzie jako metoda:

```
myGroup.addElement(myElement);
```

Jak widzisz, powyższe wywołania są do siebie podobne, ale drugie zostało tak zaprojektowane, aby operować na instancji `myGroup`.

W tej książce używamy obu terminów. Używamy słowa „funkcja" dla zastosowania ogólnego, a podczas opisywania funkcjonalności konkretnej funkcji zdefiniowanej wewnątrz klasy używamy pojęcia „metoda". Ponieważ prawie wszystkie funkcje w ActionScript znajdują się w klasach, większość z nich jest metodami.

Inaczej niż w AS2, który czasami wykorzystuje klasę `Delegate`, w AS3 metody są powiązane z instancjami, w których się znajdują. Wykonają się tak, jakby zostały wywołane z instancji, nawet wtedy gdy funkcja zostanie przekazana do innego zasięgu. Ta właściwość ActionScript 3.0 znana jest jako **domknięcie metody**.

Wywołanie funkcji

Wykonanie kodu funkcji lub metody znane jest jako **wywołanie** lub **uruchamianie**. Funkcje w ActionScript są wywoływane za pomocą nazwy funkcji, po której występują nawiasy `()`. Oficjalnie nawiasy te nazywane są **operatorem wywołania**.

Dodatkowe informacje mogą być przekazywane do Twoich funkcji poprzez dodanie **argumentów** (znanych także jako **parametry**) umieszczanych między nawiasami i rozdzielanych przecinkami. Niektóre funkcje będą posiadać argumenty opcjonalne lub wymagane. Liczba i typ wartości przekazywanych do funkcji musi odpowiadać definicji funkcji, zwanej także **sygnaturą metody**. Jeszcze inne funkcje nie wymagają wcale argumentów, a w ich wywołaniu używa się pustej pary nawiasów.

Poniżej widzisz dwa poprawne wywołania funkcji:

```
trace("Witaj świecie!");
addChild(mySprite);
```

Nie zapomnij o dodaniu nawiasów podczas wywoływania funkcji. W przeciwnym razie wyznaczona zostanie wartość funkcji jako zmiennej (obiektu typu `Function`) zamiast wykonania się kodu wewnątrz funkcji, co doprowadzi do nieoczekiwanych wyników. W dalszej części tego rozdziału dowiesz się, jak i kiedy używać funkcji bez operatora wywołania.

Wywoływanie funkcji a wywoływanie metod

Aby uzyskać dostęp do metod wewnątrz konkretnej instancji, użyj nazwy obiektu zawierającego metody, a po niej umieść kropkę i wywołanie metody. Taki sposób wywołania nazywa się **notacją kropkową**. Możesz traktować to jak wywołanie funkcji w kontekście obiektu.

```
var nameList:Array = new Array();
nameList.push("Daniel", "Julia", "Paweł");
nameList.reverse();
trace(nameList); // Wyświetli: "Paweł", "Julia", "Daniel"
```

Niektóre instrukcje ActionScript lub operatory, jak `typeof` czy `delete`, nie są technicznie funkcjami, nawet jeśli tak się zachowują. Z tego też powodu nie będziesz musiał dodawać nawiasów wokół argumentów dla tych poleceń i nie będziesz w stanie uzyskać dostępu do metod i właściwości klasy `Function` dla tych zarezerwowanych słów. Operatory te, w przeciwieństwie do większości funkcji, istnieją globalnie i będą dostępne w dowolnym miejscu kodu.

Tworzenie własnych funkcji

Aby utworzyć własne metody, należy dodać funkcje do swoich plików przechowujących deklaracje klas. Nazywa się to **definiowaniem** lub **deklarowaniem** funkcji. Przyjrzyjmy się tej operacji.

Definiowanie funkcji

Funkcje posiadają następującą strukturę:

```
public function doSomething(arg1:Object, arg2:Object):void {
  // wykonywany kod znajduje się tutaj
}
```

Rozbijmy to na poszczególne człony:

- `public` — modyfikator dostępu będący zazwyczaj słowem kluczowym, takim jak `public`, `private` czy `internal`, używany jest do określania dostępu do tej metody dla innych klas. Jeśli definiujesz funkcję poza pakietem (np. na osi czasu w programie Flash CS3), powinieneś pominąć tę część. Zawsze powinieneś definiować przestrzeń nazw dla swoich metod. W celu uproszczenia niektóre przykłady w tej książce mogą pomijać tę część.
- `function` — następnie występuje słowo kluczowe `function`. Jest ono zawsze wymagane podczas definiowania funkcji, podobnie jak słowo `var` wymagane jest podczas definiowania zmiennej.

- `doSomething` — bezpośrednio po słowie kluczowym `function` występuje nazwa funkcji. Jest ona zarazem poleceniem, jakie wywołasz, gdy zechcesz wykonać daną funkcję.

Najlepsze są opisowe nazwy funkcji. Opisują akcję i mogą być czytane z łatwością, jakbyś czytał zdanie. Niepisana zasada mówi, że nazwa funkcji powinna zaczynać się od czasownika. Np. nazwa `button()` nie jest tak wyrazista jak `drawButton()`. Idąc tym tropem, najlepszymi nazwami dla zmiennych są rzeczowniki. Możesz więc łączyć czasownik z rzeczownikami i tworzyć krótkie zdania, jak `snakeHandler.feed(python, mouse)`.

- `(arg1:Object, arg2: Object)` — za nazwą funkcji występuje rozdzielona przecinkami lista argumentów, znajdująca się pomiędzy parą nawiasów. Po każdym argumencie powinien wystąpić dwukropek oraz typ danych tego argumentu.
- `:void` — za nawiasami występuje kolejny dwukropek i typ danych dla wartości zwracanej przez funkcję. Jest to wartość, jaka zostanie zwrócona przez funkcję po zakończeniu jej wykonywania. W tym przypadku zwracany typ to `void`, ponieważ funkcja nie posiada instrukcji zwracającej. Więcej na temat typów zwracanych dowiesz się w dalszej części tego rozdziału.
- `{...}` — cały kody wykonywany po wywołaniu funkcji znajduje się pomiędzy dwoma nawiasami klamrowymi {}. Nazywa się go ciałem funkcji.

Wszystkie funkcje wymagają słowa kluczowego `function`, nazwy funkcji, nawiasów i ciała funkcji. Pozostałe obszary nie są wymagane, co nie oznacza, że nie powinieneś ich używać.

Przekazywanie parametrów do funkcji

Funkcje stają się o wiele bardziej interesujące i bardziej przydatne, gdy dostarczysz im jakichś danych zewnętrznych. Można to osiągnąć poprzez dodanie do Twojej definicji funkcji *argumentów*, nazywanych również *parametrami*. Aby tego dokonać, po prostu umieść jeden lub więcej argumentów w nawiasach w instrukcji funkcji. Nazwy, jakie tutaj zdefiniujesz, będą dostępne w trakcie wykonywania jako zmienne lokalne i będziesz mógł ich używać podczas wykonywania kodu.

Nie wszystkie funkcje będą wymagać parametrów. Niektóre będą wywoływane z pustymi nawiasami.

Przyjrzyj się przykładowi wyznaczającemu obwód okręgu, w którym jako parametr przekazywana jest średnica okręgu:

```
function getCircumference(diameter:Number):Number {
  return Math.PI * diameter;
}
```

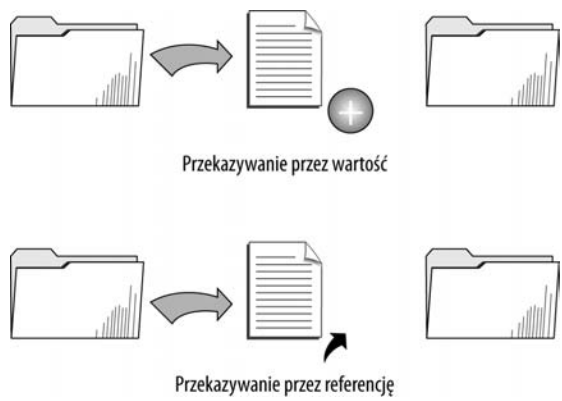
Jak widzisz, parametry te mogą być ponownie zapisywane i istnieją w lokalnym zasięgu funkcji. Oznacza to, że przesłaniają zmienne posiadające te same nazwy i istniejące w zasięgu klasy lub zasięgu globalnym.

W trybie standardów liczba argumentów musi odpowiadać definicji funkcji. Różni się to od podejścia stosowanego w poprzednich wersjach ActionScript, gdzie symetria pomiędzy argumentami w definicji funkcji oraz wywołaniem funkcji nie była wymagana.

Przekazywanie przez referencję lub przez wartość

W kontekście przekazywania wartości do funkcji ActionScript 3.0 traktuje proste oraz złożone typy danych w odmienny sposób. Proste typy danych są przekazywane *przez wartość* — czyli ich wartość jest kopiowana do funkcji, pozostawiając oryginał nietknięty, niezależnie od tego, co wydarzy się wewnątrz funkcji. Złożone typy danych, jak `Array`, są przekazywane *przez referencję*, co powoduje przekazanie oryginalnego obiektu zamiast duplikatu. Reguły te stosuje się w ten sam sposób podczas przypisywania wartości zmiennym.

Jeżeli używasz komputera, a jestem pewien, że większość z Czytelników tej książki go używa, na pewno znasz różnicę pomiędzy kopiowaniem plików a tworzeniem skrótów (lub aliasów w Mac OS). Zgodnie z tym, co pokazano na rysunku 4.1, przekazywanie poprzez wartość jest bardzo podobne do powielenia pliku, ponieważ oryginalny plik pozostaje bez zmian na swoim miejscu. Przekazywanie przez referencję przypomina tworzenie skrótu do oryginalnej wartości. Gdy utworzysz skrót do pliku tekstowego, otworzysz skrót i edytujesz plik, wprowadzone przez Ciebie zmiany zostaną zapisane w oryginalnym pliku, do którego się podłączyłeś.



Rysunek 4.1. *Kopiowanie pliku jest podobne do przekazywania parametru przez wartość, podczas gdy tworzenie skrótu przypomina przekazanie parametru przez referencję*

Poniżej przedstawiony został przykład przekazywania parametru przez wartość. Zauważ, że oryginalna wartość nie ulega zmianie:

```
function limitPercentage(percentage:Number):Number {
  // upewnij się, czy wartość procentowa jest mniejsza od 100
  percentage = Math.min(percentage, 100);
  // upewnij się, czy wartość procentowa jest większa od 0
  percentage = Math.max(percentage, 0);
  return percentage;
}
var effort:Number = 110;
var trimmedPercentage:Number = limitPercentage(effort);
trace(effort, "%"); // wyświetli: 110%
trace(trimmedPercentage, "%"); // wyświetli: 100%
```

Początkowa wartość zmiennej `effort` nie zmieniła się, chociaż zmienna argumentu `percentage` zmieniła wartość podczas wykonywania funkcji.

Dla typów złożonych sprawa wygląda inaczej. Są przekazywane *przez referencję*, co oznacza, że argument zachowuje się jak skrót lub odsyłacz do oryginalnego pliku. Zmiany argumentu są odzwierciedlane jako zmiany bezpośrednio na przekazywanej wartości. Większość typów danych przekazuje referencje podczas przypisywania zmiennej wartości. Poniższy kod przedstawia, jak zmienna `employee` jest bezpośrednio łączona z obiektem przekazywanym jako parametr.

```
function capitalizeEmployeeName(employee:Object):void {
  if (employee.name != null) {
    employee.name = employee.name.toUpperCase();
  }
}
var person:Object = {name: "Jan Kowalski"};
capitalizeEmployeeName(person);
trace(person.name); // wyświetli: JAN KOWALSKI
```

Jak widać w tym przykładzie, zmiana wielkości liter jest przeprowadzana bezpośrednio na zmiennej `employee`. Dzieje się tak dlatego, że wartość `employee` odwołuje się bezpośrednio do oryginalnej kopii `person` — dlatego nazwisko osoby jest również powiązane.

Poniżej wypisane typy danych przekazywane są przez wartość:

```
String
Number
int
uint
Boolean
Null
void
```

Pozostałe typy danych są przekazywane przez referencję.

Ustawianie wartości domyślnych

Nowością w ActionScript 3.0 jest możliwość ustawiania domyślnych wartości dla argumentów metody. Aby tego dokonać, dodaj za nazwą argumentu znak równości (=) oraz domyślną wartość:

```
function showGreeting(name:String = "obcy"):void {
  trace("Witaj, " + name + ", miło Cię poznać.");
}
showGreeting("Andrzej"); // wyświetli: Witaj, Andrzej, miło Cię poznać.
showGreeting(); // wyświetli: Witaj, obcy, miło Cię poznać.
```

Jak widzisz na przykładzie drugiego wywołania, imię zostało zastąpione domyślną wartością `"obcy"`.

Podczas stosowania domyślnych wartości należy pamiętać o jeszcze jednej zasadzie. Wszystkie argumenty posiadające domyślną wartość są uznawane za opcjonalne. Jako takie muszą być umieszczone na końcu listy argumentów. Dlatego też następujący kod:

```
function storeAddress(name:String, zip:String = null, email:String)
```

nie jest poprawnym kodem, ponieważ `email` jest parametrem wymaganym, a występuje za `zip`, który jest opcjonalny. Poprawna kolejność wygląda w tym przypadku następująco:

```
function storeAddress(name:String, email:String, zip:String = null)
```

Użycie argumentu reszty (...)

ActionScript 3.0 wprowadza nowe rozwiązanie nazywane **parametrem reszty** (...). Parametr reszty oznaczany jest symbolem ..., reprezentującym zmienną liczbę argumentów w postaci tablicy. Po nim następuje nazwa tej tablicy. Pisany jest jako trzy kropki, po których następuje nazwa używana dla tablicy zawierającej te wartości. Dodanie do własnej funkcji parametru reszty umożliwi Ci wywoływanie funkcji z dowolną liczbą parametrów. Jeśli np. potrzebujesz funkcji dodającej do siebie dowolną ilość liczb, możesz użyć parametru reszty:

```
function sum(... numbers):Number {
  var result:Number = 0;
  for each (var num:Number in numbers) {
    result += num;
  }
  return result;
}
trace(sum(1,2,3,4,5)); // wyświetli: 15
```

Wartości przekazane do funkcji zawarte są w tablicy reszty o nazwie `numbers`. Dostęp do elementów tej tablicy uzyskasz, przechodząc w pętli po jej kolejnych wartościach.

Tablice zostały dokładniej omówione w rozdziale 8.

Parametr reszty może zostać również użyty z innymi, wymaganymi parametrami. Wymagane parametry jak zwykle będą posiadać swoje własne nazwy i będą istnieć niezależnie od tablicy reszty. Wszystkie dodatkowe parametry po tych wymaganych zapisane zostaną w tablicy reszty. Zmodyfikujmy poprzedni przykład tak, aby wymagał przekazania przynajmniej jednego argumentu:

```
function sum(base:Number, ... numbers):Number {
  var result:Number = base;
  for each (var num:Number in numbers) {
    result += num;
  }
  return result;
}
trace(sum()); // zgłosi błąd czasu wykonywania
trace(sum(1,2,3,4,5)); // wyświetli: 15
```

Dostęp do obiektu arguments

Klasa `Function` definiuje pojedynczą właściwość, która jest wykorzystywana do przechowywania informacji na temat danych przekazywanych do funkcji podczas jej wywoływania. Właściwość ta nazywa się `arguments`. Jest to obiekt specjalnego typu danych `Arguments`, który pomimo iż zachowuje się podobnie do tablicy, technicznie nie jest tablicą. Każda funkcja jest instancją klasy `Function`, przez co posiada właściwość `arguments`.

Właściwość `arguments` w ActionScript 3.0 straciła na znaczeniu i pozostawiono ją jedynie w celu obsługi starszego kodu. Zalecamy zamiast niej używanie nowego parametru reszty (`...`).

Parametr reszty i obiekt `arguments` nie mogą być używane razem. Dołączenie argumentu reszty w Twojej funkcji wyłączy obiekt `arguments`.

Użyteczność obiektu `arguments` jest blokowana w trybie standardów, ponieważ tryb ten wymaga dostarczenia dokładnej liczby argumentów zdefiniowanych w deklaracji funkcji. Jeśli zdecydujesz się na użycie obiektu `arguments`, możesz wyłączyć kompilowanie w trybie standardów.

Uzyskiwanie wartości z właściwości arguments

Właściwość `arguments` jest bardzo podobna do parametru reszty w tym, że zawiera każdy parametr, który jest przekazywany do funkcji w postaci uporządkowanej tablicy. Główna różnica pomiędzy nimi polega na tym, że w tablicy argumentów znajdą się wszystkie nazwane, jak również nienazwane argumenty, podczas gdy w parametrze reszty zawarte są tylko nienazwane argumenty.

Do parametrów przekazywanych do funkcji można uzyskać dostęp za pomocą składni tablicowej. Pierwszy przekazany parametr będzie posiadać indeks 0, drugi indeks 1 i tak dalej. Poniższy przykład zostanie skompilowany poprawnie, gdy tryb standardów będzie wyłączony:

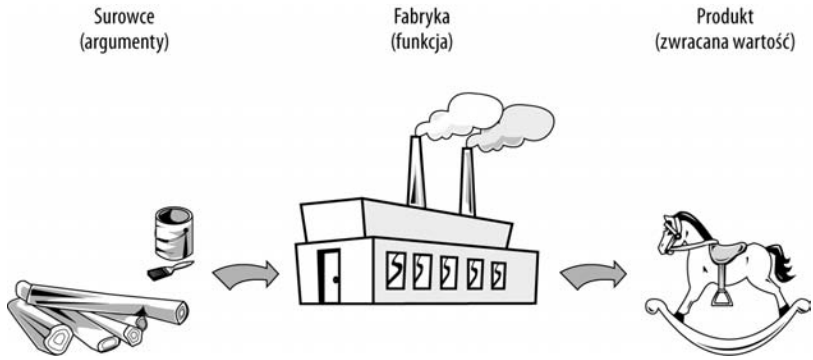
```
function sum():Number {
  var result:Number = 0;
  for each (var num:Number in arguments) {
    result += num;
  }
  return result;
}
trace(sum(1,2,3,4,5)); // wyświetli: 15
```

Właściwość `arguments.caller` nie jest już dostępna w ActionScript 3.0.

Zwracanie wyników

Funkcje potrafią o wiele więcej niż zwykłe wykonywanie predefiniowanego kodu — to zaledwie połowa ich zastosowania. Funkcje w ActionScript potrafią podobnie jak funkcje matematyczne obliczać wartości w oparciu o wprowadzone zmienne. Możliwość ta jest bardzo pożyteczna.

Możesz traktować funkcję tak, jakby była fabryką przekształcającą surowce w gotowe produkty, co przedstawiono na rysunku 4.2. W tym przypadku surowce są argumentami przekazywanymi do wywołania funkcji, a końcowy produkt jest wartością zwracaną przez funkcję.



Rysunek 4.2. *Funkcje, podobnie jak fabryki, przekształcają surowce w gotowy produkt*

Zwracanie wartości za pomocą wyrażenia return

W najprostszej formie instrukcja zwracająca jest słowem kluczowym `return`, po którym następuje zwracana wartość. Wartość może być typu prostego, może być zmienną, wynikiem działania innej funkcji czy dowolnym innym typem wyrażenia.

```
return someValue;
```

Zauważ, że w instrukcji `return` nie używa się nawiasów. Wynika to z faktu, że nie jest to funkcja, ale raczej operator. Możesz natknąć się czasem na nawiasy — np. `return ↪(someValue)` — ale nie są one wymagane.

W większości sytuacji będziesz chciał w określony sposób manipulować argumentami wewnątrz funkcji, a następnie zwrócić nową, obliczoną wartość. Poniższy przykład oblicza obwód okręgu poprzez pobranie średnicy jako parametru `diameter`, pomnożenie jej przez matematyczną stałą pi i zwrócenie wyniku w postaci liczby:

```
function getCircumference(diameter:Number):Number {
  var circumference:Number = diameter * Math.PI;
  return circumference;
}
trace(getCircumference(25)); // wyświetli: 78.53981633974483
```

Jak widzisz, wywołanie metody `getCircumference()` może zostać użyte w instrukcji `trace`, tak jakby było zmienną. Wyniki działania funkcji definiującej typ zwracany mogą być również zapisywane w zmiennej. Realizuje się to poprzez umieszczenie zmiennej z lewej strony znaku równości.

Gdy tylko w funkcji wykonana zostanie instrukcja `return`, funkcja kończy się, a reszta jej kodu nie jest przetwarzana. Z tego też powodu każda funkcja może zwrócić tylko jedną wartość. Instrukcja `return` może też zostać użyta do przedwczesnego zakończenia funkcji.

```
function getColor():String {
  var color:String = "czerwony";
  return color;
  color = "niebieski";
  return color;
}
var color:String = getColor();
trace(color); // wyświetli: czerwony
```

Ta metoda zawsze będzie zwracać „czerwony", ponieważ wykonywanie kodu tej funkcji zostaje zatrzymane po pierwszej instrukcji zwracającej.

Chcąc zwrócić z funkcji więcej niż jedną wartość, możesz utworzyć obiekt lub jeszcze lepiej własną klasę, która będzie przechowywać wszystkie Twoje odpowiedzi w jednej zmiennej.

```
return {firstName:"Paul", lastName:"Newman"};
```

Definiowanie typu zwracanej wartości dla Twojej funkcji

Gdy definiujesz typ danych dla zmiennej, stałej czy argumentu za pomocą dwukropka (:) po nazwie obiektu, wskazuje to kompilatorowi, jaki rodzaj informacji powinien być przechowywany w tym obiekcie. Funkcje działają w ten sam sposób.

Wszystkie funkcje definiujące **zwracany typ** muszą honorować go poprzez zwracanie wartości za pomocą instrukcji `return`, odpowiadającej typowi danych. I odwrotnie: wszystkie funkcje używające instrukcji `return` powinny posiadać typ zwracany. Zwracany typ — podobnie jak typ dołączony do każdego argumentu — wskazuje kompilatorowi, jakiego rodzaju danych ma się spodziewać, gdy funkcja zwróci wartość.

Konstruktory są przypadkiem specjalnym. Może Cię kusić użycie nazwy klasy jako zwracanego typu dla Twojego konstruktora, jednak ActionScript nakazuje pominięcie typu zwracanego dla wszystkich konstruktorów — np. `public function MyClass() {...}`.

Więcej informacji na temat konstruktorów znajdziesz w rozdziale 3.

W AS3 ścisłe typowanie zmiennych i funkcji nie jest wprawdzie wymagane, ale wysoce zalecane. Zauważysz, że Twój kod będzie wykonywał się szybciej przy zastosowaniu sprawdzania typów w czasie wykonywania — nie wspominając już o tym, że będzie prostszy w czytaniu, debugowaniu, użyciu i ponownym zastosowaniu.

Jeśli nie jesteś pewien, jakiego typu dane będziesz zwracać, możesz użyć typu danych reprezentowanego przez wieloznacznik (*), który odpowiada danym bez przypisanego typu. Umożliwi to zwracanie dowolnego typu danych.

W przedstawionym poniżej przykładzie typ utworzonych danych jest przekazywany do wywołania funkcji. Jak widzisz, funkcja ta posiada wiele zwracanych typów, ale tylko pierwszy pasujący zostanie wykonany. Reszta nigdy nie zostanie uaktywniona.

```
function createObject(type:String):* {
  switch (type) {
    case "Boolean": return new Boolean();
    case "Number": return new Number();
    case "String": return new String();
    case "Array": return new Array();
    default: trace("Nieznany typ"); return null;
  }
}
var myObject:* = createObject("Array");
myObject.push(1, 2, 3, 4, 5);
trace(myObject); // wyświetli: 1, 2, 3, 4, 5
```

Typ reprezentowany przez wieloznacznik może być w pewnych okolicznościach bardzo pożyteczny, jednak nie powinieneś go nadużywać. Większość Twoich metod powinna posiadać określony typ danych.

Zwracanie void

Jeśli nie potrzebujesz, żeby Twoja funkcja coś zwracała, użyj typu zwracanego `void`. Dla funkcji zwracających `void` nie jest wymagana instrukcja `return`.

```
function doNothing():void {
  // Nie będę robić niczego.
}
```

Nawet jeśli ten przykład „nie robi niczego", funkcje niezwracające wartości są bardzo powszechnie stosowane. To, że nie zwracają one wartości, wcale nie oznacza, że nie wykonują pożytecznego kodu. W rzeczywistości duża część funkcji w Twoim kodzie prawdopodobnie nie będzie posiadać instrukcji zwracających. Czasami funkcje niezwracające wartości nazywa się **procedurami** — małymi, nadającymi się do wielokrotnego stosowania blokami kodu, które mają na celu zamknięcie w sobie często wykonywanych zadań.

Od ActionScript 3.0 typ danych `Void` jest pisany małą literą, co służy większej jednolitości z innymi językami programowania. Jeśli przyzwyczaiłeś się do pisania kodu w AS2, uważaj, aby nie popełnić błędu, i nie miej złego samopoczucia, jeśli początkowo kilka takich pomyłek Ci się przytrafi.

Definiowanie funkcji za pomocą wyrażeń funkcyjnych

Do tej pory przyglądaliśmy się definiowaniu funkcji za pomocą instrukcji `function`. Istnieje jednak alternatywny sposób definiowania funkcji. ActionScript obsługuje również wyrażenia funkcyjne. Posługując się tą techniką, najpierw definiujesz funkcję jako zmienną o typie danych `Function`, a następnie ustawiasz wartość tej zmiennej na wyrażenie funkcyjne.

```
var doSomething:Function = function(arg:Object):void {
  // kod funkcji umieszczamy tutaj.
}
```

Wyrażenia funkcyjne zachowują się trochę odmiennie od instrukcji funkcyjnych. W tabeli 4.1 zamieszczono kilka kluczowych różnic.

Dla większości zastosowań zalecamy używanie w kodzie instrukcji funkcyjnych zamiast wyrażeń funkcyjnych. Spójność i prostota ich stosowania czynią je pierwszorzędnym wyborem. Istnieją jednak pewne sytuacje, w których zastosowanie wyrażenia funkcyjnego da Ci elastyczność i pożyteczność, jakiej instrukcja funkcyjna nie potrafi dać. Można do nich zaliczyć:

- Jednorazowe użycie danej funkcji.
- Zdefiniowanie funkcji wewnątrz innych funkcji.

Tabela 4.1. *Różnice pomiędzy instrukcjami funkcyjnymi a wyrażeniami funkcyjnymi*

Instrukcja funkcji	Wyrażenie funkcyjne
Może być wywoływane z dowolnego miejsca w kodzie, niezależnie od tego, gdzie w kodzie zdefiniowana jest funkcja.	Podobnie jak deklaracje zmiennej dostępne jest tylko po wystąpieniu kodu, w którym funkcja została zdefiniowana.
Łatwiejsza do czytania.	Może być trudna i kłopotliwa w czytaniu.
Może być wywoływana jako metoda za pomocą składni kropkowej.	W trybie standardów nie może być wywoływana jako metoda. Zamiast tego używa się składni kwadratowych nawiasów.
Istnieje w pamięci jako część obiektu, w którym jest zawarta.	Istnieje w pamięci niezależnie. Dobre dla funkcji „do wyrzucenia", które nie muszą być zachowywane w pamięci na długo.
Nie może zostać nadpisana lub usunięta.	Można mu przypisać nową wartość lub usunąć je za pomocą operatora `delete`.

- Dołączenie metody do prototypu klasy.
- Zmianę funkcjonalności metody w czasie wykonywania.
- Przekazywanie funkcji jako argumentu dla innej funkcji.

Dostęp do metod superklasy

Zgodnie z tym, czego dowiedziałeś się z rozdziału 3., klasy ActionScript posiadają zdolność rozszerzania funkcjonalności innej klasy. Klasa rozszerzająca funkcjonalność nazywana jest podklasą, a klasa rozszerzana nazywana jest superklasą. Koncepcja ta zwana jest **dziedziczeniem**.

Podczas tworzenia klasy masz do czynienia ze specjalnym mechanizmem, który może zostać wykorzystany do uzyskiwania dostępu do metod superklasy. Jest nim słowo kluczowe `super`. Słowo kluczowe `super` pozwala na wywołanie lub rozszerzenie istniejących metod zdefiniowanych w superklasie. Gdy używane jest samodzielnie, odwołuje się do funkcji konstruktora superklasy. Oznacza to, że może zostać użyte do wywołania całego kodu w funkcji konstruktora superklasy lub klasy rodzica. Jest to najbardziej powszechne zastosowanie.

W poniższym przykładzie kod powinien być zapisany w pliku *MySprite.as*:

```
package {
  import flash.display.*;
  public class MySprite extends Sprite {
    public function MySprite() {
      super();
      x = 100;
      y = 100;
    }
  }
}
```

W tym przykładzie `super()` zostało użyte do wywołania całego kodu funkcji konstruktora klasy `Sprite`, po czym następuje ustawienie `x` i `y` dla `MySprite` na wartość 100.

Obiekt `super` może zostać także użyty do uzyskania dostępu do pozostałych metod superklasy. Użyj po prostu składni `super.method()` do wywołania metody superklasy. W poniższym kodzie *przesłaniamy* metodę `addChild()` klasy `Sprite`, aby wyświetlić obiekt potomka, który dodajemy:

```
package {
  import flash.display.*;
  public class MySprite extends Sprite{
    override public function addChild(child:DisplayObject):DisplayObject {
      var addedChild:DisplayObject = super.addChild(child);
      trace("Właśnie dodany potomek " + addedChild.toString());
      return addedChild;
    }
  }
}
```

Poprzez wywołanie `super.addChild(child)` przekazujemy parametr `child` do metody `addChild()` superklasy. W rezultacie wykonany zostaje oryginalny kod `addChild()`, a jego wynik jest zapisywany do zmiennej `addedChild`. Następnie wyświetlana jest wartość tekstowa `addedChild`. Na końcu jest ona zwracana.

Przesłanianie metod

Być może zwróciłeś uwagę w poprzednim przykładzie na to, iż sygnatura funkcji dla metody `addChild()` została poprzedzona słowem kluczowym `override`. Powoduje to, że definiowana funkcja przesłania lub ponownie definiuje metodę superklasy.

Nowa metoda musi posiadać dokładnie taką samą **sygnaturę** (listę parametrów funkcji, zwracany typ itd.), co metoda przesłaniana. Jeśli nie masz pewności co do dokładnej sygnatury metody, zajrzyj do dokumentacji w celu sprawdzenia budowy tej metody.

Słowo kluczowe `override` musi być umieszczone przed wszystkimi przesłanianymi metodami. Pominięcie go spowodowałoby błąd podczas kompilowania. Metody zdefiniowane za pomocą słowa kluczowego `private` (a w pewnych przypadkach `internal`) nie wymagają słowa kluczowego `override`, ponieważ nie są przekazywane w dół do podklas.

Pisanie funkcji rekurencyjnych

Czasami funkcja w celu zwrócenia wartości musi być wywoływana wielokrotnie, a jej dane wejściowe są zależne od wyniku jej poprzednich wywołań. Taka sprytna forma projektowania funkcji znana jest jako **rekurencja**. Funkcje rekurencyjne są funkcjami wywołującymi siebie wewnątrz swojego ciała funkcji w celu utworzenia wyniku bazującego na wielu poziomach rekurencji. Jest to zaawansowana technika i początkującym programistom może pierwotnie wydawać się dziwna, dlatego jeśli chcesz, możesz pominąć to zagadnienie i powrócić do niego, jeśli kiedykolwiek będziesz musiał dowiedzieć się czegoś więcej na ten temat.

Oto kilka zastosowań, w których mógłbyś użyć funkcji rekurencyjnej:

- Podczas przemieszczania się po zagnieżdżonych danych, jak w przypadku pliku XML czy listy właściwości.
- Podczas obliczania funkcji matematycznych, które bazują na wielu iteracjach, jak zagadnienie silni czy obliczenia statystyczne.
- Funkcje wymagające wracania do poprzednich iteracji lub przechodzące przez złożony zestaw możliwych rozwiązań, jak algorytm poruszania się po labiryncie.

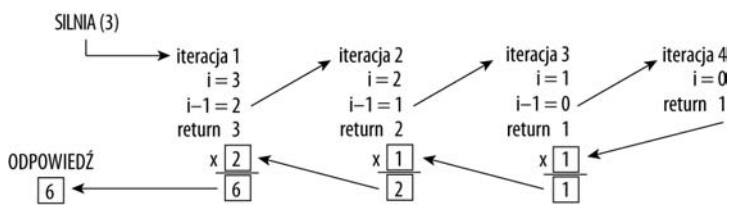
Przyjrzyjmy się często wykorzystywanej funkcji rekurencyjnej — silni. W matematyce funkcja silnia, zazwyczaj zapisywana jako dodatnia liczba całkowita, po której występuje znak wykrzyknika (`4!`), jest funkcją obliczającą iloczyn wszystkich dodatnich liczb całkowitych mniejszych bądź równych danej liczbie całkowitej. Innymi słowy, 3! = 3·2·1 = 6. Możemy ująć to w postaci funkcji rekurencyjnej:

```
function factorial(i:uint) {
  if (i == 0) {
    return 1;
  } else {
    return (i * factorial(i - 1));
  }
}
trace(factorial(3)); // wyświetli : 6
```

Co się dzieje, gdy wywołujesz tę funkcję? Przekazywana jest początkowa wartość `3`. Wartość `3` jest większa od `0`, dlatego pomijamy pierwszą instrukcję warunkową i przechodzimy do instrukcji `else`. Zwraca ona wartość `i` pomnożoną przez silnię `i-1` (2). Jednak przed zwróceniem wartości instrukcja silni uruchamiana jest ponownie drugi raz z wykorzystaniem `2` jako parametru. Cały proces rozpoczyna się od nowa. `factorial(2)` zwraca `i * factorial(i-1)` (lub 2 razy silnia z 1). Całość powtarzana jest tak długo, aż przetworzone zostaną wszystkie liczby całkowite i zakończy się pętla. Ostatecznie wszystkie zwrócone wartości zostaną wysłane w celu pomnożenia ich przez funkcję.

Leczenie z rekursivitis

Jak mogłeś się przekonać we wcześniejszym przykładzie, praca z funkcjami rekurencyjnymi może szybko się skomplikować, co znacznie utrudni wykrywanie błędów. Nazywam to **objawami rekursivitis** (nie, to nie jest prawdziwy termin medyczny). Jednym z rozwiązań tego problemu, a uważam je za przydatne, jest zapisywanie wartości zmiennych na zwykłej kartce papieru i analizowanie tego, co się wydarzy podczas każdej z iteracji. Dla wcześniej przedstawionego przykładu mógłbym utworzyć graf pokazany na rysunku 4.3.



Rysunek 4.3. *Leczenie z rekursivitis starym, sprawdzonym sposobem*

Jak widzisz, dla każdej iteracji śledzę zmiany poprzez ich zapisywanie. Analizowanie kodu w ten sposób jest o wiele prostsze. Z czasem, gdy Twoje zagadnienia będą coraz bardziej złożone, możesz skorzystać z wbudowanego debugera. Narzędzia debugujące opisujemy w rozdziale 20.

Pracując z funkcjami rekurencyjnymi, na pewno przynajmniej raz spotkasz się z problemem przepełnienia stosu. Występuje on wtedy, gdy funkcja nie posiada mechanizmu kończącego jej działanie. Problem podobny do tego to **pętle nieskończone**, które mogą wystąpić podczas pracy z pętlami `for` oraz `while`. Dokładniej opisaliśmy go w rozdziale 2. Flash Player posiada wbudowany mechanizm zatrzymujący każdą pętlę przed wykonaniem zbyt wiele razy. Jeśli Twoja funkcja rekurencyjna powtórzy się zbyt wiele razy, możesz uzyskać taką informację o błędzie:

```
Error: Error #1502: A script has executed for longer than the default timeout period
➥of 15 seconds.
```

Funkcje jako obiekty

Warto powtórzyć jeszcze raz: prawie wszystko w ActionScript jest obiektem. Funkcje nie są tutaj żadnym wyjątkiem. Każda funkcja w ActionScript 3.0 jest instancją klasy `Function` z właściwościami i metodami — jak każdy inny obiekt. W tym podrozdziale pokażemy, jak używać tej relacji z korzyścią dla siebie.

Function a function

Słowo *function* pisane małą literą jest ogólnym określeniem, którego używaliśmy do tej pory w tym rozdziale do opisywania wykonywalnego bloku kodu. *Function* pisane dużą literą oznacza klasę `Function`, której instancjami są wszystkie funkcje. Na ogół, tworząc nową instancję klasy, używa się słowa kluczowego `new`, po którym następuje nazwa klasy. Ponieważ funkcje są integralnym i często używanym elementem ActionScript, występują polecenia (jak słowo kluczowe `function`), które używane są zamiast instrukcji `new Function()`.

Technicznie zapis `new Function()`jest poprawny, ale nie robi zbyt wiele poza utworzeniem nowej, niezdefiniowanej funkcji.

Najbardziej prawdopodobne jest, że z nazwą klasy `Function` spotkasz się podczas pracy z metodami wymagającymi funkcji na wejściu lub podczas tworzenia funkcji za pomocą wyrażenia. Np. metoda `addEventListener()` pobiera jako parametr funkcję. Jeśli nie wiesz jeszcze, co to są zdarzenia, nie przejmuj się. Opiszemy je bardzo szczegółowo w części IV tej książki. Teraz przyjrzyjmy się, jakich argumentów funkcja oczekuje.

```
addEventListener(type:String, listener:Function):void
```

`addEventListener()` akceptuje pięć argumentów. W celu zachowania prostoty dołączono tutaj jedynie dwa wymagane.

Jak widzisz, drugi oczekiwany przez `addEventListener()` parametr jest funkcją nasłuchującą. Jest to funkcja, która zostanie wykonana, gdy wystąpi zdarzenie wskazanego typu (`type`). Jeżeli chcesz przekazać funkcję jako argument do innej funkcji, użyj nazwy przekazywanej funkcji, ale opuść nawiasy (operator wywołania).

```
function onClick(event:MouseEvent):void {
  // obsłuż kliknięcie
}
addEventListener(MouseEvent.CLICK, onClick);
```

Pamiętaj tutaj o pominięciu nawiasów. Jeśli napiszesz `addEventListener(MouseEvent.CLICK, onClick());`, wtedy funkcja `onclick()` zostanie wykonana i zwróci `void`, co zostanie przekazane jako Twój parametr (`void` oczywiście nie jest funkcją).

Metody i właściwości klasy Function

Klasa `Function` posiada niewiele właściwości i metod. Jako podklasa klasy `Object` dziedziczy wszystkie jej metody i właściwości, jak `toString()`. Oprócz tych dziedziczonych wartości klasa `Function` definiuje swoje własne metody i właściwości. Już przyglądaliśmy się `arguments`, jedynej właściwości definiowanej przez `Function`. Przyjrzyjmy się teraz jej metodom.

Klasa `Function` definiuje dwie metody: `call()` oraz `apply()`, będące alternatywnymi sposobami wywoływania funkcji. Jednak funkcje te nie działają tak samo w ActionScript 3.0, jak działały wcześniej w ActionScript 2.0. W większości przypadków `thisObject` jest kompletnie ignorowany. Zalecamy nie korzystać z tych metod. Jeśli musisz wywołać metody niezdefiniowane w klasie, rozważ użycie klasy `flash.util.Proxy`. Więcej informacji na ten temat znajdziesz w oficjalnej dokumentacji ActionScript 3.0.

Podsumowanie

- Metody oraz funkcje są powtarzalnymi operacjami wykonującymi bloki kodu w oparciu o przekazane parametry wejściowe.
- Użycie operatora `return` pozwoli Ci obliczyć i zwrócić wartości w zależności od zmiennych wprowadzonych do Twojej funkcji.
- Metody — podobnie jak większość innych elementów w ActionScript 3.0 — są obiektami. Należą do klasy `Function`.
- Funkcje można tworzyć za pomocą instrukcji `function` lub poprzez przypisanie wartości do obiektu typu `Function` za pomocą wyrażeń funkcyjnych.
- Parametr `...` (reszta) może zostać użyty do uzyskania dostępu do wszystkich argumentów funkcji, jeżeli liczba argumentów jest nieznana lub zmienna.
- Słowa kluczowe `super` oraz `override` mogą zostać użyte przez podklasę w celu dodania funkcjonalności do metody superklasy.
- Funkcje rekurencyjne umożliwiają powtarzanie akcji i łączenie wyników wygenerowanych przez jej poprzednie wywołania.