CHAPTER_8

익명 메소드

이 장에서는 제네릭과 함께 델파이 2009에서 추가된 문법 요소인 익명 메소드에 대해 알아봅니다. 익명 메소드 변수의 바인드 동작과 익명 메소드가 어떻게 유용하게 활용될 수 있는지도 살펴봅니다.

■익명 메소드의 문법 ■익명 메소드의 사용 ■익명 메소드 변수의 바인딩
■익명 메소드의 유용성

이름이 시사하는 것처럼, 익명 메소드(anonymous method)는 이름이 연관되지 않은 프로시저나 함수입니다. 익명 메소드는 어떤 코드 블록을 변수에 대입될 수 있거나 메소드의 파라미터로 사용될 수 있는 엔티티로 취급합니다. 또한 익명 메소드는 그 메소드가 정의된 문맥에서 변수나 변수에 지정된 값을 가리킬 수도 있습니다. 익명 메소드는 단순한 문법으로 정의되고 사용될 수 있습니다. 익명 메소드는 다른 언어들의 클로저(closure) 문법과 유사합니다.

익명 메소드의 문법

익명 메소드는 일반 프로시저나 함수와 비슷하게 정의되지만 이름이 지정되지 않습니다. 예를 들어, 아래 함수는 익명 메소드로 정의된 함수를 리턴합니다.

```
function MakeAdder(y: Integer): TFuncOfInt;
begin
  Result := { 익명 메소드의 시작 } function(x: Integer)
```

```
  begin
    Result := x + y;
  end; { 익명 메소드의 끝 }
end;
```

MakeAddr 함수는 이름 없이 선언된 함수, 즉 익명 메소드를 리턴합니다.
MakeAddr가 TFuncOfInt 타입의 값을 리턴한다는 것을 주목하십시오. 익명 메소드 타입은 메소드에 대한 참조로 선언됩니다.

```
type
  TFuncOfInt = reference to function(x: Integer): Integer;
```

위의 선언은 이 익명 메소드가 아래와 같다는 것을 나타냅니다.

- 함수입니다.
- 하나의 정수 파라미터를 받습니다.
- 정수 값을 리턴합니다.

일반적으로, 익명 함수 타입은 프로시저 혹은 함수에 대해 선언됩니다.

```
type
  TType1 = reference to procedure[(parameterlist)];
  TType2 = reference to function[(parameterlist)]: returntype;
```

아래는 타입들의 예입니다.

```
type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;
```

익명 메소드는 이름 없이 프로시저나 함수로 선언됩니다.

```
// 프로시저
procedure[(parameters)]
begin
```

```
  { 문장 블록 }
end;
// 함수
function[(parameters)]: returntype
begin
  { 문장 블록 }
end;
```

익명 메소드의 사용

익명 메소드는 다음의 예와 같이 일반적으로 어떤 것에 대입됩니다.

```
myFunc := function(x: Integer): string
begin
  Result := IntToStr(x);
end;

myProc := procedure(x: Integer)
begin
  Writeln(x);
end;
```

익명 메소드는 또한 함수에 의해 리턴되거나 메소드를 호출할 때 파라미터 값으로 전달될 수도 있습니다. 예를 들어, 아래는 바로 위에서 정의한 익명 메소드 변수 myFunc를 사용하는 코드입니다.

```
type
  TFuncOfIntToString = reference to function(x: Integer): string;

procedure AnalyzeFunction(proc: TFuncOfIntToString);
begin
  { 어떤 코드 }
end;

// 익명 메소드를 변수로서 파라미터로 전달하면서 프로시저를 호출
AnalyzeFunction(myFunc);

// 익명 메소드를 직접 사용:
AnalyzeFunction(function(x: Integer): string
begin
  Result := IntToStr(x);
end;)
```

익명 메소드 뿐만 아니라 메소드 참조도 메소드 변수에 대입될 수 있습니다. 예를 들면 다음과 같습니다.

```
type
  TMethRef = reference to procedure(x: Integer);
TMyClass = class
  procedure Method(x: Integer);
end;

var
  m: TMethRef;
  i: TMyClass;
begin
  // ...
  m := i.Method;    // 메소드 참조에 대입
end;
```

하지만, 그 반대는 되지 않습니다. 익명 메소드를 일반 메소드 포인터에 대입할 수는 없습니다. 메소드 참조는 매니지드 타입이지만 메소드 포인터는 언매니지드 타입입니다. 따라서 타입 안전성의 이유로 메소드 참조를 메소드 포인터로 대입하는 것은 지원되지 않습니다. 예를 들어, 이벤트는 메소드 포인터 값 속성이므로, 익명 메소드를 이벤트로 사용할 수 없습니다. 이 제한에 대한 더 자세한 정보를 찾아보려면 "익명 메소드 변수의 바인딩" 절을 참고하십시오.

익명 메소드 변수의 바인딩

익명 메소드의 핵심 기능은 익명 메소드가 정의된 위치에서 보이는(visible) 변수들을 참조할 수 있다는 것입니다. 더욱이, 이들 변수들은 익명 메소드에 대한 참조에 바인드(bind)되고 구속됩니다. 따라서 익명 메소드는 상태를 캡쳐(capture)하고 변수의 수명을 연장하게 됩니다.

변수의 바인딩

위에서 선언했던 함수를 다시 살펴봅시다.

```
function MakeAdder(y: Integer): TFuncOfInt;
begin
```

```
  Result := function(x: Integer)
  begin
    Result := x + y;
  end;
end;
```

변수 값이 연결(bind)된 함수의 이 인스턴스를 생성할 수 있습니다.

```
var
  adder: TFuncOfInt;
begin
  adder := MakeAdder(20);
  Writeln(adder(22));  // 42 출력
end.
```

adder 변수는 익명 메소드의 코드 블록에서 참조된 변수 y에 값 20이 연결(bind)된 익명 메소드를 가지고 있습니다. 이 연결(bind)은 그 값이 유효범위 바깥으로 나가더라도 계속 유지됩니다.

이벤트로서의 익명 메소드

메소드 참조를 사용하는 동기는 연결된 변수를 포함할 수 있는 타입을 가지기 위해서입니다. 이것은 클로저(closure)라고도 알려져 있습니다. 클로저는 그 정의된 환경을 정의된 시점에 참조된 모든 지역 변수들과 함께 가두기 때문에, 해제되어야 할 상태를 가지게 됩니다. 메소드 참조는 매니지드 타입이므로(참조 카운트가 됩니다), 메소드 참조는 이 상태를 추적하여 필요할 경우 해제할 수 있습니다. 메소드 참조나 클로저가 이벤트 같은 메소드 포인터에 자유롭게 대입될 수 있으면, 엉뚱한 위치를 가리키는 포인터(dangling pointer)나 메모리 누수(memory leak)를 가진 잘못된 프로그램을 만들기 쉬워집니다.

델파이의 이벤트는 속성에 어떤 조건이 추가된 것입니다. 타입의 종류 외에는 이벤트와 속성 사이에는 아무런 차이가 없습니다. 어떤 속성이 메소드 포인터 타입이면 이벤트가 됩니다.

속성이 메소드 참조 타입이면, 논리적으로 이벤트도 고려해야 합니다. 하지만 IDE는 메소드 참조 타입을 이벤트로 다루지 않습니다. 이것은 IDE에 컴포넌트와 커스텀 컨트롤로 설치된 클래스에 대해 중요합니다.

따라서, 메소드 참조나 클로저 값으로 대입될 수 있는 컴포넌트나 커스텀 컴포넌트에서 이벤트를 가지려면, 그 속성은 메소드 참조 타입이어야 합니다. 하지만, IDE가 이벤트로 인식하지 못하기 때문에 이런 방법은 편리하지 않습니다.

아래의 코드는 메소드 참조 타입의 속성을 사용하여 이벤트로서 동작하는 예제입니다.

```
type
  TProc = reference to procedure;
  TMyComponent = class(TComponent)
  private
    FMyEvent: TProc;
  public
    // MyEvent 속성은 이벤트로 동작합니다
    property MyEvent: TProc read FMyEvent write FMyEvent;
    // 이벤트에 일반적인 패턴으로 FMyEvent를 호출하는 다른 코드들
  end;

...

var
  c: TMyComponent;
begin
  c := TMyComponent.Create(Self);
  c.MyEvent := procedure
  begin
    ShowMessage('Hello World!'); // TMyComponent가 MyEvent를 호출할 때 보여짐
  end;
end;
```

변수 바인딩 메커니즘

메모리 누수(memory leak)가 발생하는 것을 피하기 위해, 변수의 바인딩 과정을 더 자세히 알아보는 것도 유용합니다.

프로시저, 함수 혹은 메소드(이하 "루틴")의 시작 부분에서 정의된 지역 변수들은 일반적으로 해당 루틴이 활성인 상태에서만 존재합니다. 익명 메소드는 이런 변수들의 수명을 늘릴 수 있습니다.

익명 메소드가 자신의 바디 내에서 외부의 지역 변수를 참조하면, 그 변수는 "캡쳐(capture)"됩니다. 캡쳐는 변수의 수명을 연장하는 것을 의미하므로, 선언된 루틴과 함께 사라지지 않고 익명 메소드 값만큼의 수명을 가지게 됩니다. 변수 캡쳐는 값이 아니라 변수를 캡쳐한다는 것에 주의하십시오. 익명 메소드를 구축하여 변수가 캡쳐된 후에 그 값이 변경되면, 익명 메소드가 캡쳐한 변수의 값도 변경됩니다. 이것은 두 변수가 동일한 메모리에 존재하는 동일 변수이기 때문입니다. 캡쳐된 변수는 스택이 아닌 힙에 저장되게 됩니다.

익명 메소드 값은 메소드 참조 타입이며 참조 카운트가 됩니다. 지정한 익명 메소드 값에 대한 마지막 메소드 참조가 유효 범위를 벗어나면, 혹은 제거(nil로 초기화)되거나 종료화(finalize)되면, 그 익명 메소드가 캡쳐한 변수들은 마침내 유효 범위를 벗어나게 됩니다.

이 상황은 여러 익명 메소드가 동일한 지역 변수를 캡쳐하는 경우에 더욱 복잡해집니다. 모든 상황에서 이 동작이 어떻게 이루어지는지 이해하려면, 익명 메소드 구현의 역학에 대해 더욱 세세하게 따져볼 필요가 있습니다.

지역 변수가 캡쳐될 때마다 그 지역 변수는 선언중인 루틴과 연관되는 "프레임 객체"에 추가됩니다. 한 루틴에서 선언된 모든 익명 메소드는 포함하는 루틴과 연관된 프레임 객체에서 메소드로 변환됩니다. 마지막으로, 익명 메소드가 구축되어서 혹은 변수가 캡쳐되어서 생성된 모든 프레임 객체들은, 부모 프레임이 존재하고 캡쳐된 외부 변수를 액세스할 필요가 있을 경우, 또다른 참조에 의해 부모 프레임에 연결됩니다. 하나의 프레임 객체로부터 그 부모로의 이런 연결은 레퍼런스 카운트가 됩니다. 부모 루틴으로부터 변수를 캡쳐하는 중첩된 지역 루틴에서 선언된 익명 메소드는 그 자신이 유효범위를 벗어날 때까지 부모 프레임 객체가 유효하도록 유지합니다.

예를 들면, 다음과 같은 상황을 생각해봅시다.

```
type
  TProc = reference to procedure;

procedure Call(proc: TProc);
// ...
procedure Use(x: Integer);
// ...

procedure L1;       // 프레임 F1
var
  v1: Integer;

  procedure L2;     // 프레임 F1_1
  begin
    Call(procedure // 프레임 F1_1_1
    begin
      Use(v1);
    end);
  end;

begin
  Call(procedure   // 프레임 F1_2
  var
    v2: Integer;
  begin
    Use(v1);
    Call(procedure // 프레임 F1_2_1
    begin
      Use(v2);
    end);
  end);
end;
```

각 루틴과 익명 메소드에는 어떤 프레임 객체가 어디에 링크되는지를 알아보기 쉽도록 프레임 식별자로 주석을 달았습니다.

- v1은 F1 내의 변수입니다.
- v2는 F1_2 내의 변수입니다. (F1_2_1에 의해 캡쳐됨)
- F1_1_1의 익명 메소드는 F1_1 내의 메소드입니다.
- F1_1은 F1에 링크됩니다. (F1_1_1이 v1을 사용함)
- F1_2의 익명 메소드는 F1 내의 메소드입니다.
- F1_2_1의 익명 메소드는 F1_2 내의 메소드입니다.

프레임 F1_2_1 및 F1_1_1은 프레임 객체를 필요로 하지 않는데, 그것은 이 프레임들은 익명 메소드를 선언하지도 않고 캡쳐되는 변수를 갖지도 않기 때문입니다. 이 프레임들은 중첩된 익명 메소드와 바깥의 캡쳐된 변수 사이에서 부모 지위를 가질 패스가 전혀 없습니다. (이들은 스택에 저장되는 암시적인 프레임을 갖습니다.)
익명 메소드 F1_2_1에 참조만 주어지면 변수 v1와 v2는 수명이 유지됩니다. 만약 F1의 호출보다 더 오래 남아있는 유일한 참조가 F1_1_1이라면, 변수 v1만이 살아 남습니다.
메소드 참조/프레임 연결 체인 내에서 메모리 누수를 일으키는 순환(cycle)이 생길 수 있습니다. 예를 들면, 익명 메소드를 직접 혹은 간접적으로 익명 메소드 자신이 캡쳐하는 변수에 저장하면 순환이 일어나게 되고, 따라서 메모리 누수가 발생됩니다.

익명 메소드의 유용성

익명 메소드는 호출 가능한 어떤 것에 대한 단순한 포인터 이상을 제공합니다. 익명 메소드는 몇가지 장점을 제공합니다.

- 변수 값들을 연결(bind)할 수 있습니다.
- 메소드를 정의하고 사용하는 편리한 방법입니다.
- 코드를 파라미터화하는 것이 쉽습니다.

변수의 바인드

익명 메소드는 코드 블록에 그 코드와 변수가 정의된 환경에 대한 변수 연결(binding)을 제

공하며, 심지어 그 환경이 유효범위 내에 있지 않은 경우에도 마찬가지입니다. 함수나 프로시저에 대한 포인터는 이런 역할을 할 수 없습니다.
예를 들어, 위의 코드 예제에서 adder := MakeAdder(20); 문장은 변수의 값 20로의 바인딩을 캡슐화하는 변수 adder를 만듭니다.
몇몇 다른 언어들에서는 이런 코드 구성을 클로저(closure)라고 부릅니다. 역사적으로, adder := MakeAdder(20);와 같은 표현식을 계산하는 것은 닫힌 영역(closure)을 만들어냈다는 데에서 착안한 것입니다. 클로저는 함수 내에서 참조되고 그 외부에서 선언되는 모든 변수들의 바인딩에 대한 참조를 가지는 객체를 나타내며, 변수들의 값을 캡쳐함으로써 닫히게 됩니다.

사용의 편리함

다음의 예제는 단순한 메소드 몇 개를 갖는 일반적인 클래스와 그 메소드들을 호출하는 것을 보여줍니다.

```
type
  TMethodPointer = procedure of object; // void TMethodPointer();를 대리함
  TStringToInt = function(x: string): Integer of object;

TObj = class
  procedure HelloWorld;
  function GetLength(x: string): Integer;
end;

procedure TObj.HelloWorld;
begin
  Writeln('Hello World');
end;

function TObj.GetLength(x: string): Integer;
begin
  Result := Length(x);
end;

var
  x: TMethodPointer;
  y: TStringToInt;
  obj: TObj;

begin
  obj := TObj.Create;

  x := obj.HelloWorld;
  x;
  y := obj.GetLength;
  Writeln(y('foo'));
end.
```

위 코드를 익명 메소드를 사용하여 동일한 메소드를 정의하고 호출하는 코드와 비교해봅시다.

```
type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;

var
  x1: TSimpleProcedure;
  y1: TSimpleFunction;

begin
  x1 := procedure
    begin
      Writeln('Hello World');
    end;
  x1;    //방금 정의된 익명 메소드를 호출

  y1 := function(x: string): Integer
    begin
      Result := Length(x);
    end;
  Writeln(y1('bar'));
end.
```

익명 메소드를 사용하는 이번 코드가 얼마나 간단해지고 짧아졌는지 살펴보시기 바랍니다. 이 방법은 다른 곳에서는 전혀 사용되지 않을 클래스를 생성하는 오버헤드와 수고 없이 이런 메소드들을 명확하고 간단하게 정의하려 할 때 이상적입니다. 이 코드는 더 이해하기 쉽기도 합니다.

코드를 파라미터로 사용

익명 메소드는 값 뿐만 아니라 함수와 구조를 코드로 파라미터화하는 코드를 작성하기 쉽게 해줍니다.

멀티 스레딩은 익명 메소드를 활용하기에 좋은 애플리케이션입니다. 어떤 코드를 병렬로 실행하려 하는 경우 다음과 같은 병렬 순환(parallel-for) 함수가 필요할 수 있습니다.

```
type
  TProcOfInteger = reference to procedure(x: Integer);

procedure ParallelFor(start, finish: Integer; proc: TProcOfInteger);
```

ParallelFor 프로시저는 여러 스레드들에서 한 프로시저를 반복합니다. 이 프로시저가 스

레드 혹은 스레드 풀을 이용하여 적절하고 효율적으로 구현되었다면, 간단히 멀티 프로세서의 이점을 활용할 수 있게 됩니다.

```
procedure CalculateExpensiveThings;
var
  results: array of Integer;
begin
  SetLength(results, 100);
  ParallelFor(Low(results), High(results),
    procedure(i: Integer)                                    //
    begin                                                    //   파라미터로 사용될 코드 블록
      results[i] := ExpensiveCalculation(i);                 //
    end                                                      //
    );
  // results를 사용
  end;
```

이 코드를 익명 메소드를 사용하지 않고 구현하려면 얼마나 복잡한 코드가 될지 생각해보십시오. 아마도 가상 추상 메소드를 가진 "작업" 클래스와 ExpensiveCalculation을 위한 구체화 자손 클래스, 그리고 모든 작업들을 큐에 추가하는 작업까지… 자연스럽고 통합된 코드는 아닐 것입니다.

여기서, 병렬 순환(parallel-for) 알고리즘은 코드로 파라미터화된 추상화 구현입니다. 과거에는 이런 패턴을 구현하는 일반적인 방법은 하나 이상의 추상(abstract) 메소드를 갖는 가상(virtual) 기반 클래스를 이용하는 방법이었습니다. TThread 클래스와 그 추상 Execute 메소드를 생각해보십시오. 하지만, 익명 메소드는 이런 패턴을 알고리즘과 데이터 구조를 코드로 파라미터화하여 훨씬 쉽게 해줍니다.