JavaFX

CHAPTER 03

JavaFX 첫걸음

"나는 아직 내 경력의 시작 단계에 있다. 모든 것이 새롭고, 나는 여전히 가야 하기에 배운다."

—*Orlando Bloom*

JavaFX 스크립트 기초 40
JavaFX 스크립트 언어 41
클래스 선언 43
믹스인 클래스 46
객체 상수 48
변수 49
시퀀스 52
함수 57
스트링 58
표현식과 연산자 62
요약 72

배경지식–옮긴이 작성

3장에서의 다소 생소한 개념은 믹스인 클래스이다. 믹스인 클래스는 구현 부분인 몸체를 가지는 인터페이스와 같은 개념으로 생각하거나, C/C++ 언어에서의 함수 모음인 C/CPP 파일로 보면 무난할 것으로 보인다. 즉, 공통적인 부분을 미리 구현해 놓은 후 상속하여 원하는 함수를 오버라이딩해서 사용하거나 원치 않는 부분은 오버라이딩하지 않아도 되는 점이 인터페이스나 추상 클래스와는 다른 점이다.

컴파일을 하면 "클래스명$Mixin.class" 형태의 함수가 생성되는 것도 다른 점이다. 다시 말하자면, 일종의 함수 모음집으로 상속한 모든 믹스인 클래스의 함수와 요소들은 클래스의 일부분으로 속하여 컴파일된다.

이 외에 객체 상수 개념은 자바스크립트에 익숙한 프로그래머라면 이미 사용하고 있는 방식이며, 이 단원에서는 JavaFX의 기본 구문에 대해 설명하고 있으므로 이미 알고 있는 개념이라 하더라도 꼼꼼히 살펴보기 바라며, 특히 일반적인 프로그래밍 언어에서는 제공하지 않는 시퀀스 등의 개념을 살펴보는 것도 좋을 것이다.

JavaFX 스크립트 기초

JavaFX는 부분적으로 선언형 언어이다. 선언형 언어를 사용하여 개발자는 할일을 기술하고 시스템이 처리하도록 알린다. Chalmers 대학교의 인터렉션 디자인 기술 마스터 프로그램의 프로그램 이사와 Göteborg 대학교 부교수를 역임하고 있는 Olof Torgersson는 10년 넘게 선언형 프로그래밍을 연구해오고 있다. 선언형 프로그래밍에 관한 그의 연구 결과에서 우리는 다음의 정의를 찾았다.

> "프로그래머 관점에서 기본 속성은 프로그래밍을 보다 높은 레벨로 추상화하는 것이다. 높은 레벨의 추상화에서 프로그래머는 어떻게 계산되는지 알 필요가 없고, 무엇을 계산할 것인지에만 집중할 수 있다."[역주 1)]

JavaFX 스크립트에는 객체지향과 선언형 프로그래밍의 개념이 섞여 있다. 이것은 높은 생산성은 물론 유연하고 튼튼한 애플리케이션의 기초를 제공한다. 그러나 이런 유연함은

역주 1) Torgersson, Olof. "A Note on Declarative Programming Paradigms and the Future of Definitional Programming," Chalmers University of Technology and Göeborg University, Göeborg, Sweden. http://www.cs.chalmers.se/~oloft/Papers/wm96/wm96.html.

개발자의 책임이 필요하다. JavaFX 스크립트는 관대한 언어이고 선언형이기에 알기 어려운 프로그래밍 실수를 만드는 태생적인 규칙을 포함한다. 대부분의 경우는 널 객체가 런타임 엔진에서 처리되고 Java 널 포인터 예외처리가 거의 일어나지 않기 때문이다. 그 결과 프로그램은 표현식 안에서 널을 만날 때까지 계속되고 유효한 결과를 만들어낸다. 그러나 이 결과는 개발자가 기대한 값이 아닐 수도 있다. 그러므로 개발자는 코드를 작성할 때 더 주의해야 하고 더 많이 테스트해야 한다. 처음에는 이런 점을 주의해야 하지만 JavaFX는 쉬운 사용법과 높은 생산성을 지니고 있고 사용자가 문제를 겪지 않도록 노력하고 있어 이러한 단점을 충분히 상쇄한다.

선언형 언어인 JavaFX의 이점 중 하나는 객체가 상호작용을 하도록 만드는 많은 '연동' 방법을 언어 차원에서 지원한다는 것이다. 따라서 개발자는 어떻게 보여주는지에 얽매이지 않고 무엇을 보여줄지에 집중할 수 있다. 다음 절에서 구문, 연산자와 그 외의 특징들을 포함한 JavaFX 스크립트 언어의 전반적인 개요를 볼 수 있다.

JavaFX 스크립트 언어

이미 언급했듯이 JavaFX 스크립트는 객체지향을 지원하는 선언형 스크립팅 언어이다. 이미 Java, JavaScript, Groovy, Adobe Action Script, JRuby 등의 다른 언어를 잘 알고 있다면, JavaFX 스크립트가 낯익겠지만 JavaFX에는 중요한 차이점이 있다. 전통적인 순수 스크립팅을 지원하면서 또한 객체지향적인 캡슐화와 코드의 재사용성을 지원한다. 따라서 개발자가 작은 크기부터 큰 크기까지 애플리케이션을 만들고 유지보수할 수 있다. 다른 중요한 특징은 JavaFX 스크립트와 Java의 혼용이 부드럽게 이루어진다는 것이다.

개념적으로 JavaFX 스크립트는 스크립트와 클래스라는 두 개의 레벨로 나뉜다. 스크립트 레벨에서는 변수와 함수를 정의한다. 이 변수와 함수들은 같은 스크립트 안에서 정의한 다른 클래스에서 공유할 수 있고, 만약 스크립트가 더 큰 접근 권한을 가지고 있다면 다른 스크립트와 클래스에서도 공유할 수 있다. 추가적으로 loose라고 부르는 표현식이 만들어지는데, 이는 클래스 정의 밖에서 선언한 모든 표현식들이다. 스크립트가 수행될 때 모든 loose 표현식도 수행된다.

간단하게 'Hello World'를 콘솔에 나타내는 스크립트는 다음과 같다.

```
println("Hello World");
```

리스트 3.1은 3팩토리얼(3!)을 계산하는 것을 보여준다.

리스트 3.1 3팩토리얼

```
def START = 3;
var result = START;
var a = result - 1;
while(a > 0) {
    result *= a;
    a--;
}
println("result = {result}");
```

개발자 노트 DEVELOPER NOTE

만약 스크립트가 멤버들(외부 접근이 가능한 package, protected, public 등의 함수나 변수 같은 멤버들)를 익스포트하였다면 모든 **loose 표현식**이 run 함수에 반드시 포함되어야 한다. 예를 들어, 앞의 예제에서 public을 추가하기 위해 result 변수를 바꾼다면 run 함수를 만들어야 한다.

```
public var result:Number;

function run(args : String[]) : java.lang.Object {
    var num = if(sizeof args > 0) {
            java.lang.Integer.valueOf(args[0]);
        } else {
            10;
        };
    result = num;
    var a = result - 1;
    while(a > 0) {
        result *= a;
        a--;
    }
    println("{num}! = {result}");
}
```

run 메소드는 실행될 때 커맨드라인에서 스크립트에 전달한 인수들을 String[] 시퀀스 인수로 받는다.

멤버들을 익스포트하지 않더라도 run 메소드를 계속 포함할 수 있다. 그러나 run 메소드는 비록 접근 권한자가 포함되지 않았더라도 자체적으로 익스포트된 것으로 간주한다. 그래서 일단 run 메소드를 추가하면 모든 익스포트된 loose 표현식은 반드시 run 메소드 안에 포함되어야 한다.

스크립트 레벨과 별개로, 클래스는 인스턴스 변수와 함수를 정의하고 사용하기 전에 반드시 객체로 인스턴스화해야 한다. 클래스 함수나 변수는 같은 스크립트 파일 안의 스크립트 레벨 함수와 변수에 접근할 수 있고, 접근 권한을 받은 경우에는 다른 스크립트 파일 안의 함수와 변수도 접근 가능하다. 반면에 스크립트 레벨 함수는 클래스가 객체로 생성되고, 접근 권한을 제공하였을 때만 클래스 변수와 함수에만 접근할 수 있다. 접근 권한에 대한 정의는 이번 장 뒷부분에서 자세히 다룬다.

클래스 선언

JavaFX에서 클래스를 선언할 때는 *class* 키워드를 사용한다.

```
public class Title {
}
```

개발자 노트 DEVELOPER NOTE

관습상 클래스명의 첫 글자는 대문자로 쓴다.

접근 권한자 public 키워드는 클래스가 다른 스크립트 파일에서 정의되었더라도 이 클래스를 다른 클래스나 스크립트에서 사용될 수 있다는 것을 의미한다. 만약 클래스가 접근 권한자를 가지고 있지 않다면, 클래스가 선언된 스크립트 파일 안에서만 접근이 가능하다. 예를 들어, 리스트 3.2에서 Point 클래스는 외부에 보이는 권한자(public 같은)를 가지고 있지 않으므로 ArtWork 스크립트 안에서만 사용될 수 있다.

리스트 3.2 Artwork.fx

```
class Point { // private class only
              // visible to the ArtWork class
```

(계속)

```
    var x:Number;
    var y:Number;
}

public class ArtWork {
    var location: Point;
}
```

개발자 노트 DEVELOPER NOTE

각각의 JavaFX 스크립트 파일에서 비록 명백하게 정의하지 않더라도 스크립트 파일명을 사용하여 클래스가 생성된다. 앞의 ArtWork.fx 예제에서도 Artwork란 클래스가 있다. Public class Artwork 선언을 하지 않았어도 그렇게 됐을 것이다.

또한 같은 스크립트 파일 안에서 정의된 모든 클래스 이름에 스크립트 파일의 이름이 앞에 붙는다. 예를 들어, 앞의 예제에서 Point 클래스는 ArtWork.Point와 같이 패키지 경로를 포함한 클래스 이름으로 간주된다. 물론, ArtWork가 패키지에 속한다면 패키지 이름 또한 이름으로 간주될 것이다. 예를 들어 com.acme.ArtWork.Point가 되는 것이다.

클래스를 확장하려면 일반적인 클래스 이름 뒤에 extends라는 키워드를 사용하면 된다. JavaFX 클래스는 하나의 Java나 JavaFX 클래스로 확장할 수 있다. Java 클래스를 확장하면 클래스는 반드시 디폴트(매개변수가 없는) 생성자를 갖는다.

```
public class Porsche911 extends Porsche {
}
```

JavaFX는 다수의 JavaFX 믹스인 클래스나 Java 인터페이스로 확장한다. 믹스인 클래스는 다음 절에서 다룬다.

애플리케이션은 많은 클래스를 포함할 수 있어서 **패키지**Package라고 하는 일관된 방식으로 클래스들을 구성하는 것이 좋다. 클래스나 스크립트가 패키지에 속하도록 선언하려면 스크립트 파일의 시작부분에 패키지 선언을 포함한다. 다음 예제는 Title 클래스가 com.mycompany.components 패키지에 속한다는 것을 의미한다. 이제 Title 클래스의 완전한 이름은 com.mycompany.componets.Title이다. Title 클래스가 참조될 때마다 이 완전한 이름으로 풀어서 사용한다.

```
package com.mycompany.components;
```

```
public class Title {
}
```

이름을 풀어쓰는 것을 쉽게 하기 위해 import 문을 소스 파일의 윗부분에 포함한다.

```
import com.mycompany.components.Title;

var productTitle = Title{};
```

이제 Title이 스크립트 파일 안에서 참조되는 곳마다 com.mycompany.componetns.Title로 바뀌어 사용된다. 또한 와일드카드(*)를 써서 import 선언을 할 수 있다.

```
import com.mycompany.components.*;
```

와일드 카드를 포함한 import이기에 com.mycompany.components 패키지에서 어떤 클래스를 참조하더라도 완전한 이름으로 바뀌어 쓰인다. 다음의 예제 코드는 주석에 완전한 클래스 이름을 적어서 클래스 이름이 어떻게 풀어지는지 보여준다.

```
package com.mycompany.myapplication;
import com.mycompany.components.Title;

// com.mycompany.myapplication.MyClass
public class MyClass {
    // com.mycompany.components.Title
    public var title: Title;
}
```

클래스는 public 대신에 package 키워드를 사용하여 패키지 가시성을 가질 수 있다. 이것은 클래스가 오직 같은 패키지 안의 클래스로부터 접근될 수 있다는 의미이다.

```
package class MyPackageClass {
}
```

클래스는 또한 abstract으로 선언될 수 있는데, 이것은 이 클래스가 바로 인스턴스화될 수 없다는 것을 의미한다. 그러나 서브클래스들 중 하나를 사용하여 인스턴스화될 수 있다. 추상 클래스는 독립적으로 쓰일 수 없지만 여러 클래스에서 사용하도록 상태 정보와 함수의 일부분을 캡슐화한다. 추상 클래스의 서브클래스만이 인스턴스화될 수 있고, 일반적으로 서브클래스는 추상화 클래스에서 다루지 않는 고유의 정보와 기능을 담아야 한다.

```
public abstract class MyAbstractClass {
}
```

클래스가 abstract 함수를 선언한다면 이 클래스는 abstract으로 선언되어야 한다.

```
public abstract class AnotherAbstractClass {
    public abstract function
                setXY(x:Number, y:Number) : Void;
}
```

믹스인 클래스

JavaFX는 믹스인 상속이라는 상속의 형태를 지원한다. 이를 위해 JavaFX는 mixin이라는 특수한 타입의 클래스를 포함한다. mixin 클래스는 서브클래스에 상속시키는 특정 기능을 제공하는 클래스이다. 이 클래스는 스스로 인스턴스화될 수 없다. mixin 클래스는 함수들의 기본 구현을 제공하고, 변수를 선언하고 초기화한다는 점에서 Java 인터페이스와 다르다.

JavaFX에서 mixin 클래스를 선언하려면 클래스 선언에 mixin 키워드를 포함해야 한다.

```
public mixin class Positioner {
```

mixin 클래스는 많은 함수 선언을 포함할 수 있다. 함수 선언이 함수 몸체를 가지고 있으면 그것이 함수의 기본적인 구현이다. 예를 들어, 다음의 예제는 노드 안에 다른 노드가 들어 있는 mixin 클래스 선언이다.

```
public mixin class Positioner {
    protected bound function centerX(node: Node, within: Node) : Number {
        (within.layoutBounds.width - node.layoutBounds.width)/2.0 - node.layoutBounds.minX;
    }
    protected bound function centerY(node: Node, within: Node) : Number {
        (within.layoutBounds.height - node.layoutBounds.height)/2.0 - node.layoutBounds.minY;
    }
}
```

고유 버전의 mixin 함수를 서브클래스에 구현하려면 함수를 선언할 때 반드시 override 키워드를 사용해야 한다. 예를 들어, 다음의 코드는 Positioner mixin 클래스에서 centerx()란 함수의 고유 버전을 구현한 것이다.

```
public class My Positioner extends Positioner {
    public override bound function centerX(node: Node, within: Node) : Number {
        (within.boundsInParent.width - node.boundsInParent.width )/2.0;
    }
}
```

mixin 함수가 기본적인 구현을 하지 않고 있다면, abstract으로 선언되어야 하고 서브 클래스는 이 함수를 오버라이드해서 구현해야 한다. 예를 들어, 다음의 코드는 Positioner mixin 클래스에 추가한 abstract 함수를 보여준다.

```
public abstract function bottomY(node: Node, within: Node, padding: Number) : Number;
```

서브클래스는 다음의 코드에서 보여주듯이 override 키워드를 사용하여 이 함수를 구현해야 한다.

```
public class My Positioner extends Positioner {
    public override function bottomY(node: Node, within: Node, padding: Number) : Number {
            within.layoutBounds.height - padding - node.layoutBounds.height;
    }
}
```

두 mixin이 같은 함수나 변수 이름을 가지고 있다면 시스템은 어떤 mixin이 extends 절에서 먼저 선언되었는지에 따라 함수나 변수를 사용한다. 특정 함수나 변수를 지정하려면 mixin 클래스 이름을 함수나 변수 이름과 함께 사용하면 된다. 다음의 코드를 보자.

```
public class My Positioner extends Positioner, AnotherPositioner {
    var offset = 10.0;
    public override bound function
            centerX(node: Node, within: Node) : Number {
        Positioner.centerX(node, within) + offset;
    }
}
```

mixin은 변수를 선언할 때 디폴트 값과 트리거를 가지게 할 수도 있고 안 가지게 할 수도 있다. 서브클래스는 이 변수를 상속하거나 변수 선언을 오버라이드해야 한다. 다음의 예제를 보자.

```
public mixin class Positioner {
```

```
    public var offset: Number = 10.0;
}

public class My Positioner extends Positioner {
    public override var offset = 5.0 on replace {
        println("{offset}");
    }
}
```

클래스가 JavaFX 클래스와 하나 이상의 mixin 클래스를 확장한다면, JavaFX 클래스는 변수 초기화에서 mixin 클래스에 우선한다. 변수를 수퍼클래스에서 선언했다면 수퍼클래스에서 지정한 디폴트 값을 사용한다. 디폴트 값을 수퍼클래스에서 지정하지 않았다면 변수의 타입에 해당하는 '디폴트 값'이 사용된다. mixin 클래스에서 우선순위는 extends 절에서 정의한 순서를 따른다. mixin에서 선언된 변수가 디폴트 값을 가지고, 변수가 메인 클래스에서 디폴트 값 없이 오버라이드되었다면 초기값은 mixin에서 정한 값으로 지정된다.

mixin들은 또한 init와 postinit 블록을 가지고 있다. mixin init와 postinit 블록은 수퍼클래스의 init와 postinit 블록 후에, 그리고 서브클래스의 init와 postinit 블록 전에 실행된다. mixin 클래스의 init와 positnit 블록은 서브클래스에서 extends 절에서 선언된 순서로 실행된다.

객체 상수

JavFX에서 객체는 **객체 상수**object literals를 사용하여 인스턴스화한다. 객체 상수는 만들고 싶은 클래스 이름 뒤에 인스턴스에 대한 초기화와 선언 목록을 적는 선언적 구문이다. 리스트 3.3에서 Title 클래스의 객체는 스크린의 10, 50 위치에서 "JavaFX is cool"이란 텍스트를 가지고 만들어졌다. 마우스를 클릭하면 지정한 함수를 호출한다.

리스트 3.3 객체 상수

```
var title = Title {
      text: "JavaFX is cool"
      x: 10
```

(계속)

```
        y: 50
        onMouseClicked: function(e:MouseEvent):Void {
                // 여기에 사용자 정의 코드를 작성한다.
        }
};
```

객체 상수를 선언할 때 인스턴스 변수는 쉼표나 공백, 세미콜론으로 구분된다.

객체 상수 선언에서 추상함수를 오버라이드할 수 있다. 리스트 3.4에서 객체 상수는 java.awt.event.ActionListener 인터페이스의 객체를 생성하고, 추상 Java 메소드인 void actionPer-formed(ActionEvent e) 메소드를 오버라이드한다.

리스트 3.4 객체 상수–추상 함수 오버라이딩

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
var listener = ActionListener {
    override function actionPerformed(e: ActionEvent) : Void {
        println("Action Performed!");
    }
}
```

변수

JavaFX는 **인스턴스**instance와 **스크립트**script, 두 종류의 변수를 제공한다. 스크립트 변수는 스크립트 전체에 걸쳐 정보를 가지고, 인스턴스 변수는 스크립트 파일에서 선언한 클래스의 특정 인스턴스에서만 정보를 가진다.

기본적으로 변수는 두 종류이다. **바꿀 수 없는**unassignable 것과 **바꿀 수 있는**changeable 것. 바꿀 수 없는 변수는 def 키워드를 사용해서 선언하고 한번 값을 정하면 바꿀 수 없다.

```
public def PI = 3.14;
```

이 변수는 객체 상수에서 초기화, 오버라이드, 할당 모두 할 수 없다. 이런 점 때문에 이

변수를 상수로 볼 수 있지만 순수 상수는 아니고 바인딩을 할 수 있다(바인딩은 4장에서 자세히 다룬다).

다음 예제에서는 객체를 포함하는 **바꿀 수 없는** 변수를 선언한다. 객체 인스턴스는 변할 수 없지만 이 인스턴스의 상태도 변할 수 없는 것은 아니다.

```
def centerPoint = Point{x: 100, y:100};
centerPoint.x = 500;
```

실제로 centerPoint에 할당한 point 객체는 변하지 않고 남아있다. 그러나 x와 y값인 이 객체 인스턴스의 상태는 변할 수 있다. 그렇더라도 바인딩에 사용할 때의 centerPoint는 상수다. 따라서 centerPoint의 상태가 변하면 바인딩되어 있으므로 그 변화를 알 수 있다.

바꿀 수 있는 인스턴스 변수는 var 키워드를 사용하여 선언하며 디폴트 값을 지정할 수 있다. 디폴트 값이 없으면 적당한 디폴트 값을 사용한다. 수는 0, Boolean 디폴트는 false, String 디폴트는 빈 문자열, Sequence는 빈 시퀀스, 그리고 나머지 모두는 null로 지정된다.

스크립트 변수는 클래스 선언의 외부에서 선언하고 인스턴스 변수는 클래스 선언 안에서 선언한다. 스크립트 변수를 접근 권한자 public, protected, package로 선언하면 스크립트 밖에서 사용할 때 완전한 이름으로 참조된다. 완전한 이름은 패키지 이름, 스크립트 이름, 변수 이름을 합친 것이다. 다음은 십자모양의 커서를 만들기 위해 javafx.scene.Cursor에서 사용한 public 스크립트 변수의 완전한 이름이다.

```
javafx.scene.Cursor.CROSSHAIR;
```

인스턴스 변수는 클래스 선언부 안에서 선언되고 객체가 생성될 때 만들어진다. 리스트 3.5에서는 스크립트 변수와 인스턴스 변수의 몇 가지 예를 든다.

리스트 3.5 스크립트 변수와 인스턴스 변수

```
import javafx.scene.Cursor;
import javafx.scene.paint.Color;

// javafx.scene.Cursor에서 스크립트 변수를 import
import javafx.scene.Cursor.CROSSHAIR;

// 바꿀 수 없는 스크립트 변수
def defaultText = "Replace ME"; // 스크립트에서만 접근 가능
```

(계속)

```
// 바꿀 수 있는 스크립트 변수
public var instanceCount: Integer; // Public 접근

public class Title {
    // 바꿀 수 없는 인스턴스 변수
    def defStroke = Color.NAVY; // 클래스에서만 접근
            // 완전한 이름은 javafx.scene.paint.Color.NAVY

    // 바꿀 수 있는 인스턴스 변수

    // 디폴트 값은 빈 문자열인 ""
    public var text:String;
    public var width: Number; // 디폴트 값은 0.0
    public var height = 100; // 정수 타입으로 예측
    public var stroke: Color = defaultStroke;
    public var strokeWidth = 1.0; // 숫자 타입으로 예측
    public var cursor = CROSSHAIR;
            // 완전한 이름은 javafx.scene.Cursor.CROSSHAIR

...
}
```

선언 중 어떤 것들은 타입을 적고 어떤 것들은 안 적고 있음을 알 수 있다. 타입을 선언하지 않으면 처음 할당한 값에서 타입을 추론한다. String, Number, Integer 그리고 Boolean은 빌트-인 타입이고 나머지는 JavaFx나 Java 클래스이다. (Duration과 KeyFrame 클래스 인스턴스를 쉽게 선언하는 특수 구문이 있는데 이것은 7장에서 다룬다.)

표 3.1은 변수를 위한 접근 권한자의 목록과 각각의 의미 및 한계를 보여준다. 초기화할 때 참조할 수 있으며, 객체 상수 선언 시에도 참조할 수 있다. 또한, 바인딩되는 변수를 알 수 있다. 이것은 JavaFX의 중요한 특징이고 4장에서 자세히 다룰 것이다.

또한 변수에 변환 트리거를 선언할 수 있다. 변환 트리거는 변수의 값이 변할 때마다 호출되는 JavaFX 스크립트 블록이다. 변환 트리거는 on replace 구문을 통해 선언한다.

```
public var x:Number = 100 on replace {
    println("New value is {x}");
};
public var width: Number on replace (old) {
```

```
        println("Old value is {old}, New value is {x}");
    }
```

변환 트리거는 4장에서 더 자세히 다룬다.

표 3.1 • 접근 권한자

접근 권한자	의미
var	**디폴트** 접근 허가는 스크립트 접근이고, 접근 권한자 없이 변수는 초기화될 수 있다. 오버라이드, 읽기, 할당, 바인딩은 스크립트 안에서만 가능하다.
def	**디폴트** 접근 허가는 스크립트 접근이다. 스크립트 안에서만 정의를 읽거나 바인딩할 수 있다.
public var	누구나 읽고 쓸 수 있다. 또한 어디서든 초기화, 오버라이드, 읽기, 할당, 바인딩이 가능하다.
public def	이 정의는 어디서든 읽을 수 있다. 접근 권한에 관계 없이 할당, 초기화(객체 상수에서), 오버라이드가 불가능하다. 어디서든 바인딩이 가능하다.
public-read var	누구나 읽을 수 있지만 스크립트 안에서만 쓸 수 있다.
public-init var	객체 상수에서 초기화될 수 있지만, 자신의 스크립트에서만 업데이트될 수 있다. 오직 인스턴스 변수만 가능하다.
package var	변수는 패키지에서 접근 가능하다. 이 변수는 같은 패키지 안의 클래스에서 초기화, 오버라이드, 읽기, 할당, 바인딩이 가능하다.
package def	같은 패키지 안의 클래스에서 읽거나 바인딩할 수 있다.
protected var	패키지나 서브클래스에서 접근 가능하다. 같은 패키지의 클래스나 서브클래스에서 초기화, 오버라이드, 읽기, 할당, 바인딩 가능하다.
protected def	같은 패키지의 클래스나 서브클래스에서 읽을 수 있고 바인딩 가능하다.
public-read protected var	누구나 읽고 바인딩할 수 있지만 이 변수는 같은 패키지의 클래스나 서브 클래스에서만 초기화, 오버라이드, 할당이 가능하다.
public-init protected var	객체 상수에서 초기화할 수 있고, 누구나 읽고 바인딩할 수 있다. 그러나 같은 패키지 안의 클래스나 서브클래스에서만 오버라이드나 할당이 가능하다. 오직 인스턴스 변수만이 가능하다.

시퀀스

시퀀스는 객체를 나열한 리스트이다. 프로그래밍에서 이런 리스트가 자주 사용되기 때문에 JavaFX는 중요한 클래스 특징으로 시퀀스를 지원한다. 시퀀스를 선언하고, 아이템을

추가하고, 아이템을 삭제하고, 아이템을 수정하는 동작들을 언어 자체에서 지원한다. 또한 시퀀스에서 아이템을 가져오는 기능을 강력히 지원한다.

시퀀스 선언

시퀀스를 선언할 때는 대괄호 안에 각 요소를 쉼표로 구분해서 넣는다.

```
public def monthNames = ["January", "February", "March",
                         "April", "May", "June",
                         "July", "August", "September",
                         "October", "November", "December"];
```

괄호 안의 요소가 String이므로 이것은 String 시퀀스이다. 다음처럼 선언할 수도 있다.

```
public def monthNames: String[] = [ "January", .....];
```

대괄호만 사용하여 빈 문자열을 할당할 수도 있다. 이것은 또한 시퀀스의 디폴트 값이기도 하다. 예를 들어, 다음의 두 선언문은 동일하게 빈 문자열을 선언한다.

```
public var nodes:Node[] = [];

public var nodes:Node[];
```

시퀀스의 값이 바뀔 때 그 변화에 대한 것을 처리하는 트리거 함수를 할당할 수 있다. 이에 대해서는 다음 장에서 다룬다.

Integer와 Number의 시퀀스를 간단하게 선언하기 위해 시작과 마지막 수를 지정하여 범위를 정한다. [1..9]는 1에서 9까지를 나타내는 정수의 시퀀스이고, [1.. < 9]는 9를 포함하지 않는 1부터 8까지의 정수를 나타낸다. step 함수도 사용할 수 있다. 예를 들어, 짝수만 사용하고 싶다면 [2..100 step 2]가 된다. 수의 시퀀스는 십진수로 소수를 적는다. [0.1..1.0 step 0.1]과 같다. step을 정하지 않는 경우 1이나 1.0이 사용된다.

범위는 또한 갈수록 감소하게 정할 수도 있다. 이렇게 하기 위해선 첫 번째 수가 반드시 두 번째 수보다 커야 한다. 그러나 음수의 step 함수가 없으면 빈 시퀀스를 만든다. 이것은 step의 디폴트 값이 항상 양수 1이기 때문이다.

```
var negativeNumbers = [0..-10]; // 빈 시퀀스
var negativeNumbers = [0..-10 step -1]; // 0,-1,-2,...-10
var negativeNumbers = [0..<-10 step -1]; // 0,-1,-2,...,-9
```

다른 시퀀스의 요소를 포함하는 시퀀스를 만들 때는 괄호 안에 소스가 되는 시퀀스를 포함하기만 하면 된다.

```
var negativePlusEven = [ negativeNumbers, evenNumbers ];
```

또한 다른 시퀀스에 Boolean 연산자를 사용하여 새로운 시퀀스를 만들 수 있다. 다른 시퀀스를 소스로 사용하여 Boolean 연산자를 각 요소에 적용한 다음 조건을 만족하는 것만을 모아 새 시퀀스를 만든다. 다음의 예제는 양수 중 2의 배수만을 모아서 새로운 시퀀스를 만들어낸다.

```
var evenIntegers = positiveIntegers[n | n mod 2 == 0];
```

for 루프를 통해서도 시퀀스를 만들 수 있다. for 루프의 반복 중 '반환' 하는 각각의 객체를 시퀀스에 추가한다.

```
// Text 시퀀스를 생성한다.
var lineNumbers:Text[] = for(n in [1..100]) {
      Text { content: "{n}" };
};

// indexof 연산자를 사용하여 정수 시퀀스를 만들어낸다.
var indexNumbers = for(n in nodes) {
      indexof n;
};
```

현재 시퀀스의 크기를 구할 때는 sizeof 연산자를 사용한다.

```
var numEvenNumbers = sizeof evenNumbers;
```

시퀀스 요소 접근

개별적으로 요소에 접근하려면 괄호 안에 숫자 인덱스를 사용하면 된다.

```
var firstMonth = monthNames[0];
```

또한 범위를 지정하여 시퀀스의 일부분을 취할 수 있다. 다음의 두 시퀀스는 동일한 의미이다.

```
var firstQuarter = monthNames[0..2];
```

```
var firstQuarter = monthNames[0..<3];
```

다음의 두 시퀀스도 동일하다. 두 번째 예에서는 시작값만으로 범위를 지정했고, 이 경우에는 인덱스부터 다음의 모든 요소를 반환한다.

```
var fourthQuarter = monthNames[9..11 ];

var fourthQuarter = monthNames[9.. ];
```

시퀀스를 통해 반복작업을 하려면 for 루프를 사용한다.

```
for( month in monthNames) {
    println("{month}");
}
```

시퀀스 수정

시퀀스에서 한 요소를 수정하는 방법은 해당 인덱스에 새로운 값을 할당하는 것이다.

```
var students = [ "joe", "sally", "jim"];
students[0] = "vijay";
```

개발자 노트 DEVELOPER NOTE

이 장의 앞에서 보았듯이 JavaFX는 허용적인 언어다. 그래서 현재 시퀀스 범위 밖의 인덱스에 요소를 할당하면 할당은 무시된다.

이전 예제의 학생 시퀀스를 사용해보자.

```
students[3] = "john";
```

학생 시퀀스의 현재 크기는 3이고 유효한 가장 큰 인덱스는 2이기 때문에 인덱스 3의 위치에 값을 할당한 것은 수행되지 않는다. 이와 같은 이유로 인덱스 −1 위치에 값을 할당한 것도 시퀀스의 범위에서 벗어나므로 수행되지 않는다.

게다가 존재하는 시퀀스의 범위에서 벗어난 인덱스에 접근하면 디폴트 값이 반환된다. Number의 경우에는 0이, String의 경우에는 빈 문자열이, Object의 경우에는 null이 반환된다.

시퀀스에 요소를 삽입하는 방법은 insert 문을 사용하는 것이다.

```
// "vijay" 를 students 시퀀스 마지막에 추가한다.
```

```
insert "vijay" into students;

// "mike" 를 students 시퀀스 제일 앞에 삽입한다.
insert "mike" before students[0];

// "george" 를 students 시퀀스 두 번째에 삽입한다.
insert "george" after students[1];
```

요소를 삭제하는 방법은 delete 문을 사용하는 것이다.

```
delete students[0]; // student의 첫 요소를 삭제한다.
delete students[0..1]; // students의 두 번째 요소를 삭제한다.
delete students[0..<2]; // students의 제일 앞에 있는 두 요소를 삭제한다.
delete students[1..]; // student의 두 번째부터 모든 요소를 삭제한다.
delete "vijay" from students;
delete students; // students의 모든 요소를 삭제한다.
```

네이티브 배열

네이티브 배열native array은 Java 배열을 생성하는 기능이다. 이 기능은 주로 JavaFX와 Java의 배열들을 앞뒤로 옮길 때 사용한다. 다음 코드는 Java int[] 배열을 생성하는 예제이다.

```
var ints: nativearray of Integer = [1,2,3] as nativearray of Integer;
```

네이티브 배열은 시퀀스와 비슷하게 보이지만 시퀀스와 같지는 않다. insert, delete, slice 같은 시퀀스 연산자를 사용할 수 없지만, 다음의 코드처럼 배열의 요소 값을 할당할 수 있다.

```
ints[2] = 4;
```

그러나 현재 배열의 범위 밖에 값을 할당하면 ArrayIndexOutOfBounds 예외처리가 발생한다.

또한 네이티브 배열에서 요소들만큼 반복하기 위해 for 연산자를 사용할 수 있다.

```
for(i in ints) {
    println(i);
}
for(i in ints where i mod 2 == 0) {
    println(i);
}
```

함수

함수는 행동을 정의한다. 함수는 입력 값과 함수 인수를 처리하는 부분을 캡슐화하고 표현식으로 결과를 반환한다. 변수처럼 함수도 스크립트 함수이거나 인스턴스 함수이다. 스크립트 함수는 스크립트 레벨에서 작동하고, 스크립트 레벨에서 정의한 함수나 변수에 접근한다. 인스턴스 함수는 객체의 동작을 정의하고, 함수를 선언한 클래스 안에 포함된 다른 인스턴스 함수와 인스턴스 변수에 접근한다. 게다가 인스턴스 함수는 함수가 담긴 스크립트 파일 안에 포함된 어떤 스크립트 레벨의 함수와 변수에도 접근이 가능하다.

함수를 선언할 때 function 키워드와 함수명 앞에 public, protected, package 접근 권한자를 사용할 수 있다. 접근 권한자를 지정하지 않으면 그 함수는 스크립트 파일 안에서만 사용할 수 있다. 모든 함수 인수는 괄호 안에 적는다. 함수의 반환 결과값의 타입도 지정한다. 반환값의 타입을 생략하면 함수 블록의 마지막 표현식에서 추론한다. Void라는 특수한 반환타입은 함수가 아무것도 반환하지 않는다는 것을 의미한다.

다음 예제의 두 함수 선언은 동일하다. 첫 번째 함수는 함수 블록에서 마지막 표현식이 Glow 객체를 위한 객체 상수이기 때문에 반환 타입을 Glow로 추론한다. 두 번째 함수는 명백하게 Glow라고 반환 타입을 선언하고 return 키워드를 사용하였다

```
public function glow(level: Number) {
    // 반환 타입을 Glow로 추론
    Glow { level: level };
}

public function glow(): Glow { // 반환 타입 명시
    return glow(3.0); // return 키워드 명시
}
```

return 키워드는 함수 블록의 마지막 표현식으로 쓸 수 있다. 그러나 if/else나 루프에서 즉시 값을 반환하고 싶다면 명시적으로 return을 사용해야 한다.

JavaFX에서 함수는 함수 안에서 객체이고 변수에 할당할 수 있다. 예를 들어, 함수 변수를 선언하기 위해 변수에 함수를 할당하고 변수를 통해 함수를 불러온다.

```
var glowFunction : function(level:Number):Glow;
glowFunction = glow;
glowFunction(1.0);
```

함수 정의는 자세히 기술하지 않을 수 있다. 예를 들어 함수 변수에,

```
var glowFunction:function(level:Number): Glow =
    function(level:Number) {
        Glow { level: level };
};
```

또는 객체 상수 선언 안에서,

```
TextBox {
    columns: 20
    action: function() {
        println("TextBox action");
    }
}
```

수퍼클래스로부터 함수를 오버라이드할 때는 override 키워드를 사용한다.

```
class MyClass {
      public function print() { println("MyClass"); }
}
class MySubClass extends MyClass {
      override function print() { println("MySubClass"); }
}
```

스트링

스트링 상수

스트링 상수는 큰따옴표나 작은따옴표를 사용하여 기술한다. 이렇게 두 가지를 사용하는 이유는 스트링 상수에서 특수 문자를 피하기 위해서이다. 예를 들어, 스트링 상수가 큰따옴표를 포함하는 경우이다. 스트링을 작은따옴표로 묶음으로써 큰따옴표의 기능을 배제할 수 있다. 다음의 두 예제를 보자.

```
var quote = "Winston Churchill said:
        \"Never in the field of human conflict was
        so much owed by so many to so few.\" "
var quote = 'Winston Churchill said:
```

```
"Never in the field of human conflict was
so much owed by so many to so few."'
```

중괄호를 사용하여 스트링 상수 안에 표현식을 포함시킬 수 있다.

```
var name = "Jim";
// 출력 결과 My name is Jim
println ( "My name is {name}" );
```

포함된 표현식은 JavaFX나 Java 표현식이고 객체를 반환한다. 이 객체는 toString() 메소드를 사용하여 스트링으로 변환될 것이다

```
println ( "Today is {java.util.Date{}}" );

var state = "The state is {if(running) "Running" else "Stopped"}";

println(" The state is {getStateStr()}" );

println("The state is {if(checkRunning()) "Running" else "Stopped"}");
```

또한, 문자열 상수는 여러 줄에 걸쳐 기술할 수 있다.

```
var quote = "Winston Churchill said: "
"\"Never in the field of human conflict was so much owed "
"by so many to so few.\"";
```

이 예제에서 스트링들은 하나의 스트링으로 합쳐진다. 따옴표 안에서는 스트링 상수만이 사용되며 따옴표 밖의 공백은 무시한다.

유니코드 문자는 \u + 숫자 4개의 유니코드를 사용하여 스트링 상수 안에 입력할 수 있다.

```
var thanks = "dank\u00eb"; // dankë
```

포매팅

스트링 상수에 포함된 표현식은 표시 방법을 기술한 포매팅formatting 코드를 포함할 수 있다.

```
var totalCountMessage = "The total count is {total}";
```

total이 정수integer라면 결과 스트링은 10진수로 보일 것이고, total이 수Number라면 결과 스트링은 해당 지역에 맞는 포맷으로 수를 보여줄 것이다.

```
var total = 1000.0;
```

이 결과는 다음과 같다.

```
The total count is 1000.0
```

표현식의 포맷을 정하려면 표현식 안에 포맷 코드를 넣어야 한다. % 다음에 포맷 코드를 적는다. 포맷 코드는 java.util.Formatter 클래스에 정의되어 있다. 더 자세히 알아보고 싶다면 JavaDoc 페이지를 참조하면 된다(http://java.sun.com/javase/6/docs/api/index.html).

```
println("Total is {%f total}");     // Total is 1000.000000
println("Total is {%.2f total}");   // Total is 1000.00
println("Total is {%5.0f total}");  // Total is 1000
println("Total is {%+5.0f total}"); // Total is +1000
println("Total is {%,5.0f total}"); // Total is 1,000
```

개발자 노트 DEVELOPER NOTE

스트링에 % 문자열을 포함하려면 %를 덧붙여서 기능을 배제해야 한다(%%).

```
println("%%{percentage}"); // prints %25
```

국제화

스트링을 국제화internation하는 방법은 스트링 선언 안에 "번역 키Translate Key" 구문을 사용하는 것이다. 번역 키를 생성할 때는 스트링에 값을 넣을 때 ##(샵샵)으로 시작하게 하여 지역에 맞는 언어로 번역되도록 한다. ##은 큰따옴표나 작은따옴표로 시작하기 전에 적는다. 때로는 키가 [] 안에서 작성될 수도 있다. 만약 키를 기술하지 않으면 스트링 자체를 키로 삼는다.

```
var postalCode = ## "Zip Code: ";
var postalCode = ##[postal]"Zip Code: ";
```

앞의 예에서 첫 번째 형태에서의 키는 "zip Code: "이고, 두 번째 형태에서의 키는 "postal"이다

기본적으로 로컬라이저localizer는 각각의 스크립트 이름에 맞는 속성을 찾는다. 패키지 이름에 스크립트 파일 이름을 더하고, 지역과 .fxproperties 파일 타입을 덧붙인다. 스크립트 이

름이 com.mycompany.MyClass라면 로컬라이저 코드는 com/mycompany/MyClass_xx.fxproperties를 클래스경로에서 찾을 것이다. xx는 지역정보이다. 예를 들어, 영국United Kingdom의 영어English라면 속성 파일 이름은 com/mycompany/MyClass_en_GB.fxproperties가 된다. 프랑스어를 쓰는 캐나다인이라면 com/mycompany/MyClass_fr_CA.fxproperties가 된다. 디폴트 언어가 영어라면 속성 파일은 MyClass_en.fxproperties가 된다. 더 자세히 기술한 파일을 우선시하고 적게 기술한 파일은 다음에 찾는다. 예를 들면, 키를 찾을 때 MyClass_en_GB.fxproperties를 먼저 찾고, 찾을 수 없으면 MyClass_en.xproperties를 찾는다. 키를 전혀 찾을 수 없으면 디폴트로 문자열 자체를 키로 삼는다. 다음의 예들을 보자.

예제 #1 *EXAMPLE*

```
println(##"Thank you");
```

French – MyClass_fr.fxproperties:

```
"Thank you" = "Merci"
```

German – MyClass_de.fxproperties:

```
"Thank you" = "Danke"
```

Japanese – MyClass_ja.fxproperties:

```
"Thank you" = "Arigato"
```

예제 #2 *EXAMPLE*

```
println(##[ThankKey] "Thank you");
```

French – MyClass_fr.fxproperties:

```
"ThankKey" = "Merci"
```

German – MyClass_de.fxproperties:

```
"ThankKey" = "Danke"
```

Japanese – MyClass_ja.fxproperties:

```
"ThankKey" = "Arigato"
```

문자열 안에 표현식이 있을 경우 상수 키는 표현식의 위치에 %s를 포함한다.

```
println(##"Hello, my name is {firstname}");
```

이 경우에 키는 "Hello, my name is %s"가 된다. 마찬가지로 하나 이상의 표현식을 사용한다면 키는 각각의 표현식 위치에 %s를 갖는다.

```
println(##"Hello, my name is {firstname} {lastname}");
```

여기서 키는 "Hello, my name is %s %s"가 된다.

이 파라미터 대체는 번역 문자열에도 사용된다.

French – MyClass_fr.fxproperties:

```
"Hello, my name is %s %s" = "Bonjour, je m'appelle %s %s"
```

또 다른 속성 파일을 스크립트에 적용할 수 있다. javafx.util.StringLocalizer를 사용한다.

```
StringLocalizer.associate("com.mycompany.resources.MyResources", "com.mycompany");
```

com.mycompany 패키지의 스크립트 번역 색인은 디폴트인 스크립트 이름을 쓰는 대신에 com/mycompany/resources/MyResources_xx.fxproperties 속성 파일을 찾는다. 아까처럼 xx는 지역 정보를 나타내는 코드이다.

표현식과 연산자

블록 표현식

블록 표현식block expression은 변수 선언 등의 구문과 다른 표현식들을 중괄호 안에 가지고 있는 것이다. 마지막 구문이 표현식이라면 블록 표현식의 값은 마지막 표현식의 값이 되고, 그렇지 않다면 블록 표현식은 값을 갖지 않는다. 리스트 3.6에는 두 개의 블록 표현식이 있다. 첫 표현식은 subtotal 값을 가진다. 두 번째 블록 표현식은 마지막 표현식이 Void로 선언된 println() 함수이므로 어떤 값도 가지지 않는다.

리스트 3.6 블록 표현식

```
// 값이 있는 블록 표현식
var total = {
    var subtotal = 0;
```

(계속)

```
    var ndx = 0;
    while(ndx < 100) {
        subtotal += ndx;
        ndx++;
    };
    subtotal; // 마지막 표현식
};

// 값이 없는 블록 표현식
{
    var total = 0;
    var ndx = 0;
    while(ndx < 100) {
        total += ndx;
        ndx++;
    };
    println("Total is {total}");
}
```

예외 처리

throw 문은 Java와 같고, java.lang.Throwable로 확장하는 클래스만을 던진다.

try/catch/finally 표현식은 Java와 같지만 JavaFX 문법을 사용한다.

```
try {
} catch (e:SomeException) {
} finally {
}
```

연산자

표 3.2는 JavaFX에서 사용하는 연산자operator의 목록이다. 위의 행부터 순서대로 높은 우선순위를 가진다. 같은 우선순위를 가진 연산자는 동등하게 계산된다. 할당연산자(=)는 오른쪽에서 왼쪽으로 계산하고 다른 모든 연산자들은 왼쪽에서 오른쪽으로 계산한다. 괄호는 기본적인 계산 순서를 바꾸는 데 사용한다.

표 3.2 • 연산자

우선순위	연산자	내용
1	++/--(뒤에 붙는 경우)	할당 후에 증가/감소 적용
2	++/--(앞에 붙는 경우)	할당 전에 증가/감소 적용
	-	마이너스 부호
	not	논리 연산자 complement(Boolean 값을 반대로)
	sizeof	시퀀스의 크기
	reverse	시퀀스의 순서를 반대로
	indexof	시퀀스 요소의 인덱스
3	/, *, mod	산술 연산자
4	+, -	산수 연산자
5	==, !=	비교 연산자(Java의 isEquals()와 비슷)
	<, <=, >, >=	숫자 비교 연산자
6	instanceof, as	타입 연산자
7	and	논리 연산자 AND
8	or	논리 연산자 OR
9	+=, -=, *=, /=	복합 할당
10	=>, tween	애니메이션 보간 연산자
11	=	할당 연산자

조건 표현식

if/else

if는 다른 언어에서 정의하는 if와 비슷하다. 조건을 평가하고 참이라면 표현식 블록을 계산한다. 참이 아닌 경우에 else 표현식 블록이 있다면 그 표현식 블록을 계산한다.

```
if (date == today) {
      println("Date is today");
}else {
      println("Out of date!!!");
}
```

If/else의 중요한 특징 하나는 각 표현식 블록이 계산한 결과를 변수로 할당할 수 있다는 것이다.

```
var outOfDateMessage = if(date==today) "Date is today"
                                       else "Out of Date";
```

또한 표현식 블록은 단순한 표현식보다 복잡해질 수 있다. 리스트 3.7은 if/else 문을 사용하여 outOfDateMessage에 값을 할당하는 복잡한 방식을 보여준다.

리스트 3.7 if/else 표현식을 사용한 복잡한 할당

```
var outOfDateMessage = if(date==today) {
        var total = 0;
        for(item in items) {
            total += items.price;
        }
        totalPrice += total;
        "Date is today";
    } else {
        errorFlag = true;
        "Out of Date";
    };
```

앞의 예제에서 블록의 마지막 표현식인 문자열 상수 에러 메시지는 변수로 할당되는 객체이다. 이것은 수를 포함하여 어떤 JavaFX 객체이건 가능하다.

왜냐하면 if/else는 표현식 블록이라서 다른 if/else 문과 쓰일 수 있기 때문이다.

```
var taxBracket = if(income < 8025.0) 0.10
        else if (income < 32550.0) 0.15
        else if (income < 78850.0) 0.25
        else if (income < 164550.0) 0.28
        else 0.33;
```

루프 표현식

for

for 문은 시퀀스를 사용하고, 시퀀스의 멤버들에 대해 반복할 수 있다.

```
var daysOfWeek : String[] = [ "Sunday", "Monday", "Tuesday" ];
for(day in daysOfWeek) {
      println("{indexof day}). {day}");
}
```

수를 검사하며 반복하는 전형적인 for 문처럼 대괄호 안에 시퀀스 범위를 지정하는 수를 사용할 수도 있다.

```
for( i in [0..100]} {
```

for 표현식은 새로운 시퀀스를 반환할 수도 있다. 각각의 반복작업 중 표현식 블록 계산 결과가 객체라면 그 객체는 for 표현식에서 반환할 새로운 시퀀스에 삽입된다. 예를 들어, 다음의 for 표현식 예제에서 한 주의 요일마다 반복하며 새 Text 노드가 생성된다. for 표현식은 최종적으로 한 주의 각 요일에 맞는 Text 그래픽 요소를 포함하는 새로운 시퀀스를 반환한다.

```
var textNodes: Text[] = for( day in daysOfWeek) {
    Text {content: day };
}
```

For 표현식의 다른 특징은 다른 루프를 포함할 수 있다는 것이다. 리스트 3.8은 중첩 루프를 사용한 예제를 보여준다.

리스트 3.8 중첩 루프

```
class Course {
    var title: String;
    var students: String[];
}
var courses = [
    Course {
        title: "Geometry I"
        students: [ "Clarke, "Connors", "Bruno" ]
    },
    Course {
        title: "Geometry II"
        students: [ "Clarke, "Connors", ]
```

(계속)

```
    },
    Course {
        title: "Algebra I"
        students: [ "Connors", "Bruno" ]
    },
];
for(course in courses, student in course.students) {
   println("Student: {student} is in course {course}");
}
```

이 결과는 다음과 같다.

```
Student: Clarke is in course Geometry I
Student: Connors is in course Geometry I
Student: Bruno is in course Geometry I
Student: Clarke is in course Geometry II
Student: Connors is in course Geometry II
Student: Connors is in course Algebra I
Student: Bruno is in course Algebra I
```

추가 루프는 이전의 루프와 쉼표로 구별한다. 그리고 이전 루프의 어떤 요소라도 참조할 수 있다.

또한 where는 where 절에서 참이라고 평가하는 요소들만 시퀀스에서 나오도록 할 수 있다.

```
var evenNumbers = for( i in [0..1000] where i mod 2 == 0 ) i;
```

while

while 루프는 다른 언어에서 정의하는 while과 비슷하다.

```
var ndx = 0;
while ( ndx < 100) {
    println("{ndx}");
    ndx++;
}
```

JavaFX의 for 루프와 달리 while 루프는 어떤 표현식도 반환하지 않는다. 그래서 시퀀스를 생성하는 데에 사용할 수 없다는 것을 기억해야 한다.

break/continue

break와 continue는 루프 반복을 제어한다. break는 루프를 완전히 종료할 때 사용한다. 이것은 해당 지점에서부터 루프 전부를 중지시킨다. 반면에 continue는 단순히 현재 반복을 중지하고, 해당 루프는 다음 반복부터 다시 한다. 리스트 3.9를 보자.

리스트 3.9 break/continue

```
for(student in students) {
    if(student.name == "Jim") {
        foundStudent = student;
        break; // 루프 모두 중지
               // 더는 student를 검사하지 않는다.
    }
}
for(book in Books ) {
        if(book.publisher == "Addison Wesley") {
           insert book into bookList;
           continue; // 다음 book을 검사하러 간다.
        }
        insert book into otherBookList;
        otherPrice += book.price;
}
```

타입 연산자

instanceof 연산자는 객체의 클래스 타입을 검사한다. 반면에 as 연산자는 객체를 다른 클래스로 캐스팅한다. as는 몇몇 특수한 클래스의 함수를 사용하기 위해서 보통의 클래스를 특수 클래스로 캐스팅할 때에 유용하다. 물론 객체는 그 클래스에서 상속받았어야 하고, 이를 검사할 때 instanceof가 유용하다. 그렇지 않은 객체를 캐스팅하려 하면 예외가 발생할 것이다.

다음의 리스트에서 printLower() 함수는 문자열을 소문자로 바꾼다. 그러나 다른 타입의 객체는 그대로 출력한다. 먼저 객체가 문자열인지를 테스트하고 맞다면 객체는 as 연산자를 사용하여 문자열로 캐스팅한다. 그 다음 문자열의 toLowerCase() 메소드가 결과를 소문자로 변환한다. 리스트 3.10은 instanceof와 as 연산자를 사용한 예이다.

리스트 3.10 타입 연산자

```
function printLower(object: Object ) {
    if(object instanceof String) {
        var str = object as String;
        println(str.toLowerCase());
    }else {
        println(object);
    }

}
printLower("Rich Internet Application");
printLower(3.14);
```

커맨드라인 인수 접근

익스포트된 클래스, 변수, 함수를 선언하지 않는 순수한 스크립트의 커맨드라인 인수command-line arguments는 javafx.lang.FX.getArguments():String[] 함수를 사용하여 얻을 수 있다. 이 함수는 스크립트가 시작할 때 커맨드라인에서 스크립트에 넘긴 인수를 가지고 있는 문자열 시퀀스를 반환한다. 이 함수의 다른 버전인 javafx.lang.FX.getArguments(key:String):String[] 함수는 애플릿처럼 다른 방식으로 사용할 때에 이름-값의 쌍으로 인수를 전달한다. 비슷한 방식으로 시스템 속성을 얻는 javafx.lang.FX.getProperty(key:String):String[] 함수도 있다.

만약 스크립트가 익스포트된 클래스, 변수, 함수를 가지고 있다면 스크립트 레벨에서 정의한 run 함수를 통해서 인수를 얻는다.

```
public function run(args:String[] ) {
    for(arg in args) {
        println("{arg}");
    }
}
```

개발자 노트 DEVELOPER NOTE

익스포트된 멤버를 사용한 loose 표현식

스크립트 레벨의(클래스 선언 내부가 아닌) 변수, 함수, 표현식을 **loose 표현식**이라고 한다. 이 변수와 함수가 스크립트에서 private일 때, 커맨드라인에서 스크립트를 실행한다면 특정

run 함수는 필요하지 않다. 그러나 이 표현식 중 어느 하나라도 public, public-read, protected, package 접근 권한으로 스크립트의 외부에 익스포트되면, 스크립트를 직접 실행할 때 run 함수가 필요하다. 이 run 메소드는 익스포트된 변수와 함수를 캡슐화한다.

빌트인 함수와 변수

모든 JavaFX 스크립트에서 선언 및 정의 없이 사용 가능한 함수들이 있다. 이 함수들은 javafx.lang.Builtins에 정의되어 있다.

이 중 하나인 println()을 이미 보았다. println()은 객체 인수를 받아서 한 번에 한 줄씩 콘솔에 출력한다. 이것은 Java 메소드인 System.out.println()과 비슷하다. 이것과 비슷한 함수인 print()는 새로운 줄 없이 인수를 출력한다. 인수의 toString() 메소드는 문자열을 출력하기 위해 호출된다.

```
println("This is printed on a single line");
print("This is printed without a new line");
```

javafx.lang.Builtins의 또 다른 함수 중 isInitialized()가 있다. 이 메소드는 JavaFX 객체를 받아서 그 객체가 완전히 초기화되었는지를 알려준다. 변수 트리거에서 초기화 중인 객체의 현재 상태를 정할 때 유용하다. 객체를 초기화하도록 전달한 후 기능을 실행하기 전에 약간의 시간이 필요하다. 예를 들어, 리스트 3.11에서 isInitialized()가 replace 트리거에서 쓰이는 것을 보자.

리스트 3.11 isInitialized()

```
public class Test {
    public var status: Number on replace {
        // status에 값이 할당될 때까지 초기화되지 않는다.
        if(isInitialized(status)) {

            commenceTest(status);
        }
    }
    public function commenceTest(status:Number) : Void {
        println("commenceTest status = {status}:);
    }
}
```

이 예제에서는 Test 클래스가 처음 초기화될 때 인스턴스 변수 status는 먼저 디폴트 값인 0.0을 가진다. 그리고 그 다음에 on replace 표현식 블록이 계산된다. 그러나 이는 status를 uninitial-ized 상태로 둔다. 값이 status에 대입이 될 때만 상태는 initialized로 변경될 것이다. 다음을 보자.

```
var test = Test{}; // status는 uninitialized
test.status = 1;   // 이제 status는 initialized가 된다.
```

이 경우 Test가 Test{} 객체 상수를 사용하여 만들어지면 status는 디폴트 값인 0.0을 가진다. 그러나 initialized가 아니므로 commenceTest는 객체를 생성하는 동안 호출되지 않을 것이다. 이제 status에 값을 대입하면 상태는 initialized로 변경되고 commenceTest도 호출된다. status에 디폴트 값을 대입하면 비록 그 값이 0일지라도 status는 즉시 initialized가 된다는 것을 주의하자. 다음의 예제를 보자.

```
public class Test {
    public var status: Number = 0 on replace {
        // 즉시 initialized가 된다.
        if(isInitialized(status)) {
            commenceTest(status);
        }
    }
 }
```

마지막으로 다룰 빌트인 함수는 isSameObject()이다. isSameObject()는 두 인수가 실제로 같은 인스턴스인지를 판단한다. 이 함수는 == 연산자와 다르다. JavaFX에서 == 연산자는 두 개의 객체가 같은 값인지를 보지만 그것이 같은 인스턴스임을 의미하지 않는다. == 연산자는 Java 함수 isEquals()와 비슷하고 JavaFX의 isSameObject는 Java의 == 연산자와 비슷하다. Java를 알고 있다면 조금 헷갈릴 것이다.

빌트인 변수에는 __DIR__와 __FILE__이 있다. __FILE__는 JavaFX 클래스를 리소스 URL 문자열로 가지고 있다. __DIR__은 현재의 클래스를 담고 있는 디렉터리에 해당하는 리소스 URL 문자열을 가지고 있다.

```
println("DIR = {__DIR__}");
println("FILE = {__FILE__}");
// 이미지의 위치 입력
var image = Image { url: "{__DIR__}images/foo.jpeg" };
```

다음의 예는 디렉터리 클래스경로와 JAR 기반 클래스경로를 사용한 출력의 비교이다.

클래스경로에서 Jar 파일을 사용한다.

```
$javafx -cp Misc.jar misc.Test

DIR = jar:file:/export/home/jclarke/Documents/Book/FX/code/Chapter3/Misc/dist/Misc.jar!/misc/

FILE = jar:file:/export/home/jclarke/Documents/
       Book/FX/code/Chapter3/Misc/dist/Misc.jar!/misc/Test.class
```

디렉터리 클래스경로 사용

```
$ javafx -cp . misc.Test

DIR = file:/export/home/jclarke/Documents/Book/FX/code/Chapter3/Misc/dist/tmp/misc/

FILE = file:/export/home/jclarke/Documents/Book/FX/code/Chapter3/Misc/dist/tmp/misc/Test.class
```

개발자 주의사항 DEVELOPER WARNING

__DIR__의 뒤에 오는 슬래시(/)를 주의하라

슬래시는 이미 __DIR__에 포함되어 있으므로 이미지와 같은 리소스에 대한 경로를 만들 때 슬래시를 추가하지 ***않아야 한다***. Image{ url: "{__DIR__}image/foo.jpeg"}은 옳다.

Image{ url: "{__DIR__}/image/foo.jpeg"}은 ***틀리다***. __DIR__ 뒤에 슬래시를 추가하면 이미지를 찾을 수 없고, 왜 안 되는지를 알아내기 위해 머리를 쥐어짜야 할 것이다.

요약

이 장에서는 JavaFX 스크립트 언어의 핵심 개념을 다루었다. 스크립트를 구성하는 요소와 클래스를 구성하는 요소들도 보았다. 스크립트와 인스턴스 변수를 선언하는 방법과 시퀀스를 만들고 수정하는 방법, 그리고 논리 흐름을 제어하는 방법도 보았다.

이제 JavaFX 스크립트 언어의 문법과 연산자의 기본을 이해하게 되었다. 이제 이것을 사용할 차례이다. 뒤의 장들에서는 JavaFX의 핵심 기능들을 반복하여 연습하고 그 기능들을 이용하기 위해 JavaFX 스크립트 언어를 사용하는 방법을 다룰 것이다. 다음 장에서는 JavaFX 런타임에서의 데이터 동기화 지원을 알아보자.