Основные характеристики функциональных языков программирования. Использование понятий функционального программирования (замыкания, анонимные функции) в современных объектно-ориентированных языках

Функциональные ЯП:

-динамические — то есть все основные связывания происходят во время выполнения

-отсутствует понятие состояния — то есть нет понятия присваивания

- главная операция — вызов функции

-главная абстракция — определение функции

-функции являются объектами первого класса — то есть они могут быть значениями, вычисляться, передаваться как параметры и возвращаемые значения других функций и т.п.

-структуры данных — как правило, списки (последовательности)

-простая типовая структура (небольшое количество примитивных типов данных)

Понятие переменной соотвествует математическому смыслу — переменная ОТОЖДЕСТВЛЯЕТСЯ со значением, а не хранит его. Ассоциация переменной и отождествленного с ней значения существует все время, пока существует переменная (время жизни переменной совпадает с временем жизни ассоциации переменной со значением). Нет способа изменить значение (в «чистых» функциональных языках). Если такой способ есть (Лисп — разрушающие присваивания SET и SETQ), то это значит, что язык является мультипарадигмальным и содержит процедурные конструкции вместе с функциональными.

Современные объектно-ориентированные языки, использующие процедурно-объектную парадигму начинают использовать некоторые понятия из ФП. Это относится прежде всего к понятиям замыкания и анонимным функциям (лямбда-функциям).

Замыкание (иногда говорят про лексическое или функциональное замыкание) — это конструкция, которая связывает функцию (функциональное значение) с переменными из объемлющей функцию области видимости. Про такие переменные говорят, что они «захвачены» (поэтому в русскоязычной литературе иногда замыкание переводят как «захват»). Для захваченных переменных область видимости (scope) не совпадает с областью действия (extent) — последняя — шире.

Как правило замыкания возникают, когда некоторая функция F возвращает как свой результат вложенную в нее функцию InF, которая ссылается на локальные переменные функции F (или глобальные переменные).

Пример из языка JavaScript:

```
function initAdder(x) {
     function adder(y) { return x + y } // захвачена переменная x
     return adder
}
var closure1 = initAdder(1)
var a1 = [closure1(10), closure1(20), closure1(5)] //[11,21,6]
var closure2 = initAdder(2)
var a2 = [closure2(10), closure2(20), closure2(5)] //[12,22,7]
```

Анонимная функция (иногда называемая лямбда-функцией по аналогии с соответствующим понятием языка Лисп) — это «чистое» функциональное значение без имени. Его можно передавать как параметр другой функции, возвращать как результат другой функции, а в языках с процедурными конструкциями — и присваивать. Именованные функции в этом смысле можно считать именованными функциональными константами. Использование анонимных функций позволяет сделать программы более компактными и выразительными.

Предыдущий пример на JavaScript с использованием анонимной функции:

```
function initAdder(x) {
     return function(y) { return x + y } // захвачена переменная x
} … // далее — то же самое, что и выше
```

В стандарте языка JavaScript-2015 (ECMAScript 6) для анонимных функций введена еще одна нотация, которую и называют лямбда-функцией (вотличие от анонимных функций, которые все время были в языке JavaScript). С использованием лямбда-нотации пример становится еще короче:

```
function initAdder(x) {
     return y => x + y // захвачена переменная x
} … // далее — то же самое, что и выше
```

Замыкания и анонимные функции в языках C#, Java

В языке C# изначально был введен некоторый аналог функционального тапа данных — делегаты, а начиная с версии 2 — анонимные делегаты, которые можно рассматривать как аналог анонимных функций. Начиная с версии 4 в языке появляются лямбда-выражения и лямбда-операторы (по синтаксису очень похожие на лямбда-нотацию из JavaScript 2015), которые можно рассматривать как «генераторы» анонимных функций.

```
delegate(int x, int y) { return x+y; } // анонимный делегат
(x,y)=> { return x+y; } // лямбда-оператор
```

```
(x,y)=>x+y // лямбда-выражение
```

Замечание: лямбда-выражения являются более общей конструкцией, чем лямбда-операторы, поскольку первые в отличие от вторых могут быть преобразованы в стандартный тип деревьев выражений. Деревья выражений имеют мощный API и, в частности, могут формироваться динамически. Если вы не собираетесь использовать деревья выражений, то существенной разницы между лямбда-операторами и выражениями не найдете.

Обратите внимание на то, что параметры анонимных делегатов (функций) в языке — типизированы (а тип возвращаемого значения выводится из типа выражения в `return` – при наличии нескольких операторов `return` тип должен определяться однозначно и одинаково). А лямбда-конструкции — нетипизированы. Генерация конкретных анонимных функций по лямбда-выражениям и лямбда-операторам происходит при подстановке лямбда-конструкций в выражения, например, выражение с операцией присваивания x=lambda_expression или выражение-фактический параметр какой-либо функции F(lambda_expression) и т. д. Конкретные типы аргументов генерируемой функции компилятор выводит из контекста — для присваивания — это левая часть, для обращения к функции — это тип соотвествующего формального параметра и т. п. Очень удобны для описания таких функциональных типов стандартные делегаты-обобщения (которые надо рассматривать как средство генерации конкретных функциональных типов).

Например

```
Func<int, int, int> add = (x,y)=>x+y;
```

В этом примере add – это делегат-функция с двумя параметрами типа int и возвращаемым типом int. Поэтому компилятор сгенерирует по лямбда-выражению такой же делегат, как и в примере выше.

Пример из JavaScript на C#:

```
Func<int, int> InitAdder(int x) {
     return y=>x+y; // переменная x захвачена
}
void Foo() {
     var closure1 = InitAdder(1);
     int[] a1 = {closure1(10), closure1(20), closure1(5)};
      //{11,21,6}
     var closure2 = InitAdder(2);
     int[] a2 = {closure2(10), closure2(20), closure2(5)};
```

```
        //{12,22,7}
}
```

Язык Java. В этом языке понятие замыкания появилось достаточно давно — еще в прошлом веке — для т. н. локальных внутренних классов. Эти классы объявляются локально внутри тела какого-нибудь метода (напомним, что «просто» функций в Java нет — только методы какого-то класса). Методы этого локального класса могут ссылаться на локальные переменные из объемлющего тела метода, а также на члены объемлющего класса (так как локальный класс является внутренним — о внутренних классах см. книгу Арнольда и Гослинга по языку Java). Замыкание происходит тогда, когда объект локального класса сохраняется при выходе из объемлющего блока, например, когда объект локального класса возвращается как результат метода, в котором он описан (ситуация очень похожая на возвращение функции, как значения). Правда, из соображений эффективной организации многопоточности доступ к захваченным переменным разрешался только по чтению. Менять значения захваченных переменных в Java нельзя (в отличие от таких языков как JavaScript и C#) .

Проблема с лямбда-функциями происходит из того, что в Java не было аналогов функциональных типов (в отличие от языка C#), поэтому нельзя было передавать методы класса как аргументы методов, делать их возвращаемыми значениями методов, присваивать методы переменным и т. д. Авторы языка не могли добавлять в язык абсолютно новые понятия типа функциональных типов и т. п. из соображений бинарной (точнее, байт-кодовой) совместимости со старыми программами (такого рода совместимость всегда выгодно отличала Java от языков платформы .NET).

В 2014 году в версию языка Java 8 были введены типизированные лямбда-выражения, например:

```
(Integer x, Integer y) → x + y
```

или

```
(Integer x, Integer y) → { return x + y; }
```

Разницы между первой и второй формами нет.

Отметим следующие особенности:

- типами параметров лямбда-выражений могут быть ТОЛЬКО объектные типа (если нужно использовать примитивные типы — используйте классы-обертки)

- тип возвращаемого значения выводится из возвращаемых выражений (в случае множественных операторов return тип должен определяться единственным образом, как и в C#).

Как использовать такие выражения? Одним-единственным образом — их можно преобразовывать в функциональные интерфейсы. Функциональный интерфейс — это интерфейс языка Java с единственной функцией. Имя этой функции (как и имя интерфейса)

не имеет значения при преобразовании, и нужно только при вызове. Лямбда-выражение можно преобразовать в функциональный интерфейс, если типы аргументов и возвращаемый тип выражения совпадают с соответствующими типами функции из интерфейса (напомним, что она должна быть только одна по определению). Для удобства в стандартном пакете java.util.function определены специальные обобщенные функциональные интерфейсы (по аналогии со стандартными обобщенными делегатами в .NET), например Function для функций с одним параметром, BiFunction для функций от 2 аргументов и другие. Метод, который нужно вызывать, как правило, называется apply.

Замечание. Совпадение типов в методе функционального интерфейса и в лямбда-выражении — это частный случай соответствия выражения интерфейсу. Общий случай - вариантность и контравариантность - здесь не обсуждается.

Можно считать, что при преобразовании лямбда-выражения в функциональный интерфейс происходит автоматическая генерация класса, реализующего соответствующий интерфейс, а в качестве тела замещающего метода используется код из тела лямбда-выражения.

Предыдущий пример для Java:

```
Function<Integer, Integer> initAdder(int x) {
     return (Integer y) → x + y;
}
void Foo() {
     Function<Integer, Integer> closure1 = initAdder(1);
     int[] a1 = {closure1.apply(10), closure1.apply(20), closure1.apply(5)};
      //{11,21,6}
     Function<Integer, Integer> closure2 = initAdder(2);
     int[] a2 = {closure2.apply(10), closure2.apply(20), closure2.apply(5)};
      //{12,22,7}
}
```

Заметим, что понятие функционального интерфейса и лямбда-выражения фактически ввело в язык понятие «просто» функции. Теперь осталось только сделать так, чтобы методы классов (как статические, так и нестатические) тоже можно было рассматривать как функции. Для этого в язык была введена конструкция «ссылка на метод». Эта конструкция переводится в лямбда-выражение и может быть использована в качестве такового везде, где допустимо лямбда-выражение.

Ссылки бывают трех видов:

- объект::нестатический_метод (obj::bar => (T x)→obj.bar(x))

- класс::нестатический_метод (Foo::bar => (Foo x, T y) => x.bar(y))

- класс::статический_метод (Foo::foobar => (T x)=>Foo.foobar(x))

В примерах предполагается следующий класс:

```
class Foo {
        …
        Y bar(T x) { … }
        static Y foobar(T x) { ….}
        // T, Y – имена классов
}
```