Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

Арифметические типы данных - вещественные

Первые компьютеры - для научных вычислений (ENIAC - только вещ.числа в десятичном представлении)

Первый ЯП - Фортран - INTEGER, REAL, DOUBLE PRESICION

Зачем два вещ. типа? Один (длинный) - для вычислений с повышенной точностью, второй (короткий) - для хранения результатов (ну и для вычислений тоже, если точности хватает...). Актуально и сейчас:

C/C++ - float, **double**, long double

Позже сфера применения компьютеров расширилась, появились области применения, в которых НТ-вычисления не являлись критичными.

Например, микропроцессоры для ПК - Apple II (MC-6902), IBM PC (i8088,i80286,i80386) не было встроенной вещ. арифметики (либо эмуляция, либо сопроцессор для мат. вычислений).

i80486 - 1989год - первый процессор Intel с встроенным сопроцессором (хотя еще был и i80486SX).

Но с точки зрения универсальности наличие вещественных типов в индустриальном ЯП ***абсолютно*** необходимо

Главный вопрос: как именно представлять числа?

С 50-х гг - в плавающем виде : $s * M * B^P$

По сравнению с целыми - значительно больше вариантов представления - основание M и P (обычно - 2, но могут быть и разными), как нормализовывать M (например, 1/B <= M < 1, но могут быть и такие: 1<= M < B), как представлять порядок, допускать ли ненормализованные числа и т.д. - много других тонких вопросов.

Арифметические типы данных - вещественные

Много головной боли для разработчиков ЯП.

Как всегда, наиболее абстрактная схема, обеспечивающая полноту отображения машинных типов данных и независимость от деталей представления, - в языке Ада (1983).

Крайне коротко: программист задает точность вычисление в десятичных знаках, а компилятор сам должен подобрать реализацию в терминах конкретной архитектуры (вплоть до эмуляции)

type MyFloat = digits 8;

type PrecFloat = digits 16;

Проблема - сложность реализации на некоторых архитектурах.

Не вдаемся в детали по двум причинам - мало времени, появление ОБЩЕПРИНЯТОГО стандарта на аппаратные вычисления с плавающей точкой.

Арифметические типы данных - вещественные

IEEE-754 (1985 год) - ***IEEE стандарт для двоичной арифметики с плавающей точкой (ANSI/IEEE Std 754-1985)***

Последняя редакция - 2008 год

3 основных формата - одинарной точности (32), двойной (64), четверной (128).

Очень коротко: представление чисел - двоичное, B=2. Нормализованная мантисса >= 1 и < 2 => старший разряд мантиссы всегда = 1

S E M, где S - бит знака, E - (смещенный) порядок, M - мантисса (без старшего разряда) ($1 > M >= 2^{-m}$) , m - длина мантиссы в битах

$(-1)^S * 2^{E-e} * 1.M$

Смещение порядка $e = 2^{n-1}$, где n - длина порядка в битах.

Крайние значение смещенного порядка : -2^n и $2^n - 1$ используются для хранения "особых" чисел - NaN, ∞, ненормализованных.

Арифметические типы данных - вещественные

NaN - "Not a Number" - возникает при ошибках в вычислениях(например, ∞/ ∞).

Может участвовать в вычислениях (NaN * число = NaN). Не равно никакому числу, в том числе и NaN (это не число, но представление нечислового значения).

∞ - возникает при переполнении (порядок слишком мал для представления абсолютного значения числа) или потере точности (порядок слишком велик для представления числа).

-∞ * -1 == + ∞

∞ * -1 == -∞ и т.д.

Арифметические типы данных - вещественные

$(-1)^S * 2^{E-e} * 1.M$

m - длина мантиссы в битах, n - длина порядка в битах

Значения для 3 основных форматов:

Длина	n	m
32	8	23
64	11	52
128	14	113

Пример для одинарной точности (1 - не хранится) - нумерация битов 0-31 - справа налево

S	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	1	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}	M_{17}	M_{18}	M_{19}	M_{20}	M_{21}	M_{22}	M_{23}

Арифметические типы данных - вещественные

Обладает рядом "хороших" свойств, например, если интерпретировать число как последовательность битов, образующих целое без знака n, то n+1 даст битовое представление следующего по величине вещественного числа, представимого в этом формате (естественно, что машинное множество чисел никак не может быть плотным) (отсюда очень простая операция сравнения).

Также - более корректная операция округления.

Подробнее можно прочитать в посте: https://habr.com/ru/post/112953/

Математический взгляд на приближенную вещественную арифметику:

"What Every Computer Scientist Should Know About Floating-Point Arithmetic"

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Арифметические типы данных - вещественные

Является ли IEEE-754 идеальным? Нет. Но критика, в основном, касается не столько конкретно стандарта, сколько вообще текущей подхода к реализации приближенных вычислений.

Полное следование правилам стандарта приводит к замедлению вычислений, поэтому некоторые реализации ослабляют правила для повышения скорости вычислений.

С точки зрения ЯП просто наличие такого стандарта - огромное благо.

Стандарты C/C++ почти ничего не говорят о требованиях к реализации, но фиксируют номенклатуру (float, double, long double), но на практике современные архитектуры удовлетворяют стандарту IEEE-754, поэтому и реализации выбирают эти форматы.

Арифметические типы данных - вещественные

С 90-х годов прошлого столетия номенклатура вещ. ТД следует этому стандарту, включая размер (32-64-128), а некоторые ЯП прямо декларируют соответствие этому стандарту, например, Java.

- float - 32 бита
- double - 64 бита

C# - аналогично, Go - float32, float64

Арифметические типы данных - вещественные

Интересно, что в версию стандарта IEEE-754 от 2008 года вошел еще один вариант представления - decimal.

"Иногда они возвращаются"

Первые варианты этого типа - двоично-десятичные числа (BCD) - в машинах IBM-14xx (50-е гг), язык COBOL (1959). Далее - система IBM/360-370 и язык PL/I с типом FIXED DECIMAL (M,N).

Значения типа - поле битовых тетрад (полубайтов) длины M, каждый из которых принимает значения от 0 до 9 (десятичные цифры), N последних рассматриваются как дробная часть числа.

"Рудименты" этого типа - в системе команд x86 - команды коррекции для получения корректных результатов для BCD (AAA, AAD, AAM, AAS, DAA, DAS).

То есть это тип данных для вещественных чисел с фиксированной точностью. Чем удобен?

- необходимость вычислений с фиксир. точностью (финансовые расчеты)
- удобство ввода/вывода

Арифметические типы данных - вещественные

Интересно, что в версию стандарта IEEE-754 от 2008 года вошел еще один вариант представления - decimal.

"Иногда они возвращаются"

Первые варианты этого типа - двоично-десятичные числа (BCD) - в машинах IBM-14xx (50-е гг), язык COBOL (1959). Далее - система IBM/360-370 и язык PL/I с типом FIXED DECIMAL (M,N).

Значения типа - поле битовых тетрад (полубайтов) длины M, каждый из которых принимает значения от 0 до 9 (десятичные цифры), N последних рассматриваются как дробная часть числа.

"Рудименты" этого типа - в системе команд x86 - команды коррекции для получения корректных результатов для BCD (AAA, AAD, AAM, AAS, DAA, DAS).

То есть это тип данных для вещественных чисел с фиксированной точностью. Чем удобен?

- необходимость вычислений с фиксир. точностью (финансовые расчеты)
- удобство ввода/вывода

Недостаток - неэффективное кодирование с точки зрения длины. Были разработаны альтернативные кодировки на том же принципе (кодирование десятичных цифр), но более эффективные. В стандарте IEEE-754-2008 - 3 представления (decimal32,decimal64, decimal128) с двумя способами кодирования.

ЯП с типом данных decimal - SQL, VB, C#. В последнем - 128-бит представление

Арифметические типы данных - вещественные

C#: decimal - 128-бит представление

диапазон: $\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$

28 точных десятичных знаков

decimal b = 3.998m; // m - суффикс десятичной константы

Операции - обычные, но нет автоматического преобразования в другие числовые типы (остальные числовые типы в C# - можно с расширениями). Нужно преобразовывать явно.

System.Console.WriteLine(5.3 + (double)b);

Другие ЯП:

C++ на стадии обсуждения (g++ реализует расширение стандарта)

Библиотечная поддержка:

C++ - boost::cpp_dec_float

Python - decimal.Decimal

Java - java.math.BigDecimal (и java.math.BigInteger)

Очень специфичный тип данных => более детально останавливаться не будем.

Арифметические типы данных - вещественные

Набор операций:

стандартный во всех языках -

4 арифметических (+, -, *, /)

Операции сравнения (возвращают логическое значение true или false): < , >, <=,>=, ==, !=

ВНИМАНИЕ - из-за погрешности с округлениями:

- крайне осторожно с равенством (неравенством).
- арифметика - неассоциативна.

Преобразования: безопасны только расширяющие и только в ЯП, где определено представление (C#, Java,...). В противном случае результат не определён (по стандарту).

Логический тип данных

Алгол-60 Boolean в честь Джорджа Буля (G.Bool)

Все ЯП содержат операции сравнения (возвращают лог.тип) и логические операции (комбинировать условия).

Главный вопрос - а есть ли он вообще в ЯП?

Логический тип данных

Алгол-60: Boolean в честь Джорджа Буля (G.Bool)

Все ЯП содержат операции сравнения (возвращают лог.тип) и логические операции (комбинировать условия).

Главный вопрос - а есть ли он вообще в ЯП?

В настоящее время - в каждом есть!

C99 - _Bool, C++ - bool (еще с 90-х гг)

Почему _Bool? - странноватое название?

В норме надо использовать <stdbool.h>

bool a = true, b = false;

Но если есть конфликт со старым кодом, который не хочется (невозможно, тяжело, ...) устранять, то

_Bool a = 1, b = 0;

sizeof (_Bool) == sizeof (int)

Логический тип данных

Поэтому реально главный вопрос - есть ли НЕЯВНОЕ преобразование integer <=> Boolean

C/C++ - есть для совместимости со старым кодом

int i; if (!i) .....

JavaScript - политика такая ("слабая" типизация)

Python - no comments....

Современные ЯП (C#, Go, Java, Swift, Rust,....) - не допускают неявных преобразований нелогических значений к логическим

int i; ..... if (i == 0).....

Чем плохо неявное преобразование? Ненадежность...

if (pageNum == (1 || 11|| 21|| 31)) .....

Логический тип данных

Набор операций: and, or, xor, not. Иногда добавляют equ.

Сравнение

"Ленивость" логических операций

Пример - Паскаль

Что возвращают логические операции, к каким типам операндов применимы?

"Экзотика" - Питон, JavaScript

Поскольку есть неявное преобразование ЛЮБЫХ значений к логическим, то операция not (!) возвращает логическое значение для любых операндов. Семантика = "пустое значение или непустое"

a = -9

print(not a) # False

А как привести любое значение к типу Boolean?

a = ""

print (not not a) # False

Особенно "выразительно" выглядит в JS: !!a

Хотя можно воспользоваться в JS и конструктором Boolean(a)

Для каких числовых значений !!a выдаст false?

Логический тип данных

Интереснее ситуация в Python и JavaScript для двуместных операций and(&&), or (||)

Во-первых, они ленивые (как и везде), во-вторых могут возвращать значение ЛЮБОГО типа в зависимости от типа операндов.

Наблюдение для чисто логических типов этих операций:

and - если значение первого операнда есть true, то возвращает *значение второго операнда*, иначе (если false) *значение первого операнда*

Аналогично:

or - если значение первого операнда есть false, то возвращает *значение второго операнда*, иначе (если true) *значение первого операнда*

Идея для произвольных операндов такая же, только вместо "значение первого операнда есть x" нужно понимать: "значение первого операнда, приведенное к Boolean, есть x". Обратим внимание, что в любом случае для второго операнда нет операции приведения к Boolean - оно просто берется as is.

Прием: использование логической or для указания значения по умолчанию.

```
const default_value = -1

x = GetSomeValue() || default_value
```

Если GetSomeValue() возвращает "пустое" значение (null, 0, "", undefined, NaN), то x получит значение по умолчанию.

Символьные типы данных

Character types (не путать character и symbol)

Подробнее будем говорить в лекции, посвященной представлению текста в современных ЯП.

Сейчас отметим 2 момента:

- есть ЯП вообще без символьного типа (Go, Python, JavaScript....) - как правило, он есть только в статических языках (Go?). В динамических языках - строки
- если есть, то в современных ЯП - отдельный тип данных, несовместимый явно с целым типом (Паскаль vs C/C++)

Несколько типов данных? Да, если ЯП позволяет одновременно работать со строками в разных кодировках.

Порядковые типы данных

Что выделяет порядковые типы?

Операции succ(x) и pred(x) + отношения полного порядка (=> операции сравнения) + точные левая и правая границы.

Диапазоны и перечисления.

Дапазоны: есть базовый тип (целый или перечислимый) и есть явное ограничение на левую и правую границы

Integer - диапазонный тип?

Самое главное - возможность (квазистатического) контроля за вхождением значения в диапазон.

Паскаль (Н.Вирт - 1969):

type Index = 1..N_MAX;

Модула-2 (Н.Вирт - 1980) :

TYPE IntIndex = [1..N]; NatIndex = CARDINAL[0..99];

Ада(Ж.Ишбиа, 1980):

type Index is Integer range 0..N_MAX;

Порядковые типы данных

Диапазонные типы

Где именно используются? То есть в чем состоит технологическая потребность в диапазонных типах?

Практика использования: диапазоны изменения индекса в массивах.

var i: 1..N; - скорее всего (да почти всегда) i будет индексом в каком-то массиве.

А теперь посмотрим на индексы в современных ЯП(<= 30 лет):

C#, Java, Python, JavaScript, Go, Swift, Rust, ...

ВЕЗДЕ индексы меняются в диапазоне 0..N-1, где N - число элементов.

Оберон (Н.Вирт-1988)

VAR A: ARRAY N OF REAL;

Диапазоны исчезли из современных языков.

Порядковые типы данных

Перечислимые типы (enumeration types)

Явное задание значений через перечень символических имен (возможно, с указанием конкретных целочисленных значений).

Впервые: Паскаль (или Модула-2)

TYPE DaysOfWeek = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

Набор операций: порядковые типы + ord(v)

ord (Sun) = 0, ..., ord(Sat) = 6

M-2 - функция VAL (E, i) => значение v типа E: ORD(i) = v, либо ошибка в случае выхода за границы диапазона корректных значений.

Ада:

type DaysOfWeek is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

+ много атрибутных функций (DaysOfWeek'FIRST, DaysOfWeek'LAST) и других полезных операций (упрощающих ввод-вывод, например).

Перечислимые ТД тесно связаны с другими понятиями ЯП: оператором выбора и объединениями типов (записи с вариантами)

Порядковые типы данных

Перечислимые типы (enumeration types)

Пример на языке Ада (из LRM):

```
type DaysOfWeek is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
Today : DaysOfWeek;
case Today is
        when Mon => Compute_Initial_Balance;
        when Fri => Compute_Closing_Balance;
        when Tue .. Thu => Generate_Report(Today);
        when Sat|Sun => null; -- можно и так: when others => null;
end case;
```

Порядковые типы данных

Перечислимые типы (enumeration types)
Пример на языке Модула-2:

```
TYPE DaysOfWeek = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
VAR D: DaysOfWeek;
CASE D OF
Mon..Thu:
        OfficialDressCode; WorkHard |
Fri:
        CassualDressCode; WorkHalfDay |
Sat,Sun:    /* можно и проще:    ELSE
        Shopping; Rest;Sleep
END
```

Порядковые типы данных

Перечислимые типы (enumeration types)

Примеры согласованности перечислений и объединений обсудим позже (когда будем обсуждать объединения).

Еще одна интересная возможность языка Ада: задание символьных типов как перечислений.

type Hexa is ('A', 'B', 'C', 'D', 'E', 'F'); -- символьный тип

type Mixed is ('A', 'B', '*', B, None, '?', '%'); -- тоже символьный тип

Следствие - литералы перечислений могут совпадать ("hack" - трактуются как нульместные функции => перегрузка).

type Color is (White, Red, Yellow, Green, Blue, Brown, Black);

type TrafficLight is (Red, Amber, Green); -- Red ,Green перегружены

X: Color := Red; -- конечно Color'Red

Y: TrafficLight := Red; -- конечно TrafficLight 'Red

Порядковые типы данных

Перечислимые типы (enumeration types)

```
type Color is (White, Red, Yellow, Green, Blue, Brown, Black);
type TrafficLight is (Red, Amber, Green); -- Red ,Green перегружены
X: Color := Red; -- конечно Color'Red
Y: TrafficLight := Red; -- конечно TrafficLight 'Red
```

Но:

```
procedure Foo(C:Color);
procedure Foo(L: TrafficLight); -- это корректная перегрузка
Foo(X); -- Foo(C:Color)
Foo(Y); -- Foo(L: TrafficLight)
```

Порядковые типы данных

Перечислимые типы (enumeration types)

```
type Color is (White, Red, Yellow, Green, Blue, Brown, Black);
type TrafficLight is (Red, Amber, Green); -- Red ,Green перегружены
X: Color := Red; -- конечно Color'Red
Y: TrafficLight := Red; -- конечно TrafficLight 'Red
```

Но:

```
procedure Foo(C:Color);
procedure Foo(L: TrafficLight); -- это корректная перегрузка
```

А это что за цвет?

```
Foo(Red); -- ошибка компиляции
```

Порядковые типы данных

Перечислимые типы (enumeration types)

type Color is (White, Red, Yellow, Green, Blue, Brown, Black);

type TrafficLight is (Red, Amber, Green); -- Red ,Green перегружены

X: Color := Red; -- конечно Color'Red

Y: TrafficLight := Red; -- конечно TrafficLight 'Red

Но:

procedure Foo(C:Color);

procedure Foo(L: TrafficLight); -- это корректная перегрузка

А это что за цвет?

Foo(Red); -- ошибка компиляции

Еще одно следствие - функция указания типа (не путать с преобразованием типа - к разным перечислимым типам вообще неприменима)

EnumType'Value - трактовать Value как значение типа EnumType

Порядковые типы данных

Перечислимые типы (enumeration types)

type Color is (White, Red, Yellow, Green, Blue, Brown, Black);

type TrafficLight is (Red, Amber, Green); -- Red ,Green перегружены

procedure Foo(C:Color);

procedure Foo(L: TrafficLight); -- это корректная перегрузка

Foo(Red); -- ошибка компиляции

Foo(Color'Red); -- Foo(C:Color);

Foo(TrafficLight'Red); -- Foo(L: TrafficLight)

Авторская позиция - "маленькое" расширение тянет за собой ряд новых конструкций и требований (ЯП усложняются)

Порядковые типы данных

Перечислимые типы (enumeration types)

Подведем (промежуточные) итоги:

70-е-80-е гг прошлого века - любой (императивный) ЯП имел перечислимые типы

Наглядно, надежно, эффективно.

Но:

Оберон (Н.Вирт - 1988) - перечислений НЕТ!

Проблемы с перечислимым типом:

- неявный импорт имен
- нерасширяем (противоречит концепции расширения типов - "наследование")

Кроме этих проблем можно назвать проблему рефлексии.

Порядковые типы данных

Рефлексия - использование свойств исходного программного кода в самой программе. Нас интересует динамическая рефлексия.

Примеры: по имени класса получить список имен методов, по имени метода - список типов аргументов и тип возвращаемого значения.

Рефлексия для перечислений:

- value <=> строка, представляющая имя константы-значения
- для заданного перечислимого типа => массив-строк - имен констант этого типа
- и др.

Почему это проблема?

Нагрузка на runtime-систему для статических языков

(для динамических языков runtime <=> интерпретатор, точнее, часть интерпретатора)

Порядковые типы данных

Итак, в 90-е гг:

Оберон (Н.Вирт - 1988) - перечислений НЕТ!

Java (Дж. Гослинг - 1995) - перечислений НЕТ!

Правда, C# (А.Хейльсберг - 1999) - есть, но обоснование - удобство для визуальных конструкторов.

Проблемы с перечислимым типом:

Проблемы с перечислимым типом:

- неявный импорт имен
- нерасширяем (противоречит концепции расширения типов - "наследование")
- нерасширяем (противоречит концепции расширения типов - "наследование")

Современная точка зрения на перечисления: важный и обязательный ТД.

Как решены проблемы?

- неявный импорт имен => имена перечислимых констант видимы в объемлющей ОВ только ПОТЕНЦИАЛЬНО - через обязательное уточнением именем перечислимого типа
- нерасширяем (противоречит концепции расширения типов - "наследование") - перестали считать проблемой, так как ООП перестало быть единственной и определяющей ПП

Порядковые типы данных

Современные ЯП реализуют также дополнительные технологические требования к перечислимым типам:

- удобство(рефлексия)
- возможность явного задания числовых значений, включая использование комбинаций значений (RDWR = RD_ONLY|WR_ONLY)
- надежность (нет неявных преобразований в integer и обратно, но есть явные)
- эффективность и гибкость представления (базовый тип м/б отличен от int)

Порядковые типы данных

Пример перечислимых типов - C#

```
enum Color {
        Red = 0xFF0000,
        Green = 0x00FF00,
        Blue = 0x0000FF
}
Color c = Color.Red; // явное уточнение
enum TrafficLight : byte { // базовый тип - байт
        Red, Amber, Green
}
var tl = TrafficLight.Red;
[Flags]
enum OpenMode {
        RD_ONLY = 1,
        WR_ONLY = 2,
        RDWR  = RD_ONLY | WR_ONLY
}
int m = OpenMode.RD_ONLY; // ОШИБКА трансляции
int m = (int) OpenMode.RD_ONLY; // OK
```

Неявно наследуется от класса Enum, который содержит много полезных и удобных методов - Enum.ToString(), Enum.ToString("D"), Enum.GetNames(typeof(Color)), Enum.GetValues(typeof(TrafficLight)), Enum.TryParse<Color>("Green", out c) и др.

Порядковые типы данных

Пример перечислимых типов - C++11

```
enum class Color {
        Red = 0xFF0000,
        Green = 0x00FF00,
        Blue = 0x0000FF
}
Color c = Color::Red; // явное уточнение
enum TrafficLight : unsigned char { // базовый тип - байт
        Red, Amber, Green
}
int m = Color::Green; // ОШИБКА трансляции!
int m = (int) Color::Green; // ОК
```

Порядковые типы данных

Пример перечислимых типов - Java (2005 - Java 5)
Самое "оригинальное" решение, хотя внешне все - "как у людей"

```
enum DaysOfWeek {
   Sun, Mon, Tue, Wed, Thu, Fri, Sat
}
DaysOfWeek today = Wed; // ОШИБКА!!!
DaysOfWeek today = DaysOfWeek.Wed; // OK!!!
```

Неявно (явно нельзя - все как в C# - но только внешне!!!) наследуется от класса Enum (точнее java.lang.Enum) с кучей методов:
today.name() => "Wed", today.toString()
today.ordinal() => 3
DaysOfWeek.values() => [Sun, Mon, Tue, Wed, Thu, Fri, Sat]
DaysOfWeek.valueOf("Wed") => DaysOfWeek.Wed
today.compareTo(DaysOfWeek.Sun) => работают все операции сравнения

Порядковые типы данных

Пример перечислимых типов - Java (2005 - Java 5)

В чем "экзотика"?

В C# - перечисления - типы-значения (как int, long, byte и др. целочисленные типы). Наследование от System.Enum - для компилятора и рефлексии.

В Java - перечисления - это классы (настоящие)!

Константы перечисления - это публичные статические константные члены этого класса (методы values() и valueOf() генерируются как статические методы класса)

Можно делать конструкторы перечислимого класса, методы и все остальное, что можно делать с классом, включая наследование.

Порядковые типы данных

```
import java.lang.*;
import java.io.*;

class Ideone
{
       public enum Color {
         Red(0xFF0000),  //calls constructor with value = 0xFF0000
         Green(0xFF00),  //calls constructor with value = 0xFF0000
         Blue (0xFF)   //calls constructor with value = 0xFF0000
         ; // semicolon needed when fields / methods follow
         private final int rgb;
         Color(int rgb) {
             this.rgb = rgb;
         }

         public int getRGBColor() {
             return rgb;
         }

     }
     public static void main (String[] args) throws java.lang.Exception
     {
        for (Color c: Color.values()) {
               System.out.println(c + " " + c.getRGBColor());
        }
     }
}
```

Конечно, метод ordinal() выдает 0,1,2 по-прежнему

Порядковые типы данных

Еще пример языка, в котором не было перечислений, но они появились:
Python (>=3.4) - библиотечная поддержка (язык настолько гибок, что это можно сделать на уровне библиотеки) - модуль enum

```
from enum import Enum
class Color(Enum):
        RED=1
        GREEN=2
        BLUE=3
a = Color.RED
print(a) # Color.RED
print(repr(a)) # '<Color.RED: 1>'
```

BLUE

Есть и функциональная нотация:

```
Color = Enum('Color', 'RED GREEN BLUE')  # тот же самый эффект
list(Color) # ['<Color.RED: 1>', '<Color. GREEN: 2>', '<Color. BLUE : 3>']
```

Это - класс (со всеми вытекающими поопоследствиями)

Порядковые типы данных

Пример языка с перечислениями, которые так не называются...

Язык Go

```
const ( // прямо как в языке C - просто перечислить целые константы
	Sun    int = iota // Sun == 0
	Mon               // Mon == 1
	Tue               // Tue == 2
	Wed               // …
	Thu
	Fri
	Sat
)
```

iota - ключевое слово - обнуляется, как только встречается ключевое слово const

Можно выбрать другой базовый тип....

Порядковые типы данных

Язык Go - введем новый тип данных и набор констант для него

```
type Color int
const (
        Red Color = iota
        Green
        Blue
)
```

Порядковые типы данных

Язык Go - новый тип данных + функция перевода в строку для этого типа

```
package main
import "fmt"
type Color int
const (
	Red Color = iota
	Green
	Blue
)
func (c Color) String() string {
	return [...]string{"RED", "GREEN", "BLUE"}[c]
}

func main(){
	a := [...]Color{Red, Green, Blue}
	for k:= range a{
		fmt.Println(a[k])
	}
}
```

Вопрос - ГЛАВНОЕ отличие от C#, Java, C++ enum class?

Порядковые типы данных

Вывод: в современных ЯП перечислимый ТД, в отличие от диапазонов, активно используется и развивается.

Новые аспекты перечислений - в языках Rust и Swift.

Пример из Rust (https://doc.rust-lang.org/rust-by-example/custom_types/enum.html)

Порядковые типы данных

Пример из Rust (https://doc.rust-lang.org/rust-by-example/custom_types/enum.html)

```
enum WebEvent {
        // `enum` может задавать просто набор констант
        PageLoad,
        PageUnload,
        // либо кортежи
        KeyPress(char),
        Paste(String),
        // либо с-подобные структуры
        Click { x: i64, y: i64 },
}
fn inspect(event: WebEvent) {
        match event { // аналог оператора выбора
                WebEvent::PageLoad => println!("page loaded"),
                WebEvent::PageUnload => println!("page unloaded"),
                // вытаскиваем `c` из кортежа- `enum`.
                WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
                WebEvent::Paste(s) => println!("pasted \"{}\".", s),
                // Вытаскиваем `Click` в `x` и `y`.
                WebEvent::Click { x, y } => {
                        println!("clicked at x={}, y={}.", x, y);
                },
        }
}
```

Порядковые типы данных

Пример из Rust (https://doc.rust-lang.org/rust-by-example/custom_types/enum.html)
Главная программа:

```
fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` создает owned-строку `String` из строкового среза.
    let pasted  = WebEvent::Paste("my text".to_owned());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
}
```