Понятие управления в кибернетике. Понятие алгоритма. Машина Тьюринга. RAM-машина. Тезис Черча-Тьюринга. Универсальная машина Тьюринга. Значение теоремы об универсальной машине Тьюринга. Структура компьютера по фон Нейману. Принципы архитектуры фон Неймана.

Информационный процесс – набор действий с информацией

Информационная система – система, где реализованы информационные процессы

Управление подразумевает, что информационные процессы должны идти так, как требуется, и что информационная система должна принимать решение об изменении информационных процессов на основе получаемой информации. Кибернетика – наука об управлении.

Алгоритм – последовательность шагов.

Машина Тьюринга — абстрактный исполнитель (абстрактная вычислительная машина), способный имитировать всех исполнителей.

Состоит из алфавита, состояний (память), ленты (бесконечной), считывающей головки и программы.

Random Access Machine (равнодоступная адресная машина, сокращённо *RAM-машина*) — модель машины с одним сумматором, команды программы не могут изменять сами себя. Служит теоретической моделью.

Состоит из входной (с целыми числами) и выходной (изначально пустой) лент, памяти (последовательность регистров, которые могут хранить целые числа). Программа машины хранится вне её памяти, поэтому программа не может изменять саму себя.

Тезис Черча-Тьюринга - для любой алгоритмически вычислимой функции существует вычисляющая её значения машина Тьюринга,

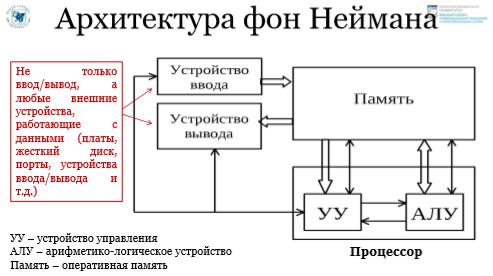
Универсальная машина Тьюринга – машина Тьюринга, которая может заменить собой любую машину Тьюринга. Получив на вход программу и входные данные, она вычисляет ответ, который вычислила бы по входным данным машина Тьюринга, чья программа была дана на вход.

Эта теорема «дала жизнь программированию».

Архитектура фон Неймана – принцип совместного хранения команд и данных в памяти компьютера.

Состоит из памяти, УУ (устройства управления), АЛУ (арифметико-логическое устройство), ввода и вывода.

Принципы: Однородность памяти (команды и данные хранятся в общей памяти), адресность (память – пронумерованные ячейки), программное управление (всё выполняется через программы) и двоичное кодирование.



Отличие языков высокого и низкого уровня. Компиляторы и интерпретаторы. Что такое JVM, JDK, JRE?

Ассемблер – языки низкого уровня (детализация на уровне процессора, машинный код). Нет циклов, но есть метки.

Языки высокого уровня • Уход от адресов, регистров и операций «переноса» (низкоуровневых операций), • Переход к понятию «переменная». • Возможность использовать условия и циклы. • Акцент на обработке данных: (Алгоритм обрабатывает цифровые данные (цифры и числа), значит он должен уметь что-то делать с числами – а это математические операции) • Первый язык высокого уровня – Fortran. (больше абстракций)

Компилятор – выполняет целиком трансляцию программы из языка высокого уровня в машинный код (дальше можно запустить этот машинный код (например, .exe)

Интерпретатор – выполняет построчную (покомандную) трансляцию и выполнение (например, Python, PHP – интерпретаторы)

Java – компилируемый язык, но превращается в байт-код, который запускается виртуальной машиной Java (.class-файлы). В литературе называется интерпретатором компилирующего типа.

Java-приложение транслируется в байт код, который потом выполняется JVM (Java Virtual Machine) (виртуальной машиной Java, реализующей кросс-платформенность (перенос на другую аппаратную платформу или операционную систему) для любого приложения на Java.

Java Development Kit (JDK) – комплект, включающий в себя компилятор Java (javac), стандартные библиотеки классов *Java*, примеры, документацию, различные утилиты и исполнительную систему *Java* (JRE).

Java Runtime Environment (JRE) – минимальная реализация виртуальной машины. Состоит из JVE и библиотеки Java-классов.

Понятие типа данных, трактовка термина тип. Примитивные типы данных, различия между примитивными и ссылочными типами данных. Приведение типов. Арифметические и логические операции. Хранение отрицательных целых чисел. Преобразование типов в при арифметических операциях. Сужение при преобразовании типов. Логические операции.

Тип данных – множество значений и операций над этими значениями. Определяет все возможные значения переменной и размер данных, которые хранятся в ней.

Примитивные типы данных – переменные, в которых хранится само значение. Память под переменную примитивного типа выделяется в момент объявления. Хранятся в Stack.

Виды: byte(1), short(2), int(4),long(8),char(2),Boolean(1),float(4),double(8);

Ссылочные типы данных – переменные, которые хранят ссылку на своё значение. Объявление переменной и выделение памяти – два разных действия. Null – пустая ссылка. Хранятся в Heap.

а++ инкремент, а—декремент.

Условные и циклические операторы языка. Условные операторы if, switch case, (P ? v1 : v2). Циклические операторы while, do while, циклы for и for each, операторы break и continue. Трасса. Теорема Бѐма-Якопини.

**switch** (ВыражениеДляВыбора) {

**case** (Значение1):

Код1;

**break**;

**default**:

КодВыбораПоУмолчанию;

**break**;

}

double y = P ? A : B; присвоить А, если верно P, иначе присвоить B (тернарный оператор).

do {

Последовательность команд

} while (условие)

**for** (type itVar: array)

{

Блок операторов;

} по всем элементам массива

break – обрывает выполнение текущей конструкции (чаще всего цикл)

continue – для цикла мгновенно начинает следующую итерацию.

Трасса - Последовательность команд, которые были вызваны при выполнении.

Теорема Бёма-Якопини (1965-1966 гг.) Любая программа, заданная в виде блок-схемы, может быть представлена с помощью последовательности, ветвления, цикла (; if-else while).

Проблема дублирования кода. Решение.

Прописать алгоритм в действии.

Вычислительная сложность. Сложность по времени и памяти. Подход к измерению сложности. Виды сложности. Примеры алгоритмов с заданной сложностью. Классы P и NP.

Вычислительная сложность – зависимость объема работы от размера входных данных.

T(n) – временная сложность (сложность по времени)

S(n) – пространственная сложность (сложность по памяти)

Подход – важно знать, насколько сильно растет сложность при увеличении размера входа. Интересует больше всего сложность в худших случаях.

Виды: Полиномиальная (два цикла рядом, инициализация элементов массива) – O(𝒏 𝒌 ) – Частные случаи – линейная и константная • Экспоненциальная (цикл в цикле, перебор комбинаций) – O(𝒌 𝒏 ) • Логарифмическая(Math.pow(x.y)) – O(𝐥𝐨𝐠 𝒏). Класс P – задачи, решаемые детерминированной машиной Тьюринга за полиномиальное время, вычислительно не сложные и ограничены сверху по времени, существуют быстрые алгоритмы решения.

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

O(log n) — логарифмическая сложность

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

O(n2) — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

класс NP – задачи, решаемые детерминированной машиной Тьюринга за экспоненциальное время. (так же теоретически решаемые недетерминированными машинами за полиномиальное время (т.е. ограничены сверху по времени полиномом – многочленом от n перемннных), но недерминированные машины не существуют в природе, т.к. они должны сразу переходить на нужную итерацию (шаг) алгоритма и находить верное решение, алгоритмов быстрого решения нет (задача о коммивояжере))

Машина Тьюринга называется *детерминированной*, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила.

(разница между p/np – задачами!? Способы определения сложности алгоритма.)

Подходы к приближенному вычислению бесконечных величин (например, сумм).

Вычисления не уходят в бесконечность. Фиксируется точность, и величины вычисляются до тех пор, пока эта точность не будет соблюдаться.

Массивы, действия с ними. Одномерные массивы объявление ссылки и выделение памяти. Обращение к элементу массива. Прямая адресация – как устроена? Как пройти по всем элементам массива. Цикл for each. Многомерные массивы - объявление, способ пройти по всем элементам. Ступенчатые массивы.

int n = 10;

int [][] a = new int[n][];

a[0] = new int[n];

a[1] = new int[n-1];

Ступенчатый массив. Подмассивы не одного размера.

Задачи с кванторами. Предикаты. Способ вычисления кванторов.

Предикат – суждение о предмете.

Кванторы – верно для всех или есть хотя бы один, для которого неверно (По сути, одно и то же)

Делать оптимально – прерывая. Через флаг (boolean flag = true) или break, или return.

Символы и строки. Тип char, номер символа в ASCII. Способ узнать номер. Что такое кодировка? Класс String. Объявление, использование (ввод-вывод, конкатенация, характерные методы). Особенности хранения и создания строковых констант в Java.

Кодировка символов (часто называемая также *кодовой страницей*) – это набор числовых значений, которые ставятся в соответствие группе алфавитно-цифровых символов, знаков пунктуации и специальных символов.

**Конкатенация** – склеивание строк.

Текстовые и бинарные файлы. Как считать данные из текстового файла.

File f = new File("input.txt") – связываем переменную с файлом, из которого считываем

Scanner scanner = new Scanner(new File("input.txt"));

int s = 0;

do {

int x = scanner.nextInt();

s += x; }

while (scanner.hasNextInt());

System.out.println(s);

scanner.close();

Методы. Объявление и вызов. Стек вызовов. Возвращаемые значения, return. Параметры, передача по ссылке, по значению. Перегрузка метода. Рекурсия. Проблемы рекурсии. Мемоизация. Примеры рекурсивных алгоритмов.

Метод – функция у класса.

Стек вызовов – множество вызовов.

Сигнатура – список параметров

Перегрузка - объявление методов с одинаковыми именами, но разными наборами параметров • Работает потому, что сигнатуры разные

Мемоизация - Запоминать вычисленные на предыдущих шагах значения • Использовать вычисленные значения вместо нахождения повторного решения. (тоже, что и кэш)

Классы и объекты. Классы. Структура класса, члены класса. Объявление объекта класса. Объявление ссылки и выделение памяти. Конструктор класса, оператор this. Методы и атрибуты в классе. Примеры.

Класс - Тип данных, состоящий из набора атрибутов и методов.

Объект – переменная этого типа.

• Экземпляр (instance) класса (хорошо)

• Объект класса (плохо) – У класса есть атрибуты и методы, у него нет объектов

Инкапсуляция. Модификаторы доступа - private, protected, public. Методы setget. Модификатор static. Объяснение модификаторов main.

Инкапсуляция – скрытие реализации объекта.

Static – общее для всех. Существует даже тогда, когда нет экземпляров.

public – чтобы ее могли запустить извне • static – потому что main не принадлежит конкретному объекту. Метод main должен запуститься, когда еще ни один объект не создан.

Пакеты. Необходимость пакетов. Оператор import. Компилирование и выполнение с помощью командной строки, учитывая пакеты. Параметры запуска команд в командной строке при компиляции. Модификатор «по умолчанию». Оператор import static.

Пакет (package) – набор классов и вложенных пакетов

Import – объявление класса, методы которого мы будем использовать в нашем коде. Теперь не нужно постоянно прописывать название класса перед названием метода.

Import static – вызов статических методов. \* - вызов всех методов в этом пакете, но не методов подпакета.

Чтобы заработало, нужно

1. создать иерархию папок, соответствующую иерархии пакетов

2. поместить исходные файлы в нужные места

3. поместить все полученное в специальную папку для исходников – исходники по-английски – source

4. компилировать из корня проекта (та папка, которая содержит sources).

(javac src\work\projects\kickgame\Game.java) (находимся в папке Project)

5. Надо при компилировании Game сказать, где лежит Player.java

6. Параметр/ключ для команды в командной строке под названием sourcepath

(javac -sourcepath src src\work\projects\kickgame\Game.java)

Разделение при компиляции

• Параметр компиляции -d : Указывает, куда поместить соответствующие скомпилированные файлы?

• Указываем как значение папку рядом с src

• Т.к. будет аналогичная иерархия папок для пакетов

• Название этой папки варьируется: bin, out, classes. Мы возьмем bin.

(javac -sourcepath src -d bin src\work\projects\kickgame\Game.java)

При запуске необходимо указывать, откуда брать .class-файлы, необходимые для запуска. Этот параметр называется classpath (при компиляции можно писать cp)

Переменная, содержащая путь (или пути через ;) к необходимым бинарникам: • Путь к корню иерархии пакетов проекта • Пути к библиотекам

В нашем случае classpath – папка bin.

java -classpath bin work.projects.kickgame.Game

Модификатор по умолчанию (default) «Прямой доступ извне разрешен всем «однопакетникам» (классам, объявленным в том же пакете)»

Наследование и полиморфизм. Правила наследования методов и атрибутов. Как наследование устроено с точки зрения памяти. Оператор super.

Наследование - Классы могут использовать готовую реализацию других классов, добавляя лишь то, чего не хватает в исходном (базовом, супер, над-, родительском классе)» (По-английски – Inheritance) Причины – дублирование кода, любое изменение родителя влечёт изменения в дочернем классе.

Полиморфизм - Возможность реализовывать уникальное поведение у нескольких подклассов при едином интерфейсе суперкласса. **( В ООП)**

Extends **–** расширяет

В дочернем конструкторе всегда вызывается родительский, первым же оператором

super – обращение к конструктору родительского класса

Добавляются новые атрибуты, методы

Дочерний класс содержит в себе родительский класс + свою уникальную часть ( « »)

Переопределение методов. Класс Object. Восходящее преобразование. Связывание. Позднее связывание. Проблемы множественного наследования. Абстрактные классы и методы.

Переопределение - Изменение потомком реализации родительского метода.

Object – корень иерархии классов, от него наследуется любой класс.

Final class – нельзя наследовать, метод – нельзя переопределять

Восходящее преобразование - Сужение интерфейса потомка до интерфейса родителя.

Связывание (binding)

• Присоединение вызова метода к телу метода.

• contract.printInfo() – вызов

• методы printInfo() есть у нескольких классах.

в Java - позднее связывание

• Реализация вызываемого метода определяется в момент выполнения!

• Компилятор не знает заранее, какая реализация printInfo будет использована для contract.printInfo()!

• Но JVM потом разрулит все, опираясь на созданные объекты. » !

• У договора тип – Contract, но в момент вызова:

• у тех, кто по факту IndividualContract, вызовется их реализация;

• у тех, кто по факту CompanyContract, вызовется их реализация;

Родитель может быть только один (проблема множественного наследования). Ибо непонятно, чьи методы/конструкторы наследовать, к кому идёт обращение super. (какую реализацию брать)

Абстрактный класс – класс, который не может иметь экземпляра. Abstract class

Абстрактный метод – метод без реализации.

Интерфейсы. Объявление, структура. Что могут содержать интерфейсы? Наследование интерфейсов. Правила реализации интерфейса классом. Восходящее преобразование в случае интерфейса. Анонимные классы. Модификатор default.

Интерфейс – сущность в джаве с абстрактными публичными методами и без атрибутов. Реализации нет 🡪 можно наследовать от нескольких интерфейсов. Интерфейсы задают поведение объекта.

public interface DeviceInterface {

void on();

void off();

String getInfo();

}

Все методы – public abstract (до Java 8). Ну и после по умолчанию

Наследующий класс обязан либо реализовать все методы интерфейса, либо быть абстрактным

public class SmartPhone extends Phone implements MakePhotoInterface{ public void makePhoto() {…} } }

Восходящее преобразование также работает

Анонимный класс - Мгновенная реализация интерфейса или абстрактного класса в коде

public static void main(String[] args) {

MakePhotoInterface camera = new MakePhotoInterface() {

public void makePhoto() {

System.out.println("Make photo as Nikon Camera");

}

};

Default метод в интерфейсе определен по умолчанию, и при реализации его можно не определять. Но это не о множественном наследовании

Перечисления. Необходимость, характерные методы. Чем являются элементы перечисления.

Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую.

enum Day{

    MONDAY,

    TUESDAY,

    WEDNESDAY,

    THURSDAY,

    FRIDAY,

    SATURDAY,

    SUNDAY

}

Необхрдимость : Чтобы у переменных этого типа явно было видно значение.

• Чтобы можно было легко перебрать все его значения,

• Чтобы не хардкодить,

values() возвращает массив из всех значений перечисления

Type[] types = Type.values();

Day current = Day.MONDAY;

        System.out.println(current);    // MONDAY

Метод **ordinal()** возвращает порядковый номер определенной константы (нумерация начинается с 0):

public static void main(String[] args) {

        Operation op = Operation.SUM;

        System.out.println(op.action(10, 4));   // 14

        op = Operation.MULTIPLY;

        System.out.println(op.action(6, 4));    // 24

    }

}

Также можно определять методы для отдельных констант:

enum Operation{

    SUM{

        public int action(int x, int y){ return x + y;}

    },

    SUBTRACT{

        public int action(int x, int y){ return x - y;}

    },

    MULTIPLY{

        public int action(int x, int y){ return x \* y;}

    };

    public abstract int action(int x, int y);

}

Элементы перечисления являются объектами.

.Исключения. Примеры известных. Блок try-catch, правила работы. Использование finally. Корректная последовательность обработки исключений. Что такое «проверяемые» и «непроверяемые» исключения.

SOLID-принципы проектирования.

Принципы, при которых приложение будет легче/надежнее поддерживать и расширять в течение долгого времени.

Single responsibility (SRP) • Принцип единственной ответственности

Должна быть единственная причина появления/изменения класса

Представьте, что класс Contract одновременно:

• Содержит методы Contract – обновить дату, узнать заказчика и т.п.

• Содержит методы для работы с хранилищем – вернуть договора 2016 года, удалить все договоры с заказчиком «Рога и Копыта».

Это и было бы нарушением принципа SRP, но в нашем примере мы разделяли этот функционал между Contract и Storage.

Соблюдение SRP должно защищать от появления божественного объекта.

Open-closed (OCP) • Принцип открытости/закрытости

Сущности должны быть открыты для расширения, но закрыты для модификации

Разработанный (вошедший в определенную версию) класс неприкосновенен для изменений

• Можно только исправлять ошибки

• Новый функционал – в новые классы, которые могут переопределять существующие (и должны, вообще говоря).

Liskov substitution (LSP) • Принцип подстановки Барбары Лисков

Замена в коде экземпляров классов на экземпляры их подклассов (наследников) не должна влиять на правильность работы программы

Проще: Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты подклассов, не зная об этом.

Некорректное переопределение:?\*\*\*

Interface segregation (ISP) • Принцип разделения интерфейса

Много интерфейсов, предназначенных для разных пользователей (других классов) лучше одного большого интерфейса, в который свален весь функционал (чтобы не реализовывать левые методы)

Dependency inversion (DIP) • Принцип инверсии зависимости

Зависимости классов должны опираться на абстракции. Зависимости не должны опираться на конкретную реализацию.

Нельзя вводить зависимость от конкретной реализации интерфейса, лучше уж от самого интерфейса. А дальше восходящее преобразование

Зацепление (coupling)

• Сильное (тесное – strong, tight, high) – использование в зависимостях конкретных реализаций, конструкторов.

• Слабое (weak, loose, low) – зависимость на уровне интерфейсов.

• В нашем примере Worker сильно связан с WebApp (т.к. мы можем вызывать метод WebApp, которого нет в App)

– Что означает, что Worker уже заточен только на работу с WebApp

Программировать же нужно «На уровне интерфейсов»

public class Worker {

private App app;

public Worker(App app) {

this.app = app;

} – хороший вариант

Вопросы в целом:

1. Цикл for – each
2. P/NP – задачи
3. Точное определение слодности алгоритма
4. Операции с кванторами
5. Чтение текстовых файлов (FileReader/FileWriter)