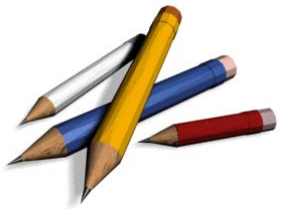
Виртуальные машины

Виртуальные машины

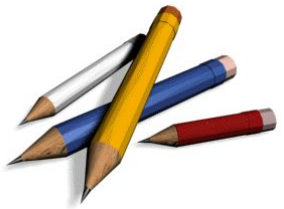
Одним из распространенных подходов, применяемых при разработке переносимого программного обеспечения, является использование *виртуальных машин*.

Виртуальные машины представляют собой практическую реализацию теоретических *абстрактных вычислительных машин* (таких как, например, машина Тьюринга или машина Поста)



Виртуальная машина

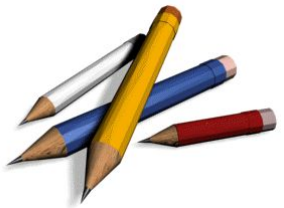
совокупность ресурсов, которые эмулируют поведение реальной вычислительной машины



Виртуальные машины

Концепция виртуальных машин была развита в 1960-е гг. при разработке операционных систем третьего поколения как расширение концепции виртуальной памяти.

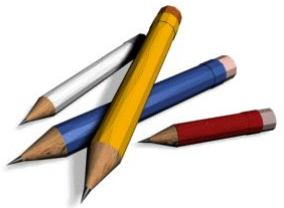
Первоначальной целью разработки виртуальных машин была организация одновременной работы нескольких вычислительных процессов в рамках одной физической системы (т.е. многозадачный и/или многопользовательский режим).



Виртуальные машины

В соответствии с концепцией ВМ, *вычислительный процесс полностью определяется содержимым того рабочего пространства (памяти), к которому он имеет доступ.*

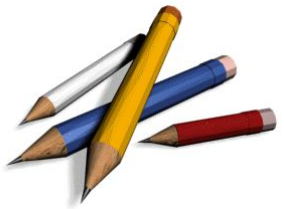
При условии, что конкретная ситуация в рабочем пространстве процесса соответствует ожидаемой, процесс не имеет никаких средств для определения того, является ли предоставленный ему ресурс действительно физическим ресурсом этого типа, или же он реализован в результате совместных действий других ресурсов, которые в совокупности приводят к аналогичным изменениям содержимого рабочего пространства процесса.



Виртуальные машины

В виртуальной машине ни один процесс не может монопольно использовать никакой ресурс, и все системные ресурсы потенциально считаются ресурсами совместного использования.

Использование виртуальных машин обеспечивает изоляцию нескольких процессов и обеспечивает определенный уровень защиты данных.

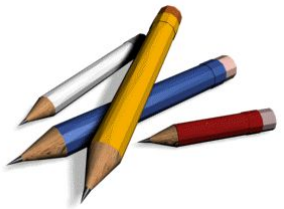


Виртуальные машины

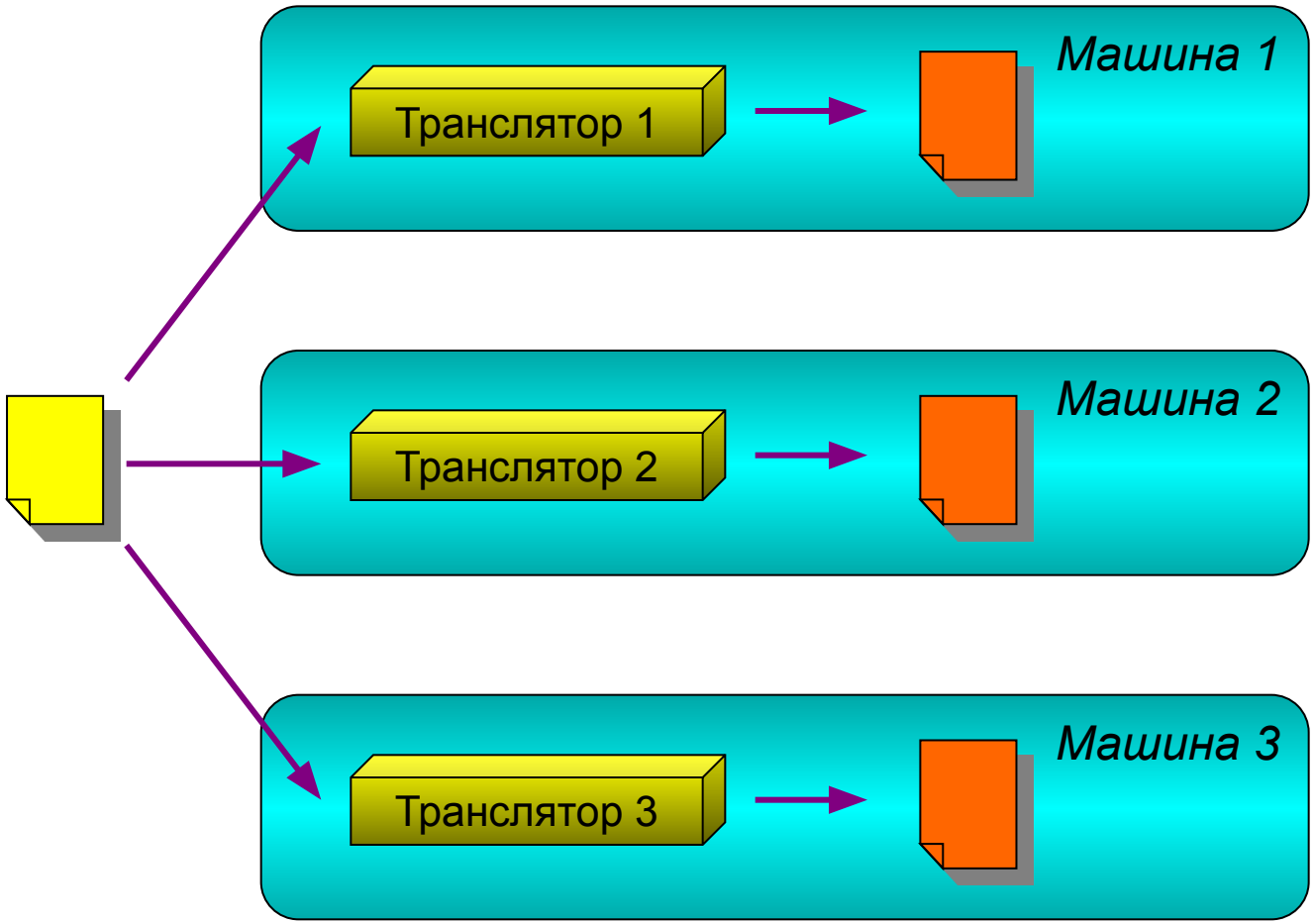
Начиная с 1970-х гг. виртуальные машины стали использоваться при разработке трансляторов языков программирования.

При таком подходе исходный язык транслируется в коды некоторой специально разработанной вычислительной машины, которая не обязательно существует.

Затем для каждой целевой платформы пишется интерпретатор виртуальной машины.

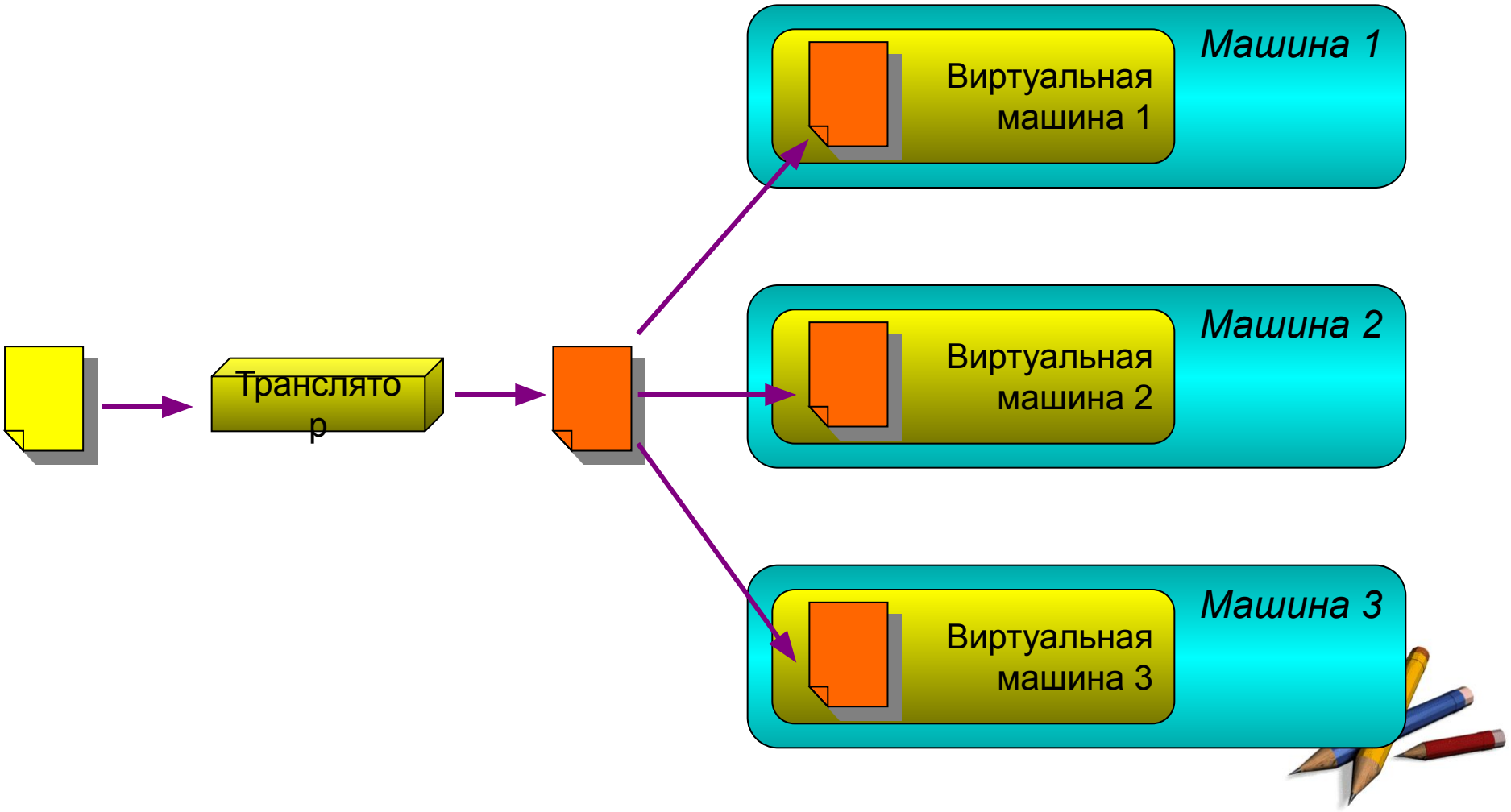


Без виртуальной машины



Виртуальные машины

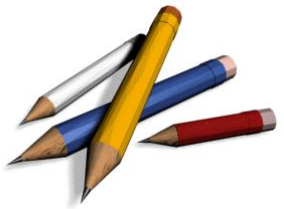
С использованием виртуальной машины



Виртуальные машины

Архитектура виртуальной машины обычно проектируется разработана таким образом, чтобы конструкции исходного языка удобно отображались в систему команд виртуальной машины, но и сама система команд при этом не была бы слишком сложной.

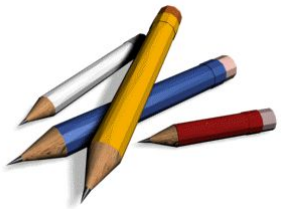
При выполнении этих условий можно достаточно быстро написать интерпретатор виртуальной машины для требуемой целевой платформы.



Виртуальные машины

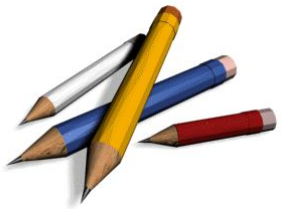
Одна из первых широко известных виртуальных машин такого рода была разработана Никлаусом Виртом при написании компилятора Pascal-P.

Этот компилятор генерировал специальный код, названный P-кодом и представляющий собой последовательность инструкций гипотетической стековой машины.



Виртуальные машины

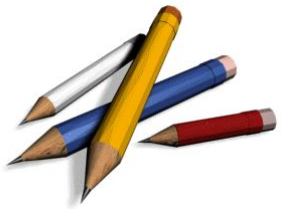
В настоящее время концепция виртуальных машин приобрела широкую известность благодаря языку Java, компиляторы которого генерируют последовательность команд для виртуальной Java-машины.



Архитектура виртуальной машины Java

Виртуальная машина Java описывается следующими характеристики:

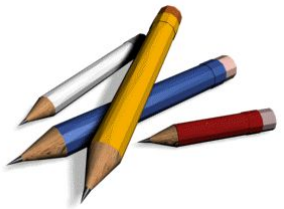
1. Поддерживаемые типы данных
2. Регистры
3. Локальные переменные
4. Стек операндов
5. Область метода
6. Среда выполнения
7. Куча и сборщик мусора
8. Система команд
9. Ограничения виртуальной машины



Поддерживаемые типы данных JVM

– почти все примитивные типы данных Java, за исключением boolean:
byte, short, int, long, float, double, char
– ссылочный тип данных *object*;
– адрес возврата *returnAdress*

Для примитивных типов данных контроль соответствия типов осуществляется на этапе компиляции, для ссылочных типов – на этапе исполнения.



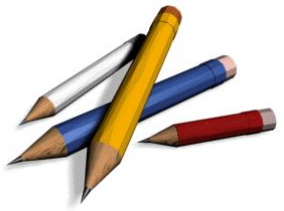
Регистры JVM (32-разрядные)

pc
адрес следующей команды, которая будет исполняться виртуальной машиной;

vars
адрес области памяти для хранения локальных переменных метода;

optop
адрес вершины стека операндов;

frame
адрес структуры среды исполнения

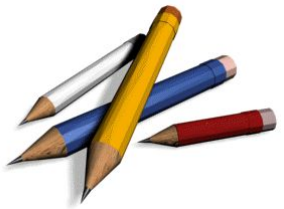


Локальные переменные

Локальные переменные метода хранятся в области памяти, начиная с адреса **vars**.

Для хранения одной переменной отводится 4 байта, значения типа *long, double* занимают 8 байт (как две переменные).

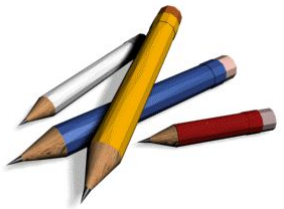
Для доступа к переменным указывается смещение от начала области (0, 4, 8, 12 и т.д.)



Стек операндов

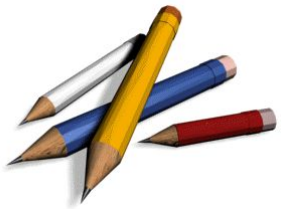
Стек операндов используется для передачи параметров методам и получения возвращаемого значения метода, а также для выполнения арифметических действий и хранения их результатов.

Каждый элемент стека операндов имеет размер 4 байта, значения всех типов хранятся в одной ячейке стека, значения типов *long, double* – две ячейки стека.



Область метода

Область метода хранит код метода и таблицы символов метода.

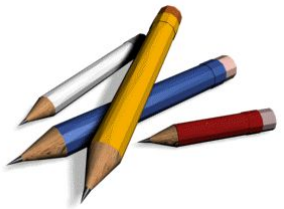


Среда исполнения

содержит информацию, которая характеризует:

- динамическую компоновку объектов и их методов;
- информацию для возврата из методов;
- информацию для распространения исключений;

Среда исполнения может быть дополнена специфической информацией, например, отладочной информацией.

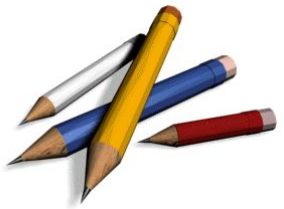


Виртуальные машины

Среда исполнения содержит ссылки на таблицу символов текущего метода.

Вызовы методов и обращения к полям классов осуществляются с использованием символических ссылок.

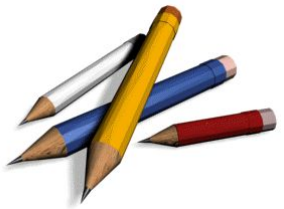
Во время динамической компоновки виртуальная машина переводит эти символические вызовы методов в фактические вызовы методов, загружая классы по мере необходимости.



При нормальном завершении текущего метода управление передается в вызывающий метод.

Среда выполнения используется для того, чтобы восстановить регистры вызывающего метода.

Выполнение продолжается в среде исполнения вызывающего метода.

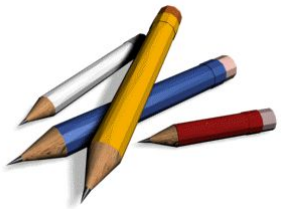


Виртуальные машины

При возникновении исключения осуществляется анализ *таблицы исключений*, связанной с текущим методом.

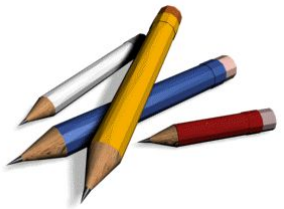
Каждая запись таблицы исключений определяет диапазон программного кода, для которого она действительна; содержит тип и адрес кода для обработки исключения.

Если в таблице исключений найдена запись, соответствующая возникшему исключению, то виртуальная машина передает управление соответствующему обработчику.



Если запись, соответствующая исключению, не найдена, то результатом текущего метода является исключение.

Состояние вызывающего метода восстанавливается по среде исполнения, и распространение исключения продолжается, как если бы исключение только что произошло в этом вызывающем методе.

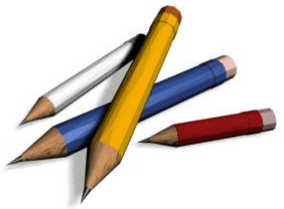


Куча и сборщик мусора

Во время исполнения Java-программы объекты хранятся в динамически распределяемой памяти – куче.

Периодически осуществляется "сборка мусора" – поиск и уничтожение объектов, не используемых в программе (на которые отсутствуют ссылки).

Для реализации сборщика мусора используются различные алгоритмы, в зависимости от требований системы.

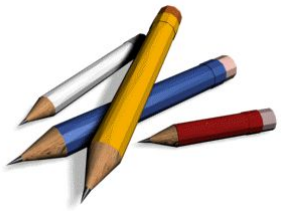


Система команд виртуальной машины

Инструкция виртуальной машины Java состоит из однобайтового кода операции, определяющего действие, которое будет выполнено, и может сопровождаться дополнительными операндами.

Число и размер дополнительных операндов определяется кодом операции.

Если дополнительный операнд содержит более одного байта, то они хранятся в порядке от старшего разряда к младшему.



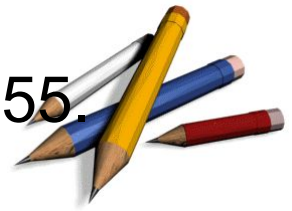
Ограничения виртуальной машины

Большинство существующих ограничений обусловлены тем, что для представления индексов операндов команд отводится 1-2 байта (8-16 бит).

Список констант ("пул констант") для каждого класса может содержать максимум 65535 элементов.

Длина кода метода ограничена 65535 байтами (включая код метода, таблицу исключений, таблицу номеров строк и таблицу локальных переменных).

Число аргументов в вызове метода ограничено 255.



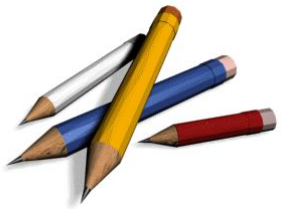
Система команд виртуальной машины Java

Система команд виртуальной машины Java включает в себя 201 инструкцию. Большинство из них относятся к двум группам:

– команды работы со стеком;

– арифметические команды.

Особенностью системы команд JVM является то, что имеется много схожих инструкций, отличающихся только обрабатываемым типом данных.



Команды загрузки констант в стек (15 команд)

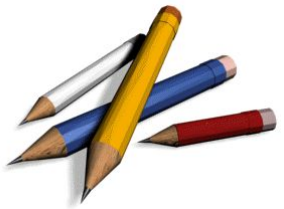
предназначены для загрузки в стек операндов часто используемых констант (null, -1, 0, 1, …)

```
типconst_значение
```

Примеры.

iconst_0 – загрузить в стек целочисленную константу 0

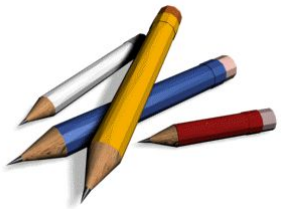
dconst_1 – загрузить в стек вещественную константу 0



Команды загрузки констант из пула констант

предназначены для загрузки в стек значений констант, находящихся в пуле констант. Загрузка выполняется безразлично к фактическому типу значений.

```
ldc индекс
ldc2 индекс
```



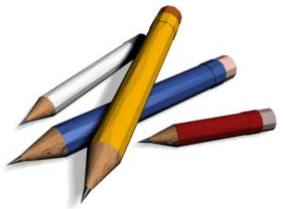
Команды загрузки значений переменных в стек (33 шт.)
предназначены для загрузки в стек значений локальных переменных из списка локальных переменных

```
типload индекс
```

Примеры.

iload 5 – загрузить в стек пятую локальную переменную (целочисленную)

dload 10 – загрузить в стек десятую локальную переменную (вещественную)

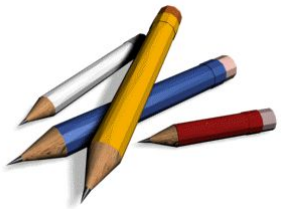


Особенности команд загрузки переменных:

1. Параметры методов хранятся как локальные переменные (с индексами 0, 1, 2, …)

2. Для часто используемых индексов (0, 1, 2, 3) имеются сокращенные варианты команд, занимающие один байт, например:

```
fload_0
aload_1
```



Команды сохранения значения вершины стека в локальной переменной (33 шт.)

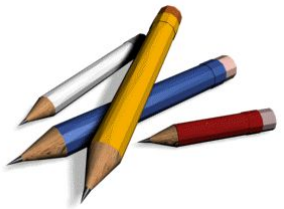
извлекают значение из вершины стека операндов и сохраняют их в списке локальных переменных

```
типstore индекс
```

Примеры.

istore 5 – сохранить вершину стека в пятой локальной переменной (целочисленной)

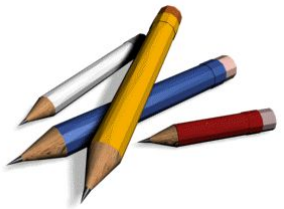
dstore 10 – сохранить вершину стека в десятой локальной переменной (вещественной)



Особенности команд сохранения переменных:

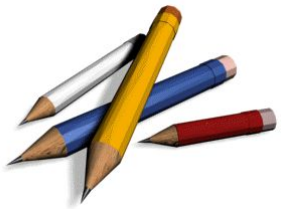
Для часто используемых индексов (0, 1, 2, 3) имеются сокращенные варианты команд, занимающие один байт, например:

```
fstore_0
astore_1
```



Прочие команды работы со стеком (9 шт.)

К этой группе относятся команды, которые удаляют (**pop**, **pop2**), дублируют (**dup, dup2**), меняют местами верхние элементы стека операндов (**swap**), а также выполняют другие, более сложные манипуляции со стеком.



Арифметические команды (24 шт.)

Все арифметические команды извлекают из стека два операнда, выполняют арифметическое действие и в стек заносят получившийся результат.

типдействие

Примеры.

iadd – сложение целых чисел

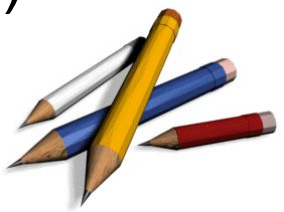
lsub – вычитание длинных целых чисел

fmul – умножение дробных чисел (одинарной точности)

ddiv – деление дробных чисел (двойной точности)

irem – остаток от деления целых чисел

fneg – изменение знака дробного числа



Логические команды (12 шт.)

Логические команды выполняют действия с целыми (длинными целыми) числами: арифметический сдвиг, "и", "или", "исключающее или"

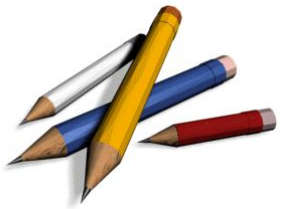
тип действие

Примеры.

iand – логическое "и" с двумя целыми числами

lor – логическое "или" с двумя длинными целыми числами

Логические команды также используются для вычисления истинности булевских выражений.



Команды преобразования типа (15 шт.)

Используются для преобразования значений базовых типов

тип2тип

Примеры.

i2f – преобразование целого числа в дробное (одинарной точности)

d2f – преобразование дробного числа (двойной точности) в дробное число (одинарной точности)

В некоторых случаях требуется прибегнуть к цепочке преобразований, например: *d2i i2s* (double → short)

Команды для работы с массивами

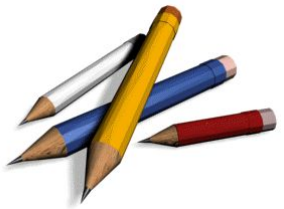
Используются:

– для создания массивов требуемого типа;
– для вычисления длины существующего массива;
– для загрузки элемента массива в стек;
– для сохранения вершины стека в элементе массива

Примеры.

newarray T_INT – создание целочисленного массива
iaload – загрузка целого числа из массива в стек
iastore – сохранение целого числа в массиве

При обращении к элементам массива в стек должен быть загружен его индекс

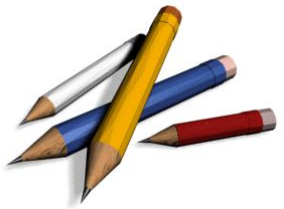


Команды передачи управления

Используются:

- для выполнения условного перехода;
- для выполнения безусловного перехода (**goto**);
- для организации выбора (switch);
- для вызова и возврата из подпрограмм;
- для вызова и возврата из методов

Команды условного и безусловного переходов используются для организации базовых алгоритмических структур (ветвление, повторение).



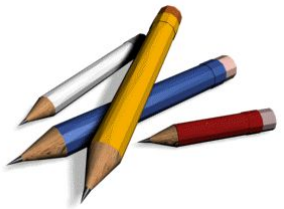
Команды условного перехода (вариант 1, 8 шт.)

```
if условие адрес
```

Извлекается значение из вершины стека. Если условие истинно, осуществляется переход по указанному адресу, например:

ifeq – переход, если значение равно 0

ifgt – переход, если значение больше 0



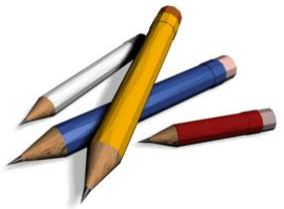
Команды условного перехода (вариант 2, 8 шт.)

```
if_типcmpусловие адрес
```

Из стека извлекаются два значения и сравниваются между собой. Если условие истинно, осуществляется переход по указанному адресу, например:

if_icmpeq – переход, если значения равны

if_icmpne – переход, если значения не равны



Команды вызова методов

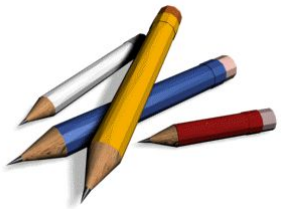
выполнение команд связано не только с передачей управления, но и с анализом разного рода таблиц.

invokevirtual – вызывает (виртуальный) метод на основе анализа информации времени выполнения;

invokenonvirtual – осуществляет вызов на основе информации времени компиляции (например, вызов метода родительского класса);

invokestatic – вызывает статический метод класса;

invokeinterface – вызывает метод, предоставленный интерфейсом.

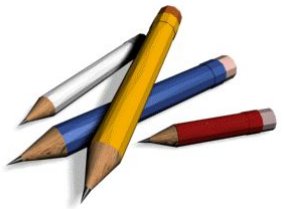


Команды манипулирования с полями объектов команды позволяют установить/прочитать обычное/статическое поле объекта:

getfield *putfield*

getstatic *putstatic*

Прочие объектные операции
создать объект, проверить тип объекта

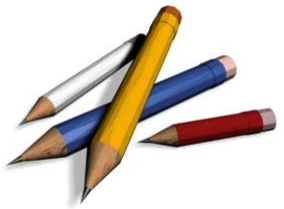


Команда возбуждения исключительной ситуации

athrow – позволяет выбросить исключительную ситуацию.

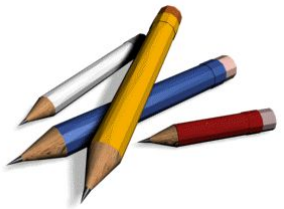
Команды синхронизации

используются для организации работы параллельных процессов (войти в критическую секцию, выйти из него).



Реализация арифметических операций

Машинный код, генерируемый компилятором Java, позволяет вычислять значения арифметических выражений на основе ПОЛИЗ.

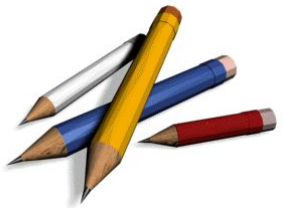


Пример. Вычисление выражения S = a * b / 2

ПОЛИЗ выражения: a b * 2 /

Алгоритм вычисления по ПОЛИЗ:

1. Загрузить в стек значение переменной *a*
2. Загрузить в стек значение переменной *b*
3. Выполнить умножение
4. Загрузить в стек константу *2*
5. Выполнить деление
6. Сохранить результат

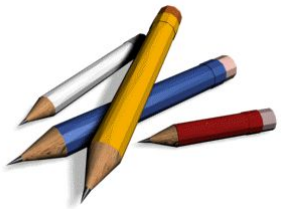


На языке Java:

```
float a,b,S;
S = a * b / 2;
```

Машинный код JVM:

fload_1 *// загрузить в стек значение первой локальной переменной*
fload_2 *// загрузить в стек вторую локальную переменную*
fmul *// выполнить умножение*
fconst_2 *// загрузить в стек константу 2*
fdiv *// выполнить деление*
fstore_3 *// результат загрузить в третью локальную переменную*

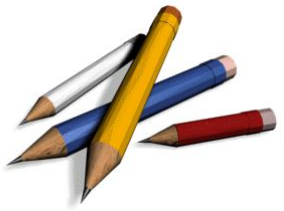


Пример 1. Реализация оператора ветвления

```
if (выражение) оператор_1; else оператор_2;
```

Машинный код:

вычислить выражение
ifne *оператор_2*
оператор_1: …
…
goto *конец*
оператор_2: …
…
конец:

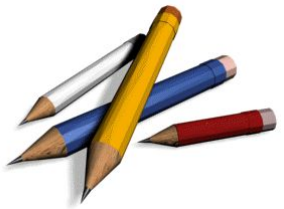


Пример 2. Реализация оператора повторения

```
while (выражение) оператор;
```

Машинный код:

цикл: вычислить выражение
ifne *конец*
оператор
goto *цикл*
конец:

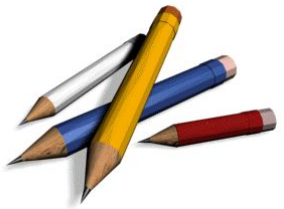


Пример 3. Реализация оператора повторения

```
do оператор; while (выражение);
```

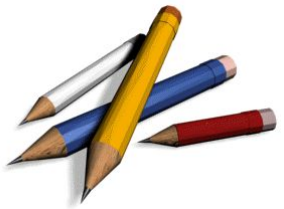
Машинный код:

цикл: оператор
вычислить выражение
ifne *цикл*
конец:



Алгоритм объектно-ориентированной реализации стековой виртуальной машины

Использование объектно-ориентированного подхода позволяет значительно упростить процесс реализации стековой виртуальной машины.

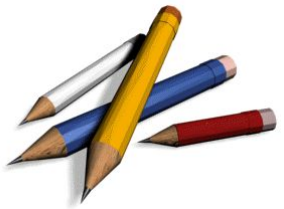


Основные классы, используемые при реализации стековой ВМ

StackVM
стековая виртуальная машина

VariableStorage
хранилище переменных

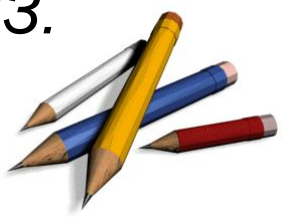
Instruction
базовый класс для команд виртуальной машины



Класс StackVM

```
public class StackVM {

    private VariableStorage storage;
    private Stack<Double> stack;

    public StackVM(VariableStorage storage) {
        this.storage = storage;
        stack=new Stack<Double>();
    }
```

Экземпляр виртуальной машины использует внешнее (глобальное) хранилище переменных и локальный стек для реализации алгоритма вычисления по ПОЛИЗ.



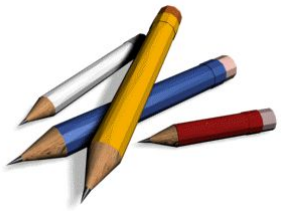
Виртуальные машины

Вычисление арифметического выражения сводится к последовательному выполнению отдельных команд виртуальной машины.

```
public double execute(Instruction[] program) {
    stack.clear();
    for(Instruction instruction:program) {
        instruction.execute(stack,storage);
    }
    return stack.peek();
}
```

Результат вычислений извлекается из вершины стека.

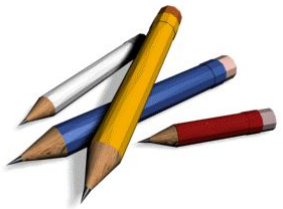
```
public double getLastValue() {
    return stack.peek();
}
```



Виртуальные машины

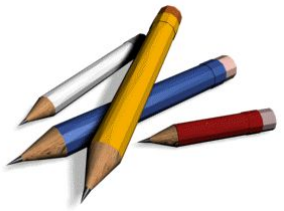
Все команды виртуальной машины реализуются как классы-потомки одного абстрактного класса (интерфейса).

```
public interface Instruction {
    public void execute(Stack<Double> stack, VariableStorage storage);
}
```



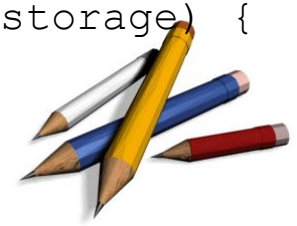
Реализация арифметических команд выполняется тривиально.

```
public class Add implements Instruction {
    public void execute(Stack<Double> stack, VariableStorage storage) {
        double op2=stack.pop();
        double op1=stack.pop();
        stack.push(op1+op2);
    }
}
```



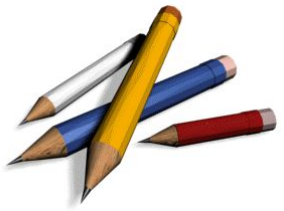
Реализация команд загрузки в стек значений переменных и констант.

```
public class Variable implements Instruction {
    private String variable;
    public Variable(String variable) {
        this.variable = variable;
    }
    public void execute(Stack<Double> stack, VariableStorage storage) {
        stack.push(storage.getVariable(variable));
    }
}
public class Constant implements Instruction {
    private double value;
    public Constant(double value) {
        this.value = value;
    }
    public void execute(Stack<Double> stack, VariableStorage storage) {
        stack.push(value);
    }
}
```



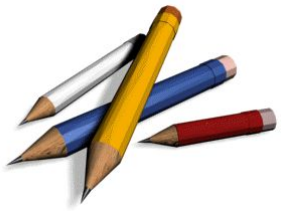
Реализация команды присваивания.

```
public class Assign implements Instruction {
    private String variable;
    public Assign(String variable) {
        this.variable = variable;
    }
    public void execute(Stack<Double> stack, VariableStorage storage) {
        storage.putVariable(variable, stack.peek());
    }
}
```



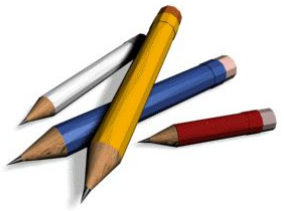
Пример. Вычисление дискриминанта

```
Instruction[] expr1={ new Constant(1),  new Assign("a") }; // a=1;
Instruction[] expr2={ new Constant(-5), new Assign("b") }; // b=-5;
Instruction[] expr3={ new Constant(6),  new Assign("c") }; // c=6;
Instruction[] expr4={  // d=b*b-4*a*c;
        new Variable("b"),
        new Variable("b"),
        new Mul(),
        new Constant(4),
        new Variable("a"),
        new Mul(),
        new Variable("c"),
        new Mul(),
        new Sub(),
        new Assign("d")
        };
```



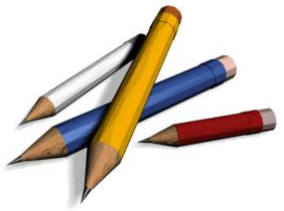
Пример. Вычисление дискриминанта

```
Instruction[] expr1={ new Constant(1),  new Assign("a") }; // a=1;
Instruction[] expr2={ new Constant(-5), new Assign("b") }; // b=-5;
Instruction[] expr3={ new Constant(6),  new Assign("c") }; // c=6;
Instruction[] expr4={  // d=b*b-4*a*c;
        new Variable("b"),
        new Variable("b"),
        new Mul(),
        new Constant(4),
        new Variable("a"),
        new Mul(),
        new Variable("c"),
        new Mul(),
        new Sub(),
        new Assign("d")
        };
```



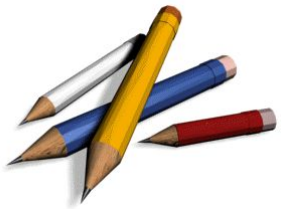
Виртуальные машины

```
VariableStorage storage = new VariableStorage();
StackVM stackVM = new StackVM(storage);

stackVM.execute(expr1); // a=1;
stackVM.execute(expr2); // b=-5;
stackVM.execute(expr3); // c=6;
stackVM.execute(expr4); // d=b*b-4*a*c;

System.out.println("Ответ: " + stackVM.getLastValue());
```

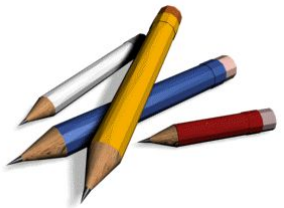


Алгоритм объектно-ориентированной реализации виртуальной машины, интерпретирующей синтаксическое дерево программы

При разнообразной обработке структур данных, содержащих объекты различных классов, оказывается удобным следовать *паттерну проектирования Visitor ("Посетитель")*



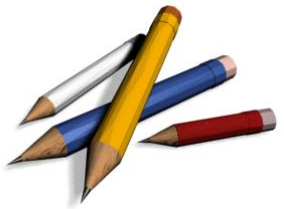
Под *паттерном проектирования* обычно понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.



Паттерн проектирования Visitor ("Посетитель")

Позволяет применять различные операции к каждому объекту из некоторой структуры данных.

Вместо того, чтобы реализовывать каждую операцию в каждом классе, все однотипные операции выносятся в отдельные классы.

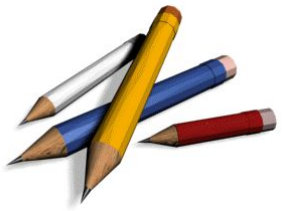


Было

```
class A {
   операция1();
   операция2();}
}
class B {
   операция1();}
   операция2();
}
```

Стало

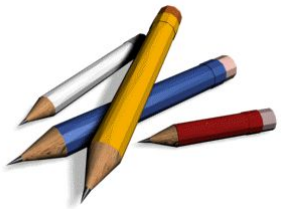
```
class A {
   принятьПосетителя();
}
class B {
   принятьПосетителя();
}
class Посетитель1 {
   операцияНадА();
   операцияНадВ();
}
class Посетитель2 {
   операцияНадА();
   операцияНадВ();
}
```



Преимущества, получаемые в результате реализации паттерна Visitor ("Посетитель")

1. Однотипные операции над различными объектами локализуются в одном классе, а не "размазываются" по нескольким классам.

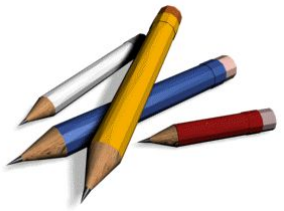
2. Легко добавлять новые операции над объектами, не изменяя их, а добавляя новые классы-посетители.



Особенности реализации паттерна Visitor ("Посетитель")

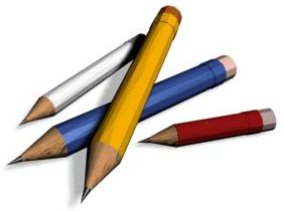
1. Все классы-посетители являются предками абстрактного класса, в котором для каждого типа объекта объявлен свой метод вида:

```
class АбстрактныйПосетитель {
    операцияНадА(ТипА объект);
    операцияНадВ(ТипВ объект);
    . . .
}
```



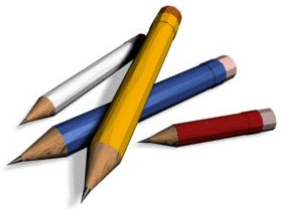
2. В каждом классе-элементе структуры данных объявляется метод, обращающийся к классу-посетителю:

```
class A {
    принятьПосетителя(АбстрактныйПосетитель п) {
        п.операцияНадA(this);
    }
}

class B {
    принятьПосетителя(АбстрактныйПосетитель п) {
        п.операцияНадB(this);
    }
}
```



3. Описываются классы-посетители, выполняющие требуемую обработку объектов структуры.

4. Пользователь паттерна должен
   а) сформировать требуемую структуру данных;
   б) выполнить полный обход структуры данных и для каждого элемента структуры вызвать метод "принятьПосетителя".



Виртуальные машины

При реализации виртуальной машины также могут быть использованы другие паттерны проектирования, например, Interpreptator ("Интерпретатор")