ВВЕДЕНИЕ В КЛАССЫ JAVA. МЕТОДЫ И КЛАССЫ

Особенности реализации основных фундаментальных свойств ООП в Java

- **Абстракция**
- Описывая поведение какого-либо объекта, например автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи.
- Мы должны абстрагироваться от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Абстракция

- Описание каждой модели производится в виде одного или нескольких классов (classes).
- Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты.
- Класс содержит описание переменных и констант, характеризующих объект. Они называются полями класса (class fields).
- Процедуры, описывающие поведение объекта, называются методами класса (class methods).
- Внутри класса можно описать и вложенные классы (nested classes) и вложенные интерфейсы.
- Поля, методы и вложенные классы первого уровня являются членами класса (class members).

Абстракция

- Вот набросок описания автомобиля:

```
class Automobile{
int maxVelocity;  // Поле, содержащее наибольшую скорость автомобиля
int speed;        // Поле, содержащее текущую скорость автомобиля
int weight;       // Поле, содержащее вес автомобиля
                  // Прочие поля...
void moveTo(int x, int y){   // Метод, моделирующий перемещение
                             // автомобиля. Параметры x и y — не поля
int a = 1;   // Локальная переменная — не поле
             // Тело метода. Здесь описывается закон
             // перемещения автомобиля в точку (x, y)
}
// Прочие методы. . .
}
```

Абстракция

- После того как описание класса закончено, можно создавать конкретные объекты, экземпляры (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них.

```
Automobile lada2110, fordScorpio, oka;
```

- Затем операцией new определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

Абстракция

- На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции new повторяется имя класса со скобками Automobile () .
- Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2110.maxVelocity = 150;
fordSc
orpio.maxVelocity = 180;
oka.maxVelocity = 350;    // Почему бы и нет?
oka.moveTo(35, 120);
```

- Текстовая строка в кавычках понимается в Java как объект класса String . Поэтому можно написать

```
int strlen = "Это объект класса String".length();
```

Абстракция

- Объект "строка" выполняет метод length() , один из методов своего класса string , подсчитывающий число символов в строке. В результате получаем значение strlen , равное 24. Подобная странная запись встречается в программах на Java на каждом шагу.
- При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Введение в классы

- В основе языка Java лежит класс.
- *Класс* — это логическая конструкция, на которой построен весь язык Java, потому что такая конструкция определяет форму и природу объекта.
- Класс формирует также основу для объектно-ориентированного программирования в Java.

Основы классов

Наиболее важным в понятии класса является то, что он определяет новый тип данных.

После определения новый тип можно использовать для создания *объектов* этого типа.

Таким образом, класс — это *шаблон для* объекта, а объект — это *экземпляр* класса.

Поскольку объект — экземпляр класса, два слова *объект* и *экземпляр* часто будут использоваться как взаимозаменяемые.

Общая форма класса

class *classname* {
type instance-variable1;
type instance-variable2;
…
type instance-variableN;
type methodname1(parameter-list) {
// тело метода }
type methodname2 (parameter-list) {
// тело метода }
…
type methodnameN(parameter-list) {
// тело метода > }

Общая форма класса

Данные или переменные, определенные в классе, называются *переменными экземпляра* или *экземплярными переменными* (instance variables).

Код содержится внутри *методов* (methods).

Все вместе, методы и переменные, определенные внутри класса, называются *членами класса* (class members).

Таким образом, именно методы определяют, как могут использоваться данные класса.

Простой класс

- Предположим, имеется класс с именем Box, который определяет три переменных экземпляра: width, height и depth.

```
class Box {
double width;
double height;
double depth;
}
```

- Класс определяет новый тип данных. В этом случае новый тип данных называется Box. Будем использовать это имя для объявления объектов типа box. Объявление класса создает только шаблон, а не фактический объект.

Простой класс

- Чтобы фактически создать Box-объект, можно воспользоваться следующим утверждением:

```
Box mybox = new Box ();    // создать Box-объект с
                           //именем mybox
```

- После выполнения этого утверждения переменная mybox станет экземпляром класса Box, становясь той самой "физической" реальностью.
- Каждый Box-объект будет содержать свою собственную копию переменных width, height и depth. Для доступа к этим переменным необходимо использовать операцию "точка" (.)

```
mybox.width =  100;
```

Использование простого класса

```
/* Программа, которая использует Box-класс.
Назовите этот файл BoxDemo.java */
class Box {
double width;
double height;
double depth; }
// Этот класс объявляет объект типа Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box ();
double vol;
// присвоить значения экземплярным переменным объекта
mybox mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// вычислить объем блока
vol = mybox.width * mybox.height * mybox.depth;
System.out.printlnt"Объем равен " + vol); } }
```

Объявление объектов

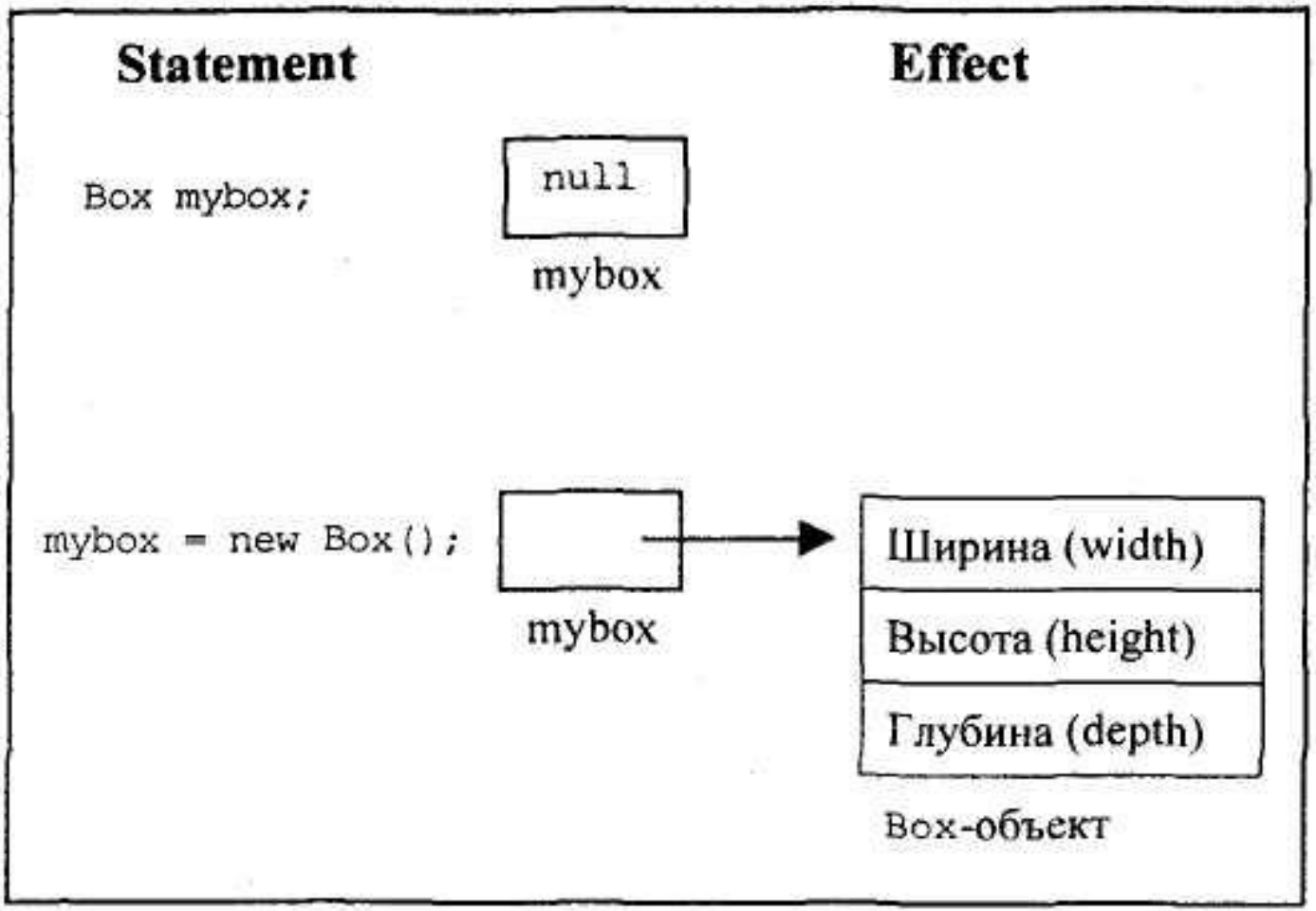
Box mybox = new Box();

- Этот оператор комбинирует два шага.

```
Box mybox;     //  объявить  ссылку на  объект
mybox = new Box();// распределить память для Box-объекта
```

- Первая строка объявляет mybox как ссылку на объект типа Box. После того как эта строка выполняется, mybox содержит значение null, которое означает, что переменная еще не указывает на фактический объект. Любая попытка использовать mybox в этой точке приведет к ошибке во время компиляции. Следующая строка распределяет фактический объект и назначает ссылку на него переменной mybox. После того, как вторая строка выполняется, вы можете использовать mybox, как если бы это был объект Box.

Объявление объекта



Операция *new*

- Операция new динамически распределяет память для объекта.

class-var = new *classname*();

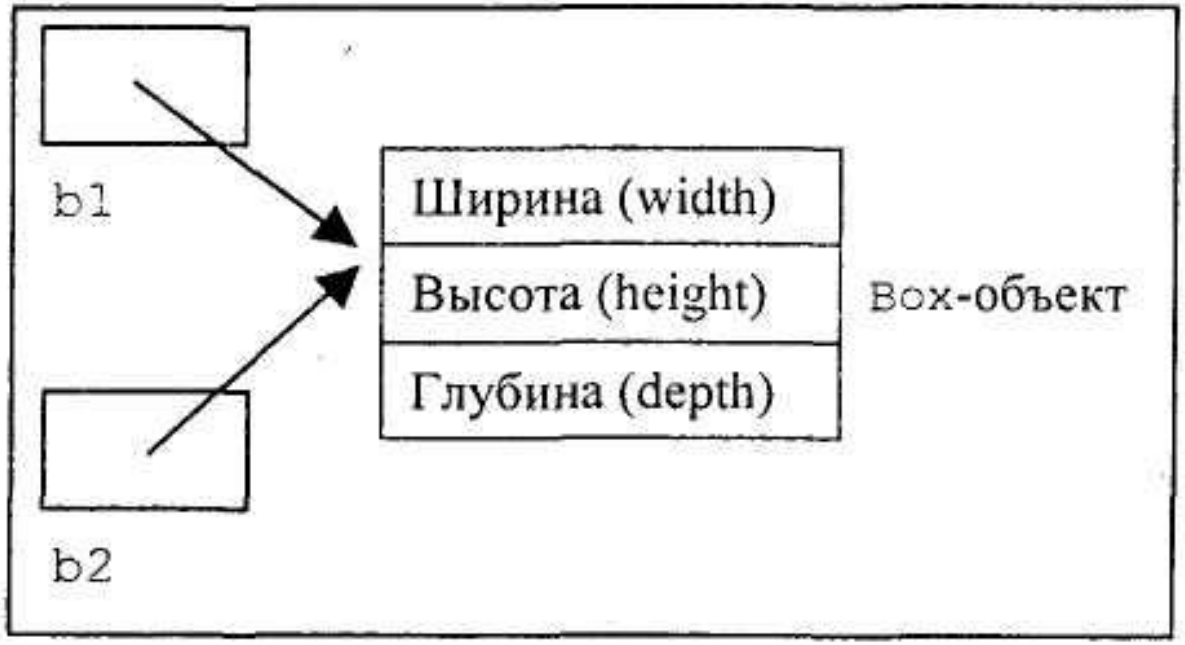
- Здесь *class-var* — переменная типа "класс", которая создается; *classname* — имя класса, экземпляр которого создается.
- За именем класса следуют круглые скобки, устанавливающие *конструктор* класса. Конструктор (constructor) определяет, что происходит, когда объект класса создается. Конструкторы имеют много существенных атрибутов.

Операция *new*

- Большинство классов явно определяют свои собственные конструкторы (внутри своих определений). Однако если явный конструктор не определен, то Java автоматически обеспечит так называемый "конструктор по умолчанию" (default constructor) или *умалчиваемый конструктор.*

Ссылочные переменные объекта

```
Box b1  = new Box();
Box b2  =b1;
```



Методы класса

- Общая форма метода такова:

```
type name (parameter-list)    {
// тело метода
}
```

- Здесь *type* определяет тип данных, возвращаемых методом. Это может быть любой допустимый тип, включая типы классов, которые вы создаете.
- *Parameter-list* — это последовательность пар тип-идентификатор, разделенных запятыми.
- Параметры —переменные, которые принимают значения *аргументов,* посылаемых методу во время его вызова.
- Методы, у которых тип возвращаемого значения отличен от void, возвращают значение вызывающей подпрограмме:

```
return value;
```

- Здесь *value* — возвращаемое значение.

Добавление метода к классу

```
// Эта программа включает метод
    внутрь класса Box.
class Box {
double width;
double height;
double depth;
// показать объем блока
void volume() {
System.out.print("Объем равен ");
System.out.println(width * height *
    depth); } }

class BoxDemo   (
public static void main(String args[])
    {
Box mybox1 = new Box () ;
Box mybox2 = new Box();
// присвоить значения переменным
    экземпляра
myboxl myboxl.width = 10;
myboxl.height = 20;
myboxl.depth = 15;
/* присвоить другие значения
переменным экземпляра mybox2 */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
// показать объем первого блока
    mybox1.volume{);
// показать объем второго блока
    mybox2.volume(); } }
```

Возврат значений

```
// вычислить и возвратить объем
double volume() {
return width * height * depth; } }
…
// получить объем первого блока
vol = mybox1.volume();
System.out.println("Объем равен " + vol);
// получить объем второго блока
vol = mybox2.volume();
System.out.println("Объем равен " + vol);
```

Параметризация методов

```
// установить размеры блока
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d; } }
…
// инициализировать каждый блок
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
```

Конструкторы

- *Конструктор* инициализирует объект после его создания. Он имеет такое же имя, как класс, в котором он постоянно находится и синтаксически подобен методу.
- Если конструктор определен, то он автоматически вызывается сразу же после того, как объект создается, и прежде, чем завершается выполнение операции new.

Конструкторы

- Конструкторы не имеют ни спецификатора возвращаемого типа, ни даже спецификатора void.
- Работа конструктора заключается в том, чтобы инициализировать внутреннее состояние объекта так, что код, создающий экземпляр, будет полностью инициализирован и пригоден для немедленного использования объекта.

Конструкторы

- Конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор return , но только пустой, без всякого возвращаемого значения.
- В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

Пример конструктора

```
class Box {
double width;
double height;
double depth;
// Это конструктор класса Box.
Box() {
System.out.println("Создание Box");
width = 10;
height = 10;
depth = 10; }
// вычислить и возвратить объем
double volume () {
return width * height * depth; } }
```

Параметризованные конструкторы

```
class Box {
double width; double height; double depth;
// Это конструктор класса Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d; }
// вычислить и возвратить объем double volume() {
return width * height * depth; }
}
```

Использование ключевого слова this

- Иногда у метода возникает необходимость обращаться к объекту, который его вызвал.
- Для этого Java определяет ключевое слово this. Его можно использовать внутри любого метода, чтобы сослаться на *текущий* объект.
- this — это всегда ссылка на объект, метод которого был вызван.
- Можно использовать this везде, где разрешается ссылка на объект текущего класса.

Использование ключевого слова this

```
//    Избыточное использование this.
Box(double w,   double h,   double d)   {
this.width = w;
this.height = h;
this.depth = d; }
```

- Использование this избыточно, но совершенно корректно. Внутри Box() this будет всегда ссылаться на вызывающий объект.

Скрытие переменной экземпляра

- В Java недопустимо объявление двух локальных переменных с одним и тем же именем внутри той же самой или включающей области действия идентификаторов.
- Однако, когда локальная переменная имеет такое же имя, как переменная экземпляра, локальная переменная *скрывает* переменную экземпляра.
- Вот почему width, height и depth не использовались как имена параметров конструктора Box() внутри класса Box.
- Если бы они были использованы для именования этих параметров, то, скажем width, как формальный параметр, скрыл бы переменную экземпляра width.

Скрытие переменной экземпляра

- Поскольку this позволяет обращаться прямо к объекту, это можно применять для разрешения любых конфликтов пространства имен, которые могли бы происходить между экземплярными и локальными переменными.

Скрытие переменной экземпляра

```
//Используйте этот вариант конструктора
// для разрешения конфликтов пространства имен.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth; }
```

Сборка «мусора»

- Так как объекты распределяются динамически с помощью операции new, можно задать вопрос, как такие объекты ликвидируются и их память освобождается для более позднего перераспределения.
- Java использует подход: освобождение памяти от объекта происходит автоматически.
- Методика, которая реализует эту процедуру, называется сборкой "мусора".

Сборка «мусора»

- Она работает примерно так: когда никаких ссылок на объект не существует, предполагается, что этот объект больше не нужен, и память, занятая объектом, может быть освобождена.
- Сборка "мусора" происходит не регулярно (если вообще происходит) во время выполнения программы.
- Кроме того, различные реализации исполняющей системы Java имеют разные подходы к сборке "мусора".

Метод *finalized*

- Иногда объекту нужно выполнять некоторые действия, когда он разрушается. Например, если объект содержит некоторый не-Java ресурс, такой как дескриптор файла или оконный шрифт, то нужно удостовериться, что до разрушения объекта эти ресурсы освобождаются.
- Для обработки таких ситуаций Java использует механизм, называемый *завершением* (finalization).
- Чтобы добавить завершение к классу, вы просто определяете метод finalize (). Исполняющая система Java вызывает этот метод всякий раз, когда она собирается ликвидировать объект данного класса. Внутри метода finalize о нужно определить те действия, которые должны быть выполнены прежде, чем объект будет разрушен.

Метод *finalized*

- Сборщик мусора отрабатывает периодически, проверяя объекты, на которые нет больше ссылок ни из выполняющихся процессов, ни косвенных — из других действующих объектов.
- Непосредственно перед освобождением всех активов исполняющая система Java вызывает для объекта метод finalize ().

```
protected void finalize()
{
// код завершения
}
```

- Здесь ключевое слово protected — спецификатор, который запрещает доступ к finalize о кодам, определенным вне этого класса.
- finalize() вызывается только перед самой сборкой "мусора". Он не запускается, когда объект выходит из области действия идентификаторов.

Перегрузка методов

- В языке Java в пределах одного класса можно определить два или более методов, которые совместно используют одно и то же имя, но имеют разное количество параметров. Когда это имеет место, методы называют *перегруженными,* а о процессе говорят как о *перегрузке метода.*

Перегрузка методов

- Чтобы определить при вызове, какую версию перегруженного метода в действительности вызывать, Java руководствуется типом и/или числом его параметров. Таким образом, перегруженные методы должны отличаться по типу и/или числу их параметров.

Перегрузка методов

```
class OverloadDemo  {
void test ()   { *
System.out.println("Параметры отсутствуют");
}
// Перегруженный метод test с одним int-параметром.
void test(int a) {
System.out.println("a: " + a);
}
// Перегруженный метод test с двумя int-параметрами.
void test(int a, int b) {
System.out.println("a и b: " + a + " " + b) ; }
// Перегруженный метод test с double-параметром,
double test(double a) {
System.out.println("Вещественное двойной точности a: " + a) return a*a;  }
}
class Overload {
public static void main(Strlng args[]) { OverloadDemo ob = new OverloadDemo(); double result;
// вызвать все версии test()
ob.test() ; ob.test(10);ob.test(10, 20);
result = ob.test(123.2);
System.out.println("Результат ob.test(123.2) : " + result)
} }
```

Перегрузка конструкторов

```
class Box { double width; double height; double
      depth;
// конструктор для инициализации размеров
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// конструктор без указания размеров
Box ()   {
width = -1; // использовать —1 для указания
height = -1;    // не инициализированного
depth = -1;     // блока
}
// конструктор для создания куба
Box(double len) {
width = height = depth = len; }
// вычислить и возвратить объем double
      volume() {
return width * height * depth; } }
```

```
class OverloadCons {
public static void main(String args[]) {
// создать блоки, используя различные
      конструкторы
Box myboxl = new Box(10, 20, 15);
Box mybox2 « new Box();
Box mycube = new Box(7);
double vol;
// получить объем первого блока
yol = myboxl.volume() ;
System.out.println("Объем myboxl равен " +
      vol);
// получить объем второго блока
vol = mybox2.volume();
System.out.println("Объем mybox2 равен " +
      vol);
// получить объем куба
vol = mycube.volume();
System.out.println("Объем mycube равен " +
      vol);
```

Передача методов в качестве параметра

- Существует практика передачи методам объектов. Например, рассмотрим следующую простую программу:

```
// Объекты можно передавать методам в качестве параметров,
class Test { int a, b;
Test(int i, int j)
{ a = i; b = j;
}
// возвратить true, если о равно вызывающему объекту
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false; } }
class PassOb {
public static void main(String args[]
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1  == ob2:     "  +  ob1.equals(ob2));
System.out.println("ob1  == ob3:     "  + ob1.equals (ob3)); } }
```

Рекурсия

- Java поддерживает *рекурсию.* Рекурсия — это процесс определения чего-то в терминах самого себя. Рекурсия — это атрибут, который позволяет методу вызвать самого себя.
- Классический пример рекурсии — вычисление факториала числа.

```
class Factorial {
// это рекурсивная функция
int fact(int n)
{ int result;
if(n==l) return 1; result = fact(n-l) * n; return result; }
}
class Recursion {
public static void main(String args[])
{ Factorial f = new Factorial();
System.out.println("Факториал 3 равен " + f.fact(3));
System.out.println("Факториал 4 равен " + f.fact(4));
System.out.println("Факториал 5 равен " + f.fact (5)); } }
```

Управление доступом

- Инкапсуляция связывает данные с кодом, который манипулирует ими и обеспечивает другой важный атрибут: *управление доступом.*
- Способ доступа к элементу определяет *спецификатор доступа,* который модифицирует его объявление.
- Спецификаторы доступа Java: public (общий), private (частный) и protected (защищенный). Java также определяет уровень доступа, заданный по умолчанию (default access level). Спецификатор protected применяется только при использовании наследования.

Управление доступом

```
/* Эта программа демонстрирует различие между методами доступа public и private.
*/
class Test {
int a;// доступ по умолчанию (public)
public int b;    // общий (public) доступ
private int c;  // частный (private) доступ
// методы для доступа к переменной c
void setc(int i) {        // установить значение c
c = i;
        }
        int getc(){        // получить значение c
return c;
}}
class AccessTest {
public static void main(String args[]) { Test ob = new Test ();
// OK, к переменным a и b возможен прямой доступ.
ob.a = 10; ob.b = 20;
// Не OK и вызовет ошибку.
        // ob.c = 100;    //Ошибка!
// Нужен доступ к c через ее методы.
        ob.setc(100)     // OK
System.out.println("a, b, и c: " + ob.a + " " + ob.b + " " + ob.getc(); } }
```

Вложенные и внутренние классы

- Существует возможность определения одного класса внутри другого. Такие классы известны как *вложенные* (nested) *классы.* Область видимости вложенного класса ограничивается областью видимости включающего класса.
- Существует два типа вложенных классов: *статические* и *нестатические.* Статический вложенный класс — это класс, который имеет модификатор static.
- Наиболее важный тип вложенного класса — внутренний (inner) класс. *Внутренний класс* — это нестатический вложенный класс, имеющий доступ ко всем переменным и методам своего внешнего класса и возможность обращаться к ним напрямую, таким же способом, как это делают другие нестатические члены внешнего класса. Итак, внутренний класс находится полностью в пределах видимости своего включающего класса.

Вложенные и внутренние классы

```
// Демонстрирует внутренний класс.
class Outer {
int outer_x = 100;
void test () {
Inner inner = new Inner ();
inner.display(); }
// это внутренний класс
class Inner {
void display() {
System.out.println("В методе display: outer_x = " + outer_x); } } }
class InnerClassDemo {
public static void main(String args[])
{ Outer outer = new Outer ();
outer. test (); } }
```

Класс Stack

- *Стек* хранит данные, используя очередь типа LIFO ("Last-In, First-Out") — последним вошел, первым вышел.
- Стеки управляются через две операции, традиционно называемые *push* (поместить) и *pop* (извлечь, вытолкнуть).
- Чтобы поместить элемент в вершину стека, нужно использовать операцию *push.* Чтобы извлечь элемент из стека, нужно использовать операцию *pop.*

Класс Stack

- push (Object item) —помещает элемент item в стек;
- pop () — извлекает верхний элемент из стека;
- peek () — читает верхний элемент, не извлекая его из стека;
- empty () — проверяет, не пуст ли стек;
- search (object item) — находит позицию элемента item в стеке. Верхний элемент имеет позицию 1, под ним элемент 2 и т. д. Если элемент не найден, возвращается — 1.

Проверка парности скобок (пример Stack)

```
import java.util.*;
class StackTest1
static boolean checkParity(String
     expression,
    String open, String close){
   Stack stack = new Stack ();
   StringTokenizer st = new
   StringTokenizer(expression,
     " \t\n\r+*/-(){}", true);
   while (st.hasMoreTokens ()) {
     String tmp = st.nextToken();
```

```
if (tmp.equals(open)) ,
     stack.push(open);
          if (tmp.equals(close))
     stack.pop();
     }
     if (stack.isEmpty ()) return
     true
return false;
}
public static void main(String[]
     args){
  System.out.println(
    checkParity("a - (b - (c - a) /
     (b + c) - 2)" , "(", ")"));
 }
}
```

Работа со строками

- В объектах класса string хранятся строки-константы неизменной длины и содержания. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее.
- Длину строк, хранящихся в объектах класса stringBuffer , можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку. Во многих случаях, когда надо изменить длину строки типа string , компилятор Java неявно преобразует ее к типу stringBuffer , меняет длину, потом преобразует обратно в тип string .

Работа со строками

String s = "Это" + " одна " + "строка";

- компилятор выполнит так:

```
String s = new StringBuffer().append("Это").append(" одна ")
    .append("строка").toString();
```

- Будет создан объект класса stringBuffer , в него последовательно добавлены строки "Это", " одна ", "строка", и получившийся объект класса StringBuffer будет приведен к типу String методом toString () .
- Символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа char .

Класс String

Класс string предоставляет девять конструкторов:

- string() — создается объект с пустой строкой;
- string (String str) — из одного объекта создается другой;
- string (StringBuf fer str) — преобразованная копия объекта класса BufferString;
- string(byte[] byteArray) — объект создается из массива байтов byteArray;

Класс String

- String (char [] charArray) — объект создается из массива charArray символов Unicode;
- String (byte [] byteArray, int offset, int count) — объект создается из части массива байтов byteArray, начинающейся с индекса offset и содержащей count байтов;
- String (char [] charArray, int offset, int count) — то же, но массив состоит из символов Unicode;
- String(byte[] byteArray, String encoding) — символы, записанные в массиве байтов, задаются в Unicode-строке, с учетом кодировки encoding ;
- String(byte[] byteArray, int offset, int count, String encoding) — то же самое, но только для части массива.

Класс String

- Конструкторы, использующие массив байтов byteArray , предназначены для создания Unicode-строки из массива байтовых ASCII-кодировок символов. Такая ситуация возникает при чтении ASCII-файлов, извлечении информации из базы данных или при передаче информации по сети.
- В самом простом случае компилятор для получения двухбайтовых символов Unicode добавит к каждому байту старший нулевой байт. Получится диапазон ' \u0000 ' — ' \u00ff ' кодировки Unicode, соответствующий кодам Latin 1.

Класс String

- Тексты на кириллице будут выведены неправильно.
  - Если же на компьютере сделаны местные установки, (в MS Windows это выполняется утилитой Regional Options в окне Control Panel), то компилятор, прочитав эти установки, создаст символы Unicode, соответствующие местной кодовой странице. В русифицированном варианте MS Windows это обычно кодовая страница CP1251.
  - Если исходный массив с кириллическим ASCII-текстом был в кодировке CP1251, то строка Java будет создана правильно. Кириллица попадет в свой диапазон '\u0400'—'\u04FF' кодировки Unicode.

Класс String

- Но у кириллицы есть еще, по меньшей мере, четыре кодировки.
  - В MS-DOS применяется кодировка CP866.
  - В UNIX обычно применяется кодировка KOI8-R.
  - На компьютерах Apple Macintosh используется кодировка MacCyrillic.
  - Есть еще и международная кодировка кириллицы ISO8859-5;
- Например, байт 11100011 (0xE3 в шестнадцатеричной форме) в кодировке CP1251 представляет кириллическую букву Г , в кодировке CP866 — букву У , в кодировке KOI8-R — букву Ц , в ISO8859-5 — букву у , в MacCyrillic — букву г .

Спасибо за внимание!