**Заметки по Java**

**По книге «Философия Java»**

**2 глава. Все является объектом**

Java основан на С++, но является более ОО языком, относится к семейству смешанных языков. Java подразумевает, что вы занимаетесь только ООП.

В Java почти все является объектом.

**Для работы с объектами используются ссылки.**

В Java вы обращаетесь с любыми данными как с объектом, и поэтому повсюду используется единый последовательный синтаксис. Хотя вы обращаетесь со всеми данными как с объектом, идентификатор, которым вы манипулируете, на самом деле представляет собой ссылку на объект. Например, для хранения слова или предложения создается ссылка String:

String s;

Однако здесь определяется только ссылка, но не сам объект. Нужно инициализировать ссылку при ее создании:

String s = “abs”;

**Все объекты должны создаваться явно**

Когда определяется ссылка, то желательно присоединить ее к новому объекту (при помощи ключевого слова new).

String s = new String(“abs”);

**Где хранятся данные**

5 разных мест для хранения:

* Регистры (Данные хранятся прямо внутри процессора. У вас нет прямого доступа к регистрам.)
* Стек (Находится в общей оперативной памяти (RAM). Ссылки на объекты хранятся в стеке, но сами объекты не помещаются в стек. Быстрый способ.)
* Куча (Пул памяти общего назначения, находится в RAM, в нем размещаются все объекты Java. new - память выделяется из кучи )
* Постоянное хранилище (Значения констант часто встраиваются прямо в код программы, т.к. они неизменны. Иногда такие данные могут размещаться в постоянной памяти ROM)
* Внешнее хранилище (Если данные хранятся вне программы, он могут существовать и тогда, когда она не выполняется. Пример: потоковые объекты (для передачи на другие машины) и долгоживущие объекты (запоминаются на диске))

**Особый случай: примитивные типы**

Требуют особого обращения, так как часто применяются, и в куче недостаточно эффективно хранить. Java следует по примеру С и С++, вместо создания переменной с помощью new создается «автоматическая» переменная, не являющаяся ссылкой. Она хранит значение и располагается в стеке, операции с ней гораздо производительнее.

Неизменность размера типа переменной – одна из причин улучшенной переносимости программ Java.

**Массивы**

Одной из основных целей Java является безопасность. Массивы гарантированно инициализируются, и к ним невозможен доступ за пределами его границ. Проверка границ массива обходится дорого, но повышение безопасности стоит того.

При объявлении массива объектов на самом деле создается массив ссылок, и каждая из этих ссылок автоматически инициализируется null.

Массивы простейших типов заполняются нулями.

**Объекты никогда не приходится удалять**

Сборщик мусора все сделает сам.

Область действия объектов отличается от области действия примитивов. Объект созданный оператором new, будет доступен до конца области действия. Если написать:

{

String s = new String(“sd);

} // конец области действия

Ссылка s исчезнет в конце области действия. Однако сам объект, на который указывала ссылка, все еще будет занимать память. Уборщик мусора, который наблюдает за объектами, созданными оператором new, определяет, на какие из них больше нет ссылок, тогда он освобождает память этих объектов, которая становится доступной для дальнейшего использования. Таким образом, никогда не придется очищать память вручную.

Если переменная используется как член класса, то гарантируется значение по-умолчанию (для int это 0 и т.п.). Такая гарантия не относится к локальным переменным, которые не являются полями класса, переменная будет инициализирована случайным значением.

**Методы** в Java определяют сообщения, принимаемые объектом. Имя метода и его список аргументов (объединяемые термином сигнатура) обеспечивают однозначную идентификацию метода.

**Видимость имен**

Для создания уникальных имен библиотек разработчики Java предлагают использовать доменное имя, записанное в обратном порядке, так как эти имена всегда уникальны.

В Java не существует проблемы опережающих ссылок.

**Ключевое слово static**

Объект появляется только после того, как он будет создан ключевым словом new, и только начиная с этого момента для него выделяется память и появляется возможность вызова методов. Но есть две ситуации, в которых такой подход недостаточен. Первая – это когда некоторые данные должны храниться в единственном числе независимо от того, сколько было создано объектов класса. Вторая – когда требуется метод, не привязанный ни к какому конкретному объекту класса, то есть метод, который можно вызвать даже при полном отсутствии объектов класса.

Такой эффект достигается использованием ключевого слова static, делающего элемент класса статическим. Данные или метод, объявленные как static, не привязаны к определенному экземпляру этого класса.

Следующий код создает статическое поле класса и инициализирует его:

Class StaticTest{

static int i = 47;

}

Теперь даже при создании двух объектов этого класса, для элемента StaticTest.i выделяется единственный блок памяти. Оба объекта совместно используют одно значение i.

Для обращения к статической переменной есть возможность обращаться прямо через имя класса:

StaticTest.i++;

То же самое касается статических методов класса. Для нестатических членов такая возможность отсутствует.

**Документация в комментариях**

Для упрощения ведения документации, код и документацию объединяют в одном файле, но для полноты картины нужен специальный синтаксис комментариев, чтобы помечать документацию, и инструмент, который извлекал бы эти комментарии и оформлял их в подходящем виде.

Инструмент для извлечения комментариев называется javadoc, он является частью пакета JDK. Он не только извлекает помеченную информацию, но также узнает имя класса или метода, к которому относится данный фрагмент документации. Результатом работы программы является html-файл. Также можно дополнить javadoc своими собственными расширениями, называемыми доклетами, в которых можно проводить специальные операции над обрабатываемыми данными.

**3 глава. Операторы**

**Простые команды печати**

Вместо System.out.println(“…”); можно использовать команду print(“…”); но нужно сделать статический импорт:

import static net.minview.util.Print.\*;

**Присваивание**

Присваивание объектов отличается от обычного присваивания. При выполнении операций с объектом вы в действительности работаете со ссылкой, поэтому присваивание одного объекта другому в действительности означает копирование ссылки из одного места в другое. Это означает, что при выполнении команды c = d для объектов в конечном итоге c и d указывают на один объект, которому изначально соответствовала только ссылка d. Во многих языках программирования можно было ожидать, что c и d будут независимы все время, но из-за присваивания ссылок изменение объекта c будет отражаться на объекте d. Это происходит из-за того, что t1 и t2 содержат одинаковые ссылки, указывающие на один объект. (Исходная ссылка, которая содержалась в с и указывала на объект с определенным значением, была перезаписана во время присваивания и фактически потеряна, её объект будет вскоре удален уборщиком мусора.)

Этот феномен совмещения имен часто называют синонимией (aliasing), и именно она является основным способом работы с объектами в Java.

Но что делать, если совмещение имен нежелательно? Тогда можно пропустить присваивание и записать:

c.value = d.value;

При этом программа сохранит 2 разных объекта. Но прямая работа с полями данных внутри объектов противоречит принципам ООП.

**Автоувеличение и автоуменьшение**

Каждый из этих операторов существует в двух версиях – префиксной и постфиксной. Префиксный инкремент значит, что оператор ++ записывается перед переменной или выражением, а при постфиксном инкременте оператор следует после переменной или выражения. Аналогично, при префиксном декременте. Для префиксного инкремента и декремента сначала выполняется операция, а затем выдается результат. Для постфиксной записи сначала выдается значение и лишь затем выполняется операция.

**Операторы сравнения**

Для сравнения объектов можно использовать метод equals(), который нужно переопределять в собственных классах.

**Ускоренное вычисление**

При работе с логическими операторами можно столкнуться с феноменом, называемым «ускоренным вычислением». Это значит, что выражение вычисляется только до тех пор, пока не станет очевидно, что он принимает значение «истина» или «ложь». В результате некоторые части логического выражения могут быть проигнорированы в процессе сравнения. Это обеспечивает потенциальный выигрыш в производительности.

**Литералы**

0x2f – шестнадцатеричное число (нижний регистр)

0X2F – шестнадцатеричное число (верхний регистр)

0177 – восьмеричное число (нижний регистр)

200L - long

200l – long(можно запутаться)

1F - float

1f - float

1D - double

1d – double

**Экспоненциальная запись**

е означает 10 в степени.

float expFloat = 1.39e – 43f;

Это 1.39 \* 10 -43

**Тернарный оператор**

Логическое-условие ? выражение0 : выражение1

Может ухудшить читаемость кода, обычно его применение ограничивается присваиванием переменной одного из двух значений.

**Перегрузка операторов**

В языке Java отсутствует перегрузка операторов.

**Оператор + и += для String**

int x = 0;

String s =”x “;

print(s + x);

print(x + “ “ + s); //Преобразует x в String

print(“” + x); //Сокращение для Integer.toString();

**Привидение типов**

В случае с типами char, byte и short можно заметить эффект повышения при использовании арифметических операторов. Любая арифметическая операция с этими типами дает результат типа int, который затем нужно явно приводить к изначальному типу (сужающие привидение, при котором возможно потеря информации).

**4 глава. Управляющие конструкции**

В Java задействованы все управляющие конструкции языка С.

If-else, while, do-while, for, return, break, switch.

**Оператор-запятая**

Позволяет записать несколько переменных в цикле for, но все эти переменные должны быть одного типа:

for (int i = 0, j = I + 10; i < 5; i++, j = i \* 2) {

}

**Синтаксис foreach**

В Java SE5 появилась новая, более компактная форма for для перебора элементов массивов и контейнеров. Не требует ручного изменения переменной int для перебора последовательности объектов – цикл автоматически представляет очередной элемент.

Пример.

for (char c : “An African Swallow”.toCharArray() )

System.out.print(c + “ “);

Синтаксис foreach также работает для любого объекта, поддерживающего интерфейс Iterable.

**Break и continue**

Команда break завершает цикл, при этом оставшиеся операторы цикла не выполняются. Команда continue останавливает выполнение текущей итерации цикла и переходит к началу цикла, чтобы начать выполнение нового шага или же вернуться в начало цикла.

**Команда goto**

Java не имеет команды goto, однако существует механизм, похожий на безусловный переход.

Метка представляет собой идентификатор с последующим двоеточием:

label1:

Единственное место, где в Java метка может оказаться полезной,- прямо перед телом цикла. Обычные версии break прерывают лишь текущий цикл, но с помощью метки можно досрочно завершать циклы и передавать выполнение в точку, адресуемую меткой:

label1:

внешний цикл {

внутренний цикл {

…

break;

….

continue;

…

continue label1;

…

break label1;

}

}

В третьем варианте команда влечет выход из обоих циклов и выполнение цикла начинается снова с внешнего цикла. В 4 варианте команда останавливает действие обоих циклов, повторный вход в итерацию не происходит.

**5 глава. Инициализация и завершение.**

**Конструктор**

Конструктор, не получающий аргументов, называется конструктором по умолчанию. Конструкторы не возвращают никогда и ничего.

**Перегрузка методов**

Даже разного порядка аргументов достаточно для того, чтобы методы считались разными. Но такой подход почти не используется, так как он усложняет сопровождение программного кода.

**Ключевое слово this**

Употребляется только в особых случаях, когда необходимо сослаться на текущий объект. Может использоваться только внутри нестатического метода и предоставляет ссылку на объект, для которого был вызван метод. При вызове метода вашего класса из другого метода класса использовать this не нужно.

Для передачи текущего объекта внешнему методу используется ключевое слово this:

class Person {

public void eat(Apple apple) {

Apple peeled = apple.getPeeled();

System.out.println("Yummy");

}

}

class Peeler {

static Apple peel(Apple apple) {

// ... remove peel

return apple; // Peeled

}

}

class Apple {

Apple getPeeled() { return Peeler.peel(this); }

}

public class PassingThis {

public static void main(String[] args) {

new Person().eat(new Apple());

}

} /\* Output:

Yummy

\*///:~

**Вызов конструкторов из конструкторов**

Иногда бывает удобно вызвать один конструктор из другого, чтобы избежать дублирования кода. Такая операция проводится с использованием this.

Обычно при употреблении this подразумевается “текущий объект”, но в конструкторе это ключевое слово имеет другой смысл: при использовании его со списком аргументов вызывается конструктор, соответствующий данному списку. Таким образом, появляется возможность прямого вызова других конструкторов:

/: initialization/Flower.java

// Calling constructors with "this"

import static net.mindview.util.Print.\*;

public class Flower {

int petalCount = 0;

String s = "initial value";

Flower(int petals) {

petalCount = petals;

print("Constructor w/ int arg only, petalCount= "

+ petalCount);

}

Flower(String ss) {

print("Constructor w/ String arg only, s = " + ss);

s = ss;

}

Flower(String s, int petals) {

this(petals);

//! this(s); // Can't call two!

this.s = s; // Another use of "this"

print("String & int args");

}

Flower() {

this("hi", 47);

print("default constructor (no args)");

}

void printPetalCount() {

//! this(11); // Not inside non-constructor!

print("petalCount = " + petalCount + " s = "+ s);

}

public static void main(String[] args) {

Flower x = new Flower();

x.printPetalCount();

}

} /\* Output:

Constructor w/ int arg only, petalCount= 47

String & int args

default constructor (no args)

petalCount = 47 s = hi

\*///:~

При вызове одного конструктора через this вызывать второй запрещается. Вдобавок вызов другого конструктора должен быть первой выполняемой операцией, иначе компилятор выдаст сообщение об ошибке.

**Значение ключевого слова static**

У статических методов не существует ссылки this. Статические методы вызываются для класса, без каких-либо объектов.

**Очистка: финализация и уборка мусора**

Уборщик мусора умеет освобождать память, выделенную оператором new, но ему неизвестно, как следует очищать специфическую память объекта. Для таких ситуаций в Java предусмотрен метод finalize(), который можно определить в собственном классе. Когда уборщик мусора готов освободить память, использованную вашим объектом, то он для начала вызывает метод и только после этого освобождает занимаемую объектом память. Таким образом, метод finalize() позволяет выполнить завершающие действия во время работы уборщика мусора, но не стоит путать его с деструктором языка C++ - ф-ей, всегда вызываемой перед разрушением объекта. В Java объекты не всегда удаляются уборщиком мусора. Уборка мусора не является уничтожением.

Если перед тем, как объект станет ненужным, необходимо выполнить некоторое завершающее действие, то это действие придется выполнить собственноручно.

**Для чего нужен метод finalize()?**

Похоже, что поддержка метода finalize() была введена в язык, чтобы сделать возможными операции с памятью в стиле С, с привлечением нестандартных механизмов выделения памяти. Это может произойти в основном при использовании методов, предоставляющих способ вызова не-Java-кода из программы на Java. В действительности программа Java может вызвать любую процедуру или функцию на любом языке. Во внешнем коде можно выделить память вызовом функций С, относящихся к семейству malloc().

Метод finalize() используется нечасто.

**Как работает уборщик мусора**

Дописать

**Инициализация членов класса**

Члены класса автоматически инициализируются. Ссылке на объект автоматически присваивается значение null.

**Явная инициализация**

Прямое присваивание значения переменной прямо в точке ее объявления в классе.

**Инициализация конструктором**

Переменные инициализируются внутри конструктора, но сначала пройдет автоматическая инициализация.

**Порядок инициализации**

Внутри класса очередность инициализации определяется порядком следования переменных, объявленных в этом классе. Определения переменных могут быть разбросаны по разным определениям методов, но в любом случае переменные инициализируются перед вызовом любого метода – даже конструктора.

**Инициализация статических данных**

Данные статических полей всегда существуют в единственном экземпляре, независимо от количества созданных объектов. Ключевое слово static не может применяться к локальным переменным, только к полям.

Если статическое поле относится к примитивному типу, при отсутствии явной инициализации ему присваивается значение по умолчанию. Если это ссылка на объект, то ей присваивается значение null.

Статическая инициализация происходит только в случае необходимости. Они инициализируются только при создании первого объекта (или при первом обращении к статическим данным). После этого статические объекты заново не инициализируются.

Сначала инициализируются статические члены, если они еще не были проинициализированы, и только затем нестатические объекты.

Обобщим знания о процессе создания объекта. Для примера возьмем класс с именем Dog:

* Хотя ключевое слово static и не используется явно, конструктор в действительности является статическим методом. При создании первого объекта типа Dog или при первом вызове статического метода/обращения к статическому полю класса Dog интерпретатор Java должен найти класс Dog.class. Поиск осуществляется в стандартных каталогах, перечисленных в переменной окружения CLASSPATH.
* После загрузки Dog.class (с созданием особого объекта Class) производится инициализация статических элементов. Таким образом, инициализация статических членов проводится только один раз, при первой загрузке объекта Class.
* При создании нового объекта конструкцией new Dog() для начала выделяется блок памяти, достаточный для хранения объекта Dog в куче.
* Выделенная память заполняется нулями, при этом все примитивные поля объекта автоматически инициализируются значениями по умолчанию.
* Выполняются все действия по инициализации, происходящие в точке определения полей класса.
* Выполняются конструкторы.

**Явная инициализация статических членов**

Java позволяет сгруппировать несколько действий по инициализации объектов static в специальной конструкции, называемой *статическим блоком.*

Этот код, как и остальная инициализация static, выполняется только один раз: при первом создании объекта этого класса или при первом обращении к статическим членам этого класса (даже если ни один объект класса не создается).

Пример:

import static net.mindview.util.Print.\*;

class Cup {

Cup(int marker) {

print("Cup(" + marker + ")");

}

void f(int marker) {

print("f(" + marker + ")");

}

}

class Cups {

static Cup cup1;

static Cup cup2;

static {

cup1 = new Cup(1);

cup2 = new Cup(2);

}

Cups() {

print("Cups()");

}

}

public class ExplicitStatic {

public static void main(String[] args) {

print("Inside main()");

Cups.cup1.f(99); // (1)

}

// static Cups cups1 = new Cups(); // (2)

// static Cups cups2 = new Cups(); // (2)

} /\* Output:

Inside main()

Cup(1)

Cup(2)

f(99)

\*///:~

**Инициализация нестатических данных экземпляра**

Имеется схожий синтаксис для инициализации нестатических переменных для каждого объекта:

import static net.mindview.util.Print.\*;

class Mug {

Mug(int marker) {

print("Mug(" + marker + ")");

}

void f(int marker) {

print("f(" + marker + ")");

}

}

public class Mugs {

Mug mug1;

Mug mug2;

{

mug1 = new Mug(1);

mug2 = new Mug(2);

print("mug1 & mug2 initialized");

}

Mugs() {

print("Mugs()");

}

Mugs(int i) {

print("Mugs(int)");

}

public static void main(String[] args) {

print("Inside main()");

new Mugs();

print("new Mugs() completed");

new Mugs(1);

print("new Mugs(1) completed");

}

} /\* Output:

Inside main()

Mug(1)

Mug(2)

mug1 & mug2 initialized

Mugs()

new Mugs() completed

Mug(1)

Mug(2)

mug1 & mug2 initialized

Mugs(int)

new Mugs(1) completed

\*///:~

Такой синтаксис необходим для поддержки инициализации анонимных внутренних классов, но он также гарантирует, что некоторые операции будут выполнены независимо от того, какой именно конструктор был вызван в программе. Секция инициализации экземпляра выполняется раньше любых конструкторов.

**Инициализация массивов**

Массивы определяются и используются с помощью оператора индексирования [ ]. Чтобы определить ссылку на массив в программе нужно написать:

int[] a1;

Квадратные скобки также могут размещаться после идентификатора, эффект будет точно таким же:

int a1[];

Инициализация

int a1 = { 1, 2, 3, 4, 5 };

В Java можно присвоить один массив другому:

a2 = a1;

В данном случае скопируются ссылки.

Все массивы содержат поле, которое можно только прочитать – length (количество элементов в массиве).

Есть метод Arrays.toString(), входящий в стандартную библиотеку java.util, метод выдает печатную версию одномерного массива.

**Списки аргументов переменной длины**

Такой список способен содержать неизвестное заранее количество аргументов неизвестного типа. Так как абсолютно все классы являются производными от общего корневого класса Object, можно создать метод, принимающий в качестве аргумента массив Object:

class A { int i; }

public class VarArgs {

static void printArray(Object[] args) {

for(Object obj : args)

System.out.print(obj + " ");

System.out.println();

}

public static void main(String[] args) {

printArray(new Object[]{

new Integer(47), new Float(3.14), new Double(11.11)

});

printArray(new Object[]{"one", "two", "three" });

printArray(new Object[]{new A(), new A(), new A()});

}

} /\* Output: (Sample)

47 3.14 11.11

one two three

A@1a46e30 A@3e25a5 A@19821f

\*///:~

Print() принимает массив объектов типа Object, перебирает его элементы и выводит их.

До выхода Java SE5 переменные списки аргументов реализовывались именно так. В Java SE5 была добавлена возможность – для определения переменного списка аргументов может использоваться многоточие:

public class NewVarArgs {

static void printArray(Object... args) {

for(Object obj : args)

System.out.print(obj + " ");

System.out.println();

}

public static void main(String[] args) {

// Can take individual elements:

printArray(new Integer(47), new Float(3.14),

new Double(11.11));

printArray(47, 3.14F, 11.11);

printArray("one", "two", "three");

printArray(new A(), new A(), new A());

// Or an array:

printArray((Object[])new Integer[]{ 1, 2, 3, 4 });

printArray(); // Empty list is OK

}

} /\* Output: (75% match)

47 3.14 11.11

47 3.14 11.11

one two three

A@1bab50a A@c3c749 A@150bd4d

1 2 3 4

\*///:~

Списки аргументов переменной длины избавляют от необходимости явной записи синтаксиса массивов – компилятор делает это автоматически.

Списки аргументов переменной длины могут использоваться и с другими типами:

public class OptionalTrailingArguments {

static void f(int required, String... trailing) {

System.out.print("required: " + required + " ");

for(String s : trailing)

System.out.print(s + " ");

System.out.println();

}

public static void main(String[] args) {

f(1, "one");

f(2, "two", "three");

f(0);

}

} /\* Output:

required: 1 one

required: 2 two three

required: 0

\*///:~

Следующий пример показывает, что список аргументов переменной длины превращается в массив, а при отсутствии элементов он становится массивом нулевой длины.

public class VarargType {

static void f(Character... args) {

System.out.print(args.getClass());

System.out.println(" length " + args.length);

}

static void g(int... args) {

System.out.print(args.getClass());

System.out.println(" length " + args.length);

}

public static void main(String[] args) {

f('a');

f();

g(1);

g();

System.out.println("int[]: " + new int[0].getClass());

}

} /\* Output:

class [Ljava.lang.Character; length 1

class [Ljava.lang.Character; length 0

class [I length 1

class [I length 0

int[]: class [I

\*///:~

Можно вызывать функцию вообще без аргументов.

Списки аргументов переменной длины могут усложнит перегрузку. При вызове перегруженной функции без аргументов компилятор не сможет определиться какой из методов следует вызвать. Как правило, список аргументов переменной длины следует использовать только в одной версии перегруженного метода.

**Перечисления**

Ключевое слово enum введено в Java SE5. Оно значительно упрощает работу программиста при группировке и использовании перечисляемых типов.

public enum Spiciness {

NOT, MILD, MEDIUM, HOT, FLAMING

}

Приведенный фрагмент создает перечислимый тип с именем Spiciness, состоящий из 4 именованных значений. Так как экземпляры перечислимых типов являются константами, то по правилом они записываются исключительно прописными буквами (если имя состоит из нескольких слов, то эти слова разделяются пробелами).

Чтобы использовать перечисление нужно создать ссылку на перечислимый тип:

public class SimpleEnumUse {

public static void main(String[] args) {

Spiciness howHot = Spiciness.MEDIUM;

System.out.println(howHot);

}

} /\* Output:

MEDIUM

\*/

На самом деле перечисления являются классами и обладают своими методами, вот некоторые из них:

toString() –вывод имени экземпляра перечислимого типа.

ordinal() – порядок объявления констант.

values() – возвращает массив констант в enum в порядке их объявления.

Удобно использовать switch для работы с enum.

**6 глава. Управление доступом.**

Смысл управления доступом (или сокрытия реализации) заключается в постепенном совершенствовании кода.

Рефакторинг – переработки работоспособности кода, направленного на то, чтобы сделать его более удобочитаемым и понятным и упростить сопровождение.

**Пакет как библиотечный модуль**

Пакет содержит группу классов, объединенных в одном пространстве имен. Для импорта классов пакета используется ключевое слово import.

Файл с исходным текстом на Java часто называют компилированным модулем. Файл .java может содержать один и только один открытый класс (public).

**Структура кода**

В результате компиляции для каждого класса, определенного в файле .java, создается класс с тем же именем, но с расширением .class. На компилируемом языке компилятор генерирует промежуточные файлы (.OBJ обычно), которые затем объединяются компоновщиком для получения исполняемого файла или библиотеки. Java работает не так. Рабочая программа представляет собой набор однородных файлов .class, которые объединяются в пакет и сжимаются в файл JAR (утилитой Java jar). Интерпретатор Java отвечает за поиск, загрузку и интерпретацию этих файлов.

Ключевые слова package и import позволяют разработчику библиотеки организовать логическое деление глобального пространства имен, предотвращающее конфликты имен независимо от того, сколько людей подключатся к Интернету и начнут писать свои классы на Java.

**Конфликты имен**

Предположим, что программа содержит следующие директивы:

import net.mindview.simple.\*;

import java.util.\*;

Эти пакеты содержат класс с именем Vector. Конфликт произойдет при попытке создать Vector:

Vector v = new Vector();

Компилятор выдаст сообщение об ошибке и заставит явно указать нужное имя:

java.util.Vector v = new java.util.Vector();

**Пользовательские библиотеки**

После компиляции методы пользовательских библиотек могут использоваться где угодно, для чего угодно, но нужно чтобы файл располагался в одном из каталогов, указанных в переменной окружения CLASSPATH. Затем в программу достаточно включить директиву import static.

**Спецификаторы доступа Java**

В Java спецификаторы доступа public, protected и private располагаются перед определением членов классов – как полей, так и методов. Каждый спецификатор доступа управляет только одним отдельным определением.

Если спецификатор доступа не указан, то используется «пакетный» уровень доступа.

**Доступ в пределах пакета**

Доступ по умолчанию не имеет ключевого слова, но часто его называют доступом в пределах пакета. Это значит, что член класса доступен для всех остальных классов текущего пакета, но для классов за пределами пакета он воспринимается как приватный.

**Public**

Объявление члена класса доступно для всех.

**Private**

Доступ к члену класса не предоставляется никому, кроме методов этого класса. Сокрытие данных.

**Protected**

Связано с понятием наследования. (Будет рассмотрено далее)

**Интерфейс и реализация**

Контроль над доступом часто называют сокрытием реализации. Помещение данных и методов в классы в комбинации с сокрытием реализации часто называют инкапсуляцией.

Доступ к типам данных ограничивается по двум причинам. Первая причина – программист-клиент должен знать, что он может использовать, а что не может. Вторая причина – разделение интерфейса и реализации.

Для большей ясности при написании классов можно использовать такой стиль: сначала записываются public-члены, затем protected-члены, затем члены с доступом в пределах пакета и, наконец, закрытые члены(private). Преимущество такой схемы в том, что при чтении исходного текста пользователь сначала видит то, что ему важно, а затем останавливается при переходе к закрытым членам, являющимся частью внутренней реализации.

**Доступ к классам**

В Java с помощью спецификаторов доступа возможно также указать, какие из классов внутри библиотеки будут доступны для ее пользователей. Если нужно, чтобы класс был открыт программисту-клиенту, то нужно добавить public для класса в целом.

Существуют некоторые ограничения:

* В каждом компилируемом модуле может существовать только один открытый (public) класс. Идея в том, что каждый компилируемый модуль содержит определенный открытый интерфейс и реализуется этим открытым классом. В модуле может содержаться произвольное количество вспомогательных классов с доступом в пределах пакета.
* Имя открытого класса должно в точности совпадать с именем файла, в котором содержится компилируемый модуль.
* Компилируемый модуль может вообще не содержать открытых классов (хотя это и не типично). В этом случае файлу можно присвоить любое имя по вашему усмотрению.

Класс нельзя объявить как private (что сделает класс недоступным для окружающих, использовать он сможет только сам себя) или protected. Поэтому выбор при задании доступа к классу небольшой: в пределах пакета или открытый(public).

Если вы хотите перекрыть доступ к классу для всех, то нужно объявить все его конструкторы со спецификатором private, соответственно запрещая кому бы то ни было создание объектов этого класса. Только вы сами, в статическом методе этого класса, сможете создавать такие объекты. Пример:

class Soup1 {

private Soup1() {}

// (1) Allow creation via static method:

public static Soup1 makeSoup() {

return new Soup1();

}

}

class Soup2 {

private Soup2() {}

// (2) Create a static object and return a reference

// upon request.(The "Singleton" pattern):

private static Soup2 ps1 = new Soup2();

public static Soup2 access() {

return ps1;

}

public void f() {}

}

// Only one public class allowed per file:

public class Lunch {

void testPrivate() {

// Can't do this! Private constructor:

//! Soup1 soup = new Soup1();

}

void testStatic() {

Soup1 soup = Soup1.makeSoup();

}

void testSingleton() {

Soup2.access().f();

}

}

Без явного определения хотя бы одного конструктора компилятор сгенерирует конструктор по-умолчанию.

**7 глава. Повторное использование классов.**

Есть два пути реализации этой идеи. Первый довольно прямолинеен: объекты уже имеющихся классов просто создаются внутри вашего нового класса. Механизм построения нового класса из объектов существующих классов называется композицией (composition). Вы просто используете функциональность готового кода, а не его структуру.

Второй способ интереснее – новый класс создается как специализация уже существующего класса. Взяв существующий класс за основу, вы добавляете к нему свой код без изменения существующего класса. Этот механизм называется наследованием (inheritance), и большую часть работы в нем совершает компилятор.

**Синтаксис композиции**

Ссылка на внедряемый объект просто включается в новый класс. Компилятор не создает объектов для ссылок «по умолчанию», и это логично, потому что во многих случаях это привело бы к лишним затратам ресурсов. Если вам понадобится проинициализировать ссылку, то нужно сделать это самостоятельно:

* В точке определения объекта. Это значит, что объект всегда будет инициализироваться перед вызовом конструктора.
* В конструкторе данного класса.
* Перед использованием объекта. Этот способ часто называют отложенной инициализацией. Он может сэкономить ресурсы в ситуациях, где создавать объект каждый раз необязательно и накладно.
* С использованием инициализации экземпляров.

Все 4 способа продемонстрированы в следующем примере:

import static net.mindview.util.Print.\*;

//Инициализация в конструкторе с композицией

class Soap {

private String s;

Soap() {

print("Soap()");

s = "Constructed";

}

public String toString() { return s; }

}

public class Bath {

private String // Initializing at point of definition:

s1 = "Happy",

s2 = "Happy",

s3, s4;

private Soap castille;

private int i;

private float toy;

public Bath() {

print("Inside Bath()");

s3 = "Joy";

toy = 3.14f;

castille = new Soap();

}

// Instance initialization:

{ i = 47; }

public String toString() {

if(s4 == null) // Delayed initialization:

s4 = "Joy";

return

"s1 = " + s1 + "\n" +

"s2 = " + s2 + "\n" +

"s3 = " + s3 + "\n" +

"s4 = " + s4 + "\n" +

"i = " + i + "\n" +

"toy = " + toy + "\n" +

"castille = " + castille;

}

public static void main(String[] args) {

Bath b = new Bath();

print(b);

}

} /\* Output:

Inside Bath()

Soap()

s1 = Happy

s2 = Happy

s3 = Joy

s4 = Joy

i = 47

toy = 3.14

castille = Constructed

\*///:~

**Синтаксис наследования**

Фактически наследование всегда используется при создании класса, потому что даже если класс не объявляется производным от другого класса, он автоматически становится производным от корневого класса Java Object.

При использовании наследования вы говорите: «Этот новый класс похож на тот старый класс». В программе этот факт выражается перед фигурной скобкой, открывающей тело класса: сначала записывается ключевое слово extends, а затем имя базового класса. Тем самым вы автоматически получаете доступ ко всем полям и методам базового класса.

Пример:

// Inheritance syntax & properties.

import static net.mindview.util.Print.\*;

class Cleanser {

private String s = "Cleanser";

public void append(String a) { s += a; }

public void dilute() { append(" dilute()"); }

public void apply() { append(" apply()"); }

public void scrub() { append(" scrub()"); }

public String toString() { return s; }

public static void main(String[] args) {

Cleanser x = new Cleanser();

x.dilute(); x.apply(); x.scrub();

print(x);

}

}

public class Detergent extends Cleanser {

// Change a method:

public void scrub() {

append(" Detergent.scrub()");

super.scrub(); // Call base-class version

}

// Add methods to the interface:

public void foam() { append(" foam()"); }

// Test the new class:

public static void main(String[] args) {

Detergent x = new Detergent();

x.dilute();

x.apply();

x.scrub();

x.foam();

print(x);

print("Testing base class:");

Cleanser.main(args);

}

} /\* Output:

Cleanser dilute() apply() Detergent.scrub() scrub() foam()

Testing base class:

Cleanser dilute() apply() scrub()

\*///:~