Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 1. Общие представления о языке Java

1.1. Java и другие языки программирования. Системное и прикладное программирование

Язык программирования Java был создан в рамках проекта корпорации *Sun* Microsystems по созданию компьютерных программно-аппаратных комплексов нового поколения. Первая версия языка была официально опубликована в 1995 году. С тех пор язык *Java* стал стандартом де-факто, вытеснив за десять лет языки C и C++ из многих областей программирования. В 1995 году они были абсолютными лидерами, но к 2006 году число программистов, использующих *Java*, стало заметно превышать число программистов, использующих C и C++, и составляет более четырех с половиной миллионов человек. А число устройств, в которых используется *Java*, превышает полтора миллиарда.

Как связаны между собой языки C, C++, JavaScript и *Java*? Что между ними общего, и в чем они отличаются? В каких случаях следует, а в каких не следует их применять? Для того чтобы ответить на этот вопрос, следует сначала остановиться на особенностях программного обеспечения предыдущих поколений и на современных тенденциях в развитии программного обеспечения.

Первоначально *программирование* компьютеров шло в машинных кодах. Затем появились языки *ASSEMBLER*, которые заменили команды процессоров мнемоническими сокращениями, гораздо более удобными для человека, чем последовательности нулей и единиц. Их принято считать языками программирования *низкого уровня* (то есть близкими к аппаратному уровню), так как они ориентированы на особенности конкретных процессоров. Именно поэтому программы, написанные на языках *ASSEMBLER*, нельзя было переносить на компьютеры с другим типом процессора – процессоры имели несовместимые наборы команд. То есть они были непереносимы на уровне исходного кода (*source code*).

Программы, написанные в машинных кодах, то есть в виде последовательности ноликов и единиц, соответствующих командам процессора и необходимым для них данным, нет необходимости как-то преобразовывать. Их можно скопировать в нужное *место* памяти компьютера и передать управление первой команде программы (задать точку *входа в программу*).

Программы, написанные на каком-либо языке программирования, сначала надо перевести из одной формы (текстовой) в другую (двоичную, то есть в машинные коды). Процесс такого перевода называется *трансляцией* (от английского *translation* – "перевод", "перемещение"). Не обязательно переводить программу из текстовой формы в двоичные коды, возможен процесс трансляции с одного языка программирования на другой. Или из кодов одного типа процессора в коды другого типа.

Имеется два основных вида трансляции – *компиляция* и *интерпретация*.

При *компиляции* первоначальный набор инструкций однократно переводится в исполняемую форму (машинные коды), и в последующем при работе программы используются только эти коды.

При *интерпретации* во время каждого вызова необходимых инструкций каждый раз сначала происходит перевод инструкций из одной формы (текстовой или двоичной) в другую – в исполняемые коды процессора используемого компьютера. И только потом эти коды исполняются. Естественно, что интерпретируемые коды исполняются медленнее, чем скомпилированные, так как перевод инструкций из одной формы в другую обычно занимает в несколько раз больше времени чем выполнение полученных инструкций. Но *интерпретация* обеспечивает большую гибкость по сравнению с компиляцией, и в ряде случаев без нее не обойтись.

В 1956 году появился язык *FORTRAN* – первый *язык программирования высокого уровня* (то есть не ориентированный на конкретную аппаратную реализацию компьютера). Он обеспечил *переносимость* программ на уровне исходных кодов, но довольно дорогой ценой. Во-первых, *быстродействие* программ, написанных на *FORTRAN*, было в несколько раз меньше, чем для ассемблерных. Во-вторых, эти программы занимали примерно в два раза больше места в памяти компьютера, чем ассемблерные. И, наконец, пришлось отказаться от поддержки особенностей периферийных устройств – общение с "внешним миром" пришлось ограничить простейшими возможностями, которые в программе одинаково реализовывались для ввода-вывода с помощью перфокарточного считывателя, клавиатуры, принтера, текстового дисплея и т.д. Тем не менее языки программирования высокого уровня постепенно вытеснили языки *ASSEMBLER*, поскольку обеспечивали не только *переносимость* программ, но и гораздо более высокую их *надежность*, а также несоизмеримо более высокую скорость разработки сложного программного обеспечения. *FORTRAN* до сих пор остается важнейшим языком программирования для высокопроизводительных численных научных расчетов.

Увеличение аппаратных возможностей компьютеров (количества памяти, быстродействия, появления дисковой памяти большого объема), а также появление разнообразных периферийных устройств, привело к необходимости пересмотра того, как должны работать программы. Массовый выпуск компьютеров потребовал унификации доступа из программ к различным устройствам. Возникла идея, что из программы можно обращаться к устройству без учета особенностей его аппаратной реализации. Это возможно, если

обращение к устройству идет не напрямую, а через прилагающуюся программу – *драйвер* устройства (по-английски *driver* означает "водитель"). Появились операционные системы – наборы драйверов и программ, распределяющих ресурсы компьютера между разными программами. Соответственно, *программное обеспечение* стало разделяться на *системное и прикладное*. Системное *программное обеспечение* – непосредственно обращающееся к аппаратуре, прикладное – решающее какие-либо прикладные задачи и использующее аппаратные возможности компьютера не напрямую, а через вызовы программ операционной системы. Прикладные программы стали приложениями операционной системы, или, сокращенно, *приложениями* (applications). Этот термин означает, что *программа* может работать только под управлением операционной системы. Если на том же компьютере установить другой тип операционной системы, *программа-приложение* первой операционной системы не будет работать.

Требования к прикладным программам принципиально отличаются от требований к системным программам. От системного программного обеспечения требуется максимальное *быстродействие* и минимальное количество занимаемых ресурсов, а также возможность доступа к любым необходимым аппаратным ресурсам. От прикладного – максимальная функциональность в конкретной *предметной области*. При этом *быстродействие* и занимаемые ресурсы не имеют значения до тех пор, пока не влияют на функциональность. Например, нет совершенно никакой разницы, реагирует *программа* на нажатие клавиши на клавиатуре за одну десятую или за одну миллионную долю секунды. Правда, на первоначальном этапе создания прикладного программного обеспечения даже прикладные по назначению программы были системными по реализации, так как оказывались вынуждены напрямую обращаться к аппаратуре.

Язык C был создан в 1972 году в одной из исследовательских групп *Bell Laboratories* при разработке операционной системы Unix. Сначала была предпринята попытка написать операционную систему на *ASSEMBLER*, но после появления в группе новых компьютеров пришлось создать платформонезависимый *язык программирования* высокого уровня, с помощью которого можно было бы писать операционные системы. Таким образом, язык C создавался как язык для создания системного программного обеспечения, и таким он остается до сих пор. Его идеология и синтаксические конструкции ориентированы на максимальную близость к аппаратному уровню реализации операций – в той степени, в какой он может быть обеспечен на аппаратно-независимом уровне. При этом главным требованием была максимальная скорость работы и минимальное количество занимаемых ресурсов, а также возможность доступа ко всем аппаратным ресурсам. Язык C является языком *процедурного программирования*, так как его базовыми конструкциями являются *подпрограммы*. В общем случае подпрограммы принято называть подпрограммами-процедурами (откуда и идет название "*процедурное программирование*") и подпрограммами-функциями. Но в C имеются только подпрограммы-функции. Обычно их называют просто *функциями*.

Язык C произвел настоящую революцию в разработке программного обеспечения, получил широкое распространение и стал промышленным стандартом. Он до сих пор применяется для написания операционных систем и программирования *микроконтроллеров*. Но мало кто в полной мере осознает причины его популярности. В чем они заключались? – В том, что он смог обеспечить необходимую функциональность программного обеспечения в условиях низкой производительности компьютеров, крайней ограниченности их ресурсов и неразвитости периферийных устройств! При этом повторилась та же история, что и с *FORTRAN*, но теперь уже для языка *системного программирования*. Переход на *язык программирования* высокого уровня, но с минимальными потерями по производительности и ресурсам, дал большие преимущества.

Большое влияние на развитие *теории программирования* дал язык PASCAL, разработанный в 1974 году швейцарским профессором Никлаусом Виртом. В данной разработке имелось две части. Первая состояла в собственно языке программирования PASCAL, предназначенном для обучения идеям *структурного программирования*. Вторая заключалась в идее *виртуальной машины*. Никлаус Вирт предложил обеспечить *переносимость* программ, написанных на PASCAL, за счет компиляции их в набор команд некой абстрактной P-машины (P- сокращение от PASCAL), а не в *исполняемый код* конкретной аппаратной платформы. А на каждой аппаратной платформе должна была работать *программа*, интерпретирующая эти коды. Говорят, что такая *программа* эмулирует (то есть имитирует) систему команд несуществующего процессора. А саму программу называют виртуальной машиной.

В связи с ограниченностью ресурсов компьютеров и отсутствием в PASCAL средств *системного программирования* этот язык не смог составить конкуренцию языку C, так как практически все промышленное *программирование* вплоть до середины последней декады двадцатого века по реализации было системным. Идеи P-машины были в дальнейшем использованы и значительно усовершенствованы в *Java*.

Развитие теории и практики программирования привело к становлению в 1967-1972 годах нового направления – *объектного программирования*, основанного на концепциях работы с *классами и объектами*. Оно обеспечило принципиально новые возможности по сравнению с процедурным. Были предприняты попытки расширения различных языков путем введения в них конструкций объектного программирования. В 1982 году Бьерном Страуструпом путем такого расширения языка C был создан язык, который он назвал "C с классами". В 1983 году после очередных усовершенствований им был создан первый *компилятор* языка C++. Два плюса означают "C с очень большим количеством добавлений". C++ является надмножеством над языком C – на нем можно писать программы как на "чистом C", без использования каких-либо конструкций объектного программирования. В связи с этим, а также дополнительными преимуществами объектного

программирования, он быстро приобрел популярность и стал промышленным стандартом, сначала "де факто", а потом и "де юре". Так что в настоящее время C++ является базовым языком *системного программирования*. Длительное время он использовался и для написания прикладных программ. Но, как мы уже знаем, требования к прикладным программам совпадают с требованиями к системным только в том случае, когда *быстродействие* компьютера можно рассматривать как низкое, а ресурсы компьютера – малыми. Кроме этого, у языков C и C++ имеется еще два принципиальных недостатка: а) низкая *надежность* как на уровне исходного кода, так и на уровне исполняемого кода; б) отсутствие переносимости на уровне исполняемого кода. С появлением компьютерных сетей эти недостатки стали очень существенным ограничивающим фактором, поскольку вопросы безопасности при работе в локальных, и, особенно, глобальных сетях приобретают первостепенную *значимость*.

В 1995 году появились сразу два языка программирования, имеющие в настоящее время огромное *значение* – *Java*, разработанный в корпорации *Sun*, и JavaScript, разработанный в небольшой фирме Netscape *Communication*, получившей к тому времени известность благодаря разработке браузера Netscape Navigator.

Java создавался как универсальный язык, предназначенный для *прикладного программирования* в неоднородных компьютерных сетях как со стороны клиентского компьютера, так и со стороны сервера. В том числе – для использования на *тонких аппаратных клиентах* (устройствах малой вычислительной мощности с крайне ограниченными ресурсами). При этом скомпилированные программы *Java* работают только под управлением виртуальной *Java*-машины, поэтому они называются *приложениями Java*. *Синтаксис* операторов *Java* практически полностью совпадает с синтаксисом языка C, но, в отличие от C++, *Java* не является расширением C – это совершенно независимый язык, со своими собственными синтаксическими правилами. Он является гораздо более сильно типизированным по сравнению с C и C++, то есть вносит гораздо больше ограничений на действия с переменными и величинами разных типов. Например, в C/C++ нет разницы между целочисленными числовыми, булевскими и символьными величинами, а также адресами в памяти. То есть, например, можно умножить символ на булевское *значение*, из которого вычтено *целое число*, и разделить результат на *адрес*! В *Java* введен вполне разумный запрет на почти все действия такого рода.

Язык JavaScript создавался как узкоспециализированный прикладной *язык программирования HTML*-страниц, расширяющий возможности *HTML*, и в полной мере отвечает этим потребностям до сих пор. Следует подчеркнуть, что язык JavaScript *не имеет никакого отношения* к *Java*. Включение слова "*Java*" в название JavaScript являлось рекламным трюком фирмы Netscape *Communication*. Он также C-образен, но, в отличие от C, является интерпретируемым. Основное назначение JavaScript – программное управление элементами *WWW*-документов. Языки *HTML* и *XML* позволяют задавать статический, неизменный внешний вид документов, и с их помощью невозможно запрограммировать реакцию на действия пользователя. JavaScript позволяет ввести элементы программирования в поведение документа. Программы, написанные на JavaScript, встраиваются в документы в виде исходных кодов (сценариев) и имеют небольшой размер. Для упрощения работы с динамически формируемыми документами JavaScript имеет свободную *типизацию* – переменные меняют тип по результату присваивания. Поэтому программы, написанные на JavaScript, гораздо менее надежны, чем написанные на C/C++, не говоря уж про *Java*.

Java, JavaScript и C++ являются объектно-ориентированными языками программирования, и все они имеют C-образный *синтаксис* операторов. Но как объектные модели, так и базовые конструкции этих языков (за исключением синтаксиса операторов), в этих языках принципиально различны. Ни один из них не является версией или упрощением другого – это совсем разные языки, предназначенные для разных целей. Итак, *Java*-универсальный язык *прикладного программирования*, JavaScript – узкоспециализированный *язык программирования HTML*-документов, C++ – универсальный язык *системного программирования*.

В 2000 году в корпорации Microsoft была разработана платформа *.Net* (читается "дотнет", DotNet- в переводе с английского "точка *Net*"). Она стала альтернативой платформе *Java* и во многом повторяла ее идеи. Основное различие заключалось в том, что для этой платформы можно использовать *произвольное* количество языков программирования, а не один. Причем классы *.Net* оказываются совместимы как в целях наследования, так и по исполняемому коду независимо от языка, используемого для их создания. Важнейшим языком *.Net* стал *Java*-образный язык C# (читается "*Си* шарп"). Фактически, C# унаследовал от *Java* большинство особенностей – динамическую объектную модель, сборку "мусора", основные синтаксические конструкции. Хотя и является вполне самостоятельным языком программирования, имеющим много привлекательных черт. В частности, компонентные модели *Java* и C# принципиально отличаются.

Java стал первым универсальным C-образным языком *прикладного программирования*, что обеспечило легкость перехода на этот язык большого числа программистов, знакомых с C и C++. А наличие средств строгой проверки типов, ориентация на работу с компьютерными сетями, *переносимость* на уровне исполняемого кода и *поддержка* платформонезависимого графического интерфейса, а также запрет прямого обращения к аппаратуре обеспечили выполнение большинства требований, предъявлявшихся к языку *прикладного программирования*. Чем больше становятся *быстродействие* и объем памяти компьютеров, тем больше потребность в разделении прикладного и системного программного обеспечения. Соответственно, для прикладных программ исчезает необходимость напрямую обращаться к памяти и другим аппаратным

устройствам компьютера. Поэтому среди прикладных программ с каждым годом растет доля программного обеспечения, написанного на *Java* и языках *.Net*. Но как по числу программистов, так и по числу устройств, использующих соответствующие платформы, *Java* в настоящее время лидирует с большим отрывом.

1.2. Виртуальная Java-машина, байт-код, JIT-компиляция. Категории программ, написанных на языке Java

Первоначально *слово* "*программа*" означало последовательность инструкций процессора для решения какой-либо задачи. Эти инструкции являлись машинными кодами, и разницы между исходным и исполняемым кодом программы не было. Разница появилась, когда программы стали писать на языках программирования. При этом программой стали называть как текст, содержащийся в файле с исходным кодом, так и исполняемый *файл*.

Для устранения неоднозначности термина "*программа*", *исполняемый код* программы принято называть приложением (application). Термин "*приложение*" - сокращение от фразы "*приложение* операционной системы". Он означает, что *исполняемый код* программы может работать только под управлением соответствующей операционной системы. Работа под управлением операционной системы позволяет избежать зависимости программы от устройства конкретного варианта аппаратуры на компьютере, где она должна выполняться. Например, как автору программы, так и пользователю совершенно безразлично, как устроено устройство, с которого считывается *информация* - будет ли это *жесткий диск* с одной, двумя или шестнадцатью считывающими головками. Или это будет CD-привод, DVD-привод или еще какой-либо другой *тип носителя*. Но *переносимость* обычных приложений ограничивается одним типом операционных систем. Например, *приложение* MS Windows® не будет работать под Linux, и наоборот. Программы, написанные на языке *Java*, выполняются под управлением специальной программы -виртуальной *Java*-машины, и поэтому обладают переносимостью на любую операционную систему, где имеется соответствующая *Java*-машина. Благодаря этому они являются не приложениями какой-либо операционной системы, а *приложениями Java*.

Программы, написанные на языке *Java*, представляют из себя наборы классов и сохраняются в текстовых файлах с расширением *.java*. (Про то, что такое классы, будет рассказано несколько позже). При компиляции текст программы переводится (транслируются) в двоичные файлы с расширением *.class*. Такие файлы содержат байт-код - инструкции для абстрактного *Java*-процессора в виде байтовых последовательностей команд этого процессора и данных к ним. Для того, чтобы байт-код был выполнен на каком-либо компьютере, он должен быть переведен в инструкции для соответствующего процессора. Именно этим и занимается *Java*-машина. Первоначально байт-код всегда интерпретировался: каждый раз, как встречалась какая-либо *инструкция Java*-процессора, она переводилась в последовательность инструкций процессора компьютера. Естественно, это значительно замедляло работу приложений *Java*.

В настоящее время используется более сложная схема, называемая *JIT*-компиляцией (Just-In-Time) - компиляцией "по ходу дела", "налету". Когда какая-либо *инструкция* (или набор инструкций) *Java*-процессора выполняется в первый раз, происходит *компиляция* соответствующего ей байт-кода с сохранением скомпилированного кода в специальном буфере. При последующем вызове той же инструкции вместо ее интерпретации происходит вызов из буфера скомпилированного кода. Поэтому *интерпретация* происходит только при первом вызове инструкции.

Сложные оптимизирующие *JIT*-компиляторы действуют еще изощренней. Поскольку обычно *компиляция* инструкции идет гораздо дольше по сравнению с интерпретацией этой инструкции, время ее выполнения в первый раз при наличии *JIT*-компиляции может заметно отличаться в худшую сторону по сравнению с чистой интерпретацией. Поэтому бывает выгоднее сначала запустить процесс интерпретации, а параллельно ему в фоновом режиме компилировать инструкцию. Только после окончания процесса компиляции при последующих вызовах инструкции будет исполняться ее скомпилированный код. - До этого все ее вызовы будут интерпретироваться. Разработанная *Sun виртуальная машина* HotSpot осуществляет *JIT*-компиляцию только тех участков байт-кода, которые критичны к времени выполнения программы. При этом по ходу работы программы происходит *оптимизация* скомпилированного кода.

Благодаря компиляции программ *Java* в платформонезависимый байт-код обеспечивается *переносимость* этих программ не только на уровне исходного кода, но и на уровне скомпилированных приложений. Конечно, при этом на компьютере, где выполняется *приложение*, должна быть установлена *программа* виртуальной *Java*-машины (*Java Virtual Machine - JVM*), скомпилированная в коды соответствующего процессора (*native* code - "родной" код). На одном и том же компьютере может быть установлено несколько *Java*-машин разных версий или от разных производителей. Спецификация *Java*-машины является открытой, точно так же, как требования к компилятору языка *Java*. Поэтому различные фирмы, а не только *Sun*, разрабатывают компиляторы *Java* и *Java*-машины.

Приложение операционной системы запускается с помощью средств операционной системы. *Приложение Java*, напротив, запускается с помощью виртуальной *Java*-машины, которая сама является приложением операционной системы. Таким образом, сначала стартует *Java*-машина. Она получает в качестве параметра *имя файла* с компилированным кодом класса. В этом классе ищется и запускается на выполнение *подпрограмма* с именем main.

Приложения *Java* обладают не только хорошей переносимостью, но и высокой скоростью работы. Однако даже при наличии *JIT*-компиляции они все-таки могут выполняться медленнее, чем программы, написанные на C или C++. Это связано с тем, что *JIT-компиляция* создает не такой оптимальный код как многопроходный *компилятор* C/C++, который может тратить очень большое время и много ресурсов на отыскивание конструкций программы, которые можно оптимизировать. А *JIT-компиляция* происходит "на лету", в условиях жесткой ограниченности времени и ресурсов. Для решения этой проблемы были разработаны компиляторы программ *Java* в код конкретных программно-аппаратных платформ (*native* code – "родной" код). Например, свободно распространяемый фондом *GNU компилятор* gjc. Правда, заметные успехи *Sun* в усовершенствовании *Java*-машины позволили практически достичь, а в ряде случаев даже обогнать по быстродействию программы, написанные на других языках. В частности, приложения *Java*, активно занимающиеся выделением-высвобождением памяти, работают быстрее своих аналогов, написанных на C/C++, благодаря специальному механизму программных слотов памяти (*slot* – "паз, отверстие для вставки чего-либо").

Виртуальная *Java*-машина не только исполняет байт-код (интерпретирует его, занимается *JIT*-компиляцией и исполняет *JIT*-компилированный код), но и выполняет ряд других функций. Например, взаимодействует с операционной системой, обеспечивая *доступ* к файлам или поддержку графики. А также обеспечивает автоматическое высвобождение памяти, занятой ненужными объектами – так называемую *сборку мусора* (*garbage collection*). Программы *Java* можно разделить на несколько основных категорий:

- Приложение (application) – аналог "обычной" прикладной программы.
- Апплет (*applet*) – специализированная программа с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера.
- *Сервлет* (*servlet*) – специализированная программа с ограниченными возможностями, работающая в WWW на стороне сервера. Используется преимущественно в рамках технологии *JSP* (Java Server Pages – *Серверных Страниц Java*) для программирования WWW-документов со стороны сервера.
- *Серверное приложение* (*Enterprise application*) – предназначено для многократного использования на стороне сервера.
- Библиотека (Java *Class Library* – библиотека классов, либо NetBeans Module – модуль платформы NetBeans) – предназначена для многократного использования программами Java

Между приложениями и апплетами *Java* имеется принципиальное различие: *приложение* запускается непосредственно с компьютера пользователя и имеет *доступ* ко всем ресурсам компьютера наравне с любыми другими программами. *Апплет* же загружается из *WWW* с постороннего сервера, причем из-за самой идеологии *WWW сайт*, с которого загружен *апплет*, в общем случае не может быть признан надежным. А сам *апплет* имеет возможность передавать данные на произвольный *сервер* в *WWW*. Поэтому для того, чтобы избежать риска утечки конфиденциальной информации с компьютера пользователя или совершения враждебных действий у апплетов убраны многие возможности, имеющиеся у приложений.

Сервлеты – это приложения *Java* , запускаемые со стороны сервера. Они имеют возможности доступа к файловой системе и другим ресурсам сервера через набор *управляющих конструкций*, предопределенных в рамках технологии *JSP* и пакета javax.*servlet*. Технология *JSP* заключается в наличии дополнительных конструкций в *HTML*-или *XML*-документах, которые позволяют осуществлять вызовы сценариев ("скриптов"), написанных на языке *Java*. В результате удается очень просто и удобно осуществлять обработку данных или элементов документа, и внедрять в нужные места документа результаты обработки. Сценарии *Java* перед первым выполнением автоматически компилируются на стороне сервера, поэтому выполняемый код выполняется достаточно быстро. Но, конечно, требует, чтобы была установлена соответствующая *Java*-машина. Например, входящая в состав *Sun Application Server* – программного обеспечения, обеспечивающего поддержку большого количества необходимых серверных возможностей для работы в *WWW*. Отметим, что *Sun Application Server* также распространяется бесплатно и входит в комплект NetBeans *Enterprise* Pack.

Первоначально *Java* позиционировался *Sun* как язык, обеспечивающий развитые графические возможности *WWW*-документов благодаря включению в них апплетов. Однако в настоящее время основными областями использования *Java* является прикладное *программирование* на основе приложений, страниц *JSP* и сервлетов, а также других видов серверных программ. При этом использование апплетов играет незначительную роль.

Виртуальную *Java*-машину часто называют исполняющей средой (*Java Runtime Environment* – *JRE*).

Существует два основных способа установки *Java*-машины на клиентский *компьютер*:

- *JRE* из поставки *Software Development Kit* (*SDK*) – Комплекта разработки программного обеспечения.
- Специализированный вариант *JRE* в составе Интернет-браузера, называющийся Java plugin.

Комплект последних версий *SDK* можно свободно загружать с сайта *Sun* http://java.sun.com/.

При использовании апплетов требуется, чтобы в состав браузера входил специализированный комплект *JRE*. Как правило, он поставляется вместе с браузером, и может при необходимости обновляться. Для MS

Internet Explorer такой комплект и его обновления могут быть свободно загружены с сайта Microsoft.

Имеется возможность установки *Java*-машины от различных производителей, не обязательно устанавливать комплект *SDK* от *Sun*. На одном и том же компьютере может быть установлено сразу несколько различных *Java*-машин, в том числе комплекты *SDK* разных версий. Правда, *опыт* показывает, что при этом некоторые программы, написанные на *Java*, теряют работоспособность (частично или полностью).

Комплекты *SDK* имеют классификацию, опирающуюся на версию *Java* (языка программирования и, соответственно, *Java*-машины) и тип создаваемых приложений. Так, ко времени написания данного текста выходили версии *SDK* 1.0, 1.1, 1.2, 1.3, 1.4, 1.5 и 1.6. У каждой версии имеется ряд подверсий, не сопровождающихся изменением языка программирования, а связанных в основном с исправлением ошибок или внесением небольших изменений в библиотеки. Например, 1.4.1_01 или 1.5.0_04.

Версии *Java* 1.0 и 1.1 принято называть *Java* 1. Все версии *Java* начиная с 1.2 называют *Java* 2. Однако более надежно классифицировать по номеру *SDK*, так как язык *Java* для версии *SDK* 1.5 очень заметно отличается по возможностям от языка *Java* для более ранних версий *SDK* – в него добавлено большое количество новых синтаксических конструкций, а также изменен ряд правил. Поэтому код, правильный в *Java* для версии *SDK* 1.5, может оказаться неправильным в *Java* для версии *SDK* 1.4. Не говоря уж про *Java* для версии *SDK* 1.3 или 1.2. Кроме того, недавно компания *Sun* перестала использовать в названиях комплектов программного обеспечения термин *Java* 2 и происходящие от него сокращения вида j2.

Комплекты разработки *SDK* одной версии отличаются по типу создаваемых с их помощью приложений. Имеется три типа *SDK*:

Java ME – комплект Java *Micro* Edition (микро-издание) http://java.sun.com/j2me/, предназначенный для программирования "тонких аппаратных клиентов". То есть устройств, обладающих малыми ресурсами – наладонных компьютеров, сотовых телефонов, *микроконтроллеров*, *смарт-карт*. Старое название *J2ME*.

Java SE – комплект Java Standard Edition (стандартное издание) http://java.sun.com/j2se/, предназначенный для программирования "*толстых клиентов*". То есть устройств, обладающих достаточно большими ресурсами – обычных компьютеров. Старое название *J2SE*.

Java EE– комплект Java Enterprise Edition http ://java.sun.com/j2ee/, предназначенный для написания серверного программного обеспечения. Старое название *J2EE*.

При распространении какого-либо продукта, написанного на *Java*, возможна установка только программного обеспечения *Java*-машины (*JRE – Java Runtime Environment*). Например, в случае использования *Java* 1.4.1_01 – комплекта j2re1.4.1_01. При этом создается *папка* с именем j2re1.4.1_01 с *вложенными папками bin* и lib. В папке *bin* содержатся файлы и папки, необходимые для работы *Java*-машины и дополнительных инструментов для работы с ней в специальных режимах. В папке lib содержатся вспомогательные файлы и библиотеки, в основном связанные с параметрами настроек системы.

Также возможна установка целиком *SDK*. Например, при установке *SDK Java* SE 1.5.0_04 создается *папка* JDK1.5.0_04 с *вложенными папками bin*, *demo*, *include*, *jre*, lib, sample, а также архивом src.*zip* с исходными кодами стандартных классов *Java*. В папке *bin* содержатся файлы инструментов разработки, в папке *demo* – файлы примеров с исходными кодами. В папке *include* – *заголовки* файлов C для доступа к ряду библиотек *Java* и отладчику виртуальной *Java*-машины на платформо-зависимом уровне – на основе интерфейсов *JNI* (*Java Native Interface*) и JVMDI (*Java Virtual Machine Debugging Interface*), соответственно. В папке *jre* находятся файлы, необходимые для работы с виртуальной *Java*-машиной. *Папка* lib содержит ряд библиотек и сопроводительных файлов, необходимых для работы инструментов из папки *bin*. В папке sample находятся примеры с исходными кодами.

Аббревиатура *JDK* расшифровывается как *Java Development Kit* – комплект разработки программного обеспечения на *Java*. К сожалению, в комплекте отсутствует даже самая простейшая документация с описанием назначения имеющихся в нем инструментов – даны ссылки на *сайт* компании *Sun*, где можно найти эту информацию. Поэтому перечислим назначение основных инструментов. Они делятся на несколько категорий.

Таблица 1.1. Средства разработки приложений

Утилита	Назначение
javac	Компилятор в режиме командной строки для программ, написанных на языке Java
java	Утилита для запуска в режиме командной строки откомпилированных программ-приложений
appletviewer	Утилита для запуска на исполнение и отладку апплетов без браузера. При этом не гарантируется работоспособность отлаженного апплета в браузере.

jdb	Отладчик программ, написанных на языке Java
javadoc	Генератор документации по классам на основе комментариев, начинающихся с /**
jar	Создание и управление Java-архивами *jar*
javah	Генератор заголовочных файлов C/C++ для подключения к программам Java внешних библиотек C/C++ на основе интерфейса *JNI*
javap	*Дизассемблер* классов
extcheck	Утилита для обнаружения конфликтов между файлами архивов *jar*
native2ascii	Утилита для конвертации в режиме командной строки параметра, передаваемого в виде текста на национальном алфавите, в последовательность символов UNICODE.

Кроме этого, имеются средства поддержки работы в *WWW* и корпоративных сетях (*интранет*) с интерфейсом *RMI* – интерфейсом удаленного вызова методов. Это программы rmic, rmiregistry, rmid. Также имеются средства поддержки информационной безопасности keytool, jarsigner, policytool, и ряд файлов других категорий утилит.

Подчеркнем, что набор утилит *JDK* рассчитан на морально устаревший *режим командной строки*, и что гораздо удобнее и правильнее пользоваться современной профессиональной средой разработки NetBeans. Для ее работы из *JDK* необходим только комплект *JRE*.

1.3.Алфавит языка Java. Десятичные и шестнадцатеричные цифры и целые числа. Зарезервированные слова

Алфавит языка Java

Алфавит языка Java состоит из букв, десятичных цифр и специальных символов. Буквами считаются латинские буквы (кодируются в стандарте ASCII), буквы национальных алфавитов (кодируются в стандарте Unicode, кодировка *UTF-16*), а также соответствующие им символы, кодируемые *управляющими последовательностями* (о них будет рассказано чуть позже).

Буквы и цифры можно использовать в качестве идентификаторов (т.е. имен) переменных, методов и других *элементов языка* программирования. Правда, при использовании в идентификаторах национальных алфавитов в ряде случаев могут возникнуть проблемы – эти символы будут показываться в виде вопросительных знаков.

Как буквы рассматривается только часть символов национальных алфавитов. Остальные символы национальных алфавитов – это специальные символы. Они используются в качестве операторов и разделителей языка Java и не могут входить в состав идентификаторов.

Латинские буквы ASCII

ABCD...XYZ – заглавные (прописные) ,

abcd...xyz – строчные

Дополнительные "буквы" ASCII

_ – знак подчеркивания,

$ – знак доллара.

Национальные буквы на примере русского алфавита

АБВГ…ЭЮЯ – заглавные (прописные),

абвг…эюя – строчные

Десятичные цифры

0 1 2 3 4 5 6 7 8 9

Десятичные и шестнадцатеричные цифры и целые числа

Целые числовые константы в исходном коде Java (так называемые литерные константы) могут быть десятичными или шестнадцатеричными. Они записываются либо символами ASCII, или символами Unicode следующим образом.

Десятичные константы записываются как обычно. Например, -137.

Шестнадцатеричная константа начинается с символов 0x или 0X (цифра 0, после которой следует латинская буква X), а затем идет само число в шестнадцатеричной нотации. Например, 0x10 соответствует $10_{16} = 16$; 0x2F соответствует $2F_{16} = 47$, и т.д. О шестнадцатеричной нотации рассказано чуть ниже.

Ранее иногда применялись восьмеричные числа, и в языках C/C++, а также старых версиях Java можно было их записывать в виде числа, начинающегося с цифры 0. То есть 010 означало $10_8 = 8$. В настоящее время в программировании восьмеричные числа практически никогда не применяются, а неадекватное использование ведущего нуля может приводить к *логическим ошибкам* в программе.

Целая константа в обычной записи имеет тип `int`. Если после константы добавить букву L (или l, что хуже видно в тексте, хотя в среде разработки выделяется цветом), она будет иметь тип `long`, обладающий более широким диапазоном значений, чем тип `int`.

Поясним теперь, что такое шестнадцатеричная нотация записи чисел и зачем она нужна.

Информация представляется в компьютере в двоичном виде – как последовательность бит. Бит – это минимальная порция информации, он может быть представлен в виде ячейки, в которой хранится или ноль, или единица. Но бит – слишком мелкая единица, поэтому в компьютерах информация хранится, кодируется и передается байтами – порциями по 8 бит.

В данном курсе под "ячейкой памяти" будет пониматься непрерывная область памяти (с последовательно идущими адресами), выделенная программой для хранения данных. На рисунках мы будем изображать ячейку прямоугольником, внутри которого находятся хранящиеся в ячейке данные. Если у ячейки имеется имя, оно будет писаться рядом с этим прямоугольником.

Мы привыкли работать с числами, записанными в так называемой десятичной системе *счисления*. В ней имеется 10 цифр (от 0 до 9), а в числе имеются *десятичные разряды*. Каждый разряд слева имеет вес 10 по сравнению с предыдущим, то есть для получения значения числа, соответствующего цифре в каком-то разряде, стоящую в нем цифру надо умножать на 10 в соответствующей степени. То есть $52 = 5 \times 10 + 2, 137 = 1 \times 10^2 + 3 \times 10^1 + 7$, и т.п.

В программировании десятичной системой *счисления* пользоваться не всегда удобно, так как в компьютерах информация организована в виде бит, байт и более крупных порций. Человеку неудобно оперировать данными в виде длинных последовательностей нулей и единиц. В настоящее время в программировании стандартной является шестнадцатеричная система записи чисел. Например, с ее помощью естественным образом кодируется цвет, устанавливаются значения отдельных бит числа, осуществляется шифрование и дешифрование информации, и так далее. В этой системе *счисления* все очень похоже на десятичную, но только не 10, а 16 цифр, и вес разряда не 10, а 16. В качестве первых 10 цифр используются обычные десятичные цифры, а в качестве недостающих цифр, больших 9, используются заглавные латинские буквы A, B, C, D, E, F:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

То есть A=10, B=11, C=12, D=13, E=14, F=15.

Заметим, что в шестнадцатеричной системе *счисления* числа от 0 до 9 записываются одинаково, а превышающие 9 отличаются. Для чисел от 10 до 15 в шестнадцатеричной системе *счисления* используются буквы от A до F, после чего происходит использование следующего шестнадцатеричного разряда. Десятичное число 16 в шестнадцатеричной системе *счисления* записывается как 10. Для того, чтобы не путать числа, записанные в разных *системах счисления*, около них справа пишут индекс с указанием основания *системы счисления*. Для десятичной *системы счисления* это 10, для шестнадцатеричной 16. Для десятичной системы основание обычно не указывают, если это не приводит к путанице. Точно так же в технической литературе часто не указывают основание для чисел, записанных в шестнадцатеричной системе *счисления*, если в записи числа встречаются не только "обычные" цифры от 0 до 9, но и "буквенные" цифры от A до F. Обычно используют заглавные буквы, но можно применять и строчные.

Рассмотрим примеры.

$$0x10 = 10_{16} = 16;$$
$$0x100 = 100_{16} = 16 \times 16 = 256;$$
$$0x1000 = 1000_{16} = (16)^3 = 4096;$$
$$0x20 = 20_{16} = 2 \times 16 = 32;$$
$$0x21 = 21_{16} = 2 \times 16 + 1 = 33;$$

$$0xF = F_{16} = 15;$$
$$0x1F = 1F_{16} = 1 \times 16 + 15 = 31;$$
$$0x2F = 2F_{16} = 2 \times 16 + 15 = 47;$$
$$0xFF = FF_{16} = 15 \times 16 + 15 = 255;$$

Более подробно вопросы представления чисел в компьютере будут рассмотрены в отдельном разделе.

Зарезервированные слова языка Java

Это слова, зарезервированные для синтаксических конструкций языка, причем их назначение нельзя переопределять внутри программы.

Таблица 1.2.

abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	*enum*
extends	false	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	*native*
new	null	package	private	protected
public	return	short	static	*super*
switch	*synchronized*	this	throw	throws
transient	true	try	void	*volatile*
while				

Их нельзя использовать в качестве идентификаторов (имен переменных, подпрограмм и т.п.), но можно использовать в строковых выражениях.

1.4. Управляющие последовательности. Символы Unicode. Специальные символы

Управляющие последовательности

Управляющие последовательности - символы формирования текста

Иногда в тексте программы в строковых константах требуется использовать символы, которые обычным образом в текст программы ввести нельзя. Например, символы кавычек (их надо использовать внутри кавычек, что затруднительно), символ вопроса (зарезервирован для тернарного условного оператора), а также различные специальные символы. В этом случае используют управляющую последовательность - символ обратной косой черты, после которой следует один управляющий символ. В таблице приведены *управляющие последовательности*, применяющиеся в языке Java.

Таблица 1.3.

Символ	Что означает
\b	возврат на один символ назад
\f	перевод на новую страницу
\n	перевод на новую строку
\r	возврат к началу строки
\t	горизонтальная табуляция
\v	вертикальная табуляция
\'	кавычка
\"	двойные кавычки
\\	обратная косая черта

\? вопросительный знак

\u начало *кодировки символа* Unicode

Управляющие последовательности – символы Unicode

Управляющая последовательность может содержать несколько символов. Например, символы национальных алфавитов могут кодироваться последовательностью "\u" , после которой идет код символа в шестнадцатеричной кодировке для кодовых таблиц *UTF-16* или UTF-8.

Например:

\u0030 - \u0039 – цифры ISO-*LATIN* от 0 до 9

\u0024 – знак доллара $

\u0041 - \u005a – буквы от A до Z

\u0061 - \u007a – буквы от a до z

Простые специальные символы

Таблица 1.4.

+	Оператор сложения
-	Оператор вычитания
*	Оператор умножения
/	Оператор деления
%	Оператор остатка от целочисленного деления
=	Оператор присваивания
~	Оператор побитового дополнения (побитовое "не")
?	Вопросительный знак – часть тернарного (состоящего из трех частей) условного оператора "? :"
:	Двоеточие – часть условного оператора "? :" . Также используется для задания метки – ставится после имени метки.
^	Оператор "исключающее или" (XOR)
&	Оператор "побитовое и" (AND)
|	Оператор "побитовое или" (OR)
!	Оператор "НЕ"
>	Больше
<	Меньше
{	Левая фигурная скобка – открытие блока кода
}	Правая фигурная скобка – закрытие блока кода
.	Точка – десятичный разделитель в числовом литерном выражении; разделитель в составном имени для доступа к элементу пакета, класса, объекта, интерфейса
(	Левая круглая скобка – используется для открытия списка параметров в операторах и для открытия группируемой части в выражениях
)	Правая круглая скобка– используется для закрытия списка параметров в операторах и для закрытия группируемой части в выражениях
[	Левая квадратная скобка – открытие *индекса массива*
]	Правая квадратная скобка – закрытие *индекса массива*
;	Точка с запятой – окончание оператора
'	Апостроф (одиночная кавычка) – открытие и закрытие символа
"	Двойные кавычки – открытие и закрытие строки символов
\	обратная косая черта (*backslash*) – используется для задания *управляющих последовательностей* символов
	знак пробела (невидимый)
	знак табуляции (невидимый)
@	Коммерческое а ("эт") – знак начала метаданных
#	– не имеет специального назначения
¤	– не имеет специального назначения
"	– не имеет специального назначения

, Запятая - разделитель в списке параметров оператора; разделитель в *составном операторе*	" - не имеет специального назначения № - не имеет специального назначения § - не имеет специального назначения

Составные специальные символы

Таблица 1.5.

Символ	Что означает
++	Оператор *инкремента* (увеличения на 1); x++ эквивалентно x=x+1
--	Оператор *декремента* (уменьшения на 1); x-- эквивалентно x=x-1
&&	Оператор "логическое И" (AND)
||	Оператор " *логическое ИЛИ*" (OR)
<<	Оператор левого побитового сдвига
>>>	Оператор беззнакового правого побитового сдвига
>>	Оператор правого побитового сдвига с сохранением знака отрицательного числа
==	Равно
!=	не равно
+=	y+=x эквивалентно y=y+x
-=	y-=x эквивалентно y=y-x
*=	y*=x эквивалентно y=y*x
/=	y/=x эквивалентно y=y/x
%=	y%=x эквивалентно y=y%x
|=	y|=x эквивалентно y=y|x
^=	y^=x эквивалентно y=y^x
>>=	y>>=x эквивалентно y= y>>x
>>>=	y>>>=x эквивалентно y= y>>>x
<<=	y<<=x эквивалентно y= y<<x
/*	Начало многострочного комментария.
/**	Начало многострочного комментария, предназначенного для автоматического создания документации по классу.
*/	Конец многострочного комментария (открываемого как /* или /**).
//	Однострочный комментарий

1.5.Идентификаторы. Переменные и типы. Примитивные и ссылочные типы

Идентификаторы - это имена переменных, подпрограмм-функций и других *элементов языка* программирования. В идентификаторах можно применять только буквы и цифры, причем первой всегда должна быть буква (в том числе символы подчеркивания и доллара), а далее может идти произвольная комбинация букв и цифр. Некоторые символы национальных алфавитов рассматриваются как буквы, и их можно применять в идентификаторах. Но некоторые используются в качестве символов-разделителей, и в идентификаторах их использовать нельзя.

Язык *Java* является *регистро-чувствительным*. Это значит, что идентификаторы чувствительны к тому, в каком регистре (верхнем или нижнем) набираются символы. Например, имена i1 и I1 соответствуют разным идентификаторам. Это правило привычно для тех, кто изучал языки C/C++, но может на первых порах вызвать сложности у тех, кто изучал язык PASCAL, который является регистро-нечувствительным.

Длина идентификатора в *Java* любая, по крайней мере, в пределах разумного. Так, даже при длине идентификатора во всю ширину экрана *компилятор* NetBeans правильно работает.

Переменная – это именованная *ячейка* памяти, содержимое которой может изменяться. Перед тем, как использовать какую-либо переменную, она должна быть задана в области программы, предшествующей месту, где эта *переменная* используется. При объявлении переменной сначала указывается *тип переменной*, а затем *идентификатор* задаваемой переменной. *Указание типа* позволяет компилятору задавать размер ячейки (объем памяти, выделяемой под переменную или *значение* данного типа), а также допустимые правила действий с переменными и значениями этого типа. В *Java* существует ряд предопределенных типов: `int` – *целое число*, `float` – *вещественное число*, `boolean` – логическое *значение*, `Object` – самый простой *объектный тип* (*класс*) *Java*, и т.д. Также имеется возможность задавать собственные *объектные типы* (классы), о чем будет рассказано позже.

Объявление переменных `a1` и `b1`, имеющих некий тип `MyType1`, осуществляется так:

```
MyType1 a1,b1;
```

При этом `MyType1` – имя типа этих переменных.

Другой пример – *объявление переменной* `j` типа `int`:

```
int j;
```

Типы бывают предопределенные и пользовательские. Например, `int` – предопределенный тип, а `MyType1` – пользовательский. Для объявления переменной не требуется никакого зарезервированного слова, а имя типа пишется перед именами задаваемых переменных.

Объявление переменных может сопровождаться их инициализацией – присваиванием начальных значений. Приведем пример такого объявления целочисленных переменных `i1` и `i2`:

```
int i1=5;
int i2=-78;
```

либо

```
int i1=5,i2=-78;
```

Присваивания вида `int i1=i2=5;`, характерные для C/C++, запрещены.

Для начинающих программистов отметим, что символ `"="` используется в *Java* и многих других языках в качестве символа *присваивания*, а не символа равенства, как это принято в математике. Он означает, что *значение*, стоящее с правой стороны от этого символа, *копируется* в переменную, стоящую в левой части. То есть, например, *присваивание* **b=a** означает, что в переменную (ячейку) с именем **b** надо скопировать *значение* из переменной (ячейки) с именем **a**. Поэтому неправильное с точки зрения математики *выражение*

x=x+1

в программировании вполне корректно. Оно означает, что надо взять *значение*, хранящееся в ячейке с именем **x**, прибавить к нему 1 (это будет происходить где-то вне ячейки **x**), после чего получившийся результат записать в ячейку **x**, заменив им прежнее *значение*.

После объявления переменных они могут быть использованы в выражениях и присваиваниях:

```
переменная=значение;
переменная=выражение;
переменная1= переменная2;
```

и так далее. Например,

```
i1=i2+5*i1;
```

Примитивными типами называются такие, для которых данные содержатся в одной ячейке памяти, и эта *ячейка* не имеет подъячеек.

Ссылочными типами называются такие, для которых в ячейке памяти (*ссылочной переменной*) содержатся не сами данные, а только адреса этих данных, то есть *ссылки* на данные. При присваивании в ссылочную переменную заносится новый *адрес*, а не сами данные. Но непосредственного доступа к адресу, хранящемуся в ссылочных переменных, нет. Это сделано для обеспечения безопасности работы с данными – как с точки зрения устранения непреднамеренных ошибок, характерных для работы с данными по их адресам в языках C/C++/PASCAL, так и для устранения возможности намеренного взлома информации.

Если ссылочной переменной не присвоено ссылки, в ней хранится нулевой *адрес*, которому дано *символическое имя* `null`. Ссылки можно присваивать друг другу, если они совместимы по типам, а также присваивать *значение* `null`. При этом из одной ссылочной переменной в другую копируется *адрес*. Ссылочные переменные можно сравнивать на *равенство*, в том числе на *равенство* `null`. При этом сравниваются не данные, а их адреса, хранящиеся в ссылочных переменных.

В *Java* все типы делятся на примитивные и ссылочные. К *примитивным типам* относятся следующие предопределенные типы: *целочисленные типы* `byte`, `short`, `int`, `long`, `char`, типы данных в формате с плавающей точкой `float`, `double`, а также булевский (*логический) тип* `boolean` и типы-перечисления, объявляемые с помощью зарезервированного слова `enum` (сокращение от *enumeration* - "*перечисление*"). Все остальные типы *Java* являются ссылочными.

В *Java* действуют следующие соглашения о регистре букв в идентификаторах:

Имена примитивных типов следует писать в нижнем регистре (строчными буквами). Например, `int, float, boolean` и т.д.

Имена ссылочных типов следует начинать с заглавной (большой) буквы, а далее для имен, состоящих из одного слова, писать все остальные буквы в нижнем регистре. Например, `Object, Float, Boolean, Collection, Runnable`. Но если имя составное, новую часть имени начинают с заглавной буквы. Например, `JButton, JTextField, JFormattedTextField, MyType` и т.д. Обратите внимание, что типы `float` и `Float`, `boolean` и `Boolean` различны - язык Java чувствителен к регистру букв!

Для переменных и методов имена, состоящие из одного слова, следует писать в нижнем регистре. Например, `i, j, object1`. Если имя составное, новую часть имени начинают с заглавной буквы: `myVariable, jButton2, jTextField2.getText()` и т.д.

Имена констант следует писать в верхнем регистре (большими буквами), разделяя входящие в имя составные части символом подчеркивания "_". Например, `Double.MIN_VALUE, Double.MAX_VALUE, JOptionPane.INFORMATION_MESSAGE, MY_CHARS_COUNT` и т.п.

Символ подчеркивания "_" рекомендуется использовать для разделения составных частей имени только в именах констант и пакетов.

Переменная примитивного типа может быть отождествлена с ячейкой, в которой хранятся данные. У нее всегда есть имя. *Присваивание* переменной примитивного типа меняет *значение* данных. Для ссылочных переменных действия производятся с адресами ячеек, в которых хранятся данные, а не с самими данными.

Для чего нужны такие усложнения? Ведь человеку гораздо естественнее работать с ячейками памяти, в которых хранятся данные, а не адреса этих данных. Ответ заключается в том, что в программах часто требуются динамически создаваемые и уничтожаемые данные. Для них нельзя заранее создать необходимое число переменных, так как это число неизвестно на этапе написания программы и зависит от выбора пользователя. Такие данные приходится помещать в динамически создаваемые и уничтожаемые ячейки. А с этими ячейками удается работать только с помощью ссылочных переменных.

Ссылочные типы Java используются в объектном программировании. В частности, для работы со строками, файлами, элементами пользовательского интерфейса. Все *пользовательские типы* (задаваемые программистом) , кроме типов-перечислений, являются ссылочными. В том числе - *строковые типы*.

Краткие итоги

Алфавит языка Java состоит из букв, десятичных цифр и специальных символов. Буквами считаются латинские буквы (кодируются в стандарте ASCII), буквы национальных алфавитов (кодируются в стандарте Unicode), а также соответствующие им символы, кодируемые *управляющими последовательностями*.

В программах разрешается пользоваться десятичными и шестнадцатеричными целыми числовыми константами. Шестнадцатеричная константа начинается с символов 0x или 0X, после чего идет само число в шестнадцатеричной нотации.

Java - универсальный язык *прикладного программирования*, JavaScript - узкоспециализированный язык программирования HTML-документов, C++ - универсальный язык *системного программирования*. Java - компилируемый, платформонезависимый, объектно-ориентированный язык с C-образным синтаксисом.

Программы Java переносимы как на уровне исходных кодов, так и на уровне скомпилированных исполняемых кодов - байт-кода. Байт-код является платформонезависимым, так как не содержит инструкций процессора конкретного компьютера. Он интерпретируется виртуальной Java-машиной (*JVM*).

JIT-компиляция (Just-In-Time) - компиляция байт-кода в код конкретной платформы в момент выполнения программы, то есть "по ходу дела", "налету". Она позволяет ускорить работу программ за счет замены интерпретации байт-кода на выполнение скомпилированного кода.

Основные категории программ Java:

Приложение (application) - аналог "обычной" прикладной программы.

Апплет (*applet*) – специализированная программа, работающая в окне WWW-документа под управлением браузера.

Сервлет (*servlet*) – специализированная программа, работающая в WWW на стороне сервера

Модуль *EJB* (Enterprise *JavaBeans*) – предназначен для многократного использования *серверными приложениями* Java

Библиотека – предназначена для многократного использования программами классов Java

Версии Java 1.0 и 1.1 принято называть Java 1. Все версии Java начиная с 1.2 принято называть Java 2.

Поставить на компьютер исполняющую среду Java (виртуальную Java-машину) можно путем установки *SDK* (*Software Development Kit*) – Комплекта разработки программного обеспечения. Имеется три типа *SDK*:

Java ME – комплект Java *Micro* Edition, предназначенный для программирования "тонких аппаратных клиентов".

Java SE – комплект Java Standard Edition, предназначенный для программирования обычных компьютеров.

Java EE– комплект Java Enterprise Edition, предназначенный для написания серверного программного обеспечения.

Язык Java является регистро-чувствительным. Исходные коды программ Java набираются в виде последовательности символов Unicode.

Управляющая последовательность применяется в случае, когда требуется использовать символ, который обычным образом в текст программы ввести нельзя. Простая *управляющая последовательность* начинается с символа `"\"` , после которого идет управляющий символ. *Управляющая последовательность* для кодирования символа Unicode начинается с последовательности из двух символов `-"\u"` , после которой следует четыре цифры номера символа в шестнадцатеричной нотации. Например, `\u1234.`

Специальные символы используются в качестве операторов и разделителей языка Java и не могут входить в состав идентификаторов. Специальные символы бывают простые и составные. Они используются в операторах, для форматирования текста и как разделители.

Идентификаторы – это имена переменных, процедур, функций и т.д. В идентификаторах можно применять только буквы и цифры, причем первой всегда должна быть буква, а далее может идти произвольная комбинация букв и цифр. Длина идентификатора в Java любая.

Переменная – это именованная ячейка памяти, содержимое которой может изменяться. При объявлении переменной сначала указывается тип переменной, а затем идентификатор задаваемой переменной.

Типы в Java делятся на примитивные и ссылочные. Существует несколько предопределенных примитивных типов, все остальные – ссылочные. Все *пользовательские типы* кроме типов-перечислений являются ссылочными. Значение `null` соответствует ссылочной переменной, которой не назначен адрес ячейки с данными.

Типичные ошибки:

Путают языки Java и JavaScript, либо считают, что JavaScript – это интерпретируемый вариант Java. Хотя эти языки не имеют друг к другу никакого отношения.

Ошибочно считают, что приложение Java может быть запущено на любом компьютере без установки исполняющей среды (*JRE*).

Не различают приложения (applications) и апплеты (applets).

При записи шестнадцатеричного числа вида `0x…` вместо ведущего нуля пишут букву O.

Ошибочно считают, что в идентификаторах Java нельзя использовать символы национальных алфавитов.

Ошибочно считают, что не имеет значения, в каком регистре набраны символы идентификатора (характерно для тех, кто раньше программировал на PASCAL или FORTRAN).

Задания

Написать в 16-ричном виде числа 0, 1, 8, 15, 16, 255, 256.

Дать ответ, являются ли допустимыми идентификаторами `i1` , `i_1` , `1i` , `i&1` , `i1234567891011` , `IJKLMN?`

Являются ли допустимыми и различными идентификаторы `myObject` , `MyObject` , `myobject` , `Myobject` , `my object` , `my_object?`

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.