3-ЛЕКЦИЯ. СРЕДА ПРОГРАММИРОВАНИЯ ДЛЯ СОЗДАНИЯ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

3.1. Платформы операционной системы мобильных приложений

Средства разработки программного обеспечения для мобильных приложений:

- JDK — помимо набора библиотек для платформ Java SE и Java EE, содержит компилятор командной строки javac и набор утилит, также работающих в режиме командной строки.
- NetBeans IDE — свободная интегрированная среда разработки для всех платформ Java — Java ME, Java SE и Java EE. Пропагандируется Oracle, владельцем технологии Java, как базовое средство для разработки ПО на языке Java и других языках (C, C++, PHP, Fortran и др.).
- Eclipse IDE — свободная интегрированная среда разработки для Java SE, Java EE и Java ME. Пропагандируется IBM, одним из важнейших разработчиков корпоративного ПО, как базовое средство для разработки ПО на языке Java и других языках (C, C++, Ruby, Fortran и др.)
- IntelliJ IDEA — среда разработки для платформ Java SE, Java EE и Java ME. Разработчик — компания JetBrains. Распространяется в двух версиях: свободной бесплатной (Community Edition) и коммерческой проприетарной (Ultimate Edition).
- JDeveloper — среда разработки для платформ Java SE, Java EE и Java ME. Разработчик — компания Oracle.
- BlueJ — среда разработки программного обеспечения на языке Java, созданная в основном для использования в обучении, но также подходящая для разработки небольших программ.
- Java для iOS — обучающее приложение и компилятор для iOS.
- Geany — свободная среда разработки программного обеспечения, написанная с использованием библиотеки GTK2

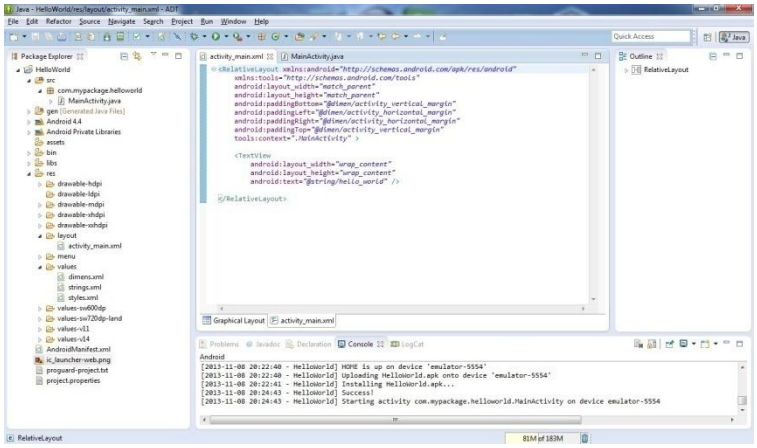
Java является одним из лидеров по количеству обнаруживаемых уязвимостей, наряду с Adobe Flash Player.

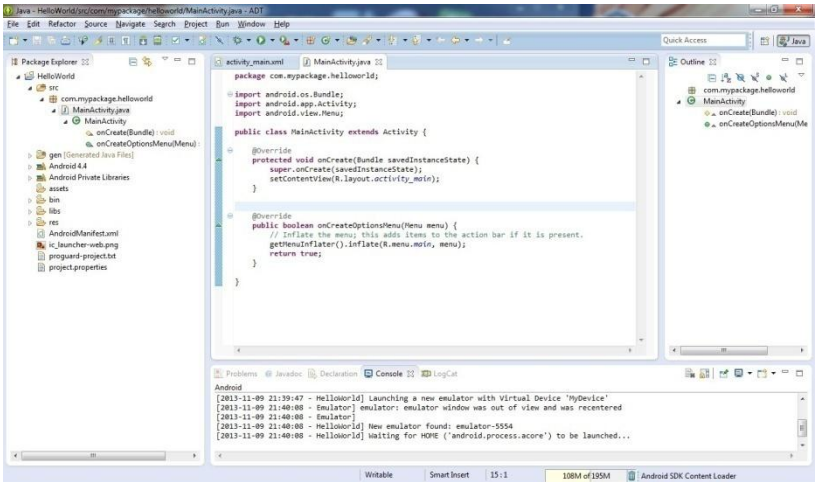
Android IDE

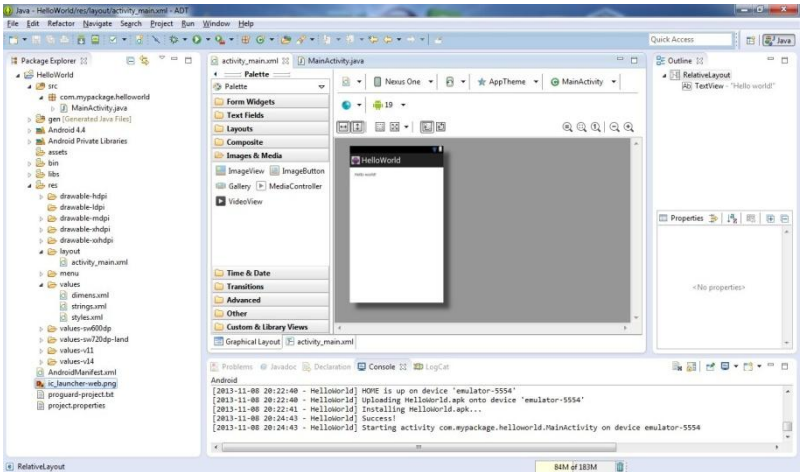
Реализован полный цикл разработки редактирование-компиляция-выполнение. Автодополнение кода, проверка ошибок в реальном времени, навигация по коду и запуск вашего приложения в одно касание.

Возможна разработка настоящих приложений для Андроида прямо на устройствах с ОС Андроид:

- Андроид-планшет с клавиатурой может стать полноценным местом разработки
- Можно просматривать и редактировать код прямо на смартфоне
- Поддерживает разработку с использованием Java/Xml и C/C++
- Полностью совместима с проектами Eclipse
- Поддерживает профессиональную разработку приложений







Android Studio (IntelliJ IDEA)

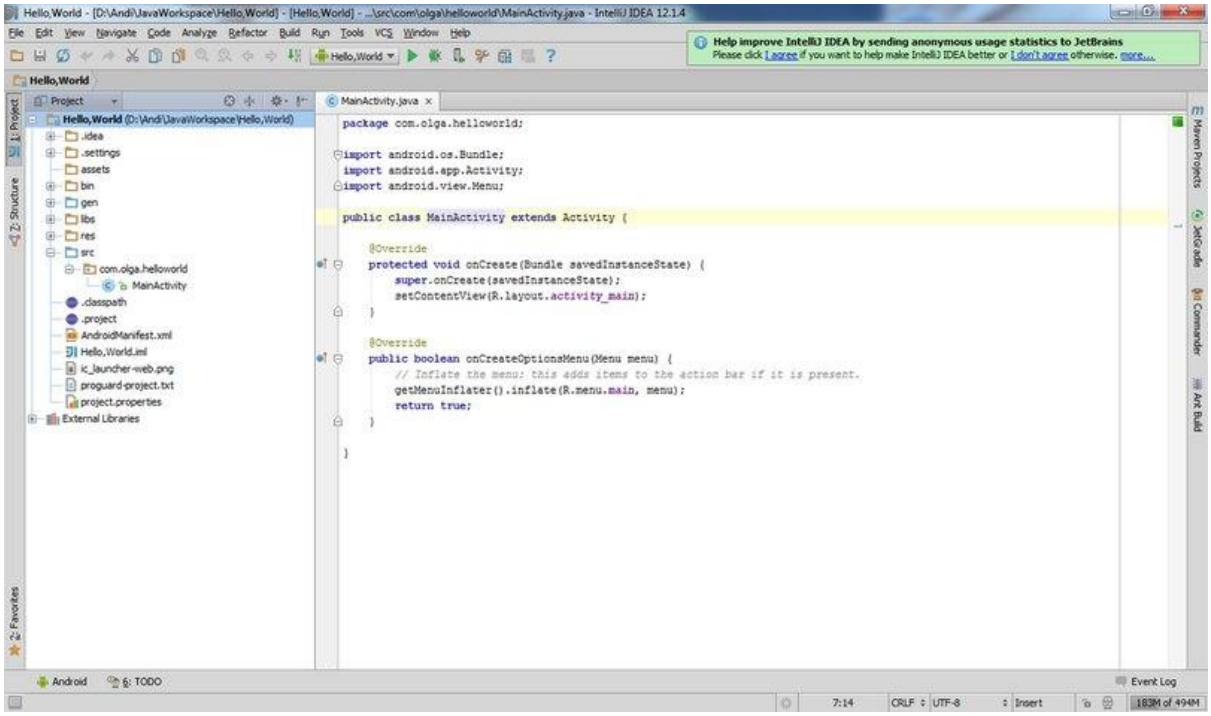
• Позволяет редактировать приложение в реальном времени, отображая его поведение одновременно на устройствах с различными диагоналями экранов

• Доступно мгновенное переключение на различные типы верстки и размеры экранов

• Раздел с подсказками и советами по оптимизации с тематическими разделами

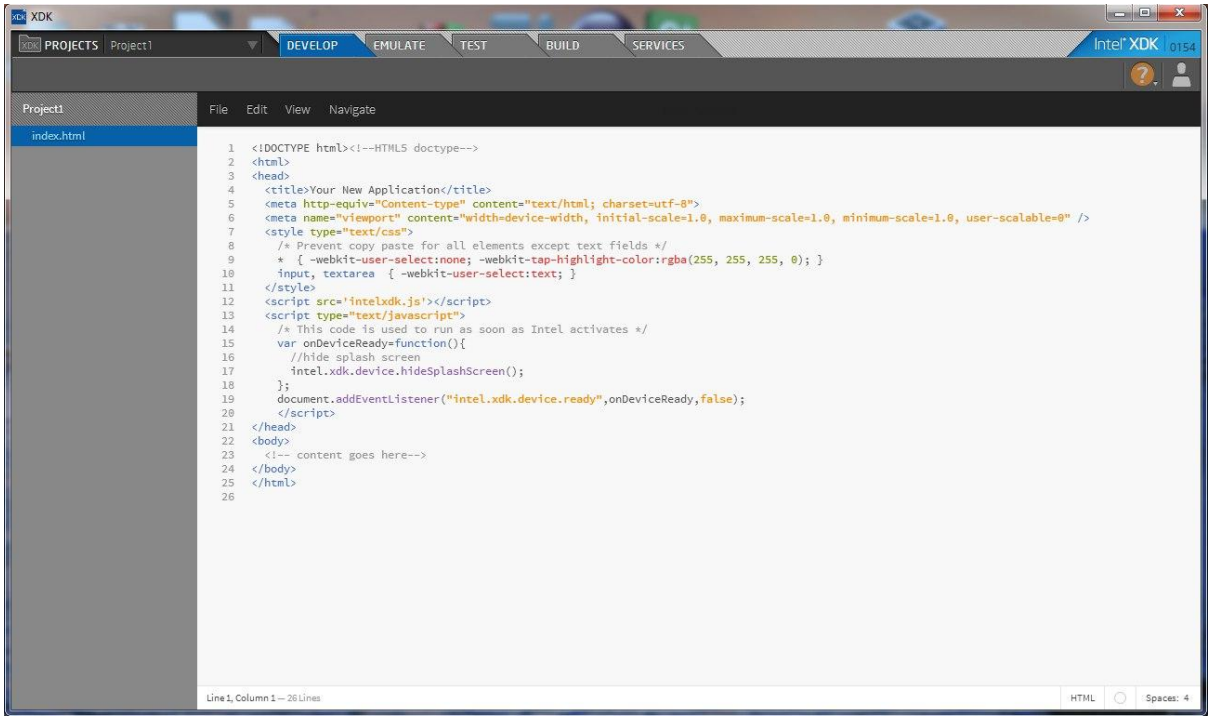
• Средство взаимодействия с бета-тестерами

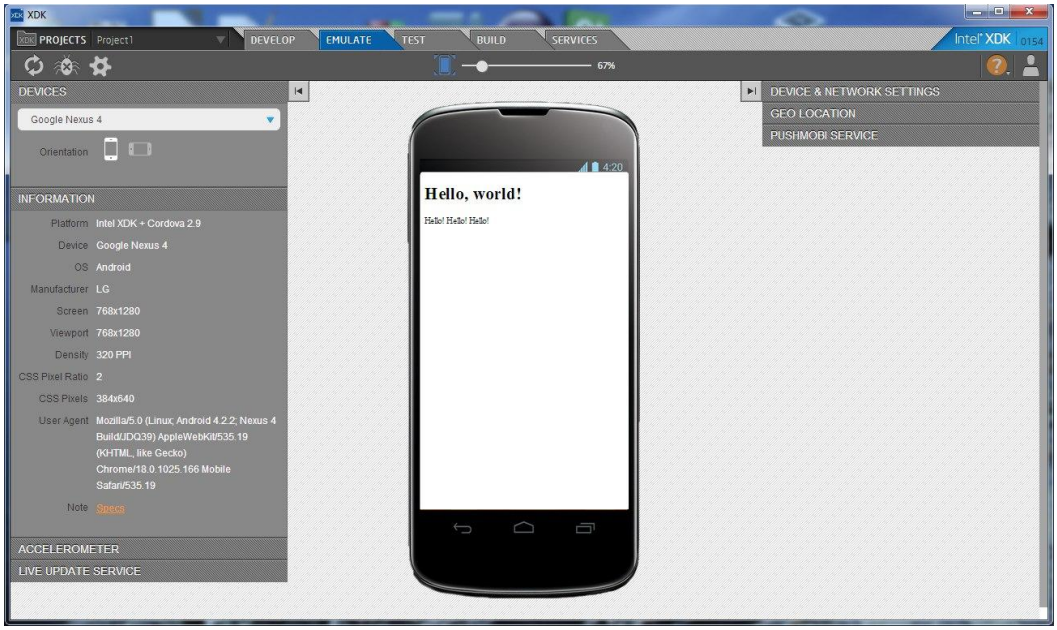
• Позволяет ускорить процесс разработки софтверных изделий, сделав его более продуктивным



Intel XDK

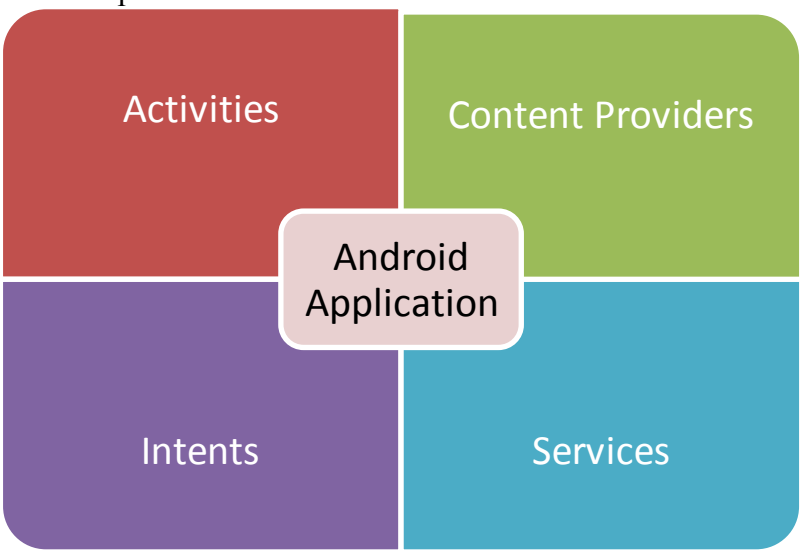
- Позволяет легко разрабатывать кроссплатформенные приложения
- Включает в себя инструменты для создания, отладки и сборки ПО, а также эмулятор устройств
- Поддерживает разработку для Android, Apple iOS, Microsoft Windows 8
- Языки разработки HTML5 и JavaScript



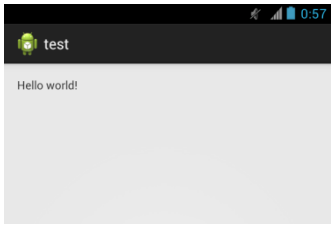


Google Android SDK (ADT Bundle)

Состав приложения



Activity – основная единица графического интерфейса (аналог окна или экранной формы)



Content Providers
Управляет распределенным множеством данных приложения
Например:

Контент-провайдер в системе Android, управляющий информацией о контактах пользователя

- Данные могут храниться в файловой системе, в базе данных SQLite, в сети
- Позволяет другим приложениям при наличии у них соответствующих прав делать запросы или даже менять данные

Intent

Intents – системные сообщения, позволяющие приложениям обмениваться информацией между собой и с операционной системой:

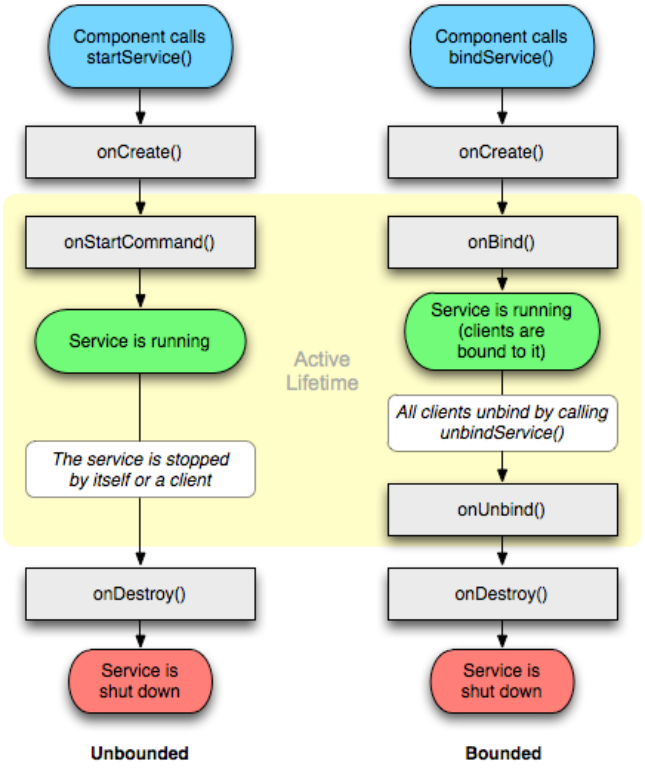
- поступление телефонного звонка
- приход sms-сообщения
- вставлена SD-карта
- запущена новая активность
- Intents – рекомендованный способ взаимодействия компонентов приложения.

Services

Приложения, не имеющие GUI и выполняющиеся в фоновом режиме.

Примеры сервисов:

- проверка электронной почты
- получение гео-информации



Marmalade SDK

Кроссплатформенное SDK от Ideaworks3D Limited. Представляет собой набор библиотек, образцов, инструментов и документаций необходимых для разработки, тестирования и развертывания приложений для мобильных устройств. Используется для разработки игр.

3.2. Язык Java для операционной системы Android

Java — объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой виртуальной Java-машине вне зависимости от компьютерной архитектуры. Дата официального выпуска — 23 мая 1995 года.

Изначально язык назывался Oak («Дуб») разрабатывался Джеймсом Гослингом для программирования бытовых электронных устройств. Впоследствии он был переименован в Java и стал использоваться для написания клиентских приложений и серверного программного обеспечения. Назван в честь марки кофе Java, которая, в свою очередь, получила наименование одноимённого острова (Ява), поэтому на официальной эмблеме языка изображена чашка с дымящимся кофе. Существует и другая версия происхождения названия языка, связанная с аллюзией на кофе-машину как пример бытового устройства, для программирования которого изначально язык создавался.

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java:

- применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,
- широкое использование платформенно-ориентированного кода (native-код) в стандартных библиотеках,
- аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами фирмы ARM).

По данным сайта shootout.alioth.debian.org, для семи разных задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее. С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++. Также примечательно исследование, проведённое компанией Google, согласно которому отмечается существенно более низкая производительность и бóльшее потребление памяти в тестовых примерах на Java в сравнении с аналогичными программами на C++.

Идеи, заложенные в концепцию и различные реализации среды виртуальной машины Java, вдохновили множество энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине. Эти идеи нашли также выражение в спецификации общеязыковой инфраструктуры CLI, заложенной в основу платформы .NET компанией Microsoft.

Java 1.0

Разработка Java началась в 1990 году, первая официальная версия — Java 1.0, — была выпущена только 26 августа 1996 года.

Java 1.2

К 1998 году была разработана обновлённая спецификация JDK 1.2, вышедшая под наименованием Java 2. Язык практически не изменился — было добавлено одно ключевое слово strictfp. Платформа получила следующие дополнения:

- Библиотека Swing для создания графического интерфейса пользователя .
- Коллекции (JSR 166).
- Поддержка файлов Policy и цифровых сертификатов пользователя.
- Библиотека Accessibility.
- Java 2D.
- Поддержка технологии drag-and-drop.
- Полная поддержка Unicode, включая поддержку ввода на японском, китайском и корейском языках.
- Поддержка воспроизведения аудиофайлов нескольких популярных форматов.
- Полная поддержка технологии CORBA.
- JIT-компилятор, улучшенная производительность.
- Усовершенствования инструментальных средств JDK, в том числе поддержка профилирования Java-программ.

Java 2

В данном случае встречается путаница. Выпускались книги, например, Beginning Java 2 by Ivor Horton (Mar 1999), фактически по JDK 1.2 (бывшее название — Java 2). Вместе с тем по сей день такие книги публикуются, например: Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. Технологии программирования на Java 2. Распределенные приложения (2011).

В то время, когда, как известно, Java 2 была исторически заменена следующими релизами, подобные названия книг дезориентируют в понимании, о какой же версии Java они написаны на самом деле. Если JDK 1.2 принято считать за Java 2, а авторы книг за Java 2 принимают JDK 7, это приводит к полной путанице.

Java 5.0

Спецификация Java 5.0 была выпущена в сентябре 2004 года. С этой версии изменена официальная индексация, вместо Java 1.5 правильнее называть Java 5.0. Внутренняя же индексация Sun осталась прежней — 1.x. Минорные изменения теперь включаются без изменения индексации, для этого используется слово «Update» или буква «u», например, Java Development Kit 5.0 Update 22. Предполагается, что в обновления могут входить как исправления ошибок, так и небольшие добавления в API, JVM.

В данной версии разработчики внесли в язык целый ряд принципиальных дополнений:

- Перечислимые типы (англ. *enum*). Ранее отсутствовавшие в Java типы оформлены по аналогии с C++, но при этом имеют ряд дополнительных возможностей.
  - Перечислимый тип является полноценным классом Java, то есть может иметь конструктор, поля, методы, в том числе скрытые и абстрактные.
  - Перечисление может реализовывать интерфейсы.
  - Для перечислений имеются встроенные методы, дающие возможность получения значений типа по имени, символьных значений, соответствующих именам, преобразования между номером и значением, проверки типа на то, что он является перечислимым.
- Аннотации — возможность добавления в текст программы метаданных, не влияющих на выполнение кода, но допускающих использование для получения

различных сведений о коде и его исполнении. Одновременно выпущен инструментарий для использования аннотированного кода. Одно из применений аннотаций — упрощение создания тестовых модулей для Java-кода.

- Средства обобщённого программирования (англ. *generics*) — механизм, аналогичный Eiffel (позже также появились и в C#, принципиально отличаются от шаблонов C++), дающий возможность создавать классы и методы с полями и параметрами произвольного объектного типа. С использованием данного механизма реализованы новые версии коллекций стандартной библиотеки Java.
- Методы с неопределённым числом параметров.
- Autoboxing/Unboxing — автоматическое преобразование между скалярными типами Java и соответствующими типами-обёртками (например, между int — Integer). Наличие такой возможности сокращает код, поскольку исключает необходимость выполнения явных преобразований типов в очевидных случаях.
- Разрешён импорт статических полей и методов.
- В язык введён цикл по коллекции объектов (итератор, англ. *foreach*).
- Было введено использование Javadoc-комментариев, которые используются для автоматического оформления документации по комментариям в исходном коде.

Java 6

Релиз версии состоялся 11 декабря 2006 года. Изменена официальная индексация — вместо ожидаемой 6.0 версия значится как 6. Минорные изменения, как и в Java 5.0, вносятся в обычные обновления версии, например, Java Standard Edition Development Kit 6 Update 27. Внесены следующие изменения:

- Коллекции — добавлены интерфейсы для организации очереди, работающей с двух сторон коллекции; организовывающие поиск по ближайшему соответствию; блокирующие себя во время ожидания элемента. Организованы новые классы, реализующие перечисленные интерфейсы.
- Добавлена поддержка японского императорского календаря (наряду с уже существующими григорианским и буддийским календарями).
- Доступны классы-потоки для чтения и передачи сжатых данных, с возможностью передачи их по сети. Сняты ограничения на количество файлов в архиве (ранее 64 Кб), длину названия файла (ранее 256 символов) и количество одновременно открытых файлов (ранее 2000 шт).
- Организована система управления кэшем и добавлена поддержка параметра «no-cache» в HTTP-запросе.
- JConsole, графический мониторинг JVM, стала официально поддерживаемой утилитой.
- Java HTTP Server, позволяет создать полноценный HTTP сервер, с минимально необходимыми функциональными свойствами.
- Повысилась скорость вычислений на 70 %, скорость операций ввода-вывода возросла в два раза.
- Swing — улучшена работоспособность OpenGL и DirectX; обработка текста на LCD; добавлен GifWriter, для работы с файлами .gif.
- Исправлено большое количество ошибок.

Java 7

Релиз версии состоялся 28 июля 2011 года. В финальную версию Java Standard Edition 7 не были включены все ранее запланированные изменения. Согласно плану развития (план «Б»), включение нововведений будет разбито на две части: Java Standard Edition 7 (без лямбда-исчисления, проекта Jigsaw, и части улучшений Coin) и Java Standard Edition 8 (все остальное), намеченный на конец 2012 года.

В новой версии, получившей название Java Standard Edition 7 (Java Platform, Standard Edition 7), помимо исправления большого количества ошибок, было представлено

несколько новшеств. Так, например, в качестве эталонной реализации Java Standard Edition 7 использован не проприетарный пакет JDK, а его открытая реализация OpenJDK, а сам релиз новой версии платформы готовился при тесном сотрудничестве инженеров Oracle с участниками мировой экосистемы Java, комитетом JCP (Java Community Process) и сообществом OpenJDK. Все поставляемые Oracle бинарные файлы эталонной реализации Java Standard Edition 7 собраны на основе кодовой базы OpenJDK, сама эталонная реализация полностью открыта под лицензией GPLv2 с исключениями GNU ClassPath, разрешающими динамическое связывание с проприетарными продуктами. К другим нововведениям относится интеграция набора небольших языковых улучшений Java, развиваемых в рамках проекта Coin, добавлена поддержка языков программирования с динамической типизацией, таких, как Ruby, Python и JavaScript, поддержка загрузки классов по URL, обновленный XML-стек, включающий JAXP 1.4, JAXB 2.2a и JAX-WS 2.2 и другие.

За 5 дней до выхода релиза Java Standard Edition 7 было обнаружено несколько серьёзных ошибок в горячей оптимизации циклов, которая включена по умолчанию и приводит виртуальную машину Java к краху. Специалисты Oracle найденные ошибки за столь короткий срок исправить не могли, но пообещали, что они будут исправлены во втором обновлении (Java 7 Update 2) и частично в первом.

Список нововведений

- Поддержка динамически-типизированных языков (InvokeDynamic) — расширение JVM (семантики байт-кода), языка Java для поддержки динамически-типизированных языков.
- Строгая проверка class-файлов — class-файлы версии 51 (Java Standard Edition 7) или более поздней версии должны быть проверены typechecking-верификатором; JVM не должна переключаться на старый верификатор.
- Изменение синтаксиса языка Java (Project Coin) — частичные изменения в языке Java, предназначенные для упрощения общих задач программирования:
  - Использование класса String в блоке switch.
  - Закрытие используемых ресурсов в блоке try (try-with-resources) — работает при использовании интерфейса AutoCloseable.
  - Объединённая обработка исключений в блоке catch (multi-catch exceptions) — перечисление обрабатываемых исключений в catch (… | … | …).
  - Повторное выбрасывание исключений (rethrowing exceptions) — передача возникшего исключения «вверх» по стеку вызовов.
  - Подчёркивания в числовых литералах для лучшего восприятия больших чисел.
  - Изменение вывода типа в Java generic при создании объекта.
  - Использование двоичных чисел (binary literals) — префикс «0b» укажет, что используется двоичное число.
  - Упрощение вызова методов varargs — уменьшение предупреждений при вызове метода с переменным числом входящих переменных.
- Модификация загрузчика классов (class-loader) — избежание тупиковых ситуаций в неиерархической топологии загрузки классов.
- Закрытие ресурсов, открытых URLClassLoader.
- Обновление коллекций (JSR 166).
- Поддержка Unicode 6.0.
- Отделение языка пользователя и языка пользовательского интерфейса — обновление обработки языков для отделения локали от языка пользовательского интерфейса.
- Новые интерфейсы I/O для платформы Java (nio.2).

- Использование JDBC 4.1 и Rowset 1.1.
- … (не закончено)

Java 8

Релиз версии состоялся 19 марта 2014 года.

Список нововведений

- Полноценная поддержка лямбда-выражений.
- Ключевое слово default в интерфейсах для поддержки функциональности по умолчанию.
- Ссылки на методы.
- Функциональные интерфейсы (предикаты, поставщики и т.д.)
- Потоки (stream) для работы с коллекциями
- Новое API для работы с датами
- … (не закончено)

Java 9

Список нововведений

- Интеграция jigsaw.
- … (не закончено)

Классификация платформ Java

Внутри Java существуют несколько основных семейств технологий:

- Java SE — Java Standard Edition, основное издание Java, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем.
- Java EE — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия.
- Java ME — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например, в мобильных телефонах, КПК, встроенных системах;
- JavaFX — технология, являющаяся следующим шагом в эволюции Java как Rich Client Platform; предназначена для создания графических интерфейсов корпоративных приложений и бизнеса.
- Java Card — технология предоставляет безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объёмом памяти и возможностями обработки.

Компанией Microsoft была разработана собственная реализация JVM (MSJVM), включавшаяся в состав различных операционных систем, начиная с Windows 98 (также входила в Internet Explorer от версии 3 и выше, что позволяло использовать MSJVM (Microsoft java virtual machine) в ОС Windows 95 и Windows NT 4 после установки IE3+ на данные ОС).

MSJVM имела существенные отличия от Sun Java, во многом ломающие основополагающую концепцию переносимости программ между разными платформами:

- отсутствие поддержки программного интерфейса вызова удаленных методов (RMI);
- отсутствие поддержки технологии JNI;
- наличие нестандартных расширений, таких, как средства интеграции Java и DCOM, работающих только на платформе Windows.

Тесная интеграция Java с DCOM и Win32 поставила под вопрос кроссплатформенную парадигму языка. Впоследствии это явилось поводом для судебных исков со стороны Sun Microsystems к Microsoft. Суд принял сторону компании Sun Microsystems. В конечном счёте между двумя компаниями была достигнута договорённость о возможности продления срока официальной поддержки пользователей нестандартной Microsoft JVM до конца 2007 года.

В 2005 году компанией Microsoft для платформы .NET был представлен Java-подобный язык J#, не соответствующий официальной спецификации языка Java и исключённый впоследствии из стандартного инструментария разработчика Microsoft Visual Studio, начиная с Visual Studio 2008.

Язык Java активно используется для создания мобильных приложений под операционную систему Android. При этом программы компилируются в нестандартный байт-код, для использования их виртуальной машиной Dalvik. Для такой компиляции используется дополнительный инструмент, а именно Software Development Kit, разработанный компанией Google.

Разработку приложений можно вести в среде Android Studio, NetBeans, в среде Eclipse, используя при этом плагин Android Development Tools (ADT) или в IntelliJ IDEA. Версия JDK при этом должна быть 5.0 или выше.

8 декабря 2014 года Android Studio признана компанией Google официальной средой разработки под ОС Android.

Следующие успешные проекты реализованы с привлечением Java (J2EE) технологий: RuneScape, Amazon, eBay, LinkedIn, Yahoo!.

Следующие компании в основном фокусируются на Java (J2EE) технологиях: SAP, IBM, Oracle. В частности, СУБД Oracle Database включает JVM как свою составную часть, обеспечивающую возможность непосредственного программирования СУБД на языке Java, включая, например, хранимые процедуры.

Программы, написанные на Java, имеют репутацию более медленных и занимающих больше оперативной памяти, чем написанные на языке Си. Тем не менее, скорость выполнения программ, написанных на языке Java, была существенно улучшена с выпуском в 1997—1998 годах так называемого JIT-компилятора в версии 1.1 в дополнение к другим особенностям языка для поддержки лучшего анализа кода (такие, как внутренние классы, класс StringBuffer, упрощенные логические вычисления и т. д.). Кроме того, была произведена оптимизация виртуальной машины Java — с 2000 года для этого используется виртуальная машина HotSpot. По состоянию на февраль 2012 года, код Java 7 приблизительно лишь в 1.8 раза медленнее кода, написанного на языке Си.

Некоторые платформы предлагают аппаратную поддержку выполнения для Java. К примеру, микроконтроллеры, выполняющие код Java на аппаратном обеспечении вместо программной JVM, а также основанные на ARM процессоры, которые поддерживают выполнение байткода Java через опцию Jazelle.

Основные возможности:

- автоматическое управление памятью;
- расширенные возможности обработки исключительных ситуаций;
- богатый набор средств фильтрации ввода-вывода;
- набор стандартных коллекций: массив, список, стек и т. п.;
- наличие простых средств создания сетевых приложений (в том числе с использованием протокола RMI);
- наличие классов, позволяющих выполнять HTTP-запросы и обрабатывать ответы;
- встроенные в язык средства создания многопоточных приложений;
- унифицированный доступ к базам данных:
  - на уровне отдельных SQL-запросов — на основе JDBC, SQLJ;
  - на уровне концепции объектов, обладающих способностью к хранению в базе данных — на основе Java Data Objects (*англ.*) и Java Persistence API;
- поддержка обобщений (начиная с версии 1.5);
- параллельное выполнение программ

3.3. Основные конструкции языка программирования

В языке Java только 8 примитивных (скалярных, простых) типов: boolean, byte, char, short, int, long, float, double. Существует также вспомогательный девятый примитивный тип — void, однако переменные и поля такого типа не могут быть объявлены в коде, а сам тип используется только для описания соответствующего ему класса, для использования при рефлексии. Кроме того, с помощью класса Void можно узнать, является ли определённый метод типа void: Hello.class.getMethod("main", Array.newInstance(String.class, 0).getClass()).getReturnType() == Void.TYPE.

Длины и диапазоны значений примитивных типов определяются стандартом, а не реализацией, и приведены в таблице. Тип char сделали двухбайтовым для удобства локализации (один из идеологических принципов Java): когда складывался стандарт, уже существовал Unicode-16, но не Unicode-32. Поскольку в результате не осталось однобайтового типа, добавили новый тип byte, причем в Java, в отличие от других языков, он не является беззнаковым. Типы float и double могут иметь специальные значения и «не число» (NaN). Для типа double они обозначаются Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, Double.NaN; для типа float — так же, но с приставкой Float вместо Double. Минимальные и максимальные значения, принимаемые типами float и double, тоже стандартизованы.

Тип	Длина (в байтах)	Диапазон или набор значений
boolean	1 в массивах, 4 в переменных	true, false
byte	1	$-128..127$
char	2	$0..2^{16}-1$, или 0..65535
short	2	$-2^{15}..2^{15}-1$, или −32768..32767
int	4	$-2^{31}..2^{31}-1$, или −2147483648..2147483647
long	8	$-2^{63}..2^{63}-1$, или примерно $-9.2\cdot10^{18}..9.2\cdot10^{18}$
float	4	$-(2-2^{-23})\cdot2^{127}..(2-2^{-23})\cdot2^{127}$, или примерно $-3.4\cdot10^{38}..3.4\cdot10^{38}$, а также , ∞, NaN
double	8	$-(2-2^{-52})\cdot2^{1023}..(2-2^{-52})\cdot2^{1023}$, или примерно $-1.8\cdot10^{308}..1.8\cdot10^{308}$, а также −∞, ∞, NaN

Такая жёсткая стандартизация была необходима, чтобы сделать язык платформенно-независимым, что является одним из идеологических требований к Java. Тем не менее, одна небольшая проблема с платформенной независимостью всё же осталась. Некоторые процессоры используют для промежуточного хранения результатов 10-байтовые регистры или другими способами улучшают точность вычислений. Для того, чтобы сделать Java максимально совместимой между разными системами, в ранних версиях любые способы повышения точности вычислений были запрещены. Однако это приводило к снижению быстродействия. Выяснилось, что ухудшение точности ради платформенной независимости мало кому нужно, тем более если за это приходится платить замедлением работы программ. После многочисленных протестов этот запрет отменили, но добавили ключевое слово strictfp, запрещающее повышение точности.

Преобразования при математических операциях в языке Java действуют следующие правила:

1. Если один операнд имеет тип double, другой тоже преобразуется к типу double.

2. Иначе, если один операнд имеет тип float, другой тоже преобразуется к типу float.

3. Иначе, если один операнд имеет тип long, другой тоже преобразуется к типу long.

4. Иначе оба операнда преобразуются к типу int.

Данный способ неявного преобразования встроенных типов полностью совпадает с преобразованием типов в C++.

В языке Java имеются только динамически создаваемые объекты. Причем переменные объектного типа и объекты в Java — совершенно разные сущности. Переменные объектного типа являются ссылками, то есть неявными указателями на динамически создаваемые объекты. Это подчёркивается синтаксисом описания переменных. Так, в Java нельзя писать:

```
double a[10][20];
Foo b(30);
```

а нужно:

```
double[][] a = new double[10][20];
Foo b = new Foo(30);
```

При присваиваниях, передаче в подпрограммы и сравнениях объектные переменные ведут себя как указатели, то есть присваиваются, копируются и сравниваются адреса объектов. А при доступе с помощью объектной переменной к полям данных или методам объекта не требуется никаких специальных операций разыменовывания — этот доступ осуществляется так, как если бы объектная переменная была самим объектом.

Объектными являются переменные любого типа, кроме примитивного. Явных указателей в Java нет. В отличие от указателей C, C++ и других языков программирования, ссылки в Java в высокой степени безопасны благодаря жёстким ограничениям на их использование, в частности:

- Нельзя преобразовывать объект типа int или любого другого примитивного типа в указатель или ссылку и наоборот.
- Над ссылками запрещено выполнять операции ++, −−, +, − или любые другие арифметические операции.
- Преобразование типов между ссылками жёстко регламентировано. За исключением ссылок на массивы, разрешено преобразовывать ссылки только между наследуемым типом и его наследником, причём преобразование наследуемого типа в наследующий должно быть явно задано и во время выполнения производится проверка его осмысленности. Преобразования ссылок на массивы разрешены лишь тогда, когда разрешены преобразования их базовых типов, а также нет конфликтов размерности.
- В Java нет операций взятия адреса (&) или взятия объекта по адресу (*). Амперсанд (&) означает всего лишь «побитовое и» (двойной амперсанд — «логическое и»). Однако для булевых типов одиночный амперсанд означает «логическое и», отличающееся от двойного тем, что цепь проверок не прекращается при получении в выражении значения false (напр. a == b && foo() == bar() не повлечёт вызовов foo() и bar() в случае, если a != b, тогда как использование & повлечёт в любом случае.

Благодаря таким специально введенным ограничениям в Java невозможно прямое манипулирование памятью на уровне физических адресов (хотя определено значение ссылки, не указывающей ни на что: null).

Если нужен указатель на примитивный тип, используются классы-обёртки примитивных типов: Boolean, Byte, Character, Short, Integer, Long, Float, Double.

Из-за того, что объектные переменные являются ссылочными, при присваивании не происходит копирования объекта. Так, если написать

```
Foo foo, bar;
```

…
bar = foo;
,

то произойдет копирование адреса из переменной foo в переменную bar. То есть foo и bar будут указывать на одну и ту же область памяти, то есть на один и тот же объект; попытка изменить поля объекта, на который ссылается переменная foo, будет менять объект, с которым связана переменная bar, и наоборот. Если же необходимо получить именно ещё одну *копию* исходного объекта, пользуются или методом (функцией-членом, в терминологии C++) clone(), создающим копию объекта, или (реже) копирующим конструктором (конструкторы в Java не могут быть виртуальными, поэтому экземпляр класса-потомка будет неправильно скопирован конструктором класса-предка; метод клонирования вызывает нужный конструктор и тем самым позволяет обойти это ограничение).

Метод clone() требует, чтобы класс реализовывал интерфейс Cloneable (об интерфейсах см. ниже). Если класс реализует интерфейс Cloneable, по умолчанию clone() копирует все поля (*мелкая копия*). Если требуется не копировать, а клонировать поля (а также их поля и так далее), надо переопределять метод clone(). Определение и использование метода clone() часто является нетривиальной задачей.

В языке Java невозможно явное удаление объекта из памяти — вместо этого реализована Сборка мусора. Традиционным приёмом, дающим сборщику мусора «намёк» на освобождение памяти, является присваивание переменной пустого значения null. Это, однако, не значит, что объект, заменённый значением null, будет непременно и немедленно удалён, но есть гарантия, что этот объект будет удалён именно в будущем. Данный приём всего лишь устраняет ссылку на объект, то есть отвязывает указатель от объекта в памяти. При этом следует учитывать, что объект не будет удален сборщиком мусора, пока на него указывает хотя бы одна ссылка из используемых переменных или объектов. Существуют также методы для инициации принудительной сборки мусора, но не гарантируется, что они будут вызваны исполняющей средой, и их не рекомендуется использовать для обычной работы.

3.4. Специальные классы и функции

Java не является процедурным языком: любая функция может существовать только внутри класса. Это подчёркивает терминология языка Java, где нет понятий «функция» или «функция-член» (англ. *member function*), а только *метод*. В методы превратились и стандартные функции. Например, в Java нет функции sin(), а есть метод Math.sin() класса Math (содержащего, кроме sin(), методы cos(), exp(), sqrt(), abs() и многие другие). Конструкторы в Java не считаются методами. Деструкторов в Java не существует, а метод finalize() ни в коем случае нельзя считать аналогом деструктора.

Конструктор — это специальный метод, который вызывается при создании нового объекта. Не всегда удобно инициализировать все переменные класса при создании его экземпляра. Иногда проще, чтобы какие-то значения были бы созданы по умолчанию при создании объекта. По сути, конструктор нужен для автоматической инициализации переменных.

Конструктор инициализирует объект непосредственно во время создания. Имя конструктора совпадает с именем класса, включая регистр, а по синтаксису конструктор похож на метод без возвращаемого значения.

private int Cat(); // так выглядит метод по имени Cat
Cat(); // так выглядит конструктор класса Cat

В отличие от метода, конструктор никогда ничего не возвращает.

Конструктор определяет действия, выполняемые при создании объекта класса, и является важной частью класса. Как правило, программисты стараются явно указать конструктор. Если явного конструктора нет, то Java автоматически создаст его для использования по умолчанию.

Создадим класс Box с конструктором, который просто установит начальные значения для коробки.

```
class Box {
   int width; // ширина коробки
   int height; // высота коробки
   int depth; // глубина коробки

   // Конструктор
   Box(int a, int b) {
      width = a;
      height = b;
      depth = 10;
   }

   // вычисляем объём коробки
   int getVolume() {
      return width * height * depth;
   }
}
```

Даже если конструктор специально НЕ определён, виртуальная машина Java обязательно его создаст (пустым).

В Java (как и в C++) используются *статические методы* (англ. *static method* — в теории программирования их также называют методами класса), которые задаются при помощи ключевого слова static. Статические поля (переменные класса) имеют тот же смысл, что и в C++: каждое такое поле является собственностью класса, поэтому для доступа к статическим полям не требуется создавать экземпляры соответствующего класса.

Например, математические функции, реализованные в классе Math, представляют собой как раз статические методы данного класса. Поэтому можно писать

```
double x = Math.sin(1);
```

вместо

```
Math m = new Math();
double x = m.sin(1);
```

Поскольку статические методы существуют независимо от объектов (экземпляров класса), они не имеют доступа к обычным (нестатическим) полям и методам данного класса. В частности, при реализации статического метода недопустимо использовать идентификатор this.

Ключевое слово final (финальный) имеет разные значения при описании поля, метода или класса.

1. Финальное **поле** класса инициализируется при описании или в конструкторе класса (а статическое поле — в статическом блоке инициализации). Впоследствии его значение не может быть изменено. Если статическое поле класса или переменная проинициализированы константным выражением, они рассматриваются компилятором как именованная константа; в таком случае их значение может быть использовано в операторах switch (для констант типа int), а

также для условной компиляции (для констант типа boolean) при использовании с оператором if.

2. Значения **локальных переменных**, а также **параметров метода**, помеченных ключевым словом final, не могут быть изменены после присвоения. При этом их значения могут использоваться внутри анонимных классов.

3. **Метод** класса, отмеченный словом final, не может быть переопределён при наследовании.

4. Финальный **класс** не может иметь наследников.

В Java методы, не объявленные явно как static, final или private, являются виртуальными в терминологии C++: при вызове метода, по-разному определённого в базовом и наследующем классах, всегда производится проверка времени выполнения.

Абстрактным методом (модификатор abstract) в Java называется метод, для которого заданы параметры и тип возвращаемого значения, но не задано тело. Абстрактный метод определяется в классах-наследниках. Аналог абстрактного метода в C++ — чисто виртуальная функция (pure virtual function). Для того чтобы в классе можно было описывать абстрактные методы, сам класс тоже должен быть описан как абстрактный. Объекты абстрактного класса создавать нельзя.

Высшей степенью абстрактности в Java является интерфейс (interface). Все методы интерфейса абстрактны: описатель abstract даже не требуется. Интерфейс в Java не считается классом, хотя, по сути, является полностью абстрактным классом. Класс может наследовать/*расширять* (extends) другой класс или *реализовывать* (implements) интерфейс. Кроме того, интерфейс может наследовать/расширять другой интерфейс.

В Java класс не может наследовать более одного класса, зато может реализовывать несколько интерфейсов. Множественное наследование интерфейсов не запрещено, то есть один интерфейс может наследоваться от нескольких.

Интерфейсы можно использовать в качестве типов параметров методов. Нельзя создавать экземпляры интерфейсов.

В Java есть интерфейсы, которые не содержат методов для реализации, а специальным образом обрабатываются JVM:

- java.lang.Cloneable
- java.io.Serializable
- java.util.RandomAccess
- java.rmi.Remote

3.5. Абстрактные и анонимные классы и их использование

Кроме обычных классов в Java есть **абстрактные классы**. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово **abstract**:

```
public abstract class Human{

    private int height;
    private double weight;

    public int getHeight() { return height;
}
    public double getWeight() { return
weight; }
```

}

Кроме обычных методов абстрактный класс может содержать **абстрактные методы**. Такие методы определяются с помощью ключевого слова abstract и не имеют никакого функционала:

```
public abstract void displayInfo();
```

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Зачем нужны абстрактные классы? Допустим, мы делаем программу для обсулживания банковских операций и определяем в ней три класса: Person, который описывает человека, Employee, который описывает банковского служащего, и класс Client, который представляет клиента банка. Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы. И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
package inheritance;

public abstract class Person {

    private String name;
    private String surname;

    public String getName() { return name; }
    public String getSurname() { return surname; }

    public Person(String name, String surname){

        this.name=name;
        this.surname=surname;
    }

    public abstract void displayInfo();
}

class Employee extends Person{

    private String bank;

    public Employee(String name, String surname, String
company) {

        super(name, surname);
        this.bank=company;
    }

    public void displayInfo(){

        System.out.println("Имя: " + super.getName() + "
Фамилия: "
```

```
                + super.getSurname() + " Работает в банке: " +
bank);
        }
    }

    class Client extends Person
    {
        private String bank;

        public Client(String name, String surname, String
company) {

            super(name, surname);
            this.bank=company;
        }

        public void displayInfo(){

            System.out.println("Имя: " + super.getName() + "
Фамилия: "
                + super.getSurname() + " Клиент банка: " + bank);
        }
    }
```

Другим хрестоматийным примером является системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```
    // абстрактный класс фигуры
    public abstract class Figure{

        float x; // x-координата точки
        float y; // y-координата точки

        public Figure(float x, float y){

            this.x=x;
            this.y=y;
        }
        // абстрактный метод для получения периметра
        public abstract float getPerimeter();
        // абстрактный метод для получения площади
        public abstract float getArea();
    }
    // производный класс прямоугольника
    class Rectangle extends Figure
    {
        private float width;
        private float height;

        // конструктор с обращением к конструктору класса
Figure
        public Rectangle(float x, float y, float width, float
```

```
height){

        super(x,y);
        this.width = width;
        this.height = height;
    }

    public float getPerimeter(){

        return width * 2 + height * 2;
    }

    public float getArea(){

        return width * height;
    }
}
```

Анонимный класс является дальнейшим развитием идеи локального класса. Вместо того чтобы объявлять локальный класс с помощью одного оператора Java и затем с помощью другого оператора создавать экземпляр данного класса, в анонимном классе эти этапы объединены в одну Java-конструкцию.

Анонимный класс не имеет имени. К тому же, экземпляр его создается в том же выражении, в котором он объявляется, поэтому такой экземпляр может быть только один. В остальном, анонимные классы по своим свойствам и принципам использования очень похожи на локальные. Интерфейсы не могут быть анонимными.

В следующем примере продемонстрировано использование анонимного класса для реализации интерфейса java.io.FilenameFilter. Реализация в данном примере используется для построения списка файлов с расширением Java, которые содержатся в заданном каталоге.

/* Простая программа, предназначенная для вывода списка всех файлов с исходными текстами Java-программ, содержащихся в каталоге, имя которого передается программе через аргумент командной строки. File f соответствует имени каталога. Выводится список файлов в каталоге с использованием заданного объекта фильтра. Анонимный класс определяется в выражении, осуществляющем вызов метода. */

```
public class mainclass {
public static void main(String[] args) {
File f = new File(args[0]);
String[] list = f.list(new FilenameFilter() {
public boolean accept(File f, String s) {
return s.endsWith(".java");}
});
for (int i = 0; i < list.length; i++) // Выводит список
System.out.println(list[i]);
}}
```

Как видно из примера, для определения анонимного класса и создания его экземпляра используется обычный синтаксис выражения с оператором new, за которым следует определение тела класса в фигурных скобках. Если имя, стоящее за ключевым словом new, является именем класса, то анонимный класс является подклассом этого именованного класса.

Поскольку анонимный класс не имеет имени, нельзя определить и его конструктор в теле класса. Это одно из основных ограничений, которые накладываются на анонимные классы. Любые аргументы, указанные в скобках после имени суперкласса в определении анонимного класса, будут неявно переданы конструктору суперкласса. Поскольку анонимные классы не имеют имен, возникает вопрос: как называются соответствующие им файлы классов? Если откомпилировать пример, то в результате будут созданы два файла классов, mainclass.class и mainclass$1.class.

Анонимным классам присваиваются номера, чтобы получить уникальные имена файлов классов на основе имени включающего класса.

3.6. Шаблоны и контейнеры

Начиная с версии Java 5 в языке появился механизм обобщённого программирования — шаблоны, внешне близкие к шаблонам C++. С помощью специального синтаксиса в описании классов и методов можно указать параметры-типы, которые внутри описания могут использоваться в качестве типов полей, параметров и возвращаемых значений методов.

```
// Объявление обобщённого класса
class GenericClass<E> {
  E getFirst() { ... }
  void add(E obj) { ... }
}

// Использование обобщённого класса в коде
GenericClass<String> var = new GenericClass<String>();
var.add("qwerty");
String p = var.getFirst();
```

Допускается обобщённое объявление классов, интерфейсов и методов. Кроме того, синтаксис поддерживает ограниченные объявления типов-параметров: указание в объявлении конструкции вида <T extends A & B & C...> требует, чтобы тип-параметр T реализовывал интерфейсы A, B, C и так далее, а конструкция <T super C> требует, чтобы тип-параметр T был типом C или одним из его предков.

В отличие от шаблонов C#, шаблоны Java не поддерживаются средой исполнения — компилятор просто создаёт байт-код, в котором никаких шаблонов уже нет. Реализация шаблонов в Java принципиально отличается от реализации аналогичных механизмов в C++: компилятор не порождает для каждого случая использования шаблона отдельный вариант класса или метода-шаблона, а просто создаёт одну реализацию байт-кода, содержащую необходимые проверки и преобразования типов. Это приводит к ряду ограничений использования шаблонов в программах на Java.

В Java можно явно проверить, к какому классу принадлежит объект. Выражение foo instanceof Foo истинно, если объект foo принадлежит классу Foo или его наследнику, или реализует интерфейс Foo (или, в общем виде, наследует класс, который реализует интерфейс, который наследует Foo).

Далее функция getClass(), определённая для всех объектов, выдаёт объект типа Class. Для каждого класса создаётся не более одного описывающего его объекта типа Class, поэтому эти объекты можно сравнивать. Так, например, foo.getClass() == bar.getClass() будет истинно, если объекты foo и bar принадлежат к одному классу.

Кроме того, объект типа Class любого типа можно получить так: Integer.class, Object.class.

Прямое сравнение классов не всегда является оптимальным средством проверки на принадлежность к классу. Зачастую вместо него используют функцию isAssignableFrom().

Эта функция определена у объекта типа Class и принимает объект типа Class в качестве параметра. Таким образом, вызов Foo.class.isAssignableFrom(Bar.class) вернёт true в случае, если Foo является предком класса Bar. Так как все объекты являются потомками типа Object, вызов Object.class.isAssignableFrom() всегда вернёт true.

В паре с упомянутыми функциями объекта типа Class используются также функции isInstance() (эквивалентно instanceof), а также cast() (преобразует параметр в объект выбранного класса).

При создании коллекции можно указать тип объектов, которые будут храниться в ней. Пример:

```
package genericProj.ex;

import java.util.Vector;

public class GenClass   {
      public static void main(String[] args)                {
            Vector<String> vect;
            vect = new Vector<String>();
            vect.add("str1");
            String str = vect.get(0);
            System.out.print(str);                }
}
```

Напечатает “str1”.

Строка Vector < String> vect; говорит о том, что вектор будет содержать только строки. Вот почему в строке String str = vect.get(0); можно обойтись без приведения типов. При создании объекта вектора указываем параметр String: vect = new Vector < String>();.

Главная цель шаблонов – type-safety, т.е. правильное соотношение типов.

В предыдущем примере однозначно был указан параметр вектора. Есть случаи, где удобнее использовать формальный параметр, подобно формальным аргументам функций. Пример:

```
package formalParam.ex;

import java.util.Vector;

public class FormalVector< E> extends Vector   {
      private static final long serialVersionUID = 1L;
      private Vector< E> vect;

      public FormalVector()     {
            vect = new Vector< E>();                }

      public void setVal(E value)     {
            vect.add(value);                }

      public E getVal(int pos)          {
            return vect.get(pos);               }
}
```

Создан параметризованный класс FormalVector < E>. E – формальный параметр. При создании экземпляра класса вместо E указываем конкретный тип. Простой пример:

```
package formalParam.ex;
```

```
public class StringValues        {
    public static void main(String[] args)        {
        FormalVector<String> vect = new FormalVector<String>();
        vect.setVal("str1");
        String str = vect.getVal(0);
        System.out.print(str);            }
}
```

Напечатает “str1”.

В классе StringValues создаётся экземпляр класса FormalVector < String>. Вместо формального параметра E указываем String. Теперь класс FormalVector будет хранить и работать с типом String. Можно создать экземпляр класса FormalVector с другим параметром, например, Integer: FormalVector < Integer> vect = new FormalVector < Integer>(). Это значит, что FormalVector < Integer> будет работать с типом Integer.

Все экземпляры параметризованного класса представляют один тип, независимо от фактического значения параметров.

Важное правило: если класс A есть потомок класса B, то это не означает, что aClass < A> есть потомок aClass < B>.

Шаблоны могут применяться и в методах. Типы аргументов можно делать параметрическими. Пример:

void aMethod(Vector < E> vect);

или так:

void aMethod(Vector < ?> vect);

Рекомендуется использовать неизвестный параметр < ?> в случаях, когда от него не зависят другие аргументы и возвращаемое значение. Если такая зависимость имеется, то следует использовать формальный параметр < E>.

Возможно использовать вместе неизвестный параметр < ?> и формальный параметр < E> :

pablic < E> void aMethod(Vector < E> vect, List < ? extends E> list);

здесь аргумент list имеет ограниченный неизвестный параметр < ? extends E>, но имеется связь между двумя аргументами – это формальный параметр < E>.

Вместо формального параметра можно указать знак вопроса, т.е. неизвестный параметр. Изменим класс StringValues(см. выше):

```
package formalParam.ex;

public class StringValues            {
    private FormalVector<String> vect;

    public StringValues()       {
        vect = new FormalVector<String>();
        vect.setVal("str1");
        printValues(vect);                }
    private void printValues(FormalVector<?> vect)       {
        Object str = vect.getVal(0);
        System.out.println(str);
    }

    public static void main(String[] args)        {
        new StringValues();          }
}
```

Напечатает “str1”.

Новым для нас является аргумент метода

private void printValues(FormalVector < ?> vect)

Знак вопроса вместо какого-то типа говорит о том, что здесь может оказаться любой тип, т.е. FormalVector < ?> - это супертип для любого конкретного типа, например, для FormalVector < String>. Обратим внимание на первую строку метода:

Object str = vect.getVal(0);

здесь str имеет тип Object. Это позволяет безопасно получать элементы коллекции методом getVal, ведь любые объекты представляют тип Object.

Неизвестный параметр может примениться и при объявлении полей:

private Vector < ?> aVector;

Ограниченный неизвестный параметр указывает, что вместо него можно подставить не любой тип, а только тип, наследуемый от определённого класса. Пример:

Vector < ? extends aClass>

здесь на место неизвестного параметра ? можно поставить aClass или его наследников, пусть даже непрямых. aClass в этом случае называется *верхней границей* неизвестного параметра. Другой пример:

Vector < ? super aClass>

Параметр вида < ? super aClass> называется *нижняя граница*. Фактический параметр должен быть родительским по отношению к aClass, или должен быть самим aClass.

Рассмотрим формальный пример. Создадим три пустых класса: ClassA

public class ClassA {}

его наследника ClassB

public class ClassB extends ClassA {}

ClassC, как наследника от ClassB:

public class ClassC extends ClassB {}

Почему классы пустые? В этом примере важно увидеть отношения наследования между классами, а не содержимое этих классов.

Теперь создадим класс MainClass, а в нём метод

static void addObject(Vector < ? extends ClassA> vect) {}

В методе addObject имеем ограниченный неизвестный параметр < ? extends ClassA>, который говорит нам, что аргументом может быть любой вектор, содержащий объекты класса ClassA или его наследников. А таковыми и являются наши три пустых класса: ClassA, ClassB, ClassC.

В коде метода main создадим объект класса ClassA

ClassA ca = new ClassA();

вектор, содержащий объекты класса ClassA

Vector < ClassA> vectA = new Vector < ClassA>();

добавим ca к vectA

vectA.add(ca);

Такой вектор соответсвует определению аргумента метода addObject, к которому мы и обращаемся:

addObject(vectA);

Для классов ClassB, ClassC действия аналогичные. Вот полный код класса MainClass:

```
import java.util.Vector;

public class MainClass          {
    static void addObject(Vector<? extends ClassA> vect)    {
    }

    public static void main(String[] args)      {
        new MainClass();
```

```
        ClassA ca = new ClassA();
        Vector<ClassA> vectA = new Vector<ClassA>();
        vectA.add(ca);
        addObject(vectA);

        ClassB cb = new ClassB();
        Vector<ClassB> vectB = new Vector<ClassB>();
        vectB.add(cb);
        addObject(vectB);

        ClassC cc = new ClassC();
        Vector<ClassC> vectC = new Vector<ClassC>();
        vectC.add(cc);
        addObject(vectC);
    }
}
```

Если метод addObject объявить так:

static void addObject(Vector < ClassA> vect) {}

т.е. без ограниченного неизвестного параметра, то получим сообщения об ошибках в строках addObject(vectB); и addObject(vectC);(несоответствие типов аргументов). Почему? Смотрите пункт “Шаблоны и подтипы”. Vector < ClassB> и Vector < ClassC> есть подтипы не Vector < ClassA>, а Vector < ? extends ClassA>.

Объявляя класс-коллекцию, мы указываем параметр, т.е. тип объектов, хранящихся в нём. Если не указать параметр, то можем получить предупреждение “raw type”. Пример:

Vector<String> vect;//Предупреждения нет;

Vector anotherVect;//Кандидат на предупреждение “raw type”, т.к. не указан параметр;

Можно объявлять коллекции без указания параметра, это сделано для работы с прежним кодом, когда ещё не было шаблонов. Ещё пример:

```
Vector<String> strVect = new Vector<String>();
Vector vect = strVect;
```

vect.add(new Double(1.2));//предупреждение “raw type”, т.к. вектор vect объявлен без параметра;

В последней строке получаем предупреждение, которое приведёт к ошибке при попытке извлечь строку из vect:

System.out.print((String) vect.get(0));

Сообщение об ошибке: java.lang.ClassCastException: java.lang.Double cannot be cast to java.lang.String. Исправить положение можно так: указать параметр для vect, Double привести к строке

```
Vector<String> vect = new Vector<String>();
vect.add((new Double(1.2)).toString());
//Извлечём строку из vect;
```

System.out.print((String) vect.get(0));//Ошибки нет; результат будет “1.2”;

Это значит, что компилятор не гарантирует соответствие типов. Пример:

```
Vector<Vector<String>> strVect = new Vector<Vector<String>>();
Vector aVect = new Vector();
strVect.add(aVect);//здесь получаем unchecked conversion;
```

Мы не знаем, что будет храниться в aVect, там может оказаться и не строка, отсюда предупреждение. Исправляем положение, указав параметр для aVect:

```
Vector<Vector<String>> strVect = new Vector<Vector<String>>();
Vector<String> aVect = new Vector<String>();
strVect.add(aVect);//Предупреждения нет;
```

Параметрические типы принято обзначать T, если имеется несколько параметров, то для следующих используют буквы, близкие к T, т.е. S, U, V и т.д. Элементы коллекций обозначают буквой E. Ключи – K, значения - V, числа – N. Во всех случаях параметры обозначают большими буквами.

Контейнер — в программировании, структура (АТД), позволяющая инкапсулировать в себя объекты разных типов, в основном они построены на основе шаблонов. Примерами контейнеров являются контейнеры из стандартной библиотеки (STL) — map, vector и др. В контейнерах часто встречается реализация алгоритмов для них.

Контейнер, в отличие от коллекции, не допускает явного задания числа элементов и не поддерживает ветвистой структуры

Класс, главной целью которого является хранение объектов, называется контейнером. Реализация контейнеров, подходящих для данной задачи, и поддержка их нужными операциями – важнейшие шаги при написании любой программы.

Template Numerical Toolkit (или TNT; переводится библиотека численных шаблонов) — библиотека шаблонов в языке C++ для манипуляций одномерными, двумерными и трёхмерными массивами. Библиотека создана в NIST и является общественным достоянием.

TNT предоставляет возможность присваивания без копирования с автоматическим подсчётом числа ссылок, поэлементных операций (сложения, вычитания, умножения и деления) и ввода-вывода массивов разной размерности, а также матричного умножения.

Более сложные операции линейной алгебры, в частности LU-разложение, обращение матриц, вычисление определителей, собственных значений и собственных векторов, QR-разложение, разложение Холецкого, сингулярное разложение, предоставляются библиотекой шаблонов JAMA, также разработанной в NIST и использующей TNT. Как и TNT, JAMA является общественным достоянием.

Поскольку TNT не содержит ничего, кроме заголовочных файлов с шаблонами, она не требует компиляции. Матрицы можно сохранять по строкам или по столбцам, для совместимости с Фортраном. Поскольку все классы используют шаблоны, одинаково легко использовать матрицы с элементами типа float, double или описанных пользователем типов. Библиотека предоставляет некоторые возможности работы с разреженными матрицами.

Существуют:

- string
- vector
- list
- ostream, istream, ofstream, ifstream, ostringstream.

Контрольные вопросы:

1. Какие платформы используются для создания ОС мобильных приложений?
2. Абстрактные классы и их использование
3. Анонимные классы и их использование
4. Использование шаблонов и контейнеров
5. Зачем нужны абстрактные классы?