ВВЕДЕНИЕ

Учебная дисциплина «Программирование для мобильных платформ» предназначена для профессиональной разработки программного обеспечения для мобильных устройств на платформе Android. Основой для изучения данной дисциплины является язык JAVA и операционная система Android.

Практикум разделен на две части: 1. «Приемы объектного программирования на JAVA»; 2. «Разработка мобильных приложений на платформе Android».

Цель лабораторного практикума «Программирование для мобильных платформ» – изучение основ программирования на языке JAVA, особенностей языка и его отличий от других языков (C++, C#, Java-script) и возможностей, которые предоставляет данная платформа для разработки мобильных систем, получение практических навыков по созданию пользовательских интерфейсов и многопоточных интерактивных приложений.

После изучения этого модуля студенты будут знать особенности проектирования приложений на свободно распространяемой платформе JAVA, особенности языка и способы использования инструментов для организации эффективной работы в JAVA, в том числе приемы обработки событий, способы организации и преимущества многопоточных приложений; владеть навыками практического применения описанных инструментальных средств и методами разработки мобильного ПО.

Требования к «входным» знаниям, умениям обучающегося в объёме компетенций бакалавра: понимание основных принципов объектно-ориентированного программирования и проектирования, знание синтаксиса языков C и C++, знакомство с WEB-технологиями.

1. ОБЗОР И СРАВНИТЕЛЬНАЯ ХАРАКТЕРИСТИКА МОБИЛЬНЫХ ПЛАТФОРМ

1.1. Операционная система iOS

Операционная система iOS (до 2010 г. называлась iPhoneOS) разрабатывается и выпускается американской компанией Apple с 2007 г. iOS — операционная система, разработанная для использования исключительно для продукции компании Apple. iPhoneOS была впервые представлена Стивом Джобсом 9 января 2007 г. одновременно с первой моделью iPhone и предназначалась для работы на смартфонах iPhone и iPod touch. Спустя некоторое время iOS стала применяться для работы в таких устройствах, как iPad, планшет от Apple, первая версия которого была выпущена 27 января 2010 г., и цифрового мультимедийного проигрывателя Apple TV, а затем и автомобильных мультимедийных систем.

В основе работы операционной системы iOS лежит ядро XNU. Оно в основном содержит в себе не только программный код, разработанный компанией Apple, но и такие программные коды, как OS NeXTSTEP и FreeBSD. Также операционная система iOS практически ничем не отличается от Apple OS X.

Условно начинку OS X / iOS можно разделить на три логических уровня (рис.1): ядро XNU, слой совместимости со стандартом POSIX (плюс различные системные демоны / сервисы) и слой NeXTSTEP, реализующий графический стек, фреймворк и API приложений. Darwin (UNIX – подобная ОС) включает первые два слоя и распространяется свободно, но только в версии для OS X. iOS-вариант, портированный на архитектуру ARM и включающий некоторые доработки, полностью закрыт и распространяется только в составе прошивок для iOS-устройств.

Ключевой компонент Darwin — гибридное ядро XNU, основанное на ядре Mach и компонентах ядра FreeBSD, таких как планировщик процессов, сетевой стек и виртуальная файловая система (слой VFS). В отличие от Mach и FreeBSD, ядро OS X использует собственный API драйверов, названный IoKit и позволяющий писать драйверы на C++ с использованием объектно-ориентированного подхода, сильно упрощающего разработку.

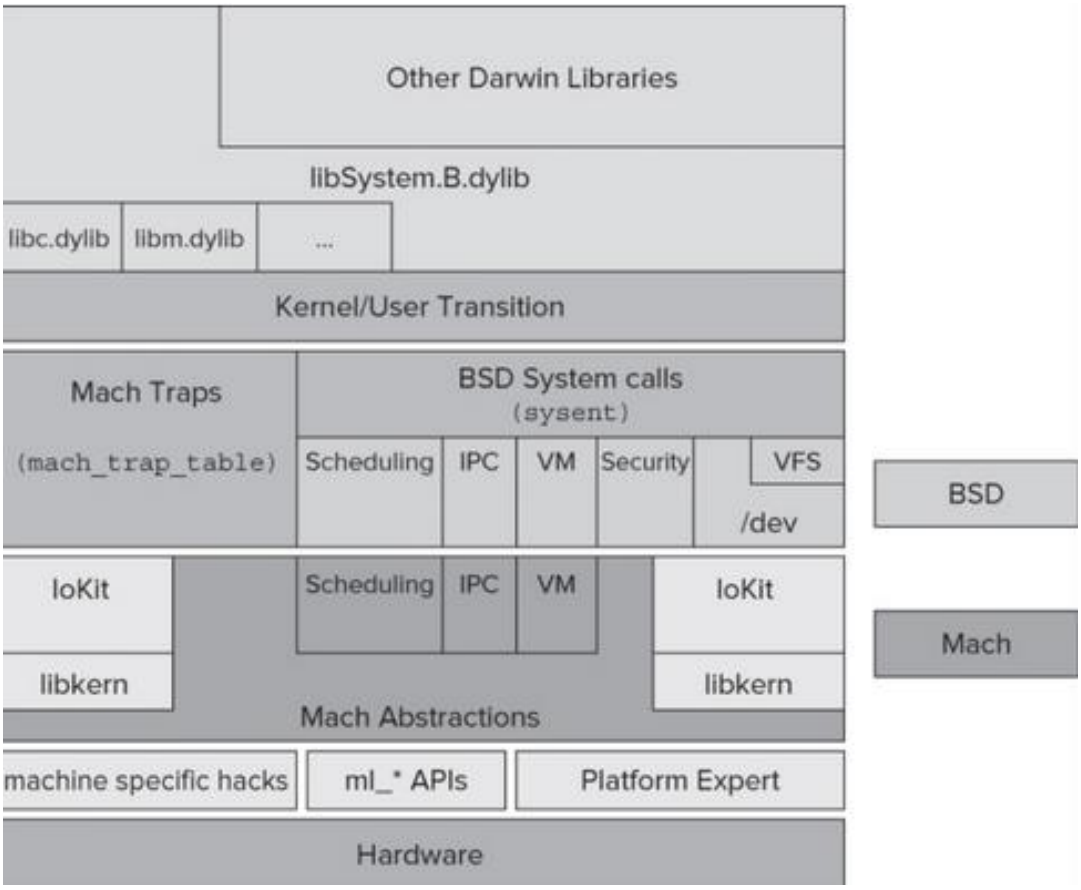


Рис.1. Структура OS X / iOS

iOS использует несколько измененную версию XNU, однако в силу того, что ядро iOS закрыто, сказать, что именно изменила Apple, затруднительно. Известно только, что оно собрано с другими опциями компилятора и модифицированным менеджером памяти, который учитывает небольшие объемы оперативки в мобильных устройствах. Во всем остальном это все то же XNU, которое можно найти в виде зашифрованного кеша (ядро + все драйверы / модули) в каталоге System/Library/Caches/com.apple.kernelcaches/kernelcache на самом устройстве.

Уровнем выше ядра в Darwin располагается слой UNIX/BSD, включающий набор стандартных библиотек языка Си (libc, libmatch, libpthread и т. д.).

На этом открытая часть ОС под названием Darwin заканчивается и начинается слой фреймворков, которые как раз и образуют то, что мы привыкли считать OS X / iOS.

Для разработки приложений обязательно понадобятся дорогие устройства с системой Mac OS X, так как ни на какой другой платформе невозможно работать с iOS SDK. Есть много различных вариантов разработки iOS приложений, но минимум для сборки и загрузки приложения в AppStore вам будет необходима Mac OS X.

Версии.

Самая свежая версия системы – 12. Примечательно, что iPhone OS стала iOS только с анонсом четвертой версии.

Языки разработки.

Разработчики из компании Apple предоставили два варианта:

– Objective-C. Ветеран с большой историей, постепенно отходящий на второй план;

– Swift. Очень молодой и быстро набирающий популярность среди разработчиков.

Оба языка являются объектно-ориентированными и успешно выполняют основные парадигмы ООП: наследование, полиморфизм, инкапсуляцию и абстракцию.

Swift – молодой, мощный и открытый язык программирования общего назначения. Официально представлен компанией Apple в июне 2014 г. Сочетает в себе все лучшее от C и Objective-C, но лишен ограничений последнего, накладываемых в угоду совместимости с C. В Swift используется строгая типизация объектов, уменьшающая количество ошибок ещё на этапе написания кода. Также в Swift добавлены современные функции, такие как дженерики, замыкания, множественные возвращаемые значения и многое другое, превращающие создание приложения в более гибкий и увлекательный процесс.

C++ также поддерживается iOS, но всё приложение целиком на нём написать не удастся. C++ подойдёт для решения логических задач или целых модулей приложения, а также для написания сложных алгоритмов, но пользовательский интерфейс должен быть написан на Objective-C или Swift.

Достоинства разработки для iOS.

Платформа для разработки – Xcode, и работать в этой среде — удовольствие. В отличие от инструментов для Android-разработки, она гибкая, быстрая, мощная и способна помогать, не становясь

излишне навязчивой. И она же становится всё лучше, несмотря на сложные меры, предпринимаемые Apple с целью удержания полного контроля над iOS приложениями и устройствами. Отладчик работает плавно, а симулятор — быстр и отзывчив. Язык Swift более лаконичен, код лучше структурирован и читабелен.

1.2. АНДРОИД

Андроид задумывался и начал писаться примерно в то же самое время (2005 г.), когда iPhone и соответственно iOS были только в проекте, поэтому идеологически это была совершенно иная ОС, созданная в первую очередь для кнопочных аппаратов, но вскоре распространился на смартфоны, планшеты, электронные книги, цифровые проигрыватели, «умные» наручные часы, игровые приставки, нетбуки, смартбуки, Google-очки, телевизоры, системы автоматического управления автомобилем и другие устройства. ОС создана на основе ядра Linux и виртуальной машины Java. Первоначально OS Android разрабатывалась компанией Android Inc, которая была приобретена компанией Google в 2005 г.

Архитектура системы также является уровневой (рис. 2), в ней можно выделить 4 основных уровня (снизу-вверх):

- Уровень ядра Linux;
- Уровень библиотек (Libraries) и уровень среды исполнения (Android Runtime) находятся на одном уровне;
- Уровень каркаса приложений (Application Framework);
- Уровень приложений (Applications).

Уровень ядра Linux. В самом низу, в основе будет располагаться ядро операционной системы. ОС Android основана на ОС Linux версии 2.6, тем самым платформе доступны системные службы ядра, такие как управление памятью и процессами, обеспечение безопасности, работа с сетью и драйверами. Также ядро служит слоем абстракции между аппаратным и программным обеспечением.

Как и в других Linux-системах, ядро Linux обеспечивает такие низкоуровневые вещи, как управление памятью, защита данных,

поддержка мультипроцессности и многопоточности. Но есть исключения: вы не найдёте в Android других привычных компонентов GNU/Linux-систем: здесь нет ничего от проекта GNU, не используется X.Org, и даже systemd. Все эти компоненты заменены аналогами, более приспособленными для использования в условиях ограниченной памяти, низкой скорости процессора и минимального потребления энергии, таким образом, Android больше похож на встраиваемую (embedded) Linux-систему, чем на GNU/Linux.

Рис.2. Архитектура ОС Андроид

Уровень библиотек отвечает за предоставление реализованных алгоритмов для вышележащих уровней, поддержку файловых форматов, осуществление кодирования и декодирования информации, отрисовку графики и многое другое. Библиотеки реализованы на C/C++ и скомпилированы под конкретное аппаратное обеспечение устройства, вместе с которым они и поставляются производителем в предустановленном виде.

Android Runtime – среда выполнения прикладных программ. Ключевыми её составляющими являются набор стандартных библиотек и виртуальная машина Dalvik (с 2014 г. на современных устройствах с процессорами Snapdragon появилась возможность выбрать среду ART, хотя преимущества ее пока еще не очевидны). Каждое приложение в ОС Android запускается в собственном экземпляре виртуальной машины, таким образом, все работающие процессы изолированы от операционной системы и друг от друга. Архитектура Android Runtime такова, что работа программ осуществляется строго в рамках окружения виртуальной машины. Благ. ря этому осуществляется защита ядра операционной системы от возможного вреда со стороны других её составляющих. Поэтому код с ошибками или вредоносное ПО не смогут испортить ОС Android и устройство на её базе. Такая защитная функция, наряду с выполнением программного кода, является одной из ключевых для Android Runtime.

Уровень каркаса приложений (ApplicationFramework). ОС Android позволяет полностью использовать API (интерфейс программирования приложений – набор готовых классов, процедур, функций, структур и констант), используемый в приложениях ядра. Архитектура построена таким образом, что любое приложение может использовать уже реализованные возможности другого приложения при условии, что последнее откроет доступ на использование своей функциональности. Таким образом, архитектура реализует принцип многократного использования компонентов ОС и приложений.

Версии.

Изначально Google рассчитывала давать версиям Android имена известных роботов, но отказалась из-за проблем с авторскими правами. Поэтому каждая версия системы, начиная с версии 1.5, получает собственное кодовое имя на тему сладостей. Кодовые имена присваиваются в алфавитном порядке латинского алфавита. Таких сладостей (как и версий) накопилось уже десять, название версии 10 начинается на букву Q.

Следует также отметить, что для каждой версии Android существует несколько версий API-версий, на момент написания этой книги их уже 29, что вызывает проблемы при разработке приложений, т.е. когда на старых устройствах вызывается новый код. К

примеру, версия построения выше минимальной и вызывается класс или метод, присутствующий в версии построения, но отсутствующий в минимальном SDK. Конечно, для решения этой проблемы уже существует несколько вариантов: можно поднять минимальную версию приложения (что отрежет часть потребительской аудитории) либо включить обработку нового кода в условную конструкцию следующим образом:

```
if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
    //новый код
    } else {
    //старый код
    }
```

Как видим, оба варианта несут в себе существенные недостатки.

Языки программирования.

Официальный язык программирования, поддерживаемый средой разработки Android Studio, – Java. В 2018 г. Java вошёл в пятёрку самых популярных языков программирования.

При разработке на Java под Android используются не только Java-классы, содержащие код, но также файлы манифеста на языке XML, предоставляющие системе основную информацию о программе, и системы автоматической сборки Gradle, Maven или Ant, команды в которых пишутся на языках Groovy, POM и XML соответственно.

В мае 2017 г. был представлен язык Kotlin, который позиционируется Google как второй официальный язык программирования под Android после Java, только чуть более простой для понимания. Знания Java все же необходимы здесь, чтобы понимать принципы работы Kotlin, общую структуру языка и его особенности. Многие разработчики считают Kotlin обёрткой над Java и рекомендуют изучать его только после того, как вы почувствуете уверенность в своих знаниях Java.

Kotlin совместим с Java и не вызывает снижения производительности и увеличения размера файлов. Отличие от Java в том,

что он требует меньше служебного, так называемого boilerplate-кода, поэтому более обтекаемый и лёгкий для чтения.

Более низкоуровневые языки также поддерживаются Android Studio с использованием Java NDK (Native Development Kit), например, C++. Это позволяет писать нативные приложения, что может пригодиться для создания игр или других ресурсоёмких программ, однако код будет запускаться не через Java Virtual Machine, а непосредственно через устройство, что дает больше контроля над такими элементами системы, как память, сенсоры, жесты и т. д., а также возможность выжать из Android-устройств максимум ресурсов. Это также означает, что пользоваться придётся библиотеками, написанными на C или C++.

Такой способ считается сложным в настройке и не слишком удобным, поэтому рекомендуется использовать его для написания только тех модулей программы, где необходимо быстро производить сложные операции: обработку и рендеринг графики, видео и сложных 3D-моделей.

Существует также возможность привлекать другие языки для разработки Android-приложений, например, Python, но такая возможность еще не приобрела широкого распространения.

Достоинства ОС Android.

Несомненным плюсом Андроида является его открытость. Это позволяет любому желающему создавать свои приложения, игры и прочие дополнения для расширения возможностей Андроид-гаджетов. Разработчики изначально продумали все так, чтобы операционная система работала максимально быстро даже на самом «бюджетном» железе. Это является несомненным плюсом, так как теперь люди даже с самыми скромными финансовыми возможностями могут наслаждаться основными преимуществами современных смартфонов, впрочем, столь большой охват порождает и основные недостатки разработки для Android. Если ориентироваться на широкий спектр устройств, придется учитывать разную производительность, бесконечное количество размеров экрана и память. Как следствие, увеличенные затраты на проектирование нескольких интерфейсов и дополнительное тестирование. Стоимость разработки в таком случае увеличивается пропорционально количеству поддерживаемых устройств.

1.3. Другие мобильные операционные системы

Среди других известных мобильных ОС – Windows Phone, Symbian, BlackBerry OS, Tizen.

Windows Phone уже сильно отстает от конкурентов: iOS впереди всех, Android догоняет, об этом свидетельствует в том числе и количество приложений для систем:

- iOS—1,2 млн;
- Android—1,2 млн;
- Windows Phone—245 тыс.

Windows Phone вышла 11 октября 2010 г., а значит является самой молодой в тройке лидеров. Она же раньше всех прекращает существование – официально поддержка операционной системы версии прекратилась 10 декабря 2019 г.

На этом прервем обзор мобильных ОС, приведем лишь несколько интересных фактов. Например, руководство Google утвердило сроки прекращения развития операционной системы Android. Ей на смену придет абсолютно новая, о которой сейчас ходит много слухов. Новая ОС носит название Fuchsia OS, однако когда она выйдет на рынок, то, скорее всего, именоваться будет иначе. Зачем нужна другая система, если Android вполне успешна?

Конечная цель Google заключается не в том, чтобы интегрировать ОС со своими сервисами, а в том, чтобы сделать саму операционную «Гуглом», с тем, чтобы она выполняла просьбы пользователя и подстраивалась под него. Android плохо подходит для решения такой задачи.

Ключевые компоненты Fuchsia — это не файлы и приложения, как в классических операционных системах, а сущности и агенты. Сущностями в «Фуксии» может быть все что угодно: место, человек, событие, вещь, email и т.д. Это единицы информации, которые позволяют операционной системе «понимать», с чем имеет дело пользователь. Ждать осталось недолго – предположительно до 2022 г.

Еще одной альтернативой для Android на территории России модет стать система Аврора. Мобильная ОС Sailfish (под этим именем Аврора проходила все сертификационные мероприятия) является продуктом сотрудников финской компании Jolla, которую ос-

новали бывшие работники финского гиганта телекоммуникационного рынка Nokia. Еще в 2016 г. партнером финской компании стало российское ООО «Открытая мобильная платформа», а в 2018 г. 75% акций разработчика операционной системы выкупил «Ростелеком». Тогда же было принято решение о том, что российская операционная система должна получить русское название.

Её основные потребители — государственные структуры. Для представителей государственных компаний, крупного бизнеса, сотрудников МВД и Минобороны это уже готовая система, которая обеспечит им устойчивую связь и систему коммуникаций в экстренных случаях. И для этой системы пока не существует приложений, но «Ростелеком» уже начал привлекать разработчиков приложений и игр для того, чтобы они писали софт для «Авроры», делая ОС, более ориентированной на простых россиян.

1.4. Проектирование и реализация мобильных приложений

1.4.1. Принципы проектирования объектно-ориентированных приложений

Разработка любого приложения начинается с его проекта. Поэтому прежде чем приступить к практике разработки ООП приложений, рассмотрим основные принципы проектирования хорошего ПО. Для того чтобы помочь программистам разрабатывать качественные ООП-приложения, Роберт Мартин – автор книги «Чистая архитектура. Искусство разработки программного обеспечения» – разработал пять принципов объектно-ориентированного программирования и проектирования, говоря о которых, с подачи Майкла Фэзерса, используют акроним SOLID. Перечислим эти принципы:

— S: Single Responsibility Principle (принцип единственной ответственности);

— O: Open-Closed Principle (принцип открытости – закрытости);

— L: Liskov Substitution Principle (принцип подстановки Барбары Лисков);

— I: Interface Segregation Principle (принцип разделения интерфейса);

— D: Dependency Inversion Principle (принцип инверсии зависимостей).

Подробный разбор этих принципов с примерами займет отдельную книгу, поэтому остановимся здесь на кратких пояснениях каждого принципа SOLID. К слову, такая книга уже есть, и это упомянутая книга Роберта Мартина.

Принцип единственной ответственности оказывается самым сложным для понимания. Принцип гласит: каждый класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

Принцип открытости – закрытости означает, что готовый код должен легко видоизменяться путем добавления нового кода, а не путем видоизменения уже имеющегося. Другими словами, программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Принцип подстановки Барбары Лисков состоит в возможности использования классов-наследников вместо родительских классов, от которых они образованы, без нарушения работы программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

Принцип разделения интерфейса (здесь под интерфейсом подразумевается набор методов) направлен на устранение недостатков, связанных с реализацией больших интерфейсов. Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.

Принцип инверсии зависимостей реализует принцип наследования и принцип абстракции в ООП и означает, что объектом зависимости должна быть абстракция, а не что-то конкретное:

— модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций;

— абстракции не должны зависеть от деталей, наоборот, детали должны зависеть от абстракций.

В лабораторных работах мы будем учиться придерживаться этих важных принципов.

1.4.2. Особенности разработки мобильных приложений

Существует три подхода технической реализации приложений для мобильных устройств.

Мобильное native-приложение — это специально разработанное приложение под конкретную мобильную платформу (iOS, Android, Windows Phone). Такое приложение разрабатывается на языке высокого уровня и компилируется в так называмый native-код ОС, обеспечивающий максимальную производительность. Главным недостатком мобильных приложений этого типа является низкая переносимость между мобильными платформами.

Мобильное web-приложение — специализированный web-сайт, адаптированный для просмотра и функционирования на мобильном устройстве. Такое приложение хоть и не зависит от платформы, однако требует постоянного подключения к сети, так как физически размещено не на мобильном устройстве, а на отдельном сервере.

Гибридное приложение — мобильное приложение, упакованное в native-оболочку. Такое приложение, как и native, устанавливается из онлайн-магазина и имеет доступ к тем же возможностям мобильного устройства, но разрабатывается с помощью web-языков HTML5, CSS и JavaScript. В отличие от native-приложения является легко переносимым между различными платформами, однако несколько уступает в производительности.

Каждый из перечисленных видов имеет свои особенности реализации, однако есть и некоторые общие проблемы. Мы сосредоточимся в основном на нативных приложениях и сложностях в их разработке. Разработчикам придется столкнуться со следующими проблемами:

— *ограничение работы батареи и памяти мобильного устройства.* Самые яркие, удобные и функциональные приложения в первую очередь сталкиваются с суровой реальностью ограничений по ресурсам. Это значит, что необходимо экономно относиться к расходованию памяти при разработке приложения, а

также сохранять и затем текущее состояние ввода пользователя, с учетом жизненного цикла мобильного приложения;

— *зависимость от сети.* Любое приложение должно уметь адаптироваться и существовать как в условиях доступа к информации в сети Интернет, так и в автономном режиме. Во-первых, нужно уметь корректно обрабатывать ошибки, которые могут быть самыми разными: от отсутствия Интернета и неправильных параметров в запросе до не отвечающего сервера и ошибок в ответе. Во-вторых, в вашем приложении может быть не один запрос, а много, и вполне возможна ситуация, что вам придется комбинировать результаты этих запросов сложным образом: выполнять их параллельно, использовать результат предыдущего запроса для выполнения, следующего и т.д. В-третьих, и это самое неприятное – запросы могут занимать значительное время, а пользователь часто не самый терпеливый и тихий человек – он может крутить устройство (и тогда вы потеряете текущие данные в Activity), а может и вовсе закрыть приложение, и тогда вы можете получить рассинхронизацию в данных (когда на сервере данные обновились, а приложение не знает об этом и отображает некорректную или устаревшую информацию). И все это нужно каким-то образом решать. Отдельно стоит вопрос его работы в фоновом режиме;

— *обновление и добавление новых данных.* Главным способом распространения приложений остаются их магазины: AppStore, Google Play, Windows Store. В то время как новые возможности веб-сайтов могут отражаться моментально, мобильные приложения должны проходить проверку в своих магазинах. Эта проверка может занимать от пары часов до нескольких месяцев;

— *обеспечение возможности тестирования классов, содержащих бизнес-логику приложения.* Это также подразумевает под собой немало внутренних проблем. Во-первых, нужно обеспечить модульность классов. Для этого нужно разделять классы по логическим слоям для каждого экрана. Т.е. вместо того, чтобы писать весь код, относящийся к одному экрану, в одной активити, нужно грамотно разделить его на несколько классов, каждый из которых будет иметь свою зону ответственности. Во-вторых, если говорить о тестах с помощью JUnit (библиотека для модульного тестирования программного обеспечения на языке Java), то нужно понимать,

что тестируемые таким образом классы должны содержать минимальное количество зависимостей от Android-классов, так как Java и ее виртуальная машина об этих классах не знает ничего. В-третьих, самая сложная логика приложения почти всегда связана с работой с данными от сервера (об этот уже упоминалось). Мы должны протестировать различные возможные ситуации, такие как ожидаемый ответ сервера, ошибка сервера и разные ответы, приводящие к разному поведению приложения. Но при выполнении теста мы не можем по своему желанию «уронить» сервер или заставить его отдать нужные нам данные. К тому же серверные запросы выполняются долго и асинхронно, а тесты должны работать последовательно. Все эти проблемы можно решить, если подменять реализацию сервера на определенном слое, к которому будут обращаться тестируемые классы.

2. ПРИЕМЫ ОБЪЕКТНОГО ПРОГРАММИРОВАНИЯ НА JAVA (JAVA для Android-разработчиков)

Напомним, что Java был первым языком для разработки Android-приложений. Все основополагающие принципы этой технологии (Java – не только название языка, но название целой технологии) перекочевали на ОС Android, где постепенно начали изменяться и приобретать индивидуальные черты дочерней системы.

2.1. Перед началом работы. Установка компонентов среды разработки

Концепция платформы JAVA

Java – так называется язык программирования, но под этим же названием кроется целая технология, составные компоненты которой представлены на рис.3 (рисунок взят из документации по Java).

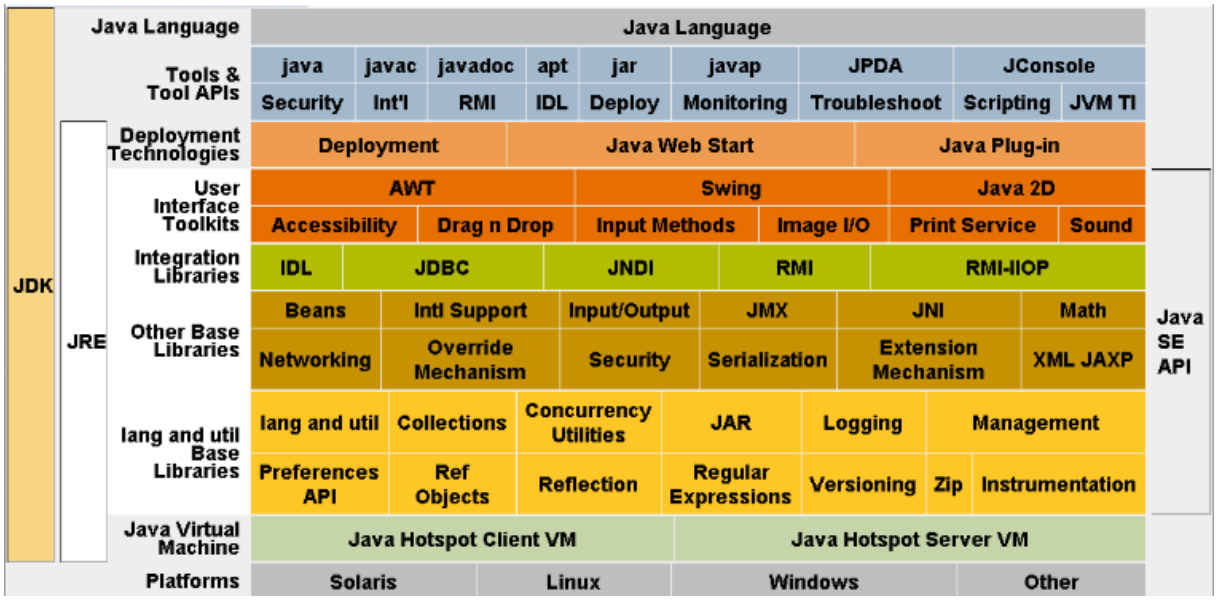


Рис.3. Структура платформы JAVA SE

Этот рисунок нужен нам для понимания того, какие роли в технологии играют необходимые нам для работы компоненты JDK и JRE.

Java Development Kit (сокращенно JDK) — бесплатно распространяемый компанией Oracle Corporation (ранее Sun Microsystems) комплект разработчика приложений на языке Java, включающий компилятор Java (javac), стандартные библиотеки

классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE). В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден применять внешний текстовый редактор и компилировать свои программы с помощью утилит командной строки.

Все современные интегрированные среды разработки приложений (IDE) на Java, такие, как JDeveloper, NetBeans IDE, Sun Java Studio Creator, IntelliJ IDEA, Borland JBuilder, Eclipse, опираются на сервисы, предоставляемые JDK. Большинство из них для компиляции Java-программ используют компилятор из комплекта JDK. Поэтому эти среды разработки либо включают в комплект поставки одну из версий JDK, либо требуют для своей работы предварительной инсталляции JDK на машине разработчика.

Установка программного обеспечения

NetBeans IDE по праву входит в пятерку лучших сред разработки, поддерживающих Java (есть также IDEA, Eclipse и др.). Данная IDE позволяет разрабатывать мобильные и корпоративные приложения, а также кроссплатформенное ПО для компьютера. Основная прелесть среды программирования NetBeans - поддержка большого числа технологий (от фиксации ошибок до рефакторинга), шаблонов без дополнительных настроек и языков программирования. таких как PHP, HTML, JavaScript, C, C ++, Ajax, JSP, Ruby и др. Всё, что необходимо для работы начинающему разработчику, уже заложено в базовый пакет.

В сентябре 2016 г. Oracle передала интегрированную среду разработки NetBeans в руки фонда Apache и это изменило политику работы с IDE: теперь существует два вида JDK – OpenJDK и Oracle JDK. OpenJDK — это эталонная реализация JDK (полностью бесплатная и распространяется под GPL), Oracle JDK базируется на OpenJDK. Oracle JDK отличается от OpenJDK наличием платной поддержки.

Последние версии NetBeans:

Текущая стабильная версия среды – NetBeans IDE 8.2.

Apache NetBeans 9.0, опубликована в июле 2018 г. Добавляет поддержку Java 9 и 10.

Apache NetBeans 10.0 выпущена 27 декабря 2018 г. Была добавлена поддержка Java 11 и улучшена поддержка PHP (7.0–7.3).

Для установки NetBeans (а затем и Android SDK) потребуется JDK версии не ниже 7 (на момент написания данного методического руководства вышла только 12-я версия Oracle JDK).

Желающим поэкспериментировать можно установить Apache NetBeans 11 на основе Oracle JDK 12.

Однако мы будем устанавливать не самую последнюю версию – NetBeans IDE 8.2. Связано это с тем, что в последних версиях пока отсутствует поддержка библиотек, которые вскоре нам понадобятся (это MPJ Express и JADE).

Ссылка для загрузки NetBeans IDE 8.2:

https://netbeans.apache.org/download/index.html, ниже представлен фрагмент страницы (рис. 4).

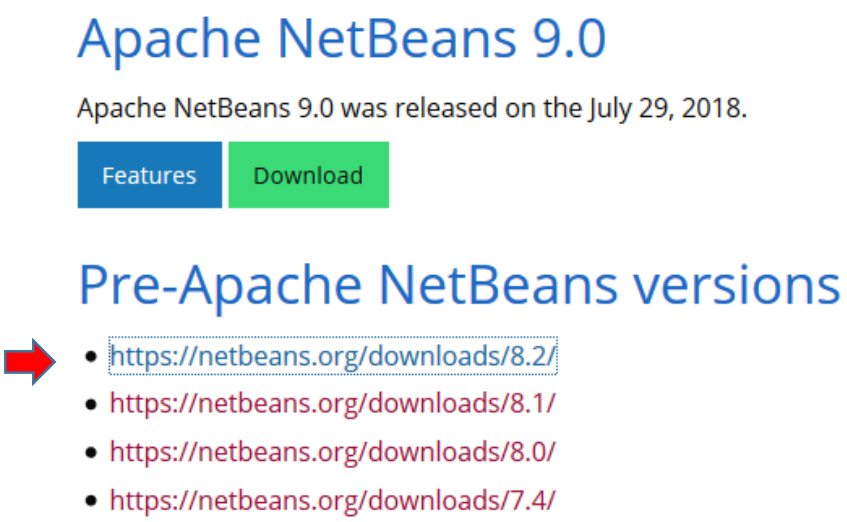


Рис.4. Страница загрузки NetBeans IDE 8.2

При переходе по указанной ссылке вам будет предложено несколько вариантов сборок IDE для скачивания (рис.5), здесь рассмотрим установку Java-сборки среды. Любые модули можно добавить или удалить впоследствии.

Выберите язык пользовательского интерфейса, вашу операционную систему и установщик с компонентами, которые вас интересуют (нам в этом курсе вполне хватит Java SE).

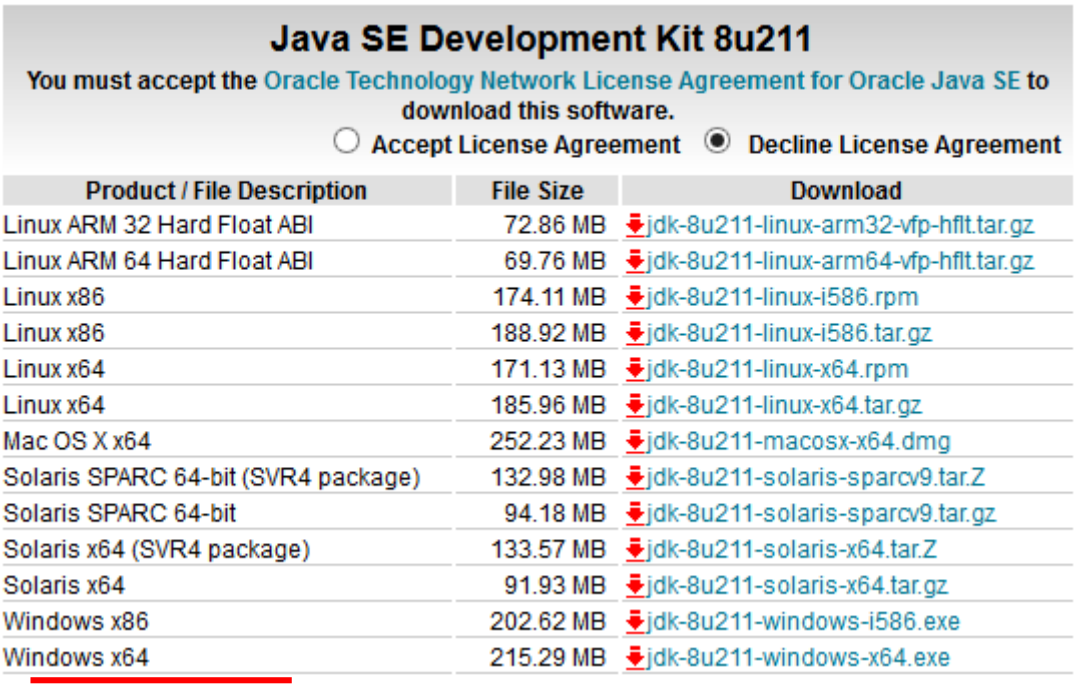
Рис.5. Страница выбора сборки среды NetBeans IDE 8.2

Сохраняем бинарный файл netbeans-8.2-javase-windows.exe, чтобы после установки JDK его запустить.

JDK скачиваем по ссылке:https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html либо выбирая сборку для Windows x64 (рис.6).

Дальше все как обычно: выбираем путь для установки JRE (как правило, это: C:\Program Files\Java\jre8). Путь для установки JDK (по умолчанию: C:\Program Files\Java\jdk1. jdk1.8.0_20).

NetBeans устанавливается по умолчанию в директорию C:\Program Files\ NetBeans номер версии.



Java SE Development Kit 8u211

You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.

○ Accept License Agreement ◉ Decline License Agreement

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.86 MB	jdk-8u211-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	69.76 MB	jdk-8u211-linux-arm64-vfp-hflt.tar.gz
Linux x86	174.11 MB	jdk-8u211-linux-i586.rpm
Linux x86	188.92 MB	jdk-8u211-linux-i586.tar.gz
Linux x64	171.13 MB	jdk-8u211-linux-x64.rpm
Linux x64	185.96 MB	jdk-8u211-linux-x64.tar.gz
Mac OS X x64	252.23 MB	jdk-8u211-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	132.98 MB	jdk-8u211-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	94.18 MB	jdk-8u211-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	133.57 MB	jdk-8u211-solaris-x64.tar.Z
Solaris x64	91.93 MB	jdk-8u211-solaris-x64.tar.gz
Windows x86	202.62 MB	jdk-8u211-windows-i586.exe
Windows x64	215.29 MB	jdk-8u211-windows-x64.exe

Рис.6. Страница выбора версии и ОС для установки JDK

ВНИМАНИЕ! Для выполнения ЛР1 (работа с командной строкой) необходимо прописать системную переменную среды. Для этого в переменную окружения ***PATH*** операционной системы вносится ***путь к каталогу BIN JDK***, например, ***C:\Program Files\Java\jdk1.8.0_60\bin***. Для того чтобы добраться до переменной окружения, нужно зайти в ***Панель управления-->Система-->Дополнительные параметры системы-->Переменные среды -->Создать (либо Изменить)***.

Добавляем также переменную ***JAVA_HOME***, устанавливаем ее в значение ***C:\Program Files\Java\jdk1.8.0_60.*** Последняя часть адреса зависит от установленной у Вас версии.

Переменная *PATH* — это системная переменная, которую операционная система использует для того, чтобы найти нужные исполняемые объекты в командной строке или окне терминала.

Проверка корректности установки системной переменной. Как только вы закончите с указанными изменениями, попробуйте следующие шаги. Если вы не видите результатов, сообщающих версию java, перезагрузите компьютер и попробуйте снова. Если это не сработает, возможно, вам придется переустановить JDK.

Откройте командную строку Windows (клавиша Windows + R → введите cmd → OK) и проверьте следующее:

Вы увидите что-то вроде этого (версия может быть более новой):

java version "1.8.0_60"

Java(TM) SE Runtime Environment (build 1.8.0_144-b01)

Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode).

ЛАБОРАТОРНАЯ РАБОТА №1

РАБОТА В КОМАНДНОЙ СТРОКЕ – КОМПИЛЯЦИЯ И ЗАПУСК НА ВЫПОЛНЕНИЕ

Разработка и выполнение простой программы на Java

Цель. Освоить основы работы в командной строке Java. Изучить структуру и синтаксис простой программы. Посмотреть возможности языка.

Для того, чтобы посмотреть возможности языка, попробуем запустить несколько примеров.

В любом текстовом редакторе создаем файл JavaApplication1.java (имя файла должно совпадать с именем класса) и пишем код для калькулятора, состоящий из нескольких классов:

Листинг 1

```
public class JavaApplication1 {

public static void main(String[] args) {
        System.out.println("Hello World!");
        Calculator calc=new Calculator();
        System.out.println("2+2="+calc.sum(2,2));
         }
public static class Adder
{
   private int sum;
   public Adder() { sum=0; }
   public Adder(int a){ this.sum=a;}

   public void add(int b)
   {
      sum+=b;
   }
   public int getSum() { return sum;}
}
```

```
public static class Calculator
{
   public int sum(int... a)
   {
      Adder adder=new Adder();
      for(int i:a)
      {
         adder.add(i);
      }
      return adder.getSum();
   }
}
      }
```

Далее необходимо скомпилировать программу в командной строке. Для этого запускаем *cmd,* переходим в директорию, где хранится файл, и выполняем компиляцию:

```
javac JavaApplication1.java
```

в результате получаем скомпилированные файлы JavaApplication1.class, JavaApplication1$Adder.class, JavaApplication1$Calculator.class, т.е. на основе каждого класса из Листинга 1 создается отдельный файл класса.

Запуск программы из командной строки:

```
java -classpath . JavaApplication1
```

Получаем:

```
"Hello World!"
2+2=4
```

директива -classpath . указывает класс, который нужно запустить на исполнение, это всегда главный класс программы.

Дополнительные примеры

Пример 1.1. Примитивное окно приложений с обработкой события Close.

Можем ли мы создать полноценное окно (как в Windows) из программы, написанной на языке Java? Конечно, можем, и для этого существует класс Frame (рамка). Если унаследовать наш класс от класса Frame, то мы получим окно со всеми его атрибутами. Запуск на выполнение потомка класса Frame происходит из функции main.

Листинг 2

```
import java.awt.*;
import java.awt.event.*;

//Создаем наш класс, наследуясь от стандартного класса Frame
class SimpleFrame extends Frame{
  SimpleFrame(String s){
    super(s);       //Передаем название окна в родительский класс
    setSize(400, 150); //Устанавливаем размер окна
    setVisible(true);  //Окно становится видимым
//Добавляем слушателя, который будет реагировать на кнопку
//закрытия окна
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent ev){
  dispose();   //Закрываем окно
  System.exit(0); //Выходим из программы
      }
    });
  }
//Создаем экземпляр нашего класса в главном классе
 public static void main(String[] args){
    new SimpleFrame(" Моя программа");
  }
}
```

Кнопка закрытия окна работать не будет, если не написать специальный код, который присутствует в приведенной программе (код располагается в конструкторе) и обрабатывает событие нажатия на кнопку закрытия окна:

```
addWindowListener(
        new WindowAdapter()
```

```
{       public void windowClosing(WindowEvent ev)
        {
                dispose();
                System.exit(0);
        }
});
```

Здесь мы увидели пример определения безымянного класса. Рассмотрим его подробнее. Мы вызываем метод addWindowListener для того, чтобы назначить слушателя оконных событий. В качестве параметра создаем объект класса WindowAdapter. Но этот класс является абстрактным – у него нет имени. Поэтому мы неявно создаем производный от него класс и переопределяем нужные нам методы – в данном случае обработку события закрытия окна (метод dispose уничтожает объект Frame), затем останавливаем виртуальную машину Java вызовом метода System.exit(0). Откомпилируем и запустим программу. При компиляции будет создан класс с именем simpleFrame$1.class.

Пример 1.2. Окно приложения с выводом текста.

Листинг 3

```
//Подключаем библиотеку для работы с графикой
import java.awt.*;
import java.awt.event.*;

//Класс с полем для рисования
class HelloWorldFrame extends Frame{
   HelloWorldFrame(String s){
   super(s);
  }
  public void paint(Graphics g){
   g.setFont(new Font("Serif", Font.ITALIC | Font.BOLD, 30));
   g.drawString("Hello, XXI century World!", 20, 100);
  }
  public static void main(String[] args){
   Frame f = new HelloWorldFrame("Здравствуй, мир XXI века!");
   f.setSize(400, 150);//Задаем размер окна
```

```
    f.setVisible(true);//Делаем его видимым
    f.addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent ev){
         System.exit(0);
       }
    });
  }
}
```

Пример 1.3. Окно приложений с графическим примитивом линии.

Листинг 4

```
import java.awt.*;
import java.awt.event.*;

class GraphTest01 extends Frame{
  GraphTest01(String s) {
    super(s);
    setBounds(0, 0, 500, 300);
    setVisible(true);
  }
  public void paint(Graphics g){
    Dimension d = getSize();
    int dx = d.width / 20, dy = d.height / 20;
    int myWidth = 250, myHeight = 250;
    g.drawLine(0, 0, myWidth, myHeight);
    g.drawLine(0, 0, d.width, d.height);
    setBackground(Color.blue);
    setForeground(Color.red);
 }

  public static void main(String[] args){
    GraphTest01 f = new GraphTest01(" Пример рисования");
    f.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent ev){
          System.exit(0);
        }
    });
```

```
    }
}
```

Справочная информация по классу Graphics

Класс Graphics – работа с графикой осуществляется через методы paint(), update().

```
// Рисует линию.
void drawLine(int startX, int startY, int endX, int endY);
// Контурный прямоугольник.
void drawRect(int top, int left, int width, int height);
// Закрытый прямоугольник.
void fillRect(int top, int left, int width, int height);
// Контурный прямоугольник с закругленными углами.
void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam);
// Закрытый прямоугольник с закругленными углами.
void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam);
// Контурный эллипс внутри заданного прямоугольника.
void drawOval(int top, int left, int width, int height);
// Закрытый эллипс внутри заданного прямоугольника.
void fillOval(int top, int left, int width, int height);
// Контурная дуга.
void drawArc(int top, int left, int width, int height, int startAngle,
int sweepAngle);
// Контурный многоугольник.
void drawPolygon(int x[], int y[], int numPoints);
// Закрытый многоугольник.
void fillPolygon(int x[], int y[], int numPoints).
```

Задания к лабораторной работе №1 (ч. 1)

Задание 1.

1. Скомпилируйте и выполните программу Калькулятор (Листинг 1) и примеры 1.1–1.3 из командной строки.

2. Используя справочную информацию по классу Graphics, поэкспериментируйте с графическими примитивами (нарисуйте круг, квадрат, картинку из примитивных фигур и линий).
3. Сформулируйте выводы о структуре Java-программы и методах использования классов.

Работа с пакетами и Jar – архивами Java (ч. 2)

Цель. Освоить основы работы с пакетами Java и архивами jar.

Мотивация. Некоторые программные продукты Java вообще не работают с безымянным пакетом (которые создаются по умолчанию). Поэтому в технологии Java рекомендуется все классы помещать в пакеты. Пакеты Java — это механизм, который служит для работы с пространством имен и имеет свой синтаксис.

Сведения о работе с пакетами

В стандартную библиотеку Java API входят сотни классов. Каждый программист в ходе работы добавляет к ним десятки своих классов. Множество классов растет и становится необозримым. Уже давно принято отдельные классы, решающие какую-то одну определенную задачу, объединять в библиотеки классов. Но библиотеки классов, кроме стандартной библиотеки, не являются ч. ю языка.

Разработчики Java включили в язык дополнительную конструкцию — пакеты (packages). Все классы Java распределяются по пакетам. Кроме классов пакеты могут содержать интерфейсы и вложенные подпакеты (subpackages). Образуется древовидная структура пакетов и подпакетов.

Эта структура в точности отображается на структуру файловой системы. При попытке поместить класс в пакет вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема. Представьте себе, что вы написали класс с именем PackTest в пакете test. Вы создаете каталог test, помещаете в этот каталог файл PackTest.Java и транслируете. Пока — все в порядке. Однако при

попытке запустить его вы получаете от интерпретатора сообщение «can't find class PackTest» («Не могу найти класс PackTest»). Ваш новый класс теперь хранится в пакете с именем test, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками - test.PackTest. Кроме того, Вам надо либо подняться на уровень выше в иерархии каталогов и снова набрать «java test.PackTest», либо внести в переменную CLASSPATH каталог, который является вершиной иерархии разрабатываемых вами классов.

Итак, все файлы с расширением class (содержащие байт-коды), образующие один пакет, хранятся в одном каталоге файловой системы. Подпакеты образуют подкаталоги этого каталога.

Каждый пакет создает одно пространство имен (namespace). Это означает, что все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если надо в одном месте программы использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: ***пакет.Класс.*** Такое уточненное имя называется полным именем класса (fully qualified name).

Если член класса не отмечен ни одним из модификаторов private, protected, public, то по умолчанию к нему осуществляется **пакетный доступ (default access), т.е. к такому члену может обратиться любой метод любого класса, но только из того же пакета. Если класс не помечен модификатором public, то все его члены, даже открытые, public, не будут видны из других пакетов.**

Члены с пакетным доступом не видны в подпакетах данного пакета.

Сведения о работе с архивами

Почему мы уделяем особое внимание архивам? Потому что архивные файлы *.jar часто используются для распространения Java-приложений, для передачи их по сети и даже для запуска приложений. Запустить на исполнение архивный файл даже легче, чем неархивный.

Для упаковки нескольких файлов в один архивный файл, со сжатием или без сжатия, в технологии Java разработан формат архивирования JAR. Имя архивного jar-файла может быть любым, но обычно оно получает расширение jar. Способ упаковки и сжатия основан на методе ZIP. Название JAR (Java ARchive) перекликается с названием известной утилиты TAR (Tape ARchive), разработанной в UNIX. Отличие jar-файлов от zip-файлов только в том, что в jar-файлы автоматически включается каталог META-INF, содержащий несколько файлов с информацией об упакованных в архив файлах.

Предположим, в нашей программе присутствуют два класса – ***Hello.class*** и ***Word.class,*** и только в ***Hello*** находится стартовая точка main(), тогда команда создания архива должна выглядеть так:

Рис.7. Схема создания архива из командной строки

Команду можно упростить, если все упаковываемые классы находятся в одной директории, например, hello:

```
jar cfm HelloWord.jar Hello hello/*
```

К сожалению, созданный архив запускаться не будет, для обеспечения его работы имя основного класса приложения, содержащего метод main(), должно быть указано в файле MANIFEST.MF, который автоматически создается при создании архива, но не содержит имени главного класса.

Поэтому следует выполнить следующий набор команд:

```
// Просмотр содержимого архива
>jar tf HelloWord.jar
// Извлечение содержимого из jar-файла
>jar xf HelloWord.jar
>MANIFEST.MF
```

Файл манифеста откроется в блокноте. В него необходимо добавить строку

Main-Class: hello. HelloWord

и пустую строку.

Обратите внимание, что главный класс вписан без расширения.

После редактирования необходимо собрать архив с новым файлом манифеста:

jar cvfm HelloWord.jar MANIFEST.MF Hello hello/*

Архивные файлы удобно использовать и в приложениях (applications). Все файлы приложения упаковываются в архив. Приложение выполняется прямо из архива, интерпретатор запускается с параметром –jar, например:

java –jar HelloWord.jar

Jar-архивы создаются с помощью классов пакета java.util.jar или посредством утилиты командной строки jar.

Задания к лабораторной работе №1 (ч. 2)

Задание 2.

Возьмите текст программы из первой части лабораторной работы – JavaApplication1.java (Листинг 1) и разделите классы так, чтобы все они находились в разных пакетах (структура приложения «Калькулятор» показана на рис.8).

Соглашение «Code Conventions рекомендует записывать имена пакетов строчными буквами. Тогда они не будут совпадать с именами классов, которые, по соглашению, начинаются с прописной буквы. Кроме того, соглашение советует использовать в качестве имени пакета или подпакета доменное имя своего сайта, записанное в обратном порядке, например, com.sun.developer.

Это обеспечит уникальность имени пакета во всем Интернете.

Создаем структуру приложения:

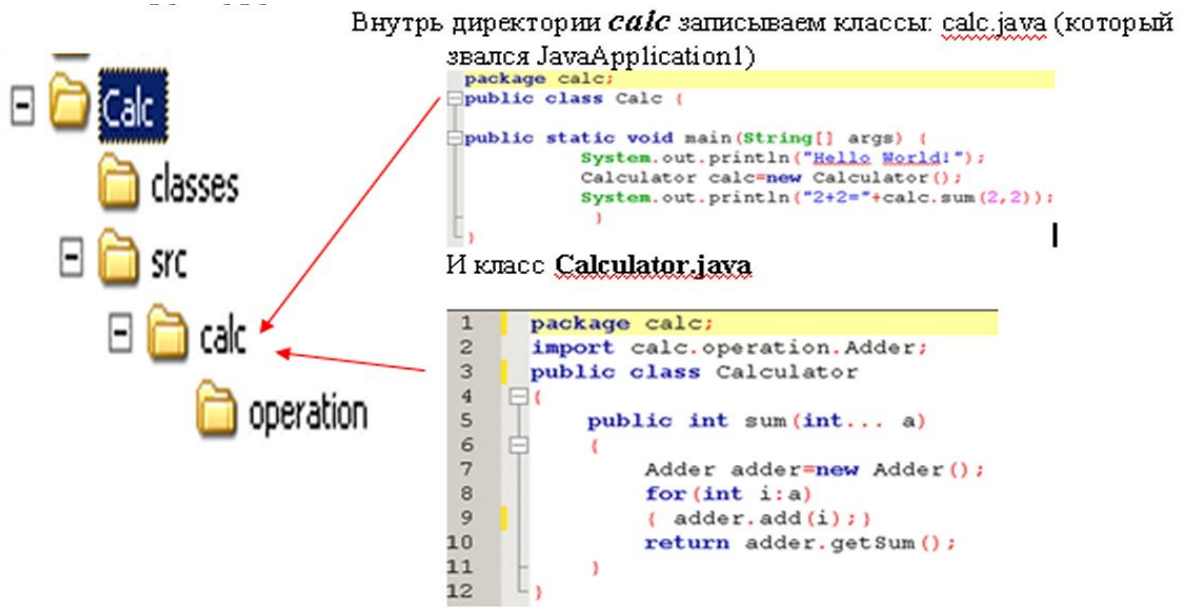


Рис.8. Структура JAVA–приложения «Калькулятор» с делением на пакеты

```
package calc.operation;
public class Adder
{
    private int sum;
    public Adder()
    { sum=0;
    }
    public Adder(int a)
    { this.sum=a;
    }
    public void add(int b)
    { sum+=b;
    }
    public int getSum()
    {
        return sum;
    }
}
```

Рис.9. Класс Adder, выполняющий суммирование переменных, находится в директории ***operation***

Таким образом, в директории calc будут находится классы Calc.java и Calculator.java. В директории operation сохраняем класс **Adder** (рис. 9).

Компилируем с указанием следующих параметров:

– **sourcepath –** пути, где находятся файлы-источники – это директория **src**;

– ключ ***d*** обозначает путь, куда необходимо положить классы, в нашем случае – это директория ***classes***;
– далее необходимо указать путь к главному классу: ***src/calc/Calc.java***:

…Calc>javac -sourcepath src -d classes src/calc/Calc.java

При этом директории ***calc*** и ***operation*** будут созданы компилятором внутри директории ***clssses*** автоматически.

Запускаем на выполнение так:
Calc\classes>java -classpath . calc/Calc
Или так:
Calc\classes>java -classpath . calc.Calc

Здесь обращаем ваше внимание на два момента класса *Calculator*: попробуйте объяснить смысл обведенных фрагментов кода (рис. 10).

```
1  package calc;
2  import calc.operation.Adder;
3  public class Calculator
4  {
5      public int sum(int... a)
6      {
7          Adder adder=new Adder();
8          for(int i:a)
9          { adder.add(i);}
10         return adder.getSum();
11     }
12 }
```

Рис.10. Особенные методы класса Calculator

Для справки приведем здесь все возможные модификаторы доступа языка Java (табл. 1). Заголовки столбцов указывают, в каких пределы разрешает доступ каждый модификатор.

Таблица 1

Модификаторы доступа к полям и методам класса

Модификаторы доступа	Класс	Пакет	Пакет и подклассы	Все классы
private	+			

"package"	+	+		
protected	+	+	*	
public	+	+	+	+

*Особенность доступа к protected-полям и методам из чужого пакета отмечена звездочкой.

Задание 3.

1. Доработайте приложение «Калькулятор», дополнив его основными математическими действиями: сложение (уже есть), вычитание, умножение, деление и дополнительная операция с учетом типов операндов согласно вариантам (табл. 2). Тип переменной указан для всех реализуемых операций. При этом каждая операция должна быть описана отдельным классом и находиться в отдельном файле.

Таблица 2

Варианты для лабораторной работы 1

Вариант	Тип переменной	Дополнительная операция
1	Byte	Сдвиг влево на 1
2	Short	Сдвиг вправо на 1
3	Int	Возведение в квадрат
4	Long	Сдвиг влево на *x* позиций
5	Double	Сдвиг вправо на *x* позиций
6	Float	Остаток от деления
7	Int	Инвертирование
8	Short	Логическое умножение

2. Выполните:

- компиляцию классов с размещением в указанной папке classes. Исходные файлы находятся в каталоге ***project\src\;***
- запуск приложения из папки ***classes;***
- создание архива jar для всего проекта, запуск приложения из jar архива.

Требования к отчету к ЛР №1

1. Титульный лист.
2. Постановка задачи.
3. Листинг к заданию 1.3. Скриншоты, демонстрирующие создание архива и его запуск.

ЛАБОРАТОРНАЯ РАБОТА №2

РАЗРАБОТКА ПРОГРАММЫ В СРЕДЕ NETBEANS, ОСНОВНЫЕ КОНЦЕПЦИИ ООП, ПРОСТЕЙШИЕ UML-ДИАГРАММЫ

Наследование и реализация полиморфизма в Java

Цель. Освоить основы работы со средой разработки NetBeans. Изучить структуру проекта. Освоить процесс построения иерархии классов на основе разработанных UML-диаграмм. Изучить синтаксис и возможности переопределенных (overriding) функций. Разработать программу согласно варианту.

Создание проекта в NetBeans

Создать новый проект в среде NetBeans можно комбинацией клавиш [Ctrl+Shift+N]. Укажем путь для сохранения проекта, зададим имя проекта JavaNew и после нажатия кнопки «Готово» получим следующий код:

Листинг 5

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package javanew;

/**
  * @author owner
 */
public class JavaNew {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
```

```
        //Здесь начало кода программы
    }
}
```

Мы видим шаблон программы, содержащий объявление класса main. Именно этот класс является входной точкой нашей программы.

В Листинге 5 сначала идет многострочный комментарий /* … */ Затем объявляется, что наш класс будет находиться в пакете ***javanew*** (по имени заданной директории).

После этого идет многострочный комментарий /** … */, предназначенный для автоматического создания документации по классу. В нем присутствует инструкция задания метаданных с помощью выражения ***@author*** – информация об авторе проекта для утилиты создания документации ***javadoc***. Метаданные – это некая информация, которая не относится к работе программы и не включается в нее при компиляции, но сопровождает программу и может быть использована другими программами для проверки прав на доступ к ней или для ее распространения, проверки совместимости с другими программами, указания параметров для запуска класса и т.п. В данном месте исходного кода имя **— *owner*** берется средой разработки из операционной системы по имени папки пользователя. Далее следует объявление класса ***public class JavaNew*** – имя класса соответствует названию проекта.

Все классы и объекты приложения вызываются и управляются из метода main, который объявлен далее и выглядит следующим образом:

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

Метод main является главным методом приложения и управляет работой запускаемой программы. Он автоматически вызывается при запуске приложения. Параметром args этого метода является массив строк, имеющий тип String[]. Это параметры командной строки, которые передаются в приложение при его запуске.

После окончания выполнения метода main приложение завершает свою работу. При объявлении любого метода в Java сначала указывается модификатор видимости, указывающий права доступа к методу, затем другие модификаторы, после чего следует

тип возвращаемого методом значения. Если модификатор видимости не указан, то это так называемый ***пакетный метод доступа – default.*** В данном случае элемент доступен классу, в котором объявлен, и другим классам в том же пакете, но не доступен классам, в том числе и наследникам, находящимся в других пакетах. Таким образом, данный уровень видимости является более строгим, чем ***protected***. (Это отличие от C++).

Внутрь метода main поместим тривиальное:

System.out.println("Hello, world!"); //Вы можете придумать что-то более оригинальное.

Класс ***System*** имеет поле ***out***. Это объект, предназначенный для поддержки вывода. У него есть метод println, предназначенный для вывода текста в режиме консоли.

Компиляция файлов проекта и запуск приложения

Для сборки проекта следует выбрать в меню среды разработки ***Run/ Build Main***.

Project (или клавиша **<F11>**, или на панели инструментов иконка с молотком). При этом происходит компиляция только тех файлов проекта, которые были изменены в процессе редактирования после последней сборки.

Пункт ***Run/ Clean and Build Main Project*** (или комбинация клавиш **<Shift> <F11>,** или на панели инструментов иконка с молотком и веником) удаляет все выходные файлы проекта (очищает папки build и dist), после чего по новой компилируются все классы проекта.

Пункт ***Build/ Generate Javadoc for (JavaNew)*** запускает создание документации по проекту. При этом из исходных кодов классов проекта выбирается информация, заключенная в документационные комментарии **/** … */**, и на ее основе создается гипертекстовый HTML-документ.

Пункт ***Run/ Compile (JavaNew)*** (или клавиша **<F9>**) компилирует выбранный файл проекта – в нашем случае файл ***JavaNew***, в котором хранятся исходные коды класса ***JavaNew.***

Для того чтобы запустить скомпилированное приложение из среды разработки, следует выбрать в меню среды разработки ***Run/ Run Main Project*** (или клавиша **<F6>**, или на панели инструментов иконка с зеленым треугольником).

Структура проекта NetBeans

В компонентной модели NetBeans пакеты приложения объединяются в единую конструкцию – модуль. Модули NetBeans являются базовой конструкцией не только для создания приложений, но и для написания библиотек. Они представляют собой оболочку над пакетами (а также могут включать другие модули).

В отличие от библиотек Java, скомпилированный модуль – это не набор большого количества файлов, а всего один файл, архив ***JAR (Java Archive, архив Java)***. В нашем случае он имеет то же имя, что и приложение, и расширение ***.jar***: это файл ***JavaNew.jar***. Модули NetBeans гораздо лучше подходят для распространения, поскольку не только обеспечивают целостность комплекта взаимосвязанных файлов, но и хранят их в заархивированном виде в одном файле, что намного ускоряет копирование и уменьшает объем занимаемого места на носителях.

Структура папок проекта NetBeans показана на рис. 11.

В папке ***build*** хранятся скомпилированные файлы классов, имеющие расширение .class.

В папке ***dist*** – файлы, предназначенные для распространения как результат компиляции (модуль JAR приложения или библиотеки, а также документация к нему).

В папке ***nbproject*** находится служебная информация по проекту.

В папке ***src*** – исходные коды классов. Кроме того, там же хранится информация об экранных формах (которые будут видны на экране в виде окон с кнопками, текстом и т.п.). Она содержится в XML-файлах, имеющих расширение .form.

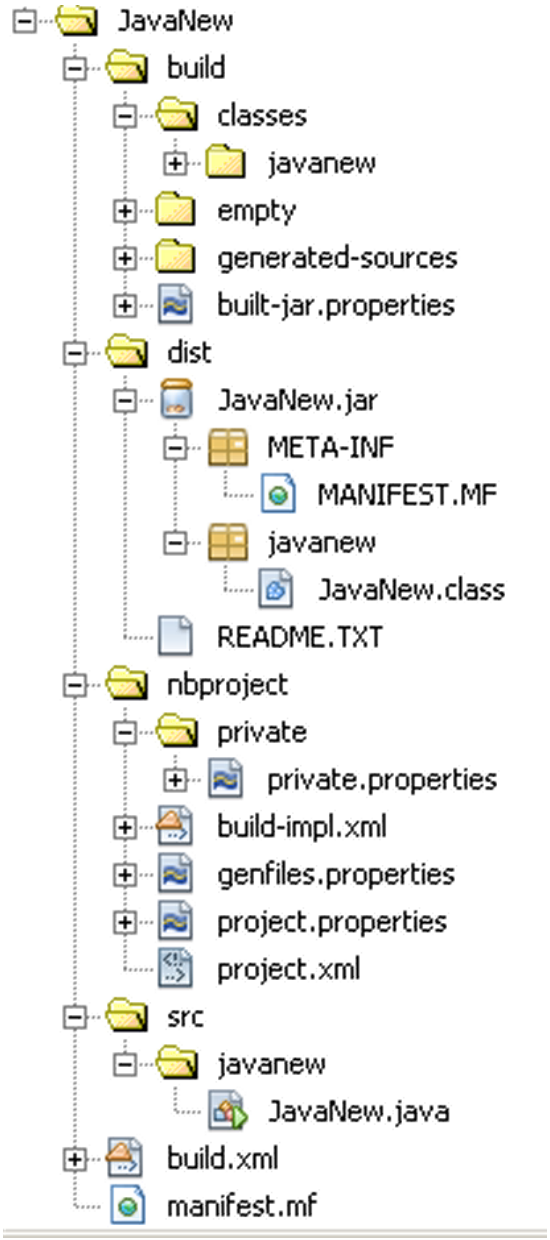


Рис.11. Структура проекта в NetBeans

В проекте еще может быть директория ***test***, хранящая сопроводительные тесты, предназначенные для проверки правильности работы классов проекта.

В файле README.TXT, находящемся в директории ***dist*** – там же, где архив JAR, предназначенный для распространения как файл приложения, содержится информация о том, как запустить архив, как его переносить и как с ним работать.

Перевод содержимого файла README.TXT:

Когда Вы компилируете проект приложения Java, которое имеет главный класс, среда разработки (IDE) автоматически копирует все файлы JAR-архивов, указанные в classpath ваших проектов, в папку dist/lib. Среда разработки также автоматически прибавляет путь к каждому из этих архивов в файл манифеста приложения (MANIFEST.MF).

Чтобы запустить проект в режиме командной строки, зайдите в папку dist и наберите в режиме командной строки следующий текст:

java -jar "JavaNew.jar"

Чтобы распространять этот проект, заархивируйте папку dist (включая папку lib) и распространяйте ZIP-архив.

Замечания:

Если два JAR-архива, указанные в classpath ваших проектов, имеют одинаковое имя, в папку lib будет скопирован только первый из них.

Если в classpath указана папка с классами или ресурсами, ни один из элементов classpath не будет скопирован в папку dist.

Если в библиотеке, указанной в classpath, также имеется элемент classpath, указанные в нѐм элементы должны быть указаны в пути classpath времени выполнения проектов.

Для того чтобы установить главный класс в стандартном проекте Java, щелкните правой кнопкой мыши в окне Projects и выберите Properties. Затем выберите Run и введите данные о названии класса в поле Main Class. Кроме того, Вы можете вручную ввести название класса в элементе Main-Class манифеста.

Класс Object

В Java существует иерархия классов и специальный класс — Object стоит во главе этой иерархии. Все остальные классы являются подклассами этого класса. Т.е. Object — суперкласс всех остальных классов. Это означает, что ссылочная переменная типа Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная типа Object может ссылаться также на любой массив. Здесь также

напомню, что в Java, в отличие от C, отсутствуют указатели, есть только ссылки.

Класс *Object* определяет методы, которые доступны в любом объекте. В этом классе есть много полезных методов, рассмотрим для начала два:

toString() и ***equals().***

public class Employee { }

эквивалентно:

public class Enployee extends Object { }

Приведенное объявление класса Employee демонстрирует объявление класса-наследника (необходимо использовать ключевое слово extends – расширяет).

Метод toString

Этот метод служит для представления объекта в виде строки. Это требуется, например, если мы хотим вывести объект на экран.

Важно знать, что метод ***toString()*** **есть у всех объектов** и все объекты используют этот метод при работе со строками. Но для того чтобы метод корректно выводил содержимое созданного вами объекта, его нужно переопределить. Что будет, если метод не переопределить? При обращении к функции toString() объекта выведется хеш-код данного объекта или его уникальный идентификатор. В терминах Java хеш-код — это целочисленный результат работы метода hashCode(), которому в качестве входного параметра передан объект. Этот метод реализован таким образом, что для одного и того же входного объекта хеш-код всегда будет одинаковым. Следует понимать, что множество возможных хеш-кодов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: «Множество объектов мощнее множества хеш-кодов». Из-за этого ограничения вполне возможна ситуация, что хеш-коды разных объектов могут совпасть. Отсюда следует, что:

- если хеш-коды разные, то и входные объекты гарантированно разные;

– если хеш-коды равны, то входные объекты не всегда равны.

Ситуация, когда у разных объектов одинаковые хеш-коды называется коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

Посмотрим, как это работает на примере. Пользователь нашей системы имеет идентификаторы: Имя, Фамилия и год рождения.

Класс, описывающий этого пользователя, имеет функции «геттеры» и «сеттеры», которые позволяют получить у объекта его идентификаторы и установить их (приставки get и set соответственно). В класс также добавлена переопределенная функция ***toString()***, которая позволяет вывести идентификаторы нашего пользователя.

Простейшая UML-диаграмма для приведенного класса USER:

Таблица 3

UML-диаграмма класса USER

User	
Здесь размещаются атрибуты или память класса	-name : String -surname : String -birthYear : int
Здесь размещаются методы, задающие поведение класса	+getName() : String +getSurname() : String +getBirthYear() : int +setName(n : String) +setSurname(s: String) +setBirthYear(y: int) +toString () : String

Значки «+» и «-» в UML-диаграмме означают модификаторы public и private соответственно. Есть еще значок «#» – protected, но здесь он нам пока не нужен.

Диаграмма класса User соответствует принципу инкапсуляции в ООП, поясните почему.

Согласно этой диаграмме разработан класс:

Листинг 6

```
class User {
   private String name;
   private String surname;
   private int birthYear;
//Конструктор
   User(String name, String surname, int birthYear)
   {
      this.name = name;
      this.surname = surname;
      this.birthYear = birthYear;
   }
//геттеры и сеттеры ввиду их простоты здесь не показаны.
 …
   @Override
   public String toString()
   {
      return this.name+" "+this.surname+", "+getAge()+" г. ";
   }
 }

public static void main(String[] args) {
User us1=new User("Иван", "Петров", "1996");
System.out.println(us1);
System.out.println(us1. toString());
System.out.println("Hach: "+us1.hashCode());
}
```

Запустите эту программу в двух вариантах: с преопределенной функцией toString() и без нее. Сделайте выводы о формате хеш-кода и о том, зачем нужна операция переопределения функции toString().

Инкапсуляция, наследование и реализация полиморфизма в Java

Проектирование архитектуры приложения

Итак, согласно принципу инкапсуляции, поля класса являются закрытыми (private), мы оставляем открытыми (public) только методы, необходимые для взаимодействия объектов и обращения к полям объекта. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, было наименьшее зацепление (low coupling) между классами – об этом же гласит первый принцип SOLID (п. 1.4.1). Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для создания других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную связность (high cohesion). Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил достижения сильной функциональной связности, введенное Карлом Либерхером (Karl J. Lieberherr), получило название закона Деметры. Закон гласит: *«В методе m() класса A следует использовать только методы класса A, методы классов, к которым принадлежат параметры метода m(), и методы классов, экземпляры которых создаются внутри метода m() »*.

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовы выполнить любое поручение, на которое рассчитана их конструкция.

А для описания ошибочного и непродуктивного подхода к проектированию программ существует термин «Антипаттерн», который описывает все известные образцы плохой архитектуры и программной модели.

Однако разработка хорошей архитектуры программного проекта – это целая наука, поэтому будем для начала придерживаться изложенных принципов SOLID, а также принципов декомпозиции задачи на модули и классы.

Считается, что хорошо спроектированные модули должны обладать следующими свойствами:

- функциональная целостность и завершенность — каждый модуль реализует одну функцию, но реализует хорошо и полностью; модуль самостоятельно (без помощи дополнительных средств) выполняет полный набор операций для реализации своей функции;
- один вход и один выход — на входе программный модуль получает определенный набор исходных данных, выполняет содержательную обработку и возвращает один набор результатных данных, т.е. реализуется стандартный принцип IPO — вход–процесс–выход;
- логическая независимость — результат работы программного модуля зависит только от исходных данных, но не зависит от работы других модулей;
- слабые информационные связи с другими модулями — обмен информацией между модулями должен быть по возможности минимизирован.

Одну и ту же задачу можно решить по-разному.

Пример 3.1. Пусть у нас есть класс, реализующий некоторую функциональность для обработки изображения:

ВАРИАНТ 1

Листинг 7

```
public static class ImageHelper
{
   public static void Save(Image image)
   {
      // сохранение изображение в файловой системе
   }

   public static int DeleteDuplicates()
   {
      // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных
   }
```

```
    public static Image SetImageAsAccountPicture(Image image, Account account)
    {
        // запрос к базе данных для сохранения ссылки на это изображение для пользователя
    }
     public static Image Resize(Image image, int height, int width)
    {
        // изменение размеров изображения
    }
    public static Image InvertColors(Image image)
    {
        // изменить цвета на изображении
    }
    public static byte[] Download(Url imageUrl)
    {
        // загрузка битового массива с изображением с помощью HTTP запроса
    }

    // и т.п.
}
```

Границы ответственности у этого класса слишком обширны. Он может изменять изображения, работать с базой данных, с файловой системой. Может скачивать изображения, взаимодействуя с сетью.

Каждая ответственность этого класса ведет к его потенциальному изменению. Получается, что этот класс будет очень часто менять свое поведение, что затруднит его тестирование и тестирование компонентов, которые его используют. Такой подход снизит работоспособность системы и повысит стоимость ее сопровождения.

ВАРИАНТ 2

Решением является выполнение декомпозиции класса по принципу единственности ответственности: один класс на одну ответственность.

Листинг 8

```
public static class ImageFileManager
{
   public static void Save(Image image)
   {
      // сохранение изображения в файловой системе
   }

public static class ImageHttpManager
{
   public static byte[] Download(Url imageUrl)
   {
 // загрузка битового массива с изображением с помощью HTTP запроса
   }
}

    public static int DeleteDuplicates()
   {
      // удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных
   }
}

public static class ImageRepository
{
   public static Image SetImageAsAccountPicture(Image image, Account account)
   {
  // запрос к базе данных для сохранения ссылки на это изображение для пользователя
   }
}

public static class Graphics
{
```

```
    public static Image Resize(Image image, int height, int width)
    {
        // изменение размеров изображения
    }

    public static Image InvertColors(Image image)
    {
        // изменить цвета на изображении
    }
}
```

Наследование является неотъемлемой частью Java. При использовании наследования мы говорим: этот новый класс похож на старый класс. В коде Java это пишется как **extends**, после которого указывается имя базового класса. Тем самым мы получаем доступ ко всем полям и методам базового класса. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов (это абстрактный базовый класс). Затем мы можем наследоваться от него и создать новый класс, который будет иметь свои уникальные характеристики. Главный наследуемый класс в Java называют ***суперклассом***. Наследующий класс называют ***подклассом.*** Получается, что подкласс – это специализированная версия суперкласса, которая наследует все члены суперкласса и добавляет собственные уникальные элементы.

Пример 3.2 демонстрирует принцип абстракции и полиморфизма в ООП.

Листинг 9

```
abstract class Pet{ // Создаем абстрактный базовый класс
//«домашнее животное»
  abstract void voice(); //и виртуальную функцию
}
/*****************************************/
class Dog extends Pet{
```

```
    @Override //Эта директива позволяет переопределить
//функцию родительского класса
     void voice(){
      System.out.println("Gav-gav!");
     }
    }
    /*********************************************/

    class Cat extends Pet{

     @Override
     void voice(){
      System.out.println("Miaou!");
     }
    }
    /*********************************************/

    class Cow extends Pet{

     @Override
     void voice(){
      System.out.println("Mu-u-u!");
     }
    }
    public class Chorus{
     public static void main(String[] args){
      Pet[] singer = new Pet[3]; //объявляем массив животных
      singer[0] = new Dog();
      singer[1] = new Cat();
      singer[2] = new Cow();
      for (Pet p: singer)
        p.voice(); //Обращаясь к элементу массива, воспроизводим
    //«сообщение» животного
     }
    }
```

Проверку соответствия сигнатуры (списка параметров) переопределяемого метода можно возложить на компилятор, записав

перед методом подкласса аннотацию @Override. Тогда компилятор пошлет на консоль сообщение об ошибке, если сигнатура помеченного метода не будет соответствовать сигнатуре ни одного метода суперкласса с тем же именем.

Все дело здесь в определении массива singer[]. Хотя массив ссылок singer[] имеет тип Pet, каждый его элемент ссылается на объект своего типа: Dog, Cat, Cow. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется динамический полиморфизм.

Знатокам C++: в языке Java все нестатические public методы являются виртуальными функциями, а в C++ виртуальность метода нужно декларировать явно.

Для переопределения метода в языках Java и C++ не требуется использования каких-либо дополнительных ключевых слов: достаточно в классе-наследнике реализовать метод с той же самой сигнатурой. В Java желательно использовать директиву @Override, чтобы компилятор мог вовремя распознать ошибку. Подробнее об этом будет сказано в лекциях к этому курсу.

Наверняка вы заметили в описании класса ***Pet*** слово ***abstract***. Класс ***Pet*** и метод ***voice()*** являются абстрактными (подобно тому, как в C++ имеются абстрактные классы, не имеющие права на реализацию и не имеющие ключевого слова ***abstract***).

Окончательные члены и классы. Пометив метод модификатором final, можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции sin(), cos() и прочие в классе Math. Мы уверены, что метод Math.cos(x) вычисляет именно косинус числа x. Разумеется, такой метод не может быть абстрактным.

Для полной безопасности поля, обрабатываемые окончательными методами, следует сделать закрытыми (private). Если пометить модификатором final параметр метода, то его нельзя будет изменить внутри метода. Если же пометить модификатором final весь класс, то его вообще нельзя будет расширить. Так определен, например, класс Math:

```
public final class Math{ . . . }
```

Метод equals

Java работает не с объектами, а с ссылками на объекты. Это объясняет то, что операции сравнения ссылок на объекты не имеют смысла, так как при этом сравниваются адреса. Для сравнения объектов на эквивалентность по значению необходимо использовать специальные методы. Таким образом, метод equals() – еще один метод наряду с toString, который почти в любом проекте требует переопределения, поскольку почти в любом проекте возникает необходимость сравнивать значения переменных между собой.

Оператор == определяет, являются ли две ссылки идентичными друг другу (т.е. относятся к одному объекту).

Метод equals определяет, являются ли объекты равными, но не обязательно идентичными. Реализация метода equals класса Object использует оператор ==.

Классы пользователей могут переопределить метод equals, чтобы реализовать предметно-ориентированный тест на равенство.

Примечание: необходимо переопределить метод hashCode, если вы переопределяете метод equals.

Пример 3.3. Сравнение дат.

Листинг 10

```
public class MyDate {
   private int day, month,year;
   public MyDate(int d, int m, int y)
   {day=d; month=m; year=y;
   }
   @Override
   public boolean equals(Object o)
   {
      boolean result =false;
//сравниваем только объекты типа MyDate.
      if(o!=null && o instanceof MyDate)
      { MyDate d=(MyDate)o; //Явное приведение типов
        if ((day==d.day)&&(month==d.month)&&(year==d.year))
```

```
            result=true;
        }
        return result;
    }
}
public class TestEquals {
    public static void main(String[] args) {
        MyDate date1=new MyDate(7,11,2014);
        MyDate date2=new MyDate(7,11,2014);
        if (date1==date2)
            System.out.println("date1 is identical date2");
        else
            System.out.println("date1 is not identical date2");

        if (date1.equals(date2))
            System.out.println("date1 is equals date2");
        else
            System.out.println("date1 is not equals date2");

         System.out.println("set date2=date1");
         date2=date1;
         if (date1==date2)
            System.out.println("date1 is identical date2");
        else
            System.out.println("date1 is not identical date2");
    }
}
```

Результат:

```
date1 is not identical date2 // ссылки не равны
date1 is equals date2          //объекты равны
set date2=date1
date1 is identical date2     //теперь и ссылки равны
```

При необходимости принимать данные из консоли вам понадобится следующий пример:

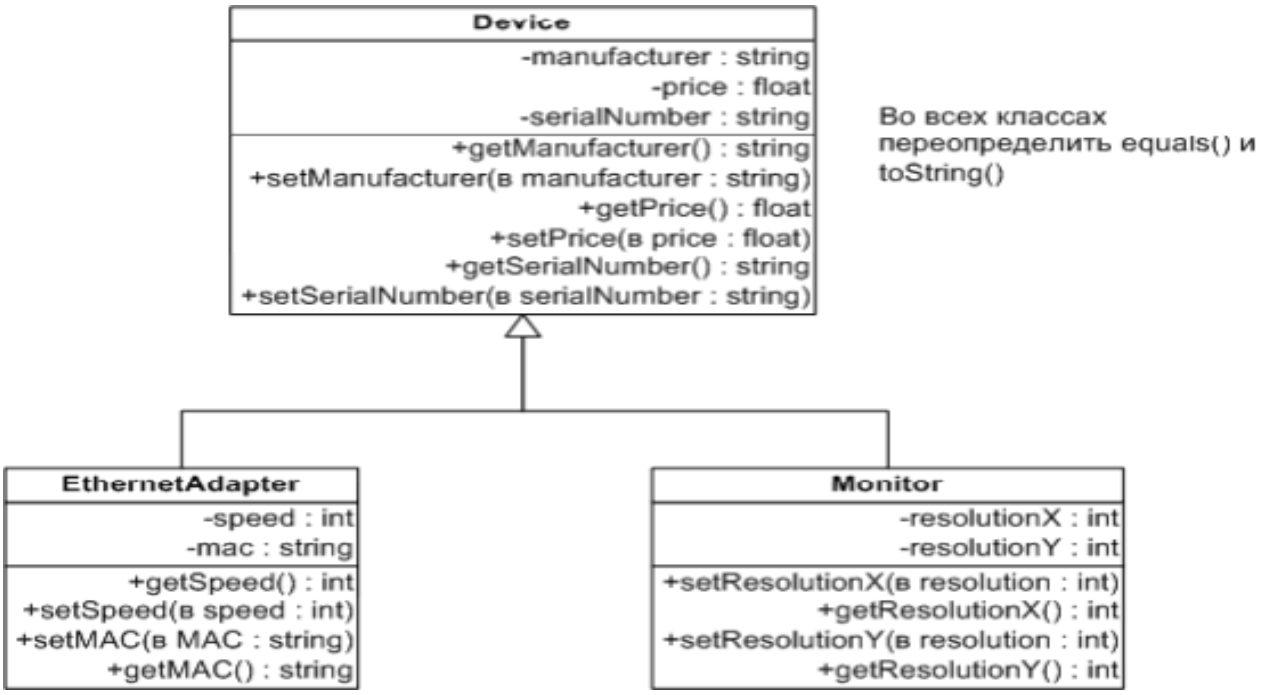
Листинг 11

```
//Пакет обеспечивает чтение данных из входного потока
import java.util.Scanner;
public class RunScanner {
      public static void main(String[ ] args) {
        System.out.println("Введите Ваше имя и нажмите <Enter>:");
        Scanner scan = new Scanner(System.in);
        String name = scan.next();
        System.out.println("Привет, " + name);
scan.close();
  }
}
```

Задания к лабораторной работе №2

1. Проверьте пример 3.3. без переопределенного метода equals(), сделайте выводы.
2. Разработайте заданную иерархию классов согласно приведенным ниже вариантам (вариант выдает преподаватель), обязательным элементом является разработка UML-диаграмм классов. Образец – вариант 1 задания. Классы должны быть разложены по пакетам, все классы должны иметь необходимые методы get и set.
3. При разработке конструкторов используйте обращение к суперклассу (super).
4. В классе main Вашего проекта продемонстрируйте создание массива разнотипных объектов (подобно примеру 3.2.), установку и вывод характеристик каждого объекта. В массив вносить только уникальные объекты (для проверки уникальности объекта используйте метод equals()).
5. Для вывода содержимого объектов выполните перегрузку метода toString(), начиная с класса-родителя.
6. Методы get и set должны быть определены в каждом неабстрактном классе.

Вариант 1. Разработайте иерархию классов в соответствии с UML-диаграммой:



В базовый класс добавьте метод «заменить», «распознать», «установить драйвер», «удалить драйвер».

Вариант 2. Разработайте UML–диаграмму и программу:

Напишите иерархию классов, описывающих имущество налогоплательщиков. Она должна состоять из абстрактного базового класса Property (Собственность) и производных от него классов Apartment, Car и Country House. Базовый класс должен иметь поле ***worth*** (стоимость), конструктор с одним параметром, заполняющий это поле, и чисто виртуальный метод расчета налога, переопределенный в каждом из производных классов. Налог на квартиру вычисляется как 1/1000·Square ее стоимости, на машину — 1/10·Volume, на дачу — 1/5000·Square_t+1/100·Square_h. В базовый класс добавить методы «продать», «переоформить». Также каждый производный класс должен иметь конструктор с одним параметром, передающий свой параметр конструктору базового класса. В класс Apartment добавьте поля Address (адрес), Square (площадь) в класс Car Volume (объем двигателя) и Year (год выпуска), в класс Country House — Square_h (площадь дома) и Square_t (площадь владений). В методе **main** создайте массив из 10 указателей на Property и заполните его объектами производных

классов (Appartment, Car и Country House). Выведите на экран величину налога для всех объектов. ***В массив вносите только уникальные объекты.***

Вариант 3. Есть базовый класс Guest (пользователь с ограниченными правами (права доступа можно представить в виде массива **bool [] rights**). От него унаследовать классы: Admin (имеет добавочный пароль и максимум прав), метод по установке прав – setRights( ), и User (должен указать фамилию и должность, не может выполнять удаление). Переопределите toString() и equals(). В методе main создайте10 элементов разных типов (здесь должны быть некоторые одинаковые объекты) и заполните массив типа Guest этими объектами производных классов. При записи в массив проверяйте на совпадение логинов (метод equals) – В массив вносите только уникальные объекты, при этом выводите сообщение об одинаковых логинах.

Guest
-login : String **-password : String** **-read:bool;** **-write:bool;** **-edit:bool;** **-delete:bool;**
+getLogin() : String **+getPassw() : String** **+getRight() : bool[]** **+setLogin(n : String)** **+setPassw(s: String)** **+toString () : String** **+equals():bool** **+login(): bool** **+outLog(): bool** **+load_File(): bool**

Вариант 4. Разработайте UML–диаграмму и программу:

Напишите иерархию классов «научный сотрудник», описывающих ставку и принцип вычисления надбавок. Она должна состоять из абстрактного базового класса **Scientist** и производных от него классов «старший научный сотрудник», «младший научный сотрудник» и «инженер». Базовый класс должен иметь поле ***salary*** (ставка) и ***seniority*** (стаж), конструктор с двумя параметрами, заполняющий эти поля, и чисто виртуальный метод расчета надбавок, переопределенный в каждом из производных классов. Надбавки вычисляются так: ***salary*(1+seniority*x)***, где x = 0,03, 0,02 и 0,01 для СтаршегоНС, МладшегоНС и ИТР соответственно. В базовый класс также необходимо внести методы «принять», «уволить», «переместить на должность», «начислить зарплату».

Также каждый производный класс должен иметь конструктор с двумя параметрами, передающий свои параметры конструктору базового класса. Создайте несколько объектов производного класса, необходимо наличие одинаковых объектов. В методе **main** создайте массив из 9 указателей типа **Scientist** и заполните его объектами производных классов. Выведите на экран величину надбавок для всех объектов и зарплат (ставка + надбавка). ***В массив вносить только уникальные объекты.***

Вариант 5. Разработайте иерархию геометрических фигур (не менее 3 дочерних классов) с базовым классом ***Shape***. Базовый класс содержит одну координату для вывода фигур на экран и виртуальные методы square() и perimeter(), которые подсчитают площадь и периметр фигуры соответственно, move(), fill() – для сдвига и заливки фигуры цветом. Дочерние классы (линия, прямоугольник, окружность и треугольник) должны содержать параметры фигуры, по которым их можно нарисовать и рассчитать площадь и периметр. В методе **main** создайте массив из 15 указателей на ***Shape*** и заполните его объектами производных классов, здесь допускаются одинаковые объекты. Выведите на экран величины площади, периметра и цвета заливки для всех объектов. ***В массив вносите программно только уникальные объекты.***

Вариант 6. См. вариант 1, модифицированный для классов: Машина (***Car***) с полями «марка», «год выпуска», «объем двигателя», «тип коробки передач», «базовое значение страхового сбора» – ***base***, и функцией расчета страхового сбора для дочерних классов «Иномарка» – InBrand – base + 0,05 · year · volume · комплектация, «Отечественный бренд» – HomeBrand: base + 0,03 · year · volume. В «Иномарку» добавляется поле «тип комплектации» с возможными условными значениями 1, 2, 3. В HomeBrand внесите поле «color» – разрешенный процент цветовых добавок небазового цвета. В базовый класс добавьте функции stat(), stop(), in_conditioner(), out_conditioner(). Каждый производный класс должен иметь конструктор с несколькими параметрами, передающий свои параметры конструктору базового класса. В методе **main** создайте набор объектов дочерних классов (не менее 10), здесь до-

пускаются одинаковые объекты. Заполните массив типа ***Car*** объектами производных классов, выведите на экран величину страхового сбора для всех объектов. В массив программа должна вносить только уникальные объекты.

Вариант 7. Разработайте UML–диаграмму и программу.

Напишите иерархию классов, в основании которой лежит абстрактный класс «Animal», содержащий поля: «Name», «Type» (порода), «Sound» и метод «Say» (издать звук), а также методы run() – бежать, do_command() – выполнить команду, eat() – есть, sleep() – спать. Производные от него классы «Dog», «Cat» и «Horse». Базовый класс должен иметь конструктор с тремя параметрами, заполняющий эти поля, и чисто виртуальный метод «Say», переопределенный в каждом из производных классов. Кошке добавьте параметр «Цвет глаз», собаке и лошади – «Вес», лошади – «Скорость бега».

Также каждый производный класс должен иметь конструктор с параметрами, передающий свои параметры конструктору базового класса. Создайте несколько объектов производного класса, необходимо наличие одинаковых объектов. В методе **main** создайте массив из 9 указателей типа **Animal** и заполните его объектами производных классов. Выведите на экран характеристики всех животных и издаваемые ими звуки. ***В массив вносите программно только уникальные объекты.***

Вариант 8. Счет (сумма: положить(), снять(), проверить баланс(), закрыть()) ← счет_зарплатный (выполнить автоплатежи()), счет_кредитный (процент, срок погашения, проверить_срок_погашения()), счет_депозитный (процент, срок вклада, посчитать сумму процентов за истекший период()). Абстрактный базовый класс – Счет, его наследники: счет_зарплатный, счет_кредитный, счет_депозитный. Разработайте UML-диаграмму классов и программу с теми же задачами, что и в предыдущих вариантах (обязательна работа с массивом и внесение в него только уникальных объектов).

Вариант 9. Печатное издание (тираж, язык, число страниц). Газета (электронная/печатная), журнал (число выпусков в год),

книга (автор). Виртуальная функция: вычислить стоимость(), для газеты (параметры: базовая_сумма, количество слов, тираж) {(bs + 5·str)/tir}, для журнала (параметры: базовая_сумма, количество статей, тираж) {(bs + 50·stat)/tir}, для книги (параметры: базовая_сумма, количество_страниц) {(bs + 500·str}. Разработайте UML-диаграмму классов и программу с теми же задачами, что и в предыдущих вариантах (обязательна работа с массивом и внесение в него только уникальных объектов).

Требования к отчету к ЛР №2

1. Постановка задачи.
2. UML-диаграмма (даже если она есть в задании) и текст программы.
3. Скриншоты исполнения программы.
4. Выполните декомпозицию модулей: классы должны находиться в разных файлах.
5. В классе main Вашего проекта продемонстрируйте создание набора разнотипных объектов (подобно примеру 3.2), установку и вывод характеристик каждого объекта, на этом этапе должны быть одинаковые (равные) объекты. Создайте массив базового типа, в массив программно вносятся только уникальные объекты (используйте метод *equals()*). Можно использовать ArrayList.

При невыполнении требований к отчету оценка снижается.

ЛАБОРАТОРНАЯ РАБОТА № 3

РАЗРАБОТКА И ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

Цель. Ознакомиться с понятием интерфейса, правилами разработки, наследования и способами прикладного использования интерфейсов в Java.

Понятие интерфейса в java

Интерфейсы подобны полностью абстрактным классам, хотя и не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существует два вида интерфейсов: интерфейсы, определяющие функциональность для классов посредством описания методов, но не их реализации; и интерфейсы, реализация которых автоматически придает классу определенные свойства. К последним относятся, например, интерфейсы Cloneable и Serializable, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно.

Первый вид интерфейсов подобен классу, в котором все поля — константы (т.е. статические — ***static*** и неизменяемые — ***final***), а все методы абстрактные.

При описании интерфейса вместо ключевого слова ***class*** используется ключевое слово ***interface***, после которого указывается имя интерфейса, а затем, в фигурных скобках список полей-констант и методов. Никаких модификаторов перед объявлением полей и методов ставить не надо: все поля автоматически становятся ***public static final***, а методы — ***public abstract***. Методы не могут иметь реализации в самом интерфейсе, т.е. после закрывающей круглой скобки сразу ставится точка с запятой.

Как Вы знаете, никаких интерфейсов в языке C++ и многих других языках нет. В Java интерфейсы были предложены для решения проблем множественного наследования, затем они стали привычным и удобным инструментом и распространились на языки мобильной разработки Kotlin и др. Множественное наследование между интерфейсами допустимо. Классы, в свою очередь, интерфейсы только реализуют. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов.

Пример 3.1. Разработаем интерфейс для работы с классом Счет (Account).

Листинг 12

```
public interface AccountAction {
// по умолчанию все методы public abstract
boolean openAccount(); // объявление сигнатуры метода
boolean closeAccount();
void blocking();
void unBlocking();
double depositInCash(int accountNumber, int amount);
boolean withdraw(int accountNumber, int amount);
boolean convert(double amount);
boolean transfer(double amount);
}
```

В интерфейсе объявлены, но не реализованы действия, которые может производить клиент со своим счетом. Реализация нецелесообразна в исходном интерфейсе из-за возможности различного способа выполнения действия в конкретной ситуации. Например, счет может блокироваться автоматически либо по требованию клиента, либо администратором банковской системы. В каждом из трех указанных случаев реализация метода **blocking()** будет уникальной и никакого базового решения предложить невозможно. С другой стороны, наличие в интерфейсе метода заставляет класс, его имплементирующий, представить реализацию методу. Программист получает повод задуматься о способе реализации функциональности, так как наличие метода в интерфейсе говорит о необходимости той или иной функциональности всем классам, реализующим данный интерфейс. Интерфейс можно сделать ориентированным на выполнение близких по смыслу задач, например, отделить действия по созданию, закрытию и блокировке счета от действий по снятию и пополнению средств. Такое разделение разумно в ситуации, когда клиенту посредством интернет-системы не предоставляется возможность открытия, закрытия и блокировки счета. Тогда вместо одного общего интерфейса можно записать два специализированных: один – для администратора, второй — для клиента.

Листинг 13. Интерфейс для общего управления счетом

```
public interface AccountBaseAction {
boolean openAccount();
boolean closeAccount();
void blocking();
void unBlocking();
}
```

Листинг 14. Интерфейс для клиентского управления счетом

```
public interface AccountOperationManager {
double depositInCash(int accountNumber, int amount);
boolean withdraw(int accountNumber, int amount);
boolean convert(double amount);
boolean transfer(int accountNumber, double amount);
```

Пример 3.2. Необходимо разработать иерархию классов и интерфейсов согласно следующей UML-диаграмме (рис.12).

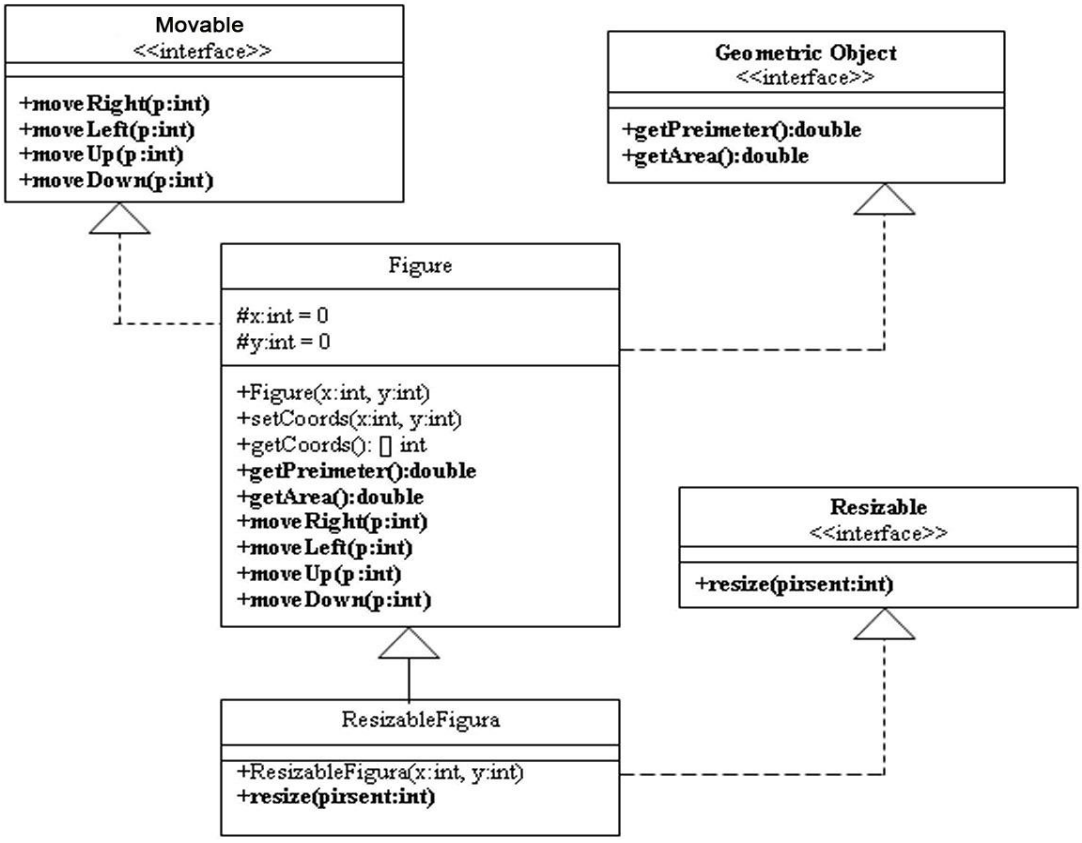


Рис.12. UML-диаграмма для задачи с интерфейсами для фигур

Опишем сначала интерфейс для объекта, который необходимо передвигать в разных направлениях.

```
public interface Movable {
   public void moveRight(int p);
   public void moveLeft(int p);
   public void moveUp(int p);
   public void moveDown(int p);
}
```

Теперь создадим класс **Figure**, который будет использовать интерфейс **Movable** и интерфейс GeometricObject**:**

```
public class Figure implements Moveable, GeometricObject {
 private int x,y;// координаты размещения объекта на плоскости
  // constructor
   public Fifure(int nx, int ny) {
      setCoords(nx,ny);
   }
// Implement methods defined in the interface GeometricObject
    @Override
    public void moveRight(int p) { ..... }
…
}
```

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ №3

Ориентируясь на диаграмму из примера 3.2, выполните следующие действия:

1. В вашем варианте задачи из ЛР №2 замените базовый класс интерфейсом. Если позволяет задача, разбейте его на два интерфейса, в качестве примера используйте листинги 12, 13, 14.
2. Разработайте UML-диаграмму и программу для созданной иерархии.

Требования к отчету к ЛР №3

Приведите UML-диаграмму и программу в отчете, добавьте скриншоты исполнения программы. Классы должны находиться в разных файлах.

ЛАБОРАТОРНАЯ РАБОТА № 4 (ч.1)

БИБЛИОТЕКИ AWT И SWING ДЛЯ ПОСТРОЕНИЯ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ. ОБРАБОТКА СОБЫТИЙ

Цель. Освоить методы разработки с графическим интерфейсом Java, ознакомиться с различными компоновщиками. Научиться обрабатывать события пользовательского интерфейса.

Данная лабораторная работа состоит из двух частей: в первой части изучаются менеджеры компоновки графического интерфейса Java, во второй – схема и методы обработки событий. По данной работе оформляется единый отчет.

ОБЩИЕ СВЕДЕНИЯ О ГРАФИЧЕСКИХ ИНТЕРФЕЙСАХ

Графический пользовательский интерфейс (GUI) – основной способ взаимодействия конечных пользователей с java-приложением. Для разработки прикладного программного обеспечения на языке Java, а точнее графического интерфейса приложений, обычно используются пакеты AWT и Swing, а также новый фреймворк JavaFX.

AWT – Abstract Window Toolkit (для доступа загружается пакет java.awt) – содержит набор классов, позволяющих выполнять графические операции и создавать оконные элементы управления, подобно тому, как это делается в VBA и Delphi.

Swing (для доступа загружается пакет javax.swing) является «надстройкой» над AWT и содержит новые классы, в основном аналогичные AWT. К именам классов добавляется J (JButton, JLabel и др.).

JavaFX представляет инструментарий для создания кросс-платформенных графических приложений на платформе Java и позволяет создавать приложения с насыщенной графикой благ. ря использованию аппаратного ускорения графики и возможностей GPU.

JavaFX дает больше возможностей по сравнению со Swing, предоставляя большой набор элементов управления, возможности по работе с мультимедиа, двухмерной и трехмерной графикой, де-

кларативный способ описания интерфейса с помощью языка разметки FXML, возможность стилизации интерфейса с помощью CSS, интеграцию со Swing и многое другое. Последняя версия фреймворка – JavaFX 12 – вышла в марте 2019 г.

Однако в нашем курсе акцент ставится на разработке мобильных приложений. В этом изучение JavaFX, при всех его достоинствах, нам не поможет, а только отнимет драгоценное время; изучение ***Swing*** поможет понять принципы разработки визуального графического интерфейса для Android.

Возвращаясь к ***Swing***, отметим, что на данный момент основные классы для построения визуальных интерфейсов содержатся в пакете javax.swing. Из пакета ***AWT*** используются классы для обработки сообщений. Никаких препятствий для совместного использования графических компонентов этих двух библиотек нет.

Для программирования графического интерфейса пользователя сначала нужно научиться создавать окна. Для этого в библиотеке ***AWT*** предусмотрен класс ***Frame***, в библиотеке ***Swing*** – класс ***JFrame***.

Напишем программу на Java, создающую пустое окно. Когда это окно будет закрыто пользователем, программа завершится.

Листинг 15

```
package javaapplication1;
import javax.swing.*;
public class main {
public static void main(String[] args) {
JFrame f = new JFrame ();
f.setSize(300, 200);
//закрывает окно
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setVisible(true);
 }
}
```

Созданное таким образом окно – пустое, но оно умеет делать все то, что требуется от окон в оконной системе: его

можно перемещать, изменять его размер, раскрывать на весь экран или минимизировать на панель.

Следующее, чему мы научимся, — выводить в созданное окно некоторый текст. Для этого существует класс ***JLabel*** (метка). Это пассивный элемент, позволяющий показывать в окнах определенный текст. Основное его достоинство состоит в том, что он вписывается в общую структуру интерфейса, т.е. занимает определенное место и может автоматически перерисовываться, например, если он располагался в закрытой другими окнами части окна и затем стал видимым.

Листинг 16

```
public static void main(String[] args) {
JFrame f = new JFrame();
//создать новый элемент "метка"
JLabel lab = new JLabel("Hello, world!");
f. setSize(300, 200);
f.setLocation(500, 200); //задает расположение окна

f.add(lab); //присоединить "метку" к окну
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
}
```

Что происходит, если менять размер созданного программой окна?

Задача 4.1 для самостоятельного решения

1. Измените текст метки на «Я метка 1» или любой другой.

2. Напишите программу, создающую три-четыре метки (кнопки) в окне. Как они будут располагаться относительно окна и друг относительно друга? А если менять размер окна?

Вот так можно добавить кнопку – для этого существует класс JButton.

Листинг 17

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

```
JFrame f = new JFrame ();
JPanel p = new JPanel();
JButton b = new JButton("Press me!");
p.add(b); //внести кнопку в панель
f.setSize(300, 200);
f.add(p); //добавить панель в окно
  //f.pack(); как вариант - "упаковывает" окно до оптимального
  // размера всех расположенных в нем компонентов.
     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     f.setVisible(true);
}
```

Задача 4.2 для самостоятельного решения

Напишите программу, создающую 8–9 меток (кнопки) в окне (только теперь добавляйте элементы в панель *JPanel*). Как они будут располагаться относительно окна и друг относительно друга? А если менять размер окна от минимального до максимального? А если вытянуть его в узкую вертикальную полоску?

Здесь кнопка добавляется в окно не непосредственно, а через специальный объект класса JPanel — так называемую панель. Панель нужна для того, чтобы управлять размещением элементов в окне. Менеджер размещения *FlowLayout* включается в работу по умолчанию.

Способы размещения элементов

Размещение элементов с помощью менеджеров (компоновщиков)

Можно, конечно, размещать компоненты «вручную», задавая их размеры и положение в контейнере абсолютными координатами в координатной системе контейнера и используя метод ***setBounds()***.

Такой способ размещает компоненты с точностью до пикселя, но не позволяет перемещать их менеджеру компонентов автомати-

чески: при изменении размеров окна с помощью мыши компоненты останутся на своих местах привязанными к левому верхнему углу контейнера. Кроме того, нет гарантии, что все мониторы отобразят компоненты так, как вы задумали.

Чтобы учесть изменение размеров окна, надо задать размеры и положение компонента относительно размеров контейнера, но тогда при всяком изменении размеров окна расположение компонента придется задавать заново.

Чтобы избавить программиста от этой кропотливой работы, в библиотеку AWT внесены два интерфейса: LayoutManager и порожденный от него интерфейс LayoutManager2, а также несколько реализаций этих интерфейсов: классы BorlerLayout, CardLayout, FlowLayout, GridLayout, GridBagLayou и др.

Эти классы названы менеджерами размещения компонентов (layout manager). Библиотека Swing добавляет к указанным классам свои менеджеры размещения, используемые контейнерами Swing.

Каждый программист может создать собственные менеджеры размещения, реализовав интерфейсы LayoutManager или LayoutManager2.

Следует отметить, что в Android тоже есть подобные компоновщики, принцип работы которых очень близок рассматриваемым компоновщикам Java.

Посмотрим, как размещают компоненты эти классы.

FlowLayout

Наиболее просто поступает менеджер размещения FlowLayout. Он укладывает в контейнер один компонент за другим слева направо как кирпичи, переходя от верхних рядов к нижним. При изменении размера контейнера «кирпичи» перестраиваются (рис.13, листинг 18).

Компоненты поступают в том порядке, в каком они заданы в методах add().

В каждом ряду компоненты могут прижиматься к левому краю, если в конструкторе аргумент align равен FlowLayout.LEFT, к правому краю, если этот аргумент FlowLayout.RIGHT, или собираться в середине ряда, если FlowLayout.CENTER.

Между компонентами можно оставить промежутки (gap) по горизонтали hgap и вертикали vgap. Это задается в конструкторе:

FlowLayout(int align, int hgap, int vgap);

Второй конструктор задает промежутки размером 5 пикселов:

FlowLayout(int align);

Третий конструктор определяет выравнивание по центру и промежутки 5 пикселов:

FlowLayout();

После формирования объекта эти параметры можно изменить методами: setHgap(int hgap); setVgap(int vgap); setAlignment(int align).

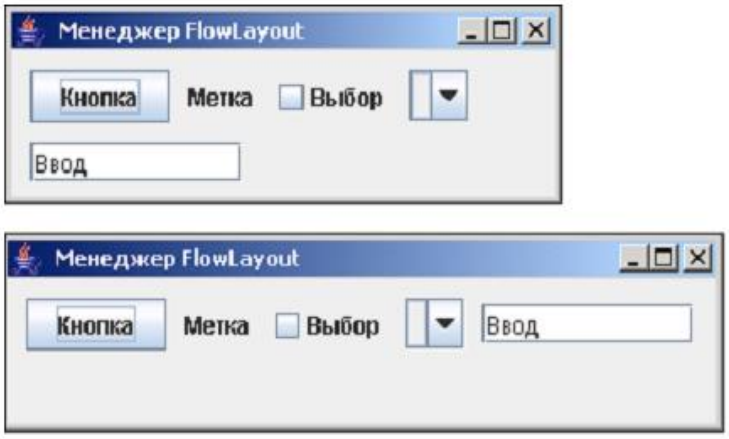


Рис.13. Компоновщик FlowLayout при сжатом и расширенном окне

Листинг 18

```
// Расположение элементов последовательно в ряд
package simpleframe;
import java.awt.FlowLayout;
import javax.swing.*;

class FlowTest extends JFrame{
   FlowTest(String s){
   super(s);
   setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
   add(new JButton("Кнопка"));
   add(new JLabel("Метка"));
   add(new JCheckBox("Выбор"));
```

```
    add(new JComboBox());
    add(new JTextField("Ввод", 10));
    setSize(300, 100);
    setVisible(true);
  }
}
public class SimpleFrame {
public static void main(String[] args) {
//В качестве параметра передаем компоновщик
JFrame f= new FlowTest("  Менеджер FlowLayout");
      f.setLocation(400,200);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

BorderLayout

Менеджер размещения BorderLayout делит контейнер на пять неравных областей, полностью заполняя каждую область одним компонентом. Области получили географические названия — NORTH, SOUTH, WEST, EAST и CENTER (рис.13, Листинг 19).

Метод add() в случае применения BorderLayout имеет два аргумента: ссылку на компонент (comp) и область (region), в которую помещается компонент, — одну из перечисленных ранее «географических» констант: add(Component comp, String region).

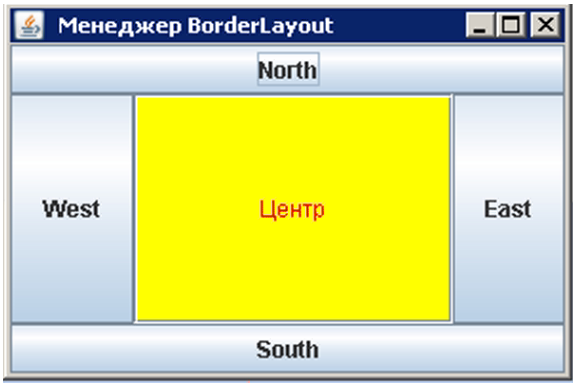


Рис.14. Компоновщик BorderLayout

Обычный метод add(Component comp) с одним аргументом помещает компонент в область CENTER.

Ссылку на компонент, помещенный в определенную область, можно получить методом getLayoutComponent(Object region).

Листинг 19

```
class BorderTest extends JFrame{
  BorderTest(String s){
    super(s);
//добавляя элемент к окну, необходимо сразу указать
//регион его размещения
    add(new JButton("North"), BorderLayout.NORTH);
    add(new JButton("South"), BorderLayout.SOUTH);
    add(new JButton("West"), BorderLayout.WEST);
    add(new JButton("East"), BorderLayout.EAST);
    JTextField tf=new JTextField("Центр");
//задаем цвет фона – желтый и цвет текста – красный
    tf.setBackground(Color.yellow);
    tf.setForeground(Color.RED);
    tf.setHorizontalAlignment(JTextField.CENTER);
    add(tf);
    setSize(300, 200);
    setVisible(true);
    setVisible(true);
  }

  public static void main(String[] args){
    JFrame f= new BorderTest("  Менеджер BorderLayout");
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

Менеджер размещения BorderLayout может показаться неудобным: он размещает не больше пяти компонентов, последние растекаются по всей области, а сами области имеют странный вид. Но дело в том, что в каждую область можно поместить не компонент, а панель, и размещать компоненты на ней, как сделано в приведенном ниже примере (рис.15, листинг 20). На панелях Panel и JPanel менеджер размещения по умолчанию — FlowLayout.

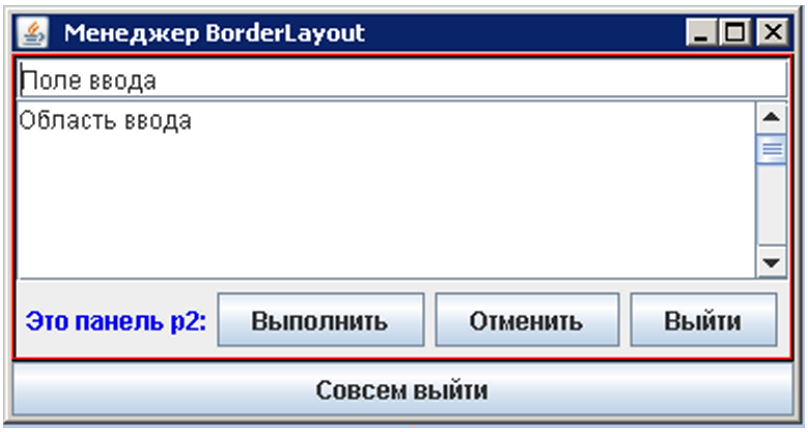


Рис.15. Компоновщик BorderLayout с панелями

Листинг 20

```
// Расположение элементов по сторонам света (BorderLayout)
 import java.awt.*;
import javax.swing.*;

class BorderPanelTest extends JFrame{

BorderPanelTest(String s){
   super(s);
// Создаем панель p2 с тремя кнопками и меткой
    JPanel p2 = new JPanel();
    JLabel l1=new JLabel(("Это панель p2:"));
  l1.setForeground(Color.BLUE);
  p2.add(l1);
  p2.add(new JButton("Выполнить"));
     p2.add(new JButton("Отменить"));
  p2.add(new JButton("Выйти"));
  p2.setBounds(5, 5, 5, 5);

JPanel p1 = new JPanel();
p1.setLayout(new BorderLayout());
       // Помещаем панель p2 с кнопками на "юге" панели p1
 p1.add(p2, BorderLayout.SOUTH);
// Поле ввода помещаем на "севере"
```

```
p1.add(new JTextField("Поле ввода", 20), BorderLay-
out.NORTH);
 // Область ввода помещается на панель с прокруткой
 JScrollPane sp = new JScrollPane(new JTextArea("Область
ввода",20,5));
// Панель прокрутки помещается в центр панели p1
p1.add(sp, BorderLayout.CENTER);
p1.setBorder(BorderFactory.createEtchedBorder(100, Color.red,
Color.black));
p1.setBounds(5, 5, 5, 5);
 // Панель p1 помещаем в "центре" контейнера
add(p1, BorderLayout.CENTER);
add(new JButton("Совсем выйти"), BorderLayout.SOUTH);
setSize(400, 200);
setVisible(true);
  }
}
public class SimpleFrame {

    public static void main(String[] args) {

   JFrame f= new BorderPanelTest("  Менеджер BorderLayout");
      f.setLocation(400,200);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

GridLayout

Менеджер размещения GridLayout расставляет компоненты в таблицу с заданным в конструкторе числом строк rows и столбцов columns:

GridLayout(int rows, int columns);

Все компоненты получают одинаковый размер. Промежутков между компонентами нет. Второй конструктор позволяет задать промежутки между компонентами в пикселях по горизонтали hgap и вертикали vgap:

GridLayout(int rows, int columns, int hgap, int vgap);

Конструктор по умолчанию GridLayout() задает таблицу размером 0x0 без промежутков между компонентами. Компоненты будут располагаться в одной строке.

Компоненты размещаются менеджером GridLayout слева направо по строкам созданной таблицы в том порядке, в котором они заданы в методах add(). Нулевое количество строк или столбцов означает, что менеджер сам создаст нужное их число (листинг 21, рис.16).

Рис.16. Компоновщик GridLayout с панелями

Листинг 21

```
// Табличное расположение элементов
import java.awt.*;
import javax.swing.*;
import java.util.*;
  class GridTest extends JFrame{
  GridTest(String s){
  super(s);
  setLayout(new GridLayout(4, 4, 5, 5));
//класс предназначен для парсинга строки с выделением из нее
//отдельных токенов (слов).
StringTokenizer st = new StringTokenizer("7 8 9 / 4 5 6 * 1 2 3 —
0 . = +");
 while(st.hasMoreTokens())
 add(new JButton(st.nextToken()));
 setSize(200, 200);
  setVisible(true);
```

```
  } }

public class SimpleFrame {
   public static void main(String[] args) {

      JFrame f= new GridTest("  Менеджер GridLayout");
      f.setLocation(400,200);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

Блочное расположение BoxLayout

Блочное расположение BoxLayout — прекрасная альтернатива всем остальным менеджерам расположения (рис.17,18, листинг 22). Обладая возможностями GridBagLayout, расположение BoxLayout не сложнее BorderLayout. Менеджер блочного расположения выкладывает компоненты в контейнер блоками: столбиком (по оси Y) или полоской (по оси X), при этом каждый отдельный компонент можно выравнивать по центру, по левому или по правому краям, а также по верху или по низу. Расстояние между компонентами по умолчанию нулевое, но для его задания существуют специальные классы (об этом чуть позже). Как располагаются компоненты, хорошо видно на рис. 17.

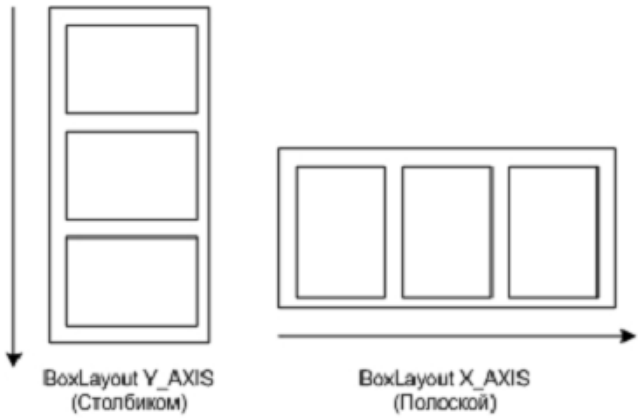


Рис.17. Расположение элементов внутри панелей при компоновке BoxLayout

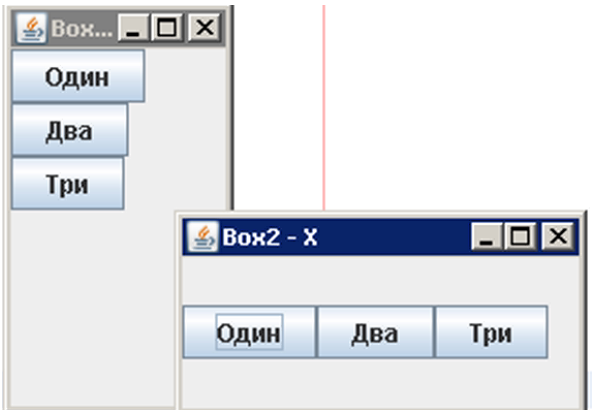


Рис.18. Компоновщик BoxLayout с панелями

Здесь отметим, что, хотя все мыслимые потребности по размещению элементов охвачены существующими компоновщиками, есть возможность разрабатывать свои уникальные варианты компоновки.

Листинг 22

```
// Блочное расположение элементов
import javax.swing.*;
import java.awt.*;

public class Box1 extends JFrame {
 public Box1() {
  super("Box1 - Y");
  setSize(400, 200);
  setDefaultCloseOperation( EXIT_ON_CLOSE );
  // получаем панель содержимого
  Container c = getContentPane();
  //устанавливаем блочное расположение по оси Y (столбиком)
  BoxLayout boxy = new BoxLayout(c, BoxLayout.Y_AXIS);
  c.setLayout(boxy);
  // добавляем компоненты
  c.add(new JButton("Один"));
  c.add(new JButton("Два"));
  c.add(new JButton("Три"));
  // выводим окно на экран
  setVisible(true);
```

```
}
static class Box2 extends JFrame {
 public Box2() {
  super("Box2 - X");
  // устанавливаем размер и позицию окна
  setSize(400, 200);
  setLocation(100, 100);
  setDefaultCloseOperation( EXIT_ON_CLOSE);
  // получаем панель содержимого
  Container c  = getContentPane();
  // устанавливаем блочное расположение по оси X (полоской)
  BoxLayout boxx = new BoxLayout(c, BoxLayout.X_AXIS);
  c.setLayout(boxx);
  // добавляем компоненты
  c.add( new JButton("Один"));
  c.add( new JButton("Два"));
  c.add( new JButton("Три"));
  // выводим окно на экран
  setVisible(true);
 }
}
public static void main(String[] args) {
 new Box1();
 new Box2();
}
}
```

В этом примере создаются два окна. В одном из них реализовано блочное расположение по оси Y, в другом — блочное расположение по оси X. Как легко убедиться, при блочном расположении компоненты действительно размещаются вплотную друг к другу.

Вы можете видеть, что конструктор класса BoxLayout несколько необычен — ему необходимо указать контейнер, в котором он будет функционировать. Ни в одном из рассмотренных нами прежде менеджеров расположения такого не требовалось.

Остальные менеджеры используются реже и остаются на самостоятельное рассмотрение.

Задание к лабораторной работе №4 (ч. 1)

1. Выполните задания 4.1 и 4.2. Ответы на вопросы разместить в отчете. Опробовать различные варианты компоновки.
2. Найдите свой вариант ко второй части лабораторной работы 4 и выполните разработку графического интерфейса для своего варианта.

Требования к отчету к ЛР №4 (ч. 1)

1. Постановка задачи.
2. Привести программу и скриншот исполнения программы.
3. Ответы на вопросы к заданиям 4.1 и 4.2.
   Классы должны находиться в разных файлах.

ЛАБОРАТОРНАЯ РАБОТА № 4 (ч. 2)

СОБЫТИЯ И ИХ ОБРАБОТКА

Во всех предыдущих примерах кнопки присутствовали, но были бесполезными — они ничего не делали. В следующем примере кнопка получит связанное с ней действие — при ее нажатии программа будет завершаться. Этого можно добиться разными способами, здесь будет приведен один из наиболее распространенных.

Этот способ возлагает обязанности обработчика события кнопки на объект безымянного класса (безымянного потому, что объект такого класса создается всего в одном месте программы, и нам нет необходимости именовать этот класс, так как на него не нужно нигде ссылаться).

Листинг 23

```
public static void main(String[] args) {
    JFrame f = new JFrame();
    JPanel p = new JPanel();
    JButton b = new JButton("Press me!");
b.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent e)
{ System.exit(0); }
});
   p.add(b);
   f.setSize(300, 200);
   f.add(p);
   f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   f.setVisible(true);
}
```

Чтобы программа работала, нужно в начало файла (после import javax.swing.*;) добавить import java.awt.event.*; .

Безымянный (анонимный) класс здесь описывается прямо в операции new после ActionListener(), что указывает на то, что создается новый анонимный класс, реализующий интерфейс ActionListener, который имеет метод actionPerformed(ActionEvent e) – именно этот метод вызывается при наступлении события ActionEvent, например, нажатия кнопки. При этом операция new ActionListener() создает объект такого класса и возвращает ссылку на этот объект; в данном примере ссылка передается методу addActionListener() кнопки, и объект безымянного класса регистрируется как слушатель кнопочного события. Таких «слушателей» может быть сколько угодно, и каждый раз после нажатия кнопки у каждого из них вызывается метод actionPerformed().

Рассмотрим общую схему обработки событий в Java. Она основана на архитектурном шаблоне «Наблюдатель» и носит название «Event-Driven» (или «Event-Delegation») – событийная модель и используется для обработки событий в большинстве языков визуального программирования.

Пользователь воздействут на исходный объект (например, кнопка – Button или поле ввода – Textfield). При срабатывании исходный объект создает объект события для захвата действия (например, щелчок мышью по *x* и *y*, введенные тексты и т. п.). Этот объект события будет передан всем зарегистрированным объектам-слушателям, и соответствующий метод обработчика события будет вызван для выполнения необходимого действия.

Чтобы «выразить интерес» к событию определенного источника, слушатели должны быть зарегистрированы в источнике. Другими словами, слушатель (и) «подписывается» на событие источника, а источник «публикует» событие всем своим подписчикам при активации. Такая схема еще известна как шаблон «подписки-публикации».

В приведенном примере метод actionPerformed() является методом обратного вызова. Другими словами, вы никогда не вызываете actionPerformed() в своих кодах явно. ActionPerformed() вызывается графической подсистемой в ответ на определенные действия пользователя.

Итак, в обработке событий участвуют три типа объектов: источник, слушатель(и) и объект события (рис.19).

Рис.19. Функциональная схема событийной модели в Java

Подробная последовательность шагов обработки событий (рис.19).

Шаг 1. Исходный объект регистрирует своих слушателей для определенного типа события.

Шаг 2. Источник инициирует событие при срабатывании. Например, нажатие Button запускает ActionEvent, нажатие кнопки мыши запускает MouseEvent, нажатие клавиши на клавиатуре запускает KeyEvent и т. д.

Источник и слушатель понимают друг друга через согласованный интерфейс, являющийся частью названия методов. Например, если источник гнерирует событие под названием XxxEvent (например, **Mouse**Event), включающее различные режимы работы (например, щелчок мышью, нажатие мыши и отпускание мыши), то нам нужно объявить интерфейс с именем XxxListener (например, **Mouse**Listener), содержащий имена методов-обработчиков (напомним, что interface содержит только абстрактные методы без реализации). Например, интерфейс MouseListener объявлен следующим образом с пятью режимами работы.

Листинг 24

```
// Интерфейс MouseListener, который объявляет подписываение
    // обработчиков для различных режимов работы.
   public interface MouseListener {
     // Вызывается при нажатии кнопки мыши:
   public void mousePressed (MouseEvent evt);
   // Вызывается после отпускания кнопки мыши:
   public void mouseReleased (MouseEvent evt);
   // Вызывается после нажатия кнопки мыши (нажат и отпущен):
   public void mouseClicked (MouseEvent evt);
   // Вызывается, когда указатель мыши вошел в компонент:
    public void mouseEntered (MouseEvent evt);
   // Вызывается, когда указатель мыши вышел за пределы
   // компонента:
    public void mouseExited (MouseEvent evt);
   }
```

Все слушатели должны предоставить собственные реализации (т.е. запрограммированные ответы) для всех абстрактных методов, объявленных в интерфейсе XxxListener. Таким образом, слушатель(и) может соответствующим образом реагировать на эти события. Например,

Листинг 25

```
    // Пример слушателя MouseListener, который обеспечивает
реализацию методов обработчика событий
   Class MyMouseListener implements MouseListener {
      @Override
      public void mousePressed (MouseEvent e) {
        System.out.println («Нажата кнопка мыши!»);
      }

      @Override
      public void mouseReleased (MouseEvent e) {
        System.out.println («Кнопка мыши отпущена!»);
      }

      @Override
      public void mouseClicked (MouseEvent e) {
```

```
        System.out.println («Кнопка мыши нажата (нажата и отпущена)!»);
    }

    @Override
    public void mouseEntered (MouseEvent e) {
        System.out.println («Указатель мыши вошел в исходный компонент!»);
    }

    @Override
    public void mouseExited (MouseEvent e) {
        System.out.println («Указатель мыши завершен – исходный компонент!»);
    }
}
```

Исходный объект порождает объект – событие XxxEvent, который инкапсулирует необходимую информацию об активации. Например, если точка с координатами *x, y* – это положение указателя мыши либо введенного текста, то для каждого из слушателей этого события в списке слушателей источник вызывает соответствующий обработчик, который предоставляет запрограммированный ответ.

На рис. 20 показана диаграмма последовательностей, которая демонстрирует шаги обработки события. Диаграмма перекликается с функциональной схемой событийной модели в Java (рис. 19).

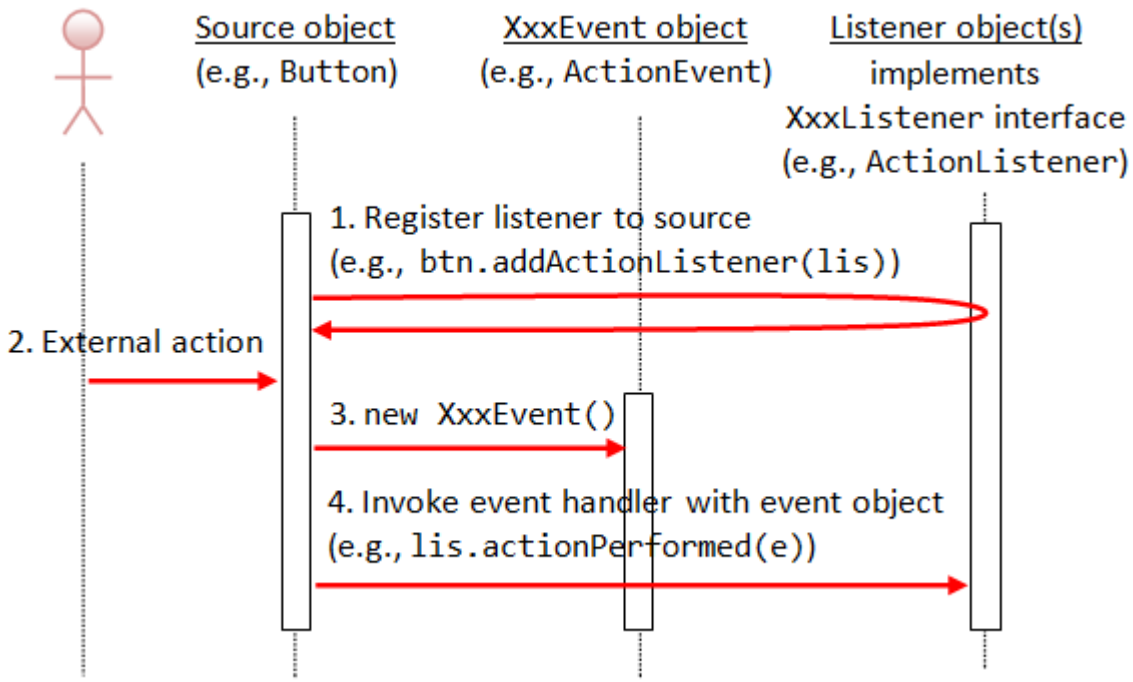


Рис.20. Диаграмма последовательностей событийной модели в Java

В качестве очередного примера напишем программу, в которой при каждом нажатии на кнопку меняется надпись на кнопке (для простоты чередуются «Hello!» и «GoodBye!»).

Листинг 26

```
// Обработка нажатия кнопки
public static void main(String[] args) {
   JFrame  f = new JFrame();
   JPanel  p = new JPanel();
   JButton b = new JButton("Hello!");

    b.addActionListener(new ActionListener() {
        boolean flag = true;
          public void actionPerformed(ActionEvent e) {
           flag = !flag;
((JButton) e.getSource()).setText(flag?"Hello!" :"Goodbye!");
      }
 });
 p.add(b);
 f.setSize(300, 200);
```

```
  f.add(p);
  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  f.setVisible(true);
 }
}
```

Обратите внимание, что после проверки флага программа обращается к событию e.getSource(), извлекая источник события (саму кнопку) и изменяя текст на ней.

Задача 4.3 для самостоятельного решения на занятии

Написать программу, при запуске которой в главном окне появляется кнопка, на которой написано 10, и при каждом нажатии это число уменьшается на 1 до тех пор, пока оно не станет равно 1. В этом случае при нажатии на кнопку программа завершается.

Не всегда, однако, удается обойтись безымянным классом обработчика событий. Тогда приходится писать явный класс. В качестве примера рассмотрим программу с двумя кнопками, в которой каждая кнопка меняет текст на другой кнопке.

Листинг 27

```
// Обработка нажатия кнопки – изменение текста соседней кнопки
    class Handler implements ActionListener {
     boolean flag = true;
     JButton b;
     public Handler (JButton bb) { b = bb;}
        @Override
         public void actionPerformed(ActionEvent e) {
         flag = ! flag;
          b.setText(flag?"Hello!":"Goodbye!");
     }
    }

    public class Events extends JFrame {
       public static void main(String[] args) {
```

```
        JFrame f = new JFrame("Events") ;
        JPanel p = new JPanel() ;
        JButton b1 = new JButton("Hello!");
        JButton b2 = new JButton("Hello!");
        Handler h1 = new Handler(b2);
        Handler h2 = new Handler(b1);
        b1.addActionListener(h1);
        b2.addActionListener(h2);
        p.add(b1);
        p.add(b2);
    f. setSize(300, 200);
    f.add(p);
    f.setDefaultCloseOperation(EXIT_ON_CLOSE);
    f.setVisible(true);
     }
       }
```

Как видим из листинга 27, при создании отдельного метода-обработчика пришлось выполнить два дополнительных действия: унаследоваться от интерфейса ActionListener и переопределить метод actionPerformed(), который и содержит программу-обработчик. Теперь можно создать объект-обработчик и присоединить его к нужной кнопке.

Интерфейс ActionListener имеет всего один метод — actionPerformed(), вызываемый у объекта – обработчика событий для выполнения надлежащих действий по обработке события.

Конечно, существуют и другие интерфейсы и методы для обработки различных событий (для получения дополнительной информации обращайтесь к лекциям автора и к справочной литературе).

Важно заметить, что существует несколько способов написания обработчиков событий:

1. Анонимный класс.
2. Отдельный класс-обработчик.
3. Обработчик во вложенном классе.
4. Лямбда-функция (похожа на анонимный класс).
5. Сам класс, например, форма, является слушателем своих событий.

Примеры для всех перечисленных способов приведены в лекциях.

Приведем пример разработки программы с графическим интерфейсом.

Для добавления графических примитивов к компоненту в Java необходимо использовать класс Graphics. Как это сделать? JFrame, как и множество других элементов таких, как JPanel или JButton, наследуют класс Component, имеющий метод paint(Graphics g). Переопределив этот метод в вашем классе, расширяющем сам JFrame или любой другой компонент, можно вмешаться в рисование этого компонента.

Листинг 28

```
//При клике на холст рисуются круги случайно выбранного
//радиуса и цвета

package paintoval;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PaintOval {
   public static void main(String[] args){

EventQueue.invokeLater(new Runnable() {
    @Override
    public void run() {

       JFrame f = new JFrame();
       f.setContentPane(new TestPanel());
       f.setSize(300, 200);
       f.setResizable(false);
       f.setTitle("Произвольные круги");
       f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       f.setVisible(true);
```

```
    }
});
}

 public static class TestPanel extends JPanel{
    LinkedList<Circle> circles = new LinkedList<>();
    Random r = new Random();

    public TestPanel(){
        addMouseListener(new MouseAdapter(){
//Сама панель является слушателем своих событий
       @Override
       public void mouseClicked(MouseEvent e){
       circles.add(new Circle(e.getX(), e.getY(), r.nextInt(51)+30,
       new Color(r.nextInt(128), r.nextInt(128), r.nextInt(128),
       r.nextInt(128)+100)));
       repaint(); //метод перерисовывает весь компонент
            }
        });
    }
//Здесь выполняется перерисовка кругов
    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);

        circles.stream().map((c) -> {
            g.setColor(c.getColor());
            return c;
        }).forEach((c) -> {
            g.fillOval(c.getX(), c.getY(), c.getSize(), c.getSize());
        });
        }
    }
}

// Класс "Окружность"
import java.awt.Color;
public class Circle{
```

```
    private int x;
    private int y;
    private int size;
    private Color color;

    public Circle(int x, int y, int s, Color c){
        this.x = x;
        this.y = y;
        size = s;
        color = c;
    }

    public int getX(){return x;}
    public int getY(){return y;}
    public int getSize(){return size;}
    public Color getColor(){return color;}
}
```

Обратите внимание, что функция paintComponent(Graphics g) явно нигде не вызывается: она неявно вызывается функцией repaint().

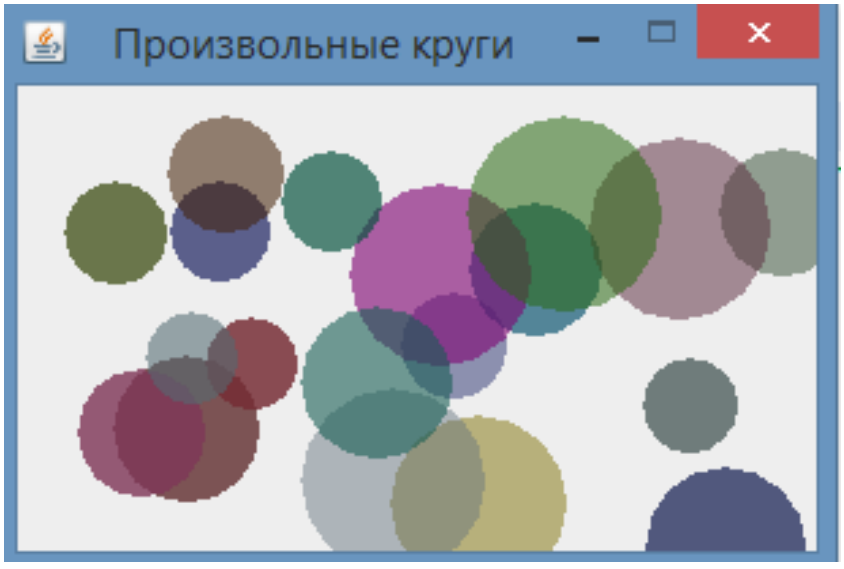


Рис.21. Результат работы программы «Произвольные круги»

Концепция потоков в Swing

Важно. Во всех предыдущих примерах (кроме последнего) графические окна были запущены без учета потокобезопасности: разработчики Swing объявили, что обращаться к компонентам из любого потока, который не является потоком диспетчера событий, небезопасно. А поэтому **правильно** конструировать пользовательский интерфейс в потоке диспетчера событий, используя вызов статического метода EventQueue.invokeLater(Runnable), принимающий в качестве аргумента экземпляр класса, реализующего интерфейс Runnable (листинг 28).

Интерфейс Runnable предназначен для реализации потоков выполнения в Java. Он содержит единственный метод void run(), который вызывается при запуске потока:

```
public interface Runnable { public void run(); }
```

Метод invokeLater() принимает в качестве параметра объект Runnable и отправляет этот объект в поток диспетчеризации событий, который выполняет метод run(). Вот почему безопасно, чтобы код Swing выполнялся в методе run().

Напомню, что поток – это легковесный процесс. В отличие от настоящих процессов, которые обычно представляют отдельные программы, несколько потоков могут относиться к одной программе и иметь общее адресное пространство, т.е. обращаться к одним и тем же структурам данных. Как вы уже догадались, потоки имеют важное значение в Swing. Программы с графическим интерфейсом управляются событиями, такими как нажатие кнопки или необходимость перерисовать окно. Программа заносит события в очередь и затем исполняет их в процессе получения. Обработка событий выполняется в так называемом потоке диспетчеризации событий (EDT). Для того чтобы интерфейс был отзывчивым, в этом потоке нельзя исполнять действия, занимающие много времени, такие как загрузка большого файла с диска или из Интернета. Для таких действий следует создавать отдельные потоки.

Поскольку последовательность действий в программе с несколькими потоками может быть труднопредсказуемой, требуется

обращать особое внимание, когда разные потоки обращаются к одной и той же структуре данных в памяти. Классы, отвечающие за компоненты Swing, не являются потокобезопасными, т.е. обращение к ним из разных потоков в неправильном порядке может привести к повреждению интерфейса. Это сделано специально, потому что попытки сделать библиотеку интерфейса потокобезопасной приводят к ее усложнению и замедлению. Поэтому для корректной работы компонентов Swing требуется, чтобы их создание и любое изменение выполнялись из потока диспетчеризации событий. Это гарантирует правильную последовательность действий.

Метод main() исполняется в главном потоке программы, отличном от потока диспетчеризации. Следовательно, для выполнения правила об однопоточном доступе к компонентам Swing, метод main() должен попросить поток диспетчеризации запланировать создание интерфейса. Это делается с помощью статического метода SwingUtilities.invokeLater(Runnable) либо EventQueue.invokeLater(Runnable) (листинг 29).

Листинг 29

```
import java.awt.*;
import javax.swing.*;

 public class Window {
 public static void main(String args[]) {
 EventQueue.invokeLater(new Runnable() {

public void run() {
JFrame frame = new JFrame("Пример окна верхнего уровня");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(300, 200);
frame.setVisible(true);
        }
 });
 }
}
```

Задача 4.4 для самостоятельного решения на занятии

Напишите программу, при запуске которой в главном окне появляются две кнопки, каждая из которых включает / отключает другую (для включения кнопки нужно вызвать метод setEnabled(true), для отключения — setEnabled (false)).

Средства для рисования в java

В листинге 28 был приведен пример рисования на компоненте типа JPanel. Поясним технологию рисования.

Класс JComponent и его наследники содержат метод paintComponent(Graphics g), который прорисовывает компонент. Поскольку это защищенный метод (protected), он доступен в подклассах JComponent и может быть переопределен в них. Именно в этих переопределенных методах можно рисовать на поверхности компонента. Напрямую вызывать метод paintComponent() не следует. Java вызывает его, когда компонент нужно перерисовать: например, когда окно показывается в первый раз или меняет размер. Если внутреннее состояние компонента изменилось и требует перерисовки, программа может запросить ее с помощью метода repaint(), определенного в классе Component. Единственный параметр метода paintComponent() имеет тип java.awt.Graphics. Передаваемый этому методу аргумент называется графическим контекстом. Об этом объекте можно думать, как о поверхности компонента. Он содержит состояние инструментов рисования: текущие цвет и шрифт, область рисования, режим рисования (как вновь рисуемые точки комбинируются с существующими) и другую информацию.

На самом деле метод paintComponent() рисует компонент не своими силами, а поручает это представителю пользовательского интерфейса, который изображает компонент в соответствии с текущим стилем оформления. Так, для класса JPanel представитель пользовательского интерфейса закрашивает панель цветом фона панели, если она объявлена непрозрачной. Следовательно, при переопределении paintComponent() в подклассе JPanel первым делом следует вызвать переопределяемый метод.

```
    @Override
protected void paintComponent(Graphics g) {
super.paintComponent(g);
...
}
```

Ели часть рисунка выйдет за пределы панели, она будет просто отброшена. На самом деле графический контекст помещается в прямоугольник, который следует перерисовать и который возвращается методом getClipBounds(). Метод paintComponent() может использовать эту информацию и рисовать только внутри данного прямоугольника.

Класс Graphics содержит методы для рисования отрезков, прямоугольников, многоугольников, эллипсов и их дуг (рис. 21). Есть также методы для вывода текста и растровых изображений.

Все координаты, используемые в методах класса Graphics, указываются в пикселях и имеют тип integer. Углы α и β, являющиеся аргументами drawArc(), измеряются в градусах и также имеют тип integer.

В дополнении к методам, показанным на рис. 21, в классе Graphics есть методы для заполнения соответсвующих фигур устновленным цвтом – fillRect(), fillPolygon(), fillOval() и fillArc(). Последний метод закрашивает сектор эллипса (фигуру, ограниченную дугой и двумя радиусами). Цвет рисования устанавливается методом setColor(Color c) класса Graphics.

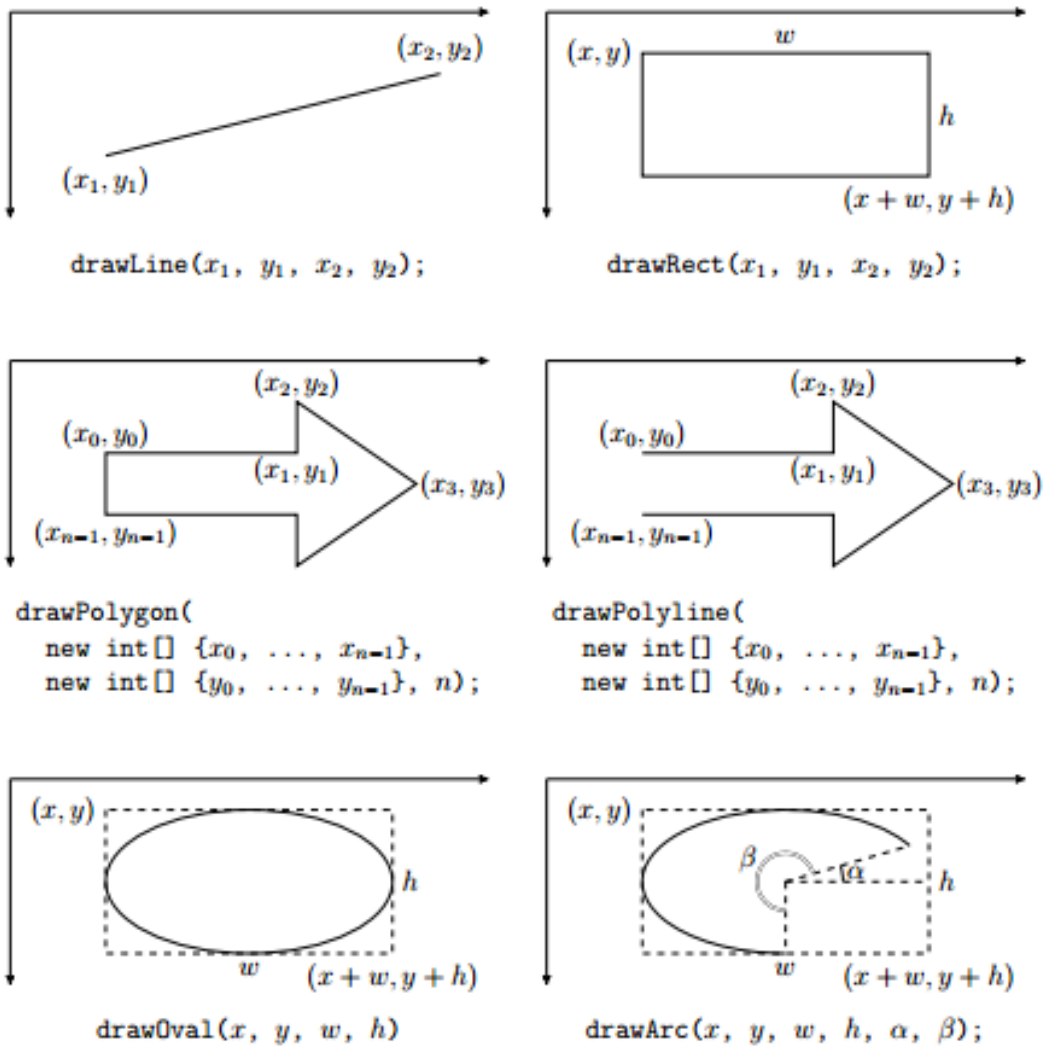


Рис.22. Рисование геометрических примитивов в Swing

Задания к лабораторной работе №4 (ч. 2)

В таблице в первой колонке после номера варианта указана степень сложности задания (чем больше число, тем выше сложность по пятибалльной шкале).

Таблица 4

Варианты заданий к лабораторной работе №4 (ч. 2)

Вариант (сложность)	Постановка задачи
1 (5)	Разработать игру 2048.
2 (5)	Разработать игру «уголки» с визуальным интерфейсом. Оба игрока – люди.

3 (5)	Разработать простой графический редактор, который при нажатии на кнопку «окружность» рисует окружность с заданным радиусом. Фигуры: линия квадрат, круг, овал. Все фигуры должны реализовать интерфейс IDrawFigure и представлять собой отдельные классы. В этом задании придется разбираться с рисованием и перерисовкой – как в примере с произвольными окружностями (листинг 28)
4 (4)	Реализовать визуальную работу банкомата — снятие и пополнение денег, вычисление остатка на счету и процентов по депозиту (класс Account и класс Bancomat)
5 (3)	Разработать простую кулинарную книгу (10 блюд для мультиварки): при нажатии на кнопку с названием блюда в окне ингредиенты отобразить ингредиенты, в окне рецепт – рецепт. С базой данных – на 5.
6 (4)	Разработать приложение–органайзер на месяц: в запись для указанного дня переносить из строки ввода задание(я) на этот день. С базой данных на 5.
7 (5)	Разработать игру типа «Balls» (Разноцветные шарики падают на квадратное поле в клеточку, игрок имеет возможность их поменять местами при наличии рядом свобоного поля. При выстраивании в ряд 5 шариков одного цвета они исчезают).
8 (3)	Разработать каталог машин в автосалоне до 15 наименований: при выборе марки машины из выпадающего списка отображать характеристики автомобиля и ее фото. С базой данных – на 5.
9 (5)	Разработать игру «Пятнашки».
10 (5)	Разработать приложение, которое имеет холст для произвольного рисования и кнопки для выбора цвета пера.
11 (5)	Тренажер клавитауры. По прямоугольному полю движется круг с буквой внутри. Если нажата клавиша, соответствующая букве, игроку добавляется балл, иначе – нет. Возможны варианты с ускорением, уровнями сложности, латинскими буквами.
12 (3)	Разработать справочник кинофильмов. При клике на списке фильмов (не менее 15 наименований) в окне отображается обложка фильма и его описание. С базой данных – на 5.

Замечание ко всем вариантам. События типа ActionEvent проявляются в компонентах Button, List, TextField, JComboBox, JTextField, кнопках класса AbstractButton и его наследниках. То есть во всех вариантах необходимо добавлять объект (или применять интерфейс) ActionListener, в котором реализовать обработчик в функции public void actionPerfomed(ActionEvent e).

Требования к отчету к ЛР №4 (ч. 2)

1. Постановка задачи.
2. UML-диаграмма классов (не забудьте о вложенных классах, если они есть).
3. Диаграмма последовательности обработки одного из событий.
4. Текст программы с комментариями. Классы должны находиться в разных файлах.
5. Скриншоты исполнения программы.

ЛАБОРАТОРНАЯ РАБОТА № 5

Многопоточность

Цель. Изучить приемы и методы организации и управления потоками.

Общие сведения о потоках

В предыдущей лабораторной работе нам уже пришлось столкнуться с потоками и вспомнить, что поток – это легковесный процесс, набор исполняемых инструкций. Практически все современные ОС позволяют исполнять несколько таких наборов одновременно. Хотя одновременно – понятие относительное. Правильно было бы сказать – одновременно с точки зрения пользователя, ибо процессор один и исполнять он может только один набор инструкций (разумеется, если процессоров либо ядер несколько, то все усложняется). Разберем наиболее простой вариант с одним процессором, где для создания видимости одновременной работы процессор исполняет определенное количество инструкций одного потока, после чего переключается на другой. Это происходит очень часто, в результате чего создается иллюзия одновременной работы (рис. 22).

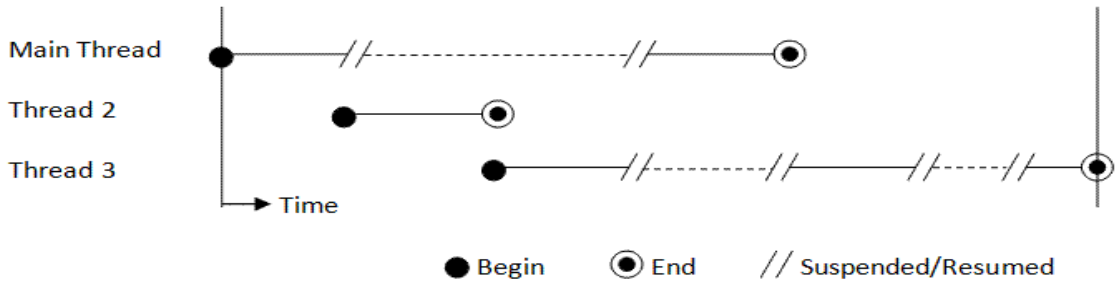


Рис. 23. Диаграмма распределения времени и работы между 3 потоками для одного процессора

Для чего это нужно? Современное программное обеспечение по большей часто асинхронно. Приложение ждет реакции пользователя, ждет прихода данных по сети, ждет готовности устройства. Приложение ждет. Если вы посмотрите на загрузку процессора, то

увидите, что чаще всего 99% времени он простаивает. Соответственно, пока одна задача находится в стадии ожидания, можно заниматься другой. Например, во время ожидания прихода данных по сети можно отрисовать то, что уже пришло. Это если говорить о браузере. Да и вообще, в ОС одновременно происходит множество разных событий. Если бы всеми ими занимался один единственный поток – все работало бы значительно медленнее. Если бы вообще работало.

Следует разграничить два понятия – поток и процесс. Процесс – это задача операционной системы. У него собственное адресное пространство, с ним может быть проассоциировано несколько потоков. Поток же – это гораздо более мелкая единица. Все потоки разделяют адресное пространство породившего их процесса и имеют доступ к одним данным.

Вот тут-то и зарыта собака. Поскольку несколько потоков имеют доступ к одним данным, они их могут менять. Иногда это может привести к очень большим проблемам. Для решения этих проблем и нужна синхронизация (о ней далее).

Базовые классы для работы с потоками

Многозадачность позволяют реализовывать следующие классы и интерфейсы:

- интерфейс Runnable;
- класс Thread;
- класс java.util.Timer;
- класс javax.swing.Timer.

Последние два связаны с установкой периодичности исполнения заданий.

Поток выполнения в Java представляется экземпляром класса Thread (рис. 24 а), листинг 30). Для того чтобы написать свой поток исполнения, необходимо наследоваться от этого класса и переопределить метод run().

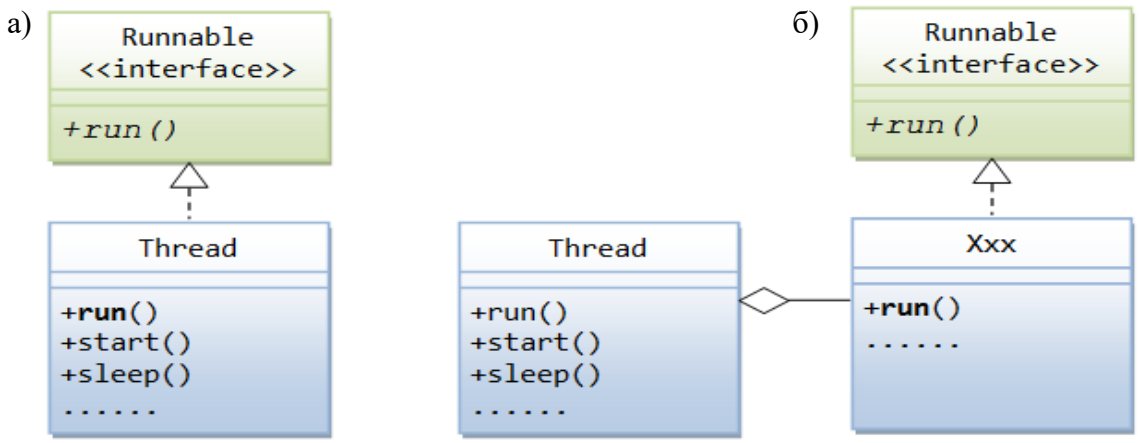


Рис.24. Два варианта создания потока: а) наследоваться от класса Thread; б) имплементировать интерфейс Runnable

Листинг 30

```
public class MyThread extends Thread {
   public void run() {
        // некоторое долгое действие, вычисление
        long sum=0;
        for (int i=0; i<1000; i++) {
        sum+=i;
  }
System.out.printin(sum);
}
```

Метод *run()* содержит действия, которые должны исполняться в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника и вызвать наследованный метод *start()*, который сообщает виртуальной машине, что необходимо запустить новый поток исполнения и начать в нем исполнять метод run().

```
MyThread t = new MyThread();
t.start();
```

В результате этого на консоли появится результат: 499500.

Когда метод *run()* завершен (в частности, встретилось выражение return), поток выполнения останавливается. Однако ничто не препятствует записи бесконечного цикла в этом методе.

В результате поток не прервет своего исполнения и будет остановлен только при завершении работы всего приложения.

Класс Thread содержит несколько методов для управления потоками:

– getName() – получить имя потока;
– getPriority() – получить приоритет потока;
– isAlive() – определить, выполняется ли поток;
– join() – ожидать завершение потока;
– run() – запустить поток;
– sleep() – приостановить поток на заданное время;
– start() – запустить поток вызовом метода start().

Обратите внимание, что метод start() все-равно вызовет run(), и при этом создастся отдельный поток. Но если мы напрямую вызовем метод run (), то новый поток не будет создан, а метод run () будет выполнен как обычный вызов метода для текущего вызывающего потока, и многопоточность не будет выполняться.

Интерфейс Runnable

Описанный подход обладает одним недостатком: поскольку в Java отсутствует множественное наследование, требование наследоваться от Thread может привести конфликту. Если еще раз посмотреть на приведенный пример (листинг 30), то станет понято, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается альтернативный способ создания своего потока исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод – уже знакомый void run() (рис. 24 б). Перепишем пример из листинга 30 с помощью интерфейса Runnable:

Листинг 31

```
public class MyRunnable implements Runnable {
public void run() {
// некоторое долгое действие, вычисление
  long sum=0;
  for (int i=0; i<1000; i++)
```

```
 sum+=i;
 System.out.printin(sum); }}
```

При этом незначительно поменяется процедура запуска потока:

```
Runnable r= new MyRunnable() ;
Thread t = new Thread(r);
// можно так:   Thread t = new Thread(new MyRunnable());
t.start ();
```

Когда объявляется новый класс с интерфейсом Runnable, необходимо использовать конструктор:

```
Thread(Runnable объект_потока, String имя_потока)
```

В первом параметре указывается экземпляр класса, реализующего интерфейс. Он определяет поведение потока. Во втором параметре передаётся имя потока.

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), содержащим полезную функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, решается в каждом конкретном случае (рис. 25). Подчеркнем, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможны только через метод Thread.start().

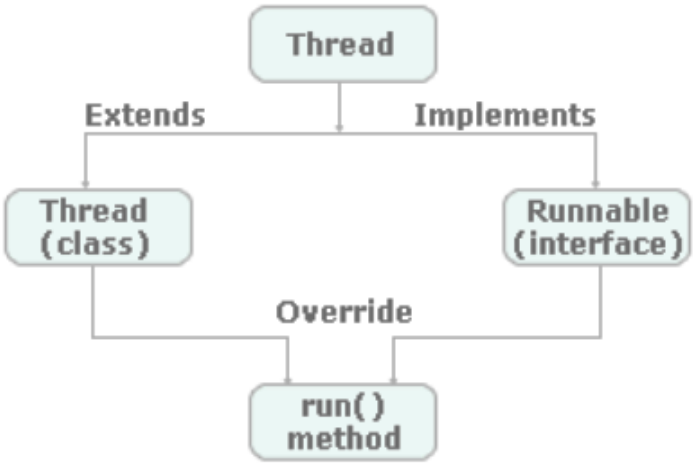


Рис. 25. Альтернативные варианты создания потоков

Документация Oracle дает следующие рекомендации. Какую из идиом с рис. 25 лучше использовать? Вторая идиома, в которой используется Runnable object, предоставляет более общий подход, поскольку имплементация интерфейса Runnable позволяет создать подкласс, отличный от Thread. Этот подход не только более гибкий, но он применим к API-интерфейсам управления потоками высокого уровня. Первая же идиома проще в использовании в простых приложениях, но ограничивается тем, что ваш класс будет потомком Thread. Однако среди ее преимуществ возможность обращения ко всем методам родительского класса Thread.

Интерфейс Runnable не имеет метода start(), а только единственный метод run(). Поэтому для запуска потока t следует создать экземпляр класса Thread с передачей экземпляра r его конструктору. Обратите внимание, что при прямом вызове метода run() поток не запустится, выполнится только тело самого метода (однократно).

А ещё Runnable является функциональным интерфейсом, начиная с Java 1.8. Это позволяет писать код задач для потоков ещё красивее:

```
public static void main(String []args){
        Runnable task = () -> {
                System.out.println("Hello, World!");
        };
        Thread thread = new Thread(task);
        thread.start();
}
```

Простые методы управления потоками

Коснемся нескольких возможностей класса Thread по передаче управления между потоками – это три простых метода:

– sleep();
– yeald();
– join().

Существуют также немного более сложные методы управления, относящиеся к методам синхронизации потоков (описаны далее).

Итак, поток можно приостановить на определенный промежуток времени (изнутри самог***о*** этого потока), дав, таким образом, другим потокам выполнить свои задачи. Делается это через статический метод Thread, sleep() с параметром – количеством миллисекунд, на которое приостанавливается поток. До истечения этого времени поток может быть выведен из состояния ожидания вызовом interrupt, о котором будет сказано далее.

Статический метод yield() тоже служит для передачи управления другим потокам: в результате его вызова происходит переключение контекста и процессор начинает исполнять код другого потока. Это нужно как в ситуациях, когда работа на текущий момент завершена и можно дать поработать другим (например, поток обрабатывает данные, все обработал, а новые еще не пришли), так и в ситуациях, когда поток занимается какими-нибудь интенсивными действиями, съедает большую часть процессора и не дает другим потокам работать (встречается в основном под Java ME, в условиях ограниченных ресурсов). Обратите внимание, что этот метод статический и действует только на текущий поток. Заставить таким образом чужой поток поделиться своим временем нельзя!

Метод join() блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания при обращении к методу join(long timeout). Этот метод часто используют при необходимости выполнить агрегацию результатов работы всех порожденных потоков потоком main.

Листинг 32 демонстрирует пример использования методов sleep() и join(). Предположим, что sleep() здесь символизирует какую-либо работу – длительные вычисления.

Листинг 32

```
class JoinThread extends Thread {
  public JoinThread (String name) {
       super(name);
  }
  public void run() {
     String nameT = getName();
     long timeout = 0;
```

```
      System.out.println("Старт потока " + nameT);
        try {
            switch (nameT) {
               case "First":  timeout = 2_000;
               case "Second":  timeout = 1_000;
              default: timeout = 0; break;
               }
           Thread.sleep(timeout);
           System.out.println("завершение потока " + nameT);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
}

//Главный класс
public class JoinRunner {
     static {
   System.out.println("Старт потока main");
   }

 public static void main(String[ ] args) {
   JoinThread t1 = new JoinThread("First");
   JoinThread t2 = new JoinThread("Second");

        t1.start();
        t2.start();
         try {
            t1.join();
// поток main остановлен до окончания работы потока t1
          } catch (InterruptedException e) {
                    e.printStackTrace();
           }
System.out.println(Thread.currentThread().getName());
// имя текущего потока
    }
}
```

Возможный вариант результата:

Старт потока main
Старт потока First
Старт потока Second
завершение потока Second
завершение потока First
main

В листинге был разработан класс-наследник Thread с именем JoinThread. Его метод run() сообщает о начале работы потока, затем устанавливает время задержки для «сна» потока в зависимости от его имени, и после завершения «сна» выдает сообщение о завершении работы потока. В main создаются два потока First и Second, главный поток main ожидает завершения потока First (рис. 26).

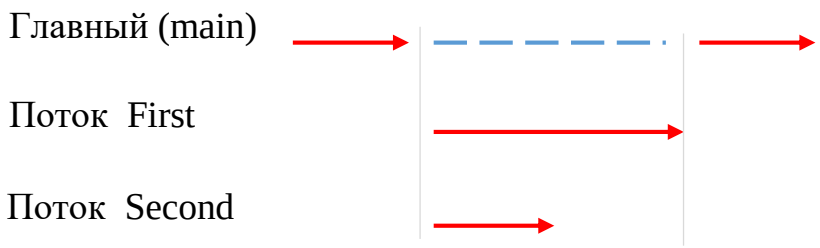


Рис. 26. Демонстрация последовательности работы потоков к листингу 32

Вызов метода yield() для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, для того чтобы другие потоки могли выполнять свои действия. Однако если требуется надежная остановка потока, то следует использовать его крайне осторожно или вообще применить другой способ.

Листинг 33

```
public class YieldRunner {
        public static void main(String[] args) {
      new Thread() {
          public void run() {
              System.out.println("старт потока 1");
```

```
                Thread.yield();
            System.out.println("завершение 1");
           }
       }.start();
       new Thread() {
            public void run() {
                System.out.println("старт потока 2");
                System.out.println("завершение 2");
                }
            }.start();
        }
  }
```

В результате может быть выведено:

```
старт потока 1
старт потока 2
завершение 2
завершение 1
```

Активизация метода yield() в коде метода run() первого объекта потока приведет к тому, что, скорее всего, первый поток будет приостановлен на некоторый квант времени, что даст возможность другому потоку запуститься и выполнить свой код.

Потоки в Java могу быть так называемыми демонами. Такой поток отличается от обычного тем, что он не препятствует окончанию работы виртуальной машины. Т.е. можно сказать, что виртуальная машина работает, пока существует хотя бы один поток, не являющийся демоном. Потоки-демоны удобно использовать для фоновых задач.

Примеры, демонстрирующие необходимость потоков

Оказывается, потоки нужны не только для ускорения работы приложения, иногда они совершенно необходимы просто для корректной его работы. Для того чтобы убедиться в необходимости потоков, рассмотрим пример «Безответного пользовательского интерфейса» (рис.27).

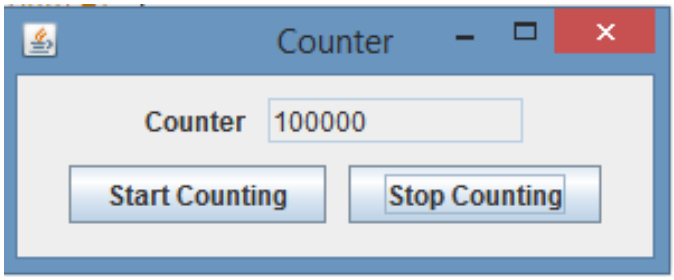


Рис 27. Пример, демонстрирующий необходимость потоков

Необходимо реализовать следующую функциональность. При нажатии на кнопку «Start Counting» начнется отсчет, результат которого должен появиться в поле «Counter». Нажатие кнопки «Stop Counting» должно остановить (приостановить) подсчет. Два обработчика событий нажатия кнопок общаются через boolean флаг под названием stop. Обработчик стоп-кнопки устанавливает Stop флаг; в то время как обработчик кнопки Start проверяет, был ли stop-флаг установлен, прежде чем продолжить счет.

Листинг 34

```
//Проект UnresponsiveUI
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**Программа иллюстрирует «Безответный интерфейс» в результате занятого диспетчера потоков Event-Dispatching Thread */

public class UnresponsiveUI extends JFrame {
  private boolean stop = false;  // start or stop the counter
  private JTextField tfCount;
  private int count = 1;

  /** Конструктор устанавливает GUI компоненты*/
  public UnresponsiveUI() {
    Container cp = this.getContentPane();
  cp.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 10));
    cp.add(new JLabel("Counter"));
```

```
    tfCount = new JTextField(count + "", 10);
    tfCount.setEditable(false);
    cp.add(tfCount);

    JButton btnStart = new JButton("Start Counting");
    cp.add(btnStart);
    btnStart.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent evt) {
        stop = false;
        for (int i = 0; i < 100000; ++i) {
//проверка нажатия клавиши Stop Counting,
//stop flag устанавливается в true
          if (stop) break;
          tfCount.setText(count + "");
          ++count;
        }
      }
    });
    JButton btnStop = new JButton("Stop Counting");
    cp.add(btnStop);
    btnStop.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent evt) {
        stop = true;  // установка флага stop в true
      }
    });

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("Counter");
    setSize(300, 120);
    setVisible(true);
  }

  /** Метод main */
  public static void main(String[] args) {
//Запуск GUI в диспетчере EDT для потокобезопасности
    SwingUtilities.invokeLater(new Runnable() {
```

```
        public void run() {
          new UnresponsiveUI();  // Let the constructor do the job
        }
      });
    }
}
```

Однако при нажатии кнопки «Start Counting» пользовательский интерфейс *замораживается* – значение счетчика не обновляется на дисплее, а пользовательский интерфейс не реагирует на нажатие кнопки «Stop Counting» или любое другое взаимодействие с пользователем.

Последовательность работы программы UnresponsiveUI следующая:

Метод main() стартует в потоке «main». Подсистема окон окружения JRE через Swingutilities.invokeLater() запускает 3 потока: «AWT-Windows» (daemon thread), «AWT-Shutdown» и «AWT-EventQueue». Последний известен как поток диспетчеризации (Event-Dispatching Thread или EDT), это единственный поток, ответственный за управление всеми событиями (такими, например, как нажатие кнопки) и обновление экрана для обеспечения безопасности потока в GUI и манипуляции GUI компонентами. Конструктор UnresponsiveUi () согласно планировщику запускается в потоке Event-Dispatching (методом invokeLater()), после этого обрабатываются все события. Поток «main» «живет» и после того, как метод main() завершается. Запускается новый поток с именем «DestroyJavaVM».

При нажатии на кнопку «Start Counting» метод actionPerformed() запускается внутри EDT, и EDT теперь полностью захвачен вычислительным циклом. Другими словами, пока происходят вычисления в цикле, EDT занят и не может обрабатывать другие события (т.е. нажатие на кнопку «Stop Counting» или закрытие окна), а также обновление формы невозможно до окончания цикла счета, а только когда EDT освободится. В результате вся форма и ее элементы «заморожены» до окончания цикла.

Рекомендуется запускать GUI в EDT через вызов метода invokeLater( ), потому что множество GUI-компонент не являются гарантированно потокобезопасными (thread-safe). Распределение доступа к GUI-компонентам в едином потоке гарантирует потокобезопасность. Предположим, что мы запустили конструктор напрямую в main() (используя поток «main»):

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

Последовательность действий будет следующей (рис.28):

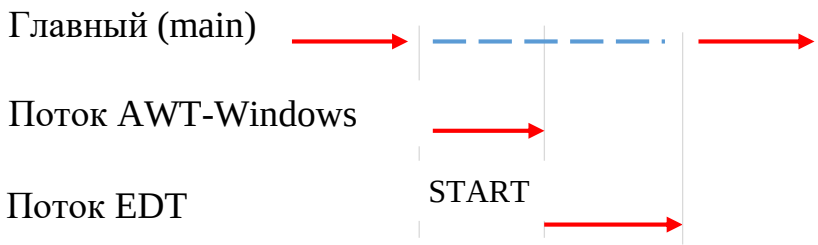


Рис.28. Диаграмма работы потоков для примера Листинг 34

Метод main () стартует в потоке «main».

Новый поток «AWT-Windows» (Daemon thread) запускается командой-конструктором «*new UnresponsiveUI ()*» (из-за «extends JFrame»).

После выполнения «*setVisibie (true)*» создаются два других потока - «*AWT-Shutdown*» и «*AWT-EventQueue*» (EDT).

Поток «main» «живет» и после того, как метод main () завершается. Запускается новый поток с именем «*DestroyJavaVM*».

Запущены 4 потока – «AWT-Windows», «AWT-Shutdown», «AWT- EventQueue» и «DestroyJavaVM».

Нажатием кнопки «Start Counting» запускается actionPerformed() внутри EDT.

В предыдущем случае планировщик EDT запускается сразу методом invokeLater( ), тогда как в последнем примере EDT стартует после setVisibie().

Листинг 35

```
//Изменим обработчик кнопки «Start Counting», запустив
//вычисления в отдельном потоке
```

```
JButton btnStart = new JButton("Start Counting");
    cp.add(btnStart);
    btnStart.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent evt) {
        stop = false;
// Создаем новый поток для вычислений
        Thread t = new Thread() {
          @Override
          public void run() {
            for (int i = 0; i < 100000; ++i) {
              if (stop) break;
              tfCount.setText(count + "");
              ++count;
            }
          }
        };
        t.start();  // вызов run()
      }
    });
```

Создается новый поток с помощью анонимного вложенного класса. Переопределяем run() метод, который выполняет наши вычисления. Метод start() запускает run() на выполнение в собственном потоке.

Способность к реагированию, таким образом, несколько улучшается, но необходимое значение подсчета все еще не отображается как следует – существует задержка в ответ на нажатие кнопки «Stop Counting». Это происходит потому, что поток подсчета добровольно не уступает управления потоку-планировщику EDT: «замороженный» EDT не может обновлять форму и реагировать на события. Однако JVM может заставить поток подсчета уступить управление согласно алгоритму-планировщику, что в результате выливается в задержку обновления формы.

Изменим программу, добавив вызов метода sleep() обработчику кнопки «Start Counting» (листинг 36):

sleep() дает возможность циклу подсчета уступить *(yield)* управление потоку планировщику (event-dispatching thread) для

обновления формы и отклик на кнопку «Stop Counting». Программа подсчета теперь будет работать так, как нужно, *sleep ()* обеспечит необходимую задержку.

Листинг 36

```
btnStart.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent evt) {
      stop = false;
      // Create a new Thread to do the counting
      Thread t = new Thread() {
        @Override
        public void run() {
*/ переопределение метода run() для определения поведения  в потоке*/
          for (int i = 0; i < 100000; ++i) {
            if (stop) break;
            tfCount.setText(count + "");
            ++count;
/**Приостановка потока методом sleep() и передача управления другим потокам. Обеспечение необходимой задержки в миллисекундах. */
            try {
              sleep(10);  // миллисекунды
            } catch (InterruptedException ex) {}
          }
        }
      };
      t.start();  // вызов run()
    }
  });
```

Метод sleep() приостанавливает текущий поток и устанавливает его в состояние ожидания на указанное количество миллисекунд. Теперь может быть запущен другой поток, а sleep() может быть прерван путем запуска метода interrupt() этого потока, что вызывает InterruptedException().

В этом случае поток, созданный для вычислений, уступает управление добровольно другим потокам после каждого счета (путем установки sleep(10)). Это позволяет потоку-планировщику обновлять окно при каждом нажатии на кнопку «Stop Counting».

Жизненный цикл потоков

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State (рис.29):

NEW – поток создан, но еще не запущен;

RUNNABLE – поток выполняется;

BLOCKED – поток блокирован;

WAITING – поток ждет окончания работы другого потока;

TIMEDWAITING – поток некоторое время ждет окончания другого потока;

TERMINATED – поток завершен.

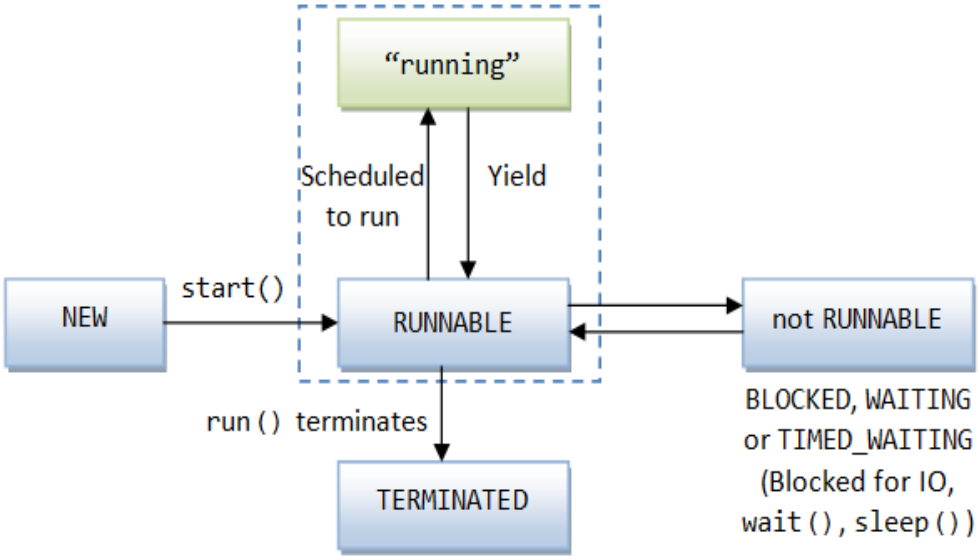


Рис. 29. Состояния жизненного цикла потоков

Получить значение состояния потока можно вызовом метода getState().

Поток переходит в состояние «неработоспособный» (WAITING) вызовом методов wait(), suspend() (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для

задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания (TIMEDWAITING) с помощью методов sleep(long millis) и wait(long timeout), при выполнении которого может генерироваться прерывание InterruptedException.

Вернуть потоку работоспособность после вызова метода suspend() можно методом resume() (deprecated-метод), а после вызова метода wait() методами notify() или notifyAll(). Поток переходит в «пассивное» состояние (TERMINATED), если вызваны методы interrupt(), stop() (deprecated метод) или метод run() завершил выполнение. После этого, чтобы запустить поток еще раз, необходимо создать новый объект потока. Метод interrupt() успешно завершает поток, если он находится в состоянии «работоспособный».

Если же поток неработоспособен, то метод генерирует исключительные ситуации разного типа в зависимости от способа остановки потока.

При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

Как указано ранее, методы *suspend(), resume()* и *stop()* являются deprecated-методами и запрещены к использованию, так как они не являются в полной мере «потокобезопасными».

Листинг 37

```
// Пример с методом join

public class ThreadJoinExample {
   public static void main(String[] args) {
      Thread t1 = new Thread(new MyRunnable(), "t1");
      Thread t2 = new Thread(new MyRunnable(), "t2");
      Thread t3 = new Thread(new MyRunnable(), "t3");

      t1.start();
 /**запускаем второй поток только после 2-секундного ожидания первого потока (или когда он умрет / закончит выполнение)*/
      try {
         t1.join(2000);
      } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }

        t2.start();

        /** запускаем 3-й поток только после того, как 1-й поток закончит свое выполнение*/
        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t3.start();

        /**даем всем потокам возможность закончить выполнение перед тем, как программа (главный поток) закончит свое выполнение*/
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Все потоки отработали, завершаем программу");
    }
}
class MyRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("Поток начал работу:::" + Thread.currentThread().getName());
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
        System.out.println("Поток отработал:::" + Thread.cur-
rentThread().getName());
    }

}
```

Программа даст следующий результат:

```
Поток начал работу:::t1
Поток начал работу:::t2
Поток отработал:::t1
Поток начал работу:::t3
Поток отработал:::t2
Поток отработал:::t3
Все потоки отработали, завершаем программу
```

Методы синхронизации потоков

Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл / объект / поток. Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова synchronized. В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе main() класса SynchroRun создаются два потока. В этом же методе создается экземпляр класса Resource, содержащий поле типа FileWriter, связанное с файлом на диске. Экземпляр Resource передается в качестве параметра обоим потокам. Первый поток записывает строку методом writing() в экземпляр класса Resource. Второй поток также пытается сделать запись строки в тот же самый объект Resource. Во избежание одновременной записи такие методы объявляются как synchronized.

Синхронизированный метод изолирует объект, после чего он становится недоступным для других потоков. Изоляция снимается, когда поток полностью выполнит соответствующий метод. Другой способ снятия изоляции — вызов метода wait() из изолированного метода. В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

Листинг 38

```
package synch;
import java.io.*;

public class Resource {
  private FileWriter fileWriter;
  public Resource (String file) throws IOException {
   // проверка наличия файла
   fileWriter = new FileWriter(file, true);
// Чтение и запись текстовых файлов можно осуществить
//с помощью классов FileReader и FileWriter из java.io.*;

  }
  public synchronized void writing(String str, int i) {
   try {
    fileWriter.append(str + i);
    System.out.print(str + i);
     Thread.sleep((long)(Math.random() * 50));
    fileWriter.append("->" + i + " ");
    System.out.print("->" + i + " ");
   } catch (IOException e) {
    System.err.print("ошибка файла: " + e);
   } catch (InterruptedException e) {
    System.err.print("ошибка потока: " + e);
   }
  }
  public void close() {
   try {
    fileWriter.close();
```

```
   } catch (IOException e) {
    System.err.print("ошибка закрытия файла: " + e);
   }
  }
}

package synch;
public class SyncThread extends Thread {
  private Resource rs;
  public SyncThread(String name, Resource rs) {
   super(name);
   this.rs = rs;
  }
  public void run() {
   for (int i = 0; i < 5; i++) {
// место срабатывания синхронизации
    rs.writing(getName(), i);
   }
  }
}

package by.bsu.synch;
import java.io.IOException;
public class SynchroRun {
  public static void main(String[ ] args)  {
   Resource s = null;
   try {
    s = new Resource ("data\\result.txt");
    SyncThread t1 = new SyncThread("First", s);
    SyncThread t2 = new SyncThread("Second", s);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
   } catch (IOException e) {
    System.err.print("ошибка файла: " + e);
   } catch (InterruptedException e) {
    System.err.print("ошибка потока: " + e);
```

```
    } finally {
     s.close();
    }
   }
  }
```

Задания к лабораторной работе №5

1. **Крестики – нолики**. Вариант графический (на отметку 5).

2. **Крестики – нолики**. Упрощенный вариант – консольный. Должны выдаваться сообщения: 1-й поток окончил! 2-й поток окончил! И имя выигравшего потока, если была выигрышная комбинация (на отметку 4).

3. **Гонки**. 3 участника наперегонки наращивают каждый свой ProgressBar. Вариант графический (на отметку 4).

4. **Гонки** – счет. 5 участников ведут счет до 10. По окончании должно выдаваться консольное сообщение каждым об окончании счета и имя потока-победителя (на отметку 4).

5. **Спорщики.** Три участника путешествия наперебой выбирают маршрут (суша, море, воздух). По окончании объявляется победитель и выбранный способ перемещения (на отметку 3).

6. **Слова.** Пять потоков в строгой очередности считывают слова из одного и того же файла (Поток1: слово1, слово5, слово9; Поток2: слово2, слово6, слово10 и т.д.) в свой массив слов выводят в консоль сначала каждый свой массив, потом весь считанный текст в обратном порядке: слово20, слово19, слово18, слово17…(на отметку 4).

7. **Лото.** Четыре игрока строго по очереди выставляют свои фишки (у каждого свой цвет фишки) на общую доску, о чем

каждый выдает сообщение, например: «Красный, фишка 3». По окончании (когда все клетки заполнены) сообщается, кто закончил первым. (Возможен графический и консольный вариант) (на отметку 5).

8. **Лото.** Предыдущая задача, но порядок игроков и размещение фишек случайное (на отметку 4).

9. **Автостоянка.** Доступно несколько машиномест. На одном месте может находиться только один автомобиль. Если все места заняты, то автомобиль не станет ждать больше определенного времени и уедет на другую стоянку (на отметку 5).

Требования к отчету к ЛР №5

1. Постановка задачи. Описание решения.
2. UML-диаграмма классов (не забудьте о вложенных классах, если они есть).
3. Временная диаграмма работы потоков (фрагмент).
4. Текст программы с комментариями. Классы должны находиться в разных файлах.
5. Скриншоты исполнения программы.

ЗАКЛЮЧЕНИЕ

Заканчивая изучение первой части курса «Программирование для мобильных платформ» подведем следующие итоги. На сегодняшний день существует множество языков программирования, платформ и фреймворков для разработки мобильных приложений. В данном руководстве были представлены ключевые моменты для изучения технологии Java и одноименного языка, который является одним из наиболее популярных языков для мобильной разработки в силу своих кросплатформенных возможностей, хорошего обеспечения документацией и учебной литературой, а также мощной информационной поддержки сообщества программистов.

Изучение языка Java – пролог для следующей части курса, в которой будут даны основы программирования для платформы Android. Набирающий популярность язык Kotlin также основан на знании Java и работает поверх JVM, но является более лаконичным и типобезопасным.

К итогам этой части курса можно также отнести знакомство с принципами SOLID.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

Основная

1. *Арнольд К., Гослинг Дж., Холмс Д.* Язык программирования Java. 3-е изд. М.: Вильямс, 2001.

2. *Эккель Б.* Философия Java. 4-е изд. СПб.: Питер, 2015.

3. *Буч Г., Рамбо Дж., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000.

4. *Макконнелл С.* Совершенный код. СПб. : Питер, 2005.

5. *Хорстманн К. С., Корнелл, Г.* Библиотека профессионала. Java. : Т.1: Основы. 10-е изд. М.: Вильямс, 2016.

6. *Хорстманн К. С., Корнелл, Г.* Библиотека профессионала. Java. : Т.2: Расширенные средства программирования. 10-е изд. М.: Вильямс, 2016.

7. *Ларман К.* Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование. 3-е изд. СПб.:Вильямс, 2013.

Дополнительная

Портянкин И. Swing. Эффектные пользовательские интерфейсы. 2-е изд. СПб.:Питер, 2011.

Шилдт Г. Java 8. Полное руководство: в 2 т. М.: ООО И.Д. Вильямс, 2015.

Шилдт Г. Swing. Руководство для начинающих. М.: ООО И.Д. Вильямс, 2007.

Блинов И. Н., Романчик В. С. Java. Методы программирования : учеб.-метод. пособие. Минск: Издательство «Четыре четверти», 2013.

Тейт. Б. Горький вкус Java. СПб. : Питер, 2003.

Фаулер М. UML. Основы. 3-е изд. СПб.: Символ-плюс, 2006.

Стелтинг С., Маасен О. Java. Применение шаблонов Java. М.: Вильямс, 2002.

СОДЕРЖАНИЕ