Учреждение образования
«Белорусский государственный университет культуры и искусств»

Факультет культурологии и социально-культурной деятельности
Кафедра информационных технологий в культуре

СОГЛАСОВАНО	СОГЛАСОВАНО
Заведующий кафедрой	Декан факультета
_______________ Т.С. Жилинская	_______________ Н.Е. Шелупенко
«___»__________ 2021 г.	«___»______________ 2021 г.

РЕПОЗИТОРИЙ БГУКИ

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

ЯЗЫКИ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

для специальности 1-21 04 01 Культурология (по направлениям)
направления специальности 1-21 04 01-02 Культурология (прикладная)
специализации 1-21 04 01-02 04 Информационные системы в культуре

Составители:
Т.И. Песецкая, доцент кафедры информационных технологий в культуре, кандидат физико-математических наук
В.С. Якимович, доцент кафедры информационных технологий в культуре, кандидат педагогических наук

Рассмотрено и утверждено
на заседании Совета университета 23.02.2021 г.
протокол № 5

Минск 2021

СОСТАВИТЕЛИ:

Т. И. Песецкая, доцент кафедры информационных технологий в культуре учреждения образования «Белорусский государственный университет культуры и искусств», кандидат физико-математических наук

В.С. Якимович, доцент кафедры информационных технологий в культуре учреждения образования «Белорусский государственный университет культуры и искусств», кандидат педагогических наук

РЕЦЕНЗЕНТЫ:

А.Г. Буравкин, Заместитель генерального директора по научной работе ОИПИ НАН Беларуси, кандидат технических наук, доцент;

В.В. Казаченок, профессор кафедры компьютерных технологий и систем ФПМИ БГУ, доктор педагогических наук, профессор.

Рассмотрен и рекомендован к утверждению:

Кафедрой информационные технологии в культуре

(протокол от 28.10.2021 г. № 3)

Советом факультета культурологии и социально-культурной деятельности

(протокол от __.__.2021 г. № __)

РЕПОЗИТОРИЙ БГУКИ

СОДЕРЖАНИЕ

I. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 4
II. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 7
Тема 1. Языки программирования. Классификация языков программирования 7
Тема 2. Системы программирования 18
Тема 3. Объектно-ориентированное программирование 24
Тема 4. Системы объектно-ориентированного программирования 29
Тема 5. Визуальное программирование 33
Тема 6. Программирование мобильных приложений с помощью визуальной среды 41
Тема 7. Основы языка объектно-ориентированного программирования 46
II ПРАКТИЧЕСКИЙ РАЗДЕЛ 84
3.1 Описание лабораторных работ 84
3.2 Тематика семинарских занятий 97
3.3 Тематика и перечень заданий для самостоятельной работы студентов 100
IV. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ 101
V. ВСПОМОГАЬЕЛЬНЫЙ РАЗДЕЛ 103
5.1 Учебно-методическая карта 103
5.2 Литература 104

РЕПОЗИТОРИЙ БГУКИ

I. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

В соответствии с учебными планами дисциплина «Языки и системы программирования» предназначена для изучения студентами высших учебных заведений по специальности 1-2104 01 Культурология (по направлениям) направления специальности 1-21 04 01-02 Культурология (прикладная) специализации 1-21 04 01-02 04 Информационные системы в культуре.

Цель изучения дисциплины «Языки и системы программирования» — формирование теоретических знаний, умений и навыков в области современных языков и систем программирования, а так же построения и разработки прикладных программ для сферы культуры и искусства.

Изучение дисциплины «Языки и системы программирования» даст возможность формировать у студентов базовые профессиональные компетенции в области проектирования и разработки программного обеспечения, знакомя их с основными парадигмами объектно-ориентированного и визуального программирования. Позволит студентам не только усвоить основные понятия и конструкции современных языков программирования, но и изучить на их основе технологию разработки программ, освоить основные типы данных и простейшие алгоритмы, научиться применять теоретические знания при разработке прикладных программ для сферы культуры и искусства.

Учебная дисциплина «Языки и системы программирования» относится к вариативной части профессионального цикла дисциплин. Она связана с дисциплинами: «Прикладная математика», «Алгоритмы обработки данных», «Системный анализ и моделирование информационных процессов». Знания, полученные по дисциплине, являются основой для дальнейшего более углубленного изучения вопросов применения языков и систем программирования в профессиональной деятельности, а также для подготовки курсовых и дипломных работ.

Основными *задачами* дисциплины являются:

– знакомство с основными теоретическими сведениями в области современных языков и систем программирования;

– изучение технологий разработки программ в среде визуального программирования;

– изучение объектно-ориентированного языка программирования для создания приложений в операционной системе Windows;

РЕПОЗИТОРИЙ БГУКИ

– приобретение умений разрабатывать программные продукты, позволяющие продвигать креативные проекты в сфере культуры и искусства.

В результате изучения дисциплины студенты должны *знать*:

– классификацию и основы современных языков программирования;

– среды и инструментальные средства языков визуального и объектно-ориентированного программирования для написания программ;

– интерфейс и синтаксис сред визуального и объектно-ориентированного программирования;

– принципы построения программы в средах визуального и объектно-ориентированного программирования.

Студенты должны *уметь*:

– использовать современные подходы к проектированию и созданию программ, учитывая основные принципы структурного программирования;

– грамотно использовать основные типы данных, функции и классы стандартной библиотеки, компоненты среды программирования;

– разрабатывать программные продукты для мобильных приложений в среде визуального программирования, позволяющие продвигать креативные проекты в сфере культуры и искусства;

– разрабатывать приложения в операционной системе Windows с помощью объектно-ориентированного языка программирования для обеспечения потребностей сферы культуры.

В результате изучения учебной дисциплины «Языки и системы программирования» студент должен *владеть* методами и приемами работы с современными языками объектно-ориентированного и визуального программирования.

Выбор среды, в которой будет проходить процесс обучения объектно-ориентированного и визуального программирования предоставляется преподавателю, который проводит изучение дисциплины на основе представленной программы. При выборе среды преподаватель должен учитывать:

– уровень минимальной подготовки студентов в области программирования;

– современные тенденции в области разработки программных продуктов;

– доступность исходного кода операционной системы, для которой предполагается создавать программные приложения;

– качество написанной документации для создания четких руководств, позволяющих организовать процесс обучения современным языкам объектно-ориентированного и визуального программирования;

РЕПОЗИТОРИЙ БГУКИ

– наличие единого стека технологий для всех проектов (для любого нового приложения стек технологий будет один и тот же), тем самым приводя к тому, что для того чтобы создавать мобильные приложения студенту необходимо освоить один язык, официальную среду разработки и минимум инструментов;

– время появления и стоимость размещения созданного мобильного приложения в магазине.

Освоение данной учебной дисциплины обеспечивает формирование следующих компетенций:

АК-1. Уметь использовать, базовые научно-теоретические знания для решения теоретических и практических задач.

АК-2. Владеть системным и сравнительным анализом.

АК-3. Уметь работать самостоятельно.

АК-7. Иметь навыки, связанные с использованием технических устройств, управлением информацией и работай с компьютерам.

АК-10. Владеть методическими знаниями и исследовательскими умениями, которые обеспечиваются решением задач инновационно-методической и научно-исследовательской деятельностью в области культурологии.

САК-5. Быть способным к критике и самокритике.

САК-6. Уметь работать в команде.

ПК-3. Реализовывать общегосударственные, региональные и ведомственные программы и проекты в области культуры и искусства.

ПК-8. Анализировать и оценивать собранные действия.

ПК-19. Разрабатывать социально-культурные проекты в коммерческой, финансово-хозяйственной деятельности.

Согласно учебным планам на изучение учебной дисциплины «Языки и системы программирования» отведено 46 часов аудиторных занятий, из них лекции — 16 часов, практические и семинарские занятия — 12 часов, лабораторные занятия —18 часов.

Дисциплина рассчитана на один семестр. Текущий контроль осуществляется при выполнении и сдаче лабораторных работ. Форма контроля – зачет.

РЕПОЗИТОРИЙ БГУКИ

II. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Тема 1. Языки программирования. Классификация языков программирования

Лекция 1 (2 часа)

Цель: Изучить спецификации языков программирования (алфавит, синтаксис, семантика) и их классификацию.

Основные вопросы:

Этапы развития языков программирования.

Спецификация языков программирования: алфавит, синтаксис, семантика.

Классификация языков программирования.

Языки низкого и высокого уровня.

Объектно-ориентированные языки.

Декларативные языки программирования.

Функциональные языки или языками искусственного интеллекта.

Языки сценариев или скрипты.

Языки, ориентированные на данные.

Этапы развития языков программирования. Спецификация языков программирования: алфавит, синтаксис, семантика. Классификация языков программирования. Языки низкого и высокого уровня. Объектно-ориентированные языки. Декларативные языки программирования. Функциональные языки или языками искусственного интеллекта. Языки сценариев или скрипты. Языки, ориентированные на данные.

Язык программирования – формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под ее управлением.

Со времени создания первых программируемых машин человечество придумало более двух с половиной тысяч языков программирования. Каждый год их число пополняется новыми. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие

РЕПОЗИТОРИЙ БГУКИ

становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования.

Алфавит − совокупность символов, отображаемых на устройствах печати и экранах и/или вводимых с клавиатуры терминала. Обычно это набор символов Latin-1 с исключением управляющих символов. Иногда в это множество включаются неотображаемые символы с указанием правил их записи (комбинирование в лексемы).

Лексика − совокупность правил образования цепочек символов (лексем), образующих идентификаторы (переменные и метки), операторы, операции и другие лексические компоненты языка. Сюда же включаются зарезервированные (ключевые) слова ЯП, предназначенные для обозначения операторов, встроенных функций и пр.

Иногда эквивалентные лексемы, в зависимости от ЯП, могут обозначаться как одним символом алфавита, так и несколькими. Например, операция присваивания значения в ЯП Basic обозначается как «=», а в языке Pascal − «:=». Операторные скобки в Си задаются символами «{» и «}», а в Pascal − **Begin** и **End.** Граница между лексикой и алфавитом, таким образом, является весьма условной, тем более что компилятор обычно на фазе лексического анализа заменяет распознанные ключевые слова внутренним кодом (например, **Begin** − 512, **End** − 513) и в дальнейшем рассматривает их как отдельные символы.

Синтаксис − совокупность правил образования языковых конструкций или предложений ЯП − блоков, процедур, составных операторов, условных операторов, операторов цикла и пр. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения конструкций. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла;

Семантика − смысловое содержание конструкций, предложений языка. Семантический анализ − это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорить о допустимости операции с данной переменной. Семантические ошибки возникают при недопустимом использовании операций, массивов, функций, операторов и пр.

Стандартизация языков программирования

РЕПОЗИТОРИЙ БГУКИ

Язык программирования может быть представлен в виде набора спецификаций, определяющих его синтаксис и семантику. Для многих широко распространѐнных языков программирования созданы международные стандарты. Специальные организации проводят регулярное обновление и публикацию спецификаций и формальных определений соответствующего языка. В рамках таких комитетов продолжается разработка и модернизация языков программирования и решаются вопросы о расширении или поддержке уже существующих и новых языковых конструкций.

Типы данных

Современные цифровые компьютеры обычно являются двоичными и данные хранят в двоичном (бинарном) коде (хотя возможны реализации и в других системах счисления). Эти данные как правило отражают информацию из реального мира (имена, банковские счета, измерения и др.), представляющую высокоуровневые концепции. Особая система, по которой данные организуются в программе, – это система типов языка программирования; разработка и изучение систем типов известна под названием теория типов.

Языки могут быть *классифицированы* как системы со *статической типизацией и языки с динамической типизацией*. Статически-типизированные языки могут быть в дальнейшем подразделены на языки *с обязательной декларацией*, где каждая переменная и объявление функции имеет обязательное объявление типа, и *языки с выводимыми* типами. Иногда *динамически-типизированные языки* называются латентно-типизированными.

Структуры данных

Системы типов в языках высокого уровня позволяют определять сложные, составные типы, так называемые структуры данных. Как правило, структурные типы данных образуются как декартово произведение базовых (атомарных) типов и ранее определѐнных составных типов.

Основные структуры данных (списки, очереди, хеш-таблицы, двоичные деревья и пары) часто представлены особыми синтаксическими конструкциями в языках высокого уровня. Такие данные структурируются автоматически.

Парадигма программирования

Язык программирования строится в соответствии с той или иной базовой моделью вычислений и парадигмой программирования. Несмотря на то, что большинство языков ориентировано на *императивную модель вычислений, задаваемую фон-неймановской архитектурой ЭВМ*, существуют и другие подходы.

РЕПОЗИТОРИЙ БГУКИ

Можно упомянуть языки со *стековой вычислительной моделью* (Forth, Factor, Postscript и др.), а также *функциональное* (Лисп, Haskell, ML и др.) и *логическое программирование* (Пролог) и язык Рефал, основанный на модели вычислений, введѐнной советским математиком А. А. Марковыммладшим.

В настоящее время также активно развиваются проблемно-ориентированные, декларативные и визуальные языки программирования.

Способы реализации языков

Языки программирования могут быть реализованы как *компилируемые и интерпретируемые*. Программа на компилируемом языке при помощи специальной программы *компилятора* преобразуется (компилируется) в набор инструкций для данного типа процессора (машинный код) и далее записывается в исполнимый модуль, который может быть запущен на выполнение как отдельная программа. Другими словами, компилятор переводит исходный текст программы с языка программирования высокого уровня в двоичные коды инструкций процессора.

Если программа написана на *интерпретируемом* языке, то интерпретатор непосредственно выполняет (интерпретирует) исходный текст без предварительного перевода. При этом программа остаѐтся на исходном языке и не может быть запущена без интерпретатора. Можно сказать, что процессор компьютера − это *интерпретатор* машинного кода.

Кратко говоря, компилятор переводит исходный текст программы на машинный язык сразу и целиком, создавая при этом отдельную машинно-исполняемую программу, а интерпретатор выполняет исходный текст прямо во время исполнения программы («интерпретируя» его своими средствами).

Разделение на компилируемые и интерпретируемые языки является условным.

Кроме того, большинство современных «чистых» интерпретаторов не исполняют конструкции языка непосредственно, а компилируют их в некоторое высокоуровневое промежуточное представление (например, с разыменованием переменных и раскрытием макросов).Для любого интерпретируемого языка можно создать компилятор − например, язык Лисп, изначально интерпретируемый, может компилироваться без каких бы то ни было ограничений. Создаваемый во время исполнения программы код может так же динамически компилироваться во время исполнения.

Как правило, скомпилированные программы выполняются быстрее и не требуют для выполнения дополнительных программ, так как уже переведены на машинный язык. Вместе с тем, при каждом изменении текста программы требуется еѐ перекомпиляция, что создаѐт трудности при разработке. Кроме того, скомпилированная программа может выполняться только на том же типе компьютеров и, как правило, под той же операционной

РЕПОЗИТОРИЙ БГУКИ

системой, на которую был рассчитан компилятор. Чтобы создать исполняемый файл для машины другого типа, требуется новая компиляция.

Интерпретируемые языки обладают некоторыми специфическими дополнительными возможностями (см. выше), кроме того, программы на них можно запускать сразу же после изменения, что облегчает разработку. Программа на интерпретируемом языке может быть зачастую запущена на разных типах машин и операционных систем без дополнительных усилий. Однако интерпретируемые программы выполняются заметно медленнее, чем компилируемые, кроме того, они не могут выполняться без дополнительной программы-интерпретатора.

Некоторые языки, например, Java и C#, находятся между компилируемыми и интерпретируемыми. А именно, программа компилируется не в машинный язык, а в машинно независимый код низкого уровня, байт-код. Далее байт-код выполняется виртуальной машиной. Для выполнения байт-кода обычно используется интерпретация, хотя отдельные его части для ускорения работы программы могут быть транслированы в машинный код непосредственно во время выполнения программы по технологии компиляции «на лету» (Just-in-time compilation, JIT).

Для Java байт-код исполняется виртуальной машиной Java (Java Virtual Machine, JVM), для C# − Common Language Runtime. Подобный подход в некотором смысле позволяет использовать плюсы как интерпретаторов, так и компиляторов. Следует упомянуть также язык Форт(Forth), имеющий и интерпретатор, и компилятор.

Классификация языков и подходов к программированию.

Первые языки программирования возникли относительно недавно. Различные исследователи указывают в качестве времени их создания 20-е, 30-е и даже 40-е годы XX столетия. Как и следовало ожидать, первые языки программирования, как и первые ЭВМ, были довольно примитивны и ориентированы на численные расчеты. Это были и чисто теоретические научные расчеты (прежде всего, математические и физические), и прикладные задачи, в частности, в области военного дела.

Программы, написанные на ранних языках программирования, представляли собой линейные последовательности элементарных операций с регистрами, в которых хранились данные. Нужно отметить, что ранние языки программирования были оптимизированы под аппаратную архитектуру конкретного компьютера, для которого предназначались, и хотя они обеспечивали высокую эффективность вычислений, до стандартизации было еще далеко. Программа, которая была вполне работоспособной на одной вычислительной машине, зачастую не могла выполняться на другой.

РЕПОЗИТОРИЙ БГУКИ

Таким образом, ранние языки программирования существенно зависели от того, что принято называть средой вычислений и приблизительно соответствовали современным машинным кодам или языкам ассемблера.

Следующее десятилетие ознаменовалось появлением языков программирования так называемого "высокого уровня", по сравнению с ранее рассмотренными предшественниками, соответственно именуемыми низкоуровневыми языками. При этом различие состоит в повышении эффективности труда разработчиков за счет абстрагирования от конкретных деталей аппаратного обеспечения. Одна инструкция (оператор) языка высокого уровня соответствовала последовательности из нескольких низкоуровневых инструкций, или команд. Исходя из того, что программа, по сути, представляла собой набор директив, обращенных к компьютеру, такой подход к программированию получил название императивного.

Еще одной особенностью языков высокого уровня была возможность повторного использования ранее написанных программных блоков, выполняющих те или иные действия, посредством их идентификации и последующего обращения к ним, например по имени. Такие блоки получили название функций или процедур, и программирование приобрело более упорядоченный характер.

Кроме того, с появлением языков высокого уровня зависимость реализации от аппаратного обеспечения существенно уменьшилась. Платой за это стало появление специализированных программ, преобразующих инструкции языков в коды той или иной машины, или трансляторов, а также некоторая потеря в скорости вычислений, которая, впрочем, компенсировалась существенным выигрышем в скорости разработки приложений и унификацией программного кода.

Нужно отметить, что операторы и ключевые слова новых языков программирования были более осмысленными, чем безликие цифровые последовательности кодов, что также обеспечивало повышение производительности труда программистов.

В 60-х годах возникает новый подход к программированию, который до сих пор успешно конкурирует с императивным, а именно, *декларативный* подход.

Суть подхода состоит в том, что программа представляет собой не набор команд, а описание действий, которые необходимо осуществить. Этот подход существенно проще и прозрачнее формализуется математическими средствами. Следовательно, программы проще проверять на наличие ошибок (тестировать), а также на соответствие заданной технической спецификации (верифицировать).

РЕПОЗИТОРИЙ БГУКИ

Высокая степень абстракции также является преимуществом данного подхода. Фактически, программист оперирует не набором инструкций, а абстрактными понятиями, которые могут быть достаточно обобщенными.

На начальном этапе развития декларативным языкам программирования было сложно конкурировать с императивными в силу объективных трудностей эффективной реализации трансляторов. Программы работали медленнее, однако они могли решать более абстрактные задачи с меньшими трудозатратами. В частности, язык SML был разработан как средство доказательства теорем.

Различные диалекты языка LISP (в частности, Interlisp, Common Lisp, Scheme), возникли потому, что ядро и идеология этого языка оказались весьма эффективными при реализации символьной обработки (анализе текстов). Другие характерные примеры декларативных языков программирования: SML, Haskell, Prolog.

Одним из путей развития декларативного стиля программирования стал *функциональный* подход, возникший после создания языка LISP. Отличительной особенностью данного подхода является то, что любая программа, написанная на таком языке, может интерпретироваться как функция с одним или несколькими аргументами. Такой подход дает возможность прозрачного моделирования текста программ математическими средствами, а значит, весьма интересен с теоретической точки зрения.

Сложные программы при таком подходе строятся посредством агрегирования функций. При этом текст программы представляет собой функцию, некоторые аргументы которой можно также рассматривать как функции. Таким образом, повторное использование кода сводится к вызову ранее описанной функции, структура которой, в отличие от процедуры императивного языка, математически прозрачна.

Более того, типы отдельных функций, используемых в функциональных языках, могут быть переменными. Таким образом обеспечивается возможность обработки разнородных данных (например, упорядочение элементов списка по возрастанию для целых чисел, отдельных символов и строк) или полиморфизм.

Еще одним важным преимуществом реализации языков *функционального* программирования является автоматизированное динамическое распределение памяти компьютера для хранения данных. При этом программист избавляется от обязанности контролировать данные, а при необходимости может запустить функцию "сборки мусора" – очистки памяти от тех данных, которые больше не потребуются программе (обычно этот процесс периодически инициируется компьютером).

РЕПОЗИТОРИЙ БГУКИ

Таким образом, при создании программ на функциональных языках программист сосредотачивается на области исследований (предметной области) и в меньшей степени заботится о рутинных операциях (обеспечении правильного с точки зрения компьютера представления данных, "сборке мусора" и т.д.).

Поскольку функция является естественным формализмом для языков функционального программирования, реализация различных аспектов программирования, связанных с функциями, существенно упрощается. В частности, становится прозрачным написание рекурсивных функций, т.е. функций, вызывающих самих себя в качестве аргумента. Кроме того, естественной становится и реализация обработки рекурсивных структур данных (например, списков – базовых элементов, скажем, для языков семейства LISP, деревьев и др.)

Благодаря реализации механизма сопоставления с образцом, такие языки как ML и Haskell вполне применимы для символьной обработки. Естественно, языки функционального программирования не лишены недостатков. Часто к ним относят нелинейную структуру программы и относительно невысокую эффективность реализации. Однако первый недостаток достаточно субъективен, а второй успешно преодолен современными реализациями, в частности, рядом последних трансляторов языка SML, включая и компилятор для среды Microsoft .NET.

В 70-х годах возникла еще одна ветвь языков декларативного программирования, связанная с проектами в области *искусственного интеллекта*, а именно языки логического программирования. Согласно логическому подходу к программированию, программа представляет собой *совокупность правил или логических высказываний*. Кроме того, в программе допустимы логические причинно-следственные связи, в частности, на основе операции импликации.

Таким образом, языки *логического* программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых экспертных систем.

На языках логического программирования естественно формализуется логика поведения, и они применимы для описаний правил принятия решений, например, в системах, ориентированных на поддержку бизнеса.

Важным преимуществом такого подхода является достаточно высокий уровень машинной независимости, а также возможность откатов – возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения (скажем, очередного хода при игре в шахматы), что избавляет от необходимости поиска решения путем полного перебора вариантов и увеличивает эффективность реализации.

РЕПОЗИТОРИЙ БГУКИ

Одним из недостатков логического подхода в концептуальном плане является специфичность класса решаемых задач. Другой недостаток практического характера состоит в сложности эффективной реализации для

принятия решений в реальном времени, скажем, для систем жизнеобеспечения. Нелинейность структуры программы является особенностью декларативного подхода и, строго говоря, представляет собой оригинальную особенность, а не объективный недостаток. В качестве примеров языков логического программирования можно привести Prolog (название возникло от слов PROgramming in LOGic) и Mercury.

Важным шагом на пути к совершенствованию языков программирования стало появление *объектно-ориентированного* подхода к программированию (ООП) и соответствующего класса языков.

В рамках данного подхода программа представляет собой описание объектов, их свойств (или атрибутов), совокупностей (или классов), отношений между ними, способов их взаимодействия и операций над объектами (или методов).

Несомненным преимуществом данного подхода является концептуальная близость к предметной области произвольной структуры и назначения. Механизм наследования атрибутов и методов позволяет строить производные понятия на основе базовых и таким образом создавать модель сколь угодно сложной предметной области с заданными свойствами.

Еще одним теоретически интересным и практически важным свойством объектноориентированного подхода является поддержка механизма обработки событий, которые изменяют атрибуты объектов и моделируют их взаимодействие в предметной области.

Перемещаясь по иерархии классов от более общих понятий предметной области к более конкретным (или от более сложных – к более простым) и наоборот, программист получает возможность изменять степень абстрактности или конкретности взгляда на моделируемый им реальный мир.

Использование ранее разработанных (возможно, другими коллективами программистов) библиотек объектов и методов позволяет значительно сэкономить трудозатраты при производстве программного обеспечения, в особенности типичного.

Объекты, классы и методы могут быть полиморфными, что делает реализованное программное обеспечение более гибким и универсальным. Сложность адекватной (непротиворечивой и полной) формализации объектной теории порождает трудности тестирования и верификации созданного программного обеспечения. Вероятно, это обстоятельство является одним из самых существенных недостатков объектно-ориентированного подхода к программированию.

РЕПОЗИТОРИЙ БГУКИ

Пожалуй, наиболее известным примером объектно-ориентированного языка программирования является язык С++, развившийся из императивного языка С. Наиболее востребоваными объектноориентированными языками программирования, помимо С++, являются на сегодняшний момент Java и C#. Другие примеры объектно-ориентированных языков программирования: Visual Basic, Eiffel, Oberon.

Развитием событийно управляемой концепции объектно-ориентированного подхода стало появление в 90-х годах целого класса языков программирования, которые получили название *языков сценариев или скриптов*. В рамках данного подхода программа представляет собой совокупность возможных сценариев обработки данных, выбор которых инициируется наступлением того или иного события (щелчок по кнопке мыши, попадание курсора в определенную позицию, изменение атрибутов того или иного объекта, переполнение буфера памяти и т.д.). События могут инициироваться как операционной системой, так и пользователем.

Основные достоинства языков данного класса унаследованы от объектно-ориентированных языков программирования. Это интуитивная ясность описаний, близость к предметной области, высокая степень абстракции, хорошая переносимость. Широкие возможности повторного использования кода также унаследованы сценарными языками от объектно-ориентированных предков.

Существенным преимуществом языков сценариев является их совместимость с передовыми инструментальными средствами автоматизированного проектирования и быстрой реализации программного обеспечения, или так называемыми CASE- (Computer-Aided Software Engineering) и RAD- (Rapid Application Development) средствами. Естественно, что вместе с достоинствами объектно-ориентированного подхода языки сценариев унаследовали и ряд недостатков. К последним, прежде всего, относятся сложность тестирования и верификации программ и возможности возникновения в ходе эксплуатации множественных побочных эффектов, проявляющихся за счет сложной природы взаимодействия объектов и среды, представленной интерфейсами с большим количеством одновременно работающих пользователей программного обеспечения, операционной системой и внешними источниками данных.

Характерные примеры сценарных языков программирования: VBScript, PowerScript, LotusScript, JavaScript.

Еще один весьма важный класс языков программирования – *языки поддержки параллельных вычислений*.

Программы, написанные на этих языках, представляют собой совокупность описаний процессов, которые могут выполняться как в

РЕПОЗИТОРИЙ БГУКИ

действительности одновременно, так и в псевдопараллельном режиме. В последнем случае устройство, обрабатывающее процессы, функционирует в режиме разделения времени, выделяя время на обработку данных, поступающих от процессов, по мере необходимости (а также с учетом последовательности или приоритетности выполнения операций).

Языки параллельных вычислений позволяют достичь заметного выигрыша при обработке больших массивов информации, поступающих от одновременно работающих пользователей, либо имеющих высокую интенсивность (как, например, видеоинформация или звуковые данные высокого качества).

Другой весьма значимой областью применения языков параллельных вычислений являются системы реального времени, в которых пользователю необходимо получить ответ от системы непосредственно после запроса. Такого рода системы отвечают за жизнеобеспечение и принятие ответственных решений. Обратной стороной достоинств рассматриваемого класса языков программирования является высокая стоимость разработки программного обеспечения, следовательно, создание относительно небольших программ широкого (например, бытового) применения зачастую нерентабельно.

Примерами языков программирования с поддержкой параллельных вычислений могут служить Ada, Modula-2 и Oz. Итак, в данной лекции были рассмотрены история и эволюция языков программирования и основные подходы к разработке программных систем. Была сделана попытка классификации языков и подходов к программированию, а также проведен

РЕПОЗИТОРИЙ БГУКИ

анализ достоинств и недостатков, присущих тем или иным подходам и языкам.

Заметим, что приведенную классификацию не следует считать единственно верной, поскольку языки программирования постоянно развиваются и совершенствуются, и недавние недостатки устраняются с появлением необходимых инструментальных средств и теоретических обоснований.

Тема 2. Системы программирования

Лекция 2 (2 часа)

Цель: Изучить основные составляющие системы программирования.

Основные вопросы:

Основные понятия: редактор текста, транслятор, компоновщик (редактор связей), отладчик, библиотеки подпрограмм.

Инструментальные средства разработки программ.

Среды разработки.

Компилятор или интерпретатор.

Библиотеки стандартных программ и функций.

Отладочные программы.

Графические библиотеки.

Система программирования - это комплекс программных средств, предназначенных для кодирования, тестирования и отладки программного обеспечения. Имеет интерфейс, удобный пользователю.

Такие комплексы, как правило, включают следующие программные модули.

1. *Текстовые редакторы*, служащие для создания текстов исходных программ.

2. *Компиляторы*, предназначенные для перевода исходного текста на входном языке в язык машинных кодов. В результате создаются объектные модули. Это программы на машинном языке, они записываются на диск с расширением .exe.

РЕПОЗИТОРИЙ БГУКИ

3. *Компоновщики*, позволяющие объединять несколько объектных модулей, порождаемых компилятором, в одну программу. В результате создается загрузочный модуль.

4. *Загрузчики*, обеспечивающие подготовку готовой программы к выполнению.

5. *Библиотеки прикладных программ*, содержащие в себе наиболее часто используемые подпрограммы в виде готовых объектных модулей.

6. *Отладчики*, выполняющие программу в заданном режиме (например, пошаговом) с целью поиска, обнаружения и локализации ошибок. Используются на этапе компиляции.

Основным модулем системы программирования всегда является компилятор. Именно технические характеристики компилятора, прежде всего, влияют на эффективность результирующих программ, порождаемых системой программирования.

Транслятор – это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на результирующем (выходном) языке.

Близко по смыслу к этому понятию понятие компилятор.

Компилятор – это транслятор, который осуществляет перевод исходной программы в эквивалентную ей объектную программу на языке машинных команд или языке ассемблера (.exe файл). Таким образом, компилятор отличается от транслятора тем, что его результирующая программа написана обязательно на языке машинных команд или языке ассемблера. Результирующая программа транслятора в общем случае может быть написана на любом языке (например, транслятор с языка Pascal на язык C). Таким образом, компиляторы – это вид трансляторов (*см. Лекция 1*).

Существует принципиально отличное понятие: интерпретатор.

Интерпретатор – это программа, которая воспринимает входную программу на исходном языке, переводит каждый оператор или строку в машинный язык и выполняет их. Интерпретатор не порождает результирующую программу и никакого результирующего кода.

Интерпретаторы удобны для быстрой отладки программ, тем самым укорачивая обычный цикл разработки. Однако с другой стороны, интерпретаторы в сравнении с компиляторами обычно проигрывают по скорости выполнения в несколько раз (*см. Лекция 1*).

Термин *отладка* может иметь разные значения, но в первую очередь он означает устранение ошибок в коде. Делается это по-разному. Например, отладка может выполняться путем проверки кода на наличие опечаток или с помощью анализатора кода. Код можно отлаживать с помощью

РЕПОЗИТОРИЙ БГУКИ

профилировщика производительности. Кроме того, отладка может производиться посредством отладчика.

Отладчик − это узкоспециализированное средство разработки, которое присоединяется к работающему приложению и позволяет проверять код.

Среды разработки ПО (Программного обеспечения) являются объединением программных средств, которые предназначены для написания (создания) программных продуктов. Среда разработки включает в свое содержание: компилятор, интерпретатор, отладчик, средства автоматизации сборки, а также редактор текста.

Когда в среде разработки ПО присутствуют все вышеназванные компоненты, среду называют интегрированной. Такие среды разработки увеличивают темп, а также удобность разработки за счѐт: автоматизации, возможности производить весь цикл создания и разработки ПО.

Таблица 1. Интегрированные среды разработки ПО

Среда разработки/критерий	Поддержка разработчиком	Является Кроссплатформенной	Поддерживает более пяти языков программирования	Шкала популярности от 1 до 5	Поддерживает русский язык	Включает в свой состав компилятор
Visual studio	✓	✗	✓	5	✓	✓
NetBeans	✓	✓	✓	4,4	✗	✓
Geany	✓	✓	✓	5	✓	✗
Komodo	✓	✓	✓	4	✗	✓
Kylix	✗	✗	✗	2	✗	✓
Eclipse	✓	✓	✓	5	✓	✓

Обычная среда разработки ПО предназначена для разработки только на одном языке программирования. Интегрированная среда разработки предоставляет право выбрать язык программирования для разработки, удобный разработчику (из языков поддерживаемых данной средой). Примером интегрированных сред служат: Visual Studio, Komodo, Geany, Kylix, NetBeans, Eclipse (*см. Таблица 1*).

Стандартная библиотека языка программирования − набор модулей, классов, объектов, констант, глобальных переменных, шаблонов, макросов, функций и процедур, доступных для вызова из любой программы, написанной на этом языке и присутствующих во всех реализациях языка.

РЕПОЗИТОРИЙ БГУКИ

В ранних языках программирования (таких как Фортран, Алгол, Кобол, Бейсик) функциональность, не связанная непосредственно с языком, но необходимая для создания большинства реальных программ, включалась непосредственно в язык и нередко реализовывалась непосредственно на уровне транслятора или среды исполнения (для интерпретируемых языков). К числу таких функций относились прежде всего те, которые было невозможно, неудобно или неэффективно реализовывать средствами самого языка программирования: команды ввода-вывода, вычисление стандартных математических и логических функций, специфические операции с некоторыми типами данных, работа с внешними устройствами, с «сырой» памятью, взаимодействие с операционной системой и прочее. Такой подход имел недостатки: компилятор оказывался привязан не только к архитектуре компьютера, которая статична и меняется не так часто, но и к его программной среде, в первую очередь − к операционной системе, которая может меняться гораздо чаще.

В 1970-х годах в процедурных языках широко распространилась концепция модульности. Появились синтаксические средства для описания программы как набора относительно независимых модулей. Естественным следствием стало то, что разработчики языков программирования стали выносить входившие ранее в состав языков универсальные функции во внешние библиотечные модули и подключать эти модули к программам при необходимости. Но исключение стандартных функций из языка могло привести к проблемам совместимости: каждый производитель компилятора мог предоставлять собственные библиотеки. Поэтому вместе со спецификацией языка программирования разработчики начали предлагать спецификацию на стандартную библиотеку − ту часть программного окружения, которая обязана присутствовать в любой реализации данного языка. С этого момента любая реализация того или иного языка программирования разделяется на три части: реализация собственно языка (компилятор), реализация стандартной библиотеки и дополнительные средства (среда разработки, редактор связей, средства документирования, дополнительные библиотеки), которые обычно не стандартизируются вместе с языком, хотя на них могут существовать отдельные стандарты. Теоретически, любой программист, использующий только стандартные языковые средства и стандартную библиотеку, может быть уверен, что его программа будет работать одинаково на любой платформе.

В некоторых случаях стандартная библиотека описывается непосредственно в стандарте языка (Python, Perl, Java), в некоторых − отдельными стандартами (Си), иногда она определяется исходя из практического наличия функций в реализациях языка (Basic, Pascal).

РЕПОЗИТОРИЙ БГУКИ

Существуют два подхода для определения круга задач, которые должна выполнять стандартная библиотека языка программирования.

Первый − стандартная библиотека должна содержать в себе только те процедуры и функции, которые используются практически всеми и обладают максимальной универсальностью. В частности, этой позиции придерживается Бьѐрн Страуструп, автор языка C++. Одним из оснований этой концепции является простое соображение: чем более специфична предоставляемая библиотекой функциональность, тем труднее реализовать еѐ так, чтобы она в полной мере удовлетворяла требованиям каждого конкретного случая; если библиотечная реализация не будет универсальной и эффективной во всех вариантах применения, то для большинства реальных задач она окажется неподходящей, следовательно − бесполезной.

Второй − стандартная библиотека должна содержать в себе максимально возможное количество типичных алгоритмов, обеспечивать простую работу с большинством (в идеале, со всеми) объектами, с которыми может взаимодействовать программа. Одним из примеров реализации этого подхода является язык Python с девизом «Batteries included» (батарейки в комплекте). Доводом в пользу этого подхода является представление, согласно которому скорость написания программ и их корректность в большинстве случаев важнее эффективности (по крайней мере, для большинства прикладных программ), поэтому лучше всего предоставить программисту максимум готовых, тщательно проверенных механизмов, которые он сможет использовать, чтобы программирование «вручную» было необходимо только для действительно нетривиальных алгоритмов; это сэкономит время и убережѐт программиста от технических ошибок в написании того, что уже было реализовано.

Каждая библиотека предоставляет возможности для решения каких-то конкретных задач:

выполнения математических операций;

работы с графикой;

работы с файлами;

работы с сетью;

шифрования и так далее.

Для популярных языков программирования уже написано очень много библиотек. Но чем их больше, тем быстрее появляются новые разработчики и тем быстрее создаются новые библиотеки − этот процесс бесконечен. Библиотеки бывают открытыми (англ. FOS, Free and Open Source − бесплатные и с открытым исходным кодом) и коммерческими. *Открытые библиотеки* создаются сообществом программистов − здесь каждый может предложить исправления, написать новые функции или сообщить об

РЕПОЗИТОРИЙ БГУКИ

ошибках. *Коммерческие библиотеки* разрабатываются компаниями, в которых есть штат программистов и тестировщиков. Также многие пишут собственные библиотеки и используют их в своих проектах. Они могут быть встроены в язык или добавляться отдельно.

По способу подключения к основной программе библиотеки можно разделить на два типа: динамические и статические. *Динамические* − это файл с машинным кодом, который подключается во время исполнения. Его в любой момент можно заменить на другие. В этом одновременно и плюс − динамическую библиотеку можно обновить почти без труда, и минус − требуется ровно столько же усилий, чтобы заменить ее на что-нибудь вредоносное. *Статические* − это исходный код на языке программы или объектный модуль, который упаковывается в саму программу. Такую библиотеку очень сложно подменить, поэтому, чтобы обновить её, придётся заново компилировать всю программу.

В зависимости от возможностей языка, стандартная библиотека может содержать:

процедуры и функции

макросы

глобальные переменные

классы

шаблоны

Обычно стандартная библиотека содержит основные алгоритмы и структуры данных, необходимые для:

работы с динамической памятью;

файловыми операциями ввода-вывода;

операциями ввода-вывода данных на терминал;

конвертацией данных между типами;

функции для работы со структурами данных: строками, массивами, списками и т. п.;

математические операции;

функции для работы с сетью;

функции для обеспечения обработки исключений и ошибок в программе;

функции для поддержки многопоточности.

Бьёрн Страуструп выдвинул следующие 11 принципиальных требований к средствам, предлагаемым стандартной библиотекой:

Востребованность − быть важными доступными для всех пользователей языка.

Используемость − прямо или косвенно использоваться всеми программистами для решения всех задач, связанных с целями библиотеки.

РЕПОЗИТОРИЙ БГУКИ

Эффективность − для большинства применений библиотечная реализация должна быть не хуже реализации тех же механизмов «вручную».

Независимость от алгоритмов − предоставлять возможность задавать алгоритмы в качестве параметров. Например, стандартная функция сортировки должна давать возможность определить любой алгоритм сравнения элементов.

Математическая примитивность − каждая компонента должна предоставлять одну определѐнную функцию либо группу функций, используемых только совместно.

Удобство, эффективность и безопасность − по крайней мере, в большинстве типичных вариантов применения.

Завершѐнность − «Стандартная библиотека может оставить множество функций другим библиотекам, но если уж она взялась за какую-то задачу, то должна обеспечить достаточную функциональность, чтобы отдельным пользователям и разработчикам не приходилось заменять еѐ средства».

Органично сочетаться с языком − служить естественным продолжением языковых средств, типов, операций.

Типобезопасность − безопасность с точки зрения типов по умолчанию.

Поддержка общепринятых стилей программирования.

Расширяемость − способность единообразно работать со встроенными типами данных и с типами, определяемыми пользователем.

Графическая библиотека − это совокупность программ, предназначенная для рендеринга компьютерной графики и визуализации на мониторе. Обычно это предполагает предоставление оптимизированных версий функций, которые обрабатывают общие задачи рендеринга. Используя эти функции, программа может собрать изображение для вывода на монитор. Это освобождает программиста от задачи создания и оптимизации этих функций и позволяет им сосредоточиться на создании графической программы. Графические библиотеки в основном используются в видеоиграх и симуляциях.

Тема 3. Объектно-ориентированное программирование

Лекция 3 (2 часа)

Цель: Изучить основные понятие объектных технологий.

Основные вопросы:

Классы и объекты.

РЕПОЗИТОРИЙ БГУКИ

Особенности работы с объектами. М
Модификаторы доступа.
Инкапсуляция. Наследование.
Полиморфизм и перегрузка методов.
Абстрактные классы.
Интерфейсы и особенности их создания.

Объектно-ориентированное программирование (ООП) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Объект – это нечто, имеющее четко определенные границы. Однако, этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Класс – это множество объектов, обладающих общей структурой, поведением и семантикой. Отдельный объект – это экземпляр класса. Класс представляет лишь абстракцию существенных свойств объекта.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. Например: торговый автомат имеет свойство: способность принимать монеты; этому свойству соответствует динамическое значение – количество принятых монет. Пример описания состояния объекта:

```
struct PersonnelRecord {
	char name[100];
	int socialSecurityNumber;
	char department[10];
	float salary;
	};
```

Поведение объекта – это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений. Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append() и pop() для того, чтобы управлять объектом очередью:

РЕПОЗИТОРИЙ БГУКИ

```
class Queue {
public:
Queue();
Queue(const Queue&);
virtual ~Queue();
virtual Queue& operator=(const Queue&);
virtual int operator==(const Queue&) const;
int operator!=(const Queue&) const;
virtual void clear();
virtual void append(const void*);
virtual void remove(int at);
virtual int length() const;
virtual int isEmpty() const;
...
};
```

Индивидуальность объекта – это такое свойство объекта, которое отличает его от всех других объектов. В большинстве языков программирования при создании объект именуется, поэтому многие путают адресуемость и индивидуальность. Невозможность отличить имя объекта от самого объекта является источником множества ошибок в ООП.

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации. Пусть члену класса требуется защита от «несанкционированного доступа». Как разумно ограничить множество функций, которым такой член будет доступен? Очевидный ответ для языков, поддерживающих объектно-ориентированное программирование, таков: доступ имеют все операции, которые определены для этого объекта, иными словами, все функции члены. Например:

```
class window
{
 // ...
protected:
 Rectangle inside;
 // ...
};
class dumb_terminal: public window
{
```

РЕПОЗИТОРИЙ БГУКИ

```
 // ...
public:
 void prompt();
 // ...
};
```

Здесь в базовом классе window член inside типа Rectangle описывается как защищенный (protected), но функции-члены производных классов, например, dumb_terminal::prompt(), могут обратиться к нему и выяснить, с какого вида окном они работают. Для всех других функций член window::inside недоступен.

В таком подходе сочетается высокая степень защищенности с гибкостью, необходимой для программ, которые создают классы и используют их.

Неочевидное следствие из этого: нельзя составить полный и окончательный список всех функций, которым будет доступен защищенный член, поскольку всегда можно добавить еще одну, определив ее как функцию-член в новом производном классе. Для метода абстракции данных такой подход часто бывает мало приемлемым. Если язык ориентируется на метод абстракции данных, то очевидное для него решение – это требование указывать в описании класса список всех функций, которым нужен доступ к члену. Так например в С++ для этой цели используется описание частных (private) членов.

Важность инкапсуляции, т.е. заключения членов в защитную оболочку, резко возрастает с ростом размеров программы и увеличивающимся разбросом областей приложения.

Наследование представляет собой способность производить новый класс из существующего базового класса. Производный класс – это новый класс, а базовый класс – существующий класс. Когда вы порождаете один класс из другого (базового класса), производный класс наследует элементы базового класса. Для порождения класса из базового начинайте определение производного класса ключевым словом class, за которым следует имя класса, двоеточие и имя базового класса, например class dalmatian: dog.

Когда вы порождаете класс из базового класса, производный класс может обращаться к общим элементам базового класса, как будто эти элементы определены внутри самого производного класса. Для доступа к частным данным базового класса производный класс должен использовать интерфейсные функции базового класса.

Внутри конструктора производного класса ваша программа должна вызвать конструктор базового класса, указывая двоеточие, имя конструктора

РЕПОЗИТОРИЙ БГУКИ

базового класса и соответствующие параметры сразу же после заголовка конструктора производного класса. Чтобы обеспечить производным классам прямой доступ к определенным элементам базового класса, в то же время защищая эти элементы от оставшейся части программы, C++ обеспечивает защищенные (protected) элементы класса. Производный класс может обращаться к защищенным элементам базового класса, как будто они являются общими. Однако для оставшейся части программы защищенные элементы эквивалентны частным.

Если в производном и базовом классе есть элементы с одинаковым именем, то внутри функций производного класса C++ будет использовать элементы производного класса. Если функциям производного класса необходимо обратиться к элементу базового класса, вы должны использовать оператор глобального разрешения, например base class:: member.

Полиморфный объект представляет собой такой объект, который может изменять форму во время выполнения программы. В объектно-ориентированных языках класс является абстрактным типом данных. Полиморфизм реализуется с помощью наследования классов и виртуальных функций. Класс-потомок наследует сигнатуры методов класса-родителя, а реализация, в результате переопределения метода, этих методов может быть другой, соответствующей специфике класса-потомка. Другие функции могут работать с объектом класса-родителя, но при этом вместо него во время исполнения будет подставляться один из классов-потомков.Это называется поздним связыванием.

Класс-потомок сам может быть родителем. Это позволяет строить сложные схемы наследования – древовидные или сетевидные.

Абстрактные (или чисто виртуальные) методы не имеют реализации вообще. Они специально предназначены для наследования. Их реализация должна быть определена в классах-потомках. Класс может наследовать функциональность от нескольких классов. Это называется множественным наследованием. Множественное наследование создаѐт проблему (когда класс наследуется от нескольких классов-посредников, которые в свою очередь наследуются от одного класса (так называемая «Проблема ромба»): если метод общего предка был переопределѐн в посредниках, неизвестно, какую реализацию метода должен наследовать общий потомок. Решается эта проблема через виртуальное наследование.

Интерфейс класса будем понимать как именованный набор свойств и сигнатур методов, без их конкретной реализации. Сама реализация будет конкретизирована в каждом классе наследнике интерфейса индивидуально. Например, в C# от одного родительского класса может быть образовано несколько классов наследников, но не наоборот. Нельзя организовать

РЕПОЗИТОРИЙ БГУКИ

иерархию классов так, чтобы у одного класса наследника было несколько родительских классов. Для решения этой проблемы можно использовать технологию интерфейсов, которая позволяет классу наследовать от нескольких интерфейсов. Кроме того предварительное описание интерфейсов обязывает реализовать все заявленные в них методы, приводит к унификации классов наследников и не позволяет компилировать библиотеку классов с не полностью описанным классом наследником интерфейса. Наследование интерфейсов разными классами очень похоже на полиморфизм методов в классах, но с одним важным дополнением. При полиморфизме объявленный в родительском классе виртуальный метод может получить частную реализацию в каждом из классов наследников. При наследовании интерфейсов объявленный метод не просто может, но и обязан получить частную реализацию в каждом из классов наследников. Удобство такого подхода состоит еще и в том, что можно объекты разных классов объединять в один список объектов и затем в цикле обращаться к определенному методу интерфейса, который реализован в каждом классе по-своему.

РЕПОЗИТОРИЙ БГУКИ

Тема 4. Системы объектно-ориентированного программирования

Лекция 4 (2 часа)

Цель: Рассмотреть основные системы объектно-ориентированного программирования.

Основные вопросы:

Обзор систем и языков объектно-ориентированного программирования.

Область применения технологии объектно-ориентированного программирования.

Системы быстрой разработки приложений.

Современные системы объектно-ориентированного программирования.

Системы визуального программирования.

Процедурно-ориентированные языки мало подходят для написания программных систем, где центральным место занимают данные, а не алгоритмы. С определенного момента возникла острая необходимость в языковой поддержке описания произвольных объектов окружающего мира.

Вводятся абстрактные типы данных. Как хорошо сказал Шанкар: «Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи».

Первым языком, в котором нашли свое выражение идеи построения программ на основе данных и объектов стал язык Simula 67. Концепции, заложенные в языке Simula получили свое развитие в серии языков Smalltalk-72,-74,-76,-80, а также в языках C++ и Objective C. При внесении объектно-ориентированного подхода в язык Pascal появился язык Object Pascal. В 90-х годах компания Sun представила миру язык Java, как воплощение идеи платформенной независимости и наиболее полную реализацию концепций объектно-ориентированного программирования, положенных в основу языков Simula 67, Smalltalk, C++.

Насколько появление объектных языков и объектной модели явилось прямым развитием модульного подхода при написании программных систем, настолько и появление объектно-ориентированных механизмов явилось следующим шагом в развитии объектной модели.

Проследим концепцию объектно-ориентированных механизмов на примере классификации животного мира. Система эта иерархична. Рассмотрим такое существо как домашняя любимица Мурка. Прежде всего следует отметить, что Мурка является кошкой, причем, скорее всего, кошкой определенной породы, например сиамской. Таким образом, наша Мурка является объектом класса сиамских кошек. Если полностью выписать классификационную принадлежность Мурки, то мы получим, что она принадлежит к сиамской породе домашних кошек, роду кошек, семейству кошачьих, отряду хищных, классу млекопитающих, подтипу позвоночных, типу хордовых, царству животных. Каждая из ступеней определяет ряд характеристик и особенностей, которыми обладает объект Мурка. Тот факт, что данный объект принадлежит к классу сиамских кошек говорит нам о том, что он имеет определенный окрас, длину шерсти и особенности поведения. Кроме того, мы знаем, что при этом наша кошка Мурка является представителем вида домашних кошек, что определяет ее размер, сильно отличающийся от размера, к примеру, тигра. Как представитель семейства кошачьих, Мурка обладает определенными особенностями анатомического строения; будучи хищником, она способна употреблять в пищу других животных; от класса млекопитающих она унаследовала способность выкармливать детенышей молоком и теплокровное строение организма и так далее. В терминах объектно-ориентированного подхода, мы говорим, что

РЕПОЗИТОРИЙ БГУКИ

перечисленные классы образуют иерархию наследования, построенную на основе отношения «обобщение/специализация».

Классы не всегда образуют строго пирамидальную иерархию. Например, если мы определим класс домашних животных (представители которого могут жить в доме человека и взаимодействовать со своими хозяевами), то обнаружим, что домашние животные могут быть как теплокровными, так и хладнокровными. Свойство быть домашним животным не вписывается естественным образом в нашу иерархию живых организмов, а скорее играет роль признака (интерфейса), или «подмешивания». В дальнейшем это понятие будет рассмотрено более подробно.

Сложность окружающей действительности и многообразие объектов реального мира побуждает человека вырабатывать классификационные схемы, для того чтобы мыслить и оперировать объектами. Поэтому, совершенно естественным стало внедрение поддержки объектного подхода в языках программирования.

Наличие иерархических отношений наследования между абстракциями (классами и интерфейсами), экземплярами которых являются объекты в программе, является отличительной чертой объектно-ориентированного подхода в программировании.

Программа написанная на объектно-ориентированном языке представляет собой совокупность объектов, каждый из которых принадлежит к определенному абстрактному типу данных (классу), а классы образуют иерархию наследования.

Процесс программирования состоит в последовательной или итеративной реализации компонент программной системы средствами конкретного языка. Хорошо программировать − значит правильно и по назначению использовать средства, которые предоставляет язык. Но для создания сложной программной системы требуется нечто большее, чем умение правильно программировать на том или ином языке. Основополагающей и определяющей составляющей процесса создания программной системы является этап проектирования. Проектирование, в отличие от программирования, основное внимание уделяет правильному и эффективному структурированию сложных систем.

Если объектно-ориентированное программирование основано на технологии представления программной системы в виде объектов, то совершенно естественным является определение объектно-ориентированного проектирования как проектирования на основе объектной модели. Буч дает следующее определение объектно-ориентированного проектирования(дизайна) − ООД:

РЕПОЗИТОРИЙ БГУКИ

Объектно-ориентированное проектирование − это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором − алгоритмами.

По аналогии с проектированием можно выделить отличительные особенности объектно-ориентированного анализа (ООА) по сравнению со структурным анализом, основанным на потоках данных в системе.

Буч [] определяет ООА следующим образом:

Объектно-ориентированный анализ − это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области. Можно дать современные пояснения роли и сути ООА в процессе разработке программного обеспечения основываясь на методологии RUP(Rational Unified Process). Основными объектами, с которыми оперирует объектно-ориентированный анализ являются прецеденты взаимодействия (Use-Cases) и актеры (Actors). Причем, прецеденты взаимодействий рассматриваются в сфере отношений, схожих с отношениями возникающими между классами (расширения (extention), включения(inclusion), генерализации(generalization)).

Подводя итог рассмотрению истории развития языков и методологий программирования, следует упомянуть парадигмы различных методов построения программных систем, сформулированные Страуструпом [§2].

Процедурное программирование: Реши какие требуются процедуры; используй наилучшие доступные алгоритмы.

Модульное программирование: Реши, какие требуются модули; разбей программу так, чтобы скрыть данные в модулях.(Принцип сокрытия данных)

Объектное программирование: Реши, какие требуются типы; обеспечь полный набор операция для каждого типа.

Объектно-ориентированное программирование: Реши, какие требуются классы; обеспечь полный набор операций для каждого класса; явно вырази общность через наследование.

Обобщенное программирование: Реши, какие требуются алгоритмы; параметризуй их так, чтобы они могли работать со множеством подходящих типов и структур данных.

Современными языками объектно-ориентированного программирования являются C++ и Java. С середины 90-х годов многие объектно–ориентированные языки реализуются как системы визуального

РЕПОЗИТОРИЙ БГУКИ

программирования, в которых интерфейсная часть программного продукта создается в диалоговом режиме, практически без написания программных операторов. К объектно – ориентированным системам визуального проектирования относятся Visual Basic, Delphi, C++ Builder, Visual C++. Язык VBA (Visual Basic for Applications) – язык приложений Microsoft Office (Excel, Word, Access, Power Point и др). VBA соблюдает основной синтаксис языка и правила программирования языков Basic – диалектов, позволяет создавать макросы для автоматизации выполнения некоторых операций и графический интерфейс пользователя, интеграцию между различными программными продуктами.

Визуальное программирование - (то же самое, что системы быстрой разработки приложений RAD - Rapid Application Development) - способ создания программы для ЭВМ путѐм манипулирования графическими объектами вместо написания еѐ текста.

В визуальном программировании используются специальные объемные (3D) или плоские (2D) графические или псевдографические среды.

То есть в отличие от языков низкого (1GL), среднего (2GL), высокого уровня (3GL), где программирование и язык носят существенно линейный, последовательный характер, в визуальных средах (4GL) мы имеем дело с с существенно разветвленными "пространственными" структурами типа блок-схем. При этом "кирпичиками" этих блок-схем являются заранее разработанные подпрограммы и функции с унифицированным автоматическим "интеллектуальным" интерфейсом. При соединении таких "кирпичиков" их взаимный интерфейс настраивается без участия программиста.

Таким образом, визуальное программирование позволяет нам программировать на уровне алгоритмов, а не программного кода.

Программный код на языках 3GL, 2Gl, 1GL пакет визуального программирования генерирует из составленной программистом "блок-схемы" в автоматическом режиме сам.

Программист лишь в нестандартных случаях корректирует программный код, либо создает на нем дополнительные "пользовательские объекты" - модули-кирпичики для последующего использования в визуальном программировании.

РЕПОЗИТОРИЙ БГУКИ

Тема 5. Визуальное программирование

Лекция 5 (2 часа)

Цель: рассмотреть основные понятия визуального программирования, выявить достоинства и недостатки сред визуального программирования.

Основные вопросы:

Графические языки программирования и визуальные средства разработки.

Типы языков визуального программирования.

Среды визуальной разработки приложений.

Достоинства и недостатки визуального программирования и перспективы развития.

Визуальное программирование - четвертое поколение языков программирования (4GL - 4-th Generation Languages) в отличие от предыдущих поколений перешло к иной, событийной парадигме, то есть программа в них выполняется не последовательно от начального ввода данных до выдачи отчета, как в языках предыдущих поколений, а отдельными короткими "атомарными" звеньями кода (алгоритма), начинаясь с некоторого инициирующего события (прерывания) и заканчиваясь либо новым внешним событием, меняющим ход программы, либо генерируя свое событие - прерывание для другого "атомарного" звена. Таким образом программа в 4GL, как правило, не имеет характера однозначной, заранее предопределенной цепи последовательных звеньев - блоков, как в языках 1GL - 3GL, но некоторой разветвленной сети программных звеньев, последовательность выполнения которых диктуется внешними событиями. Такая парадигма является адекватным ответом на новый характер работы современного программного обеспечения, которое работает в режиме интерактивного взаимодействия с другими программами, устройствами и человеком.

Сами средства визуального программирования состоят из следующих частей:

− *визуальные средства разработки, визуальные среды* под которыми, как правило, подразумевают средства проектирования интерфейсов или какуюлибо CASE-систему для быстрой разработки приложений или SCADA-систему для программирования микроконтроллеров.

− *язык программирования для визуальной среды* − языки программирования со своим синтаксисом, например, графический язык, либо язык, адаптированный к применению в визуальной среде. Это, как правило, языки 3GL и ниже, которые "не видны" конструктору визуального пакета при

РЕПОЗИТОРИЙ БГУКИ

блочном или "кнопочном" программировании. Визуальный пакет в автоматическом режиме генерирует код на таких языках. Лишь при отладке, доводке программного продукта программист вынужден вникать в эти языки. Примеры: ActionScript для пакета Flash, ObjectPascal для пакета Delphi, PHP для систем управления контентом сайта (CMS).

Языки визуального программирования могут быть дополнительно классифицированы в зависимости от типа и степени визуального выражения, на следующие типы:

− языки на основе объектов, когда визуальная среда программирования предоставляет графические или символьные элементы, которыми можно манипулировать интерактивным образом в соответствии с некоторыми правилами;

− языки, в интегрированной среде разработки которых на этапе проектирования интерфейса применяются формы, с возможностью настройкой их свойств. Примеры: Delphi и C++ Builder фирмы Borland, C#

− языки схем, основанные на идее "фигур и линий", где фигуры (прямоугольники, овалы и т. п.) рассматриваются как субъекты и соединяются линиями (стрелками, дугами и др.), которые представляют собой отношения. Пример: UML.

Некоторые авторы не относят языки типа Delphi и C++ Builder фирмы Borland, C# к визуальным языкам, но только определяют их как визуальные среды для текстовых языков. Такая точка зрения имеет лишь частичное обоснование. Дело в том, что за любым визуальным языком, в его основании, обязательно лежит текстовый язык. Примеры: Flash и ActionScript, любой векторный формат графики и язык его скрипт-контента (WMF, VRML). Другое дело, что многие визуальные среды требуют существенного участия программиста в текстовом программировании, так как не являются совершенными, полностью автоматически формирующими код уровня 3GL и ниже. Иное дело - графические языки, в отличие от визуальных, предназначенные для построения графических объектов, а не как инструментальное средство программирования.

Визуальное программирование это способ создания программ путем манипулирования графическими объектами вместо написания кода в текстовом виде.

Визуальное программирование позволяет программировать, используя графические или символьные элементы, которыми можно манипулировать интерактивным образом согласно некоторым правилам, причем пространственное графических объектов использовать в качестве элементов синтаксиса программы. Значительная часть визуальных языков программирования базируется на идее ―фигур и линий", где фигуры (

РЕПОЗИТОРИЙ БГУКИ

прямоугольники, овалы и др.). Рассматриваются как субъекты и соединяются линиями (стрелками, дугами и др.), которые представляют собой отношения. Языки визуального программирования могут быть дополнительно классифицированы в зависимости от типа и степени визуального выражения, на типы:

Природно-визуальные языки имеют неотъемлемое визуальное выражение, для которого нет очевидного текстового эквивалента (например, графический язык G в средеLabVIEW).

Визуально-преобразованый язык является невизуальным языком с наложенным визуальным представлением.

Значительное количество современных языков программирования имеет развитые визуальные средства для разработки графического интерфейса, причем осуществляется программирование размещенных на специальных формах объектов с настройкой их свойств и поведения. CodeGear Delphi и C++ Builder, Microsoft Visual Studio и языки, которые включает в себя это средство (Visual Basic, Visual C#, Visual J# и т.д.) часто путают с графическими языками программирования, предназначенными для задач компьютерной графики. Все перечисленные языки являются визуальными средами, а не графическими языками.

Перечень языков визуального программирования

A-Flow, программное обеспечение общего назначения, которое не требует написания кода	Mindscript
AgentSheets, простой в использовании авторский средство разработки	Morphic
Alice	MST Workshop
AudioMulch	Lego Mindstorms NXT, визуальный язык программирования для набора робототехники Lego Mindstorms
Macromedia Authorware	OpenAlea.Visualea
Apple Automator	OpenBlocks
Aviary Peacock	OpenDX
Baltie	OpenMusic
Befunge	OpenWire
DRAKON, язык, разработанный для проекта космического корабля Буран	OutSystems
EICASLAB	Piet
Executable UML	PointDragon
eXpecco	Prograph
	Ptolemy
	PWCT
	[PWGL], речь, основанная на

РЕПОЗИТОРИЙ БГУКИ

<table>
<tr>
<td>Flowcode
Flowstone DSP
[FxEngine Framework]
JMCAD
G, язык, используемый в LabVIEW
Game Maker, легкая в использовании среда для разработки игр
Google App Inventor, средство для создания приложений Google Android, основанный на OpenBlocks и Kawa
GNU Radio Companion
Grasshopper 3D
Helix
HiAsm
Illumination Software Creator
Kodu, программное средство для создания игр с 3D-интерфейсом, разработанный в Microsoft Research
Kwikpoint
KTechLab
LabVIEW
Ladder logic
Lava
Lily
Limnor
Mama (программное средство)
Marten
Max
Max / MSP
Pure Data
jMax
Microsoft Visual Programming Language, язык для робототехники, которая является одним из компонентов Microsoft Robotics Studio</td>
<td>Common Lisp, CLOS и OpenGL
[Pypes]
Quartz Composer
Quest3D
Reaktor
Red-R
SCADA
Scala Multimedia
Scicos
Simulink
Основанные на Squeak
Etoys - графический скриптовый язык программирования
Scratch, программное средство Массачусетского технологического института для детей 7-12 лет
Autodesk Softimage
Stagecast Creator
SourceBinder
Subtext
SynthMaker
SynthEdit
Tarpipe
Tersus
TestShell
ThingLab
ToonTalk, система программирования для детей
Agilent VEE
VisSim
Virtools
VISION / HPC
WireFusion
Vsxu</td>
</tr>
</table>

Согласно закону Мура, вычислительная мощь компьютеров будет возрастать с огромным темпом каждые полтора года. Однако, в отрасли

РЕПОЗИТОРИЙ БГУКИ

разработки ПО таких высоких результатов достичь не удается, несмотря на постоянно совершенствующиеся технические возможности.

Обычно начальный код программных систем представляется в виде текста. В большинстве случаев основная их часть создается вручную, иногда на помощь приходят автоматизированные генераторы, хотя сути это не меняет: для того, чтобы разобраться в коде, его нужно будет, по крайней мере, прочесть, что является не такой уж простой задачей. Для того, чтобы немного разобраться в масштабах проблемы, приведем наглядный пример. Представим, что перед некоторым разработчиком программного обеспечения стоит цель проанализировать код программной системы, объем которой составляет миллион строк (примерный объем современных крупным проектов). Допустим, одну строку специалист прочитает за три секунды, а работать он будет обычные восемь часов в сутки. Выполнив несложные арифметические расчеты, мы выясним, что всего программисту придется затратить на эту работу свыше ста дней, что в переводе на пятидневную рабочую неделю составляет полгода на рабочем месте. Кроме того, в коде ведь еще нужно разобраться, мало просто его прочесть. Зачастую даже опытный специалист может довольно долго разбираться в коде, написанным, например, начинающим программистом. Поэтому и совершенно нет смысла, да и возможности, подсчитывать, сколько времени займут все эти операции.

Практически любые попытки визуально представить и объяснить принцип работы третьему лицу программный код наталкиваются на потребность делать это с нескольких разных точек зрения. Предположим, если взять модель автомобиля в масштабе 1:43, то это даст возможность составить ясное представление о реальном продукте, в том числе и ребенку. С программным кодом сделать то же самое проблематично. На помощь и приходит различные методы визуализации программного кода.

Один из наиболее востребованных в настоящее время способов визуализации и моделирования программ является UML. UML (англ. UnifiedModelingLanguage − унифицированный язык моделирования) − язык визуального описания принципов работы программных продуктов. UML является языком широкого профиля, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

Однако здесь разработчики сталкиваются сразу с несколькими проблемами:

РЕПОЗИТОРИЙ БГУКИ

–прогрессивные инструментальные средства отчасти могут применяться, как правило, к довольно узким задачам: проектированию графического интерфейса, формированию структуры БД;

– проблема синхронизации начального кода, обычно оформленного как набор текстовых файлов, и его зрительного представления;

–UML нередко критикуют за его громоздкость и трудность в понимании.

Однако, применение UML не ограничивается моделированием программного обеспечения. Его помимо прочего применяют для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

Кроме того унифицированный язык моделирования дает возможность разработчикам ПО добиться соглашения и обобщения в графической интерпретации программного кода, описывая все его аспекты и принципы работы.

Поэтому, несмотря на перечисленные недостатки UML, у него имеются и множество достоинств:

Объектно-ориентированность языка. Исходя из этого, методы описания сравнительно недалеки от современных объектно-ориентированных языков;

Детальность. Язык позволяет описать программную систему со всех сторон, учитывая все детали и нюансы, не упуская ничего важного;

При наглядном и правильном оформлении диаграммы становятся просты и понятны обывателю;

UML гибок. Это означает, что язык имеет возможность добавления новых стандартов, что выводит его применение за пределы программной инженерии;

UML получил обширное распространение и постоянно развивается.

Плюсы и минусы языка моделирования известны уже давно. С одной стороны, это мощная среда для визуального представления программного кода, построения моделей разной сложности и назначения. Другая же сторона медали – реализация этой модели.

Перейдем от формальностей к конкретным примерам. Наиболее удачным стартом в процессе создания программного обеспечения могут стать диаграммы UML. Они довольно просты и «пластичны». С их помощью команда разработчиков с самого начала разработки может обсуждать все детали и нюансы проекта. Но только начальный этап подготовки завершен и все аспекты согласованны, в этот момент и начинаются сложности. В самом начале разработки очень сложно учесть абсолютно все детали и требования. Поэтому, чем дальше идет время, тем больше проект будет претерпевать изменения и расходится со своим первоначальным видом, а UML диаграммы

РЕПОЗИТОРИЙ БГУКИ

перестают быть актуальными и их нужно переделывать (но этим уже никто не занимается). В конце концов, остается единственный документ, описывающий систему полностью – это сам программный код - вещь трудночитаемая. А что, если на поздних этапах разработки подключится новый специалист? Ему будет весьма проблематично понять принцип работы системы. Но это еще не все. Что, если специалист, ведущий разработку с самого начала, который держит в голове большую часть идеи программной реализации, покинет проект по каким-либо причинам? Тогда разработка системы может растянуться на весьма продолжительной срок, а то и вовсе зайдет в тупик.

Есть еще один архиважный в процессе разработки программных продуктов момент: сбор требований. Как говорит Брукс, "наиболее тяжелой частью создания системы программного обеспечения является решение о том, что вы собираетесь создавать. Никакая другая часть работы не может настолько навредить созданию системы, как эта, в том случае, если она неверна. Любую другую часть легче исправить, нежели эту." Так что для создания адекватной системы ПО необходима технология, которая может определить и создать систему, которая будет отвечать поставленным требованиям. Если такая система есть, то уже можно говорить о высоком опыте разработчиков, они точно знают что делают и их проект делает шаги к успеху.

Так как этапы анализа и проектирования независимо от модели ЖЦ считаются характеризующими при построении корпоративных информационных систем, в работе представлен анализ современных методик анализа и проектирования их применимости в разных типах проектов. Значимость этого изыскания обусловлена, с одной стороны, тем, что разработчики обыкновенно регламентируют собственные средства (подходы, нотации), как универсальные, а с иной, тем, что настоящая информация по методологии применения отсутствует.

Если взглянуть издалека, то неотъемлемым преимуществом любой системы – это ее безостановочное динамичное развитие и поддержка. Так методы визуализации программного кода совершенствуются по сей день, и имеют широкое распространение. Например, UML расширяет и позволяет вводить собственные текстовые и графические стереотипы, что способствует его применению не только в сфере программной инженерии.

Но из главного преимущества плавно вытекает и главный недостаток. А заключается он в том, что унифицированный язык моделирования хочет быть чересчур универсальным. UML должен быть, с одной стороны, совместим со всеми языками программирования, а с другой, подходить для решения конкретных задач. Единственным выходом может быть –

РЕПОЗИТОРИЙ БГУКИ

ограничение использования языка моделирования, но этот шаг может привести к избытку заумных слов и формализации документирования проекта, что непростительно в современное время, ведь процесс разработки должен быть прозрачен и понятен.

Хочется отметить, что с развитием программного обеспечения повышается и сложность его разработки. Среднестатистическому человеку процесс создания программ разного уровня порой являет собой непостижимую тайну. Заказчики программных систем обычно и являются этими «обычными» людьми. Но, чтобы они могли донести все свои требования специалистам, которые возьмутся за создание ПО, клиент и исполнитель должны уметь разговаривать на одном языке. Таким языком и является UML. Да, в нем много недостатков, его часто критикуют, но никто не ставит под сомнение важность его существования. Все, что когда-либо было создано, имеет возможность совершенствоваться, идти в ногу с прогрессом. Возможно, уже в скором времени актуальность моей статьи будет исчерпана и про недостатки визуализации программного кода уже и нечего будет сказать. Сделать языки программирования намного ближе к естественным языкам – одна из главных задач современности, и UML в какой-то степени приближает нас к осуществлению этой задачи.

https://scienceforum.ru/2015/article/2015010980
http://bourabai.kz/einf/4gl.htm

РЕПОЗИТОРИЙ БГУКИ

Тема 6. Программирование мобильных приложений с помощью визуальной среды

Лекция 6 (2 часа)

Цель: Рассмотреть особенности программирования мобильных приложений с помощью визуальной среды.

Основные вопросы:

Интерфейс и синтаксис среды визуального программирования.

Представление основных алгоритмических структур.

Типы данных.

Создание игрового проекта для мобильного устройства.

Загрузка созданного приложения в магазин приложений.

Сколько существует программирование, столько существуют в нем и краеугольные камни разработки интерфейсов, в частности графических интерфейсов пользователя. Программирование вручную привычных пользователю окон, кнопок, меню, обработка событий мыши и клавиатуры, включение в программы изображений и звука требует все больше и больше времени программиста. В ряде случаев этот сервис занимает до 80-90% объема программных кодов. Однако, этот труд нередко пропадал почти впустую, поскольку через некоторое время менялся общепринятый стиль графического интерфейса и все приходилось начинать заново.

Выход из этой ситуации обозначился благодаря двум подходам. Первыйиз них — стандартизация многих функций интерфейса, благодаря чему появилась возможность использовать библиотеки, имеющиеся, например, в Windows. В итоге при смене стиля графического интерфейса (например, при переходе от Windows 3.x к Windows 95) приложения смогли автоматически приспосабливаться к новой системе без какого-либо перепрограммирования. На этом пути создались условия для решения одной из важнейших задач совершенствования техники программирования — повторного использования кодов. Однажды разработанные формы, компоненты, функции могут быть впоследствии неоднократно использованы разработчиком или другими программистами для решения их задач. Каждый программист получает доступ к наработкам других программистов и к огромным библиотекам, созданным различными фирмами. Важным условием реализации такого подхода является обеспеченность совместимости программного обеспечения, разработанного на разных языках.

Вторым революционным шагом, кардинально облегчившим жизнь программистов, явилось появление визуального программирования, возникшего в Visual Basic и нашедшего воплощение в системах Delphi и C++ Builder фирмы Borland.

Визуальное программирование позволило свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность за минуты или часы сделать то, на что ранее уходили месяцы работы. В современном виде в C++ Builder это выглядит так.

Среда предоставляет вам формы (в приложении их может быть несколько), на которых размещаются компоненты. Обычно это оконная форма, хотя могут быть и невидимые формы. На форму с помощью мыши переносятся и размещаются пиктограммы компонентов, имеющихся в библиотеках C++ Builder. С помощью простых манипуляций вы можете изменять размеры и расположение этих компонентов. При этом вы все время в процессе проектирования видите результат — изображение формы и

РЕПОЗИТОРИЙ БГУКИ

расположенных на ней компонентов. Вам не надо мучиться, многократно запуская приложение и выбирая наиболее удачные размеры окна и компонентов. Результаты проектирования вы видите, даже не компилируя программу, немедленно после выполнения какой-то операции с помощью мыши.

Но достоинства визуального программирования не сводятся к этому. Самое главное заключается в том, что во время проектирования формы и размещения на ней компонентов C++ Builder автоматически формирует коды программы, включая в нее соответствующие фрагменты, описывающие данный компонент. А затем в соответствующих диалоговых окнах пользователь может изменить заданные по умолчанию значения каких-то свойств этих компонентов и, при необходимости, написать обработчики каких-то событий. То есть проектирование сводится, фактически, к размещению компонентов на форме, заданию некоторых их свойств и написанию, при необходимости, обработчиков событий.

Компоненты могут быть визуальные, видимые при работе приложения, и невизуальные, выполняющие те или иные служебные функции. Визуальные компоненты сразу видны на экране в процессе проектирования в таком же виде, в каком их увидит пользователь во время выполнения приложения. Это позволяет очень легко выбрать место их расположения и их дизайн — форму, размер, оформление, текст, цвет и т.д. Невизуальные компоненты видны на форме в процессе проектирования в виде пиктограмм, но пользователю во время выполнения они не видны, хотя и выполняют для него за кадром весьма полезную работу.

Визуальные среды активно используются для разработки мобильных приложений. В настоящее время рынок мобильных игр не стоит на месте и активно развивается. Если пару лет назад этот рынок был догоняющим, то в 2016 году он стал самым прибыльным рынком всей мировой игровой индустрии, обогнав игры для персональных компьютеров, Mac и консолей [Казанцев]. Ежемесячно в магазинах приложений App Store и Google Play появляются десятки тысяч новых игр [Казанцев]. Однако качество большинства выпускаемых игр оставляет желать лучшего, и пользователи выбирают именно качественные игры. Но не только качество привлекает пользователей, а также новизна, новые игровые механики или новые сочетания игровых механик. Под новизной понимается новый игровой контент, если пользователь проходит одну из игр и ему нравятся игры данного жанра, то он с большой вероятностью будет искать подобную игру, но более качественную и популярную, с хорошими отзывами других пользователей. Опыт приходит со временем и качество игрового контента повышается с каждой новой игрой. Чем больше игр выпускает разработчик,

РЕПОЗИТОРИЙ БГУКИ

тем больше вероятность, что качество конечного продукта будет выше. И для программиста в команде разработчиков встает вопрос: как ускорить процесс разработки с его стороны? Для этого необходимо разработать архитектуру приложения, которая будет по большей части неизменна и лишь улучшаться с каждым новым проектом. При расширении команды, для новых программистов недолжно появляться трудностей с архитектурой проекта, поэтому код должен быть поддерживаемым и расширяемым.

Архитектура проекта должна быть правильно организована и структурирована, что позволит упростить работу с большим объемом кода, если за проект решится взяться другой программист. Для решения этой задачи необходимо выбрать шаблоны проектирования, причем без учета текущего проекта, сделав архитектуру, как можно более абстрактной, чтобы получить модель, подходящую для всех игровых проектов.

В настоящее время на рынке есть множество платформ для разработки игр. Каждая из них обладает своими особенностями. Для игры необходимо найти платформу, которая будет удовлетворять следующим требованиям:

1) возможность разработки для мобильных устройств;

2) возможность разработки 2D игр;

3) удобство использования;

4) наличие качественной документации;

5) свободно распространяемый.

Проанализируем самые популярные приложения.

Unity – кросс-платформенный инструмент для разработки двухмерных и трехмерных приложений и игр, работающий под операционными системами Windows и OS X. Позволяет разрабатывать под все самые известные платформы, такие как: PC, Linux, Mac, IOS, Android, Xbox One, PS4 и т.д. Unity имеет очень простой интерфейс, который разбит на несколько окон: Hierarchy, где находятся названия всех объектов на сцене, которые можно группировать; Scene, где можно рассмотреть игровое поле под нужным ракурсом; Inspector, в котором находятся все свойства выделенного объекта и его компоненты; Project, где находятся все материалы проекта; Toolbar, где находится меню с инструментами. Проект в Unity делится на сцены – отдельные файлы, содержащие свой набор объектов, скриптов и настроек. Основным объектом игровой логики является игровой объект – сущность, которая включает в себя компоненты. Также Unity предоставляет интегрированные сервисы для вовлечения, удержания и монетизации игроков.

Достоинства:

1) удобство использования и простота освоения;

2) качественная документация;

РЕПОЗИТОРИЙ БГУКИ

3) большое сообщество разработчиков использующих Unity;

4) возможность настроить и доработать среду разработки под нужный проект;

5) интегрированные сервисы монетизации и аналитики;

6) кросс-платформенность.

Недостатки:

1) необходимо глубокое знание одного из используемых языков программирования;

2) обновления могут испортить уже рабочий код;

3) условно-бесплатный.

Unreal Engine 4 – кросс-платформенный инструмент для разработки игр, работающий под операционными системами Windows и OS X. Позволяет вести разработку игр под все популярные платформы, такие как: PC, Mac, Linux, Android, IOS и другие. Исходные коды движка находятся в открытом доступе, поэтому при желании можно его доработать под свои нужды. Окно редактора состоит из стандартных окон, такие как: Scene outliner, где находятся содержимое сцены; Content Browser, где находятся все файлы и материалы проекта; Details – окно свойств объекта; Modes – режим работы с контентом. Основным объектом игровой логики является элемент Blueprint – чертеж. Это сборка из компонентов, которая образует сложный объект игрового мира. Управление этим объектом осуществляется с помощью C++ класса или редактора графов. Вместе они дополняют друг друга.

Достоинства:

1) качественная документация;

2) большое сообщество разработчиков использующих Unreal Engine;

3) большое количество обучающего материала;

4) кросс-платформенность.

Недостатки:

1) сложность в освоении;

2) необходимость отдавать 5 % прибыли от игры.

Defold – кросс-платформенный инструмент для профессиональной разработки игр от компании King, известной своими 2D играми. Является абсолютно бесплатным. Defold имеет простой и легкий в освоении интерфейс, а набор инструментов предназначен для работы с 2D проектами. Вся игровая логика контролируется с помощью скриптов, написанных на языке Lua. При создании пользовательских материалов имеется возможность использования OpenGL ES. Платформа была запущена в марте 2016 года, и до сих пор находится в бета тестировании, поэтому у нее нет большого сообщества разработчиков.

РЕПОЗИТОРИЙ БГУКИ

Достоинства:

1) направленность на 2D;

2) абсолютно бесплатный;

3) перспективный.

Недостатки:

1) мало обучающих материалов;

2) поддержка только одного языка программирования.

Так же, как и программирование игр для настольных компьютеров и консольных систем, создание игр для мобильных телефонов требует использования и комбинирования различных приемов и методов программирования, а так же навыков создания игры на миниатюрном мобильном устройстве, имеющем беспроводное соединение с сетью, что делает эту сферу программирования достаточно гибкой с одной стороны, но требующей разносторонней подготовки разработчиков с другой стороны.

Тема 7. Основы языка объектно-ориентированного программирования

Лекция 7 (4 часа)

Цель: Изучить Основы языка объектно-ориентированного программирования на примере C#

Основные вопросы:

Общие сведения о языке объектно-ориентированного программирования: синтаксис, семантика, лексемы, константы, концепция типа данных, стандартные типы данных, переменные, операции, выражения, структура программы, ввод-вывод данных.

Представление основных алгоритмических структур: условный оператор, оператор варианта, операторы циклов, операторы перехода, процедуры выхода из циклов, исключения.

Составные типы данных: массивы, строки, функции обработки строк, перечисления и структуры, файлы.

Среда разработки Visual Studio.Net - это программный продукт, обеспечивающий создание программных продуктов на основе открытости для языков программирования и единого каркас среды разработки.

РЕПОЗИТОРИЙ БГУКИ

Открытость

Среда разработки теперь является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой Microsoft - Visual C++ .Net (с управляемыми расширениями), Visual C# .Net, J# .Net, Visual Basic .Net, - в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами-производителями. Таких расширений среды Visual Studio сделано уже достаточно много, практически они существуют для всех известных языков - Fortran и Cobol, RPG и Component Pascal, Oberon и SmallTalk. Я у себя на компьютере включил в среду компилятор одного из лучших объектных языков - языка Eiffel.

Открытость среды не означает полной свободы. Все разработчики компиляторов при включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение, которое можно считать и главным достоинством, состоит в том, что все языки, включаемые в среду разработки Visual Studio .Net, должны использовать единый каркас - Framework .Net. Благодаря этому достигаются многие желательные свойства: легкость использования компонентов, разработанных на различных языках; возможность разработки нескольких частей одного приложения на разных языках; возможность бесшовной отладки такого приложения; возможность написать класс на одном языке, а его потомков - на других языках. Единый каркас приводит к сближению языков программирования, позволяя вместе с тем сохранять их индивидуальность и имеющиеся у них достоинства. Преодоление языкового барьера - одна из важнейших задач современного мира. Благодаря единому каркасу, Visual Studio .Net в определенной мере решает эту задачу в мире программистов.

Framework .Net - единый каркас среды разработки. В каркасе Framework .Net можно выделить два основных компонента:

- статический - FCL (Framework Class Library) - библиотеку классов каркаса;

- динамический - CLR (Common Language Runtime) - общеязыковую исполнительную среду.

Каркас стал единым для всех языков среды. Поэтому, на каком бы языке программирования ни велась разработка, она использует классы одной и той же библиотеки. Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными хранилищами данных и прочая универсальность.

Встроенные примитивные типы

РЕПОЗИТОРИЙ БГУКИ

Важной частью библиотеки FCL стали классы, задающие примитивные типы - те типы, которые считаются встроенными в язык программирования. Типы каркаса покрывают все множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic - Integer, а в языке C# - int, проецируется на один и тот же тип каркаса System.Int32. В каждом языке программирования, наряду с "родными" для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе. Поэтому, по сути, все языки среды разработки могут пользоваться единой системой встроенных типов, что, конечно, способствует облегчению взаимодействия компонентов, написанных на разных языках.

Структурные типы

Частью библиотеки стали не только простые встроенные типы, но и структурные типы, задающие организацию данных - строки, массивы, перечисления, структуры (записи). Это также способствует унификации и реальному сближению языков программирования.

Архитектура приложений

Существенно расширился набор возможных архитектурных типов построения приложений. Помимо традиционных Windows- и консольных приложений, появилась возможность построения Web-приложений. Большое внимание уделяется возможности создания повторно используемых компонентов - разрешается строить библиотеки классов, библиотеки элементов управления и библиотеки Web-элементов управления. Популярным архитектурным типом являются Web-службы, ставшие сегодня благодаря открытому стандарту одним из основных видов повторно используемых компонентов. Для языков C#, J#, Visual Basic, поддерживаемых Microsoft, предлагается одинаковый набор из 12 архитектурных типов приложений. Несколько особняком стоит Visual C++, сохраняющий возможность работы не только с библиотекой FCL, но и с библиотеками MFC и ATL, и с построением соответствующих MFC и ATL-проектов. Компиляторы языков, поставляемых другими фирмами, создают проекты, которые удовлетворяют общим требованиям среды, сохраняя свою индивидуальность. Так, например, компилятор Eiffel допускает создание проектов, использующих как библиотеку FCL, так и собственную библиотеку классов.

Модульность

Число классов библиотеки FCL велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединяются в группы, называемые пространством имен (Namespace). Для динамического компонента CLR физической

РЕПОЗИТОРИЙ БГУКИ

единицей, объединяющей классы и другие ресурсы, является сборка (assembly).

Основным пространством имен библиотеки FCL является пространство System, содержащее как классы, так и другие вложенные пространства имен. Так, уже упоминавшийся примитивный тип Int32 непосредственно вложен в пространство имен System и его полное имя, включающее имя пространства - System.Int32.

В пространство System вложен целый ряд других пространств имен. Например, в пространстве System.Collections находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов - списками, очередями, словарями. В пространство System.Collections, в свою очередь, вложено пространство имен Specialized, содержащее классы со специализацией, например, коллекции, элементами которых являются только строки. Пространство System.Windows.Forms содержит классы, используемые при создании Windows-приложений. Класс Form из этого пространства задает форму - окно, заполняемое элементами управления, графикой, обеспечивающее интерактивное взаимодействие с пользователем.

Общеязыковая исполнительная среда CLR - динамический компонент каркаса. Наиболее революционным изобретением Framework .Net явилось создание исполнительной среды CLR. С ее появлением процесс написания и выполнения приложений становится принципиально другим

Двухэтапная компиляция. Управляемый модуль и управляемый код.

Компиляторы языков программирования, включенные в Visual Studio .Net, создают модули на промежуточном языке MSIL (Microsoft Intermediate Language), называемом далее просто - IL. Фактически компиляторы создают так называемый управляемый модуль - переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и метаданные - всю необходимую информацию как для CLR, так и конечных пользователей, работающих с приложением. В зависимости от выбранного типа проекта, PE-файл может иметь расширения exe, dll, mod или mdl.

Следует отметить, PE-файл, имеющий расширение exe, хотя и является exe-файлом, но это не совсем обычный исполняемый Windows файл. При его запуске он распознается как специальный PE-файл и передается CLR для обработки. Исполнительная среда начинает работать с кодом, в котором специфика исходного языка программирования исчезла. Код на IL начинает выполняться под управлением CLR. По этой причине код называется *управляемым*. Исполнительную среду можно рассматривать как своеобразную виртуальную IL-машину. Эта машина транслирует "на лету" требуемые для исполнения участки кода в команды реального процессора, который в действительности и выполняет код.

РЕПОЗИТОРИЙ БГУКИ

Виртуальная машина

Каркас Framework.Net перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание PE-модулей на IL отделены от выполнения, и эти процессы могут быть реализованы на разных платформах. В состав CLR входят трансляторы JIT (Just In Time Compiler), которые и выполняют трансляцию IL в командный код той машины, где установлена и функционирует исполнительная среда CLR. Конечно, в первую очередь Microsoft реализовала CLR и FCL для различных версий Windows, включая Windows 98/Me/NT 4/2000, 32 и 64-разрядные версии Windows XP и семейство .Net Server. Для операционных систем Windows CE и Palm разработана облегченная версия Framework .Net.

Следует отметить, что CLR, работая с IL-кодом, выполняет достаточно эффективную оптимизацию и, что не менее важно, защиту кода. Зачастую нецелесообразно выполнять оптимизацию на уровне создания IL-кода - она иногда может не улучшить, а ухудшить ситуацию, не давая CLR провести оптимизацию на нижнем уровне, где можно учесть даже особенности процессора.

Дизассемблер и ассемблер

Для готовых PE-файлов иногда необходимо анализировать его IL-код и связанные с ним метаданные. В состав Framework SDK входит дизассемблер - ildasm, выполняющий дизассемблирование PE-файла и показывающий метаданные, а также IL-код с комментариями в наглядной форме.

Профессионалы, предпочитающие работать на низком уровне, могут программировать на языке ассемблера IL. В этом случае в их распоряжении будет вся мощь библиотеки FCL и все возможности CLR.

Метаданные

Переносимый исполняемый PE-файл является самодокументируемым файлом и, как уже говорилось, содержит и код, и метаданные, описывающие код. Файл начинается с манифеста и включает в себя описание всех классов, хранимых в PE-файле, их свойств, методов, всех аргументов этих методов - всю необходимую CLR информацию. Поэтому помимо PE-файла не требуется никаких дополнительных файлов и записей в реестр - вся нужная информация извлекается из самого файла. Среди классов библиотеки FCL имеется класс Reflection, методы которого позволяют извлекать необходимую информацию. Введение метаданных - не только важная техническая часть CLR, но это также часть новой идеологии разработки программных продуктов.

Еще одной важной особенностью построения CLR является то, что исполнительная среда берет на себя часть функций, традиционно входящих в ведение разработчиков трансляторов, и облегчает тем самым их работу. Один

РЕПОЗИТОРИЙ БГУКИ

из таких наиболее значимых компонентов CLR - сборщик мусора (Garbage Collector). Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения. В ряде языков программирования (классическим примером является язык C/C++) память освобождает сам программист, в явной форме отдавая команды как на создание, так и на удаление объекта. В этом есть своя логика - "я тебя породил, я тебя и убью". Однако можно и нужно освободить человека от этой работы. Неизбежные ошибки программиста при работе с памятью тяжелы по последствиям, и их крайне тяжело обнаружить. Как правило, объект удаляется в одном модуле, а необходимость в нем обнаруживается в другом, далеком модуле. Обоснование того, что программист не должен заниматься удалением объектов, а сборка мусора должна стать частью исполнительной среды, появилось достаточно давно. Наиболее полно оно обосновано в работах Бертрана Мейера и в его книге "Object-Oriented Construction Software", первое издание которой появилось еще в 1988 году.

В CLR эта идея реализована в полной мере. Задача сборки мусора снята не только с программистов, но и с разработчиков трансляторов, она решается в нужное время и в нужном месте - исполнительной средой, ответственной за выполнение вычислений. Здесь же решаются и многие другие вопросы, связанные с использованием памяти, в частности, проверяются возможные нарушения использования "чужой" памяти и другие нарушения, например, с использованием нетипизированных указателей. Данные, удовлетворяющие требованиям CLR и допускающие сборку мусора, называются управляемыми данными.

CLR позволяет работать как с управляемыми, так и с неуправляемыми данными. Однако использование неуправляемых данных регламентируется и не поощряется. Так, в C# модуль, использующий неуправляемые данные (указатели, адресную арифметику), должен быть помечен как небезопасный (unsafe), и эти данные должны быть четко зафиксированы. Исполнительная среда, не ограничивая возможности языка и программистов, вводит определенную дисциплину в применении потенциально опасных средств языков программирования.

Исключительные ситуации

Когда при вызове некоторой функции (процедуры) обнаруживается, что она не может нормальным образом выполнить свою работу, включаются варианты обработки такой ситуации. Функция может возвращать код ошибки или специальное значение типа HResult, может выбрасывать исключение, тип которого характеризует возникшую ошибку. В CLR принято во всех таких ситуациях выбрасывать исключение. Косвенно это влияет и на язык

РЕПОЗИТОРИЙ БГУКИ

программирования. Выбрасывание исключений наилучшим образом согласуется с исполнительной средой. В языке C# выбрасывание исключений, их дальнейший перехват и обработка - основной рекомендуемый способ обработки исключительных ситуаций.

События

У CLR есть свое видение того, что представляет собой тип. Есть формальное описание общей системы типов CTS - Common Type System. В соответствии с этим описанием, каждый тип, помимо полей, методов и свойств, может содержать и события. При возникновении событий в процессе работы с тем или иным объектом данного типа посылаются сообщения, которые могут получать другие объекты. Механизм обмена сообщениями основан на делегатах - функциональном типе. В язык C# встроен механизм событий, полностью согласованный с возможностями CLR.

Исполнительная среда CLR обладает мощными динамическими механизмами - сборки мусора, динамического связывания, обработки исключительных ситуаций и событий. Все эти механизмы и их реализация в CLR созданы на основании практики существующих языков программирования. Но уже созданная исполнительная среда, в свою очередь, влияет на языки, ориентированные на использование CLR. Поскольку язык C# создавался одновременно с созданием CLR, то, естественно, он стал языком, наиболее согласованным с исполнительной средой, и средства языка напрямую отображаются в средства исполнительной среды.

Общие спецификации и совместимые модули

Уже говорилось, что каркас Framework .Net облегчает межъязыковое взаимодействие. Для того чтобы классы, разработанные на разных языках, мирно уживались в рамках одного приложения, для их бесшовной отладки и возможности построения разноязычных потомков они должны удовлетворять некоторым ограничениям. Эти ограничения задаются набором общеязыковых спецификаций - CLS (Common Language Specification). Класс, удовлетворяющий спецификациям CLS, называется CLS-совместимым. Он доступен для использования в других языках, классы которых могут быть клиентами или наследниками совместимого класса.

Спецификации CLS точно определяют, каким набором встроенных типов можно пользоваться в совместимых модулях. Понятно, что эти типы должны быть общедоступными для всех языков, использующих Framework .Net. В совместимых модулях должны использоваться управляемые данные и выполняться некоторые другие ограничения. Ограничения касаются только интерфейсной части класса, его открытых свойств и методов. Закрытая часть класса может и не удовлетворять CLS. Классы, от которых не требуется

РЕПОЗИТОРИЙ БГУКИ

совместимость, могут использовать специфические особенности языка программирования.

Как уже отмечалось, Visual Studio .Net для языков C#, Visual Basic и J# предлагает 12 возможных видов проектов. Среди них есть пустой проект, в котором изначально не содержится никакой функциональности; есть также проект, ориентированный на создание Web-служб. В этой книге, направленной, прежде всего, на изучение языка C#, основным видом используемых проектов будут обычные Windows-приложения. На начальных этапах, чтобы не усложнять задачу проблемами пользовательского интерфейса, будем рассматривать также консольные приложения.

Давайте разберемся, как создаются проекты и что они изначально собой представляют. Поговорим также о сопряженных понятиях: решение (solution), проект (project), пространство имен (namespace), сборка (assembly). Рассмотрим результаты работы компилятора Visual Studio с позиций программиста, работающего над проектом, и с позиций CLR, компилирующей PE-файл в исходный код процессора.

С точки зрения программиста, компилятор создает решение, с точки зрения CLR - сборку, содержащую PE-файл. Программист работает с решением, CLR - со сборкой.

Решение содержит один или несколько проектов, ресурсы, необходимые этим проектам, возможно, дополнительные файлы, не входящие в проекты. Один из проектов решения должен быть выделен и назначен стартовым проектом. Выполнение решения начинается со стартового проекта. Проекты одного решения могут быть зависимыми или независимыми. Например, все проекты одной лекции данной книги могут быть для удобства собраны в одном решении и иметь общие свойства . Изменяя стартовый проект, получаем возможность перехода к нужному примеру. Заметьте, стартовый проект должен иметь точку входа - класс, содержащий статическую процедуру с именем Main, которой автоматически передается управление в момент запуска решения на выполнение. В уже имеющееся решение можно добавлять как новые, так и существующие проекты. Один и тот же проект может входить в несколько решений.

Проект состоит из классов, собранных в одном или нескольких пространствах имен. Пространства имен позволяют структурировать проекты, содержащие большое число классов, объединяя в одну группу близкие классы. Если над проектом работает несколько исполнителей, то, как правило, каждый из них создает свое пространство имен. Помимо структуризации, это дает возможность присваивать классам имена, не задумываясь об их уникальности. В разных пространствах имен могут

РЕПОЗИТОРИЙ БГУКИ

существовать одноименные классы. Проект - это основная единица, с которой работает программист. Он выбирает тип проекта , а Visual Studio создает скелет проекта в соответствии с выбранным типом.

Структура программы в C#

Инструкции

Базовым строительным блоком программы являются инструкции (statement). Инструкция представляет некоторое действие, например, арифметическую операцию, вызов метода, объявление переменной и присвоение ей значения. В конце каждой инструкции в C# ставится точка с запятой (;). Данный знак указывает компилятору на конец инструкции. Например:

```
Console.WriteLine("Привет");
```

Данная строка представляет вызов метода Console.WriteLine, который выводит на консоль строку. В данном случае вызов метода является инструкцией и поэтому завершается точкой с запятой.

Набор инструкций может объединяться в блок кода. Блок кода заключается в фигурные скобки, а инструкции помещаются между открывающей и закрывающей фигурными скобками:

```
{
   Console.WriteLine("Привет");
   Console.WriteLine("Добро пожаловать в C#");
}
```

В данном блоке кода две инструкции, которые выводят на консоль определенную строку. Одни блоки кода могут содержать другие блоки:

```
{
   Console.WriteLine("Первый блок");
   {
      Console.WriteLine("Второй блок");
   }
}
```

Метод Main

Точкой входа в программу на языке C# является метод Main. При создании проекта консольного приложения в Visual Studio, например, создается следующий метод Main:

```
class Program
{
   static void Main(string[] args)
   {
      // здесь помещаются выполняемые инструкции
   }
```


РЕПОЗИТОРИЙ БГУКИ

```
}
```

По умолчанию метод Main размещается в классе Program. Название класса может быть любым. Но метод Main является обязательной частью консольного приложения. Если мы изменим его название, то программа не скомпилируется.

По сути и класс, и метод представляют своего рода блок кода: блок метода помещается в блок класса. Внутри блока метода Main располагаются выполняемые в программе инструкции.

Регистрозависимость

C# является регистрозависимым языком. Это значит, в зависимости от регистра символов какое-то определенные названия может представлять разные классы, методы, переменные и т.д. Например, название обязательного метода Main начинается именно с большой буквы: "Main". Если мы назовем метод "main", то программа не скомпилируется, так как метод, который представляет стартовую точку в приложении, обязательно должен называться "Main", а не "main" или "MAIN".

Комментарии

Важной частью программного кода являются комментарии. Они не являются собственно частью программы, при компиляции они игнорируются. Тем не менее комментарии делают код программы более понятным, помогая понять те или иные его части. Есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //. А многострочный комментарий заключается между символами /* текст комментария */. Он может размещаться на нескольких строках. Например:

```
using System;
 namespace HelloApp
{
   /*
      программа, которая спрашивает у пользователя имя
      и выводит его на консоль
   */
   class Program
   {
      // метод Main - стартовая точка приложения
      static void Main(string[] args)
      {
         Console.Write("Введите свое имя: ");
         string name = Console.ReadLine();       // вводим имя
      }
```

РЕПОЗИТОРИЙ БГУКИ

```
    }
}
```

Переменные

Для хранения данных в программе применяются переменные. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

тип имя_переменной

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

- имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
- в имени не должно быть знаков пунктуации и пробелов
- имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

Например, определим простейшую переменную:

```
string name;
```

В данном случае определена переменная name, которая имеет тип string. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:

```
string name;
string Name;
```

После определения переменной можно присвоить некоторое значение:

```
string name;
name = "Tom";
```

РЕПОЗИТОРИЙ БГУКИ

Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение.

Также мы можем сразу при определении присвоить переменной значение. Данный прием называется инициализацией:

```
string name = "Tom";
```

Отличительной чертой переменных является то, что в программе можно многократно менять их значение.

Литералы представляют неизменяемые значения (иногда их еще называют константами). Литералы можно передавать переменным в качестве значения. Литералы бывают логическими, целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово null.

Логические литералы

Есть две логических константы - true (истина) и false (ложь):

```
Console.WriteLine(true);
Console.WriteLine(false);
```

Целочисленные литералы

Целочисленные литералы представляют положительные и отрицательные целые числа, например, 1, 2, 3, 4, -7, -109. Целочисленные литералы могут быть выражены в десятичной, шестнадцатеричной и двоичной форме.

С целыми числами в десятичной форме все должно быть понятно, так как они используются в повседневной жизни:

```
Console.WriteLine(-11);
Console.WriteLine(5);
Console.WriteLine(505);
```

Числа в двоичной форме предваряются символами 0b, после которых идет набор из нулей и единиц:

```
Console.WriteLine(0b11);       // 3
Console.WriteLine(0b1011);     // 11
Console.WriteLine(0b100001);   // 33
```

Для записи числа в шестнадцатеричной форме применяются символы 0x, после которых идет набор символов от 0 до 9 и от A до F, которые собственно представляют число:

```
Console.WriteLine(0x0A);   // 10
Console.WriteLine(0xFF);   // 255
Console.WriteLine(0xA1);   // 161
```


РЕПОЗИТОРИЙ БГУКИ

Вещественные литералы

Вещественные литералы представляют вещественные числа. Этот тип литералов имеет две формы. Первая форма - вещественные числа с фиксированной запятой, при которой дробную часть отделяется от целой части точкой. Например: 1; 2; 3; 3.14; 100.001; -0.38.

Также вещественные литералы могут определяться в экспоненциальной форме MEp, где M — мантисса, E - экспонента, которая фактически означает "*10^" (умножить на десять в степени), а p — порядок. Например:

```
Console.WriteLine(3.2e3);   // по сути равно 3.2 * 10<sup>3</sup> = 3200
Console.WriteLine(1.2E-1);  // равно 1.2 * 10<sup>-1</sup> = 0.12
```

Символьные литералы

Символьные литералы представляют одиночные символы. Символы заключаются в одинарные кавычки.

Символьные литералы бывают нескольких видов. Прежде всего это обычные символы: 1; 2; 3; '2'; 'A'; 'T'.

Специальную группу представляют управляющие последовательности Управляющая последовательность представляет символ, перед которым ставится обратный слеш. И данная последовательность интерпретируется определенным образом. Наиболее часто используемые последовательности:

'\n' - перевод строки

'\t' - табуляция

'\' - обратный слеш

И если компилятор встретит в тексте последовательность \t, то он будет воспринимать эту последовательность не как слеш и букву t, а как табуляцию - то есть длинный отступ.

Также символы могут определяться в виде шестнадцатеричных кодов, также заключенный в одинарные кавычки.

Еще один способ определения символов представляет использования шестнадцатеричных кодов ASCII. Для этого в одинарных кавычках указываются символы '\x', после которых идет шестнадцатеричный код символа из таблицы ASCII. Коды символов из таблицы ASCII можно посмотреть здесь.

Например, литерал '\x78' представляет символ "x":

```
Console.WriteLine('\x78');    // x
Console.WriteLine('\x5A');    // Z
```

И последний способ определения символьных литералов представляет применение кодов из таблицы символов Unicode. Для этого в одинарных кавычках указываются символы '\u', после которых идет шестнадцатеричный код Unicode. Например, код '\u0411' представляет кириллический символ 'Б':

РЕПОЗИТОРИЙ БГУКИ

```
Console.WriteLine('\u0420');    // Р
Console.WriteLine('\u0421');    // С
```

Строковые литералы

Строковые литералы представляют строки. Строки заключаются в двойные кавычки:

```
Console.WriteLine("hello");
Console.WriteLine("фыва");
Console.WriteLine("hello word");
```

Если внутри строки необходимо вывести двойную кавычку, то такая внутренняя кавычка предваряется обратным слешем:

```
Console.WriteLine("Компания \"Рога и копыта\"");
```

Также в строках можно использовать управляющие последовательности. Например, последовательность '\n' осуществляет перевод на новую строку:

```
Console.WriteLine("Привет \nмир");
```

При выводе на консоль слово "мир" будет перенесено на новую строку:

Привет
мир

null – представляет ссылку, которая не указывает ни на какой объект. То есть по сути отсутствие значения.

Типы данных

Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.

В языке C# есть следующие примитивные типы данных:

bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

```
bool alive = true;
bool isDead = false;
```

byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

```
byte bit1 = 1;
byte bit2 = 102;
```

sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

```
sbyte bit1 = -101;
sbyte bit2 = 102;
```

РЕПОЗИТОРИЙ БГУКИ

short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

```
short n1 = 1;
short n2 = 102;
```

ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

```
ushort n1 = 1;
ushort n2 = 102;
```

int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

```
int a = 10;
int b = 0b101;  // бинарная форма b =5
int c = 0xFF;   // шестнадцатеричная форма c = 255
```

uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

```
uint a = 10;
uint b = 0b101;
uint c = 0xFF;
```

long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

```
long a = -10;
long b = 0b101;
long c = 0xFF;
```

ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

```
ulong a = 10;
ulong b = 0b101;
ulong c = 0xFF;
```

float: хранит число с плавающей точкой от -3.4*1038 до 3.4*1038 и занимает 4 байта. Представлен системным типом System.Single

double: хранит число с плавающей точкой от ±5.0*10-324 до ±1.7*10308 и занимает 8 байта. Представлен системным типом System.Double

decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0*10-28 до ±7.9228*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal

РЕПОЗИТОРИЙ БГУКИ

char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

```
char a = 'A';
char b = '\x5A';
char c = '\u0420';
```

string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

```
string hello = "Hello";
string word = "world";
```

object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

```
object a = 22;
object b = 3.14;
object c = "hello code";
```

Для вывода данных на консоль здесь применяется интерполяция: перед строкой ставится знак $ и после этого мы можем вводить в строку в фигурных скобках значения переменных. Консольный вывод программы:

```
Имя: Tom
Возраст: 33
Вес: 78,65
Работает: False
```

Использование суффиксов

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

```
float a = 3.14F;
float b = 30.6f;
 decimal c = 1005.8M;
decimal d = 334.8m;
```

Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul:

```
uint a = 10U;
long b = 20L;
```

РЕПОЗИТОРИЙ БГУКИ

ulong c = 30UL;

Использование системных типов

Выше при перечислении всех базовых типов данных для каждого упоминался системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа. Например, следующие переменные будут эквивалентны по типу:

```
int a = 4;
System.Int32 b = 4;
```

Неявная типизация

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

```
var hello = "Hell to World";
var c = 20;
Console.WriteLine(c.GetType().ToString());
Console.WriteLine(hello.GetType().ToString());
```

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. В примере выше использовалось выражение Console.WriteLine(c.GetType().ToString());, которое позволяет нам узнать выведенный тип переменной c. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int или System.Int32

Эти переменные подобны обычным, однако они имеют некоторые ограничения. Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

```
// этот код работает
int a;
a = 20;
// этот код не работает
var c;
c= 20;
```

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

```
// этот код не работает
var c=null;
```

Так как значение null, то компилятор не сможет вывести тип данных.

double или decimal

Из выше перечисленного списка типов данных очевидно, что если мы хотим использовать в программе числа до 256, то для их хранения мы можем

РЕПОЗИТОРИЙ БГУКИ

использоват переменные типа byte. При использовании больших значений мы можем взять тип short, int, long. То же самое для дробных чисел - для обычных дробных чисел можно взять тип float, для очень больших дробных чисел - тип double. Тип decimal здесь стоит особняком в том плане, что несмотря на большую разрядность по сравнению с типом double, тип double может хранить большее значение. Однако значение decimal может содержать до 28 знаков после запятой, тогда как значение типа double - 15-16 знаков после запятой.

Decimal чаще находит применение в финансовых вычислениях, тогда как double - в математических операциях. Общие различия между этими двумя типами можно выразить следующей таблицей:

	Decimal	Double
Наибольшее значение	$\sim 10^{28}$	$\sim 10^{308}$
Наименьшее значение (без учета нуля)	10^{-28}	$\sim 10^{-323}$
Знаков после запятой	28	15-16
Разрядность	16 байт	8 байт
Операций в секунду	сотни миллионов	миллиарды

В C# используется большинство операций, которые применяются и в других языках программирования. Операции представляют определенные действия над операндами - участниками операции. В качестве операнда может выступать переменной или какое-либо значение (например, число). Операции бывают унарными (выполняются над одним операндом), бинарными - над двумя операндами и тернарными - выполняются над тремя операндами. Рассмотрим все виды операций.

Бинарные арифметические операции:

Операция сложения двух чисел	int x = 10; int z = x + 12; // 22
Операция вычитания двух чисел	int x = 10; int z = x - 6; // 4
Операция умножения двух чисел	int x = 10; int z = x * 5; // 50

Операция деления двух чисел	int x = 10;
	int z = x / 5; // 2
	double a = 10;
	double b = 3;
	double c = a / b; // 3.33333333

При делении стоит учитывать, что если оба операнда представляют целые числа, то результат также будет округляться до целого числа:

РЕПОЗИТОРИЙ БГУКИ

double z = 10 / 4; //результат равен 2

Несмотря на то, что результат операции в итоге помещается в переменную типа double, которая позволяет сохранить дробную часть, но в самой операции участвуют два литерала, которые по умолчанию рассматриваются как объекты int, то есть целые числа, и результат то же будет целочисленный.

Для выхода из этой ситуации необходимо определять литералы или переменные, участвующие в операции, именно как типы double или float:

Операция инкремента

Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1, а потом ее значение возвращается в качестве результата операции.

И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.

```
int x1 = 5;
int z1 = ++x1; // z1=6; x1=6
Console.WriteLine($"{x1} - {z1}");
int x2 = 5;
int z2 = x2++; // z2=5; x2=6
Console.WriteLine($"{x2} - {z2}");
```

Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и постфиксная (x--).

```
int x1 = 5;
int z1 = --x1; // z1=4; x1=4
Console.WriteLine($"{x1} - {z1}");
int x2 = 5;
int z2 = x2--; // z2=5; x2=4
Console.WriteLine($"{x2} - {z2}");
```

При выполнении сразу нескольких арифметических операций следует учитывать порядок их выполнения. Приоритет операций от наивысшего к низшему:

Инкремент, декремент

Умножение, деление, получение остатка

Сложение, вычитание

Для изменения порядка следования операций применяются скобки.

Рассмотрим набор операций:

```
int a = 3;
int b = 5;
int c = 40;
```

РЕПОЗИТОРИЙ БГУКИ

```
int d = c---b*a;    // a=3  b=5  c=39  d=25
Console.WriteLine($"a={a}  b={b}  c={c}  d={d}");
```

Здесь мы имеем дело с тремя операциями: декремент, вычитание и умножение. Сначала выполняется декремент переменной c, затем умножение b*a, и в конце вычитание. То есть фактически набор операций выглядел так:

```
int d = (c--)-(b*a);
```

Но с помощью скобок мы могли бы изменить порядок операций, например, следующим образом:

```
int a = 3;
int b = 5;
int c = 40;
int d = (c-(--b))*a;    // a=3  b=4  c=40  d=108
Console.WriteLine($"a={a}  b={b}  c={c}  d={d}");
```

Ассоциативность операторов

Как выше было отмечено, операции умножения и деления имеют один и тот же приоритет, но какой тогда результат будет в выражении:

```
int x = 10 / 5 * 2;
```

Стоит нам трактовать это выражение как (10 / 5) * 2 или как 10 / (5 * 2)? Ведь в зависимости от трактовки мы получим разные результаты.

Когда операции имеют один и тот же приоритет, порядок вычисления определяется ассоциативностью операторов. В зависимости от ассоциативности есть два типа операторов:

- Левоассоциативные операторы, которые выполняются слева направо;
- Правоассоциативные операторы, которые выполняются справа налево.

Все арифметические операторы являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение 10 / 5 * 2 необходимо трактовать как (10 / 5) * 2, то есть результатом будет 4.

Условные выражения

Отдельный набор операций представляет условные выражения. Такие операции возвращают логическое значение, то есть значение типа bool: true, если выражение истинно, и false, если выражение ложно. К подобным операциям относятся операции сравнения и логические операции.

Операции сравнения

В операциях сравнения сравниваются два операнда и возвращается значение типа bool - true, если выражение верно, и false, если выражение неверно.

==	
Сравнивает два операнда на равенство. Если они равны, то	int a = 10; int b = 4;

РЕПОЗИТОРИЙ БГУКИ

операция возвращает true, если не равны, то возвращается false:	bool c = a == b; // false

!=
Сравнивает два операнда и возвращает true, если операнды не равны, и false, если они равны.
	int a = 10;
	int b = 4;
	bool c = a != b; // true
	bool d = a!=10; // false

<
Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:
	int a = 10;
	int b = 4;
	bool c = a < b; // false

>
Операция "больше чем". Сравнивает два операнда и возвращает true, если первый операнд больше второго, иначе возвращает false:
	int a = 10;
	int b = 4;
	bool c = a > b; // true
	bool d = a > 25; // false

<=
Операция "меньше или равно". Сравнивает два операнда и возвращает true, если первый операнд меньше или равен второму. Иначе возвращает false.
	int a = 10;
	int b = 4;
	bool c = a <= b; // false
	bool d = a <= 25; // true

>=
Операция "больше или равно". Сравнивает два операнда и возвращает true, если первый операнд больше или равен второму, иначе возвращается false:
	int a = 10;
	int b = 4;
	bool c = a >= b; // true
	bool d = a >= 25; // false

Операции <, > <=, >= имеют больший приоритет, чем == и !=.

Также в C# определены логические операторы, которые также возвращают значение типа bool. В качестве операндов они принимают значения типа bool. Как правило, применяются к отношениям и объединяют несколько операций сравнения.

<table>
<tr><td>|</td><td>bool x1 = (5 > 6) | (4 < 6); // 5 > 6 -</td></tr>
</table>

РЕПОЗИТОРИЙ БГУКИ

<table>
<tr><td>Операция логического сложения или логическое ИЛИ. Возвращает true, если хотя бы один из операндов возвращает true.</td><td>false, 4 < 6 - true, поэтому возвращается true
bool x2 = (5 > 6) | (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false</td></tr>
<tr><td>&
Операция логического умножения или логическое И. Возвращает true, если оба операнда одновременно равны true.</td><td>bool x1 = (5 > 6) & (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
bool x2 = (5 < 6) & (4 < 6); // 5 < 6 - true, 4 < 6 - true, поэтому возвращается true</td></tr>
<tr><td>||
Операция логического сложения. Возвращает true, если хотя бы один из операндов возвращает true.</td><td>bool x1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
bool x2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false</td></tr>
<tr><td>&&
Операция логического умножения. Возвращает true, если оба операнда одновременно равны true.</td><td>bool x1 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
bool x2 = (5 < 6) && (4 < 6); // 5 < 6 - true, 4 < 6 - true, поэтому возвращается true</td></tr>
<tr><td>!
Операция логического отрицания. Производится над одним операндом и возвращает true, если операнд равен false. Если операнд равен true, то операция возвращает false.</td><td>bool a = true;
bool b = !a;   // false</td></tr>
<tr><td>^
Операция исключающего ИЛИ. Возвращает true, если либо первый, либо второй операнд (но не одновременно) равны true, иначе возвращает false</td><td>bool x5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
bool x6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false</td></tr>
</table>

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны.

В выражении z=x|y; будут вычисляться оба значения - x и y.

РЕПОЗИТОРИЙ БГУКИ

В выражении же z=x||y; сначала будет вычисляться значение x, и если оно равно true, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно true. Значение y будет вычисляться только в том случае, если x равно false

То же самое касается пары операций &/&&. В выражении z=x&y; будут вычисляться оба значения - x и y.

В выражении же z=x&&y; сначала будет вычисляться значение x, и если оно равно false, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно false. Значение y будет вычисляться только в том случае, если x равно true

Поэтому операции || и && более удобны в вычислениях, так как позволяют сократить время на вычисление значения выражения, и тем самым повышают производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

Условные конструкции

Условные конструкции - один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей в зависимости от определенных условий.

В языке C# используются следующие условные конструкции: if..else и switch..case

Конструкция if/else

Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
1
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
```

После ключевого слова if ставится условие. И если это условие выполняется, то срабатывает код, который помещен далее в блоке if после фигурных скобок. В качестве условий выступают ранее рассмотренные операции сравнения.

В данном случае у нас первое число больше второго, поэтому выражение num1 > num2 истинно и возвращает true, следовательно, управление переходит к строке Console.WriteLine("Число {num1} больше числа {num2}");

РЕПОЗИТОРИЙ БГУКИ

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
else
{
   Console.WriteLine($"Число {num1} меньше числа {num2}");
}
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию else if, мы можем обрабатывать дополнительные условия:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
else if (num1 < num2)
{
   Console.WriteLine($"Число {num1} меньше числа {num2}");
}
else
{
   Console.WriteLine("Число num1 равно числу num2");
}
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
int num1 = 8;
int num2 = 6;
if(num1 > num2 && num1==8)
{
   Console.WriteLine($"Число {num1} больше числа {num2}");
}
```

РЕПОЗИТОРИЙ БГУКИ

В данном случае блок if будет выполняться, если num1 > num2 равно true и num1==8 равно true.

Конструкция switch

Конструкция switch/case аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
   case "Y":
      Console.WriteLine("Вы нажали букву Y");
      break;
   case "N":
      Console.WriteLine("Вы нажали букву N");
      break;
   default:
      Console.WriteLine("Вы нажали неизвестную букву");
      break;
}
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце каждого блока case должен ставиться один из операторов перехода: break, goto case, return или throw. Как правило, используется оператор break. При его применении другие блоки case выполняться не будут.

Однако если мы хотим, чтобы, наоборот, после выполнения текущего блока case выполнялся другой блок case, то мы можем использовать вместо break оператор goto case:

```
int number = 1;
switch (number)
{
   case 1:
      Console.WriteLine("case 1");
      goto case 5; // переход к case 5
   case 3:
      Console.WriteLine("case 3");
      break;
   case 5:
```

РЕПОЗИТОРИЙ БГУКИ

```
            Console.WriteLine("case 5");
            break;
        default:
            Console.WriteLine("default");
            break;
    }
```

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default, как в примере выше.

Применение оператора return позволит выйти не только из блока case, но и из вызывающего метода. То есть, если в методе Main после конструкции switch..case, в которой используется оператор return, идут какие-либо операторы и выражения, то они выполняться не будут, а метод Main завершит работу.

Оператор throw применяется для выброса ошибок и будет рассмотрен в одной из следующим тем.

Тернарная операция

Тернарную операция имеет следующий синтаксис: [первый операнд - условие] ? [второй операнд] : [третий операнд]. Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
int x=3;
int y=2;
Console.WriteLine("Нажмите + или -");
string selection = Console.ReadLine();
int z = selection=="+"? (x+y) : (x-y);
Console.WriteLine(z);
```

Здесь результатом тернарной операции является переменная z. Если мы выше вводим "+", то z будет равно второму операнду - (x+y). Иначе z будет равно третьему операнду.

Циклы

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

for

foreach

while

do...while

Цикл for

Цикл for имеет следующее формальное определение:

РЕПОЗИТОРИЙ БГУКИ

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
```

Рассмотрим стандартный цикл for:

```
for (int i = 0; i < 9; i++)
{
    Console.WriteLine($"Квадрат числа {i} равен {i*i}");
}
```

Первая часть объявления цикла - int i = 0 - создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и другой числовой тип, например, float. И перед выполнением цикла его значение будет равно 0. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. Пока условное выражение возвращает true, будет выполняться цикл. В данном случае цикл будет выполняться, пока счетчик i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.

В итоге блок цикла сработает 9 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
int i = 0;
for (; ;)
{
    Console.WriteLine($"Квадрат числа {++i} равен {i * i}");
}
```

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; i <;). У нас нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Мы также можем опустить ряд блоков:

```
int i = 0;
for (; i<9;)
{
    Console.WriteLine($"Квадрат числа {++i} равен {i * i}");
}
```

РЕПОЗИТОРИЙ БГУКИ

Этот пример по сути эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Цикл do

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Например:

```
int i = 6;
do
{
    Console.WriteLine(i);
    i--;
}
while (i > 0);
```

Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:

```
int i = -1;
do
{
    Console.WriteLine(i);
    i--;
}
while (i > 0);
```

Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

Цикл while

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int i = 6;
while (i > 0)
{
    Console.WriteLine(i);
    i--;
}
```

Операторы continue и break

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

РЕПОЗИТОРИЙ БГУКИ

Например:

```
for (int i = 0; i < 9; i++)
{
   if (i == 5)
      break;
   Console.WriteLine(i);
}
```

Хотя в условии цикла сказано, что цикл будет выполняться, пока счетчик i не достигнет значения 9, в реальности цикл сработает 5 раз. Так как при достижении счетчиком i значения 5, сработает оператор break, и цикл завершится.

Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию. Для этого мы можем воспользоваться оператором continue:

```
for (int i = 0; i < 9; i++)
{
   if (i == 5)
      continue;
   Console.WriteLine(i);
}
```

В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующей итерации:

Цикл foreach

Цикл foreach перебирает коллекции, например, массивы, и будет рассмотрен далее в теме массивов.

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:

```
тип_переменной[] название_массива;
```

Например, определим массив целых чисел:

```
int[] numbers;
```

После определения переменной массива мы можем присвоить ей определенное значение:

```
int[] nums = new int[4];
```

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new, мы выделили память для 4 элементов массива: new int[4]. Число 4 еще называется длиной массива. При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.

РЕПОЗИТОРИЙ БГУКИ

Также мы сразу можем указать значения для этих элементов:

```
int[] nums2 = new int[4] { 1, 2, 3, 5 };
int[] nums3 = new int[] { 1, 2, 3, 5 };
int[] nums4 = new[] { 1, 2, 3, 5 };
int[] nums5 = { 1, 2, 3, 5 };
```

Все перечисленные выше способы будут равноценны.

Для обращения к элементам массива используются индексы. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0. А чтобы обратиться к четвертому элементу в массиве, нам надо использовать индекс 3, к примеру: nums[3]. Используем индексы для получения и установки значений элементов массива:

```
int[] nums = new int[4];
nums[0] = 1;
nums[1] = 2;
nums[2] = 3;
nums[3] = 5;
Console.WriteLine(nums[3]);  // 5
```

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим исключение IndexOutOfRangeException.

Перебор массивов. Цикл foreach

Цикл foreach предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

```
foreach (тип_данных название_переменной in контейнер)
{
   // действия
}
```

Например:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
foreach (int i in numbers)
{
   Console.WriteLine(i);
}
```

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

Подобные действия мы можем сделать и с помощью цикл for:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
```

РЕПОЗИТОРИЙ БГУКИ

```
    Console.WriteLine(numbers[i]);
}
```

В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = numbers[i] * 2;
    Console.WriteLine(numbers[i]);
}
```

Многомерные массивы

Массивы характеризуются таким понятием как ранг или количество измерений. Выше мы рассматривали массивы, которые имеют одно измерение (то есть их ранг равен 1) - такие массивы можно представлять в виде горзонтального ряда элемента. Но массивы также бывают многомерными. У таких массивов количество измерений (то есть ранг) больше 1.

Массивы которые имеют два измерения (ранг равен 2) называют двухмерными. Например, создадим одномерный и двухмерный массивы, которые имеют одинаковые элементы:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };
int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Массив массивов

От многомерных массивов надо отличать массив массивов или так называемый "зубчатый массив":

```
int[][] nums = new int[3][];
nums[0] = new int[2] { 1, 2 };          // выделяем память для первого подмассива
nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для второго подмассива
nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива
```

Здесь две группы квадратных скобок указывают, что это массив массивов, то есть такой массив, который в свою очередь содержит в себе другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты: new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

РЕПОЗИТОРИЙ БГУКИ

Основные понятия массивов:

Ранг (rank): количество измерений массива

Длина измерения (dimension length): длина отдельного измерения массива

Длина массива (array length): количество всех элементов массива

Например, возьмем массив

int[,] numbers = new int[3, 4];

Массив numbers двухмерный, то есть он имеет два измерения, поэому его ранг равен 2. Длина первого измерения - 3, длина второго измерения - 4. Длина массива (то есть общее количество элементов) - 12.

Классы и объекты

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

По умолчанию проект консольного приложения уже содержит один класс Program, с которого и начинается выполнение программы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова class:

```
class Person
{


}
```

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:

```
using System;
namespace HelloApp
{
   class Person
   {
```

РЕПОЗИТОРИЙ БГУКИ

```
    }
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

```
using System;
namespace HelloApp
{
    class Person
    {
        public string name; // имя
        public int age = 18;     // возраст

        public void GetInfo()
        {
            Console.WriteLine($"Имя: {name}  Возраст: {age}");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Person tom;
        }
    }
}
```

В данном случае класс Person представляет человека. Поле name хранит имя, а поле age - возраст человека. А метод GetInfo выводит все данные на консоль. Чтобы все данные были доступны вне класса Person переменные и метод определены с модификатором public. Поскольку поля фактически те же переменные, им можно присвоить начальные значения, как в случае выше, поле age инициализировано значением 18.

РЕПОЗИТОРИЙ БГУКИ

Так как класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе Main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение null. Поэтому вначале необходимо создать объект класса Person.

Конструкторы

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Конструктор по умолчанию

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Person:

```
class Person
{
    public string name; // имя
    public int age;     // возраст

    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person tom = new Person();
        tom.GetInfo();      // Имя: Возраст: 0

        tom.name = "Tom";
        tom.age = 34;
        tom.GetInfo();  // Имя: Tom Возраст: 34
        Console.ReadKey();
```

РЕПОЗИТОРИЙ БГУКИ

```
    }
}
```

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";.

Консольный вывод данной программы:

Имя: Возраст: 0

Имя: Tom   Возраст: 34

Создание конструкторов

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы:

```
class Person
{
    public string name;
    public int age;

    public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор

    public Person(string n) { name = n; age = 18; }         // 2 конструктор

    public Person(string n, int a) { name = n; age = a; }   // 3 конструктор

    public void GetInfo()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

РЕПОЗИТОРИЙ БГУКИ

```
static void Main(string[] args)
{
    Person tom = new Person();              // вызов 1-ого конструктора без параметров
    Person bob = new Person("Bob");         //вызов 2-ого конструктора с одним параметром
    Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами


    bob.GetInfo();          // Имя: Bob  Возраст: 18
    tom.GetInfo();          // Имя: Неизвестно  Возраст: 18
    sam.GetInfo();          // Имя: Sam  Возраст: 25
}
```

Консольный вывод данной программы:

Имя: Неизвестно Возраст: 18

Имя: Bob Возраст: 18

Имя: Sam Возраст: 25

При этом если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

Стоит отметить, что начиная с версии C# 9.0 мы можем сократить вызов конструктора, убрав из него название типа:

```
Person tom = new ();          // аналогично new Person();
Person bob = new ("Bob");       // аналогично new Person("Bob");
Person sam = new ("Sam", 25);   // аналогично new Person("Sam", 25);
```

Ключевое слово this

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

```
class Person
{
    public string name;
    public int age;

    public Person() : this("Неизвестно")
```


РЕПОЗИТОРИЙ БГУКИ

```
        {
        }
        public Person(string name) : this(name, 18)
        {
        }
        public Person(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
        public void GetInfo()
        {
            Console.WriteLine($"Имя: {name}  Возраст: {age}");
        }
    }
```

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

```
public Person(string name) : this(name, 18)
{
}
```

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Инициализаторы объектов

РЕПОЗИТОРИЙ БГУКИ

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

```
Person tom = new Person { name = "Tom", age=31 };
tom.GetInfo();          // Имя: Tom  Возраст: 31
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.

Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

РЕПОЗИТОРИЙ БГУКИ

II ПРАКТИЧЕСКИЙ РАЗДЕЛ

3.1 Описание лабораторных работ

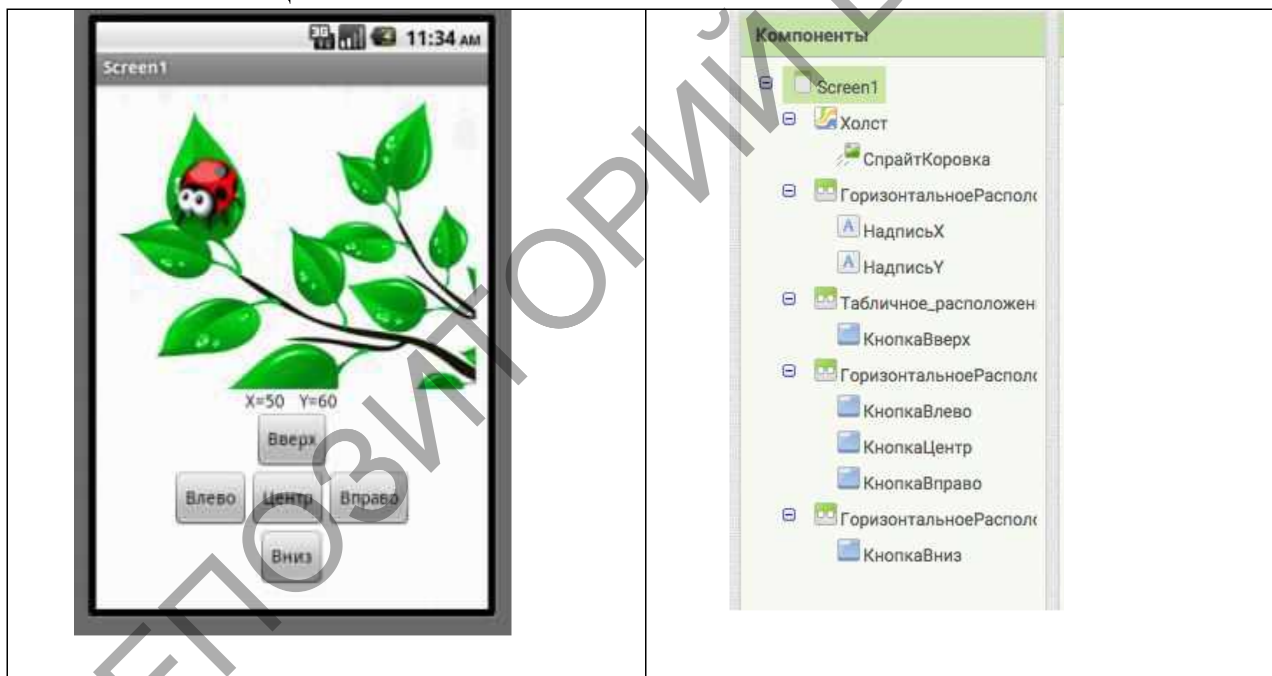
Тема 6. Программирование мобильных приложений с помощью визуальной среды

Лабораторная работа 1. (2 часа)
Основы работы с AppInventor

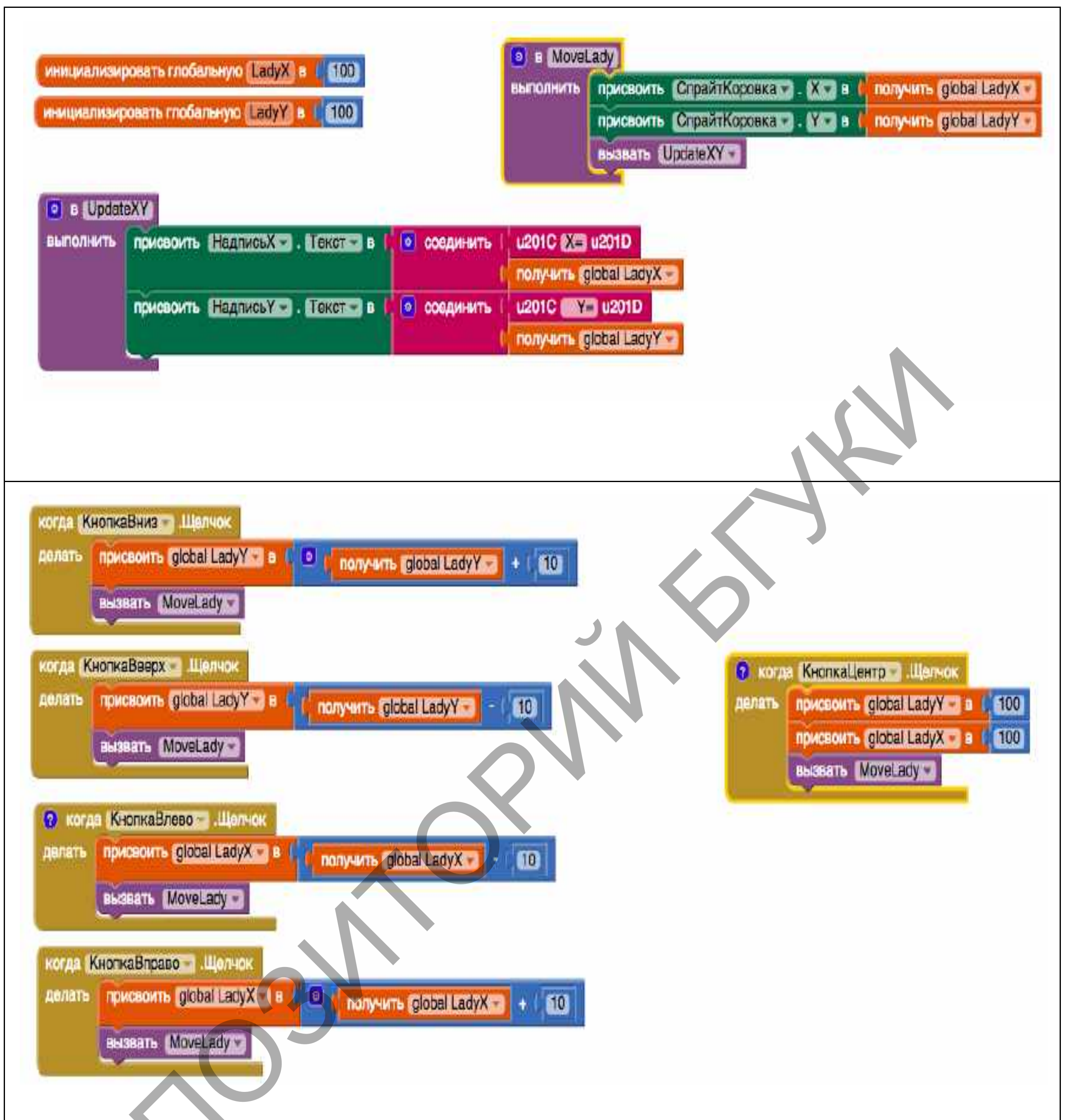
Цель: ознакомиться с основами работы AppInventor

Задание 1. Ознакомиться с интерфейсом приложения AppInventor предварительно посмотрев видеоматериал: https://youtu.be/A92xyAL02GY

Задание 2. Создать приложение, позволяющее управлять движением объекта с помощью кнопок.



РЕПОЗИТОРИЙ БГУКИ



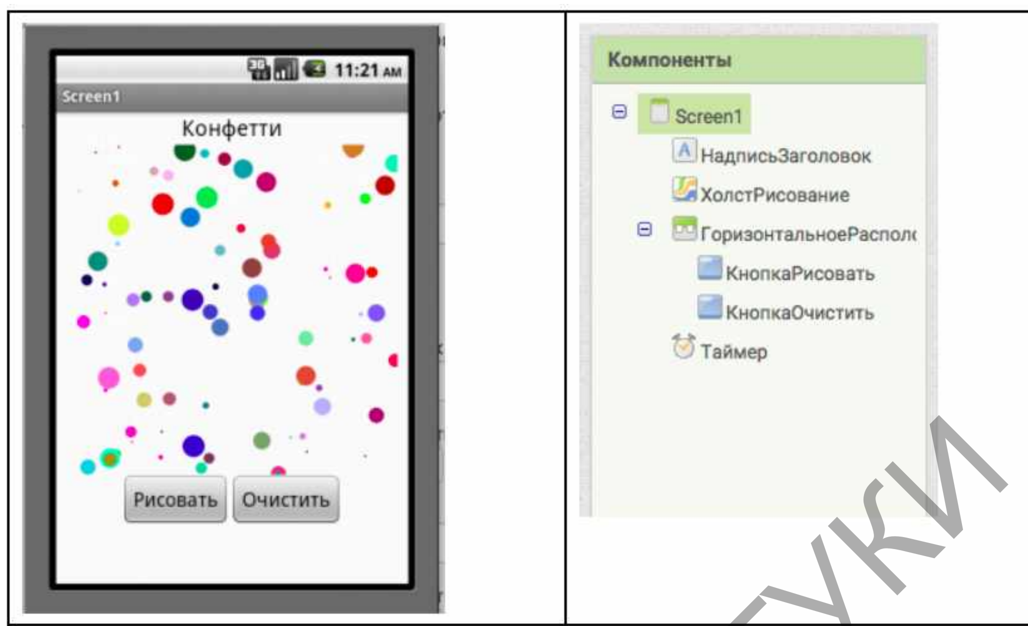
РЕПОЗИТОРИЙ БГУКИ

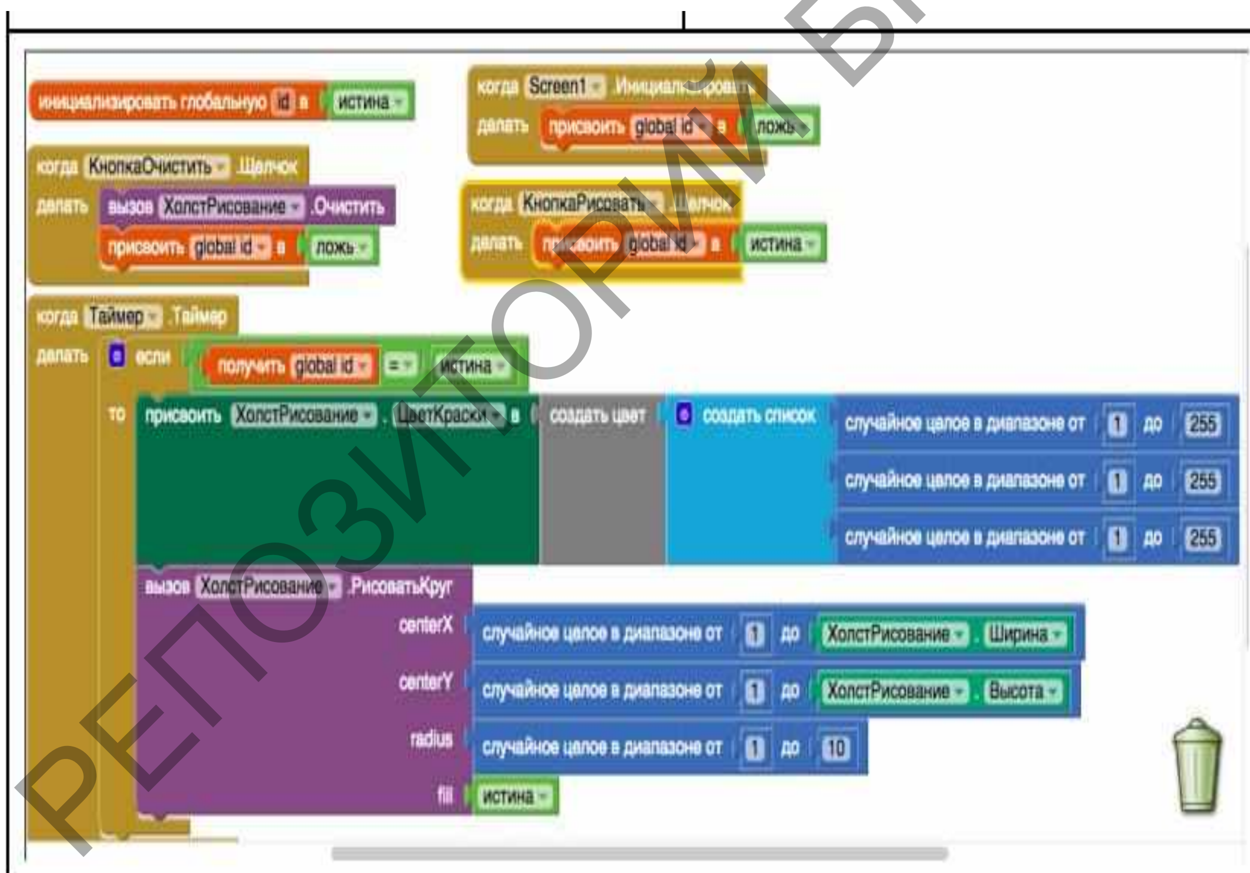
Лабораторная работа 2. (2 часа)

Основы работы с AppInventor. Графические приложения

Цель: Создать графическое приложение в AppInventor

Задача 1. Создать приложение ―Конфетти", в котором при нажатии кнопки холст случайным образом закрашивается точками различного диаметра и цвета.





Тема 7. Основы языка объектно-ориентированного программирования

Лабораторная работа 3 (4 часа)
Основы программирования на C#. Типы данных. Циклы. Массивы.

РЕПОЗИТОРИЙ БГУКИ

Цель: Ознакомиться с основами программирования на C#

Задание 1. Ознакомится с программной средой разработки приложений на языке C#.

Задание 2. Создать программу, в которой определена переменная, значение которой меняется и выводится на консоль:

Пример программного кода:

```
using System;
 namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {
         string name = "Tom";     // определяем переменную и инициализируем ее
          Console.WriteLine(name);    // Tom
          name = "Bob";      // меняем значение переменной
         Console.WriteLine(name);    // Bob
          Console.Read();
      }
   }
```

Задание 3. Определить несколько переменных разных типов и вывести их значения на консоль:

```
using System;
namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {
         string name = "Tom";
         int age = 33;
         bool isEmployed = false;
         double weight = 78.65;

         Console.WriteLine($"Имя: {name}");
         Console.WriteLine($"Возраст: {age}");
         Console.WriteLine($"Вес: {weight}");
```


РЕПОЗИТОРИЙ БГУКИ

```
            Console.WriteLine($"Работает: {isEmployed}");
        }
    }
}
```

Задание 4. Найти количество положительных чисел в массиве { -4, -3, -2, -1, 0, 1, 2, 3, 4 }.

А) для решения задачи рекомендуется использовать предложенный фрагмент кода:

```
int[] numbers = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };
int result = 0;
foreach(int number in numbers)
{
   if(number > 0)
   {
      result++;
   }
}
Console.WriteLine($"Число элементов больше нуля: {result}");
```

Б) для решения задачи рекомендуется использовать предложенный фрагмент кода.

Задание 5. Перевернуть массив в обратном порядке (инверсия массива). Для выполнения задания можно использовать следующий фрагмент кода.

```
int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };
int n = numbers.Length; // длина массива
int k = n / 2;        // середина массива
int temp;             // вспомогательный элемент для обмена значениями
for(int i=0; i < k; i++)
{
   temp = numbers[i];
   numbers[i] = numbers[n - i - 1];
   numbers[n - i - 1] = temp;
}
foreach(int i in numbers)
{
   Console.Write($"{i} \t");
}
```

Задание 6. Сделать сортировку массива чисел, вводимых с экрана.

```
using System;
```

РЕПОЗИТОРИЙ БГУКИ

```
namespace SortApp
{
   class Program
   {
      static void Main(string[] args)
      {
         // ввод чисел
         int[] nums = new int[7];
         Console.WriteLine("Введите семь чисел");
         for (int i = 0; i < nums.Length; i++)
         {
            Console.Write("{0}-е число: ", i + 1);
            nums[i] = Int32.Parse(Console.ReadLine());
         }

         // сортировка
         int temp;
         for (int i = 0; i < nums.Length-1; i++)
         {
            for (int j = i + 1; j < nums.Length; j++)
            {
               if (nums[i] > nums[j])
               {
                  temp = nums[i];
                  nums[i] = nums[j];
                  nums[j] = temp;
               }
            }
         }

         // вывод
         Console.WriteLine("Вывод отсортированного массива");
         for (int i = 0; i < nums.Length; i++)
         {
            Console.WriteLine(nums[i]);
         }
         Console.ReadLine();
      }
   }
}
```

РЕПОЗИТОРИЙ БГУКИ

Лабораторная работа 4 (2 часа)
Рекурсивные функции

Цель: Научиться работать с рекурсивными функциями.

Задание 1. Написать программу вычисления факториала, которое использует формулу n! = 1 * 2 * … * n. Например, факториал числа 5 равен 120 = 1 * 2 * 3 * 4 * 5.

```
static int Factorial(int x)
{
   if (x == 0)
   {
      return 1;
   }
   else
   {
      return x * Factorial(x - 1);
   }
}
```

Задание 2. Вычислить числа Фиббоначчи. Так n-й член последовательности Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1. Последовательность Фибоначчи выглядит следующим образом 0 (0-й член), 1 (1-й член), 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... Для определения чисел этой последовательности определим следующий методы:

Метод 1

```
static int Fibonachi(int n)
{
   if (n == 0)
   {
      return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return Fibonachi(n - 1) + Fibonachi(n - 2);
   }
```

РЕПОЗИТОРИЙ БГУКИ

```
}
```
Метод 2

```
static int Fibonachi(int n)
{
   if (n == 0 || n == 1)
   {
      return n;
   }
   else
   {
      return Fibonachi(n - 1) + Fibonachi(n - 2);
   }
}
```

Лабораторная 5 (4часа)
Основы работы в Unity. Функция Start. Объекты GameObjects и Prefabs

Цель: ознакомиться с основами работы с объектами GameObjects и Prefabs

Задание 1. Ознакомиться с интерфейсом Unity, используя Unity User Manual:

https://docs.unity3d.com/ru/2018.4/Manual/UsingTheEditor.html

Задание 2. Создать стену из кубических блоков 4 на 10.

Пример кода.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Wall : MonoBehaviour {
// Use this for initialization
public GameObject block;
public int width = 10;
public int height = 4;

void Start()
{
for (int y=0; y<height; ++y)
 {
```

РЕПОЗИТОРИЙ БГУКИ

```
  for (int x=0; x<width; ++x)
   {
     Instantiate(block, new Vector3(x,y,0), Quaternion.identity);
    }
 }
}
// Update is called once per frame
void Update () {
}
}
```

Задание 3. Создать цилиндрическую стену из кубических блоков 4 на 20, так чтобы кубик располагались в шахматном порядке.

Пример кода

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class M_Circle : MonoBehaviour {
public GameObject block;
public int height = 4;
public int numberOfObjects = 20;
public float radius = 5f;
public float eps;
// Use this for initialization
void Start () {
eps =  Mathf.PI * 2 / (height + 1);
for (int y = 0; y < height; ++y) {
for (int i = 0; i < numberOfObjects; i++)
```

РЕПОЗИТОРИЙ БГУКИ

```
{
    float angle = i * Mathf.PI * 2 / numberOfObjects+y* Mathf.PI/6;
    float x = Mathf.Cos(angle) * radius;
    float z = Mathf.Sin(angle) * radius;
    Vector3 pos = transform.position + new Vector3(x, y, z);
    float angleDegrees = -angle*Mathf.Rad2Deg;
    Quaternion rot = Quaternion.Euler(0, angleDegrees, 0);
    Instantiate(block, pos, rot);
}
}
}
// Update is called once per frame
void Update () {
}
}
```

Задание 4. Создать конусообразную стену из кубических блоков 4 на 20, так чтобы кубик располагались в шахматном порядке.

Лабораторная 6 (4 часа) *Основы работы в Unity. Функция Update. Класс MonoBehaviour. Компонент Transform*

Цель: ознакомиться с основами работы с классом MonoBehaviour и компонентом Transform.

Задание 1. Создать сферу, которая катится по плоскости, достигая конца плоскости, возвращается обратно. Движение сферы заканчивается по нажатию клавиши.

Задание 2. Создать две сферы, которые катятся по плоскости, достигая конца плоскости, возвращаются обратно. Движение сфер заканчивается по нажатию клавиши. Обратите внимание, что время от времени сферы сталкиваются друг с другом.

Задание 3. Создать две сферы, которые катятся по плоскости, достигая конца плоскости, возвращаются обратно. При встрече и возможности столкновения одна сфера пропускает другую, при этом обе сферы меняют цвет. Движение сфер заканчивается по нажатию клавиши.

Для решения задач можно использовать пример кода.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

РЕПОЗИТОРИЙ БГУКИ

```
public class SphereXX : MonoBehaviour {
public bool flag=true;
public int i_step=0;
public int X_pos= 5; // coordinate of the plane end
// Use this for initialization
void Start () {
transform.GetComponent<Renderer> ().material.color = Color.red;
}
// Update is called once per frame
void Update () {
if(flag==true && transform.position.x < X_pos ) //Кнопка смены цвета
{
transform.position += new Vector3 (1, 0, 0) * Time.deltaTime;
transform.GetComponent<Renderer> ().material.color = Color.red;
i_step ++ ;
if (transform.position.x >= X_pos)//Цвет кубика
{
flag=false;
transform.GetComponent<Renderer> ().material.color = Color.magenta;
}
}

if(flag==false)
{
transform.position -= new Vector3 (1, 0, 0) * Time.deltaTime;
transform.GetComponent<Renderer> ().material.color = Color.magenta;
i_step -- ;
if (i_step==0)
{
flag=true;
transform.GetComponent<Renderer> ().material.color = Color.red;
}
}
}
}

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class SphereZZ : MonoBehaviour {
```

РЕПОЗИТОРИЙ БГУКИ

```
public GameObject SphereXX;
public bool flag=true;
public bool check_move =false;
public int i_step=0;
public float Z_pos= 5; // coordinate of the plane end
public float Z_control=1;
public float X_control=1;
public float eps = 0.1f;

// Use this for initialization
void Start () {
transform.GetComponent<Renderer> ().material.color = Color.blue;
}
// Update is called once per frame
void Update () {
if(flag==true && transform.position.z < Z_pos ) //Кнопка смены цвета
{
transform.GetComponent<Renderer> ().material.color = Color.blue;
SphereXX.transform.GetComponent<Renderer> ().material.color =
Color.red;
if (transform.position.z >= -Z_control-eps && transform.position.z <= -
Z_control+eps)
check_move = true;
else
check_move = false;
if (check_move == false) {
transform.position += new Vector3 (0, 0, 1) * Time.deltaTime;
i_step++;
} else if ((SphereXX.transform.position.x < -X_control ||
SphereXX.transform.position.x > X_control))
{
transform.position += new Vector3 (0, 0, 1) * Time.deltaTime;
transform.GetComponent<Renderer> ().material.color = Color.blue;
i_step++;
} else {
transform.GetComponent<Renderer> ().material.color = Color.cyan;
SphereXX.transform.GetComponent<Renderer> ().material.color =
Color.magenta;
}
if (transform.position.z >= Z_pos)//Цвет кубика
```

РЕПОЗИТОРИЙ БГУКИ

```
{
flag=false;
}
}

if(flag==false) //Кнопка смены цвета
{
transform.GetComponent<Renderer> ().material.color = Color.blue;
SphereXX.transform.GetComponent<Renderer> ().material.color = Color.red;
if (transform.position.z <= Z_control+eps && transform.position.z >= Z_control-eps) {
check_move = true;
}
else
check_move = false;
if (check_move == false) {
transform.position -= new Vector3 (0, 0, 1) * Time.deltaTime;
i_step--;
} else if ((SphereXX.transform.position.x < -X_control || SphereXX.transform.position.x > X_control)) {
transform.position -= new Vector3 (0, 0, 1) * Time.deltaTime;
i_step--;
} else {
transform.GetComponent<Renderer> ().material.color = Color.cyan;
SphereXX.transform.GetComponent<Renderer> ().material.color = Color.magenta;
}
if (i_step==0)//Цвет кубика
{
flag=true;
}
}
}
}
```

РЕПОЗИТОРИЙ БГУКИ

3.2 Тематика семинарских занятий

При подготовке к семинарским занятиям рекомендуется:

1. Подготовить краткий конспект по каждому вопросу семинарского занятия.
2. Подготовить интерактивную презентацию по любому из вопросов.

Семинар 1. Тема: Языки программирования. Классификация языков программирования.

Цель: Изучить парадигмы программирования, повлиявшие на развитие языков программирования.

Вопросы для обсуждения:

1. Парадигма программирования: объектно-ориентированное программирование;
2. Парадигма программирования: функциональное программирование.
3. Парадигма программирования: процедурное программирование.
4. Парадигма программирования: метапрограммирование.
5. Парадигма программирования: обобщѐнное программирование.
6. Парадигма программирования: логическое программирование.

Семинар 2. Тема: Системы программирования

Цель: Изучить специфики и назначение библиотек в программировании.

Вопросы для обсуждения:

1. Библиотеки: содержание стандартной библиотеки.
2. Библиотеки: подходы к определению круга задач (максимальная универсальность алгоритмов и максимальный охват алгоритмов).
3. Библиотеки: статические и динамические библиотеки.
4. Библиотеки: открытые и коммерческие библиотеки.
5. Библиотеки: требования к средствам стандартной библиотекой (11 принципов Бьѐрна Страуструпа).
6. Библиотеки: специфика и особенности графических библиотек.
7. Библиотеки: Утилиты для работы с библиотеками.

РЕПОЗИТОРИЙ БГУКИ

Семинар 3. Тема: Объектно-ориентированное программирование.

Цель: изучить основные понятия объектно-ориентированного программирования.

Вопросы для обсуждения:

1. Классы и объекты.
2. Свойства и поведение объектов.
3. Инкапсуляция и наследование.
4. Полиморфизм и перегрузка методов.
5. Абстрактные классы.
6. Интерфейсы и особенности их создания.

Семинар 4. Тема: Системы объектно-ориентированного программирования

Цель: Рассмотреть области применения технологий объектно-ориентированного программирования на примерах языков программирования.

Вопросы для обсуждения:

1. Объектно-ориентированный язык программирования C#.
2. Объектно- ориентированный язык программирования: Visual Basic.
3. Объектно- ориентированный язык программирования: JavaScript.
4. Объектно- ориентированный язык программирования: Python.
5. Объектно- ориентированный язык программирования: F#.
6. Объектно- ориентированный язык программирования: PHP.

Семинар 5. Тема: Визуальное программирование

Цель: Рассмотреть области применения технологий визуального программирования на примерах визуальных сред программирования.

Вопросы для обсуждения:

1. Способ визуализации и моделирования программ UML.
2. Программное средство Scratch.
3. Средство для создания приложений Google App Inventor.
4. Программное средство для создания игр с 3D-интерфейсом Kodu.
5. Язык для робототехники Microsoft Visual Programming Language.

РЕПОЗИТОРИЙ БГУКИ

6. Визуальный язык программирования для набора робототехники Lego Mindstorms Lego Mindstorms NXT.

Семинар 6. Тема: Программирование мобильных приложений с помощью визуальной среды

Цель: Изучить основы программирования мобильных приложений с помощью визуальной среды App Inventor.

Вопросы для обсуждения:

1. Интерфейс и синтаксис среды визуального программирования App Inventor.
2. Представление основных алгоритмических структур App Inventor.
3. Типы данных App Inventor.
4. Создание игрового проекта для мобильного устройства App Inventor.
5. Загрузка созданного приложения App Inventor в магазин приложений.

РЕПОЗИТОРИЙ БГУКИ

3.3 Тематика и перечень заданий для самостоятельной работы студентов

Тема 6. Программирование мобильных приложений с помощью визуальной среды

1. Ознакомиться с технологиями создания Android-приложений с помощью среды разработки Intellij IDEA на базе Android Studio (4 часа)
2. Проанализировать онлайн конструкторы для создания мобильных приложений (4 часа).
3. Проанализировать онлайн конструкторы для создания игр (4 часа).
4. Создать мобильное игровое приложение при помощи веб-платформы MIT App Inventor (12 часов).

Тема 7. Основы языка объектно-ориентированного программирования

1. Изучить технологии создания 2D игр в среде Unity (12 часов).
2. Изучить технологии создания 3D игр в среде Unity (12 часов).
3. Создать игровое приложение в среде Unity (24 часа).

РЕПОЗИТОРИЙ БГУКИ

IV. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

Контрольные вопросы

1. Инструментальные средства разработки программ. Среды разработки. Компилятор или интерпретатор.
2. Библиотеки стандартных программ и функций. Отладочные программы.
3. Графические библиотеки. Утилиты для работы с библиотеками.
4. Понятие объектных технологий. Классы и объекты. Особенности работы с объектами.
5. Особенности работы с объектами: Модификаторы доступа. Инкапсуляция. Полиморфизм и перегрузка методов. Наследование. Абстрактные классы.
6. Интерфейсы и особенности их создания.
7. Ссистемы и языки объектно-ориентированного программирования.
8. Область применения технологии объектно-ориентированного программирования.
9. Системы быстрой разработки приложений.
10. Современные системы объектно-ориентированного программирования.
11. Графические языки программирования и визуальные средства разработки.
12. Типы языков визуального программирования.
13. Среды визуальной разработки приложений. Достоинства и недостатки визуального программирования и перспективы развития.
14. Интерфейс и синтаксис среды визуального программирования.
15. Визуальное программирование: Представление основных алгоритмических структур. Типы данных.
16. Создание игрового проекта для мобильного устройства. Загрузка созданного приложения в магазин приложений.
17. Объектно-ориентированное программирование: синтаксис, семантика, лексемы, константы.
18. Объектно-ориентированное программирование: концепция типа данных, стандартные типы данных, переменные, операции, выражения,
19. Объектно-ориентированное программирование: структура программы, ввод-вывод данных.
20. Представление основных алгоритмических структур: условный оператор, оператор варианта, операторы циклов.
21. Представление основных алгоритмических структур: Операторы перехода, процедуры выхода из циклов, исключения.

РЕПОЗИТОРИЙ БГУКИ

22. Составные типы данных: массивы, строки, перечисления и структуры, файлы.
23. Современные высокоуровневые технологии программирования.
24. Разработка приложений Windows, основные принципы.
25. Ссообщения и события, программирование, управляемое событиями.
26. Визуальная разработка: основные компоненты форм, графики и анимации,
27. Визуальная разработка: обработка исключений, интерфейсы,
28. Визуальная разработка: универсальные классы, делегаты, атрибуты.

РЕПОЗИТОРИЙ БГУКИ

V. ВСПОМОГАЬЕЛЬНЫЙ РАЗДЕЛ

5.1 Учебно-методическая карта

Номер раздела, темы, занятия	Название раздела, темы	Количество аудиторных часов				Формы контроля занятий
		лекции	практические занятия	лабораторные занятия	управляемая самостоятельная работа студента	
1	**2**	**3**	**4**	**5**	**6**	**7**
1.	Языки программирования. Классификация языков программирования	2	2			
2.	Системы программирования	2	2			
3.	Объектно-ориентированное программирование	2	2			
4.	Системы объектно-ориентированного программирования	2	2			
5.	Визуальное программирование	2	2			
6.	Программирование мобильных приложений с помощью визуальной среды	2	2	4	24	
7.	Основы языка объектно-ориентированного программирования	4		14	42	
	Всего	**16**	**12**	**18**	**66**	**Зачет**

РЕПОЗИТОРИЙ БГУКИ

5.2 Литература

Основная

1. *Макдональд, М.* Веб-разработка. Исчерпывающее руководство = Creating a Website. The missing manual / Мэтью Макдональд ; [пер. с англ. С. Черников]. – 4-е изд. – Санкт-Петербург : Питер, 2017. - 638 с.
2. *Маклафлин, Бр.* Объектно-ориентированный анализ и проектирование = Head First Object-Oriented Analysis and Design / Б. Маклафлин, Г. Поллайс, Д. Уэст ; [пер. с англ. Е. Матвеев]. - Санкт-Петербург : Питер, 2017. – 601 с.
3. *Трофимов, В. В.* Алгоритмизация и программирование : учебник для вузов / В. В. Трофимов, Т. А. Павловская ; под ред. В. В. Трофимова. - Москва : Юрайт, 2020. – 136.
4. *Кушняров, П. В.* Дополненная реальность в образовании: виды и алгоритм создания / П. В. Кушняров // Веснік адукацыі. – 2019. - № 6. – С. 12-16. – Бібліягр.: с. 16 (3 назв.).
5. *Елисеев, В.С.* Разработка мобильного приложения с рекламной интеграцией / В.С. Елисеев ; Донской государственный технический университет. – Ростов-на-Дону : б.и., 2020. – 83 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=595615 (дата обращения: 12.01.2021). – Текст : электронный.
6. *Зайцев, М.Г.* Объектно-ориентированный анализ и программирование : учебное пособие / М.Г. Зайцев ; Новосибирский государственный технический университет. – Новосибирск : Новосибирский государственный технический университет, 2017. – 84 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=576800 (дата обращения: 12.01.2021).
7. *Пушкарев, А.Н.* Языки программирования: учебно-методическое пособие для студентов направления «Информационные системы и технологии» (академический и прикладной бакалавриат) (Дидактические материалы для самостоятельной работы) / А.Н. Пушкарев ; Тюменский государственный университет. – Тюмень : Тюменский государственный университет, 2018. – 48 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=571547 (дата обращения: 12.01.2021).
8. *Дубровин, В.В.* Программирование на C# : учебное пособие : в 2 ч. / В.В. Дубровин ; Тамбовский государственный технический университет. – Тамбов : Тамбовский государственный технический университет (ТГТУ), 2017. – Ч. 1. – 81 с.– Режим доступа: по подписке. –

РЕПОЗИТОРИЙ БГУКИ

URL:https://biblioclub.ru/index.php?page=book&id=499439 (дата обращения: 12.01.2021).

9. Мирошниченко, И.И. Языки и методы программирования : учебное пособие / И.И. Мирошниченко, Е.Г. Веретенникова, Н.Г. Савельева ; Ростовский государственный экономический университет (РИНХ). – Ростов-на-Дону : Издательско-полиграфический комплекс РГЭУ (РИНХ), 2019. – 188 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=567706 (дата обращения: 12.01.2021).

10. Колесникова, Т. Г. Языки программирования : учебное пособие / Т. Г. Колесникова. –Кемеровский государственный университет. – Кемерово : Кемеровский государственный университет, 2019. – 182 с.– Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=573802 (дата обращения: 12.01.2021).

11. Пирская, Л.В. Разработка мобильных приложений в среде Android Studio : учебное пособие / Л.В. Пирская ; Южный федеральный университет. – Ростов-на-Дону ; Таганрог : Южный федеральный университет, 2019. – 125 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=598634 (дата обращения: 12.01.2021).

12. Гуськова, О.И. Объектно ориентированное программирование в Java : учебное пособие / О.И. Гуськова ; Московский педагогический государственный университет. – Москва : Московский педагогический государственный университет (МПГУ), 2018. – 240 с. : ил. – Режим доступа: по подписке. – URL:https://biblioclub.ru/index.php?page=book&id=500355 (дата обращения: 12.01.2021). – Библиогр. в кн. – ISBN 978-5-4263-0648-6.

13. Теоретические основы информационных технологий : учеб.-метод. комплекс / Сост. : П. В. Гляков, Т. С. Жилинская, Т. И. Песецкая. – Минск : БГУКИ, 2017. – 319 с.

14. Кнут, Д. Искусство программирования для ЭВМ: В 3 т. Т.1: Основные алгоритмы / Д. Кнут. – М. : Мир, 2000. – 884 с.

15. Соколова, В. В. Вычислительная техника и информационные технологии. Разработка мобильных приложений : учебное пособие для прикладного бакалавриата / В. В. Соколова. — Москва : Издательство Юрайт, 2019. – 175 с.

16. Федотенко М. Разработка мобильных приложений. Первые шаги. / Мария Федотенко. – Лаборатория знаний, 2019 г. – 336 с.

Бонд Д. "Unity и C#. Геймдев от идеи до реализации" / Д. Бонд. – Издательский Дом ПИТЕР, 2019. – 928 с.

РЕПОЗИТОРИЙ БГУКИ

17. Хокинг Д. Unity в действии. Мультиплатформенная разработка на C# / Джозеф Хокинг. – Питер, 2019. – 336 с.

18. Мэннинг, Д. Unity для разработчика : мобильные мультиплатформенные игры / Джон Мэннинг, Пэрис Батфилд-Эддисон ; пер. А. Киселев. - Санкт-Петербург [и др.] : Питер, 2018. – 352 с.

Дополнительная

1. Гляков, П. В. Игровая форма обучения алгоритмизации культу¬рологов-менеджеров / П. В. Гляков // Информатизация образования. – 2010. – № 3. – С. 21-27.

2. Гончарова, С. А. Мобильные приложения как медиаканал XXI в. / С. А. Гончарова, Т. С. Жилинская // Культура. Наука. Творчество = Культура. Навука. Творчасць = Culture. Science. Arts : сборник научных статей : XII Международная научно-практическая конференция (Минск, 3 мая 2018 г.) / Белорусский государственный университет культуры и искусств [и др.]. – Минск, 2018. – [Вып. 12]. – С. 174-178.

18. Мэннинг, Д. Unity для разработчика : мобильные мультиплатформенные игры / Джон Мэннинг, Пэрис Батфилд-Эддисон ; пер. А. Киселев. - Санкт-Петербург [и др.] : Питер, 2018. – 352 с.

РЕПОЗИТОРИЙ БГУКИ