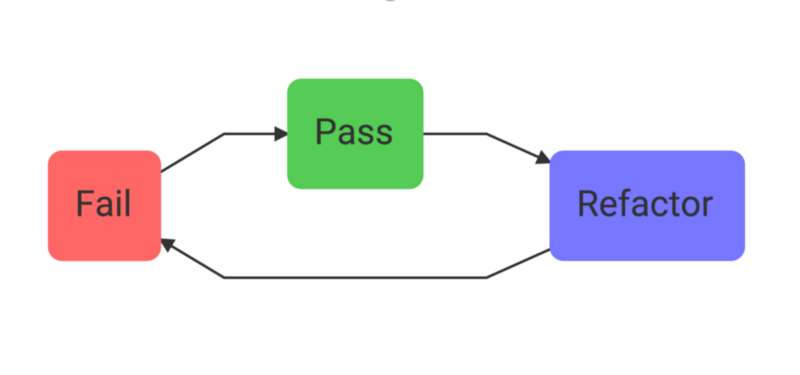
**Вопрос 1. Различные концепции разработки программного обеспечения (TDD, waterfall, cascade, iterative)**

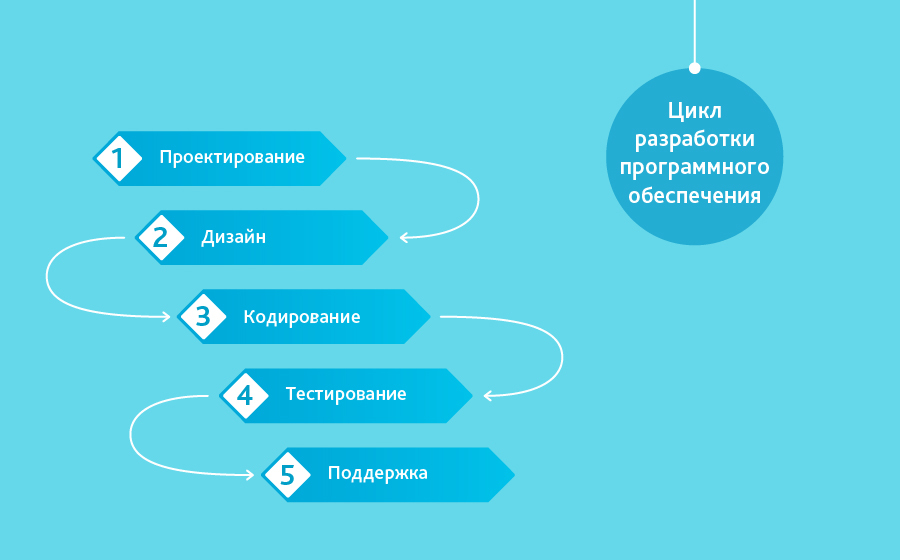
**TDD** расшифровывается как Test Driven Development (разработка через тестирование). Процесс, реализуемый в ходе применения этой методологии очень прост:



Вот основные принципы применения TDD:

1. Прежде чем писать код реализации некоей возможности, пишут тест, который позволяет проверить, работает ли этот будущий код реализации, или нет. Прежде чем переходить к следующему шагу, тест запускают и убеждаются в том, что он выдаёт ошибку. Благодаря этому можно быть уверенным в том, что тест не выдаёт ложноположительные результаты, это — своего рода тестирование самих тестов.
2. Создают реализацию возможности и добиваются того, чтобы она успешно прошла тестирование.
3. Выполняют, если это нужно, рефакторинг кода. Рефакторинг, при наличии теста, который способен указать разработчику на правильность или неправильность работы системы, вселяет в разработчика уверенность в его действиях.

**«Waterfall Model» (каскадная модель или «водопад»)**



Одна из самых старых, подразумевает последовательное прохождение стадий, каждая из которых должна завершиться полностью до начала следующей. В модели Waterfall легко управлять проектом. Благодаря её жесткости, разработка проходит быстро, стоимость и срок заранее определены. Но это палка о двух концах. Каскадная модель будет давать отличный результат только в проектах с четко и заранее определенными требованиями и способами их реализации. Нет возможности сделать шаг назад, тестирование начинается только после того, как разработка завершена или почти завершена. Продукты, разработанные по данной модели без обоснованного ее выбора, могут иметь недочеты (список требований нельзя скорректировать в любой момент), о которых становится известно лишь в конце из-за строгой последовательности действий. Стоимость внесения изменений высока, так как для ее инициализации приходится ждать завершения всего проекта. Тем не менее, фиксированная стоимость часто перевешивает минусы подхода. Исправление осознанных в процессе создания недостатков возможно, и, по нашему опыту, требует от одного до трех дополнительных соглашений к контракту с небольшим ТЗ.

Данная методология разработки ПО используется:

1. Только тогда, когда требования известны, понятны и зафиксированы. Противоречивых требований не имеется.
2. Нет проблем с доступностью программистов нужной квалификации.
3. В относительно небольших проектах.

«Incremental Model» (инкрементная модель)

В инкрементной модели полные требования к системе делятся на различные сборки. Терминология часто используется для описания поэтапной сборки ПО. Имеют место несколько циклов разработки, и вместе они составляют жизненный цикл «мульти-водопад». Цикл разделен на более мелкие легко создаваемые модули. Каждый модуль проходит через фазы определения требований, проектирования, кодирования, внедрения и тестирования. Процедура разработки по инкрементной модели предполагает выпуск на первом большом этапе продукта в базовой функциональности, а затем уже последовательное добавление новых функций, так называемых «инкрементов». Процесс продолжается до тех пор, пока не будет создана полная система.



Инкрементные модели используются там, где отдельные запросы на изменение ясны, могут быть легко формализованы и реализованы.

Данная методология разработки ПО используется:

1. Когда основные требования к системе четко определены и понятны. В то же время некоторые детали могут дорабатываться с течением времени.
2. Требуется ранний вывод продукта на рынок.
3. Есть несколько рисковых фич или целей.

**Вопрос 2. Процедурно и объектно-ориентированные языки программирования и среды разработки.**

Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Решение задачи в объектно-ориентированном стиле, применяя объектно-ориентированное программирование (ООП), оказывается ближе к реальности. Сложные задачи — это стихия ООП. В решении задачи участвуют программные объекты и ответственность за решение делится между ними. Часто, в процессе можно найти некоторые паттерны, поэтому при решении задачи некоторый участок кода используется многократно. Результат достигается путём распределения ответственности между объектами программы и повторным использования промежуточных решений.

Объектно-ориентированный язык программирования (ОО-язык) — язык, построенный на принципах объектно-ориентированного программирования.

В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Примеры языков: C++, C#, Java, JavaScript, Python, Swift, PHP, Objective-C.

Процеду́рное программи́рование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Процедурное программирование (ПП). Инструкции для решения задачи выполняются одна за другой, сверху вниз, иногда возникают изменения в их последовательности. Среди команд нельзя выделить приоритетные, для решения задачи они просто должны быть выполнены.

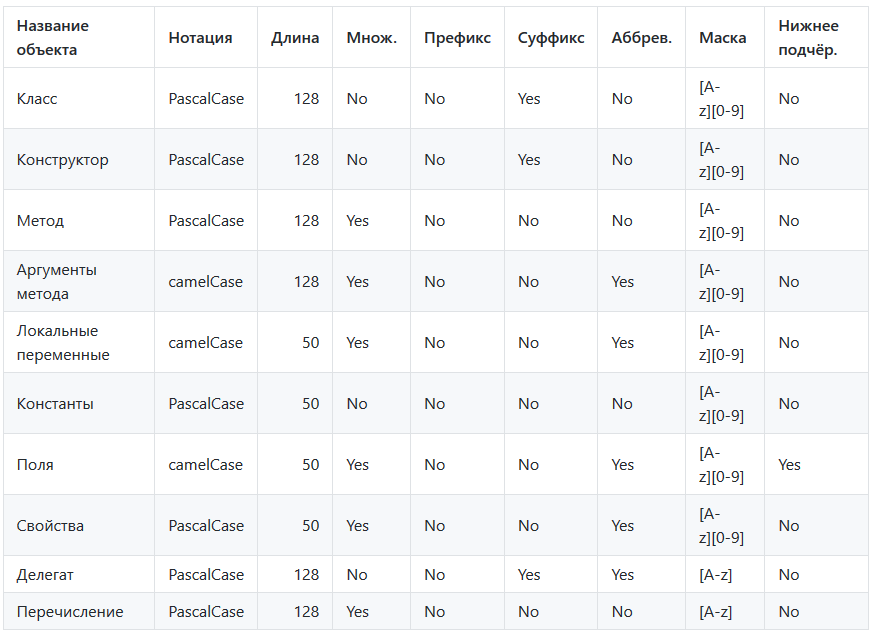
Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

Примеры языков: C, Go, Basic, Pascal.

Среды разработки: IntelliJ IDEA, NetBeans, Eclipse, Qt Creator, Geany, Embarcadero RAD Studio, Code::Blocks, Xcode, Microsoft Visual Studio.

**Вопрос 3. Соглашение о написании кода.**

Стандарт оформления кода (стандарт кодирования, стиль программирования) (англ. coding standards, coding convention или programming style) — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.



**Вопрос 4. Абстрактные классы.**

Абстрактный класс в объектно-ориентированном программировании — базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (и не содержать) абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание.

abstract class MyClass {…}

**Вопрос 5. Перечисления**

Перечисляемый тип (сокращённо перечисле́ние, англ. enumeration, enumerated type) — в программировании тип данных, чьё множество значений представляет собой ограниченный список идентификаторов.

enum Cardsuit { Clubs, Diamonds, Spades, Hearts }

**Вопрос 6. Создание классов. Элементы класса**

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

Экземпляры класса называются – объектами класса.

Классы ООП состоят из элементов различных типов:

Поля данных: хранят состояние класса с помощью переменных и структур.

Методы: подпрограммы для манипулирования указанными данными.

Некоторые языки допускают третий тип - свойства. Это что-то среднее между первыми двумя.

**Вопрос 7. Конструкторы и деструкторы класса**

Конструктор класса (от англ. constructor) — специальный блок инструкций, вызываемый при создании объекта. Записывается как метод, только с названием класса и без возвращаемого типа.

Дестру́ктор — специальный метод класса, служащий для деинициализации объекта (например освобождения памяти). Записывается так, как конструктор, только перед названием ставится тильда (~).

Однако на деле при очистке сборщик мусора вызывает не деструктор, а метод Finalize класса Person. Все потому, что компилятор C# компилирует деструктор в конструкцию. Метод Finalize уже определен в базовом для всех типов классе Object, однако данный метод нельзя так просто переопределить. И фактическая его реализация происходит через создание деструктора.

Когда наступает момент сборки мусора, сборщик видит, что данный объект должен быть уничтожен, и если он имеет метод Finalize, то он копируется в еще одну таблицу и окончательно уничтожается лишь при следующем проходе сборщика мусора. И здесь мы можем столкнуться со следующей проблемой: а что если нам немедленно надо вызвать деструктор и освободить все связанные с объектом неуправляемые ресурсы? В этом случае мы можем использовать второй подход - реализацию интерфейса IDisposable.

Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

**Вопрос 8. Создание объектов. Инициализаторы**

Как правило, после создания объекта, его нужно «привести» в некое начальное состояние, т.е. инициализировать.

- Первый случай — это простое создание объекта с вызовом конструктора без аргументов

SomeClass tmpObj = new SomeClass();

- Следующий вариант — это вызов конструктора с аргументами.

Person somePerson = new Person("Иван", 26);

- Третий вариант инициализации приходится использовать тогда, когда «базовый» вариант не походит (как и во втором случае), но в классе нет конструктора, удовлетворяющего всем потребностям.

Person somePerson = new Person("Иван", 26);

somePerson.Height = 190;

- Можно задать значения открытым свойствам сразу после создания объекта.

Person somePerson = new Person() { Name = "Иван", Age = 26, Height = 190 };

**Вопрос 9. Разделяемые классы и разделяемые методы**

Классы могут быть разделяемыми (частичными). То есть мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно. Для этого используется ключевое слово partial.

Такие классы могут иметь разделяемые методы. Причём определение такого метода может находиться в одном разделяемом классе, а реализация в другом.

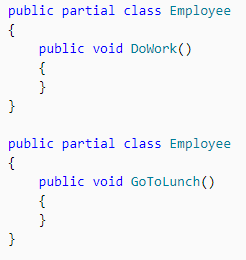
Можно разделить определение класса, структуры, интерфейса или метода между двумя или более исходными файлами. Каждый исходный файл содержит часть определения класса или метода, а во время компиляции приложения все части объединяются.

Существует несколько ситуаций, когда желательно разделение определения класса.

+ При работе над большими проектами распределение класса между различными файлами позволяет нескольким программистам работать с ним одновременно.

+ При работе с использованием автоматически создаваемого источника код можно добавлять в класс без повторного создания файла источника.

+ Чтобы разделить определение класса, используйте модификатор ключевого слова partial.



Ключевое слово partial указывает, что другие части класса, структуры или интерфейса могут быть определены в пространстве имен. Все части должны использовать ключевое слово partial. Для формирования окончательного типа все части должны быть доступны во время компиляции. Все части должны иметь одинаковые модификаторы доступа, например public, private и т. д.

**Вопрос 10. Структуры**

Структура – композитный тип данных, инкапсулирующий без сокрытия набор значений различных типов. Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Структура является типом значения, то есть не относится к ссылочным типам данных, как например класс, следовательно, она хранится в стеке. Структуры не поддерживают наследование. Многие примитивные типы, например, int, double и т.д., по сути являются структурами.

В отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении.

**Вопрос 11. Классы и пространства имён**

Пространства имен (namespace) — это способ, благодаря которому .NET избегает конфликтов имен между классами. Они предназначены для того, чтобы исключить ситуации, когда вы определяете класс, представляющий заказчика, называете его Customer, а после этого кто-то другой делает то же самое (подобный сценарий достаточно распространен).

Пространство имен — это не более чем группа типов данных, но дающая тот эффект, что имена всех типов данных в пределах пространства имен автоматически снабжаются префиксом - названием пространства имен. Пространства имен можно вкладывать друг в друга. Например, большинство базовых классов .NET общего назначения находятся в пространстве имен System. Базовый класс Array относится к этому пространству, поэтому его полное имя — System.Array.

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

Экземпляры класса называются – объектами класса.

Классы ООП состоят из элементов различных типов:

Поля данных: хранят состояние класса с помощью переменных и структур.

Методы: подпрограммы для манипулирования указанными данными.

Некоторые языки допускают третий тип - свойства. Это что-то среднее между первыми двумя.

**Вопрос 12. Понятие наследования. Основные решаемые задачи**

Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Наследование является механизмом повторного использования кода и способствует независимому расширению программного обеспечения через открытые классы и интерфейсы. Установка отношения наследования между классами порождает иерархию классов.

Например, если бы «наследования» не существовало, у процессора болталось бы без надобности поле, хранящее разрешение монитора по горизонтали и вертикали. Но, у нас-таки есть принцип наследования! Мы можем создать нечто абстрактное — компьютерную железяку, которая обладает общими характеристиками и поведением всех железяк. И на основе её создать множество классов, которые уже будут содержать более специфические характеристики и поведение. Мы можем продолжать плодить классы, пока у нас не получится специальный класс, который будет описывать центральный процессор и не содержать ничего лишнего. Однако, учтите, что бы вы не сделали с родительским классом, это отразится на всех его потомках в любом “колене”.

**Вопрос 13. Понятие инкапсуляции. Основные решаемые задачи**

Инкапсуляция (англ. encapsulation, от лат. in capsula) — упаковка данных и функций в единый компонент.

В общем случае в разных языках программирования термин «инкапсуляция» относится к одной или обеим одновременно следующим нотациям:

- механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;

- языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

Пренебрегая формализмом и способствуя интуитивному восприятию, инкапсуляцию можно определить с помощью латинского in capsula — размещение в оболочке, изоляция, закрытие чего-либо инородного с целью исключения влияния на окружающее, обеспечение доступности главного, выделение основного содержания путём помещения всего мешающего, второстепенного в некую условную капсулу (чёрный ящик). Однако данное определение является лишь приближением.

В C# часто для инкапсуляции используются свойства.

**Вопрос 14. Понятие полиморфизма. Основные решаемые задачи**

Полиморфизм — способность функции обрабатывать разные типа данных. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение:

- внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов, и их количеством);

- внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

**Вопрос 15. Основные возможности *Visual Studio***

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

**Вопрос 16. Шаблоны в *Visual Studio***

В Visual Studio есть набор стандартных шаблонов таких как: приложение WPF, WFA, ConsoleApp, библиотека классов (.NET Framework, Standart, Core, модульные тесты и т.п.

Также можно создавать свои шаблоны и экспортировать уже готовые.

**Вопрос 17. Структура проектов и решений *Visual Studio***

Работая с системой Visual Studio, пользователь открывает решение. При повторном редактировании специальных файлов создается временное решение, которое можно уничтожить по окончании работы. Однако решение позволяет управлять текущими файлами, поэтому в большинстве случаев его сохранение означает, что пользователь может вернуться к тому, что он делал накануне, и вновь открыть файлы, с которыми он работал.

Решение можно интерпретировать как контейнер связанных между собой проектов. Проекты внутри решения не обязательно должны быть написаны на одном и том же языке программирования или иметь одинаковый тип.

Наиболее распространенным способом структурирования приложений в среде Visual Studio является одно отдельное решение, содержащее много проектов. Каждый проект можно создать из набора исходных файлов и папок.

**Вопрос 18. Сервисные средства разработки приложений**

????

Возможно – это ConsoleApp и т.п.

**Вопрос 19. Отладка приложений в Visual Studio**

Отладка, точки останова, вывод значений переменных, пошаговое выполнение.

Вопрос 20. Платформа .NET Framework 4

Платформа .NET включает не только среду разработки для нескольких языков программирования, называемую Visual Studio.NET, но и множество других средств, например, механизмы поддержки баз данных, электронной почты и др.

Важнейшими задачами при создании программ являются:

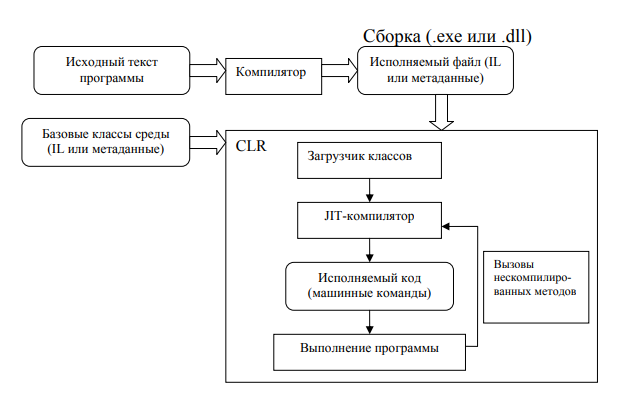
*-переносимость* – возможность выполнения на различных типах компьютеров;

*-безопасность* – невозможность несанкционированных действий;

*-надёжность* – способность выполнять необходимые функции в определённых условиях;

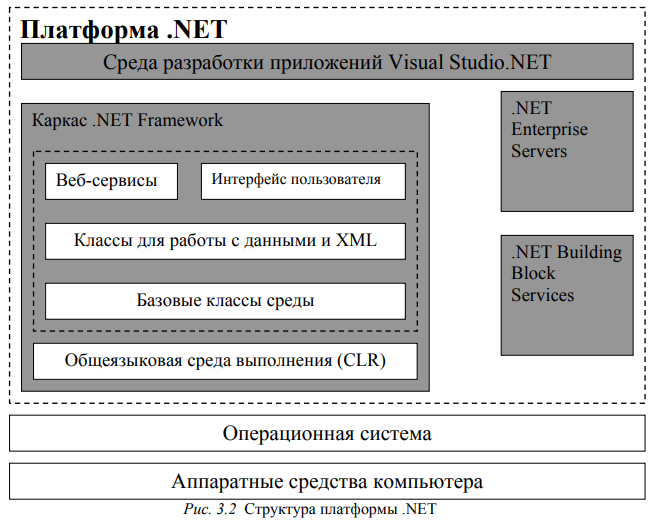
*-использование готовых компонентов* – для ускорения разработки; -межъязыковое взаимодействие – возможность применять одновременно несколько языков программирования.

В состав платформы .NET для обеспечения переносимости входят компиляторы, переводящие программу не в машинные коды, а в промежуточный язык (Microsoft Intermediate Language, MSIL, или просто IL), который не содержит команд, зависящих от языка, операционной системы и типа компьютера. Программа на этом языке выполняется не самостоятельно, а под управлением системы, которая называется общеязыковой средой выполнения (Common Language Runtime, CLR). Среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JITкомпилятор, переводящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. JIT означает «just in time», что можно перевести так «вовремя», то есть компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в ­ЭШе (область ОП для временного хранения информации) для дальнейшего использования.



Компилятор в качестве результата своего выполнения создаёт так называемую сборку – файл с расширением exe или dll, который содержит код на языке IL и метаданные.

Платформа .NET содержит огромную библиотеку классов, которые можно использовать при программировании на любом языке .NET. Библиотека имеет несколько уровней. На самом нижнем находятся базовые классы среды, которые используются при создании любой программы. Над этим слоем находится набор классов, позволяющий работать с базами данных и XML. Классы самого верхнего уровня поддерживают разработку распределенных приложений, а также веб- и Windowsприложений. Программа может использовать классы любого уровня. Библиотека классов вместе с CLR образуют каркас (framework), то есть основу платформы.



**Вопрос 21. Сборки в .NET**

Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является файл exe или dll (в зависимости от выбранных настроек), который называется сборкой приложения. Сборка является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Сборки кристаллизуют всю библиотеку классов .NET - при написании кода и создании сборки своего приложения мы используем пространства имен, которые размещены в других сборках .NET.

Сборки имеют следующие составляющие:

- Манифест, который содержит метаданные сборки

- Метаданные типов. Используя эти метаданные, сборка определяет местоположение типов в файле приложения, а также места размещения их в памяти

- Собственно код приложения на языке MSIL, в который компилируется код C#

- Ресурсы

Все эти компоненты могут находиться в одном файле, и тогда сборка представляет один единственный файл в формате exe или dll.

**Вопрос 22. Компоненты CLR (.Net Framework)**

Ядром платформы .NET Framework является общеязыковая исполняющая среда (Common Language Runtime) или сокращенно CLR.

На компьютере храниться в виде файла динамической библиотеки mscoree.dll, в проекте данная сборка подключается автоматически.

Какие задачи решает CLR?

- Загрузка пользовательских сборок

- Управление памятью

- Работа с потоками

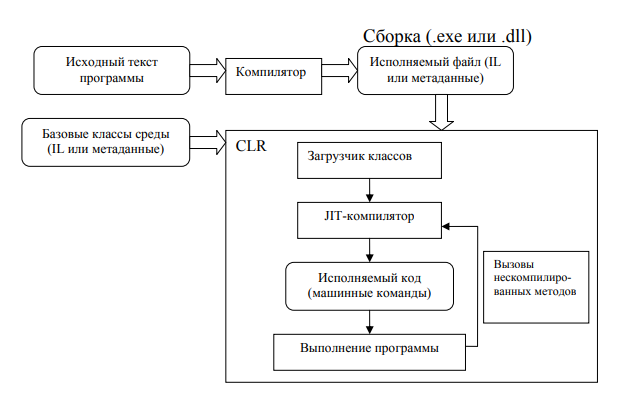
- Безопасность выполнения кода

- Очистка памяти (сборщик мусора)

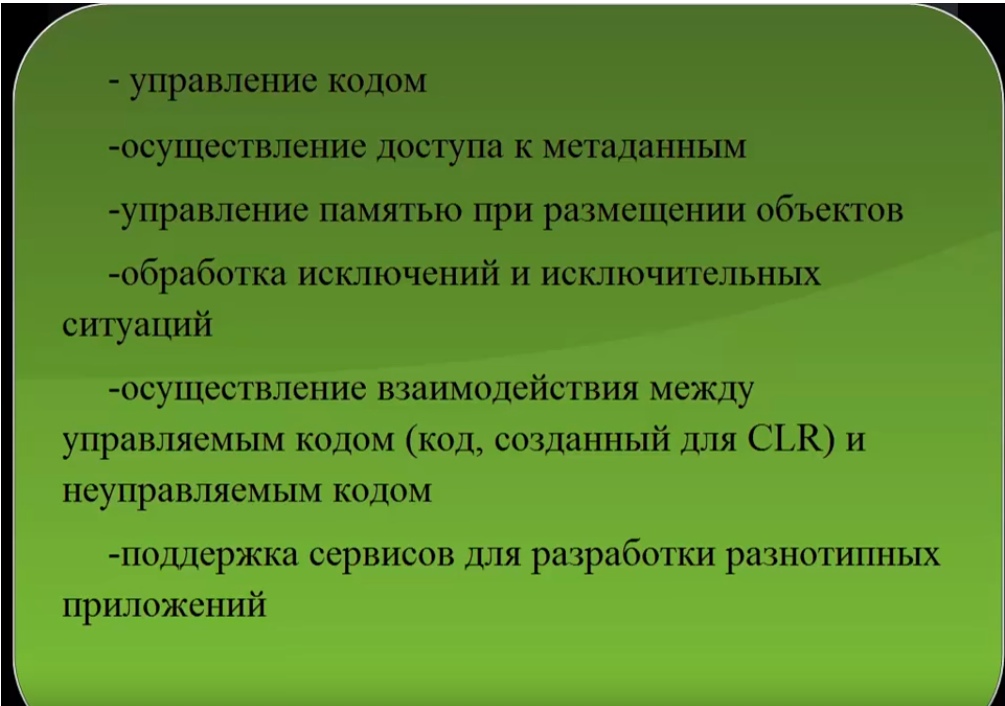
- и многое другое

CLR компилирует код приложения на языке CIL (реализация компиляции которого компанией Microsoft называется MSIL) во время его исполнения, а также предоставляет MSIL-программам (а следовательно, и программам, написанным на языках высокого уровня, поддерживающих .NET Framework) доступ к библиотеке классов .NET Framework, или так называемой .NET FCL (англ. Framework Class Library).

Среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JITкомпилятор, переводящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. JIT означает «just in time», что можно перевести так «вовремя», то есть компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в ­ЭШе (область ОП для временного хранения информации) для дальнейшего использования.



Компилятор в качестве результата своего выполнения создаёт так называемую сборку – файл с расширением exe или dll, который содержит код на языке IL и метаданные.



**Вопрос 24. Автодокументирование**

Генератор документации — программа или пакет программ, позволяющая получать документацию, предназначенную для программистов (документация на API) и/или для конечных пользователей системы, по особому образу комментированному исходному коду и, в некоторых случаях, по исполняемым модулям (полученным на выходе компилятора).

**Вопрос 25. Переменные и типы данных. Неявно типизированные переменные. Тип dynamic.**

Типы значений

Простые типы: sbyte, short, int, long – со знаком целочисленные; byte, ushort, uint, ulong – без знака; сhar – символы Юникода; float, double – вещественные; decimal – с повышенной точностью; bool – логическое.

Перечисление: enum.

Структуры: struct.

Ссылочные типы

Типы классов: object – исходный базовый класс; string – строки Юникода; class C {…} – пользовательский.

Типы интерфейса: interface I {...}.

Типы массивов: int[] или int[, ].

Типы делегатов: delegate int D(...).

Неявно типизированная переменная – var. Она получает определённый тип при присваивании значения, дальше этот тип не изменяется.

Тип dynamic – почти как var, но dynamic переменная не строго типизирована на протяжении всего времени её существования.

**Вопрос 26. Область видимости переменной**

Локальная, Глобальная, Пакет или пространство имён, также private, protected, public, internal (в сборке).

**Вопрос 27. Преобразование типов данных**

- Неявное преобразование (implicit conversion), для корректности проверяются типы (не значения).

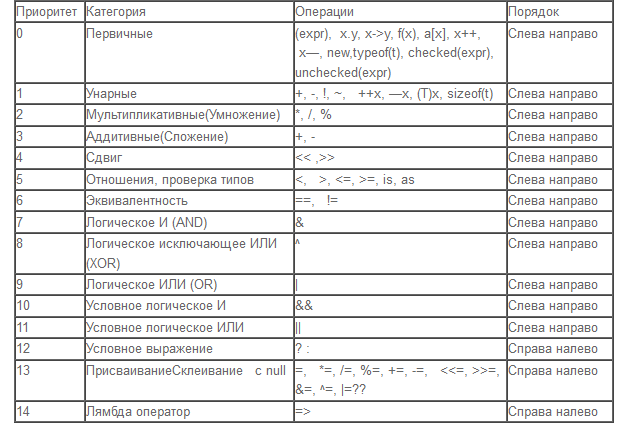
- Явное преобразование (explicit conversion), если сужающее, то можно использовать проверку переполнения с помощью ключевого слова checked, если переполнение произошло, то выбрасывает OverflowException.

**Вопрос 28. Переменные только для чтения и константы**

Константа (const) проинициализирована при объявлении и её значение не может быть изменено.

Только для чтения (readonly) может быть проинициализирована при объявлении либо в конструкторе класса, изменять после инициализации нельзя.

**Вопрос 29. Выражения и операции. Приоритет операций**



**Вопрос 30. Оператор выбора if**

if (условие)

оператор (операторы)

else

оператор (операторы)

**Вопрос 31. Оператор switch**

switch(выражение) {

case константа1:

последовательность операторов

break;

case константа2:

последовательность операторов

break;

case константаЗ:

последовательность операторов

break;

...

default:

последовательность операторов

break;

}

**Вопрос 32. Операторы цикла**

for (int i = 0; i < 9; i++)

{

Console.WriteLine($"Квадрат числа {i} равен {i\*i}");

}

int i = 6;

while (i > 0)

{

Console.WriteLine(i);

i--;

}

int i = 6;

do

{

Console.WriteLine(i);

i--;

}

while (i > 0);

foreach (тип имя\_переменной\_цикла in коллекция)

оператор;

break – для выхода из цикла, continue – для пропуска итерации.

**Вопрос 33. Массивы**

Структура данных в виде набора компонентов (элементов **массива**), расположенных в памяти непосредственно друг за другом

int[] arr = new int[133];

int[] arr = new int[] {12,23,34,45, 56};

int[] arr = {0,4,8,32};

int[] arr = new int[3] {-5,2,3};

var varArr = new[] { "123", null, "exe" };

**Вопрос 34. Неявно типизированные массивы**

var varArr = new[] { "file", null, "apricot" };

dynamic varArr = new[] { "file", null, "apricot" };

**Вопрос 35. Класс Array**

Каждый создаваемый массив получает большую часть функциональности от класса System.Array. Общие члены этого класса позволяют работать с массивом с использованием полноценной объектной модели. Таким образом, методы и свойства, определенные в классе Array, можно использовать с любым массивом C#. Давайте разберем некоторые полезные особенности класса Array.

Класс Array является абстрактным, поэтому создать массив с использованием какого-либо конструктора нельзя. Однако вместо применения синтаксиса C# для создания экземпляров массивов также возможно создавать их с помощью статического метода CreateInstance(). Это исключительно удобно, когда заранее неизвестен тип элементов массива, поскольку тип можно передать методу CreateInstance() в параметре как объект Type.

// Создаем массив типа string, длиной 5

Array myArr = Array.CreateInstance(typeof(string),5);

// Инициализируем первые два поля массива

myArr.SetValue("Name",0);

myArr.SetValue("Age",1);

// Считываем данные из массива

string s = (string)myArr.GetValue(1);

**Вопрос 36. Операторы break и continue**

break – для выхода из цикла, continue – для пропуска итерации.

**Вопрос 37. Символы и строки.**

Char и string – в кодировке Unicode

**Вопрос 38. Класс String**

В языке C# строковые значения представляет тип string, а вся функциональность работы с данным типом сосредоточена в классе System.String. Собственно, string является псевдонимом для класса System.String. Объекты этого класса представляют текст как последовательность символов Unicode. Максимальный размер объекта String может составлять в памяти 2 ГБ, или около 1 миллиарда символов.

string s1 = "hello";

string s2 = null;

string s3 = new String('a', 6); // результатом будет строка "aaaaaa"

string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

Compare: сравнивает две строки с учетом текущей культуры (локали) пользователя

CompareOrdinal: сравнивает две строки без учета локали

Contains: определяет, содержится ли подстрока в строке

Concat: соединяет строки

CopyTo: копирует часть строки или всю строку в другую строку

EndsWith: определяет, совпадает ли конец строки с подстрокой

Format: форматирует строку

IndexOf: находит индекс первого вхождения символа или подстроки в строке

Insert: вставляет в строку подстроку

Join: соединяет элементы массива строк

LastIndexOf: находит индекс последнего вхождения символа или подстроки в строке

Replace: замещает в строке символ или подстроку другим символом или подстрокой

Split: разделяет одну строку на массив строк

Substring: извлекает из строки подстроку, начиная с указанной позиции

ToLower: переводит все символы строки в нижний регистр

ToUpper: переводит все символы строки в верхний регистр

Trim: удаляет начальные и конечные пробелы из строки

**Вопрос 39. Класс StringBuilder**

Динамическая строка StringBuilder. Операции не создают новые строки.

Метод Append - этот метод добавляет к строке подстроку.

Кроме метода Append класс StringBuilder предлагает еще ряд методов для операций над строками:

Insert: вставляет подстроку в объект StringBuilder, начиная с определенного индекса

Remove: удаляет определенное количество символов, начиная с определенного индекса

Replace: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку

AppendFormat: добавляет подстроку в конец объекта StringBuilder

**Вопрос 40. Регулярные выражения**

Регуля́рные выраже́ния (англ. regular expressions) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. wildcard characters). Для поиска используется строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.

Некоторые элементы синтаксиса регулярных выражений:

^: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")

$: конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)

.: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")

\*: предыдущий символ повторяется 0 и более раз

+: предыдущий символ повторяется 1 и более раз

?: предыдущий символ повторяется 0 или 1 раз

\s: соответствует любому пробельному символу

\S: соответствует любому символу, не являющемуся пробелом

\w: соответствует любому алфавитно-цифровому символу

\W: соответствует любому не алфавитно-цифровому символу

\d: соответствует любой десятичной цифре

\D : соответствует любому символу, не являющемуся десятичной цифрой

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex.

**Вопрос 41. Классы Regex, Match, MatchCollection**

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else

{

Console.WriteLine("Совпадений не найдено");

}

Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений (MatchCollection).

Каждый элемент такой коллекции представляет объект Match. Его свойство Value возвращает найденное совпадение.

Проверка на соответствие строки формату IsMatch.

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой.

**Вопрос 42. Методы. Создание и вызов метода**

[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])

{

// тело метода

}

**Вопрос 43. Перегруженные методы**

Перегруженные методы должны отличаться по:

- Количеству параметров

- Типу параметров

- Порядку параметров

- Модификаторам параметров

Add(int a, int b)

Add(int a, int b, int c)

Add(int, int, int, int)

Add(double, double)

Add(int, double)

Add(double, int)

Add(ref int, ref int)

**Вопрос 44. Массивы параметров методов**

params

static void Addition( int x, string mes, params int[] integers)

{}

**Вопрос 45. Необязательные параметры и именованные аргументы**

// Аргументы b и с указывать при вызове необязательно

static int mySum(int a, int b = 5, int c = 10)

{

return a + b + c;

}

// Использование именованных аргументов

// при вызове метода

int sum1 = mySum(a: 3, b: 10);

**Вопрос 46. Выходные параметры методов**

Параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out.

static void Sum(int x, int y, out int a)

{

a = x + y;

}

int z;

Sum(x, 15, out z);

**Вопрос 47. Ссылочные и значимые типы. Передача значимого типа по ссылке**

Типы значений:

Целочисленные типы (byte, sbyte, char, short, ushort, int, uint, long, ulong)

Типы с плавающей запятой (float, double)

Тип decimal

Тип bool

Перечисления enum

Структуры (struct)

Ссылочные типы:

Тип object

Тип string

Классы (class)

Интерфейсы (interface)

Делегаты (delegate)

С помощью ref можно передать переменную по ссылке, работает, как и out, но с ref всегда нужно использовать инициализированные переменные, иначе будет ошибка.

Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещаются поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

static void Calculate(int t)

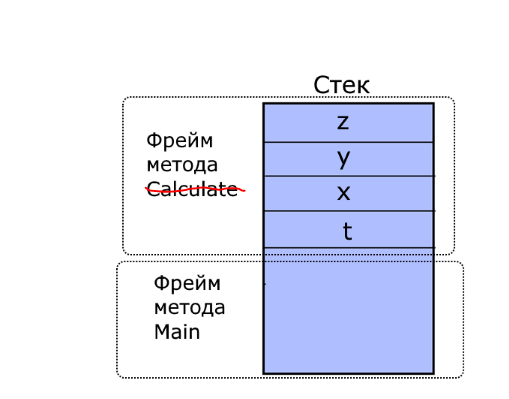
    {

        int x = 6;

        int y = 7;

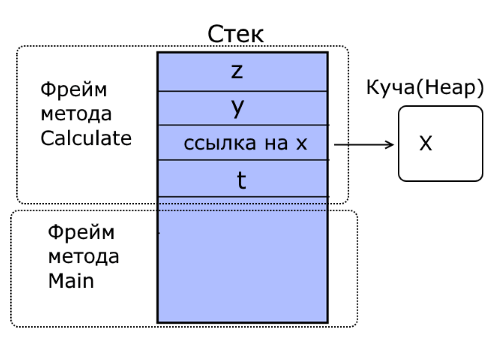
        int z = y + t;

    }



Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, то ссылка из стека удаляется, и память очищается. После этого в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, и удаляет этот объект и очищает память.



**Вопрос 48. Упаковка и распаковка**

- Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе).

- Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

**Вопрос 49. Обнуляемые типы**

Значение null по умолчанию могут принимать только объекты ссылочных типов. Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null. Для этого надо использовать знак вопроса ? после типа значений.

int? z1 = 5;

bool? enabled1 = null;

Double? d1 = 3.3;

Но фактически запись ? является упрощенной формой использования структуры System.Nullable<T>. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в конкретной задача уже подставляется конкретный тип данных.

Nullable<int> z2 = 5;

Nullable<bool> enabled2 = null;

Nullable<System.Double> d2 = 3.3;

Если использовать ссылочный тип (класс), то будет ошибка.

Вопрос 50. Рефакторинг. Модульное тестирование

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Рефакторинг (англ. refactoring), или перепроектирование кода, переработка кода, равносильное преобразование алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Доп. инф.

-----------------------------------------------------------------------------------------------------------

**Рефлексия** представляет собой процесс выявления типов во время выполнения приложения. Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. И рефлексия как раз и позволяет определить все эти составные элементы приложения.

Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой
* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

Эти классы представляют составные блоки типа и приложения: методы, свойства и т.д. Но чтобы получить информацию о членах типа, нам надо воспользоваться классом System.Type.

Класс System.Type представляет изучаемый тип, инкапсулируя всю информацию о нем.

Чтобы управлять типом и получать всю информацию о нем, нам надо сперва получить данный тип. Это можно сделать тремя способами: с помощью ключевого слова typeof, с помощью метода GetType() класса Object и применяя статический метод Type.GetType().

Для управления сборками в пространстве имен System.Reflection имеется класс **Assembly**. С его помощью можно загружать сборку, исследовать ее.

С помощью динамической загрузки мы можем реализовать технологию позднего связывания. **Позднее связывание** позволяет создавать экземпляры некоторого типа, а также использовать его во время выполнения приложения.

Использование позднего связывания менее безопасно в том плане, что при жестком кодировании всех типов (ранее связывание) на этапе компиляции мы можем отследить многие ошибки. В то же время позднее связывание позволяет создавать расширяемые приложения, когда дополнительный функционал программы неизвестен, и его могут разработать и подключить сторонние разработчики.

Ключевую роль в позднем связывании играет класс **System.Activator**. С помощью его статического метода **Activator.CreateInstance()**можно создавать экземпляры заданного типа.

**Вопрос 51. Класс Assert**

Класс Assert из пространства имен Microsoft.VisualStudio.TestTools.UnitTesting с помощью своих статических методов позволяет верифицировать результат выполнения некоторого действия. Ранее уже было рассмотрено несколько методов, в частности, метод Assert.IsNotNull(), проверяющий, не равен ли некоторый объект значению null. Кроме того, при тестировании нам доступен еще ряд методов:

AreEqual(object expected, object actual): проверяет, равны ли оба объекта. Имеет различные перегруженные версии, позволяющие сравнивать различные типы объектов

AreEqual<T>(T expected, T actual): обобщенная версия предыдущего метода. Например, Assert.AreEqual<string>("Index", result.MasterName)

AreNotEqual(object expected, object actual): проверяет, не равны ли оба объекта. Тест проходит успешно, если объекты не равны

AreNotEqual<T>(T expected, T actual): обобщенная версия предыдущего метода

AreSame(object expected, object actual): проверяет, указывают ли оба объекта на один и тот же объект в памяти

AreNotSame(object expected, object actual): проверяет, указывают ли оба объекта на разные объекты в памяти. Если они указывают на один и тот же объект, то тест заканчивается неудачно

Equals(object objA, object objB): проверяет на равенство оба объекта

IsFalse(bool condition): проверяет, равно ли условие condition значению false

IsTrue(bool condition): проверяет, равно ли условие condition значению true

IsNull(object value): проверяет, имеет ли объект value значение null

IsInstanceOfType(object value, Type expectedType): проверяет, представляет ли объект value тип expectedType

**Вопрос 52. Инкапсуляция. Модификаторы доступа**

public: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.

private: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.

protected: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.

internal: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).

protected internal: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

private protected: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal. Интерфейсы public.

**Вопрос 53. Свойства класса. Автоматические свойства**

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойства. Они обеспечивают простой доступ к полям класса, узнать их значение или выполнить их установку.

[модификатор\_доступа] возвращаемый\_тип произвольное\_название

{

// код свойства

}

Автоматические свойства: public string Name { get; set; }

**Вопрос 54. Индексаторы**

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

возвращаемый\_тип this [Тип параметр1, ...]

{

get { ... }

set { ... }

}

**Вопрос 55. Статические поля и методы**

Статические (static) поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

**Вопрос 56. Статические классы и статические конструкторы**

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров

- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса

- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз.

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы.

**Вопрос 57. Методы расширения. Перегрузка операций.**

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

public static class StringExtension

{

public static int WordCount(this string str, char c)

{

int counter = 0;

for (int i = 0; I <str.Length; i++)

{

if (str[i] == c)

counter++;

}

return counter;

}

}

Вызываем метод таким способом:

int i = s.WordCount(c);

Перегрузка операций (операторов):

public static возвращаемый\_тип operator оператор(параметры)

{ }

Можно перегрузить следующие операторы:

унарные операторы +, -, !, ~, ++, --

бинарные операторы +, -, \*, /, %

операции сравнения ==, !=, <, >, <=, >=

логические операторы &&, ||

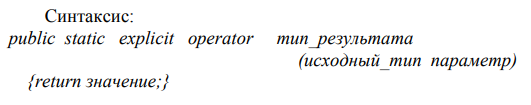
операторы присваивания +=, -=, \*=, /=, %=

**Вопрос 58. Операции преобразования**

Эти операции преобразуют объект некоторого класса в значение другого типа. Фактически, операция преобразования перегружает операцию приведения типов.

Существуют две формы операторов преобразования: явная и неявная.

Явная



Операция выполняет преобразование из типа параметра в тип результата. Одним из этих типов должен быть класс, для которого определяется операция. Нельзя определять преобразование к типу object и наоборот. Явное преобразование выполняется при использовании операции приведения типа. Например, определим в классе Student операцию преобразования студента (т.е. объекта класса) к строковому типу.

Неявная форма

*public static implicit operator тип\_результата*

*(исходный\_тип параметр)*

*{return значение;}*

Преобразование выполняется автоматически в следующих случаях:

• при присваивании объекта переменной, тип которой совпадает с типом результата;

• при передаче объекта в метод на место параметра с типом результата;

• при использовании объекта в выражении, содержащем переменные, тип которых совпадает с типом результата;

• при явном приведении типа.

Для одной и той же пары типов, участвующих в преобразовании, нельзя определить одновременно обе формы операции преобразования.

Например, дополним класс Student операцией преобразования строки в студента в неявной форме.

**Вопрос 59. Интерфейсы. Явная и неявная реализация интерфейса.**

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I.

В целом интерфейсы могут определять следующие сущности:

- Методы

- Свойства

- Индексаторы

- События

Однако интерфейсы не могут определять статические члены, переменные, константы.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Неявная имплементация (реализация) всегда является открытой (public), поэтому к методам и свойствам можно обращаться напрямую.

var imp = new ImplicitTest();

imp.DoTest();

Явная имплементация всегда закрыта (private).

Чтобы получить доступ к имплементации необходимо кастовать инстанцию класса к интерфейсу (upcast to interface).

var exp = new ExplicitTest();

((ITest)exp).DoTest();

**Вопрос 60. Переопределение и сокрытие методов**

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы и свойства называют виртуальными.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Фактически сокрытие представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для сокрытия членов класса применяется ключевое слово new.

Если метод в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нас не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

**Вопрос 61. Определение герметизированных классов и методов**

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором sealed.

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

Пример:

public override sealed void Display()

{

Console.WriteLine($"{FirstName} {LastName} работает в {Company}");

}

Также можно использовать модификатор sealed для классов.

**Вопрос 62. Вызов методов и конструкторов базового класса**

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова base мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public void Display()

{

Console.WriteLine(Name);

}

}

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

}

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

public override void GetInfo()

{

// Вызов метода GetInfo из базового класса

base.GetInfo();

Console.WriteLine("Employee ID: {0}", id);

}

**Вопрос 63. Присваивание и ссылка на классы в иерархии наследования и через интерфейс**

public interface A

{

int Sum();

}

// Унаследованный интерфейс

public interface B : A

{

int Del();

}

class MyOperation : B

{

int x = 10, y = 5;

public int Sum()

{

return x + y;

}

public int Del()

{

return x / y;

}

}