Автор сего опуса снимает с себя всякую ответственность за то, что может с Вами случиться после прочтения.
Также убедительная просьба в случае ошибок или неточностей обращаться по известному Вам адресу.

~~~~~~

1. JIT-компиляция. Переносимость Java-программ:

   JIT-компиляция(Just in time) – компилятор Java порождает независимый от операционной системы и аппаратной платформы байт-код, который транслируется в машинный код по ходу выполнения программы. Благодаря этому достигается высокая скорость выполнения программы за счет увеличения затрат памяти и уменьшения скорости компиляции;

   Переносимость Java-программ: виртуальные машины Java(связка JIT – компилятор, run-time, Библиотеки классов) существуют практически для всех аппаратных платформ и операционных систем, перенос программы на другую платформу не требует ее перекомпиляции. То есть, написанное один раз работает на всех платформах.

2. Установка JDK на компьютер под управлением ОС Linux:

   Зайти на сайт Oracle, согласиться там с лицензионным соглашением и скачать оттуда пакет JDK в соответствие с разрядностью системы. Затем разархивировать полученный пакет в домашний каталог и добавить в .profile (.bashrc):

   ```
   export JAVA_HOME =$HOME/jdk1.8.0 _31
   export PATH=$JAVA_HOME /bin:$PATH
   ```

   В конце концов перезагрузиться или перелогиниться. PROFIT.

3. Компиляция и запуск Java-программ:

   javac Name.java – компиляция
   java Name – запуск

4. Понятие объекта. Характеристика объекта, позволяющая организовать позднее связывание при вызове методов:

   Объект — это самоописывающая структура данных, обладающая внутренним состоянием и способная обрабатывать передаваемые ей сообщения.

   Раннее связывание, как было отмечено выше, происходит на этапе компиляции. Оно применяется при вызове обычных методов (не виртуальных).

   Позднее связывание(Late binding) происходит во время выполнения программы. Выполняется оно при вызове виртуальных функций класса-потомка для определения того, какой именно метод следует вызывать.

   Исходя из того, что раннее связывание выполняется на этапе компиляции, а позднее - в рантайме, первый вариант обладает лучшим быстродействием, однако второй необходим для реализации полиморфизма.

   В Java ко всем методам по умолчанию применяется позднее связывание (если они не помечены модификатором final) в отличие от, скажем, C++, где по умолчанию применяется раннее связывание. В контексте C++ можно рассказать про таблицу виртуальных методов.

5. Понятие инкапсуляции. Модификаторы доступа:

   Инкапсуляция — один из основных китов(принципов) объектно-ориентированного программирования, который заключается в том, что доступ к членам класса извне осуществляется только через механизм передачи сообщений.

   Перед объявлением члена класса идет модификатор доступа, управляющий доступом к члену:

   - private – только из тела класса;
   - public – из любого места;
   - protected – для самого класса и классов из того же пакета, а также для наследников класса;
~~~~~~

- без модификатора — для самого класса и классов из того же пакета.

6. Понятие класса. Синтаксис объявления класса в Java. Категории членов класса:
   Объект является самоописывающей структурой, поэтому он содержит информацию о классе, к которому принадлежит.
   Класс — тип данных, значениями которого являются объекты, имеющие сходное внутреннее состояние и обрабатывающие одинаковый набор сообщений.
   Класс можно рассматривать как шаблон, порождающий объекты. Из-за этого объекты можно назвать экземплярами класса.
   В Java все значения являются классами, кроме значений примитивных типов(char, byte, short, int, long, float, double, boolean). К тому же int можно применить обертку класса Integer.
   Классы делятся на публичные и непубличные, в каждом файле может быть только один публичный класс(имя файла должно совпадать с именем этого класса). А непубличных классов может быть несколько, они видны только внутри файла, в котором объявлены.
   public class Cat {} | class AnotherCat {}
   Внутри фигурных скобок объявляются члены класса:
   - экземплярные поля(отвечают за хранение внутреннего состояния объектов)
   - статические поля(хранят данные, одинаковые для всех объектов)
   - экземплярные методы(отвечают за обработку передаваемых объекту сообщений)
   - статические методы(выполняют операции, для которых не нужен доступ к внутреннему состоянию объектов)
   - экземплярные конструкторы(инцилизируют только что созданные объекты)
   - статический конструктор(набор Static-блоков, инициализирует статические поля класса)
   - вложенные классы(представляют собой объекты необходимые для реализации данного класса)
7. Операция доступа к членам класса в Java. Обращение к полям и вызов методов:
   В общем случае для обращения к членам класса используется бинарная операция «.»
   cat.setName(“Мурка”); // обращение к экземплярному методу
   Cat.count++; // обращение к статическому полю
8. Понятие экземплярного поля, объявление экземплярных полей:
   Экземплярное поле — именнованая составная часть внутреннего состояние объекта. Объявление происходит как объявление структур в Си, каждое объявление предваряется модификатором доступа.
   Обращение происходит по ссылке на конкретный объект:
   System.out.println(redCat.color);
9. Понятие статического поля, объявление статических полей:
   Статическое поле — поле, разделяемое всеми объектами данного класса. Такие поля объявляются с модификатором static.
   Обращение происходит через класс:
   System.out.println(Cat.count);
10. Понятие экземплярного метода. Объявление и вызов экземплярных методов в Java:
    Экземплярный метод — подпрограмма, производящая обработку переданного объекту сообщения. Передача сообщения сводится к вызову конкретного экземплярного метода. Суть в том, что экземплярный метод имеет доступ к внутреннему состоянию объекта, то есть может читать и изменять его. Это происходит за счет передачи в метод ссылки на объект, в java – this, так как ссылка передается неявно. (например, движение точки)
11. Понятие статического метода. Объявление и вызов статических методов в Java:
    Статический метод — метод, не имеющий доступа к внутреннему состоянию объекта.

Такой метод объявляется с модификатором static. (например, какой-то вывод или сравнение двух значений)

12. Понятие сигнатуры методов. Перегрузка методов:
    Сигнатура метода — информация о количестве и типах формальных параметров методов. Если у метода нет параметров, он имеет пустую сигнатуру.
    Перегрузка метода — это объявление двух или более одноименных методов, имеющих разные сигнатуры.
    Решение о том, какой будет вызван метод происходит на этапе компиляции путем сопоставления типов параметров вызываемого метода с сигнатурами методов, имеющих такое же имя, что и вызываемый.
13. Понятие раннего и позднего связывания. Виртуальные методы:
    Раннее связывание — определение адреса вызова экземплярного метода во время компиляции программы, в то время как под поздним связыванием подразумевается выбор экземплярного метода во время выполнения программы, на основе информации о классе объекта.
    Методы, для вызовов который применяется позднее связывание — виртуальные. В Java все методы являются виртуальными.
14. Понятие экземплярного конструктора, его объявление. Конструктор по умолчанию:
    Экземплярный конструктор — экземплярный метод, предназначенный для инициализации только что созданного объекта. Главным образом состоит в заполнении информации о классе, к которому принадлежит объект. Операция создания объекта объединена с вызовом конструктора, что гарантирует, что объект будет инициализирован. Фактически не имеют имен, могут быть перегружены.
    Конструктор по умолчанию — экземплярный конструктор с пустой сигнатурой. Автоматически создается компилятором, если не определен никакой иной конструктор.
15. Размещение объектов в памяти. Объектные ссылки:
    Все объекты могут располагаться только в динамической памяти(куче) в отличие от значений примитивных типов. Объекты не могут вкладываться друг в друга, то есть не могут лежать в локальных переменных и параметрах методов, в полях объектов и элементах массивов. Вместо всего этого хранятся указатели на объекты, которые в Java называются объектными ссылками.
    Тем самым не требуется различать объект и ссылку на него. Стоит заметить, что значения примитивных типов не могут располагаться на верхнем уровне кучи.
16. Создание объектов и массивов, массивовые литералы:
    Создание объекта производится с помощью операции «new»:
        new name(fact_constructor_params)
    Массивы — тоже объекты, они могут жить только на верхнем уровне кучи.
    Для создания массива применяется особая форма оператора «new»:
    new type[size].
    Тип массива записывается в виде: type[].
    Есть специальная форма для создания инициализированного массива(массивовый литерал):
        new type[]{value1, value2, …}
    Они могут применяться не только в объявлениях но и, например, при вызове метода.
17. Статический конструктор. Static – блоки:
    Статический конструктор — статический метод, предназначенный для инициализации статических полей класса. Он не имеет параметров и вызывается до вызова любого статического метода или конструктора класса.
    Как правило они не слишком нужны, так как статические поля можно инициализировать прямо при их объявлении.
    В Java объявление класса может содержать один или несколько статических блоков

формата:

```
static {
    operator_sequence
}
```

Они выполняются при первом обращении к классу, то есть, при создании объекта, вызове статического метода, обращении к статическому полю.

Если таких блоков несколько, то они будут выполняться в том порядке, в котором перечислены. Их совокупность играет роль статического конструктора.

18. Понятие подтипа. Явная и неявная субтипизация:

Тип данных A является подтипом для типа данных B, если код, рассчитанных на обработку значений типа B может также корректно использоваться для обработки значений типа A.

Субтипизация — свойство языков программирования, обозначающее возможность использования подтипов в программах.

Различают явную и неявную субтипизацию.

В языке программирования с неявной субтипизацией решение о том, будет ли тип являться подтипом другого типа принимается на основе анализа структуры значений этих типов. Иначе это называется структурной типизацией.

Для явной субтипизации тип A является подтипом B только в случае, если A является наследником B. Что и происходит в Java.

19. Понятие наследования. Синтаксис наследования. Вызов конструктора базового класса из конструктора производного класса:

Наследование — способ получения нового класса на основе уже существующего класса, сочетающий в себе усложнение внутреннего состояния объектов, расширение ассортимента обрабатываемых сообщений и изменение реакции на некоторые сообщения.

Наследник называется производным классом или подклассом, базовый класс также называется суперклассом.

При наследовании подкласс получается все экземплярные поля и методы базового класса, кроме конструкторов. Тело любого конструктора производного класса должно начинаться с вызова одного из конструкторов базового класса.

```
Class cat extends Animal {
    ...
}
```

По умолчанию каждый класс является наследником встроенного класса Object.

Если конструктор базового класса является конструктором по умолчанию, то его вызов добавляется компилятором в конструктор производного класса автоматически, в ином случае это нужно сделать вручную, с помощью оператора super(fact_params).

20. Операция динамического приведения типа:

Операция динамического приведения объектной ссылки obj к типу T проверяет, является ли тип T одним из типов объекта obj и возвращает obj, если является. В противном случае операция порождает исключение ClassCastException.

```
(T)obj
```

21. Понятие переопределения метода. Переопределение методов в Java:

Переопределение метода(override) — замена тела экземплярного метода базового класса в производном классе, позволяющее объекту подкласса обрабатывать иначе то же сообщение.

Разница между перегрузкой и переопределением состоит в том, что при переопределении в классе не появляется новый метод, просто заменяется реализация метода из базового класса. Любой метод может быть переопределен.

22. Абстрактный метод и абстрактный класс, их объявление:

Абстрактный метод — не имеющий тела виртуальный метод, который должен быть

переопределен в производных от него классах. В Java объявляется с модификатором abstract, вместо тела метода ставится точка с запятой.

Абстрактный класс имеет абстрактные методы объявленные в нем самом или унаследованные от базового класса и не переопределенные. Такой класс также объявляется с модификатором abstract, создавать экземпляры таких классов запрещено.

23. Интерфейс, его объявление, наследование интерфейсов, реализация:

Интерфейс — тип данных, представляющий собой набор абстрактных методов. Он служит «контрактом» между разработчиком класса и кода, который использует данный класс. Аналог интерфейса — чистый абстрактный класс, в котором отсутствуют экземплярные поля и неабстрактные экземплярные методы. Создание экземпляров интерфейсов, как и абстрактных классов невозможно.

```
Interface {
    ...
}
```

Внутри тела перечисляются методы без модификаторов доступа и тел.

Реализация интерфейса — класс, являющийся подтипом этого интерфейса. Если этот класс неабстрактный, то в нем должны быть определены ВСЕ методы интерфейса.

Класс может реализовать сразу несколько интерфейсов, таким образом компенсируется отсутствие множественного наследования.

Реализуемые классом интерфейсы перечисляются после ключевого слова implements.

```
class Cat extends Animal implements Haunter, AfraidWater {
    ...
}
```

Интерфейс может быть наследником других интерфейсов, это перечисляется после ключевого слова extends. Для интерфейсов реализовано множественное наследование.

24. Экземплярные и статические вложенные классы, их объявление:

Класс A называется экземплярным вложенным классом(inner class) в классе B, если из методов класса A доступно внутреннее состояние объекта класса B. При этом класс B называется объемлющим классом.

Объекты вложенных классов имеют неявное поле, содержащее в себе ссылку на объект объемлющего класса, могут создаваться в экземплярных методах объемлющего класса, а также в экземплярных методах всех вложенных в этот объемлющий класс экземплярных классов.

Класс A называется статическим вложенным классом(static nested class) в классе B, если из методов класса A доступны статические поля класса B.

В Java объявление вложенного класса выглядит как обычное объявление класса, расположенное внутри тела объемлющего класса. При этом статический вложенный класс имеет модификатор static.

25. Понятие обобщенного класса, его объявление:

Контейнерные классы реализуются с помощью обобщенных классов.

Обобщенный класс — класс, имеющий формальные типовые параметры.

Формальный типовой параметр — тип, который используется в объявлении обобщенного класса и при этом неизвестен во время компиляции обобщенного класса.

class Name<param1, param2, … , paramN > (в качестве имен параметров можно использовать заглавные латинские буквы)

Фактические типовые параметры указываются при использовании класса в угловых скобках после его имени и подставляются на место формальных типовых параметров.

26. Ограниченный обобщенный класс. Верхняя граница типового параметра:

Ограниченный обобщенный класс — обобщенный класс, хотя бы для одного формального типового параметра которого задана верхняя граница.

Говорят, что для формального типового параметра задана его верхняя граница, если

базовый класс для типового параметра не Object. Другими словами, такому типовому параметру разрешено сопоставлять не любой класс, а только некоторый класс A или любого наследника A.

class Name<…, param extends A,...>

Задание верхней границы позволяет вызывать методы класса A из тела обобщенного класса.

27. Ковариантность массивов:

    В Java массивы ковариантны, то есть, если класс B наследник A, то среди типов массива B[] имеется тип A[].

    В частности:

    Integer[] ints = new Integer[10]

    Number[] numbers = ints

    Но если затем попытаться положить в numbers не Integer то программа скомпилируется, но на этапе выполнения упадет с исключением.

28. Инвариантность обобщенных классов:

    Stack<Integer> ints = new Stack<Integer>()

    Stack<Number> numbers = ints (incompatible types)

    Суть в том, что обобщенные классы — инвариантны, то есть, если A — обобщенный класс, и класс B наследник класса T, то классы A<B> и A<T> отношением наследования не связаны. Но, если, например, SortedMass<T> - наследник Mass<T>, то SortedMass<Integer> наследник Mass<Integer>.

29. Шаблон обобщенного класса, частичные шаблоны:

    Шаблон обобщенного класса A — неявный супертип для любого класса, порожденного из A. Шаблон получается параметризацией класса A специальным фактическим параметром «?».

    Можно объявить переменную типа шаблон, но нельзя создать объект этого типа. В частности, можно создать массив с элементами типа шаблон.

    Если тип переменной — шаблон некоторого обобщенного класса A, то этой переменной можно присвоить ссылку на любой объект класса A, независимо от того, какой фактический типовой параметр был передан A при создании объекта.

    Если обобщенный класс имеет несколько формальных типовых параметров, то возможно получить частичный шаблон, передав классу «?» вместо части фактических типовых параметров.

30. Шаблоны обобщенных классов, ограниченные сверху:

    Фактический параметр «?» шаблона имеет верхнюю границу, по умолчанию совпадающую с верхней границей соответствующего формального параметра обобщенного класса.

    Верхнюю границу можно уточнить:

    A<…, ? extends X,...>

    Естественно, уточнение верхней границы снизит количество подтипов шаблона.

    У шаблона, ограниченного сверху, недоступны методы, тип параметров которых соответствует «?» в шаблоне. Например, у шаблона Stack<?> недоступен метод push.

    Относительно контейнерных классов можно сказать, что использование их шаблонов, ограниченных сверху, не позволяет написать код, который осуществляет запись новых значений в контейнер.

31. Шаблоны обобщенных классов, ограниченные снизу:

    По умолчанию «?» ограничен сверху, однако можно, наоборот ограничить его снизу:

    A<…, ? super X,...>

    Подтипами такого шаблона будут все полученные из A классы, у которых в качестве соответствующего фактического типового параметра выступает X или любой суперкласс X.

    У ограниченного снизу шаблона контейнерного класса можно вызвать методы для

записи значений того класса, которым он ограничен. Однако нельзя вызвать те методы, тип возвращаемого значения которых соответствует «?» в шаблоне. То есть, Stack<? Super Integer> не сможет выполнить метод pop. То есть для контейнерных классов нельзя будет написать код, считывающий значение из объекта — контейнера.
Producer – extends, Consumer – super; (PECS)

32. Объявление и вызов обобщенных методов:
Обобщенный метод — это статический или экземплярный метод класса, имеющий формальные типовые параметры.
В объявлении метода типовые параметры предшествуют типу возвращаемого значения.
Вызов статического обобщенного метода:
Cat.<fact params>method(…)
Вызов экземплярного обобщенного метода:
myCat.<fact params>method(…), где myCat объект класса Cat.
33. Нештатная, исключительная, ошибочная ситуация. Два способа перехвата нештатных ситуаций:
Нештатная ситуация — ситуация, в которой выполнение некоторого фрагмента кода по тем или иным причинам невозможно.
Исключительная ситуация — нештатная ситуация, возникшая в силу внешних по отношению к программе причин. (не открылся файл, упал интернет)
Ошибочная ситуация — нештатная ситуация, возникшая из-за ошибки в коде программы. (выход за границу массива, деление на ноль, переполнение стека)
Перехват нештатных ситуаций — механизм, обеспечивающий продолжение работы программы при возникновении нештатной ситуации.

- Обработка кодов возврата: функция, во время выполнения которой может возникнуть нештатная ситуация, возвращает некоторое значение, которое говорит о том, что функция успешно или нет выполнила свою задачу. Перехват ситуации заключается в том, что в коде, вызывающем эту функцию производится обработка возвращаемых ею значений.
- Обработка исключений: в случае возникновения нештатной ситуации генерируется так называемое исключение, описывающее нештатную ситуацию. Генерация исключения передает управление на фрагмент кода, являющийся обработчиком исключения.

34. Исключение, иерархия классов исключений:
Исключение — объект, описывающий нештатную ситуац
Throwable – базовый класс всех исключений.
Error - базовый класс для исключений, приводящих к падению программы, то есть системные ошибочные ситуации.
Exception – базовый класс для всех классов несистемных исключений.
RuntimeException – базовый класс для классов несистемных исключений, описывающих ошибочные ситуации.
35. Операторы перехвата исключений в Java:
Порожденное и неперехваченное исключение приводит к падению программы, с выводом дампа стека вызовов, записанного в исключении.

```
try {
    code //может возникнуть нештатная ситуация
}
catch (SomeException e) {
    code
}
catch (SomeException2 e) {
    code
```

```
}
…
finally {
    code //вызывается при любом выходе из try-блока
}
```

36. Порождение исключения, модификатор throws:
    Порождение исключения выполняется оператором:
    ```
    throw exception;
    ```
    Например, throw new SomeException();
    Если класс исключения не является подклассом классов Error и RuntimeException, то для любого порожденного исключения в коде программы должен быть предусмотрен обработчик. Это требования выражается так, если в процессе выполнения метода m могут порождаться исключения x, y и z, и эти исключения не перехватываются в теле метода, то объявление метода выглядит так:
    ```
    type m(params) throws x, y, z {
        ...
    }
    ```
37. Понятие пакета. Имена пакетов и квалифицированные имена классов. Создание пакетов, пакет по умолчанию:
    Пакет — контейнер классов, представляющий для них отдельное пространство имен и дополнительные возможности по управлению доступом.
    Имя пакета — это непустая последовательность идентификаторов, разделенных точками. В именах пакетов не принято использовать заглавные буквы(разве что для аббревиатур) и допускается наличие символов подчеркивания.
    Квалифицированное имя класса - это имя, с помощью которого можно обращаться к публичному классу извне пакета, которому этот класс принадлежит.
    ```
    package_name.class_name
    ```
    Для создания пакета нужно пометить в начало каждого файла, содержащего исходный текст классов, которые вы планируете включить в проект директиву:
    ```
    package package_name;
    ```
    Классы, объявленные в файлах, в которых отсутствует директива package, считаются принадлежащими безымянному пакету по умолчанию. (default package)
    Компилятор Java требует, чтобы файлы пакета размещались в файловой системе в каталоге, путь к которому относительно текущего каталога соответствовал имени пакета. (ru.bmstu.iu9 → ru/bmstu/iu9)
    Файлы, принадлежащие пакету по умолчанию, должны храниться в текущем каталоге.
38. Правила видимости для классов пакетов Java и их членов. Импорт классов. Импорт статических членов класса:
    Один Java-файл может содержать не более одного публичного класса и произвольное число непубличных классов.
    Публичные классы можно использовать как внутри, так и снаружи пакета, а непубличные извне пакета невидны, однако доступны из любого места внутри пакета.
    К членам класса пакета, объявленным без модификатора доступа можно обращаться из любого места внутри пакета, но запрещено обращаться снаружи пакета.
    Для того, чтобы обратиться к классу, принадлежащему другому пакету, этот класс нужно импортировать, есть три способа:
    - Использование квалифицированного имени
    - Указание класса в директиве import
    - Использование директивы import для импорта всех классов пакета:
      ```
      import java.util.*;
      ```

    Для упрощения доступа к статическим членам классов существует специальная форма директивы import, позволяющая импортировать их в текущий файл.

import static package_name.class_name.member_name
import static package_name.class_name.*

39. Локальные и анонимные классы, lambda-выражения:

Можно сделать поле класса, а также локальную переменную или формальный параметр неизменяемыми. Для этого при объявлении нужно использовать модификатор final и, кроме того, неизменяемое поле нужно инициализировать в конструкторе класса, а неизменяемую переменную — при ее объявлении.

Локальный класс(local class) — экземплярный вложенный класс, объявляемый внутри метода объемлющего класса и имеющий доступ к неизменяемым локальным переменным и параметрам внутри этого метода.

Анонимный класс — локальный класс, объявление которого совмещено с созданием его экземпляра, для его создания используется специальная форма оператора new:

```
new BasicClassNameOrInterface(constructor params) {
    … //тело класса
}
```

Функциональный интерфейс — интерфейс с единственным абстрактным методом.

Создание экземпляра анонимного класса, реализующего функциональный интерфейс синтаксически может быть представлено в виде лямбда-выражения, имеющего две формы:

```
(formal params) → method_body
(formal params) → expr
```

~~~~~~

40. Ссылочный тип данных в C++, константные ссылки:

Ссылка — типизированный указатель, к которому неприменимы арифметические операции, который не может быть нулевым. Кроме того, недопустимы ссылки на ссылки.

Для объявления используется префиксный декларатор “&” (&var_name).

При этом ссылки в глобальных и локальных переменных должны быть инициализированы при их объявлении(в полях объектов — в конструкторе класса). Более того! Значение, полученное ссылкой при инициализации в дальнейшем не может быть изменено.

Для получения ссылки на объект не нужно использовать «&», для доступа к значению указатель разыменовывать не требуется.

Ссылка может быть возвращаемым значением функции, однако нужно следить, чтобы не вернуть ссылку на локальную переменную, которая не будет существовать после завершения выполнения функции.

Константные ссылки:

```
const type &var_name
```

Это такие ссылки, значение по которым изменять нельзя, но можно читать.

41. Объявление класса в C++, секции в объявлении класса:

Любая структура является классом, однако принято объявлять классы с помощью конструкции:

```
class Name {
    ...
};
```

Отличие от структуры заключается в том, что по умолчанию все члены структуры доступны извне структуры, а члены класса — нет.

В теле класса располагаются объявления полей, прототипы методов и конструкторов, а также в некоторых случаях определения методов и конструкторов(однако злоупотреблять такими определениями не стоит, чтобы не мешать раздельной компиляции).

Для управления доступом к членам класса предусмотрены секции(public, private,
~~~~~~

protected):

```
class Name {
    public:
    ….
    private:
    ….
    protected:
    ….
};
```

Секции могут быть перечислены внутри объявления в любом порядке и количестве, члены public-секции видны отовсюду, члены private только из методов данного класса, а protected из методов данного класса и его классов-наследников.

42. Объявление экземплярных и статических полей в C++, определения статических полей:
    Экземплярные поля класса объявляются также, как и поля структур в C. При этом объявление статических полей начинается с модификатора static. По умолчанию поля недоступны извне класса.
    Каждое статическое поле должно быть дополнительно определено в одной из единиц компиляции проекта(cpp-файл).
    Особенностью языка C++ является то, что он расчитан на ту же схему компиляции, что и C. То есть, программа на C++ состоит из набора cpp-файлов, каждый из которых компилируется в объектный файл. Для того, чтобы их связать применяется линкер.
    Так как один и тот же класс на практике может использовать в нескольких cpp-файлах, то в каждом файле должно быть его определение. НО. На практике объявление класса выносится в отдельный hpp-файл, который включается в нужные cpp-файлы.
    Так как статическое поле — фактически глобальная переменная, то она должна быть явно помещена в один из объектных файлов, для этого в соответствующем cpp-файле должно быть дано ее определение.
    type class_name::var_name = value;
43. Объявление методов, виртуальные и абстрактные методы:
    Объявление метода состоит из прототипа и определения.
    Прототип помещается в тело класса, а определение — в нужный cpp-файл(как и со статическими полями). Прототипы статических методов объявляются с модификатором static.
    Определение метода происходит с помощью применения квадроточия (type class_name::method_name(params) { … }).
    По умолчанию для вызова методов применяется раннее связывание, чтобы включить позднее связывание в объявлении прототипа метода нужно указывать модификатор virtual.
    Прототип абстрактного метода заканчивается на «= 0», класс, в котором есть абстрактный метод, автоматически становится абстрактным.
44. Объявление экземплярных конструкторов:
    Объявление экземплярного конструктора, как и объявление обычного экземплярного метода, состоит из прототипа конструктора в теле класса и его определения.
    В объявлении конструктора его имя совпадает с именем класса, также для конструктора не указывается тип возвращаемого значения.
    Чтобы запретить создание экземпляра класса достаточно объявить все его конструкторы в private или protected секции класса.
    Из-за специфики C++ существует специфичный принцип: никогда не вызывается из конструктора виртуальные методы, это может привести к проблемам, в случае переопределения этих методов в классах наследниках.
45. Создание объектов в динамической памяти, удаление объектов:

Динамическое выделение памяти под значение любого типа выполняется операцией new:

new primitive_type_name;
new primitive_type_name(value);
new class_name(fact_constructor_params);

New возвращает указатель на значение в динамической памяти и совмещен с инициализацией(вызовом конструктора).

В C++ нет сборщика мусора, то есть автоматическое управление памятью не предусмотрено. Все объекты, созданные с помощью new, должны быть в какой-то момент явно освобождены.

Для этого существует оператор delete:

delete obj_pointer;

46. Создание массивов в динамической памяти, удаление массивов:

В C++ не принято использовать функцию malloc, так как для создания массивов также используется специальная форма операции new:

new el_type[size];

Массив объектов можно выделять только в случае, если в классе определен конструктор по умолчанию. При этом этот конструктор будет вызван для каждого объекта в массиве.

В C++ нет сборщика мусора, то есть автоматическое управление памятью не предусмотрено. Все объекты, созданные с помощью new, должны быть в какой-то момент явно освобождены.

Для этого существует оператор delete:

delete []mass_pointer;

47. Объявление деструкторов, как осуществляется вызов деструктора:

Внутреннее состояние объекта может содержать указатели(ссылки) на другие объекты , созданные в его конструкторе. В языке без сборщика мусора освобождение такого объекта может привести к утечке памяти.

Деструктор — экземплярный метод, предназначенный для освобождения ресурсов, принадлежащих объекту, непосредственно перед удалением этого объекта.

В языке C++ деструкторы жизненно необходимы:

~class_name();

Деструктор вызывается при уничтожении объекта, в частности, его автоматически вызывает оператор delete.

48. Объекты в автоматической и глобальной памяти, автоматический вызов деструкторов:

Автоматическая. В отличие от Java, в C++ объекты могут размещаться в глобальных и локальных переменных, в параметрах методов, элементах массивов и полях других объектов.

Синтаксис объявления переменной, содержащей объект:

class_name var_name(constructor_params);

Такие объявления можно применять и для переменных, тип которых не является классом.

Объявление переменной, содержащей объект, вызывает экземплярный конструктор класса. А при выходе из блока, где она была объявлена, автоматически вызывается деструктор.

Глобальная. Если объект размещается в глобальной переменной или в статическом поле класса, то конструктор для него вызывается до передачи управления в функцию main, а деструктор после завершения main.

49. Объекты в полях других объектов, их инициализация:

Если объект класса X содержится в поле x объекта Y, то конструктор для этого поля вызывается из конструктора Y, таким образом:

Y::Y(formal_constructor_Y_params)

```
    : x(fact_constructor_X_params) {
        ….
}
```
Причем, любое поле класса может быть инициализировано подобным образом, даже если оно не является классом.

50. Проблема копирования объектов, конструктор копий и перегрузка операции присваивания:
    Передача объекта в качестве параметра при вызове метода автоматически влечет создание его копии. Копирование производится для того, чтобы при изменении объекта, переданного в метод, не изменялся оригинальный объект(семантика копирования). Также копирование объектов осуществляется при инициализации объявляемых переменных.
    По умолчанию при копировании создается побитовая копия внутреннего состояния объекта, но этого может быть не достаточно, так как, например, объект может содержать в себе указатель на динамический массив. В такой ситуации копирование приведет к тому, что в двух объектах будет указатель на один и тот же массив, а при удалении одного из объектов второй потеряет доступ к этому массиву и впоследствии не сможет его освободить.
    Для того, чтобы подобное не происходило, требуется объявить конструктор копий.
    ```
    class_name(const class_name &obj);
    ```
    Этот конструктор будет автоматически вызываться, когда требуется создать копию объекта.
    Проблема копирования возникает также и при присваивании, для того, чтобы избежать этого требуется переопределить операцию присваивания.
    ```
    class_name& operator=(const class_name &obj);
    ```
51. Одиночное наследование, вызов конструктора базового класса:
    ```
    class Cat: public Animal {
        ….
    };
    ```
    Если конструктор базового класса является конструктором по умолчанию, то конструктор производного класса вызовет его автоматически, иначе же требуется сделать это вручную(аналогично конструктору полей).
    ```
    Cat::Cat(formal_params)
        : Animal(fact_params) {
            ….
    }
    ```
    Private-наследование: все методы базового класса становятся приватными в производном.
    Public-наследование: права доступа для полей базового класса у наследника не изменяются.
    Protected-наследование: все поля базового класса в наследнике становятся protected.
    Однако приватные поля базового класса никогда не будут доступны в наследнике.
52. Переопределение методов:
    Для переопределения метода в классе-наследнике, как и в Java, достаточно просто объявить и переопределить этот метод в производном классе.
    *Можно привести любой пример
53. Динамическое приведение типов:
    Операция динамического приведения объекта obj к типу T проверяет, является ли тип T одним из типов объекта obj и возвращает obj, если является. Иначе операция либо возвращает нулевой указатель, либо порождает исключение.
    Операция может применяться к ссылкам и указателям на объекты:
    ```
    dynamic_cast<T> (obj)
    ```

Нулевых ссылок не существует, поэтому в случае с ссылкой порождается исключение std::bad_cast.

В случае если в классе нет виртуальных методов, то в нем не содержится информации о типе, что делает невозможным приведение типов для данного класса.

54. Множественное наследование, вызов конструкторов базовых классов:

Поддерживается множественное наследование:

```
class B: mod A1, mod A2, …., mod An { //mod = public||private||protected
    ….
};
```

При этом как и в случае одиночного наследования из конструктора производного класса должны вызываться конструкторы всех базовых классов, кроме, может быть тех, которые имеют конструкторы по умолчанию.

55. Разрешение противоречий в именах наследуемых членов классов при множественном наследовании:

При множественном наследовании в производном классе могут возникать методы с одинаковыми именами и сигнатурами, или же просто одноименные поля. Это приведет к ошибкам на этапе компиляции.

Для разрешения этих противоречий следует использовать квалифицированные имена (обращаться через квадроточие по имени базового класса).

56. Понятие иерархии наследования, классы противоречия, основная проблема противоречивых иерархий:

Иерархия наследования — ориентированный ациклический граф, множеством узлов которого является множество классов программы. При этом если класс A является непосредственным базовым классом для класса B, то из узла B идет дуга в A.

Класс B является классом противоречием, если существует такой класс A, что в иерархии наследования можно провести не менее двух непересекающихся путей из A в B.

Иерархия — противоречивая, если в ней есть классы-противоречия.

Основной проблемой противоречивых иерархий является возможность многократного включения полей базового класса в производный класс.

57. Виртуальное наследование:

Позволяет избежать многократного включения базового класса в производный.

Виртуальное наследование — способ реализации наследования, гарантирующий, что базовый класс не будет включен ни в один из производных классов более чем в одном экземпляре:

```
class B: mod virtual A {
    ….
};
```

Где A — виртуальный базовый класс для B.

58. Понятие шаблона, виды параметров шаблона, значения параметров шаблона по умолчанию:

В отличие от Java, в C++ обобщенные классы отсутствуют. Вместо них в C++ используется развитый язык макроопределений, предназначенный для генерации кода во время компиляции программы.

Шаблон(template) – «рецепт» для генерации кода класса или метода:

```
template <formal_params>
```

Тело шаблона представляет собой определение класса или функции, тем самым позволяя объявить шаблон соответственно класса или функции.

Формальные параметры шаблона — идентификаторы, областью видимости которых является тело шаблона. Они могут обозначать типы, значения или, в свою очередь, другие шаблоны. При применении шаблона формальные параметры в его теле заменяются на конкретные типы, значения и шаблоны, полученный код

компилируется.

Типовые параметры шаблона обозначают типы: синтаксически имя типового параметра в списке формальных параметров шаблона предваряется ключевым словом typename(class).

Не типовые параметры представляют собой значения, а не типы. Их синтаксис похож на объявление переменных. Внутри тела шаблона имя такого параметра обозначает константу указанного типа.

*тут можно привести пример

Шаблонные формальные параметры шаблона — позволяют параметризовать шаблон другим шаблоном, синтаксически такой параметр записывается как объявление шаблона без тела.

*пример

Формальные параметры шаблона могут иметь значения по умолчанию. Синтаксически это оформляется путем добавления после имени параметра знака «=», за которым следует значение. Если формальный параметр имеет значение по умолчанию, то все следующие за ним параметры также должны иметь значения по умолчанию. Такие значения используются в случае, если при инстанциации шаблона соответствующие фактические параметры не указаны.

Зависимый идентификатор — любое имя внутри определения шаблона, которое зависит от формальных параметров шаблона. Смысл зависимых идентификаторов становится ясен компилятору только при инстанциации шаблона, когда известны его фактические параметры. Поэтому по умолчанию компилятор считает, что зависимый идентификатор обозначает поля и методы. Чтобы указать, что это тип, перед идентификатором нужно указать ключевое слово typename.

59. Инстанциация шаблона. Выведение фактических параметров шаблона при инстанциации, требования к фактическим параметрам шаблона:

Инстанциация шаблона — порождение кода по шаблону и списку фактических параметров. Инстанциация осуществляется при первом использовании конструкции:

```
template_name<fact_params>
```

порождается код, полученный путем подстановки в тело шаблона фактических параметров.

Если параметры шаблона функции используются в списке формальных параметров этой функции, то компилятор может вывести значения фактических параметров шаблона самостоятельно (примером является шаблон матрицы и метод умножения матриц).

Тело шаблона накладывает требования на его фактические параметры. Если фактические параметры шаблона не удовлетворят требованиям, инстанциация шаблона с этими фактическими параметрами приведет к ошибке времени компиляции.

60. Специализация шаблона функции:

Специализация шаблона — разработка отдельной версии порождаемого шаблоном кода для конкретного набора фактических параметров:

```
template <formal_params>
type name(…)
{….}
```

Тогда специализированная версия будет иметь вид:

```
template<>
type name<fact_params>(….)
{….}
```

Однако, стоит учесть, что это не дает преимуществ перед обычной перегрузкой функций. И перегрузка имеет больший приоритет, чем специализация.

61. Специализация шаблона класса, частичная специализация:

Специализированные версии шаблонов классов создаются аналогично специализации шаблонов функций.

При частичной специализации фиксируется только часть фактических параметров шаблона. (н-р: кортеж и одного элемента любого типа)

62. Перегрузка операций присваивания(обычной и составной):

Операция присваивания для некоторого класса A перегружается путем объявления в классе метода:

```
A& operator= (const &obj);
```

Перегруженная операция должна работать по следующей схеме:

1. проверить, не присваивается ли объект сам себе, иначе к шагу 5
2. выделить память, в которую будет скопировано содержимое obj
3. освободить память используемую внутри объекта
4. выполнить копирование
5. вернуть *this

Вызвать можно так:

```
a = b или a.operator=(b);
```

Операция присваивания возвращает левое значение, то есть, результату, возвращаемому операцией присваивания можно что-то присвоить.

```
A x, y, z;
(x = y) = z //будет работать, не нужно это пытаться ограничить, работает в том числе и для встроенных типов
```

Составные операции присваивания перегружаются по той же схеме, что и обычная операция присваивания:

```
A& operator += (const A &obj);
```

Стоит также предусматривать возможность присваивания объекта самому себе. Составные операции также возвращают левое значение:

```
(x += y) += z; //работает
```

63. Перегрузка бинарных арифметических операций:

Прототип перегруженной бинарной операции:

```
const A& operator+ (const A &other) const;
```

Ключевое слово const обозначает, что метод не меняет внутреннее состояние объекта для которого он вызван. Если объект — константный, то к нему применимы только константные методы.

Удобно реализовывать бинарные ариметические операции через операции составного присваивания (return A(*this) += other). Стоит при этом избежать лишнего вызова конструктора копий и деструктора.

64. Перегрузка операции сравнения:

Операции сравнения имеют следующие прототипы:

```
bool operator== (const A &other) const;
bool operator!=  (const A &other) const;
bool operator<   (const A &other) const;
bool operator>   (const A &other) const;
bool operator<= (const A &other) const;
bool operator>= (const A &other) const;
```

При этом удобно при реализации одних операций сравнения использовать уже реализованные другие операции. Например, реализовать «!=», через отрицание в ее коде результата вызова операции «==».

Это обеспечивает непротиворечивость реализации операций сравнения.

65. Перегрузка операции []:

Прототип:

```
type1& operator[] (type2 index);
```

Операция возвращает ссылку, что позволяет использовать ее как результат в левой

части операции присваивания.

К сожалению, таким образом перегруженную операцию сравнения нельзя применять к константным объектам, поэтому принято определять отдельную версию операции для константных объектов:

```
const type1& operator[] (type2 index) const;
```

В этом случае необязательно, чтобы операция возвращала ссылку. Поэтому если «тип1» допускает «дешевое» копирование, то от ссылки можно избавиться.

Эта операция не может быть перегружена с помощью функции.

66. Перегрузка операций * и →:

Такая перегрузка позволяет использовать объекты класса так, как будто они являются указателями:

```
type& operator*();
type operator→ (); //метод класса A
type& operator* (A &a); //вне класса
```

Особенностью → является то, что она последовательно применяется к своему возвращаемому значению до тех пор, пока не получится указатель. То есть, например, если в классе A операция → возвращает ссылку на объект класса B, а в классе B операция → возвращает указатель на объект класса C, то применение операции → к переменной x типа A открывает доступ к полям и методам объекта класса C.

67. Перегрузка операции приведения типа:

Прототип:

```
operator type() const;
```

Например:

```
struct A {
    int x;
    operator int() const;
};
A::operator int() const { return x; }
```

Нужно учесть, что, хотя операция приведения типа возвращает значение, в её объявлении тип возвращаемого значения не указывается, что напоминает прототип конструктора.

68. Перегрузка операции ( ):

Перегрузка данной операции позволяет пользоваться объектами, словно они являются функциями.

Прототип:

```
type operator() (formal_params);
```

Можно сказать, что эта операция n-арная, так как внутри скобок может располагаться произвольное количество формальных параметров.

Поэтому в классе можно объявить сколько угодно таких операций с различными сигнатурами.

69. Порождение исключения:

В отличие от Java в качестве исключения в C++ может служить любое значение или объект.

Для порождения исключения используется оператор throw, который может вызывать как с указанием значения, описывающего исключительную ситуацию, так и без него:

```
throw value;
throw;
```

Механизм исключений не предусматривает сохранение информации о стеке вызовов в момент порождения исключения, поэтому сообщение, выводимое при аварийном завершении программы менее информативно, чем в Java.

Для создания собственного класса-исключения рекомендуется наследовать его от библиотечного класса exception(из заголовочного файла <exception>). В производном

классе нужно переопределить виртуальный метод what, возвращающий строку с описанием исключительной ситуации.

70. Операторы перехвата исключений:
    Как и в Java, участки кода, в которых ожидается возникновение исключительной ситуации, обрамляются try-блоками. С try-блоками связаны один или несколько catch-блоков, осуществляющих перехват исключений по типу:

```
try {
    /*….*/
}
catch (type_exception1 var) {
    /*….*/
}
catch (type_exception2 var) {
    /*….*/
}
```

    Catch-блок, который может перехватить любое исключение записывается так:

```
catch(….) {
    /*….*/
}
```

71. Жизненный цикл объектов-исключений:
    Существует несколько различных случаев:
    1. В случае, если объект-исключение порождается в куче и перехватывается по указателю: в конце соответствующего catch-блока объект-исключение нужно явно уничтожить с помощью delete.
    2. В случае, если порождается временный объект-исключение и при перехвате создается его копия: после завершения catch-блока автоматически будет уничтожена копия, затем оригинальный объект.
    3. В случае, если порождается временный объект-исключение и перехватывается по ссылке, то есть копия не создается: объект будет автоматически уничтожен после завершения catch-блока.

    При передаче управления в catch-блок автоматические объекты уничтожаются.
72. Перехват исключений в инициализаторах конструкторов:
    Конструкторы в C++ могут иметь списки инициализаторов, в которых вызываются конструкторы объектов, расположенных в полях и конструкторы базовых классов. Такие конструкторы могут порождать исключения, и нужно уметь их перехватывать, например, чтобы логировать неудачное создание объекта.
    Для этого существует специальная конструкция:

```
class::class(constructor_params) try
    : initialization_list
{ /*…. */ }
catch (exception) { /*…. */}
….
```

    Стоит отметить, что если catch-блок не оканчивается порождением другого исключения, старое исключение не уничтожается, а передается дальше.
73. Спецификатор throw в заголовках функций:
    Как и в Java в C++ можно перечислить типы исключений, которые может порождать вызов функции. Для этого после списка формальных параметров функции следует разместить спецификатор throw:

```
throw (exception_type_list)
```

    Семантика этого спецификатора несколько отличается от такой же конструкции в Java:
    1. вызов функции, для которой разрешенные исключения не указаны, может порождать любые исключения.

2. вызов функции со спецификатором throw() не порождает никаких исключений.
3. компилятор не следит за тем, какие исключения порождаются внутри функции: вместо этого неразрешенные исключения просто не могут выйти за пределы функции и приводят к аварийному завершению программы.

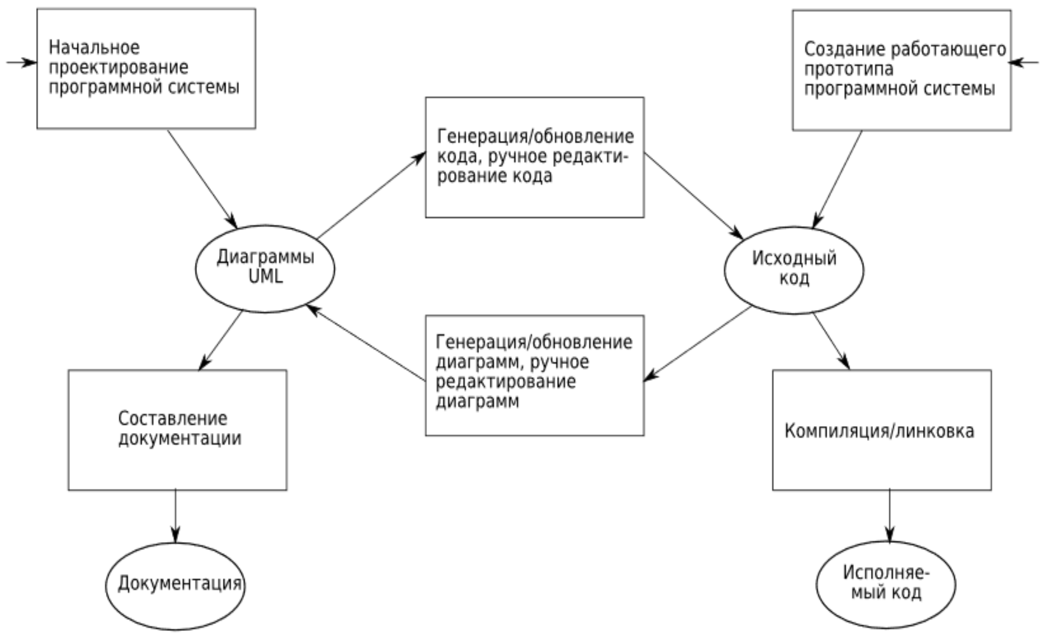
Для соблюдения полиморфизма переопределение виртуальных методов не снимает ограничений, накладываемых спецификатором throw, то есть, переопределенному методу не разрешается порождать исключения, не порождаемые методом базового класса.

~~~~~~

Unified Modeling Language (UML) – это стандартизованный язык моделирования общего назначения, используемый при разработке программного обеспечения.

В диаграмме три составляющие: структурная диаграмма(классы, компоненты, объекты и т.д.), поведенческая диаграмма, диаграмма взаимодействия.

74. Цикл разработки программной системы с помощью UML:



Разработка может быть организована, как чередующиеся две фазы: работа с моделью, работа с исходным кодом. Начинать можно с любой части, например, начали с ООП и начертили модель классов, после этого по этой модели мы пишем исходный код на языке программирования, мы его компилируем и отлаживаем, что-то добавляем и изменяем. После этого по коду мы вносим изменения в модель и работает уже с ней, затем снова обновляем код, таким образом все и чередуется. В любой момент мы можем получить исходники и документацию.

75. Классы на диаграммах, атрибуты и операции:

Диаграмма классов (class diagram) – диаграмма языка UML, на которой представлены классы с атрибутами и операциями, а также связывающие их отношения.

Временные аспекты функционирования программы не указываются.

Атрибут — некий присущий объекту класса признак, позволяющий один объект класса отличать от другого. Иначе говоря — свойства и поля.

Операции — сервис, предоставляемый каждым объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и объекты этого класса. Иначе говоря — методы.

76. Зависимость, обобщение, реализация:
~~~~~~

Зависимость — самое слабое отношение, говорящее о том, что класс неким образом использует другой класс.

Обобщение — отношение классификации между более общим элементом и более частным или специальным элементом, проще говоря — наследование от дочернего к предку.

Реализация — отношение, при котором один элемент реализует поведение, которое определяет другой элемент.

77. Ассоциация, направленная ассоциация, агрегация, композиция:

Ассоциация — отношение, при котором внутреннее состояние одного объекта содержит ссылку на другой объект, и наоборот.

Направленная ассоциация — ассоциация, выражающая несимметричную связь сущностей, соответствующих объектами.

Агрегация — направленная ассоциация, при которой объект некоторого класса является составной частью внутреннего состояния другого объекта.

Композиция — агрегация, при которой агрегируемый объект является неотъемлемой частью агрегирующего объекта.