Содержание

1. Билет 1.

1.1 Вопрос по Java. Понятия объекта и инкапсуляции. Понятие класса. Объявление классов в Java. Члены класса. Доступ к членам класса.

Определение 1.1.1. Объект – это самоописывающая структура данных обладающая внутренним состоянием и способная обрабатывать передаваемое её сообщение.
Если разбить на пункты:

1. Описывает сам себя
2. Имеет внутреннее состояние (Внутри могут быть другие объекты, переменные, одним словом поля)
3. Может обработать сообщение (Различные методы (метод - это способ передачи сообщения))

Определение 1.1.2. Понятие инкапсуляции – внутреннее состояние объекта должно быть доступно только в самом объекте и изменяться с помощью сообщений, те доступ к внутренностям только через сообщения Официальное определение - доступ к внутреннему состоянию(полям) объекта происходит через сообщения(методы)

Класс в Java объявляется ключевым словом **class**. Пример объявления класса

```
class A {

}
```

Перед словом класс может быть модификатор доступа В каждом файле обязательно должен быть публичный класс и у него обязательно должно имя совпадать с именем класса. Членами класса являются:

1. Экземплярные поля – хранят внутреннее состояние объекта. Именованная составная часть внутреннего состояния объекта.
2. Экземплярные методы – это подпрограмма выполняющая обработку передаваемого объекту сообщения. Имеет доступ к внутреннему состоянию объекта и передает сообщение.
3. Экземплярные конструкторы – инициализируют только что созданные объекты класса
4. Статические поля – хранят данные общие для всех объектов класса

5. Статические методы – выполняют действия для которых не нужен доступ к конкретному объекту класса.

6. Вложенные классы – объекты необходимые для реализации данного класса.

7. Статический конструктор – инициализирует статические поля класса (в Java такого нет, его функцию выполняют статические блоки см. лекцию 4)

Доступ к членам класса зависит от их модификатора доступа. Модификаторы:

1. **private** – доступ только из тела класса

2. **Без модификатора** – доступ только класса или классов того же пакета

3. **protected** – доступ из класса, классов из того же пакета и наследников класса

4. **public** – доступ откуда угодно

1.2 Вопрос по C++. Понятие шаблона. Типовые формальные параметры шаблон. Нетиповые формальные параметры шаблона.

Определение 1.2.1. Шаблон – это обобщенный тип или функция.

Определение 1.2.2. Типовые формальные параметры шаблона – это параметры которые определяют типы данных с которыми будет работать шаблон(задаются typename или class)

Определение 1.2.3. Нетиповые формальные параметры шаблона – это параметры которые задают какие-то константы и значения, не связанные с типами.

Определение 1.2.4. Шаблонные формальные параметры шаблона – позволяет параметризовать шаблон другим шаблоном, такой шаблом синтаксически записывается как шаблон без тела

Шаблон объявляется словом template затем идут скобочки <> внутри которых перечисляются формальные параметры. Затем идет тело шаблона.

Определение 1.2.5. Формальные параметры – это индентификаторы которые видны только в теле шаблона и могут обозначать типы данных, значения, другие шаблоны

При применении шаблона, формальные параметры заменяются на конкретные типы данных и значения и шаблоны и код компилируется.

2. Билет 2.

2.1 Вопрос по Java. Экземплярные поля. Статические поля

Определение 2.1.1. 1. Экземплярные поля – хранят внутреннее состояние объекта. Именованная составная часть внутреннего состояния объекта. 2. Статические поля – хранят данные общие для всех объектов класса

2.2 Вопрос по C++. Виртуальное наследование.

Виртуальное наследование – это способ реализации наследования, гарантирующий, что базовый класс не будет включён ни в один из производных классов более чем в одном экземпляре.

```
class A {
public:
    int x = 10;
    virtual int f() {
        return 5;
    }
};

class B: public A {
public:
    int f() {
        return 6;
    }
}

class C: public A, B {

};
```

Без виртуального наследования при создании экземпляра класса C будет создаваться два поля x, что нам не особо нужно и не понятно к какому полю необходимо обратиться.
При использовании виртуального наследования будет создаваться одно поле x

```
class A {
public:
    int x = 10;
```

```
    virtual int f() {
        return 5;
    }
};

class B: public virtual A {
public:
    int f() {
        return 6;
    }
}

class C: public virtual A, B {

};
```

3. Билет 3.

3.1 Вопрос по Java. Экземплярные и статические методы. Объявление методов в Java. Перегрузка методов. Виртуальные методы.

Определение 3.1.1. Экземплярные методы – это подпрограмма выполняющая обработку передаваемого объекту сообщения. Имеет доступ к внутреннему состоянию объекта и передает сообщение

Определение 3.1.2. Статические методы – методы, которые выполняют действия к которым не нужен доступ к конкретному объекту класса.

Пример объявления метода в **Java**:

```
модификатор_доступа тип_возвращаемых_данных имя_метода(сигнатура) {
    тело метода
}
```

Модификаторы:

1. **private** – доступ только из тела класса
2. **Без модификатора** – доступ только класса или классов того же пакета
3. **protected** – доступ из класса, классов из того же пакета и наследников класса
4. **public** – доступ откуда угодно

Если метод статичный, то перед модификатором пишут **static**

Определение 3.1.3. Перегрузка – объявление для одного класса нескольких методов с одинаковым именем и разными сигнатурами.

Определение 3.1.4. Сигнатура – это информация о количестве и типах формальных параметров метода. Раннее связывание – определение адреса вызываемого экземплярного метода во время компиляции программы. Например, в c++.

Определение 3.1.5. Позднее связывание – определение адреса вызываемого экземплярного метода на основе информации о классе объекта во время выполнения программы. Например, в Java. Экземплярные методы для которых выполняется позднее связывание называются виртуальными.

Замечание. Все методы в **Java** виртуальные.

Выводы:

1. Для всех экземплярных методов будет выполнятся позднее связывание
2. Конкретная реализация метода будет определятся во время исполнения
3. Нет необходимости знать точно тип объекта для работы с ним через виртуальные методы – достаточно знать, что объект принадлежит классу или наследнику класса, в котором объявлен метод.

3.2 Вопрос по C++. Иерархия наследования и классы противоречия. Основная проблема противоречивых иерархий.

Иерархия наследования – это ориентированный ацикличный (направленный и без циклов) граф, в котором если класс A базовый и B наследует его, то от A к B проведено ребро.
Класс противоречия – это класс в который из какого-то другого класса ведут как минимум 2 пути. Пример, тут класс A будет классом противоречия

```
class A {
public:
    int x = 10;
    virtual int f() {
        return 5;
    }
};

class B: public A {
public:
    int f() {
        return 6;
    }
}

class C: public A, B {

};
```

Основной проблемой класса противоречий является многократное включение полей и методов базового класса в произвольный.

4. Билет 4.

4.1 Вопрос по Java. Понятие экземплярного конструктора. Конструктор по умолчанию. Объявление конструктора в Java.

Определение 4.1.1. Экземплярный конструктор – экземплярный метод, который инициализирует только что созданный объект.

Определение 4.1.2. Конструктор по умолчанию – экземплярный конструктор с пустой сигнатурой В java конструктор объявляется в теле класса и имеет то же имя, что и класс. Ничего не возвращает, соответственно не нужно указывать тип возвращаемых данных.

Пример:

```
class Dog {
    Dog() {
        System.out.Println("Dog was born");
    }
}
```

4.2 Вопрос по C++. Наследование от нескольких базовых классов. Противоречия в именах наследуемых членов класса. Разрешение противоречий в именах.

C++ поддерживает множественное наследование. Пусть класс A наследует классы AA, AAA, и AAAA, тогда это будет записываться как: class A : AA, AAA, AAAA Также есть возможность установить модификатор доступа для наследуемых классов, например: class A : public AA, AAA, AAAA При множественном наследовании может возникнуть проблема одинаковых имен в различных базовых классах. Такая проблема разрешается конструкцией, когда мы обращаемся к конкретному базовому классу от объекта.
Например:

```
    int main() {
        C CC;
        std::cout << CC.A::x;

        return 0;
    }
```

5. Билет 5.

5.1 Вопрос по Java. Размещение объектов в памяти. Операция «new». Массивовые литералы.

Определение 5.1.1. Если линейно упорядоченное множество A разбито на две (непересекающиеся) части B и C, причём любой элемент B меньше любого элемента C, то B называют *начальным отрезком* множества A. Другими словами, подмножество B линейно упорядоченного множества A является *начальным отрезком*, если любой элемент B меньше любого элемента $A \setminus B$.

Операция "new"в Java используется для создания нового объекта в памяти. При вызове оператора "new"происходят следующие действия:

1. Выделение памяти: При вызове оператора "new"резервируется достаточное количество памяти для хранения объекта в куче (heap). Куча - это область памяти, используемая для динамического выделения памяти для объектов во время выполнения программы.

2. Вызов конструктора: После выделения памяти вызывается конструктор класса объекта. Конструктор инициализирует объект и устанавливает его начальное состояние.

3. Возвращение ссылки: После создания и инициализации объекта оператор "new"возвращает ссылку на этот объект. Ссылка может быть использована для доступа к объекту и вызова его методов.

Пример использования оператора "new"для создания объекта класса Person:

```
Person person = new Person();
```

Что касается массивовых литералов, в Java существует возможность инициализации массивов при их объявлении с помощью литералов. Массивовый литерал - это удобный способ представления значений массива без явного вызова оператора "new"и инициализации каждого элемента отдельно.
Пример инициализации массива целых чисел с помощью массивового литерала:

```
int[] numbers = {1, 2, 3, 4, 5};
```

5.2 Вопрос по C++. Динамическое приведение типов.

В C++ динамическое приведение типов используется для проверки и преобразования указателей или ссылок на базовые классы в указатели или ссылки на их производные классы. Динамическое приведение типов обеспечивает безопасное приведение типов во время выполнения программы.

Синтаксис динамического приведения типов выглядит следующим образом:

```
dynamic_cast<новый_тип>(выражение)
```

Пример использования динамического приведения типов:

```cpp
#include <iostream>

class Base {
public:
    virtual void foo() {
        std::cout << "Base::foo()" << std::endl;
    }
};

class Derived : public Base {
public:
    void foo() override {
        std::cout << "Derived::foo()" << std::endl;
    }

    void bar() {
        std::cout << "Derived::bar()" << std::endl;
    }

    int x = 0;
};

int main() {
    Base* basePtr = new Derived();

    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr) {
        derivedPtr->foo();  // Вызов метода производного класса
        derivedPtr->bar();  // Вызов дополнительного метода производного класса
        derivedPtr->x++;
        std::cout << derivedPtr->x;
    } else {
        std::cout << "Dynamic cast failed." << std::endl;
```

```
    }

    delete basePtr;

    return 0;
}
```

Замечание. Важно помнить, что dynamic cast выполняется во время выполнения программы и требует полиморфных типов (наличия хотя бы одной виртуальной функции). Если объект не является экземпляром класса или его производного класса, dynamic cast вернет нулевой указатель или бросит исключение. Поэтому рекомендуется проверять результат dynamic cast на нулевое значение перед использованием его методов или членов для предотвращения ошибок доступа к памяти.

6. Билет 6.

6.1 Вопрос по Java. Понятие статического конструктора. «Static»-блоки.

Определение 6.1.1. Статический конструктор - конструктор. который используется для инициализации любых статических данных или для выполнения определенного действия, которое требуется выполнить только один раз. Он вызывается автоматически перед созданием первого экземпляра или ссылкой на какие-либо статические члены.

В Java не существует понятия "статического конструктора". Однако, в Java есть "статические блоки"(static blocks), которые выполняются при загрузке класса и позволяют инициализировать статические поля и выполнять другие операции, связанные с классом.
Статический блок выглядит следующим образом:

```
static {
    // Код инициализации или другие операции
}
```

Когда класс загружается в память впервые, статический блок выполнится перед любым другим кодом в классе (то есть при первом запуске класса, ещё до того, как этот класс будет использоваться в программе, до создания его экземпляров, вызова статических методов и обращения к ним и тд.). Если объявление класса содержит несколько статических блоков, то они будут выполняться в том порядке, в котором они перечислены в теле класса.

```
public class MyClass {
    static int x;

    static {
        // Инициализация статического поля x
        x = 10;
    }
}
```

6.2 Вопрос по C++. Основной способ наследования в C++. Переопределение методов.

Основной способ наследования в C++ - это одиночное наследование (single inheritance). Он позволяет классу наследовать свойства и поведение только от одного базового класса.
Синтаксис для объявления класса с одиночным наследованием выглядит следующим образом:

```
class DerivedClass : public BaseClass {
    // Определения членов класса DerivedClass
};
```

Где DerivedClass - это производный класс (subclass), а BaseClass - это базовый класс (base class), от которого DerivedClass наследуется. Ключевое слово public указывает, что доступ к публичным членам базового класса сохраняется и в производном классе.
Переопределение методов - это процесс создания метода в производном классе с тем же именем, аргументами и возвращаемым значением, что и метод в базовом классе. При вызове этого метода у объекта производного класса будет использована его реализация, а не реализация из базового класса.

```
#include <iostream>

class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal makes a sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog barks" << std::endl;
    }
};

int main() {
    Animal animal;
    animal.makeSound();  // Выводит: "Animal makes a sound"

    Dog dog;
    dog.makeSound();     // Выводит: "Dog barks"

    Animal* animalPtr = &dog;
    animalPtr->makeSound();  // Выводит: "Dog barks"
```

```
    return 0;
}
```

7. Билет 7.

7.1 Вопрос по Java. Понятие субтипизации. Явная и неявная субтипизация.

Определение 7.1.1. Тип данных A является подтипом типа данных B, если программный код, рассчитанный на обработку значений типа B, может быть корректно использован для обработки значений типа A.

Определение 7.1.2. Субтипизация (Subtype) – свойство ЯП, означающее возможность спользования подтипов в программах.

Замечание. В Java используется явная субтипизация

Определение 7.1.3. Явная субтипизация - это явное приведение объекта к типу его подтипа. Явное приведение необходимо, когда компилятор не может автоматически сделать вывод о типе во время компиляции. Если объект не является экземпляром подтипа, при выполнении программы может быть сгенерировано исключение ClassCastException.

Пример:

```
ParentClass parent = new ChildClass(); // Явное приведение ChildClass к ParentClass
```

Замечание. В этом примере ChildClass является подклассом ParentClass, и мы явно приводим объект типа ChildClass к типу ParentClass. Это позволяет нам присвоить объект ChildClass переменной типа ParentClass.

Замечание. В ЯП, поддерживающих неявную типизацию, решение о том, является ли A подтипом типа B, происходит на основе структуры значений этих типов.

7.2 Вопрос по C++. Проблема копирования объектов. Объявление конструктора копий. Перегруженная операция присваивания.

Проблема копирования объектов в C++ возникает, когда у вас есть класс, который содержит динамически выделенные ресурсы (например, память), и вы пытаетесь создать копию объекта. При поверхностном копировании объектов проблема может возникнуть, так как обе копии объекта будут ссылаются на одни и те же ресурсы. Это может привести к неожиданному поведению программы, например, двойному удалению памяти или изменению данных в нескольких объектах при изменении только одного объекта.
Для решения этой проблемы в C++ можно использовать конструктор копий и перегруженную операцию присваивания.

1. Конструктор копий - это специальный метод класса, который создает новый объект, инициализируя его значениями другого объекта того же типа. Конструктор копий вызывается при создании копии объекта. Пример объявления конструктора копий:

```
class MyClass {
public:
    MyClass(const MyClass& other); // Конструктор копий
};
```

   Внутри конструктора копий должен быть написан код, который реализовывает лубокое копирование ресурсов, чтобы каждый объект имел свои собственные независимые копии ресурсов.

2. Перегруженная операция присваивания - это оператор, который позволяет присвоить значения одного объекта другому объекту того же типа. Перегруженная операция присваивания вызывается при присваивании одного объекта другому. Пример объявления перегруженной операции присваивания:

```
class MyClass {
public:
    // Перегруженная операция присваивания
    MyClass& operator=(const MyClass& other);
};
```

   Внутри перегруженной операции присваивания можно реализовать логику глубокого копирования ресурсов.

Объявление конструктора копий и перегруженной операции присваивания позволяет контролировать процесс копирования объектов и обеспечивает правильное управление ресурсами при создании копий объектов. Это особенно важно, когда объект содержит динамически выделенные ресурсы, такие как память.

8. Билет 8.

8.1 Вопрос по Java. Явная и неявная субтипизация. Вызов конструктора базового класса. Операция приведения типа. Переопределение метода.

Определение 8.1.1. Тип данных A является подтипом типа данных B, если программный код, рассчитанный на обработку значений типа B, может быть корректно использован для обработки значений типа A.

Определение 8.1.2. Субтипизация (Subtype) – свойство ЯП, означающее возможность спользования подтипов в программах.

Замечание. В Java используется явная субтипизация

Определение 8.1.3. Явная субтипизация - это явное приведение объекта к типу его подтипа. Явное приведение необходимо, когда компилятор не может автоматически сделать вывод о типе во время компиляции. Если объект не является экземпляром подтипа, при выполнении программы может быть сгенерировано исключение ClassCastException.

Пример:

```
ParentClass parent = new ChildClass(); // Явное приведение ChildClass к ParentClass
```

Замечание. В этом примере ChildClass является подклассом ParentClass, и мы явно приводим объект типа ChildClass к типу ParentClass. Это позволяет нам присвоить объект ChildClass переменной типа ParentClass.

Замечание. В ЯП, поддерживающих неявную типизацию, решение о том, является ли A подтипом типа B, происходит на основе структуры значений этих типов.

Вызов конструктора базового класса в Java: При создании объекта производного класса в Java, конструктор базового класса вызывается автоматически перед выполнением конструктора производного класса. Это обеспечивает инициализацию состояния базового класса перед инициализацией состояния производного класса.

Пример:

```
public class BaseClass {
    public BaseClass() {
        // Конструктор базового класса
    }
}
```

```
public class DerivedClass extends BaseClass {
    public DerivedClass() {
        super(); // Вызов конструктора базового класса
        // Конструктор производного класса
    }
}
```

Переопределение метода позволяет классу-наследнику предоставить свою собственную реализацию метода, который уже определен в его базовом классе. Для переопределения метода в Java используется аннотация **@Override**.

Пример переопределения метода:

```
public class ParentClass {
    public void printMessage() {
        System.out.println("Hello, I'm the parent class");
    }
}

public class ChildClass extends ParentClass {
    @Override
    public void printMessage() {
        System.out.println("Hello, I'm the child class");
    }
}
```

Замечание. Переопределение методов позволяет классам-наследникам изменять поведение унаследованных методов, предоставляя свою собственную реализацию.

8.2 Вопрос по C++. Объекты в глобальной памяти. Объекты в полях других объектов.

В C++ объекты могут находиться как в глобальной памяти, так и в полях других объектов. Рассмотрим оба случая более подробно:

Объекты в глобальной памяти.

В C++ вы можете создавать объекты в глобальной области видимости, за пределами функций и классов. Эти объекты инициализируются до вызова функции main() и остаются в памяти на протяжении всего времени выполнения программы. Пример объекта в глобальной памяти:

```cpp
#include <iostream>

class MyClass {
public:
    MyClass() {
        std::cout << "Constructor called\n";
    }

    ~MyClass() {
        std::cout << "Destructor called\n";
    }
};

MyClass globalObject; // Объект в глобальной памяти

int main() {
    // Ваш код
    return 0;
}
```

Замечание. В этом примере класс MyClass имеет объект globalObject, который создается в глобальной области. Конструктор вызывается перед выполнением функции main(), а деструктор вызывается после завершения программы.

Объекты в полях других объектов.

В C++ объекты могут быть членами других объектов в качестве их полей. Когда вы создаете экземпляр класса, содержащего объекты-члены, память для этих объектов выделяется вместе с памятью для объекта-хозяина. Поля объектов могут быть инициализированы в конструкторе класса.

```cpp
#include <iostream>

class InnerClass {
public:
    InnerClass() {
        std::cout << "InnerClass constructor called\n";
    }
```

```
    ~InnerClass() {
        std::cout << "InnerClass destructor called\n";
    }
};

class OuterClass {
public:
    InnerClass innerObject; // Объект в поле другого объекта

    OuterClass() {
        std::cout << "OuterClass constructor called\n";
    }

    ~OuterClass() {
        std::cout << "OuterClass destructor called\n";
    }
};

int main() {
    OuterClass outerObject; // Создание объекта класса OuterClass
    return 0;
}
```

Замечание. В этом примере класс OuterClass содержит объект innerObject класса InnerClass в качестве своего поля. При создании объекта outerObject в функции main(), память выделяется как для самого объекта outerObject, так и для его поля innerObject. Конструкторы вызываются в правильном порядке при создании объекта, а деструкторы вызываются при завершении программы.

9. Билет 9.

9.1 Вопрос по Java. Абстрактные классы.

Определение 9.1.1. В Java абстрактный класс - это класс, который не может быть инстанциирован напрямую, то есть нельзя создать объект типа абстрактного класса. Абстрактные классы обычно используются в качестве базовых классов для других классов и содержат абстрактные методы, которые должны быть реализованы в производных классах.

Пример:

```
abstract class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound();
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Dog " + name + " barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.makeSound();
    }
}
```

Замечание. В этом примере Animal - абстрактный класс с полем name и абстрактным методом makeSound(). Класс Dog наследуется от Animal и реализует абстрактный метод makeSound(). В методе main() создается экземпляр класса Dog с именем "Buddy"и вызывается метод makeSound(), который выводит сообщение "Dog Buddy barks".

9.2 Вопрос по C++. Объявление деструктора. Объекты в автоматической памяти. Автоматический вызов деструктора.

Определение 9.2.1. Деструктор – это экземплярный метод, предназначенный для освобождения ресурсов, принадлежащих объекту, непосредственно перед уничтожением этого объекта. Деструктор позволяет освобождать ресурсы, выделенные объекту во время его жизни, например, освобождать память, закрывать файлы.

Для объявления деструктора в C++ используется тильда ( ) перед именем класса, и деструктор должен иметь тот же самый идентификатор, что и класс. Вот пример объявления деструктора:

```
class MyClass {
public:
    // Конструктор класса
    MyClass() {
        // Код конструктора
    }

    // Деструктор класса
    ~MyClass() {
        // Код деструктора
    }

    // Другие методы класса
};
```

Деструктор вызывается автоматически при выходе объекта из области видимости, в которой он был создан, или при вызове оператора delete для динамически созданного объекта.
Например:

```
void someFunction() {
    MyClass obj; // Объект создается в автоматической памяти

    // Код, использующий объект obj
```

```
} // При выходе из области видимости вызывается деструктор obj
```

Замечание. В данном примере объект obj создается в автоматической памяти внутри функции someFunction(). Когда выполнение программы выходит из области видимости функции someFunction(), деструктор MyClass() вызывается автоматически для объекта obj, что позволяет освободить ресурсы, занятые этим объектом.

Определение 9.2.2. Автоматическая переменная — это локальная переменная, которая выделяется и освобождается автоматически, когда поток выполнения программы входит и выходит из области действия переменной.

10. Билет 10.

10.1 Вопрос по Java. Объявление интерфейсов. Реализация интерфейсов. Наследование интерфейсов.

В Java интерфейс представляет собой контракт, описывающий набор методов, которые класс должен реализовать. Интерфейсы определяются с помощью ключевого слова interface и могут содержать только абстрактные методы (без реализации), константы и методы по умолчанию (с реализацией).

Пример объявления интерфейса:

```
public interface MyInterface {
    void myMethod(); // Абстрактный метод

    int CONSTANT = 10; // Константа

    default void defaultMethod() {
        // Метод с реализацией по умолчанию
    }
}
```

Интерфейсы в Java реализуются классами с помощью ключевого слова **implements**. Класс, реализующий интерфейс, должен предоставить реализацию **всех** абстрактных методов, объявленных в интерфейсе.

Замечание. Можно наследовать несколько интерфесов через запятую после ключевого слова implements

Пример наследования класса от интерфейса:

```
public class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        // Реализация абстрактного метода
    }
}
```

Замечание. В этом примере класс MyClass реализует интерфейс MyInterface и предоставляет реализацию абстрактного метода myMethod(). Класс также может предоставлять свои собственные методы и поля, дополнительные к интерфейсу.

Интерфейсы также могут наследоваться друг от друга с помощью ключевого слова **extends**. Класс, реализующий интерфейс-наследник, должен предоставить реализацию всех абстрактных методов во всех унаследованных интерфейсах.

Пример наследования интерфейсов:

```
public interface MyInterface2 extends MyInterface {
    void anotherMethod(); // Дополнительный абстрактный метод
}
```

Замечание. В этом примере интерфейс MyInterface2 наследуется от MyInterface и добавляет еще один абстрактный метод anotherMethod(). Класс, реализующий MyInterface2, должен предоставить реализацию и для myMethod() и для anotherMethod().

Интерфейсы в Java позволяют определять контракты, которые классы должны соблюдать. Они являются важным средством для достижения полиморфизма и проектирования гибких и расширяемых систем.

10.2 Вопрос по C++. Создание объектов в динамической памяти. Создание массивов в динамической памяти. Удаление объектов и массивов.

В C++ вы можете создавать объекты в динамической памяти с использованием оператора new. Это позволяет вам явно управлять временем жизни объектов и создавать их во время выполнения программы. Затем, после использования, вы должны удалить объекты из динамической памяти с помощью оператора delete. Аналогично, для создания массивов в динамической памяти используется оператор new[], а для удаления массивов — оператор delete[].

Пример создания объекта в динамической памяти и его удаления:

```
MyClass* myObject = new MyClass(); // Создание объекта в динамической памяти

// Использование объекта...

delete myObject; // Удаление объекта из динамической памяти
```

Замечание. В этом примере класс MyClass создается в динамической памяти с помощью оператора new, и указатель myObject хранит адрес этого объекта. После использования объекта его можно удалить с помощью оператора delete, чтобы освободить выделенную память.

Пример создания массива в динамической памяти и его удаления:

```
int size = 5;
int* myArray = new int[size]; // Создание массива в динамической памяти

// Использование массива...

delete[] myArray; // Удаление массива из динамической памяти
```

Замечание. В этом примере массив myArray создается в динамической памяти с помощью оператора **new[ ]**, и размер массива определяется переменной size. После использования массива он удаляется с помощью оператора **delete[ ]**.

11. Билет 11.

11.1 Вопрос по Java. Экземплярные и статические вложенные классы. Локальные классы. Анонимные классы.

В **Java** есть несколько типов вложенных классов, включая экземплярные и статические вложенные классы, локальные классы и анонимные классы.

1. **Экземплярные и статические вложенные классы**

   Экземплярные вложенные классы являются членами другого класса и имеют доступ ко всем членам внешнего класса, включая приватные. Статические вложенные классы также являются членами внешнего класса, но объявляются с модификатором **static**. Они не имеют доступа к нестатическим членам внешнего класса, только к статическим.

2. **Локальные классы**

   Локальные классы объявляются внутри блока кода, например, внутри метода или конструктора. Они могут иметь доступ к локальным переменным и параметрам метода, но только тем, которые объявлены как **final** или эффективно финализированы (не изменяются после инициализации).

3. **Анонимные классы**

   Анонимные классы - это безымянные классы, которые создаются и инстанциируются одновременно внутри выражения. Они могут быть использованы для создания одноразовых классов, реализующих интерфейсы или расширяющих классы. Обычно они создаются в качестве подклассов или реализаций интерфейсов.

Примеры использования:

```
// Экземплярный вложенный класс
class Outer {
    class Inner {
        // ...
    }
}

// Статический вложенный класс
class Outer {
    static class Nested {
        // ...
```

```
    }
}

// Локальный класс
class Outer {
    void method() {
        class Local {
            // ...
        }
    }
}

// Анонимный класс
interface MyInterface {
    void doSomething();
}

class Outer {
    void method() {
        MyInterface obj = new MyInterface() {
            @Override
            public void doSomething() {
                // ...
            }
        };
    }
}
```

11.2 Вопрос по C++. Объявление конструкторов.

В C++ конструкторы объявляются внутри класса и используются для инициализации объектов этого класса. Конструкторы имеют тот же имя, что и класс, и не возвращают значения, включая void.

Существуют различные виды конструкторов в C++:

1. **Конструктор по умолчанию**

Конструктор по умолчанию создает объект без аргументов. Если класс не имеет явно определенного конструктора, компилятор автоматически создает конструктор по умолчанию. Если вы определяете свой конструктор с аргументами, конструктор по умолчанию не будет автоматически создан компилятором.

2. **Параметризованный конструктор**

   Параметризованный конструктор принимает аргументы и используется для инициализации объекта с заданными значениями. Он позволяет передать значения аргументов в конструктор при создании объекта.

3. **Конструктор копирования**

   Конструктор копирования создает новый объект, инициализированный существующим объектом того же класса. Он используется для создания копии объекта и обычно принимает ссылку на существующий объект в качестве аргумента.

Примеры объявления конструкторов в C++:

```
class MyClass {
public:
    // Конструктор по умолчанию
    MyClass();

    // Параметризованный конструктор
    MyClass(int value);

    // Конструктор копирования
    MyClass(const MyClass& other);
};

// Определение конструкторов

// Конструктор по умолчанию
MyClass::MyClass() {
    // ...
}

// Параметризованный конструктор
MyClass::MyClass(int value) {
```

```
    // ...
}

// Конструктор копирования
MyClass::MyClass(const MyClass& other) {
    // ...
}
```

При использовании конструкторов в C++ вы можете создавать и инициализировать объекты с помощью соответствующих конструкторов в момент их создания. Например:

```
MyClass obj1;              // Использование конструктора по умолчанию
MyClass obj2(10);          // Использование параметризованного конструктора
MyClass obj3 = obj2;       // Использование конструктора копирования
```

12. Билет 12.

12.1 Вопрос по Java. Функциональные интерфейсы и лямбда-выражения.

Определение 12.1.1. Функциональный интерфейс - это интерфейс с одним методом. Лямбда выражение может являться реализацией этого интерфейса.

Вот пример функционального интерфейса *Predicate* в **Java**:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Замечание. Интерфейс Predicate определяет метод test, который принимает один аргумент и возвращает логическое значение. Он используется для проверки условия на заданном объекте.

Теперь рассмотрим лямбда-выражения, которые являются компактным способом определения реализации функциональных интерфейсов в Java. Каждое лямбда-выражение состоит из аргументов, стрелки -> и тела выражения.
Вот **примеры** использования лямбда-выражений:

```
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        // Пример использования Predicate
        Predicate<Integer> isPositive = num -> num > 0;
        boolean result = isPositive.test(5);
    }
}
```

12.2 Вопрос по C++. Объявление полей. Определения статических полей. Объявление методов. Виртуальные и абстрактные методы.

В C++, поля класса объявляются внутри его определения. Они представляют переменные, которые хранят данные для каждого объекта класса. Поля класса могут быть различных типов и модификаторов доступа (public, private, protected).

Пример объявления полей класса в C++:

```
class MyClass {
public:
    int myField;         // Поле класса типа int с модификатором доступа public
    double anotherField; // Поле класса типа double с модификатором доступа public

private:
    // Поле класса типа std::string с модификатором доступа private
    std::string name;
};
```

Статические поля класса общие для всех объектов этого класса. Они объявляются с использованием ключевого слова static и обычно инициализируются вне определения класса.
Пример объявления и определения статического поля класса в C++:

```
class MyClass {
public:
    static int count; // Статическое поле класса типа int
};

// Определение статического поля класса
int MyClass::count = 0;
```

Методы класса объявляются внутри его определения и представляют функции, которые могут быть вызваны на объектах этого класса. Методы могут иметь различные модификаторы доступа (public, private, protected) и могут принимать аргументы и возвращать значения.
Пример объявления методов класса в C++:

```
class MyClass {
public:
    // Объявление метода без аргументов и возвращаемого значения
    void myMethod();
     // Объявление метода с двумя аргументами и возвращаемым значением
    int anotherMethod(int x, int y);

private:
    // Объявление приватного метода без аргументов и возвращаемого значения
```

```
    void privateMethod();
};
```

Виртуальные методы в C++ используются для обеспечения полиморфизма. Они объявляются с помощью ключевого слова **virtual** в базовом классе и могут быть переопределены в производных классах.

Пример объявления виртуального метода в C++:

```
class MyBaseClass {
public:
    virtual void myMethod();    // Объявление виртуального метода
};

void MyBaseClass::myMethod() {
    // Реализация виртуального метода
}
```

Абстрактные методы в C++ объявляются в абстрактном классе. **Абстрактный класс** - это класс, содержащий один или несколько абстрактных методов, которые не имеют реализации в самом классе, но должны быть реализованы в производных классах.

Пример объявления абстрактного метода и абстрактного класса в C++:

```
class MyAbstractClass {
public:
    // Объявление абстрактного метода с помощью "= 0"
    virtual void myAbstractMethod() = 0;
};

// Производный класс должен реализовать абстрактный метод
class MyDerivedClass : public MyAbstractClass {
public:
    void myAbstractMethod() override {
        // Реализация абстрактного метода
    }
};
```

13. Билет 13.

13.1 Вопрос по Java. Необходимость обобщений. Контейнерные классы. Основная проблема при использовании необобщённых контейнерных классов.

Обобщения в **Java** представляют собой механизм, который позволяет создавать обобщенные классы, интерфейсы и методы, которые могут работать с различными типами данных. Они позволяют параметризовать типы данных, которые используются в контейнерах и алгоритмах, чтобы обеспечить типовую безопасность и повысить переиспользование кода.
Обобщения в **Java** имеют несколько преимуществ и позволяют:

1. **Безопасность типов**. Обобщения позволяют проверять типы данных во время компиляции, что помогает предотвратить ошибки типизации на этапе выполнения программы.
2. **Повышение переиспользования кода**. Обобщенные классы и методы могут быть использованы с различными типами данных без необходимости дублирования кода.
3. **Улучшение производительности**. Обобщения позволяют избежать ненужного приведения типов (casting), что может привести к улучшению производительности программы.

Основная проблема при использовании необобщенных контейнерных классов в Java заключается в потенциальных ошибках типизации на этапе выполнения программы.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList myList = new ArrayList();
        myList.add("Hello");
        myList.add(10);

        // тут мы получим Exception
        int a = (int)(myList.get(0));
        System.out.println(a);
    }
}
```

13.2 Вопрос по C++. Объявление класса. Секции в объявлении класса.

В C++, объявление класса включает имя класса, список членов класса и опциональные секции, такие как секции доступа и секцию наследования. Объявление класса обычно размещается в заголовочных файлах (.h или .hpp), а его определение - в файле реализации (.cpp).
Вот **пример** объявления класса в C++:

```
class MyClass {
public:
    // Публичные члены класса

private:
    // Приватные члены класса

protected:
    // Защищенные члены класса
};
```

Определение класса может содержать реализацию методов и дополнительные детали, но объявление класса служит для представления структуры класса и его доступных членов.
В объявлении класса есть несколько секций:

1. **Секция доступа**: public, private и protected определяют доступность членов класса для других частей программы. Члены, объявленные в public секции, доступны извне класса. Члены, объявленные в private секции, доступны только внутри класса. Члены, объявленные в protected секции, доступны внутри класса и его производных классов (наследников).

2. **Члены класса**: Это переменные и функции, которые определяются внутри класса. Они могут быть объявлены в любой секции доступа и определяют поведение и состояние объектов класса.

3. **Секция наследования**: При наследовании одного класса от другого используется синтаксис class DerivedClass : public BaseClass. Здесь DerivedClass является производным классом (наследником), а BaseClass - базовым классом. Секция наследования указывает, какие члены базового класса будут доступны в производном классе.

Пример объявления класса с использованием секций:

```
class Shape {
public:
    // Публичные члены класса Shape

protected:
    // Защищенные члены класса Shape

private:
    // Приватные члены класса Shape
};

class Rectangle : public Shape {
public:
    // Публичные члены класса Rectangle

private:
    // Приватные члены класса Rectangle
};
```

Замечание. В этом примере класс Shape объявлен с публичными, защищенными и приватными секциями. Класс Rectangle наследуется от класса Shape с помощью секции наследования public. Это означает, что публичные члены Shape будут доступны внутри класса Rectangle.

14. Билет 14.

14.1 Вопрос по Java. Понятие обобщённого класса. Ограниченные обобщённые классы.

В Java, **обобщённый класс** представляет собой класс, который параметризуется типом данных. Обобщённые классы позволяют создавать классы, методы или интерфейсы, которые могут работать с различными типами данных без необходимости повторного написания кода для каждого типа.
Обобщённый класс объявляется с использованием параметра типа (type parameter), который указывается в угловых скобках (<>). Параметр типа представляет собой имя, которое используется внутри класса для указания типа данных. Обычно используются одиночные заглавные буквы, такие как T, E, K и V, но можно использовать любое допустимое имя.
Вот **пример** обобщённого класса в Java:

```
public class Box<T> {
    private T contents;

    public void setContents(T contents) {
        this.contents = contents;
    }

    public T getContents() {
        return contents;
    }
}
```

Замечание. В этом примере класс Box объявлен как обобщённый с параметром типа T. Параметр типа T может быть любым типом данных, и его значения будут определены при создании экземпляра класса Box. Методы setContents и getContents могут работать с типом T, позволяя сохранять и извлекать значения заданного типа.

Ограниченные обобщённые классы позволяют ограничить диапазон допустимых типов данных, которые могут быть использованы в обобщённом классе. Это достигается с помощью ключевого слова extends или super, за которым следует ограничивающий тип.
Вот пример ограниченного обобщённого класса, который ограничен типом Number:

```
public class NumericBox<T extends Number> {
    private T value;
```

```
    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

Замечание. В этом примере класс NumericBox является ограниченным обобщённым классом с параметром типа T, который ограничен типом Number. Это означает, что тип T может быть только подтипом Number или самим типом Number. Это позволяет использовать операции и методы, доступные для типа Number внутри класса NumericBox.

14.2 Вопрос по C++. Ссылки как возвращаемые значения функций. Константные ссылки.

В C++, ссылки могут использоваться как возвращаемые значения функций, а также могут быть объявлены как константные ссылки. Давайте рассмотрим каждый случай подробнее.

1. **Ссылки как возвращаемые значения функций**

   В C++ функция может возвращать ссылку на объект вместо его копии. Это может быть полезно для избегания ненужного копирования объектов и эффективной передачи объекта для дальнейших манипуляций. Часто ссылки возвращаются вместе с перегруженными операторами, чтобы обеспечить возможность цепочки операций.

   Вот **пример** функции, возвращающей ссылку:

   ```
   int& increment(int& value) {
       value++;
       return value;
   }
   ```

   В этом примере функция increment принимает ссылку на int в качестве аргумента и увеличивает его на 1. Затем она возвращает ссылку на измененное значение. Это позволяет использовать функцию increment в цепочке операций или сохранить измененное значение.

```
int x = 5;
// Изменение значения x на 10 с помощью ссылки
increment(x) = 10;
```

2. **Константные ссылки**

   В C++ можно объявить ссылки как константные, чтобы указать, что объект, на который они ссылаются, не может быть изменен. Константные ссылки часто используются для передачи аргументов в функции, чтобы избежать нежелательных изменений и для обеспечения эффективности (избегая копирования больших объектов).

   Вот **пример** объявления константной ссылки:

```
void modifyValue(const int& value) {
    // Недопустимо! Нельзя изменять значение через константную ссылку
    // value = 10;
    // ...
}

int main() {
    int x = 5;
    modifyValue(x);  // Передача x по константной ссылке
    // ...
    return 0;
}
```

15. Билет 15.

15.1 Вопрос по Java. Ковариантность массивов. Инвариантность обобщённых классов. Шаблоны обобщённых классов. Особенности шаблонов.

Ковариантность массивов в Java означает, что массивы являются ковариантными по отношению к их типам элементов. Это означает, что если B является подтипом A, то массив типа B[ ] также является подтипом массива типа A[ ]. Ковариантность массивов позволяет присваивать массивы более специфичных типов массивам менее специфичных типов.

Пример:

```
class Fruit {}
class Apple extends Fruit {}

public static void main(String[] args) {
    Apple[] apples = new Apple[5];
    Fruit[] fruits = apples;  // Ковариантность массивов

    fruits[0] = new Apple();  // Допустимо
    fruits[1] = new Fruit();  // Ошибка
}
```

Замечание. В этом примере массив apples типа Apple[] присваивается массиву fruits типа Fruit[], потому что Apple является подтипом Fruit. Однако, при попытке присвоить объект типа Fruit массиву fruits, будет вызвано исключение ArrayStoreException, потому что массив fruits по-прежнему фактически является массивом Apple[]. Это происходит из-за принципа ковариантности, который не применяется к элементам массива, а только к самим массивам.

Инвариантность обобщённых классов В отличие от ковариантности массивов, обобщённые классы в Java являются инвариантными по отношению к их типам параметров. Это означает, что если B является подтипом A, то MyClass<B> и MyClass<A> не имеют отношения подтипов и несовместимы.

Пример:

```
class MyClass<T> {}

MyClass<Apple> myApples = new MyClass<Apple>();
MyClass<Fruit> myFruits = myApples;  // Ошибка компиляции
```

Замечание. В этом примере MyClass<Apple> не является подтипом MyClass<Fruit>, несмотря на то, что Apple является подтипом Fruit. Это связано с инвариантностью обобщённых типов в Java, где параметризованные типы не являются ковариантными или контравариантными.

Шаблоны обобщённых классов: Шаблоны обобщённых классов в **Java** предоставляют возможность создания классов, методов и интерфейсов, которые могут работать с различными типами данных. Шаблоны обобщённых классов позволяют параметризовать типы данных, которые будут использоваться внутри класса.

```
class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}

public static void main(String[] args) {
    Box<String> stringBox = new Box<>();
    stringBox.setItem("Hello");
    String item = stringBox.getItem();  // Возвращается значение типа String

    Box<Integer> intBox = new Box<>();
    intBox.setItem(42);
    int intValue = intBox.getItem();  // Возвращается значение типа int
}
```

Замечание. В этом примере класс Box<T> является шаблоном обобщённого класса, где T является параметром типа. Это позволяет создавать экземпляры Box с различными типами данных. Класс Box имеет методы для установки и получения значения, и тип данных T будет определён во время создания экземпляра класса.

Особенность шаблонов - можно объявить переменную типа шаблон, но невозможно создать его объект.

```
Stack<?> x; // OK
x = new Stack<?>(); // Error!
```

15.2 Вопрос по C++. Объявление переменных ссылочного типа. Инициализация и использование ссылок. Ссылки в формальных параметрах функций.

Объявление переменных ссылочного типа. Для объявления переменной ссылочного типа используется амперсанд (&) после типа данных. Например:

```
int a = 10;
int& ref = a;
```

Замечание. В этом примере мы объявляем переменную ref как ссылку на int и инициализируем ее значением переменной a. Обратите внимание, что ссылка должна быть инициализирована при объявлении.

Инициализация ссылок. Ссылки должны быть инициализированы при объявлении и могут быть связаны только с одним объектом. Например:

```
int a = 10;
int& ref = a;  // Инициализация ссылки

int b = 20;
ref = b;  // Неправильно, это присваивание значений, а не изменение связанного объекта
```

Замечание. В этом примере мы инициализируем ссылку ref значением переменной a. Обратите внимание, что после инициализации ссылка ref становится альтернативным именем для переменной a. Изменение значения ref также изменяет значение a.

Использование ссылок. После инициализации ссылку можно использовать так же, как и обычную переменную. Например:

```
int a = 10;
int& ref = a;

ref += 5;  // Изменение значения через ссылку

std::cout << a << std::endl;  // Вывод: 15
```

Замечание. В этом примере мы изменяем значение переменной a через ссылку ref и выводим измененное значение переменной a.

Ссылки в формальных параметрах функций. Ссылки часто используются в формальных параметрах функций для передачи аргументов по ссылке, а не по значению. Например:

```
void increment(int& value) {
    value++;
}

int main() {
    int a = 10;
    increment(a);
    std::cout << a << std::endl;  // Вывод: 11
    return 0;
}
```

Замечание. В этом примере функция increment принимает ссылку на int в качестве параметра. Изменение значения параметра value внутри функции также изменяет значение переменной a в main().