**OOP, Classes and Objects, Inheritance, Class Object, Interfaces**

1. **Дайте развернутое объяснение трем концепциям ООП.**

**Инкапсуляция** – принцип, объединяющий данные в код, позволяющий защищать данные от прямого внешнего доступа и неправильного использования. Доступ к данным класса возможен только через методы этого же класса.

**Наследование** – процесс, при котором один класс может наследовать свойства другого класса и добавлять к ним свойства и методы, характерные только для него.

**Полиморфизм** - механизм, использующий одно и тоже имя метода для решения похожих, но несколько отличающихся задач в различных объектах при наследовании от одного суперкласса.

1. **Опишите процедуру инициализации полей класса и полей экземпляра класса. Когда инициализируются поля класса, а когда – поля экземпляров класса. Какие значения присваиваются полям по умолчанию? Где еще в классе полям могут быть присвоены начальные значения?**

Первый способ инициализации – это присвоить значения переменным при их создании.

Class Student{

String name = “Bill”; int age = 20;}

Второй способ – инициализировать поля через конструктора класса.

Class Student{

String name ;

int age ;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

Поля объекта инициализируются во время конструирования экземпляра класса. Инициализация полей класса произойдет во время загрузки класса.

По умолчанию объектам присваивается null, цифровым типам 0, значениям типа boolean - false, char = ' ' переменной типа char присваивается нуль-символ, который отображается в виде символа «пробел».

1. **Приведите правила, которым должен следовать компонент java-bean.**

Компоненты **JavaBean** – это многократно используемые классы Java.

Чтобы класс работал как bean, он должен соответствовать определенным соглашениям об именах методов, конструкторе и поведении.

Правила:

 класс должен иметь конструктор без параметров, с модификатором доступа public.

 свойства класса должны быть доступны через get, set и другие методы, которые должны подчиняться стандартному соглашению об именах. Это позволяет инструментам автоматически определять и обновлять содержимое bean.

 класс должен быть Serializable. Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние bean независимым от платформы и виртуальной машины способом.

 класс должен иметь переопределенные методы equals(), hashCode() и toString().

1. **Дайте определение перегрузке методов. Как вы думаете, чем удобна перегрузка методов? Укажите, какие методы могут перегружаться, и какими методами они могут быть перегружены? Можно ли перегрузить методы в базовом и производном классах? Можно ли private метод базового класса перегрузить public методов производного? Можно ли перегрузить конструкторы, и можно ли при перегрузке конструкторов менять атрибуты доступа у конструкторов?**

Перегрузка методов – это создание нескольких версий одного метода с одним и тем же именем, но разным списком параметров.

Перегрузка методов удобна, тем что не нужно создавать новых методов, для работы с разным набором входных параметров.

Методы могут перегружаться методами с различными по типу и количеству параметрами. Статические методы могут перегружаться нестатическими методами, и наоборот.

Чтобы внутри подкласса обратиться к методу суперкласса, нужно уточнить имя метода, словом super.

При переопределении права доступа к методу можно только расширить. Поэтому метод private базового класса можно перегрузить public методом произвольного.

Перегрузка конструкторов реализуется так же, как и перегрузка методов. Перегрузка конструкторов позволяет установить различные способы начальной инициализации объектов класса.

1. **Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните правила, которым следует компилятор при разрешении перегрузки; в том числе, если методы перегружаются примитивными типами, между которыми возможно неявное приведение или ссылочными типами, состоящими в иерархической связи.**

Раннее связывание (статическое) происходит во время компиляции, т.е. код знает, какой метод вызвать после компиляции исходного кода на Java в файлы классов.

Позднее связывание (динамическое) происходит во время выполнения, после запуска программы виртуальной машиной Java. В этом случае то, какой метод вызвать, определяется конкретным объектом, так что в момент компиляции информация недоступна, ведь объекты создаются во время выполнения.

Перегрузка реализует раннее связывание, т.е. версия вызываемого метода определяется на этапе компиляции.

При перегрузке примитивными типами если у нас будет метод, принимающий значения типа short, то он и выполнится если мы передаем тип short. Если нет метода с такими входными данными, то выполняется расширяющее преобразование примитивов. Выбирается метод, для которого нужно наиболее близкое расширяющее преобразование. В данном случае, например, до int.

Расширяющее преобразование примитивов имеет приоритет над автоупаковкой/автораспаковкой.

Автоупаковка/автораспаковка имеет приоритет перед методом с переменным количеством аргументов.

Сужающее преобразование примитивов автоматически не выполняется.

Если перегрузка осуществляется ссылочными типами, состоящими в иерархической связи.

class Main {

    static class Monster {}

    static class Goblin extends Monster {}

    static class Hobgoblin extends Goblin {}

    static void method1(Monster obj) {

        System.out.println("1");

    }

    static void method1(Goblin obj) {

        System.out.println("2");

    }

     public static void main(String[] args) {

        Monster obj = new Hobgoblin();

        method1(obj);

    }

}

В консоль будет выведено 1. Так как переменная obj имеет тип Monster. Выбор перегруженного метода осуществляется на этапе компиляции, а не на этапе выполнения, поэтому метод выбирается на основе типа объявленной переменной.

1. **Объясните, как вы понимаете, что такое неявная ссылка this? В каких методах эта ссылка присутствует, а в каких – нет, и почему?**

Каждый экземпляр класса (объект) имеет неявную ссылку this на себя, которая передается неявно и нестатическим методам класса. После этого каждый метод знает, какой объект его вызвал. Если мы обращаемся к атрибуту, например, с именем id в методах можно писать this.id, но это не обязательно, так как записи id и this.id будут равносильны. Но если в методе есть локальная переменная или параметр с таким же именем, как и поле класса, то для обращения к полю класса необходимо указывать this.id. Без указания обращение будет производиться к локальной переменной.

1. **Что такое финальные поля, какие поля можно объявить со спецификатором final? Где можно инициализировать финальные поля?**

Модификатор final используется для определения констант в качестве члена класса, локальной переменной или параметра метода. Методы, объявленные как final, нельзя замещать в подклассах. Для классов, объявленных со спецификатором final, нельзя создавать подклассы.

Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке класса, заключенном в {}, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса.

1. **Что такое статические поля, статические финальные поля и статические методы. К чему имеют доступ статические методы? Можно ли перегрузить и переопределить статические методы? Наследуются ли статические методы?**

Поле данных, объявленное в классе как static, является общим для всех объектов класса и называется переменной класса.

Для работы со статическими атрибутами используются статические методы, объявленные со спецификатором static.

По причине недоступности указателя this статические поля и методы не могут обращаться к нестатическим полям и методам напрямую, так как они не «знают», к какому объекту относятся, да и сам экземпляр класса может быть не создан. Для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

Переопределение статических методов невозможно, так как обращение к статическому атрибуту или методу осуществляется посредством задания имени класса, которому они принадлежат.

Статические методы могут быть перегружены так же, как и любой другой метод.

1. **Что такое логические и статические блоки инициализации? Сколько их может быть в классе, в каком порядке они могут быть размещены и в каком порядке вызываются?**

Статический блок инициализации служит для инициализации статических переменных.

static {

   //...

}

Статический блок — это, в сущности, конструктор для всего класса. Он ставится между определениями полей и функциями класса. Команды будут выполняться в одном из двух случаев, том, который наступит раньше:

1. При создании первого объекта класса в процессе работы программы, перед запуском конструктора.
2. При первом вызове статической функции, перед выполнением.
3. **Что представляют собой методы с переменным числом параметров, как передаются параметры в такие методы и что представляет собой такой параметр в методе? Как осуществляется выбор подходящего метода, при использовании перегрузки для методов с переменным числом параметров?**

Методы с переменным числом параметров принимают в метод нефиксированное число параметров, что позволяет отказаться от создания сложного объекта для последующей передачи его в метод.

Список параметров в методе выглядит следующим образом (Тип… args). А в случае необходимости передачи параметров других типов:

(Тип1 t1, Тип2 t2, Тип… args)

Методы с переменным числом аргументов могут быть перегружены:

void methodName(Integer...args) { }

void methodName(int x1, int x2) { }

Такие реализации отличаются между собой типом получаемых значений.

1. **Чем является класс Object? Перечислите известные вам методы класса Object, укажите их назначение.**

Класс **Object** является суперклассом для всех классов. В классе **Object** определен набор методов, которые наследуются всеми классами:

**protected Object clone()** — создает и возвращает копию вызывающего объекта; **public  boolean  equals(Object  ob)**  —  предназначен для использования и пере определения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов одного и того же типа**; public Class<? extends Object> getClass()** — возвращает объект типа Class; **protected void finalize()** — автоматически вызывается сборщиком мусора (garbage collection) перед уничтожением объекта; **public int hashCode()** — вычисляет и возвращает хэш-код объекта (число, в общем случае вычисляемое на основе значений полей объекта); **public String toString()** — возвращает представление объекта в виде строки.

1. **Что такое хэш-значение? Объясните, почему два разных объекта могут сгенерировать одинаковые хэш-коды?**

В классе Object есть метод int hashCode(), его можно переопределить и как правило в каждом классе он возвращает число, являющееся уникальным идентификатором объекта.

Все одинаковые по содержанию объекты имеют одного типа должны иметь одинаковые хэш-коды.

Два разных объекта могут содержать разные хэш-коды, в том случае, если количество созданных объектов превысило допустимое значение для int, тогда возникает ситуация, что объекты, созданные после превышения, будут с одинаковыми хэш-кодами.

1. **Как вы думаете, для чего используется наследование классов в java-программе? Приведите пример наследования. Как вы думаете, поля и методы, помеченными модификатором доступа private, наследуются?**

С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Производный класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором private.

Если метод или поле имеют модификатор доступа private, то они доступны только в рамках класса. Такие члены класса не наследуются, поэтому их невозможно заместить в подклассах.

Пример наследования. Имеется класс Person описывающий человека:

class Person {

    private String name;

    public String getName(){ return name; }

    public Person(String name){

        this.name=name;

    }

    public void display(){

        System.out.println("Name: " + name);

    }

}

Добавим еще одни класс который описывает сотрудника предприятия Employee и наследуемся от класса Person.

class Employee extends Person{

      private String company;

    public Employee(String name, String company) {

         super(name);

        this.company=company;

    }

    public void work(){

        System.out.printf("%s works in %s \n", getName(), company);

    }

}

У класса Employee поле String company, так же метод work. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

1. **Укажите, как вызываются конструкторы при создании объекта производного класса? Что в конструкторе класса делает оператор super()? Возможно ли в одном конструкторе использовать операторы super() и this()?**

Ключевое слово super применяется для обращения к конструктору суперкласса и для доступа к полю или методу суперкласса.

super(список\_параметров);

Конструкция super() имеет такие же ограничения, как и конструкция this(): если она используется, то вызов super() должен быть первым оператором в конструкторе, и он может использоваться только в конструкторе. Это влечет за собой то, что вызовы this() и super() не могут встретиться в одном и том же конструкторе. Конструкция this() используется для связывания конструкторов в одном и том же классе, и самый последний конструктор в этой последовательности может вызвать конструктор суперкласса с помощью конструкции super(). Итак, конструкция this() приводит к формированию цепочки конструкторов в одном и том же классе, a super() приводит к связыванию конструкторов подкласса с конструкторами суперкласса. Использование цепочки гарантирует, что все конструкторы суперкласса будут вызваны, начиная с конструктора класса, объект которого порождается, и далее вверх до вершины по иерархии наследования, т.е. всегда до класса Object. Обратите внимание, что тело конструкторов выполняется в порядке обратном порядку их вызова, так как super() может быть только первым оператором в конструкторе. Это гарантирует, что конструктор из класса Object выполняется первым, а за ним конструкторы других классов вниз по иерархии наследования вплоть до класса, от которого порождается объект.

1. **Объясните, как вы понимаете утверждения: “ссылка базового класса может ссылаться на объекты своих производных типов” и “объект производного класса может быть использован везде, где ожидается объект его базового типа”. Верно ли обратное и почему?**

Ссылка базового класса может ссылаться на объекты своих производных типов. Например, у нас есть класс Animal и от него наследуется класс Dog. Тогда мы может создать переменную типа Animal, которая будет ссылаться на объект класса Dog.

Animal animal = new Dog();

Обратное будет неверно так как мы не сможем создать ссылку типа Dog на объект Animal, это вызовет ошибку.

Родительский класс определяет поведение, которое реализуют все его наследники, везде где используется базовый класс используются и его подклассы.

1. **Что такое переопределение методов? Как вы думаете, зачем они нужны? Можно ли менять возвращаемый тип при переопределении методов? Можно ли менять атрибуты доступа при переопределении методов? Можно ли переопределить методы в рамках одного класса?**

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, то метод подкласса переопределяет метод суперкласса. Когда переопределённый метод вызывается из своего подкласса, он всегда будет ссылаться на версию этого метода, определённую подклассом. А версия метода из суперкласса будет скрыта.

**У переопределенного метода должны быть тот же тип возвращаемого значения, что и у метода родителя.**

**Модификатор доступа у переопределенного метода также не может отличаться от «оригинального».**

1. **Определите правило вызова переопределенных методов. Можно ли статические методы переопределить нестатическими и наоборот?**

Переопределенный метод вызывается в соответствии с объектом, которому принадлежит метод, а не по типу ссылки.

Статические методы можно перегружать и «переопределять» в подклассах, но их доступность всегда зависит от типа ссылки и атрибута доступа, и никогда — от типа самого объекта

1. **Какие свойства имеют финальные методы и финальные классы? Как вы думаете, зачем их использовать?**

Основное свойство, что нельзя такие методы переопределить в классе наследнике, если в суперклассе он объявлен со спецификатором final. Метод с final говорит о том, что версия метода окончательная. Такое объявление гарантирует обращение именно к этой реализации метода.

От класса со спецификатором final нельзя наследоваться, а также все методы этого класса по умолчанию приобретают final.

Применение **final** в объявлениях классов и методов способно повысить уровень безопасности кода. Если класс снабжен модификатором **final**, никто не в состоянии расширить класс и, вероятно, нарушить его.

1. **Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникать при явном преобразовании ссылочных типов.**

Для примера имеется следующая иерархия классов:

**public class Program**{

    public static void main(String[] args) {

        Person tom = new Person("Tom");

        tom.display();

        Person sam = new Employee("Sam", "Oracle");

        sam.display();

        Person bob = new Client("Bob", "DeutscheBank", 3000);

        bob.display();

    }

}

// класс человека

**class Person** {

    private String name;

    public String getName() { return name; }

    public Person(String name){

        this.name=name;

    }

    public void display(){

        System.out.printf("Person %s \n", name);

    }

}

// служащий некоторой компании

**class Employee extends Person**{

    private String company;

    public Employee(String name, String company) {

        super(name);

        this.company = company;

    }

    public String getCompany(){ return company; }

    public void display(){

        System.out.printf("Employee %s works in %s \n", super.getName(), company);

    }

}

// класс клиента банка

**class Client extends Person{**

    private int sum; // Переменная для хранения суммы на счете

    private String bank;

    public Client(String name, String bank, int sum) {

        super(name);

        this.bank=bank;

        this.sum=sum;

    }

    public void display(){

        System.out.printf("Client %s has account in %s \n", super.getName(), bank);

    }

    public String getBank(){ return bank; }

    public int getSum(){ return sum; }

}

Восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии) или upcasting осуществляется автоматически.

Object tom = new Person("Tom");

Object sam = new Employee("Sam", "Oracle");

Object kate = new Client("Kate", "DeutscheBank", 2000);

Person bob = new Client("Bob", "DeutscheBank", 3000);

Person alice = new Employee("Alice", "Google");

Обратное не всегда верно. Например, объект Person не всегда является объектом Employee или Client. Поэтому нисходящее преобразование или downcasting от суперкласса к подклассу автоматически не выполняется. В этом случае нам надо использовать операцию преобразования типов.

Object sam = new Employee("Sam", "Oracle");

// нисходящее преобразование от Object к типу Employee

Employee emp = (Employee)sam;

emp.display();

System.out.println(emp.getCompany());

В данном случае переменная sam приводится к типу Employee. И затем через объект emp мы можем обратиться к функционалу объекта Employee.

1. **Что такое объект класса Class? Чем использование метода getClass() и последующего сравнения возвращенного значения с Type.class отличается от использования оператора instanceof?**

В исполняемой java-программе каждому классу соответствует объект типа *Class*, который содержит необходимую для описания класса информацию (поля, методы и т.д.). Класс **Class** не имеет открытого конструктора – объекты этого класса создаются автоматически Java-машиной по мере загрузки классов и вызовов метода [defineClass](http://java-online.ru/java-classloader.xhtml#defineClass) загрузчика классов.

Метод **getClass()**  возвращает класс объекта, содержащий сведения об объекте: public final Class<?> getClass().

Используя оператор instanceof, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа — имя типа, на совместимость с которым проверяется объект. Оператор instanceof опирается не на тип ссылки, а на свойства объекта, на который она ссылается. Но этот оператор возвращает истинное значение не только для того типа, от которого был порожден объект.

1. **Укажите правила переопределения методов equals(), hashCode() и toString().**

Java предлагает следующие правила для переопределения методов:

1. Рефлексивность: Объект должен равняться себе самому.
2. Симметричность: если a.equals(b) возвращает true, то b.equals(a) должен тоже вернуть true.
3. Транзитивность: если a.equals(b) возвращает true и b.equals(c) тоже возвращает true, то c.equals(a) тоже должен возвращать true.
4. Согласованность: повторный вызов метода equals() должен возвращать одно и тоже значение до тех пор, пока какое-либо значение свойств объекта не будет изменено. То есть, если два объекта равны в Java, то они будут равны пока их свойства остаются неизменными.
5. Сравнение null: объект должны быть проверен на null. Если объект равен null, то метод должен вернуть false, а не NullPointerException. Например, a.equals(null) должен вернуть false.

Метод hashCode() возвращает хэшкод объекта, вычисление которого управляется следующими соглашениями:

• все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;

• различные по содержанию объекты одного типа могут иметь различные хэш-коды;

• во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен.

Метод toString() следует переопределять таким образом, чтобы, кроме стандартной информации о пакете в котором находится класс, и самого имени класса, он возвращал значения полей объекта, вызвавшего этот метод (т. е. всю полезную информацию объекта), вместо хэшкода, как это делается в классе Object.

1. **Что такое абстрактные классы и методы? Зачем они нужны? Бывают ли случаи, когда абстрактные методы содержат тело? Можно ли в абстрактных классах определять конструкторы? Могут ли абстрактные классы содержать неабстрактные методы? Можно ли от абстрактных классов создавать объекты и почему?**

Абстрактные классы объявляются с ключевым словом abstract и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Объекты таких классов создать нельзя с помощью оператора new, но можно создать объекты подклассов, которые реализуют все эти методы. При этом допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

1. **Что такое интерфейсы? Как определить и реализовать интерфейс в java-программе? Укажите спецификаторы, которые приобретают методы и поля, определенные в интерфейсе. Можно ли описывать в интерфейсе конструкторы и создавать объекты? Можно ли создавать интерфейсные ссылки и если да, то на какие объекты они могут ссылаться?**

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Общее определение интерфейса имеет вид:

[public] interface Имя [extends Имя1, Имя2,…, ИмяN] { /\* реализация интерфейса \*/ }

Все объявленные в интерфейсе методы автоматически трактуются как public и abstract, а все поля — как public, static и final, даже если они так не объявлены.

Интерфейсы не имеют конструкторов, поэтому мы не можем использовать "условный" конструктор интерфейса (так как интерфейсы не обладают конструкторами) для создания объекта интерфейса.

Допустимо объявление ссылки на интерфейсный тип или использование ее в качестве параметра метода. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. При вызове метода через такую ссылку будет вызываться его реализованная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

1. **Для чего служит интерфейс Clonable? Как правильно переопределить метод clone() класса Object, для того, что объект мог создавать свои адекватные копии?**

Класс Object содержит protected метод clone(), осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод clone() как public для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода super.clone(), которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс Cloneable. Интерфейс Cloneable не содержит методов, относится к интерфейсам, а его реализация гарантирует, что метод clone() класса Object возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение CloneNotSupportedException.

1. **Для чего служат интерфейсы Comparable и Comparator? В каких случаях предпочтительнее использовать первый, а когда – второй? Как их реализовать и использовать?**

Данные интерфейсы используются для реализации сортировок.

Интерфейс Comparable содержит один единственный метод int compareTo (E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

Можно полагаться на встроенный механизм сравнения, приведенный выше, либо мы также можем определить и свою логику.

Если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить. На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>.

Интерфейс Comparator содержит ряд методов, ключевым из которых является метод compare (). Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс.

**Generic classes and Interfaces, Enums**

1. **Что такое перечисления в Java. Как объявить перечисление? Чем являются элементы перечислений? Кто и когда создает экземпляры перечислений?**

Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

enum Day{

    MONDAY,

    TUESDAY,

    WEDNESDAY,

    THURSDAY,

    FRIDAY,

    SATURDAY,

    SUNDAY;

}

Элементы перечислений являются константами. При этом объекты перечисления инициализируются прямым объявлением без помощи оператора new. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления.

1. **Можно ли самостоятельно создать экземпляр перечисления? А ссылку типа перечисления? Как сравнить, что в двух переменных содержится один и тот же элемент перечисления и почему именно так?**

Нельзя создавать экземпляры Enum вне границ Enum, поскольку у Enum нет public конструктора, и компилятор не позволит вам внести любой подобный конструктор. Так как компилятор генерирует большинство кода в ответ на декларацию Enum типа, он не допускает public конструкторов внутри Enum, что заставляет объявлять экземпляры Enum внутри себя.

Для сравнения enum можно использовать == и .equals(). Перечисления имеют жесткий контроль над экземплярами, который позволяет использовать == для сравнения экземпляров.

1. **Что такое анонимные классы?**

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков и т. д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного-единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе.

Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора new.

1. **Что такое параметризованные классы? Для чего они необходимы? Приведите пример параметризованного класса и пример создания объекта параметризованного класса? Объясните, ссылки какого типа могут ссылаться на объекты параметризованных классов? Можно ли создать объект, параметризовав его примитивным типом данных?**

Обобщения (generic)- это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра. Обобщения добавили в язык безопасность типов.

Ниже приведен пример параметризованного класса Message с конструкторами и методами, также инициализация и исследование поведения объектов при задании различных параметров.

**public class Message <T>** {

private T value;

public Message() {

}

public Message (T value) {

this.value = value;

}

public T getValue() {

return value;

}

public void setValue(T value) {

this.value = value;

}

public String toString() {

if (value == null) {

return null;

}

return value.getClass().getName() + " :" + value;

}

}

**public class Main** {

public static void main(String[ ] args) {

// параметризация типом Integer

Message<Integer> ob1 = new Message<Integer>();

ob1.setValue(1); // возможен только тип Integer для метода setValue

int v1 = ob1.getValue();

System.out.println(v1);

// параметризация типом String

Message<String> ob2 = new Message<String>("Java");

String v2 = ob2.getValue();

System.out.println(v2);

В результате выполнения этой программы будет выведено:

**1**

**Java**

**Exceptions and Errors**

1. **Что для программы является исключительной ситуацией? Какие способы обработки ошибок в программах вы знаете?**

Иногда в процессе выполнения программы могут возникать ошибки, при том некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Подобные ситуации называются исключениями.

Для работы с исключениями в Java существуют специальные блоки кода: try, catch и finally.

Второй способ чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор throw. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения.

1. **Что такое исключение для Java-программы? Что значит “программа выбросила исключение”? Опишите ситуации, когда исключения выбрасываются виртуальной машиной(автоматически), и когда необходимо их выбрасывать вручную?**

Исключение — некая исключительная, незапланированная ситуация, которая произошла при работе программы. Примеров исключений в Java может быть много. Каждое исключение представлено отдельным классом. Все классы исключений происходят от общего “предка” — родительского класса Throwable. Название класса-исключения обычно коротко отображает причину его возникновения:

* FileNotFoundException (файл не найден)
* ArithmeticException (исключение при выполнении математической операции)
* ArrayIndexOutOfBoundsException (указан номер ячейки массива за пределами его длины). Например, если попытаться вывести в консоль ячейку array[23] для массива array длиной 10.

1. **Приведите иерархию классов-исключений, делящую исключения на проверяемые и непроверяемые. В чем особенности проверяемых и непроверяемых исключений?**

В вершине иерархии исключений стоит класс **Throwable**, который наследуется от **Object**. Каждый из типов исключений является подклассом Throwable. Два непосредственных наследника класса Throwable делят иерархию подклассов исключений на две различные ветви.

Класс **Ехception** используется для описания исключительных ситуации, которые должны перехватываться программным кодом пользователя. Класс **Error** предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

Все исключения делятся на 2 вида — проверяемые (checked) и непроверяемые (unchecked).

Компилятор Java знает о самых распространенных исключениях, и знает, в каких ситуациях они могут возникнуть. Например, он знает, что если программист в коде считывает данные из файла, может легко возникнуть ситуация, что файл не существует. И таких ситуаций, которые он может заранее предположить, очень много. Поэтому компилятор заранее проверяет наш код на наличие потенциальных исключений. Если он их найдет, то не скомпилирует код, пока мы не обработаем их или не пробросим наверх.

Второй вид исключений — “непроверяемые”. Они происходят от класса **RuntimeException**. Чем же они отличаются от проверяемых? Казалось бы, тоже есть куча разных классов, которые происходят от **RuntimeException** и описывают конкретные runtime-исключения. Разница в том, что этих ошибок компилятор не ожидает.

1. **Объясните работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному блоку try? Можно ли вкладывать блоки try друг в друга, можно ли вложить блок try в catch или finally? Как происходит обработка исключений, выброшенных внутренним блоком try, если среди его блоков catch нет подходящего? Что называют стеком операторов try? Как работает блок try с ресурсами**.

Для защиты программного кода от исключений необходимо использовать связанные блоки с ключевыми словами **try catch**.

Первое идет ключевое слово try, далее блок кода (защищенный блок кода) в фигурных скобках (фигурные скобки обязательны). После защищенного блока кода располагается ключевое слово catch (обязательно, если не присутствует блок finally), после которого в скобках указывается тип исключения, которое необходимо обработать, в следующих фигурных скобках, располагается код, который так или иначе обрабатывает сгенерированное исключение. Если выполнение кода в блоке try происходит без каких-либо проблем, то блок catch игнорируется, если же в блоке try возникает исключение, то выполнение кода прекращается и передается блоку catch c соответствующем исключением.

В отдельных случаях блок программного кода может вызвать исключения различных типов. Для того, чтобы локализовать обработку подобных ситуаций, можно использовать несколько **catch**-разделов для одного *try-блока*. Блоки наиболее специализированных классов исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса.

Операторы *try* можно вкладывать друг в друга. Если у оператора *try* низкого уровня нет раздела *catch*, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы catch внешнего оператора try.

Как только в какой-то строчке кода в блоке try возникнет исключение, код после нее уже не будет выполнен. Выполнение программы сразу “перепрыгнет” в блок catch. Если код в блоке catch не отработал, потому что мы, например, “настроили” этот блок на перехват ArithmeticException, а код в блоке try выбросил другой тип — FileNotFoundException. Для FileNotFoundException мы не написали сценарий, поэтому программа выведет в консоль ту информацию, которая выводится по умолчанию для FileNotFoundException.

«try c ресурсами» это оператор try, в котором объявляются один или несколько ресурсов. Ресурс — это объект, который должен быть закрыт после того, как программа закончит с ним работу. «try c ресурсами» берет всю работу по закрытию ресурсов на себя. Главное отличие от привычного блока try в круглых скобках, в которых создаются ресурсы, которые впоследствии нужно закрыть, ресурсы будут закрываться снизу-вверх автоматически после завершения работы блока try.

1. **Укажите правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченной исключение быть сгенерировано снова, и, если да, то как и кто в этом случае будет обрабатывать повторно сгенерированное исключение? Может ли блок catch выбрасывать иные исключения, и если да, то опишите ситуацию, когда это может быть необходимо.**

Для того, чтобы обработать несколько разных типов исключений необходимо для каждого из них создать свой блок catch. При подобном подходе к обработке исключений следует учитывать одно очень важное правило, которое касается очередности блоков catch — Java обрабатывает эти блоки в порядке очередности и если возникает ситуация что один из блоков catch выполнен не будет ни при каких обстоятельствах возникнет ошибка компиляции с сообщением о недостижимости кода (unreachable code). Подобное происходит, когда исключение суперкласса идет перед исключением подкласса.

При необходимости можно создавать собственные исключения. Достаточно создать собственный класс. Его название должно заканчиваться на “Exception”. Компилятору это не нужно, но читающим твой код программистам сразу будет понятно, что это класс-исключение. Кроме того, нужно указать что класс происходит от класса Exception. Это уже нужно для компилятора и корректной работы. Исключение выбрасывается с помощью слова throw.

1. **Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не будет вызван? Может ли блок finally выбрасывать исключений? Может ли блок finally выполнится дважды?**

Если в методе генерируется исключение оставшиеся в нем операторы не выполняются, тогда возникает следующая проблема – если в методе задействованы локальные ресурсы, то они не высвобождаются. В этом случае на помощь придет блок finally. Принцип его работы очень прост – он выполняется всегда при любых обстоятельствах. Если в блоке try не было сгенерировано никаких исключений, то следом выполняется блок finally, если в блоке try было сгенерировано исключение, то выполняется блок catch, а затем finally. Есть только одно исключение – если перед блоком finally был сделан вызов [System.exit(0)](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#exit(int)), то программа завершает свою работу без выполнения блока finally. Блок finally не может выполниться дважды.