5. Basics of OOP

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

1. Дайте развернутое объяснение трем концепциям ООП.

Инкапсуляция – свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Для Java корректно будет говорить, что инкапсуляция это «сокрытие реализации».

Наследование – свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

Полиморфизм – свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Абстракция – означает выделение значимой информации и исключение из рассмотрения незначимой. С точки зрения программирования это правильное разделение программы на объекты. Абстракция позволяет отобрать главные характеристики и опустить второстепенные.

2. 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 значению?

Инициализация полей, указанных в классе происходит особым образом в зависимости от того, является ли поле статическим или нет. Если у поля стоит ключевое слово static, то данное поле относится к самому классу, а не слово static не указано, то данное поле относится к экземпляру класса.

```
public class HelloWorld{
    private int number;
    private static int count;

    public static void main(String []args){
        HelloWorld object = new HelloWorld();
        System.out.println(object.number);
    }
}
```

В данном примере, инициализация полей происходит в разное время. Поле number будет инициализировано после того, как будет создан объект object класса HelloWorld. А вот поле count будет инициализировано тогда, когда класс будет загружен виртуальной Java машиной. Статические переменные инициализируются тогда, когда о классе становится известно при выполнении.

Переменные, которые являются полями, если для них не указано значение, то они инициализируются значением по умолчанию. Для числовых значением это 0 или 0.0 для чисел с плавающей точкой. Для boolean это false. А для всех переменных объектных типов значение будет null.

Полям класса могут быть присвоены начальные значения следующим способом: в месте объявления поля; в инициализационном блоке; в конструкторе класса.

3. Приведите правила, которым должен следовать компонент java-bean.

1) Способность к инстанцированию нового экземпляра (бин – не интерфейс, не абстрактный класс).

2) Наличие конструктора по умолчанию (конструктора без параметров).

3) Возможность сериализации.

4) Следование соглашениям об именах и способах проектирования.

5) Использование модели делегирования обработки событий.

4. ***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 когда несколько методов одного класса могут иметь одно и то же имя, отличаясь лишь набором параметров. Для того чтобы перегрузить метод, достаточно объявить его новый вариант, отличающийся от уже существующих, а все остальное сделает компилятор. Нужно лишь соблюсти одно условие: тип и/или число параметров в каждом из перегружаемых методов должны быть разными.

Использование перегрузки делает ваш код чище и проще для чтения, а также помогает избежать ошибок в программе. Перегрузка метода позволяет использовать одно и то же имя и изменять только то, что необходимо — параметры. Также очень легко найти перегруженные методы, поскольку они сгруппированы в коде. Главная ценность перегрузки заключается в том, что она обеспечивает доступ к группе родственных методов по общему имени.

Изменение имени переменной не является перегрузкой. Также мы не можем перегрузить метод, изменяя возвращаемое значение в сигнатуре метода.

Перегружать методы в базовом и производном классе можно.

private метод базового класса перегрузить public методов производного не допускается. Можно создать такой же метод public и его вариации перегрузки в производном классе.

Конструктор – это схожая с методом структура, назначение которой состоит в создании экземпляра класса. Их тоже можно перегружать (создавать несколько конструкторов с разным набором аргументов) и для них

тоже принципиально важен порядок передачи аргументов. При перегрузке конструкторов менять атрибуты доступа у конструкторов можно.

5. ***Объясните, что такое раннее и позднее связывание? Перегрузка – это раннее или позднее связывание? Объясните привила, 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.***

Существует два типа связывания методов в языке Java: ранее связывание (его ещё называют статическим) и позднее (соответственно, динамическое) связывание. Вызов метода в Java означает, что этот метод привязывается к конкретному коду или в момент компиляции, или во время выполнения, при запуске программы и создании объектов.

Статическое связывание носит более статический характер, так как происходит во время компиляции, то есть код «знает», какой метод вызывать после компиляции исходного кода на Java в файлы классов. А поскольку это относится к ранней стадии жизненного цикла программы, то называется также ранним связыванием (early binding).

Динамическое связывание 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. А поскольку это происходит на поздней стадии жизненного цикла программы, то называется в языке Java поздним связыванием (late binding).

Фундаментальное различие между статическим и динамическим связыванием в Java состоит в том, что первое происходит рано, во время компиляции на основе типа ссылочной переменной, а второе – позднее, во время выполнения, с использованием конкретных объектов.

ключевые различия между ранним и поздним связыванием в языке Java:

1) Статическое связывание происходит во время компиляции, а динамическое – во время выполнения.

2) Поскольку статическое связывание происходит на ранней стадии жизненного цикла программы, его называют ранним связыванием. Аналогично, динамическое связывание называют также поздним связыванием, поскольку оно происходит позже, во время работы программы.

3) Статическое связывание используется в языке Java для разрешения перегруженных методов, в то время как динамическое связывание используется в языке Java для разрешения переопределенных методов.

4) Аналогично, приватные, статические и терминальные методы разрешаются при помощи статического связывания, поскольку их нельзя переопределять, а все виртуальные методы разрешаются при помощи динамического связывания.

5) В случае статического связывания используются не конкретные объекты, а информация о типе, то есть для обнаружения нужного метода используется тип ссылочной переменной. С другой стороны, при динамическом связывании для нахождения нужного метода в Java используется конкретный объект.

Всегда нужно помнить, что **приватные**, **статические** и **final-методы** связываются при помощи **статического связывания**, а **виртуальные – динамического**. Аналогично, лучший пример статического связывания – перегрузка методов, а переопределение – динамического.

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

Ключевое слово this требуется для того, чтобы метод мог сослаться на вызвавший его объект.

Как правило, применять this нужно в двух случаях:

1) Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;

2) Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора.

Применение оператора this можно обобщить до следующего:

1) Обращение к не статичным полям класса.

2) Получение текущего объекта.

3) Для решения конфликтов локальных и глобальных.

4) Вызов другого конструктора из конструктора.

Ключевое слово this в Java используется только в составе методов либо конструкторов экземпляра класса. Но неявно ключевое слово this передается во все методы (поэтому this часто называют неявным параметром) и может быть использовано для обращения к объекту, вызвавшему метод.

Ключевое слово this не используется в статических методах, так как нельзя ссылаться из статического контекста.

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

Ключевое слово final может применяться к классам, методам, переменным (в том числе аргументам методов).

Для переменных примитивного типа final означает, что однажды присвоенное значение не может быть изменено.

Для ссылочных переменных final означает, что после присвоения объекта, нельзя изменить ссылку на данный объект. Это важно! Ссылку изменить нельзя, но состояние объекта изменять можно.

Существуют два способа установить начальное значение неизменных данных:

1) с помощью непосредственного присваивания (=) значения при объявлении final-поля данных класса;

2) с помощью конструктора класса. Этот способ подходит только для нестатических неизменных данных. Сначала в классе объявляется поле

класса со спецификатором final. Это поле называется пустой константой. Затем эта пустая константа обязательно инициализируется значением в конструкторе класса. Количество перегруженных конструкторов, которые инициализируют пустые константы, может быть любым.

8. 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?

Модификатор static в Java напрямую связан с классом, если поле статично, значит оно принадлежит классу, если метод статичный, аналогично – он принадлежит классу. Исходя из этого, можно обращаться к статическому методу или полю используя имя класса.

Нельзя получить доступ к НЕ статическим членам класса, внутри статического контекста, как вариант, метода или блока. Результатом компиляции будет ошибка.

Важным моментом является то, что НЕ возможно переопределять (Override) статические методы. Если объявить такой же метод в классе-наследнике (subclass), т.е. метод с таким же именем и сигнатурой, будет лишь «спрятан» метод суперкласса (superclass) вместо переопределения. Это явление известно как сокрытие методов (hiding methods). Это означает, что при обращении к статическому методу, который объявлен как в родительском, так и в дочернем классе, во время компиляции всегда будет вызван метод исходя из типа переменной. В отличие от переопределения, такие методы не будут выполнены во время работы программы. Статические методы перегружаются также, как и обычные.

9. 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?

Блоки инициализации используются для инициализации переменных внутри класса. Есть два типа блоков инициализации - статический и нестатический. Синтаксис очень простой - просто пишем выражение внутри блока из двух скобок: { //…} – нестатический; static{ //…} – статический. Статический блок инициализации используется для статических переменных, нестатический - для всех остальных. Блоки инициализации делают код читабельнее, и позволяют вызывать любые методы.

Блоки инициализации могут быть размещены в любом порядке и любом количестве, однако вызываются сначала статические блоки, потом нестатические. Порядок вызова идет сверху вниз.

10. 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?

В языке Java существуют методы, которые могут принимать переменное количество аргументов. Для указания аргументов переменной длины служат три точки (. . .). Например: static void test(int... array).

Наряду с параметром переменной длины у метода могут быть и "обычные" параметры. Но параметр переменной длины должен быть последним среди всех параметров, объявляемых в методе. Например: static void test(double d, int... array).

По факту, переменное число аргументов ничто иное, как просто украшение синтаксиса – это всего-лишь массив передаваемых данных.

Выбор подходящего метода осуществляется исходя из типа передаваемых параметров.

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

Хотя можно создать обычный класс, который не является наследником, но фактически все классы наследуют от класса Object. Все остальные классы, даже те, которые добавляем в проект, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object.

Класс Object содержит следующие методы:

toString – метод toString служит для получения представления данного объекта в виде строки;

hashCode – метод hashCode позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты;

getClass – метод getClass возвращает объект типа Class, который представляет информацию о классе текущего объекта на этапе выполнения программы;

equals – метод equals сравнивает два объекта на равенство;

clone – метод clone возвращает клон текущего объекта. Клон - это новый объект, являющийся копией текущего;

notify – метод notify возобновляет единичный поток, который ожидает на объектном мониторе;

notifyAll – метод notifyAll возобновляет все потоки, которые ожидают на объектном мониторе;

wait – метод wait осуществляет остановку текущего потока пока другой поток не вызовет notify() или notifyAll метод для этого объекта.

Оба метода, **hashcode** и **equals**, должны быть переопределены, если необходимо придать понятию равенства объектов иной смысл, отличный от того, который предлагается классом Object. По умолчанию считается, что любые два различных объекта "не равны", Т.е. метод equals возвращает значение false, и их хеш-коды, как правило, различны.

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

Хеш-код это число, у которого есть свой предел, который для java ограничен примитивным целочисленным типом int. В терминах Java, хеш-

код — это целочисленный результат работы метода, которому в качестве входного параметра передан объект.

Метод **hashcode** реализован таким образом, что для одного и того-же входного объекта, хеш-код всегда будет одинаковым. Следует понимать, что множество возможных хеш-кодов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: «Множество объектов мощнее множества хеш-кодов». Из-за этого ограничения, вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.

На основании вышеизложенного можно сделать вывод, что:

- Если хеш-коды разные, то и входные объекты гарантированно разные.
- Если хеш-коды равны, то входные объекты не всегда равны.

Ситуация, когда у разных объектов одинаковые хеш-коды называется коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

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

Идея наследования в Java заключается в том, что вы можете создавать новые классы, основанные на существующих классах. Когда вы наследуете от существующего класса, вы можете повторно использовать методы и поля родительского класса. Более того, вы также можете добавлять новые методы и поля в ваш текущий класс.

private поля и методы доступны методам только внутри класса поэтому, хотя они и наследуются подклассами, но подклассы не могут получить доступ к **private** полям и методам супер класса.

14. 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()?

При создании объекта в первую очередь вызывается конструктор его базового класса, а только потом - конструктор самого класса, объект которого мы создаем, именно поэтому оператор super() всегда должен стоять в конструкторе первым. Иначе логика работы конструкторов будет нарушена и программа выдаст ошибку.

Оба this() и super() являются вызовами конструктора. Конструкторский вызов всегда должен быть первым выражением. Таким образом, мы не можем иметь два оператора в качестве первого оператора, поэтому мы можем вызывать super() или мы можем вызвать this() из конструктора, но не оба.

15. Объясните, как вы понимаете утверждение: «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?

Обратное не верно, потому, что суперклассу неизвестно, что именно подкласс добавляет в него.

16. 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?

Переопределение метода (англ. *Method overriding*) в объектно-ориентированном программировании - одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

Сигнатура (тип возврата, тип параметров, количество параметров и порядок параметров) должна быть такой же, какой была определена в родительском классе. Переопределение метода выполняется для достижения полиморфизма во время выполнения программы. В рамках одного класса методы не переопределяются.

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

ParentClass parent = new ChildClass();

parent.metod(); //вызовется статический метод класса ParentClass, если метод не статический, то вызовется динамический метод класса ChildClass.

А вообще, статические методы переопределять нельзя и наоборот тоже.

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

Обозначая метод класса модификатором final, мы имеем ввиду, что ни один производный класс не в состоянии переопределить этот метод, изменив его внутреннюю реализацию. Другими словами, речь идет о финальной версии метода. Класс, помеченный как final, не поддается наследованию и все его методы косвенным образом приобретают свойство 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. Если признаком final обозначен метод, можно доверять его внутренней реализации, не опасаясь "подделки". Уместно применить final, например, в объявлении метода, предусматривающего проверку пароля, вводимого пользователем, чтобы гарантировать точное исполнение того, что методом предусмотрено изначально.

Во многих случаях для достижения достаточного уровня безопасности кода вовсе нет необходимости обозначать весь класс как final – вполне возможно сохранить способность класса к расширению, пометив модификатором final только его "критические" структурные элементы. В этом случае останутся в неприкосновенности основные функции класса и одновременно разрешится его наследование с добавлением новых членов, но без переопределения "старых". Разумеется, поля, к которым обращается код методов final, должны быть в свою очередь обозначены как final либо private,

поскольку в противном случае любой производный класс получит возможность изменять их содержимое, воздействуя на поведение соответствующих методов.

19. Укажите правила приведения типов при наследовании. Напишите примеры явного и неявного преобразования ссылочных типов. Объясните, какие ошибки могут возникнуть при явном преобразовании ссылочных типов?

1) Приведение типов можно выполнять только в иерархии наследования.

2) Для того, чтобы проверить корректность приведения суперкласса к подклассу, следует выполнить операцию instanceof

Но в целом при наследовании лучше свести к минимуму приведение типов и выполнении операции instanceof.

В процессе своей работы компилятор проверяет, не обещаете ли вы слишком много, сохраняя значение в переменной. Так, если вы присваиваете переменной суперкласса ссылку на объект подкласса, то обещаете *меньше* положенного, и компилятор разрешает сделать это. А если вы присваиваете объект суперкласса переменной подкласса, то обещаете больше положенного, и поэтому вы должны подтвердить свои обещания, указа в скобках имя класса для приведения типов.

Расширяющие преобразования ссылок:

Расширяющие преобразования ссылок это преобразования производных ссылочных типов в типы их предков, которые не требуют никаких действий на этапе исполнения и никогда не генерируют ошибок. Такими преобразованиями в Java являются:

- преобразование любого класса или интерфейса в его предка (в частности, в тип Object);
- преобразование класса в интерфейс, который он реализует;
- преобразование любого массива в тип Object или тип Cloneable ;
- преобразование массива типа S в массив типа T, если S и T ссылочные типы, и преобразование S в T является расширяющим;
- преобразование нулевого типа в любой ссылочной тип.

Сужающие преобразования ссылок:

Сужающие преобразования ссылок это преобразования производных ссылочных типов в типы их потомков. Эти преобразования требуют проверки своей легитимности на этапе исполнения и могут генерировать исключение **ClassCastException**. Такими преобразованиями в Java являются:

- преобразование любого класса в его потомка (в частности, преобразование типа Object в любой другой класс);
- преобразование класса в интерфейс, когда класс не является финальным и не реализует данный интерфейс (в частности, преобразование типа Object в любой интерфейс);
- преобразование типа Object в любой массив;
- преобразование любого интерфейса в класс, который не является финальным;

- преобразование любого интерфейса в класс, который является финальным и реализует данный интерфейс;
- преобразование интерфейса J в интерфейс K, когда J не является потомком K, и не существует метода, декларированного и в J, и в K с одинаковой сигнатурой, но разными типами результата;
- преобразование массива типа S в массив типа T, если S и T ссылочные типы, и преобразование S в T является сужающим.

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

Каждый раз, когда Java-машина загружает в память новый класс, она создает объект типа Class, посредством которого можно получить некоторую информацию о загруженном классе. К каждому классу и объекту привязан такой «**объект класса**».

Метод getClass() возвращает класс объекта, содержащий сведения об объекте: *public final Class<?> getClass()*. Метод является конечным и переопределению не подлежит. Метод прост и эффективен, ровно до тех пор, пока не применен механизм наследования. При наследовании этот метод становится менее полезным.

```
class A{}
class B extends A{};
public class Example{
    public static void main(String... args){
        A a = new A();
        A a1 = new A();
        B b = new B();
        A ab = new B();
        System.out.println(a.getClass() == a1.getClass()); // true
        System.out.println(a.getClass() == b.getClass()); // false
        System.out.println(a.getClass() == ab.getClass()); // false
    }
}
```

Reflection – это способность класса получить информацию о самом себе. В Java есть специальные классы: **Field** – поле, **Method** – метод, по аналогии с **Class** для классов. Т.к. объект типа **Class** дает возможность получить информацию о классе, то объект типа Field–получить информацию о «поле класса», а Method–о «методе класса». И вот что с ними можно делать:

```
Class[] interfaces = List.class.getInterfaces();
Class parent = String.class.getSuperclass();
Method[] methods = List.class.getMethods();
```

Оператор instanceof служит для проверки к какому классу принадлежит объект. a instanceof B возвращает истину, если в переменной a содержится ссылка на экземпляр класса B, подкласс B (напрямую или косвенно, иначе говоря, состоит в иерархии наследования классов) или реализует интерфейс B (так же напрямую или косвенно).

Подводя итог, можно сделать вывод, что instanceof и getClass() == Type.class выполняют разные вещи, а именно:

instanceof проверяет, является ли ссылка объекта в левой части (LHS) экземпляром типа в правой части (RHS) или некоторым подтипом.

getClass() == ... проверяет, идентичны ли типы.

Поэтому рекомендуется использовать альтернативу, которая дает ответ, который нужен.

21. Укажите правила переопределения методов equls(), hashCode() и toString().

Если методы equals() и hashCode() не переопределены, вместо них будут вызваны методы класса Object. В этом случае методы не выполняют реальной цели equals() и hashCode(), которая состоит в том, чтобы проверить, имеют ли объекты одинаковые состояния.

Реализация методов в базовых классах Java:

```
public boolean equals(Object obj) { return (this == obj); }
@HotSpotIntrinsicCandidate
public native int hashCode();
```

Метод equals() используется для сравнения объектов. Чтобы определить одинаковые объекты или нет, equals() сравнивает значения полей объектов:

```
static class Simpson {
private String name; private int age; private int weight;
@Override
public boolean equals(Object o) {
 // 1 if (this == o) { return true; }
 // 2 if (o == null || getClass() != o.getClass()) { return false; }
 // 3 Simpson simpson = (Simpson) o;
return age == simpson.age
      && weight == simpson.weight
      && name.equals(simpson.name); }
```

Для оптимизации производительности при сравнении объектов используется метод hashCode(). Метод hashCode() возвращает уникальный идентификатор для каждого объекта, что упрощает сравнение состояний объектов.

Если хэш-код объекта не совпадает с хэш-кодом другого объекта, то можно не выполнять метод equals(): вы просто знаете, что два объекта не совпадают. С другой стороны, если хэш-код одинаковый то, необходимо выполнить метод equals(), чтобы определить, совпадают ли значения полей.

```
public class HashcodeConcept {
 public static void main(String... args) {
 Simpson homer = new Simpson(1, "Homer");
 Simpson bart = new Simpson(2, "Homer");
 boolean isHashcodeEquals = homer.hashCode() == bart.hashCode();
if (isHashcodeEquals) { System.out.println("Следует сравнить методом
equals.");
} else {
```

```
        System.out.println("Не следует сравнивать методом equals, т.к. " +
"идентификатор отличается, что означает, что объекты точно не равны.");
        }
        }

        static class Simpson {
        int id; String name;
        public Simpson(int id, String name) { this.id = id; this.name = name; }

       @Override
        public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Simpson simpson = (Simpson) o;
       return id == simpson.id && name.equals(simpson.name);
       }
       @Override
        public int hashCode() { return id; }
       @Override
        public String toString() { return "Id: " + this.id + "name:" + this.name;
}

       } }
```

Метод hashCode(), который всегда возвращает одно и то же значение, допустим, но не эффективен. В этом случае сравнение всегда будет возвращать true, поэтому метод equals() будет выполняться всегда. В этом случае нет никакого улучшения производительности.

Рекомендации: Выполняйте метод equals() только для объектов с одинаковым хэш-кодом. *Не выполняйте* equals(), если хэш-код отличается.

Правила сравнения объектов: Когда сравнение hashCode() возвращает false, метод equals() *также должен возвращать* false. Если хэш-код отличается, то объекты определенно не равны.

Благодаря переопределению метода toString() можно отображать информацию в желаемом и удобно читаемом формате.

22. 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?

Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

В абстрактных классах можно определять конструкторы, но мы не можем использовать конструктор абстрактного класса для создания его объекта.

Кроме обычных методов абстрактный класс может содержать абстрактные методы. Такие методы определяются с помощью ключевого слова abstract и не имеют никакого функционала. Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Примером абстрактного класса является системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами.

23. 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?

Механизм наследования очень удобен, но он имеет свои ограничения. В частности мы можем наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface**.

Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**.

Если класс применяет интерфейс, то он должен реализовать все методы интерфейса. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.

В тоже время мы не можем напрямую создавать объекты интерфейсов.

При создании объектов класса в качестве типа объектной переменной может указываться имя реализованного в классе интерфейса. Другими словами, если класс реализует интерфейс, то ссылку на объект этого класса

можно присвоить интерфейсной переменной — переменной, в качестве типа которой указано имя соответствующего интерфейса.

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

У объекта java.lang.Object есть метод protected Object clone() throws CloneNotSupportedException. Этот метод позволяет создавать копии объектов. Он бросает исключение CloneNotSupportedException для всех объектов, которые не реализуют интерфейс java.lang.Cloneable, а для классов, реализующих этот интерфейс, возвращает копию объекта, которая создана копированием всех полей исходного объекта. Содержимое полей не копируется. Если поле является ссылочным, то оно будет указывать на тот же самый объект, что и исходное поле.

Во многих случаях простого копирования недостаточно, так как некоторые поля могут содержать сложные структуры, для которых нужно создать копию, чтобы объекты были действительно независимы. А они должны быть независимы, так как это копирование. Тогда нужно будет переопределить метод копирования своим методом, вызвать super.clone() и создать копии структур для тех полей, которые содержат эти сложные изменяемые структуры. Массивы уже реализуют интерфейс java.lang.Cloneable.

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

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.

```
lass Person implements Comparable<Person>{

    private String name;
    Person(String name){

        this.name = name;
    }
    String getName(){return name;}

    public int compareTo(Person p){

        return name.compareTo(p.getName());
    }
}
```

Интерфейс **Comparator**. Может возникнуть проблема, что 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.

```
class PersonComparator implements Comparator<Person>{

    public int compare(Person a, Person b){

        return a.getName().compareTo(b.getName());
    }
}
```

Generic classes and Interfaces, Enums

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

Кроме отдельных примитивных типов данных и классов в Java есть такой тип как enum или перечисление. 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.

Все перечисления неявно наследуются от класса java.lang.Enum. Поскольку в Java нет множественного наследования, то перечисление не может наследоваться от какого-либо другого класса дополнительно, но может реализовывать сколько угодно интерфейсов. Следует отметить, что конструктор по умолчанию приватный, то есть имеет модификатор private. Любой другой модификатор будет считаться ошибкой. Поэтому создать константы перечисления с помощью конструктора мы можем только внутри перечисления.

Константы перечисления это экземпляры класса перечисления, которые инициализируются самостоятельно.

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

Попытка явного создания экземпляра типа перечисления является ошибкой времени компиляции. Метод final clone в перечислении гарантирует, что константы перечисления никогда не могут быть клонированы, а специальная обработка механизмом сериализации гарантирует, что дубликаты экземпляров никогда не создаются в результате

десериализации. Рефлексивное создание экземпляров типов перечислений запрещено. Вместе эти четыре вещи гарантируют, что никакие экземпляры типа перечисления не существуют за пределами тех, которые определены константами перечисления. Вся цель этого-обеспечить безопасное использование == сравнить Enum экземпляров.

Элементы enum - это статически доступные экземпляры enum-класса. Их статическая доступность позволяет нам выполнять сравнение с помощью оператора сравнения ссылок ==.

3. Что такое анонимные классы?

Основная особенность - анонимный класс не имеет имени. Анонимный класс является подклассом существующего класса или реализации интерфейса.

Поскольку анонимный класс не имеет имени, он не может иметь явный конструктор. Также к анонимному классу невозможно обратиться извне объявляющего его выражения, за исключением неявного обращения посредством объектной ссылки на суперкласс или интерфейс. Анонимные классы никогда не могут быть статическими, либо абстрактными, и всегда являются конечными классами. Кроме того, каждое объявление анонимного класса уникально.

4. 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?

Дженерики (обобщения) — это особые средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

```
class Gen<T> {
 T ob;
T Gen(T o)
{ ob = o;
}
T getob() { return ob; }
// Создаём объект
Gen<Integer> iOb = new Gen<Integer>(77);
// Показать тип данных, используемый iOb
iOb.showType();
```

Объект - это суперкласс всех объектов и может представлять любой пользовательский объект. Поскольку все примитивы не наследуются от "Object", поэтому мы не можем использовать его как общий тип. Следовательно, создать объект, параметризовав его примитивным типом данных нельзя.

Exceptions and Errors

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

Возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**. В программе исключения могут возникать в результате неправильных действий пользователя, отсутствии необходимого ресурса на диске, или потери соединения с сервером по сети. Причинами исключений при выполнении программы также могут быть ошибки программирования или неправильное использование API. В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

Создание блоков кода, для которых мы предусматриваем обработку исключений в Java, производится в программе с помощью конструкций try{}catch, try{}catch{}finally, try{}finally{}.

2. Что такое исключение для Java-программы? Что значит «программа выбросила исключение»? Опишите ситуации, когда исключение выбрасываются виртуальной машиной (автоматически), и когда необходимо их выбрасывать вручную?

Исключение (exception)— это событие, которое возникает во время выполнения программы и прерывает нормальный поток выполнения инструкций.

Когда возникает какая-нибудь ошибка внутри метода, метод создаёт специальный объект, называемый **объектом-исключением** или просто **исключением** (exception object), который передаётся системе выполнения. Этот объект содержит информацию об ошибке, включая тип ошибки и состояние программы, в котором произошла ошибка. Создание объекта-исключения и передача его системе выполнения называется **броском исключения** (throwing an exception).

Иерархия исключений в Java представлена следующим образом: родительский класс для всех Throwable. От него унаследовано 2 класса: Exception и Error. От класса Exception унаследован еще RuntimeException.

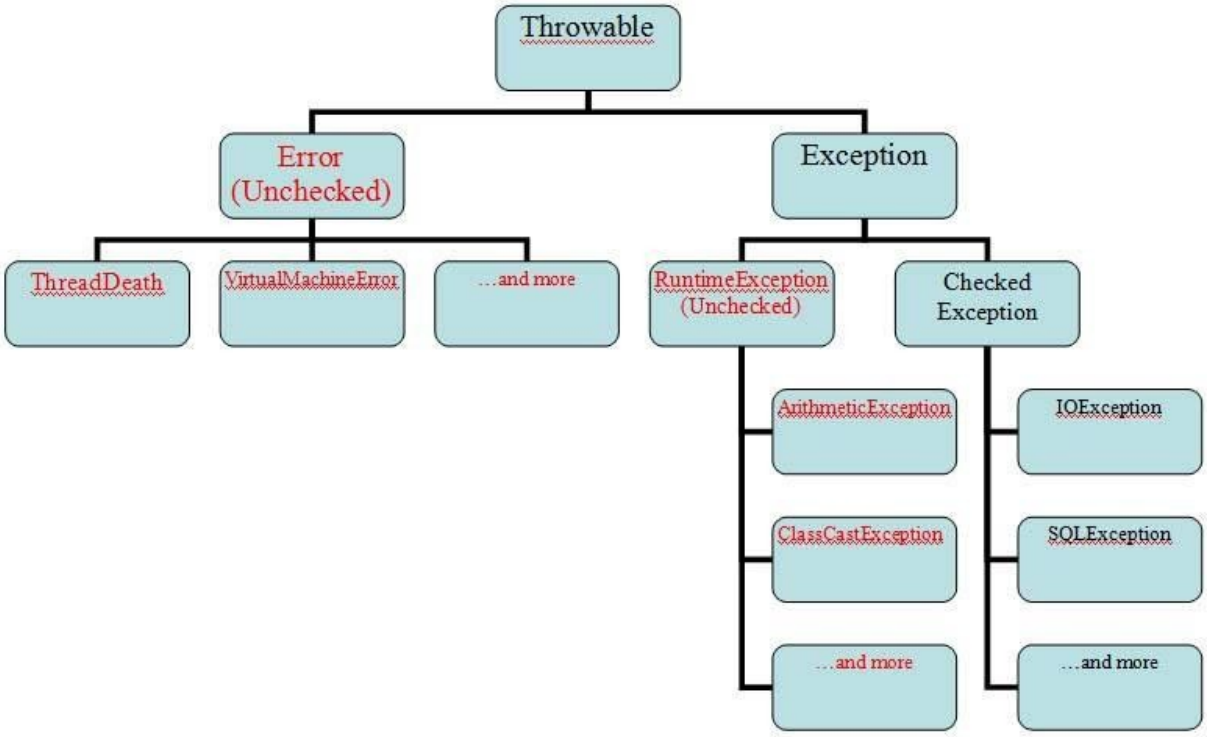
Error – критические ошибки, который могут возникнуть в системе (например, StackOverflowError). Как правило обрабатывает их система. Если они возникают, то приложение закрывается, так как при данной ситуации работа не может быть продолжена.

RuntimeException – это непроверенные исключения. Они возникают во время выполнения приложения. К таким исключениям относится, например, NullPointerException. Они не требуют обязательного заключения в блок try-catch. Когда RuntimeException возникает, это свидетельствует о ошибке, допущенной программистом (неинициализированный объект, выход за пределы массива и т.д.). Поэтому данное исключение не нужно обрабатывать, а нужно исправлять ошибку в коде, чтобы исключение вновь не возникало.

```
public int calculateSquare(Rectange rect) {
 if (rect == null)
 throw new NullPointerException(“Rectangle can't be null”);
```

В данном примере метод принимает объект класса Rectangle. В описании метода содержится строка @throws, которая описывает исключение, которое может быть выброшено и при каких условиях. Однако, сигнатура метода не содержит конструкции throws. Это значит, что при вызове метода его не нужно оборачивать блоком try-catch. А программист должен не допустить передачи в метод неинициализированного объекта.

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



Исключительные ситуации, возникающие в программе, можно разделить на две группы:

1) Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно;

2) Восстановление возможно.

К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции. К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным.

Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

4. Объясните работу оператора try-catch-finally. Когда данный оператор следует использовать? Сколько блоков catch может соответствовать одному 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.

- **try** – определяет блок кода, в котором может произойти исключение;
- **catch** – определяет блок кода, в котором происходит обработка исключения;
- **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

При возбуждении исключения в блоке **try** обработчик исключения ищется в следующем за ним блоке **catch**. Если в catch есть обработчик данного типа исключения – управление переходит к нему. Если нет, то **JVM** ищет обработчик этого типа исключения в цепочке вызовов методов до тех пор, пока не будет найден подходящий **catch**. После выполнения блока **catch** управление передается в необязательный блок **finally**.

Операторы **try** можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора try низкого уровня нет раздела catch, соответствующего возбужденному исключению, то в поисках подходящего обработчика будут проверены разделы catch внешнего оператора try.

Конструкцию try-with-resources ввели в Java 7. Она дает возможность объявлять один или несколько ресурсов в блоке try, которые будут закрыты автоматически без использования finally блока.

В качестве ресурса можно использовать любой объект, класс которого реализует интерфейс java.lang.AutoCloseable или java.io.Closable.

Если try блок также выбрасывает исключение, оно побеждает, а исключение из close() метода подавляется.

5. Укажите правило расположения блоков catch в зависимости от типов перехватываемых исключений. Может ли перехваченное исключение быт 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.

Порядок расположения блоков **catch** определяет очередность перехвата исключений. Нужно учитывать, что каждый такой блок перехватит все

исключения указанного класса или любого его подкласса. Если не учесть это, то можно получить недостижимый блок catch.

Исключение можно генерировать в блоке catch, создавая тем самым цепочку исключений. Обычно разработчики поступают так в том случае, если им надо изменить тип исключения. Если вы создаете подсистему, используемую другими программистами, имеет смысл генерировать такие исключения, которые дали бы возможность сразу определить, что ошибка возникла именно в ней. Необходимо помнить, что только исключения, сгенерированные в блоке try, могут быть перехвачены блоком catch. Это означает, что исключение, сгенерированное в блоке catch, не будет перехвачено этим же блоком catch, в котором оно находится. Вместо этого **стек начнёт раскручиваться** и исключение будет передано caller-у, который находится на уровне выше в стеке вызовов.

6. Когда происходит вызов блока finally? Существуют ли ситуации, когда блок finally не буде вызван? Может ли блок finally выбрасывать исключения? Может ли блок finally выполниться дважды?

Блок finally будет выполнен даже если вы выполняете return в try-блоке, а также если исключение не будет поймано в блоке catch (например, если блок catch отсутствует, или тип исключения не совпадает с типом, обрабатываемым блоком catch). А также, если код catch-блока сам выбросит исключение. Во всех этих случаях код *после* блока finally выполнен не будет.

Код finally не будет выполнен, если код до него успеет вызвать System.exit() или произойдёт креш JVM (или процесс будет каким-либо внешним образом уничтожен). С помощью демона тоже можно не дойти до выполнения finally.