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

1. Дайте развернутое объяснение трем концепциям ООП.

Инкапсуляция

Инкапсуляция выступает договором для объекта, что он должен скрыть, а что открыть для доступа другими объектами. В JAVA мы используем модификатор доступа private для того, чтобы скрыть метод и ограничить доступ к переменной из внешнего мира. JAVA также располагает различными модификаторами доступа: public, по умолчанию, protected, private, которые используются для ограничения видимости на разных уровнях. Но конечной целью является инкапсуляция тех вещей, которые не должны быть изменены. Лучше всего работает подход, при котором, у класса должна быть только одна причина для изменения, и инкапсулирование воплощает в реальность проектирование этой “одной причины”. Правильным в инкапсуляции считается сокрытие часто изменяющихся вещей во избежание повреждения других классов.

Преимущества:

- Мы можем защитить внутреннее состояние объекта с помощью сокрытия его атрибутов.
- Это улучшает модульное построение кода, так как предотвращает взаимодействие объектов неожиданными способами.
- Повышается практичность кода.
- Это поддерживает договорные отношения конкретного объекта.
- Инкапсуляция облегчает поддержку ПО.
- Изменения в коде могут производиться независимо друг от друга.

Наследование

Наследование — это включение поведения (т.е. методов) и состояния (т.е. переменных) базового класса в производный класс, таким образом они становятся доступны в этом производном классе. Главное преимущество наследования в том, что оно обеспечивает формальный механизм повторного использования кода и избегает дублирования. Унаследованный класс расширяет функциональность приложения благодаря копированию поведения родительского класса и добавлению новых функций. Это делает код сильно связанным. Если вы захотите изменить суперкласс, вам придется знать все

детали подклассов, чтобы не разрушить код. Наследование — это форма повторного использования программного обеспечения, когда из уже существующего класса (суперкласса) создается новый класс (подкласс), который расширяет свою функциональность и при этом использует некоторые свойства суперкласса. Так что, если у вас есть класс-родитель, а потом появляется класс-наследник, то наследник наследует все вещи, которыми обладает родитель.

Преимущества:

- Усовершенствованное повторное использование кода.
- Устанавливается логическое отношение «is a» (является кем-то, чем-то). Например: Dog is a*n* animal. (Собака является животным).
- Модуляризация кода.
- Исключаются повторения.

Недостаток:

- Сильная связанность: подкласс зависит от реализации родительского класса, что делает код сильно связанным.

Полиморфизм

Полиморфизм в программировании — это способность предоставлять один и тот же интерфейс для различных базовых форм (типов данных). Это означает, что классы, имеющие различную функциональность, совместно используют один и тот же интерфейс и могут быть динамически вызваны передачей параметров по ссылке. Классический пример — это класс Shape (фигура) и все классы, наследуемые от него: square (квадрат), circle (круг), dodecahedron (додекаэдр), irregular polygon (неправильный многоугольник), splat (клякса) и так далее. В этом примере каждый класс будет иметь свой собственный метод Draw() и клиентский код может просто делать:

```
Shape shape = new Shape();
```

Shape.area() чтобы получить корректное поведение любой фигуры Красота полиморфизма заключается в том, что код, работая с различными классами, не должен знать, какой класс он использует, так как все они работают по одному принципу. Процесс, применяемый объектно-ориентированными языками программирования для реализации динамического полиморфизма, называется динамическим связыванием. *Примечание:* Полиморфизм — это способность выбирать более конкретные методы для исполнения в

зависимости от объекта. Полиморфизм осуществляется тогда, когда не задействованы абстрактные классы.

Преимущества:

- Создание повторно используемого кода. То есть, как только класс создан, реализован и протестирован, он может свободно использоваться без заботы о том, что конкретно в нем написано.
- Это обеспечивает более универсальный и слабосвязанный код.
- Понижается время компиляции, что ускоряет разработку.
- Динамическое связывание.
- Один и тот же интерфейс может быть использован для создания методов с разными реализациями.
- Вся реализация может быть заменена с помощью использования одинаковых сигнатур метода.

Переопределение методов как часть полиморфизма. Переопределение взаимодействует с двумя методами: методом родительского класса и методом производного класса. Эти методы имеют одинаковые имя и сигнатуры. Переопределение позволяет вам производить одну и ту же операцию различными путями для разных типов объектов.

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 значения?

Поле класса по умолчанию 0 для примитивных типов, null для ссылок. Инициализировать другими значениями можно при объявлении или в динамическом и статическом (для статичных полей) блоках. Поля экземпляров инициализируются при создании объектов в конструкторе, методах (например сеттеры) или прямо при публичном модификаторе доступа поля.

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

Чтобы класс мог работать как bean, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении. Эти соглашения

дают возможность создания инструментов, которые могут использовать, замещать и соединять JavaBeans.

Правила описания гласят:

- Класс должен иметь конструктор без параметров, с модификатором доступа public. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
- Свойства класса должны быть доступны через get, set и другие методы (так называемые методы доступа), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean'ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
- Класс должен быть сериализуем. 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().

Так как требования в основном изложены в виде соглашения, а не интерфейса, некоторые разработчики рассматривают JavaBeans, как Plain Old Java Objects, которые следуют определённым правилам именования.

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 создание внутри одного класса методов с одинаковыми именами, но разными входными параметрами. Удобство в отсутствии необходимости придумывать новые имена методам, выполняющих схожие задачи. Любые методы могут быть перегружены. Методы с разными аргументами в базовом и производном классах не являются перегрузкой. Private метод базового класса нельзя перегрузить public методом

производного. Конструкторы можно перегружать, также можно при перегрузке конструкторов менять атрибуты доступа у конструкторов.

5. 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 – на этапе компиляции (static, final), позднее связывание – в рантайме (virtual в Java по умолчанию). Перегрузка – раннее связывание. Вызывается метод, соответствующих типу переданного аргумента. В случае примитивных типов при передаче значения можно добавить соответствующий идентификатор (.lf). При передаче ссылок, состоящих в иерархической связи. вызывается метод, соответствующий типу переданной ссылки, а не объекту, на который она ссылается.

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

this – это ссылка на текущий экземпляр класса. Присутствует в нестатических методах, отсутствует в статических, потому что не на что ссылаться.

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

С помощью final отмечаются поля, которые инициализируются только один раз. Инициализировать можно на месте, в конструкторе или в методах класса, но только 1 раз.

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?

Кроме обычных методов и полей класс может иметь статические поля, методы, константы и инициализаторы. Для объявления статических переменных, констант, методов и инициализаторов перед их объявлением указывается ключевое слово static. При создании объектов класса для каждого объекта создается своя копия нестатических обычных полей. А статические поля являются общими для всего класса. Поэтому они могут использоваться без создания объектов класса. Также статическими бывают константы, которые являются общими для всего класса. Статические инициализаторы предназначены для инициализации статических переменных, либо для выполнения таких действий, которые выполняются при создании самого первого объекта. Статические методы также относятся ко всему классу в целом.

При использовании статических методов надо учитывать ограничения:

– в статических методах мы можем вызывать только другие статические методы и использовать только статические переменные;

– статический метод нельзя переопределить. По аналогии с переменными, можно сказать, что этот метод "один для класса и его наследников" – так же, как статическая переменная "одна для класса и всех его объектов";

– все доступные методы наследуются подклассами;

– подкласс наследует все открытые и защищенные члены своего родителя, независимо от того, в каком пакете находится подкласс. Если подкласс находится в том же пакете, что и его родитель, он также наследует члены package-private родителя. Наследуемые элементы можно использовать как есть, заменять их, скрывать или дополнять новыми члены;

– единственное отличие от унаследованных статических (class) методов и унаследованных нестатических (instance) методов заключается в том, что при написании нового статического метода с той же сигнатурой старый статический метод просто скрыт, а не переопределен.

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?

Логические блоки инициализации – это участки кода, находящиеся внутри класса, но вне методов. Статические блоки инициализации – тоже самое, но со словом static. Сначала вызываются статические блоки в порядке объявления, потом нестатические, тоже в порядке объявления.

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?

Методы с переменным числом параметров могут принимать в качестве аргументов неопределенное количество параметров, которые передаются списком через запятую. Объявляется такой метод как func_name(type ... v). Выбор функции при перегрузке осуществляется, как и в обычном методе.

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

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

- public final Class<?> getClass() – возвращает класс объекта
- public int hashCode() – возвращает хеш-код объекта
- public boolean equals(Object obj) – возвращает истинность одинаковости объектов
- protected Object clone() – создает и возвращает копию объекта
- public String toString() – возвращает строковое представление объекта и другие.

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

Метод hashCode(). Зачем он нужен? Ровно для той же цели — сравнения объектов. Но ведь у нас уже есть equals()! Зачем же еще один метод? Ответ: для повышения производительности. Хэш-функция, которая представлена в Java методом hashCode(), возвращает числовое значение фиксированной длины для любого объекта. В случае с Java метод hashCode() возвращает для любого объекта 32-битное число типа int. Сравнить два числа между собой — гораздо быстрее, чем сравнить два объекта методом equals(), особенно если в нем используется много полей. Если в программе будут сравниваться объекты, гораздо проще сделать это по хэш-коду, и только если они равны по hashCode() — переходить к сравнению по equals(). Таким образом работают основанные на хеше структуры данных — например, известная HashMap! У разных объектов хеш-код должен быть разный. Но на практике иногда происходит по-другому. Очень часто это происходит из-за несовершенства формулы для вычисления хеш-кода.

13. 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?

Для создания иерархии классов, соответствующей предметной области, для которой создается программа.

```
abstract class Shape {
  abstract double getVolume();
}
class Cube extends Shape {
  double getVolume() {return a*a*a;}
}
class Sphere extends Shape {
  double getVolume() {return 4.*pi*r*r*r/3.;}
}
```

Нет, поля и методы, помеченныеи модификатором доступа private, не наследуются.

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()?

Cat cat = new Cat();

Сначала для хранения объекта выделяется память. Далее Java-машина создает ссылку на этот объект (в нашем случае ссылка — это Cat cat). В завершение происходит инициализация переменных и вызов. Первое что произойдет — про инициализируются статические переменные класса. После инициализации статических переменных класса-предка инициализируются статические переменные класса-потомка. Третьими по счету будут инициализированы нестатические переменные класса-предка. Наконец, дело дошло до конструктора базового класса. Начало его работы — четвертый пункт в процессе создания объекта. Теперь пришла очередь инициализации нестатических полей класса-потомка. Вызывается конструктор дочернего класса super – вызывает конструктор родителя. Должен стоять первым оператором в конструкторе потомке.

Да, возможно использовать super и this в одном конструкторе.

15. 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?

Эти утверждения означают одно и то же. Обратное, т.е. "ссылка производного класса может ссылаться на объекты своих базовых типов" и "объект базового класса может быть использован везде, где ожидается объект его производного типа" – неверные утверждения, т.к. в базовых типах может не быть тех интерфейсов, которые есть в производных, но обратное (интерфейсы базового есть в производных) выполняется всегда, поэтому изначальные выражения верны.

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?

Если в иерархии классов совпадают имена и сигнатуры типов методов из подкласса и суперкласса, то говорят, что метод из подкласса переопределяет метод из суперкласса. Переопределение методов выполняется только в том случае, если имена и сигнатуры типов обоих методов одинаковы. В противном случае оба метода считаются перегружаемыми.

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

При вызове, выполняется метод класса экземпляра, а не класса ссылки. Статические методы переопределить нестатическими нельзя, наоборот – тоже нельзя.

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

Класс, помеченный как final, не поддается наследованию и все его методы косвенным образом приобретают свойство final. Употребление модификатора final в объявлении метода или класса накладывает серьезные ограничения на возможность дальнейшего использования и развития кода. Применение final в объявлении метода – это верный показатель того, что реализация метода самодостаточна и полностью завершена. Другие программисты, которые захотят воспользоваться вашим классом, расширив его функции в угоду собственным потребностям, будут стеснены в выборе средств достижения цели либо полностью лишены таковых. Пометив признаком final класс в целом, вы запретите возможность его наследования и, вероятно, существенно снизите его практическую ценность для других. Собравшись применить модификатор final, убедитесь, готовы ли ВЫ к подобным жертвам и стоит ли их принести.

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

Приведение типов

Иногда ссылку на объект требуется привести к типу другого класса. Для такого приведения типов существует только одна причина: необходимость использовать все функциональные возможности объекта после того, как его фактический тип был на время забыт. 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.

Правила приведения типов при наследовании:
Приведение типов можно выполнять только в иерархии наследования 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.

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

Объект класса Class представляет классы и интерфейсы в запущенном приложении. Сравнение class и getClass() будет true, только если классы совпадают, instanceof же вернет true, также когда сравниваются класс и его родитель.

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

Соблюдать контракты hashCode и equals. toString желательно переопределять всегда (чтобы не выводил неинформативную информацию). Контракт equals (equals должен быть):

- рефлексивным (x.equals(x) = true)
- симметричным (y.equals(x) = x.equals(y))
- транзитивным (если x.equals(y) = true и y.equals(z) = true, то x.equals(z) = true)

- консистентным (если не менялись объекты, то и equals не должен изменить возвращаемое значение)

Контракт hashCode:

- внутренняя консистентность (не менялся объект – не меняется и hashCode)
- консистентность с equals (если x.equals(y) = true, то и x.hashCode() = y.hashCode())
- коллизии (даже если x.equals(y) = false, но hashCode могут совпадать)

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 — это максимально абстрактная, о-о-о-чень приблизительная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в готовом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.

Пример абстрактных классов Java:

```
public abstract class Car {
  private String model;
  private String color;
  private int maxSpeed;
  public abstract void gas();
  public abstract void brake();
  public String getModel() {
     return model;
  }
  public void setModel(String model) {
     this.model = model;
  }
  public String getColor() {
     return color;
  }
```

```
  public void setColor(String color) {
     this.color = color;
  }
  public int getMaxSpeed() {
     return maxSpeed;
  }
  public void setMaxSpeed(int maxSpeed) {
     this.maxSpeed = maxSpeed;
  }
}
```

Вот так выглядит самый простой абстрактный класс. Для чего он может понадобиться? Прежде всего, он максимально абстрактно описывает нужную сущность — автомобиль. Слово abstract находится здесь недаром. В мире не существует «просто машин». Есть грузовики, гоночные автомобили, седаны, купе, внедорожники. Абстрактный класс — это просто «чертеж», по которому позже будет создаваться классы-автомобили.

Да, в абстрактном классе в Java можно объявить и определить конструкторы. Поскольку создавать экземпляры абстрактных классов нельзя, вызвать такой конструктор можно только при формировании цепочки конструкторов, то есть при создании экземпляра конкретного класса-реализации.

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?

Создание интерфейса очень похоже на создание обычного класса, только вместо слова class мы указываем слово interface. Для чего нужен:

```
public interface Swimmable  {
    public void swim();
}
```

Мы создали интерфейс Swimmable — «умеющий плавать». Это что-то вроде пульта, у которого есть одна «кнопка»: метод swim() — «плыть».

Для этого метод, т.е. кнопку пульта, нужно имплементировать. Чтобы использовать интерфейс, его методы должны реализовать какие-то классы нашей программы.
Придумаем класс, объекты которого подойдут под описание «умеющий плавать». Например, подойдет класс утки — Duck:

```
public class Duck implements Swimmable {
   public void swim() {
      System.out.println("Уточка, плыви!");
   }
   public static void main(String[] args) {
      Duck duck = new Duck();
      duck.swim();
   }
}
```

Класс Duck «связывается» с интерфейсом Swimmable при помощи ключевого слова implements. Похожий механизм используют для связи двух классов в наследовании, только со словом «extends».

«public class Duck implements Swimmable» можно для понятности перевести дословно: «публичный класс Duck реализует интерфейс Swimmable». Это значит, что класс, связанный с каким-то интерфейсом, должен реализовать все его методы. В классе Duck прямо как в интерфейсе Swimmable есть метод swim(), и внутри него содержится какая-то логика.

Конструкторы нельзя объявить в интерфейсах. Проблема, возникающая при разрешении конструкторов в интерфейсах, возникает из-за возможности реализовать несколько интерфейсов одновременно. Когда класс реализует несколько интерфейсов, которые определяют разные конструкторы, класс должен будет реализовать несколько конструкторов, каждый из которых удовлетворяет только одному интерфейсу, но не другим. Невозможно построить объект, который вызывает каждый из этих конструкторов. 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.

Как и в случае с объектными ссылками суперкласса, через интерфейсную ссылку можно сослаться не на все члены объекта реализующего интерфейс класса. Доступны только те методы, которые объявлены в соответствующем интерфейсе. С учетом того, что класс может реализовать несколько интерфейсов, а один и тот же интерфейс может быть реализован в разных классах, ситуация представляется достаточно пикантной.

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

Иногда необходимо получить копию объекта, которая не зависела бы от оригинала. С которой можно было бы производить манипуляции, при этом, не изменяя оригинал. При обыкновенном присваивание объектов (obj1 = obj2;) передаются ссылки на объект. В итоге два экземпляра ссылаются на один объект, и изменение одного приведет к изменению другого. Это не то, что нам нужно. В данном случае, нам на помощь придет интерфейс Cloneable и метод clone() класса Object. И так, если нам необходимо получить независимый клон объекта, то необходимо вызвать метод clone(). Данный метод объявлен, как protected, а это значит, что метод защищен, и может быть доступен только при наследовании объекта. Это не является проблемой, потому как любой класс, является потомком класса Object. Однако при защищенном методе класс может клонировать только свои собственные объекты. Чтобы клонировать другие объекты, метод clone() необходимо расширить до public.

Пример расширения метода clone().

```
public User clone() throws CloneNotSupportedException {
    return (User)super.clone();
}
```

Метод clone() может выбрасывать исключение CloneNotSupportedException. Данное исключение возникает в случае, когда клонируемый класс не имеет реализации интерфейса Cloneable. Интерфейс Cloneable не реализует ни одного метода. Он является всего лишь маркером, говорящим, что данный класс реализует клонирование объекта. Само клонирование осуществляется

вызовом родительского метода clone(). Данный вид клонирования называется поверхностным клонированием. Его можно использовать только в том случае, если у клонируемого класса объявлены неизменяемые типы объекты.
Если есть необходимость воспользоваться «клонированием по умолчанию», которое реализовано в классе Object, нужно:

а) Добавить интерфейс Cloneable своему классу;

б) Переопределить метод clone и вызвать в нем базовую реализацию:

```
class Point implements Cloneable
{
 int x;
 int y;
 public Object clone()
 {
  return super.clone();
 }
}
```

Или можно написать реализацию метода clone полностью:

```
class Point
{
 int x;
 int y;
 public Object clone()
 {
  Point point = new Point();
  point.x = this.x;
  point.y = this.y;
  return point;
 }
}
```

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

Comparable – функциональный интерфейс, содержащий метод compareTo. Comparator – интерфейс, содержащий много методов для сравнения. Comparable используется для быстрой реализации сравнения, но необходимо

редактировать класс. Comparator же позволяет реализовать разные способы сортировки, и для его использования редактировать сравниваемый класс не нужно. Пример:

```
class MyNumber implements Comparable {
  private int n;
  MyNumber(int n) {
    this.n = n;
  }

  public int getN() {
    return n;
  }

  int compareTo(MyNumber o) {
    if (o.n < n) {
      return 1;
    } else if (o.n > n) {
      return -1;
    } else {
      return 0;
    }
  }
}

class MyNumberComparator implements Comparator {
  int compare(MyNumber n1, MyNumber n2) {
    if (n1.getN() < n2.getN()) {
      return -1;
    } else if (n1.getN() > n2.getN()) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Generic classes and Interfaces, Enums

26. 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:

```
enum Day{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

Перечисление фактически представляет новый тип, поэтому мы можем определить переменную данного типа и использовать ее. Объявляя enum мы неявно создаем класс производный от java.lang.Enum. Условно конструкция enum Season { ... } эквивалентна class Season extends java.lang.Enum { ... }. И хотя явным образом наследоваться от java.lang.Enum нам не позволяет компилятор, все же в том, что enum наследуется, легко убедиться с помощью reflection.

27. 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?

Вы не можете 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.

28. Можем ли мы указать конструктор внутри Enum?

Этот вопрос часто следует за предыдущим. Да, можно, но следует помнить, что подобное возможно лишь с указанием private или package-private конструкторов. Конструкторы с public и protected — не допустимы в Enum.

29. Что лучше использовать для сравнения enum'ов — == или equals()?

Они оба работают. Как написано в документации, "допустимо использовать оператор == вместо метода equals, если доподлинно известно, что хотя бы один из них ссылается на перечислимый тип" ("it is permissible to use the == operator in place of the equals method when comparing two object references if it is known that at least one of them refers to a n enum constant"). Причина этого очень простая — каждый из объектов enum'а создаётся только единожды, и поэтому, если вы создадите десять переменных равных SomeEnum.RED, они все будут ссылаться на один и тот же объект (а оператор == как раз это и проверяет).

30. Что такое анонимные классы?

Анонимные классы – внутренние классы без названия.

31. 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?

```
class Account<T>{
    private T id;
    private int sum;
    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }
    public T getId() { return id; }
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

С помощью буквы T в определении класса class Account<T> мы указываем, что данный тип T будет использоваться этим классом. Параметр T в угловых скобках называется универсальным параметром, так как вместо него можно подставить любой тип. При этом пока мы не знаем, какой именно это будет тип: String, int или какой-то другой. Причем буква T выбрана условно, это может и любая другая буква или набор символов. После объявления класса мы можем применить универсальный параметр T: так далее в классе объявляется переменная этого типа, которой затем присваивается значение в конструкторе.

Метод getId() возвращает значение переменной id, но так как данная переменная представляет тип T, то данный метод также возвращает объект типа T: public T getId().

Используем данный класс:

```
public class Program{

    public static void main(String[] args) {

        Account<String> acc1 = new Account<String>("2345", 5000);
        String acc1Id = acc1.getId();
        System.out.println(acc1Id);

        Account<Integer> acc2 = new Account<Integer>(2345, 5000);
        Integer acc2Id = acc2.getId();
        System.out.println(acc2Id);
    }
}
class Account<T>{

    private T id;
    private int sum;

    Account(T id, int sum){
        this.id = id;
        this.sum = sum;
    }

    public T getId() { return id; }
```

```
    public int getSum() { return sum; }
    public void setSum(int sum) { this.sum = sum; }
}
```

При определении переменной данного класса и создании объекта после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо универсального параметра. При этом надо учитывать, что они работают только с объектами, но не работают с примитивными типами. То есть мы можем написать Account<Integer>, но не можем использовать тип int или double, например, Account<int>. Вместо примитивных типов надо использовать классы-обертки: Integer вместо int, Double вместо double.

Exceptions and Errors

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

Для программы является исключительной ситуацией невозможность дальнейшего выполнения. Способом обработки ошибок является создание try-catch блоков.

33. 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?

Для Java-программы является исключительной ситуацией создание объекта Throwable. “Программа выбросила исключение” – означает появление исключительной ситуации, которая не была обработана самой программой, и была выведена пользователю. Исключения делятся на checked и unchecked, первые создаются вручную, вторые – автоматически.

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

Исключения делятся на несколько классов, но все они имеют общего предка — класс Throwable. Его потомками являются подклассы Exception и Error.

Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы и предсказуемы. Например, произошло деление на ноль в целых числах.

Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM.

В Java все исключения делятся на три типа: контролируемые исключения (checked) и неконтролируемые исключения (unchecked), к которым относятся ошибки (Errors) и исключения времени выполнения (RuntimeExceptions, потомок класса Exception).
Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса Exception (но не RuntimeException). Обработка исключения может быть произведена с помощью операторов try…catch, либо передана внешней части программы. Например, метод может передавать возникшие в нём исключения выше по иерархии вызовов, сам его не обрабатывая.
Неконтролируемые исключения не требуют обязательной обработки, однако, при желании, можно обрабатывать исключения класса RuntimeException.

35. 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.

Кратко о ключевых словах try, catch, finally, throws
Обработка исключений в Java основана на использовании в программе следующих ключевых слов:

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.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций:

try{}catch, try{}catch{}finally, try{}finally{}.

throw – используется для возбуждения исключения;

throws – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

36. 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.

Блоки catch располагаются в порядке от более конкретных к более общим. Повторно сгенерированное исключение передается на уровень выше. Блок catch может выбрасывать иные исключения (которые не перехватываются catch) при невозможности их обработки, чтобы их обработали выше.

37. 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?

Вызов блока finally происходит после выполнения блоков try и catch (если он был вызван). Не существуют ситуации, когда блок finally не будет вызван. Блок finally может выбрасывать исключения. Блок finally не может выполнится дважды.