4. Programming with classes

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

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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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.

12. 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.