ООП

Концепция ООП

Объектно-ориентированное программирование: парадигма программирования, в которой главной идеей являются понятия *объектов* и *классов*.

ООП возникло в результате развития идей процедурного программирования, где данные и функции (методы) их обработки формально не связаны.

Инкапсуляция

Инкапсуляция - свойство системы, позволяющее объединить данные и методы, а

так же скрыть детали реализации от пользователя.
Инкапсуляция - один из главных принципов ООП, поэтому и не удивительно, что в Java предлагается свое решение этой проблемы. Всякий раз, когда подклассу требуется сослаться на его непосредственный суперкласс, это можно сделать с помощью ключевого слова super.
У ключевого слова super имеются две общие формы. Первая форма служит для вызова конструктора суперкласса, вторая - для обращения к члену суперкласса, скрываемому членом подкласса.

Вызов конструкторов суперкласса
с помощью ключевого слова super
Из подкласса можно вызывать конструктор, определенный в его суперклассе,
используя следующую форму ключевого слова super:
super (список _ аргументов) ;

где список_ аргументов определяет любые аргументы, требующиеся конструктору в суперклассе. Вызов метода supe r () всегда должен быть первым
оператором,выполняемым в конструкторе подкласса.
Другое применение ключевого

слова super

Вторая форма ключевого слова super действует подобно ключевому слову this, за исключением того, что ссылка всегда делается на суперкласс того подкласса, в котором используется это ключевое слово.

В общем виде эта форма применения ключевого слова super выглядит следующим образом:

super . член

где член может быть методом или переменной экземпляра. Вторая форма применения ключевого слова super наиболее пригодна в тех случаях, когда имена членов

подкласса скрывают члены суперкласса с такими же именами.

Под инкапсуляцией подразумевается сокрытие пол внутри объекта с целью защиты данных от внешнего, бесконтрольного изменения со стороны других объектов. Доступ к данным (полям) предоставляется посредством публичных методов (геттеров/сеттеров). Это защитный барьер позволяет хранить информацию в безопасности внутри объекта.

Принцип работы инкапсуляции

Инкапсуляция позволяет нам пользоваться возможностями класса без создания угрозы безопасности данных за счет ограничения прямого доступа к его полям. Также она позволяет изменять код классов не создавая проблем их пользователям (другим классам). В Java данный принцип достигается за счет использования ключевого слова **private**

```
public class Car {
```

```
    private String carBrand;

    public void
setCarBrand(String carBrand)
{
        this.carBrand =
carBrand;
    }
}
```

Наследование

Наследование (отношение «является», **is a** relationship) - свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Одним из основополагающих принципов объектно-ориентированного программирования является наследование, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать класс, который

определяет характеристики, общие для набора связанных элементов. Затем этот общий класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять свои особые характеристики.
В терминологии Java наследуемый класс называется суперклассом, а наследующий класс - подКЛассом. Следовательно, подкласс - это специализированная версия суперкласса.
Он наследует все члены, определенные в суперклассе, добавляя к ним собственные, особые элементы.

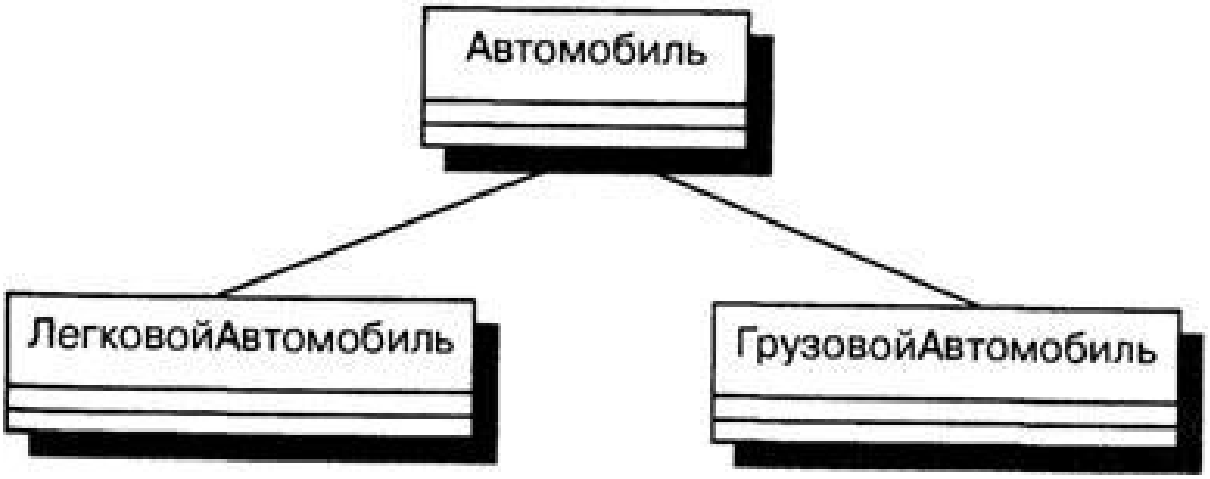
Для каждого создаваемого подкласса можно указать только один суперкласс.
В Java не поддерживается наследование нескольких суперклассов в одном подклассе. Как отмечалось ранее, можно создать иерархию наследования, в которой один подкласс становится суперклассом другого подкласса. Но ни один из классов не может стать суперклассом для самого себя.
Несмотря на то что подкласс включает в себя все члены своего суперкласса, он не может иметь доступ к тем членам супер класса, которые объявлены как private.
Помните! Член класса, который объявлен закрытым, остается закрытым в пределах своего класса. Он недоступен для любого кода за пределами его класса , в том числе для подклассов.

Это особая функциональность в объектно-ориентированных языках программирования, которая позволяет описывать новые классы на основе уже существующих. При этом поля и методы

класса-предка становятся доступны и классам-наследникам. Данная фича делает классы более чистыми и понятным за счет устранения дублирования программного кода.

Принцип работы наследования

Наследование — еще одна важная концепция ООП, которая позволяет сэкономить время на написании кода. Возможности наследования раскрываются в том, что новому классу передаются свойства и методы уже описанного ранее класса. Класс, который наследуется называется дочерним (или подклассом). Класс, от которого наследуется новый класс — называется родительским, предком и т. д. В языке программирования Java используется ключевое слово **extends** для того, чтобы указать на класс-предок.




```
public class Car {
```

```
        ……
}
public class PassengerCar extends Car {
        ……
}
public class Truck extends  Car {
        ……
}
```

Полиморфизм

Полиморфизм - свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Данный принцип позволяет программистам использовать одни и те же термины для описания различного

поведения, зависящего от контекста. Одной из форм полиморфизма в Java является переопределение метода, когда различные формы поведения определяются объектом из которого данный метод был вызван. Другой формой полиморфизма является перегрузка метода, когда его поведение определяется набором передаваемых в метод аргументов.

Принцип работы полиморфизма

Полиморфизм предоставляет возможность единообразно обрабатывать объекты с различной реализацией при условии наличия у них общего интерфейса или класса. По-простому: способность вызывать нужные методы у объектов, имеющие разные типы (но находящиеся в одной иерархии). При этом происходит автоматический выбор нужного метода в зависимости от типа объекта.

Рассмотрим примеры полиморфизма в Java: переопределение и перегрузка методов.

В случае с переопределением метода, дочерний класс, используя концепцию полиморфизма, может изменить (переопределить) поведение метода родительского класса. Это позволяет программисту по разному использовать один и тот же метод, определяя поведение из контекста вызова (вызывается метод из класса предка или класса наследника).

В случае же с перегрузкой, метод может проявлять различное поведение в зависимости от того, какие аргументы он принимает. В данном случае контекст вызова определяется набором параметров метода.

- **Абстракция**. Абстракция означает использование простых вещей для описания чего-то сложного. Например, мы все знаем как пользоваться телевизором, но в тоже время нам не нужно обладать знаниями о том, как он работает чтобы смотреть его. В Java под абстракцией подразумеваются такие вещи, как **объекты**, **классы** и **переменные**, которые в свою очередь лежат в основе более сложного кода. Использование данного принципа позволяет избежать сложности при разработке ПО.

Принцип работы абстракции

Основная цель использования данной концепции — это уменьшение сложности компонентов программы за счет скрытия от программиста, использующего эти компоненты, ненужных ему подробностей. Это позволяет реализовать более сложную логику поверх предоставленной абстракции, не вдаваясь в подробности ее реализации.

Приготовление кофе с помощью кофемашины является хорошим примером абстракции. Все, что нам надо знать, что бы ей пользоваться: как налить воды, засыпать

кофейные зерна, включить и выбрать вид кофе, который хотим получить. А, как машина будет варить кофе — нам знать не нужно.

В данном примере кофемашина представляет собой абстракцию, которая от нас скрывает все подробности варки кофе. Нам лишь остается просто взаимодействовать с простым интерфейсом, который не требует от нас

Модификатор final

Модификатор **final** (неизменяемый) может применяться к классам, методам и переменным.

Ключевое слово final

Поле может быть объявлено как final (завершенное). Это позволяет предотвратить изменение содержимого переменной, сделав ее, по существу, константой.

Следовательно, завершенное поле должно быть инициализировано во время его объявления.. Значение такому полю можно присвоить и в пределах конструктора, но первый подход более распространен. Ниже приведен ряд примеров объявления завершенных полей

```
final int FILE_NEW = 1;
final int FILE_ OPEN = 2;
final int FILE_ SAVE = 3;

final int FILE_ SAVEAS = 4;
final int FILE_QUIT = 5;
```

Теперь во всех последующих частях программы можно

пользоваться полем

FILE _ OPEN и прочими полями таким образом, как если бы они были константами,

без риска изменить их значения. В практике программирования на java идентификаторы всех завершенных полей принято обозначать прописными буквами, как
в приведенном выше примере.
Кроме полей, объявленными как final могут быть параметры метода и локальные переменные. Объявление параметра как final препятствует его изменению

в пределах метода, тогда как аналогичное объявление

локальной переменной - присвоению ей значения больше одного раза.

Ключевое слово final можно указывать и в объявлении методов, но в этом случае оно имеет совсем иное назначение, чем в переменных. Это дополнительное применение ключевого слова final более подробно описано в следующей главе, посвященной наследованию.

Ключевое слово final в сочетании с наследованием

Ключевое слово final можно использовать тремя способами. Первый способ служит для

создания эквивалента именованной константы. Такое применение ключевого слова final было описано в предыдущей главе. А два других способа его применения относятся к наследованию. Рассмотрим их подробнее.

Предотвращение переопределения с помощью ключевого слова final

Несмотря на то что переопределение методов является одним из самых эффективных языковых средств Java, иногда его желательно избегать. Чтобы запретить переопределение метода, в

начале его объявления следует указать ключевое слово final.

Методы, объявленные как final, переопределяться не могут.

Иногда методы, объявленные как final , могут способствовать увеличению производительности программы. Компилятор вправе встраивать вызовы этих

методов, поскольку ему известно, что они не будут переопределены в подклассе.Нередко при вызове небольшого завершенного метода компилятор Java может встраивать байт-код для подпрограммы непосредственно в скомпилированный

код вызывающего метода, тем

самым снижая издержки на вызов метода. Такая возможность встраивания вызовов присуща только завершенным методам. Как правило, вызовы методов разрешаются в Java динамически во время выполнения.

Такой способ называется поздним связыванием. Но поскольку завершенные методы не могут быть переопределены, их вызовы могут быть разрешены во время компиляции. И такой способ называется ранним связыванием.

Предотвращения наследования с помощью ключевого слова final

Иногда требуется предотвратить наследование класса. Для этого в начале объявления класса

следует указать ключевое слово final. Объявление класса завершенным неявно делает завершенными и все его методы. Нетрудно догадаться,
что одновременное объявление класса как abstract и final недопустимо, поскольку абстрактный класс принципиально является незавершенным и только его подклассы предоставляют полную реализацию методов.

К переменным:

```
final double PI = 3.14;
```

- константы

К методом:

```
final void grow(){}
```

- запрещено переопределение

метода

К классам:

`final class A{}` - запрещено наследование

Модификатор static

Модификатор static (единственный) применяется к методам, переменным и логическим блокам.

К методам:

`static void grow(){}` - вызывать функцию можно обращаться через имя класса `Man.grow();`

К переменным:

`static int counter;` - переменная общая для всех объектов

1. Назовите принципы ООП и расскажите о каждом.

Объе́ктно-ориенти́рованное программи́рование (ООП) — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Основные принципы ООП: абстракция, инкапсуляция, наследование, полиморфизм.

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.

Пример: описание должностей в компании. Здесь название должности значимая информация, а описание обязанностей у каждой должности это второстепенная информация. К примеру главной характеристикой для «директор» будет то, что это должность чем-то управляет, а чем именно (директор по персоналу, финансовый директор, исполнительный директор) это уже второстепенная информация.

Инкапсуляция — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Для Java корректно будет говорить, что инкапсуляция это «сокрытие реализации». Пример из жизни — пульт от телевизора. Мы нажимаем кнопочку «увеличить громкость» и она увеличивается, но в этот момент происходят десятки процессов, которые скрыты от нас. Для Java: можно создать класс с 10 методами, например вычисляющие площадь сложной фигуры, но сделать из них 9 private. 10 й метод будет называться «вычислить Площадь()» и объявлен public, а в нем уже будут вызываться

необходимые скрытые от пользователя методы. Именно его и будет вызывать пользователь.

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. Пример (чуть переделанный) из Thinking in Java:

```
1  public interface Shape {
2      void draw();
3      void erase();
4  }
5  public class Circle implements Shape {
6      public void draw() {
7          System.out.println("Circle.draw()");
8      }
9  }
10 public class Triangle implements Shape {
11     public void draw() {
12         System.out.println("Triangle.draw()");
13     }
14 }
15
16 public class TestPol {
17
18     public static void main(String[] args) {
19         Shape shape1 = new Circle();
20         Shape shape2 = new Triangle();
```

```
21          testPoly(shape1);
22          testPoly(shape2);
23      }
24
25      public static void testPoly(Shape shape) {
26          shape.draw();
27      }
28  }
29  //Вывод в консоль:
30  //Circle.draw()
31  //Triangle.draw()
```

Есть общий интерфейс «Фигура» и две его реализации «Треугольник» и «Круг». У каждого есть метод «нарисовать». Благодаря полиморфизму нам нет нужды писать отдельный метод для каждой из множества фигур, чтобы вызвать метод «нарисовать». Вызов полиморфного метода позволяет одному типу выразить свое отличие от другого, сходного типа, хотя они и происходят от одного базового типа. Это отличие выражается различным действием методов, вызываемых через базовый класс (или интерфейс).

Здесь приведен пример полиморфизма (также называемый динамическим связыванием, или поздним связыванием, или связыванием во время выполнения), в котором продемонстрировано как во время выполнения программы будет выполнен тот метод, который принадлежит передаваемому объекту.

Если бы не было полиморфизма и позднего связывания, то эта же программа выглядела бы примерно так:

```
1  public static void testPolyCircle(Circle circle) {
2          circle.draw();
3      }
4      public static void testPolyTriangle(Triangle triangle) {
```

```
        triangle.draw();
    }
```

Т.е. для каждого класса (фигуры) мы бы писали отдельный метод. Здесь их два, а если фигур (классов) сотни?

2. Дайте определение понятию “класс”.

Класс – это шаблон, описывающий общие свойства группы объектов. Этими свойствами могут быть как характеристики объектов (размер, вес, цвет и т.п.), так и поведения, роли и т.п.

3. Что такое поле/атрибут класса?

Поле (атрибут) класса — это характеристика объекта. Например для фигуры это может быть название, площадь, периметр.

```
public class Circle implements Shape {

    private String name;
    private Double area;
    private String perimeter;

}
```

4. Как правильно организовать доступ к полям класса?

Модификатор доступа — private. Доступ через методы get\set.

5. Дайте определение понятию “конструктор”.

Конструктор — это специальный метод, который вызывается при создании нового объекта. Конструктор инициализирует объект непосредственно во время создания. Имя конструктора совпадает с именем класса, включая регистр, а по синтаксису конструктор похож на метод без возвращаемого значения.

```
public class Circle implements Shape {

   public Circle() {
    }
}
```

6. Чем отличаются конструкторы по умолчанию, копирования и конструктор с параметрами?

Конструктор по умолчанию не принимает никаких параметров. Конструктор копирования принимает в качестве параметра объект класса. Конструктор с параметрами принимает на вход параметры (обычно необходимые для инициализации полей класса).

```
    //конструктор по умолчанию
    public Circle() {
    }

    //конструктор копирования
    public Circle(Circle circle) {
                          this(circle.getName(),  circle.getArea(),
circle.getPerimeter()); //будет вызван конструктор с параметрами
ниже
    }

    //конструктор с параметрами
    public Circle(String name, Double area, String perimeter) {
        this.name = name;
        this.area = area;
        this.perimeter = perimeter;
    }
```

Обращаю внимание, что тема копирования (clone()) достаточно глубокая с возможностью возникновения множества неявных проблем. Немного можно почитать

здесь http://habrahabr.ru/post/246993/.

7. Какие модификации уровня доступа вы знаете, расскажите про каждый из них.

- **private (закрытый) — доступ к члену класса не предоставляется никому, кроме методов этого класса. Другие классы того же пакета также не могут обращаться к private-членам.**
- **default, package, friendly, доступ по умолчанию, когда никакой модификатор не присутствует — член класса считается открытым внутри своего собственного пакета, но не доступен для кода, расположенного вне этого пакета.Т.е. если package2.Class2 extends package1.MainClass, то в Class2 методы без идентификатора из MainClass видны не будут.**
- **protected (защищённый) — доступ в пределах пакета и классов наследников. Доступ в классе из другого пакета будет к методам public и protected главного класса. Т.е. если package2.Class2 extends package1.MainClass, то внутри package2.Class2 методы с идентификатором protected из MainClass будут видны.**
- **public (открытый) — доступ для всех из любого другого кода проекта**

Модификаторы в списке расположены по возрастающей видимости в программе.

8. Расскажите об особенностях класса с единственным закрытым (private) конструктором.

Невозможно создать объект класса у которого единственный private конструктор за пределами класса. Поэтому нельзя унаследоваться от такого класса. При попытке унаследоваться будет выдаваться ошибка: There is no default constructor available in имяКласса. А при попытке создать объект этого класса: ИмяКласса() has private access in ИмяКласса

9. О чем говорят ключевые слова “this”, “super”, где и как их можно использовать?

super — используется для обращения к базовому классу, а this к текущему. Пример:

```
1  public class Animal {
2
3      public void eat() {
4          System.out.println("animal eat");
5      }
6  }
7
8  public class Dog extends Animal {
9
10     public void eat() {
11         System.out.println("Dog eat");
12     }
13     public void thisEat() {
14         System.out.println("Call Dog.eat()");
15         this.eat();
16     }
17
18     public void superEat() {
19         System.out.println("Call Animal.eat()");
20         super.eat();
21     }
22
23 }
24
25 public class Test {
26
27     public static void main(String[] args) {
28         Dog dog = new Dog();
29         dog.eat();
30         dog.thisEat();
31         dog.superEat();
32     }
```

```
}
Dog eat
Call Dog.eat()
Dog eat
Call Animal.eat()
animal eat
```

Если написать super(), то будет вызван конструктор базового класса, а если this(), то конструктор текущего класса. Это можно использовать, например, при вызове конструктора с параметрами:

```
public Dog() {
        System.out.println("Call empty constructor");
    }

    public Dog(String name) {
        System.out.println("Call constructor with Name");
        this.name = name;

    }

    public Dog(String name, Double weight) {
        this(name);
        this.weight = weight;
              System.out.println("Call  constructor  with  Name  and Weight");
    }
}
..
public static void main(String[] args) {
Dog dog1 = new Dog("name", 25.0);
}
//Вывод
Call constructor with Name
```

```
23  Call constructor with Name and Weight
```

10. Дайте определение понятию “метод”.

Метод — это последовательность команд, которые вызываются по определенному имени. Можно сказать что это функция и процедура (в случае void метода).

11. Что такое сигнатура метода?

Сигнатура метода в Java — это имя метода плюс параметры (причем порядок параметров имеет значение).

В сигнатуру метода не входит возвращаемое значение, бросаемые им исключения, а также модификаторы.

Ключевые слова public, protected, private, abstract, static, final, synchronized, native, strictfp в т.ч. аннотации для метода — это модификаторы и не являются частью сигнатуры.

http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.4.2

12. Какие методы называются перегруженными?

Java позволяет создавать несколько методов с одинаковыми именами, но разными параметрами. Создание метода с тем же именем, но с другим набором параметров называется перегрузкой. Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров.

```
1  public void method() { }
2
3  public void method(int a) { }
4
5  public void method(String str) { }
```

13. Могут ли нестатические методы перегрузить статические?

Да. Это будут просто два разных метода для программы. Статический будет доступен по

имени класса.

14. Расскажите про переопределение методов. Могут ли быть переопределены статические методы?

Метод в классе-наследнике, совпадающий по сигнатуре с методом из родительского класса называется переопределенным методом. Переопределить базовый статический метод нельзя: **Instance method имяМетода in классНаследник cannot override method имяМетода in родительскийКласс**

15. Может ли метод принимать разное количество параметров (аргументы переменной длины)?

Да. Запись имеет вид **method(type … val)**. **Например public void method(String … strings), где strings это массив, т.е. можно записать**

```
1  public void method (String ... strings) {
2          for (String s : strings) {
3
4          }
5      }
```

16. Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?

При переопределении метода нельзя сузить модификатор доступа к методу (например с public в MainClass до private в Class extends MainClass). Изменить тип возвращаемого значения при переопределении метода нельзя, будет ошибка **attempting to use incompatible return type.**

Но можно сузить возвращаемое значение, если они совместимы. Например:

```
1  public class Animal {
2
3      public Animal eat() {
4          System.out.println("animal eat");
5          return null;
6      }
7
8      public Long calc() {
9          return null;
10     }
11
12 }
13 public class Dog extends Animal {
14
15     public Dog eat() {
16         return new Dog();
17     }
18 /*attempting to use incompatible return type
19     public Integer calc() {
20         return null;
21     }
22 */
23 }
```

17. Как получить доступ к переопределенным методам родительского класса?

super.method();

18. Какие преобразования называются нисходящими и восходящими?

Преобразование от потомка к предку называется восходящим, от предка к потомку — нисходящим. Нисходящее преобразование должно указываться явно с помощью указания нового типа в скобках.

Например:

```
Animal dog = new Dog(); //восходящее преобразование. Будет потерян доступ ко всем методам, которые есть только у класса Dog.


int x = 100;

byte y = (byte) x;    //нисходящее преобразование. Должно быть указано явно
```

19. Чем отличается переопределение от перегрузки?

Переопределение используется тогда, когда вы переписываете (переделываете, переопределяете) УЖЕ существующий метод. Перегрузка — это использование одного имени, но с разными входными параметрами. Например нам нужно, чтобы метод toString() для нашего класса выдавал какой-то осмысленный текст. Тогда мы переопределяем метод из класса Object и реализуем этот метод так, как нам это нужно.

```
@Override
public String toString() {
return "Хочу чтобы писался текст, а не название класса@2234SD!"
}
```

Тогда как перегрузка обычно используется, чтобы не придумывать каждый раз новое имя, когда методы отличаются только входными параметрами. При перегрузке необходимый метод определяется на этапе компиляции на основе сигнатуры вызываемого метода, тогда как при переопределении нужный метод будет выявлен во время выполнения исходя из реального типа объекта.

20. Где можно инициализировать статические/нестатические поля?

Статические поля можно инициализировать при объявлении, в статическом или динамическом блоке инициализации. Нестатические поля можно инициализировать при объявлении, в динамическом блоке инициализации или в конструкторе

21. Зачем нужен оператор instanceof?

Оператор instanceof возвращает true, если объект является экземпляром класса или его потомком.

```
1  public class MainClass {
2      public static void main(String[] a) {
3
4      String s = "Hello";
5      int i = 0;
6      String g;
7      if (s instanceof java.lang.String) {
8        // попадем сюда, т.к. выражение будет true
9        System.out.println("s is a String");
10     }
11     if (i instanceof Integer) {
12       //  это  отобразится,  т.к.  будет  использована  автоупаковка
(int -> Integer)
13       System.out.println("i is an Integer");
14
15     }
16     if (g instanceof java.lang.String) {
17       // g не инициализирована и поэтому сюда мы не попадем, т.к.
18       // g - null и instanceof вернет false для null.
19       System.out.println("g is a String");
     }
```

```
20  }
```

22. Зачем нужны и какие бывают блоки инициализации?

Блоки инициализации представляют собой наборы выражений инициализации полей, заключенные в фигурные скобки и размещаемые внутри класса вне объявлений методов или конструкторов. Блок инициализации выполняется так же, как если бы он был расположен в верхней части тела любого конструктора. Если блоков инициализации несколько, они выполняются в порядке следования в тексте класса. Блок инициализации способен генерировать исключения, если их объявления перечислены в предложениях throws всех конструкторов класса.

Бывают статические и нестатические блоки инициализации. Также возможно создать такой блок в анонимном классе.

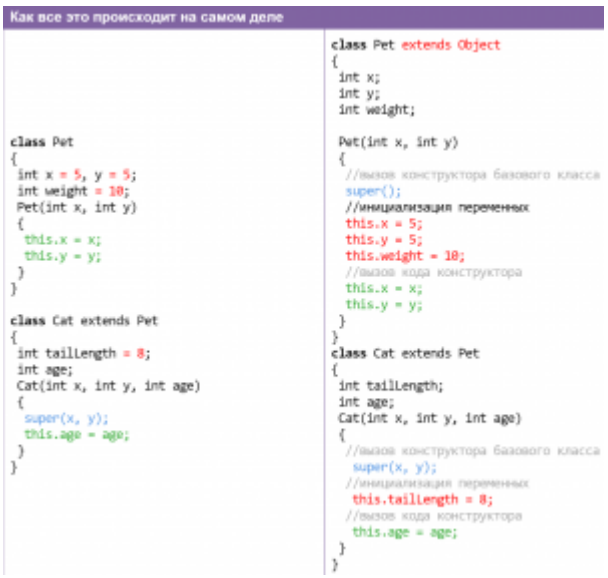
```
1   class Foo {
2         static List<Character> abc;
3         static {
4               abc = new LinkedList<Character>();
5               for (char c = 'A'; c <= 'Z'; ++c) {
6                     abc.add( c );
7               }
8         }
9   }
10  //Пример нестатического блока инициализации
11  class Bar {
```

```
12        {
13              System.out.println("Bar: новый экземпляр");
14        }
15  }
16  //Пример инициализации в анонимном классе
17  JFrame frame = new JFrame() {{
18        add(new JPanel() {{
19              add(new JLabel("Хабрахабр?") {{
20                    setBackground(Color.BLACK);
21                    setForeground(Color.WHITE);
22              }});
23        }});
24  }};
```

23. Каков порядок вызова конструкторов и блоков инициализации двух классов: потомка и его предка?

Сначала вызываются все статические блоки от первого предка до последнего наследника. Потом попарно вызываются динамический блок инициализации и конструктор в той же последовательности (от предка до последнего потомка).

Хорошая картинка, демонстрирующая что происходит на самом деле при инициализации с ресурса javarush.ru:



Вот простенький пример, который это демонстрирует:

```
public class Pet {

    private String name;

    static {

        System.out.println("Static block in Pet");

    }

    {

        System.out.println("First block in Pet");

    }

    {

        System.out.println("Second block in Pet");

    }


    public Pet() {

        System.out.println("Pet empty constructor");

    }


    public Pet(String name) {

        System.out.println("Pet constructor with Name " + name);

        this.name = name;

    }

}
```

```
public class Cat extends Pet {

    private String name;
    static {
        System.out.println("Static block in Cat");
    }
    {
        System.out.println("First block in Cat");
    }
    {
        System.out.println("Second block in Cat");
    }

    public Cat() {
        System.out.println("Cat empty constructor");
    }

    public Cat(String name) {
          super(name); // without this will call super(). Если эту строчку убрать, то будет вызван конструктор Pet();
        System.out.println("Cat constructor with Name: " + name);
```

```
        this.name = name;
    }

}

public class TestInitOrder {

    public static void main(String[] args) {
        Cat cat = new Cat("Rizhick");
    }
}
//Вывод
Static block in Pet
Static block in Cat
First block in Pet
Second block in Pet
Pet constructor with Name Rizhick
First block in Cat
Second block in Cat
Cat constructor with Name: Rizhick
```

2

4

3

4

4

4

5

4

6

4

7

4

8

4

9

5

0

5

1

5

2

5

3

5

4

```
55
56
57
58
59
60
61
62
```

24. Где и для чего используется модификатор abstract?

Абстрактным называется класс, на основе которого не могут создаваться объекты. При этом наследники класса могут быть не абстрактными, на их основе объекты создавать, соответственно, можно. Для того, чтобы превратить класс в абстрактный перед его именем надо указать модификатор abstract.

Абстрактный метод — метод, который не имеет реализации. Если в классе есть хотя бы один абстрактный метод, то весь класс должен быть объявлен абстрактным.

```
1  public abstract class Fighter {
2      abstract void fight();
3  }
4
5  public class JudoFighter extends Fighter {
6      @Override
7      void fight() {
8              System.out.println("Учу ушу, руками машу! Бью с лету в
   душу...");
9      }
10 }
```

Использование абстрактных классов и методов позволяет описать некую абстракцию, которая должна быть реализована в других классах. Например, мы можем создать абстрактный класс Fighter и объявить в нём абстрактный метод fight(). Т.к. стилей борьбы может быть много, то, например, для JudoFighter extends Fighter метод fight() будет описывать приемы в стиле дзюдо и т.д.

25. Можно ли объявить метод абстрактным и статическим одновременно?

Нет. Получите: Illegal combination of modifiers: 'abstract' and 'static'. Модификатор abstract говорит, что метод будет реализован в другом классе, а static наоборот указывает, что этот метод будет доступен по имени класса.

26. Что означает ключевое слово static?

Модификатор static говорит о том, что метод или поле класса принадлежат не объекту, а классу. Т.е. доступ можно будет получить и не создавая объекта класса. Поля помеченные static инициализируются при инициализации класса. К примеру,

Class.forName(«MyClass», true, currentClassLoader), где второй параметр указывает на необходимость проведения инициализации.

На методы, объявленные как static, накладывается ряд ограничений.

- **Они могут вызывать только другие статические методы.**
- **Они должны осуществлять доступ только к статическим переменным.**
- **Они не могут ссылаться на члены типа this или super.**

27. К каким конструкциям Java применим модификатор static?

- **К методу.**
- **К внутреннему классу.**
- **К полю.**
- **К импортируемым классам (с 5-ой java). Например, import static org.junit.Assert.assertThat;**

28. Что будет, если в static блоке кода возникнет исключительная ситуация?

Если в явном виде написать любое исключение в static-блоке, то компилятор не скомпилирует исходники. Это все от того, что компилятор умный. В остальном, взаимодействие с исключениями такое же как и в любом другом месте. Если unchecked исключение вывалится в static-блоке, то класс не будет инициализирован.

Какое исключение выбрасывается при ошибке в блоке инициализации?

Для static:

- **java.lang.ExceptionInInitializerError — если исключение наследуется от RuntimeException.**

Для init:

- **exception, который и вызвал исключение, если он наследуется от**

RuntimeException.

Верно для static и init:

- **java.lang.Error — если исключение вызвано Error.**
- **java.lang.ThreadDeath — смерть потока. Ничего не вываливается.**

29. Можно ли перегрузить static метод?

Перегрузить можно, но переопределить нельзя.

```
1     public Animal eat() {
2         System.out.println("animal eat");
3         return null;
4     }
5     public static Animal eat(String s) {
6         System.out.println("test static overload");
7         return null;
8     }
```

30. Что такое статический класс, какие особенности его использования?

Это вложенный класс, который может обращаться только к статическим полям обертывающего его класса, в том числе и приватным. Доступ к нестатическим полям обрамляющего класса может быть осуществлен только через ссылку на экземпляр обрамляющего объекта. К классу высшего уровня модификатор static неприменим.

В примере показано, что для инициализации внутреннего статического класса нет нужды в инициализации родителя. Но в случае обычного внутреннего класса такой номер не пройдет:

```
1  public class Test {
2    class A { }
3    static class B { }
4    public static void main(String[] args) {
5      /*will fail - compilation error, you need an instance of Test
   to instantiate A*/
6      A a = new A();
7      /*will compile successfully, no instance of Test is needed to
8  instantiate B */
9      B b = new B();
10   }
   }
```

Статические вложенные классы, не имеют доступа к нестатическим полям и методам обрамляющего класса, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. В этом плане static nested классы очень похожи на любые другие классы верхнего уровня.

31. Какие особенности инициализации final static переменных?

Переменные должны быть инициализированы во время объявления или в static блоке.

32. Как влияет модификатор static на класс/метод/поле?

Модификатор static говорит о том, что метод или поле класса принадлежат не объекту, а классу.

Внутри static метода нельзя вызвать не статический метод по имени класса.

Про static класс смотрите ответ выше.

33. О чем говорит ключевое слово final?

Может быть применено к полям, методам или классам. В зависимости к какой сущности приложено данное ключевое слово — будет и различный смысл в его применении.

- **Для класса. Класс помеченный при помощи final не может иметь наследников.**
- **Для метода. Метод помеченный при помощи final не может быть переопределен в классах наследниках.**
- **Для поля. Поле помеченное при помощи слова final не может изменить свое значение после инициализации (инициализируется либо при описании, либо в конструкторе, статическом или динамическом блоке).**
- **Значение локальных переменных, а так же параметров метода помеченных при помощи слова final не могут быть изменены после присвоения.**

34. Дайте определение понятию “интерфейс”.

Ключевое слово interface используется для создания полностью абстрактных классов. Создатель интерфейса определяет имена методов, списки аргументов и типы возвращаемых значений, но не тела методов.

Наличие слова interface означает, что именно так должны выглядеть все классы, которые реализуют данный интерфейс. Таким образом, любой код, использующий конкретный интерфейс, знает только то, какие методы вызываются для этого интерфейса, но не более того.

```
1  public interface SomeName{

2      void method();
```

```
3       int getSum();
4   }
```

35. Какие модификаторы по умолчанию имеют поля и методы интерфейсов?

Интерфейс может содержать поля, но они автоматически являются статическими (static) и неизменными (final). Все методы и переменные неявно объявляются как public.

36. Почему нельзя объявить метод интерфейса с модификатором final или static?

Вообще с 8й версии можно static, но нужно чтобы было тело метода. Например

```
1   public interface Shape {
2       static void draw() {
3           System.out.println("Wow! It is impossible!");
4       };
5   }
```

final модификатор просто бессмысленный. Все методы по умолчанию абстрактные, т.е. их невозможно создать не реализовав где-то еще, но это нельзя будет сделать, если у метода идентификатор final.

37. Какие типы классов бывают в java (вложенные… и.т.д.)

38. Какие особенности создания вложенных классов: простых и статических.

- **Обычные классы (`Top level classes`)**
- **Интерфейсы (`Interfaces`)**
- **Перечисления (`Enum`)**
- **Статические вложенные классы (`Static nested classes`)**
    - **Есть возможность обращения к внутренним статическим полям и методам класса обертки.**

    - **Внутренние статические классы могут содержать только статические методы.**
- **Внутренние классы-члены (`Member inner classes`)**
    - **Есть возможность обращения к внутренним полям и методам класса обертки.**
    - **Не может иметь статических объявлений.**
    - **Нельзя объявить таким образом интерфейс. А если его объявить без идентификатора `static`, то он автоматически будет добавлен.**
    - **Внутри такого класса нельзя объявить перечисления.**
    - **Если нужно явно получить `this` внешнего класса — `OuterClass.this`**
- **Локальный класс (`Local inner classes`)**
    - **Видны только в пределах блока, в котором объявлены.**
    - **Не могут быть объявлены как `private`/`public`/`protected` или `static` (по этой причине интерфейсы нельзя объявить локально).**
    - **Не могут иметь внутри себя статических объявлений (полей, методов, классов).**
    - **Имеют доступ к полям и методам обрамляющего класса.**
    - **Можно обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором `final`.**
- **Анонимные классы (`Anonymous inner classes`)**
    - **Локальный класс без имени.**

39. Что вы знаете о вложенных классах, зачем они используются? Классификация, варианты использования, о нарушении инкапсуляции.

40. В чем разница вложенных и внутренних классов?

41. Какие классы называются анонимными?

Вложенный класс — это класс, который объявлен внутри объявления другого класса. Вложенные классы делятся на статические и нестатические (non-static). Собственно нестатические вложенные классы имеют и другое название — внутренние классы (inner classes).

Внутренние классы в Java делятся на такие три вида:

- **внутренние классы-члены (member inner classes);**
- **локальные классы (local classes);**
- **анонимные классы (anonymous classes).**

Внутренние классы-члены ассоциируются не с самим внешним классом, а с его экземпляром. При этом они имеют доступ ко всем его полям и методам.

Локальные классы (local classes) определяются в блоке Java кода. На практике чаще всего объявление происходит в методе некоторого другого класса. Хотя объявлять локальный класс можно внутри статических и нестатических блоков инициализации.

Анонимный класс (anonymous class) — это локальный класс без имени.

Использование вложенных классов всегда приводит к некоторому нарушению инкапсуляции — вложенный класс может обращаться к закрытым членам внешнего класса (но не наоборот!). Если это обстоятельство учитывается в архитектуре вашего приложения, не стоит уделять ему особого внимания, поскольку внутренний класс всего лишь является специализированным членом внешнего класса.

Подробнее http://www.quizful.net/post/inner-classes-java

42. Каким образом из вложенного класса получить доступ к полю внешнего класса?

Если вложенный класс не статический и поле не статическое, то можно просто обратиться к этому полю из внутреннего класса, если только у внутреннего класса не существует поля с таким же литералом, в этом случае нужно обращаться через ссылку на внешний класс так — OuterClass.this.имяПоля

43. Каким образом можно обратиться к локальной переменной метода из анонимного класса, объявленного в теле этого метода? Есть ли какие-нибудь ограничения для такой переменной?

Также как и локальные классы, анонимные могут захватывать переменные, доступ к

локальным переменным происходит по тем же правилам:

- Анонимный класс имеет доступ к полям внешнего класса.
- Анонимный класс не имеет доступ к локальным переменным области, в которой он определен, если они не финальные (final) или неизменяемые (effectively final).
- Как и у других внутренних классов, объявление переменной с именем, которое уже занято, затеняет предыдущее объявление.
- Вы не можете определять статические члены анонимного класса.

Анонимные классы также могут содержать в себе локальные классы. Конструктора в анонимном классе быть не может.

```
1  public class Animal {
2
3      Integer classAreaVar2 = 25;
4
5      public void anonymousClassTest() {
6          final Integer[] localAreaVar = {25};
7          //Анонимный класс
8          ActionListener listener = new ActionListener() {
9              @Override
1              public void actionPerformed(ActionEvent e) {
0                  //можно использовать переменные класса без
1  указания final
1                  classAreaVar2 = classAreaVar2 + 25;
1
```

```
                    //нельзя использовать локальные переменные, если
они не final;

                     /*Local variable is accessed from within inner
class: needs to be declared final */

                    localAreaVar[0] = localAreaVar[0] +5;

                }

            };

        }

}
```

44. Как связан любой пользовательский класс с классом Object?

Все классы являются наследниками суперкласса Object. Это не нужно указывать явно. В результате объект Object может ссылаться на объект любого другого класса.

45. Расскажите про каждый из методов класса Object.

- **public final native Class getClass() — возвращает в рантайме класс данного объекта.**
- **public native int hashCode() — возвращает хеш-код**
- **public boolean equals(Object obj) — сравнивает объекты.**
- **protected native Object clone() throws CloneNotSupportedException — клонирование**

объекта

- **public String toString() — возвращает строковое представление объекта.**
- **public final native void notify() — просыпается один поток, который ждет на “мониторе” данного объекта.**
- **public final native void notifyAll() — просыпаются все потоки, которые ждут на “мониторе” данного объекта.**
- **public final native void wait(long timeout) throws InterruptedException — поток переходит в режим ожидания в течение указанного времени.**
- **public final void wait() throws InterruptedException — приводит данный поток в ожидание, пока другой поток не вызовет notify() или notifyAll() методы для этого объекта.**
- **public final void wait(long timeout, int nanos) throws InterruptedException — приводит данный поток в ожидание, пока другой поток не вызовет notify() или notifyAll() для этого метода, или пока не истечет указанный промежуток времени.**
- **protected void finalize() throws Throwable — вызывается сборщиком мусора, когда garbage collector определил, что ссылок на объект больше нет.**

46. Что такое метод equals(). Чем он отличается от операции ==.

47. Если вы хотите переопределить equals(), какие условия должны удовлетворяться для переопределенного метода?

48. Если equals() переопределен, есть ли какие-либо другие методы, которые следует переопределить?

49. В чем особенность работы методов hashCode и equals? Каким образом реализованы методы hashCode и equals в классе Object? Какие правила и соглашения существуют для реализации этих методов? Когда они применяются?

Это метод, определенный в Object, который служит для сравнения объектов. При сравнении объектов при помощи == идет сравнение по ссылкам. При сравнении по equals() идет сравнение по состояниям объектов (реализация метода equals для нового созданного класса ложится на плечи разработчиков). С точки зрения математики equals() обозначает отношение эквивалентности объектов. Эквивалентным называется отношение, которое является симметричным, транзитивным и

рефлексивным.

- **Рефлексивность: для любого ненулевого x, x.equals(x) вернет true;**
- **Транзитивность: для любого ненулевого x, y и z, если x.equals(y) и y.equals(z) вернет true, тогда и x.equals(z) вернет true;**
- **Симметричность: для любого ненулевого x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) вернет true.**
- **Также для любого ненулевого x, x.equals(null) должно вернуть false**

При переопределении equals() обязательно нужно переопределить метод hashCode(). Равные объекты должны возвращать одинаковые хэш коды.

Хеш-код — это число. Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины. В терминах Java, хеш-код — это целочисленный результат работы метода, которому в качестве входного параметра передан объект.

Этот метод 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.

Здесь главное понять, 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.

Статья с хабра по equals и hashCode: http://habrahabr.ru/post/168195/

50. Какой метод возвращает строковое представление объекта?

someObject.toString();

51. Что будет, если переопределить equals не переопределяя hashCode? Какие могут возникнуть проблемы?

Нарушится контракт. Классы и методы, которые использовали правила этого контракта могут некорректно работать. Так для объекта HashMap это может привести к тому, что пара, которая была помещена в Map возможно не будет найдена в ней при обращении к Map, если используется новый экземпляр ключа.

52. Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode?

Те, которые используют при определении метода `equals()`. Хэш код должен быть равномерно распределен на области возможных принимаемых значений.

53. Как вы думаете, будут ли какие-то проблемы, если у объекта, который используется в качестве ключа в hashMap изменится поле, которое участвует в определении hashCode?

Будут. При обращении по ключу мы можем не найти значение.

54. Чем отличается абстрактный класс от интерфейса, в каких случаях что вы будете использовать?

Синтаксические отличия интерфейса от абстрактного класса? Таблица 1

		Абстрактные классы abstract	Интерфейсы interface
1	Возможность содержать неопределенные переменные	Может содержать и пустые и заполненные переменные	Только явно определенные КОНСТАНТЫ, static, final

2	Модификаторы доступа	Любой модификатор доступа	public по умолчанию
3	Реализация методов	Может содержать готовые и абстрактные методы, обязательно реализуемые в классе наследнике, имеет конструктор	Может содержать только пустые методы, которые обязательно реализовывать в реализующем классе (либо методы default), не имеет конструктора
4	Наследование и реализация	Наследуется только один абстрактный класс	Можно реализовать множество интерфейсов, с возможностью использования в них полей с одинаковым именем переменных
5	Возможность наследоваться	Может наследовать другой класс и реализовывать интерфейс (implement interface, extend class)	Интерфейс может расширять другой интерфейс, но не может наследовать класс (extend interface) для имплементации класса слово - implements
6	Использования полей	Поля могут использоваться только после создания экземпляра класса наследника	Поля могут использоваться в реализуемых классах без создания экземпляра, т.к. все поля интерфейса - Константы

Абстрактные классы используются только тогда, когда есть «is a» тип отношений; интерфейсы могут быть реализованы классами которые не связаны друг с другом.

Абстрактный класс может реализовывать методы; интерфейс может реализовывать статические методы начиная с 8й версии.

Интерфейс может описывать константы и методы. Все методы интерфейса по умолчанию являются публичными (public) и абстрактными (abstract), а поля — public static final. С java 8 в интерфейсах можно реализовывать default и статические методы.

В Java класс может наследоваться (реализовывать) от многих интерфейсов, но только от одного абстрактного класса.

С абстрактными классами вы теряете индивидуальность класса, наследующего его; с интерфейсами вы просто расширяете функциональность каждого класса.

интерфейс может реализовывать статические методы и начиная с 8й версии.

+ default методы с 8 версии

55. Можно ли получить доступ к private переменным класса и если да, то каким образом?

```
public class SomeClass {

    private String name = "SomeNameString";
    private Integer x = 25;

}

public class TestPrivateAccess {

    public static void main(String[] args) {
        SomeClass someClass = new SomeClass();

        try {
                                        Field reflectField =
SomeClass.class.getDeclaredField("name"); //NoSuchFieldException e
                                       Field reflectField2 =
SomeClass.class.getDeclaredField("x"); //NoSuchFieldException e

            /* Если не дать доступ, то будет ошибка
                        java.lang.IllegalAccessException: Class ..
.TestPrivateAccess
                can not access a member of class .. .SomeClass with
```

```
modifiers "private"
            */
            reflectField.setAccessible(true);
            reflectField2.setAccessible(true);

                                    String   fieldValue   =   (String)
reflectField.get(someClass); //IllegalAccessException ex

                                  Integer   fieldValue2   =   (Integer)
reflectField2.get(someClass); //IllegalAccessException ex

                             System.out.println(reflectField);//private
java.lang.String
ru.javastudy.interview.oop.privateFieldAccess.SomeClass.name
            System.out.println(fieldValue); //SomeNameString

                            System.out.println(reflectField2);//private
java.lang.Integer
ru.javastudy.interview.oop.privateFieldAccess.SomeClass.x
            System.out.println(fieldValue2); //25
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();;
        }
```

```
30          }
31      }
32
33
34
35
36
37
38
```

56. Что такое volatile и transient? Для чего и в каких случаях можно было бы использовать default?

volatile — не используется кэш (имеется ввиду область памяти в которой JVM может сохранять локальную копию переменной, чтобы уменьшить время обращения к переменной) при обращении к полю. Для volatile переменной JVM гарантирует синхронизацию для операций чтения/записи, но не гарантирует для операций изменения значения переменной.

transient — указание того, что при сериализации/десериализации данное поле не

нужно сериализовать/десериализовывать.

57. Расширение модификаторов при наследовании, переопределении и сокрытии методов. Если у класса-родителя есть метод, объявленный как private, может ли наследник расширить его видимость? А если protected? А сузить видимость?

Действует общий принцип: расширять видимость можно, сужать нельзя. private методы видны только внутри класса, для потомков они не видны. Поэтому их и расширить нельзя.

58. Имеет ли смысл объявлять метод private final?

Нет, такой метод и так не виден для наследников, а значит не может быть ими переопределен.

59. Какие особенности инициализации final переменных?

- **Для поля. Поле помеченное при помощи слова final не может изменить свое значение после инициализации.**
  Не статическое final поле можно инициализировать: при описании, в конструкторе (во всех), в статическом блоке, в динамическом блоке.
  Статическое final поле (static final) инициализируется либо в статическом блоке, либо при описании.
- **Значение локальных переменных, а так же параметров метода помеченных при помощи слова final не могут быть изменены после присвоения.**

60. Что будет, если единственный конструктор класса объявлен как final?

К конструктору не применимо ключевое слово final.

61. Что такое finalize? Зачем он нужен? Что Вы можете рассказать о сборщике мусора и алгоритмах его работы.

Метод finalize() вызывается перед тем, как объект будет удален garbage collector (сборщик мусора, далее gc). Существует много различных реализаций gc. Основа работы следующая: gc помечает объекты, на которые больше не ссылаются другие

объекты для их удаления. Затем на одном из проходов помеченные объекты удаляются.

Вызов finalize() не гарантируется, т.к. приложение может быть завершено до того, как будет запущена ещё одна сборка мусора. Да, можно отменить сборку объекта с помощью метода finalize(), присвоив его ссылку какому-то статическому методу.

62. Почему метод clone объявлен как protected? Что необходимо для реализации клонирования?

Это указывает на то, что хоть метод и есть в классе Object и разработчик желает им воспользоваться, то его нужно переопределить. Для этого нужно реализовать интерфейс Cloneable, чтобы соблюсти контракт.