1. Инкапсуляция.

Инкапсуляция - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя. Иными словами, инкапсуляция - это договор для объекта, что он должен скрыть, а что открыть для доступа другим объектам. Это свойство позволяет защитить важные данные для компонента, кроме того, оно дает возможность пользователю не задумываться о сложности реализации используемого компонента, а взаимодействовать с ним посредством предоставляемого интерфейса - публичных данных и методов.

Данный механизм реализуется с помощью модификаторов доступа private, package-private, protected, public:

- public - доступ к компоненту из экземпляра любого класса и любого пакета;
- protected - доступ к компоненту из экземпляра родного класса или классов-потомков, а также внутри пакета;
- package-private - доступ к компоненту только внутри пакета;
- private - доступ к компоненту только внутри класса.

```
public class Car {
    private double engineLiters = 1.5;
    private String color = "orange";

    public void changeColor(String newColor) {
        color = newColor;
    }

    private void disassembleEngine() {
        System.out.println("Engine is disassembled");
    }
}
```

Пользователь класса Car может перекрасить автомобиль, но разобрать двигатель он не может - эта функциональность от него скрыта.

Плюсы:

- Полный контроль над данными класса
- Достижение модульного построения кода
- Упрощение поддержки программы

Простой пример из жизни - современный автомобиль. Водителю совсем не обязательно знать, какие процессы происходят в двигателе при движении, коробке передач при переключении скоростей, и в рулевой тяге при повороте руля. Более того, владельцам автомобиля с коробкой автомат даже не нужно переключать передачи! То есть, все сложное устройство (реализация) системы "автомобиль" инкапсулировано в интерфейс, состоящий из двух педалей, руля и регулятора громкости на магнитоле. А этим простым интерфейсом может пользоваться любой человек, не боясь сломать сложную внутреннюю систему.

2. Методы класса String.

1. char charAt(int index)
   возвращает символ по указанному индексу
2. int compareTo(Object o)
   сравнивает данную строку с другим объектом
3. int compareToIgnoreCase(String str)
   сравнивает две строки лексически, игнорируя регистр символов
4. String concat(String str)
   объединяет указанную строку с данной строкой, путем добавления ее в конец
5. boolean contentEquals(StringBuffer sb)
   возвращает true, если эта строка представляет собой ту же последовательность символов, которая указана в StringBuffer

6. static String copyValueOf(char[] data)
   возвращает строку, которая представляет собой последовательность символов в заданном массиве
7. boolean endsWith(String suffix)
   проверяет, заканчивается ли эта строка указанным окончанием
8. boolean startsWith(String prefix)
   проверяет, начинается ли эта строка с заданного префикса
9. boolean equals(Object anObject)
   сравнивает данную строку с указанным объектом
10. boolean equalsIgnoreCase(String anotherString)
    сравнивает данную строку с другой строкой, игнорируя регистр символов
11. byte[] getBytes(String charsetName)
    кодирует эту строку в последовательность байтов, сохраняя результат в новый массив байтов
12. void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
    копирует символы из строки в массив символов назначения
13. char[] toCharArray()
    преобразует эту строку в новый массив символов
14. int hashCode()
    возвращает хэш-код строки
15. int indexOf(int ch)
    возвращает индекс первого вхождения указанного символа в строке
16. int indexOf(String str)
    возвращает индекс первого вхождения указанной подстроки в данной строке
17. int lastIndexOf(int ch)
    возвращает индекс последнего вхождения указанного символа в строке
18. int lastIndexOf(String str)
    возвращает индекс последнего вхождения указанной подстроки в данной строке
19. String intern()
    возвращает каноническое представление для строкового объекта
20. int length()
    возвращает длину строки
21. boolean matches(String regex)
    сообщает, соответствует ли эта строка заданному регулярному выражению
22. String replace(char oldChar, char newChar)
    возвращает новую строку после замены всех вхождения oldChar на newChar
23. String replaceAll(String regex, String replacement)
    заменяет каждую подстроку, соответствующую заданному регулярному выражению, указанной строкой
24. String replaceFirst(String regex, String replacement)
    заменяет первые подстроки, соответствующие заданному регулярному выражению, указанной строкой
25. String[] split(String regex)
    разделяет строку по регулярному выражению
26. CharSequence subSequence(int beginIndex, int endIndex)
    возвращает новую последовательность символов, которая является подстрокой строки
27. String substring(int beginIndex, int endIndex)
    возвращает новую строку, которая является подстрокой строки
28. String toLowerCase()
    преобразует все символы в строке в нижний регистр
29. String toUpperCase()
    преобразует все символы в строке в верхний регистр
30. String toString()
    строка возвращает себя
31. String trim()
    возвращает копию строки, обрезая начальные и конечные пробелы
32. String valueOf(primitive data type x)
    возвращает строковое представление переданного аргумента

3. Множественные catch.

Исключительные ситуации (исключения) возникают во время выполнения программы, когда появившаяся проблема не может быть решена в текущем контексте. При возникновении исключения в приложении создается объект, описывающий это исключение, текущий ход выполнения приложения останавливается, и включается механизм обработки исключений: управление передается соответствующему блоку catch, в котором он обрабатывается. Если в блоке try может быть сгенерировано в разных участках кода несколько типов исключений, то необходимо наличие нескольких блоков catch, если только блок catch не обрабатывает все типы исключений. Подклассы исключений в блоках catch должны следовать перед любым из их суперклассов, иначе суперкласс будет перехватывать эти исключения:

```
try {
    FileInputStream fileInputStream = new FileInputStream(fileName);
} catch (FileNotFoundException e) {
    System.err.println("fnfe");
} catch (IOException e) {
    System.err.println("ioe");
}
```

Если файл fileName не будет найден, то будет сгенерирован FileNotFoundException и выведено на консоль "fnfe". Если же файл будет найден, но, например, не доступен для записи, то будет сгенерирован IOException и не консоль будет выведено "ioe".

Бывают ситуации, когда в процессе выполнения программы могут возникнуть исключения разного типа, но их обработка ничем не отличается друг от друга:

```
try {
    //some code
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

В Java 7 появилась возможность объединить такие исключения в одну конструкцию, чтобы не дублировать один и тот же код:

```
try {
    //some code
} catch (ClassNotFoundException | NoSuchMethodException | FileNotFoundException e) {
    e.printStackTrace();
}
```

4. Что такое синхронизация, понятие монитора.

Когда несколько потоков нуждаются в доступе к разделяемому ресурсу, им необходим некоторый способ гарантии того, что ресурс будет использоваться одновременно только одним потоком. Процесс, с помощью которого это достигается, называется синхронизацией. Синхронизация в Java гарантирует, что никакие два потока не смогут выполнить синхронизированный метод одновременно или параллельно. Базовая синхронизация в Java возможна при использовании синхронизированных методов и синхронизированных блоков с использованием ключевого слова synchronized. Когда какой либо поток входит в синхронизированный метод или блок, он приобретает блокировку, и всякий раз, когда поток выходит из синхронизированного метода или блока, JVM снимает блокировку. Если синхронизированный метод вызывает другой синхронизированный метод, который требует такой же замок, то текущий поток, который держит замок может войти в этот метод не приобретая замок.

Ключом к синхронизации является концепция монитора (также называемая семафором). Монитор — это объект, который используется для взаимоисключающей блокировки, его также называют mutex. Только один поток может захватить и держать

монитор в заданный момент. Когда поток получает блокировку, говорят, что он вошел в монитор. Все другие потоки пытающиеся войти блокированный монитор, будут приостановлены, пока первый не вышел из монитора. Говорят, что другие потоки ожидают монитор. При желании поток, владеющий монитором, может повторно захватить тот же самый монитор. В Java каждый объект имеет свой собственный монитор. Статический метод захватывает монитор экземпляра класса Class, того класса, на котором он вызван. Он существует в единственном экземпляре. Нестатический метод захватывает монитор экземпляра класса, на котором он вызван.

---

5. Статические методы, особенности работы со статическими методами.

Статические методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя this на конкретный экземпляр, вызвавший метод. Статические методы реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции. По причине недоступности указателя this статические методы не могут обращаться к нестатическим полям и методам напрямую, так как они не «знают», к какому объекту относятся, да и сам экземпляр класса может быть не создан.

Для объявления статических методов перед их объявлением используется ключевое слово static. Вызов статического метода всегда следует осуществлять с помощью указания на имя класса, а не объекта. Переопределение статических методов невозможно, так как статические методы связываются во время компиляции, а не в runtime, то есть полиморфизм на статические методы не распространяется.

Статические методы можно импортировать при помощи ключевых слов import static, после чего к статическим методам можно будет обращаться без указания имени класса:

```
import static java.lang.System.out;
public class StaticImport {
	public static void main(String[] args) {
		out.println("hi");
	}
}
```

Статические методы следует использовать, когда не нужен доступ к состоянию объекта, а все параметры задаются явно, или когда методу нужен доступ только к статическим полям класса.

---

6. Merge Sort, принцип и краткое описание алгоритма

Принцип

Основная идея заключается в том, чтобы разбить массив на максимально малые части, которые могут считаться отсортированными (массивы с одним элементом), а потом сливать их между собой в порядке, необходимом для сортировки.

Алгоритм

1. Если в массиве меньше 2 элементов, то он уже отсортирован, алгоритм завершает свою работу;
2. Массив разбивается на 2 части (примерно пополам), для которых рекурсивно вызывается тот же алгоритм сортировки;
3. После сортировки двух частей массива производится слияние двух упорядоченных массивов в один упорядоченный массив;

Процедура слияния:

1. объявляются счетчики индексов для результирующего массива и для двух сливаемых частей, счетчики инициализируются 0;
2. в цикле с условием (пока не дошли до конца какого-либо из сливаемых массивов) на каждом шаге берется меньший из двух первых (по индексу счетчиков) элементов

сливаемых массивов и записывается в результирующий массив. Счетчик индекса результирующего массива и счетчик массива, из которого был взят элемент, увеличиваются на 1.

3. Когда мы вышли из цикла пункта 2, т.е. мы дошли до конца какого-то из сливаемых массивов, мы добавляем все оставшиеся элементы второго сливаемого массива в результирующий массив.

Свойства

- время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n*log n);
- затраты памяти – O(n);
- устойчивая;
- количество обменов – обычно O(n*log n) обменов;

Плюсы

- проста для понимания;
- независимо от входных данных стабильное время работы;
- устойчивая;

Минусы

- нужно O(n) дополнительной памяти;
- в среднем на практике несколько уступает в скорости Quick Sort;

---

4. Entry point в Java классе: назначение, структура (точка входа, метод main).

Каждому приложению требуется точка входа, чтобы Java знала, откуда начать выполнение кода. В Java такой точкой входа является метод main(), который имеет следующую сигнатуру:

public static void main(String[] args)

public static void main(String... args)

По крайней мере один класс в программе должен иметь метод main(). В программе может быть несколько точек входа, но начинается выполнение кода всегда с какой-то одной. Аргументы командной строки передаются в метод **main()** в массив строк **args** (называться он может иначе, это не принципиально), эти аргументы могут быть каким-то образом использованы в методе **main()**.

В методе **main()** обычно описывают самый основной алгоритм работы, например, создание каких-то объектов и вызов их методов, но конкретная программная реализация подсчетов и прочего обычно находится вне этого метода, скорее этот метод лучше использовать для консолидации функциональности других классов, которые созданы для решения определенных задач.