1. Что такое объектно-ориентированный подход.

Объектно-ориентированный подход - это такой способ программирование, который напоминает процесс человеческого мышления. ООП более структурировано, чем другие способы программирования (например, процедурное) и позволяет создавать модульные программы с представление данных на определенном уровне абстракции. Основная цель ООП - это повышение эффективности разработки программ.

Весь окружающий мир состоит из объектов, которые представляются как единое целое, и такие объекты взаимодействуют друг с другом. Базом в ООП является понятие объекта, который имеет определенные свойства. Каждый объект знает как решать определенные задачи, то есть располагает методами решения. Программа, написанная с использованием ООП, состоит из объектов, которые могут взаимодействовать друг с другом. Все объекты с одинаковыми наборами атрибутов принадлежат к одному классу. Объединение объектов в классы определяется семантикой, то есть смыслом. Каждый класс имеет свои особенности поведения, которые определяют этот класс. Один класс отличается от другого именем и набором "сообщений", которые можно посылать объектам данных классов (интерфейсом).

Характеристики ООП:

- все является объектом
- объекты взаимодействуют между собой путем обмена сообщениями, при котором один объект требует, чтобы другой объект выполнил некоторое действие.
- каждый объект является представителем класса, который выражает общие свойства объектов.
- в классе задается функциональность (поведение) объекта, а все объекты одного класса могут выполнять одни и те же действия.
- классы организованы в иерархическую структуру.

ООП основывается на 4х принципах:

Абстракция - выделение некоторых существенных характеристик объекта, которые выделяют его из всех других объектов. Абстракция позволяет отделить существенные особенности поведения объекта от деталей их реализации. Например, главной характеристикой объекта "директор" будет то, что этот объект чем-то управляет, а чем и как именно (финансами или персоналом), то есть детали реализации, являются второстепенной информацией.

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.

Наследование - это свойство системы, позволяющее описать новый класс на основе уже существующего класса, при этом функциональность может быть заимствована частично или полностью. Класс, от которого производится наследование, называется базовым или суперклассом, а новый класс называется потомком, дочерним или производным классом. Производный класс **расширяет** функциональность базового класса благодаря добавлению новой функциональности, специфической только для производного класса.

Полиморфизм - это свойство системы, позволяющее использовать один и тот же интерфейс для общего класса действий. При этом каждое действие зависит от конкретного объекта. То есть полиморфизм - это способность выбирать более конкретные методы для исполнения в зависимости от типа объекта.

При обсуждении принципов ООП следует упомянуть 5 принципов SOLID:

1. Единственность ответственности (Single Responsibility)
2. Принцип открытости и закрытости (Open/Close Principle)
3. Принцип замещения Лисков (The Liskov Substitution Principle)
4. Принцип разделения интерфейса (The Interface Segregation Principle)
5. Инверсия зависимости (The Dependency Inversion Principle)

2. Перечислите методы класса Collections. Укажите назначение этих методов.

Collections - это класс состоящий из статических методов, осуществляющих различные служебные операции над коллекциями. К ним можно обращаться так:

Collections.method(Collection);

Для работы с любой коллекцией:

frequency(Collection, Object)	возвращает кол-во вхождений элемента в коллекции
disjoint(Collection, Collection)	true, если в коллекциях нет общих элементов
addAll(Collection, T[])	добавляет все элементы массива в коллекцию
min(Collection)	минимальный элемент коллекции
max(Collection)	максимальный элемент коллекции

Для работы со списками:

fill(List, Object)	заменяет каждый элемент в списке значением
sort(List)	сортирует слиянием за O(n*log n)
reverse(List)	изменяет порядок всех элементов
rotate(List, int)	передвигает все элементы на заданный диапазон
binarySearch(List, Object)	ищет элемент (бинарный поиск)
shuffle(List)	перемешивает элементы в случайном порядке
copy(List dest, List src)	копирует один список в другой
replaceAll(List, Object old, Object new)	заменяет все вхождения одного значения на другое
swap(List, int, int)	меняет местами элементы на указанных позициях
indexOfSubList(List src, List trg)	индекс первого вхождения списка trg в список src
lastIndexOfSubList(List src, List trg)	индекс последнего вхождения списка trg в список src

Создание копий коллекции:

newSetFromMap(Map)	создает Set из Map
unmodifiableCollection(Collection)	создает неизменяемую копию коллекции
synchronizedCollection(Collection)	создает потокобезопасную копию коллекции
asLifoQueue(Deque)	создает очередь last in first out из Deque
<T> Set<T> singleton(T o)	создает неизменяемый Set с заданным объектом (только с ним)

3. Повторный выброс исключения.

Программа может сама явно выбрасывать исключения, используя оператор throw. Это называется «ручным» выбросом исключения:

throw ThrowableInstance;

Выбрасывать можно только объектом типа Throwable или подкласса Throwable. Получить объект Throwable можно двумя способами: использовать объект, полученный в блоке catch или создать Throwable объект с помощью оператора new.

При достижении оператора throw выполнение кода прекращается, блок try/catch проверяется на наличие соответствующего обработчика catch. Если он существует, то управление передается в этот блок catch.

Если метод может породить исключение, которое сам не обрабатывает, он должен передать это исключение вызывающему его методу, чтобы в нем оно было обработано или передано дальше. Передача исключений вызывающему методу в Java обеспечивается добавлением ключевого слова throws в заголовок объявления метода. После ключевого слова throws должны быть перечислены через запятую типы исключений, которые метод может выбросить, кроме исключений RuntimeException и ошибок Error.

Перехваченное исключение может быть выброшено снова. Такая ситуация и называется повторным выбросом исключения. Например, блок catch отлавливает исключения типа IOException и выбрасывает их вновь.

```
try{
    // some operations
} catch(IOException e) {
```

```
        throw e;
    }
```

В Java 7 была введена возможность передавать «вверх» по стеку вызова исключение более точного типа, если данные типы указаны при объявлении метода.

```
public static void ioMethod() throws IOException {
    try{
        throw new IOException ("this is IOException ");
    } catch(Exception e) {
        throw e;
    }
}
```

---

4. Опишите жизненный цикл потока.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State:
NEW — поток создан, но еще не запущен;
RUNNABLE — поток выполняется;
BLOCKED — поток блокирован;
WAITING — поток ждет окончания работы другого потока;
TIMED_WAITING — поток некоторое время ждет окончания другого потока;
TERMINATED — поток завершен.
При создании потока он получает состояние «новый» (**NEW**) и не выполняется.

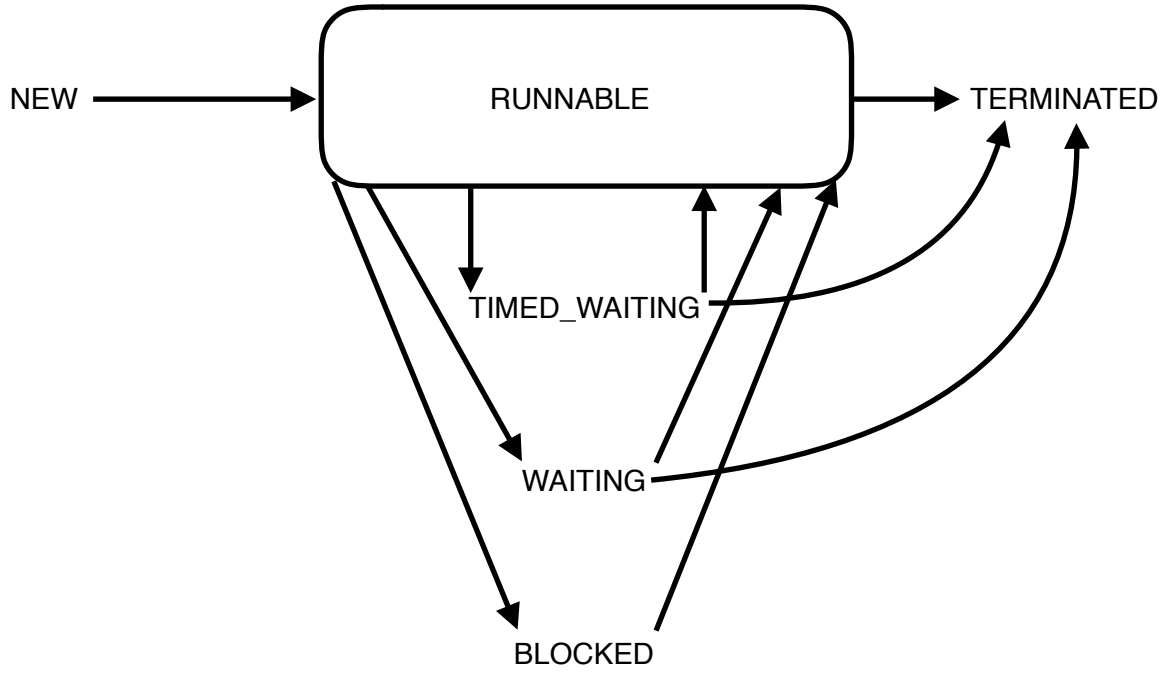
Для перевода потока из состояния «новый» в состояние «работоспособный» (**RUNNABLE**) следует выполнить метод start(), который вызывает метод run() — основной метод потока.

Поток переходит в состояние «неработоспособный» в режиме ожидания (**WAITING**) вызовом методов join(), wait(), suspend() (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания по времени (**TIMED_WAITING**) с помощью методов yield(), sleep(long millis), join(long timeout) и wait(long timeout), при выполнении которых может генерироваться прерывание InterruptedException. Вернуть потоку работоспособность после вызова метода suspend() можно методом resume() (deprecated метод), а после вызова метода wait() — методами notify() или notifyAll().

Поток переходит в «пассивное» состояние (**TERMINATED**), если вызваны методы interrupt(), stop() (deprecated-метод) или метод run() завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод interrupt() успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток неработоспособен, например, находится в состоянии TIMED_WAITING, то метод инициирует исключение InterruptedException. Чтобы это не происходило, следует предварительно вызвать метод isInterrupted(), который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

Методы suspend(), resume() и stop() являются deprecated-методами и запрещены к использованию, так как они не являются в полной мере «потокобезопасными».

Получить текущее значение состояния потока можно вызовом метода getState().



5. Интерфейсы: определение, методы и поля интерфейсов.

Назначение интерфейса — описание или спецификация функциональности, которую должен реализовывать каждый класс, его имплементирующий. Класс, реализующий интерфейс, предоставляет к использованию объявленный интерфейс в виде набора public-методов в полном объеме. Интерфейсы подобны полностью абстрактным классам, хотя и не являются классами.

В интерфейсе не могут быть объявлены поля без инициализации, все поля неявно public final static. Внутри интерфейса не может быть реализован ни один из объявленных методов. Все объявленные в интерфейсе методы неявно public и abstract.

```
public interface BankAction {
	int startCash = 0;
	void openAccount();
	void blockAccount();
	void closeAccount();
}

public class TinkoffBankAction implements BankAction {
	private void openAccount() {
		System.out.println("Account is open");
	}

	private void blockAccount() {
		System.out.println("Account is blocked");
	}

	private void closeAccount() {
		System.out.println("Account is closed");
	}
}
```

Класс может реализовывать любое число интерфейсов, которые указываются через запятую в объявлении класса после слова implements. Такой класс обязан предоставить реализацию всех методов, которые определены в интерфейсе, или же объявить себя абстрактным. Иными словами, интерфейс - это указание на то, что должен делать класс без указания на то, как именно это делать.

Можно объявить ссылку типа интерфейса, она сможет указывать на объект любого класса, который реализует этот интерфейс. При вызове метода через эту ссылку будет вызываться его реализованная версия метода у объекта, на который указывает эта ссылка, используя механизм динамического связывания.

---

6. Quick Sort, принцип и краткое описание алгоритма

Принцип

Опирается на принцип «разделяй и властвуй». Выбирается опорный элемент, это может быть любой элемент. От выбора элемента не зависит корректность работы алгоритма, но в отдельных случаях выбор этого элемента может повысить эффективность работы алгоритма. Оставшиеся элементы сравниваются с опорным и переставляются так, чтобы массив представлял собой последовательность: элементы меньше опорного-равные опорному элементы-элементы больше опорного. Для частей «больше» и «меньше» опорного элемента рекурсивно выполняется та же последовательность операций, если размер этой части составляет больше 1 элемента.

На практике входные данные обычно делят не на 3, а на 2 части. Например, «меньше опорного элемента» и «больше или равны опорному элементу». В общем случае разделение на 2 части эффективнее.

Алгоритм

1. проверяется, что во входном массиве больше 1 элемента, в противном случае алгоритм завершает свое действие;
2. с помощью какой-то схемы разбиения (например, схема разбиения Хоара или Ломуто) элементы в массиве меняются местами и определяется опорный элемент;
3. массив разбивается на 2: «меньше опорного элемента» и «больше или равны опорному элементу»;
4. рекурсивно вызывается этот же алгоритм для частей массива, полученных в пункте 3;

Разбиение Ломуто:

1. последний элемент выбирается опорным элементом, индекс хранится в переменной
2. каждый раз, когда находится элемент меньший или равный опорному, индекс увеличивается, а элемент вставляется перед опорным

Свойства

- время работы: лучшее – O(n*log n), среднее – O(n*log n), худшее – O(n^2);
- затраты памяти – O(log n);
- неустойчивая;
- количество обменов – обычно O(n*log n) обменов;

Плюсы

- в среднем очень быстрая;
- требует небольшое количество дополнительной памяти;

Минусы

- зависит от данных, на плохих данных деградирует до O(n^2);
- может привести к ошибке переполнения стека;
- сложность реализации;
- неустойчива;

---

3. Java Heap: принципы работы, структура.

В связи с тем, что Java постоянно развивается, на данный момент существует ряд противоречий в терминологии относительно типов памяти в Java. Например, часть ресурсов термины non-heap и stack приравнивают друг к другу. В своем ответе я подразумеваю разделение памяти в Java на три области: stack, heap, non-heap. Принцип работы памяти heap я описываю на примере применения одного из сборщиков мусора HotSpot VM – Serial GC. Есть и другие сборщики мусора, принцип работы которых отличается в той или иной мере.

Heap (Куча):

- используется для выделения памяти под объекты;
- создание нового объекта происходит в куче;
- объекты кучи доступны разным потокам;
- в куче работает сборщик мусора;
- куча имеет больший размер памяти, чем стек (stack);

Heap делится на 2 части:

- Young Generation Space тоже делится на 2 части:
    - Eden Space – область памяти, в которую попадают только созданные объекты, после сборки мусора эта область памяти должна освободиться полностью, а выжившие объекты перемещаются в Survivor Space
    - Survivor Space – область памяти, которая обычно подразделяется на две подчасти («from-space» и «to-space»), между которыми объекты перемещаются по следующему принципу:
        - «from-space» постепенно заполняется объектами из Eden после сборки мусора;
        - возникает необходимость собрать мусор в «from-space»;
        - работа приложения приостанавливается и запускается сборщик мусора;
        - все живые объекты «from-space» копируются в «to-space»;
        - после этого «from-space» полностью очищается;
        - «from-space» и «to-space» меняются местами («to-space» становится «from-space» и наоборот);
        - все объекты, пережившие определенное количество перемещений между двумя частями Survivor Space перемещаются в Old Generation;
- Old Generation Space – область памяти, в которую попадают долгоживущие объекты, пережившие определенное количество сборок мусора. Работает по принципу перемещения живых объектов к началу «old generation space», таким образом мусор остается в конце (мусор не очищается, поверх него записываются новые объекты), имеется указатель на последний живой объект, для дальнейшей аллокации памяти указатель просто сдвигается к концу «old generation space»;

Non-heap делится на 2 части:

- Permanent Generation – это пул памяти, который содержит интернированные строки, информацию о метаданных, классах, это пространство зарезервировано для классов, статических данных и т.п. Metaspace – замена Permanent Generation в Java 8. Основное различие в том, что Metaspace может динамически расширять свой размер во время выполнения. Размер Metaspace по умолчанию не ограничен.
- Code Cache – используется JVM при JIT-компиляции, здесь кэшируется скомпилированный код.