1. Определение инкапсуляции, наследования, полиморфизма

Инкапсуляция, наследование и полиморфизм (и абстракция) - это основные принципы парадигмы объектно-ориентированного программирования.

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.

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.

Классический пример наследования это иерархия геометрических фигур:

Класс Shape является базовым классом, а классы Circle и Square - производными классами. Они наследуют функциональность базового класса, и добавляют свою собственную:

```
public class Shape {
    protected String color = "white";

    public void setColor(String color) {
        this.color = color;
    }

    public void printMe() {
        System.out.println("I'm a Shape, color is " + color);
    }
}


public class Circle extends Shape {
    private double radius;

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public void printMe() {
        System.out.println("I'm a Circle, color is " + color + ", radius is " + radius);
    }
}


public class Square extends Shape {
    private int sideLength;

    public void setSideLength(int sideLength) {
        this.sideLength = sideLength;
    }

    @Override
    public void printMe() {
        System.out.println("I'm a Square, color is " + color + ", side is " + sideLength);
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Shape();
        shape.printMe();

        Circle circle = new Circle();
        circle.setColor("red");
        circle.setRadius(4.2);
        circle.printMe();

        Square square = new Square();
        square.setSideLength(5);
        square.setColor("green");
        square.printMe();
    }
```

```
}
```

/*Output:
I'm a Shape, color is white
I'm a Circle, color is red, radius is 4.2
I'm a Square, color is green, side is 5

Методы setRadius и setSideLength принадлежат сугубо производным классам, так как базовый класс не может решить соответствующих задач. Метод *setColor* реализован только в базовом классе, и эта функциональность полностью наследуется производными классами. А вот метод printMe реализован в базовом классе и переопределен в производных классах - таким образом мы получим разное поведение для каждого объекта.

Плюсы:

- Отсутствие дублирования кода
- Достижение модульного построения кода
- Иерархическая структура программы

Минусы:

- Сильная связанность: производный класс зависит от реализации базового класса.

Простой пример из жизни - новая модификация автомобиля. Если в предыдущей модификации были доступны опции поворота руля и торможения, то в новой версии эти опции тоже будут доступны, только к ним добавятся еще противотуманные фары и магнитола. То есть, новая модификация получит функциональность предыдущей и свою собственную. Старая модификация же функциональностью новой обладать не будет.

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.

Рассмотрим предыдущий пример:


```
public class Shape {
    protected String color = "white";

    public void printMe() {
        System.out.println("I'm a Shape");
    }
}


public class Circle extends Shape {
    private double radius;

    @Override
    public void printMe() {
        System.out.println("I'm a Circle");
    }
}


public class Square extends Shape {
    private int sideLength;

    @Override
    public void printMe() {
        System.out.println("I'm a Square");
    }
```

```
}
```

Один и тот же метод *printMe* будет выполнять разный код в зависимости от того, для какого объекта этот метод вызывается:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[5];
        for (int i = 0; i < shapes.length; i++) {
            shapes[i] = nextShape();
        }
        for (Shape shape : shapes) {
            shape.printMe();
        }
    }

    private static Shape nextShape() {
        Random random = new Random();
        switch (random.nextInt(3)) {
            case 0:
                return new Circle();
            case 1:
                return new Square();
            default:
                return new Shape();
        }
    }
}
```

```
/* Output:
I'm a Circle
I'm a Square
I'm a Shape
I'm a Circle
I'm a Square
```

В методе main мы создали массив ссылок типа Shape, а затем заполнили его ссылками на случайно созданные объекты разных типов. После этого последовательно вызвали метод printShape, в который передали ссылки из массива shapes. На момент написания программы и на момент компиляции неизвестно, на объект какого типа будет указывать очередная ссылка в массиве. Однако при вызове метода printMe для каждого элемента массива будет вызван метод того объекта, на который реально указывает ссылка. Это и есть пример динамического полиморфизма, реализованный с помощью механизма позднего или динамического связывания.

Удобство полиморфизма заключается в том, что код, работая с различными классами, не должен знать, какой именно класс он использует, так как все они работают по одному принципу.

Плюсы:

- Обеспечение слабосвязанного кода
- Удобство реализации и ускорение разработки
- Отсутствие дублирования кода

Простой пример из жизни - обучаясь вождению, человек может и не знать, каким именно автомобилем ему придется управлять после обучения. Однако это совершенно не важно, потому что основные и доступные для пользователя части автомобиля (интерфейс) устроены по одному и тому же принципу: педаль газа находится справа от педали тормоза, руль имеет форму круга и используется для поворота автомобиля.

2. Переопределение метода toString()

Метод toString в Java используется для предоставления ясной и достаточной информации об объекте (Object) в удобном для человека виде. Правильное переопределение метода toString может помочь в ведении журнала работы и в отладке Java программы, предоставляя ценную и важную информацию. Поскольку toString() определен в классе java.lang.Object и его реализация по умолчанию не предоставляет много информации, всегда лучшей практикой является переопределение данного метода в производном классе. По умолчанию реализация toString создает вывод в виде:

package.class@hashCode

```
public class User {
    private String name;
    private String surname;
    private int birthYear;

    public User(String name, String surname, int birthYear) {
        this.name = name;
        this.surname = surname;
        this.birthYear = birthYear;
    }

    public int getBirthYear() {
        return birthYear;
    }

    @Override
    public String toString() {
        return this.name + " " + this.surname + ", " + getBirthYear() + " г.р.";
    }
}

public class Main {
    public static void main(String[] args) {
        User user1 = new User("Petya", "Lee", 1988);
        System.out.println(user1.toString());
    }
}
```

/*Output:

Petya Lee, 1988 г.р.

3. Блок try-с ресурсами.

В Java 7 реализована возможность автоматического закрытия ресурсов в блоке try с ресурсами (try with resources). В операторе try открывается ресурс (файловый поток ввода), который затем читается. При завершении блока try данный ресурс автоматически закрывается, поэтому нет никакой необходимости явно вызывать метод close() у потока ввода, как это было в предыдущих версиях Java. Автоматическое управление ресурсами возможно только для тех ресурсов, которые реализуют интерфейс java.lang.AutoCloseable: InputStreamReader, FilterInputStream, FileReader и многие другие.

```
try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

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.

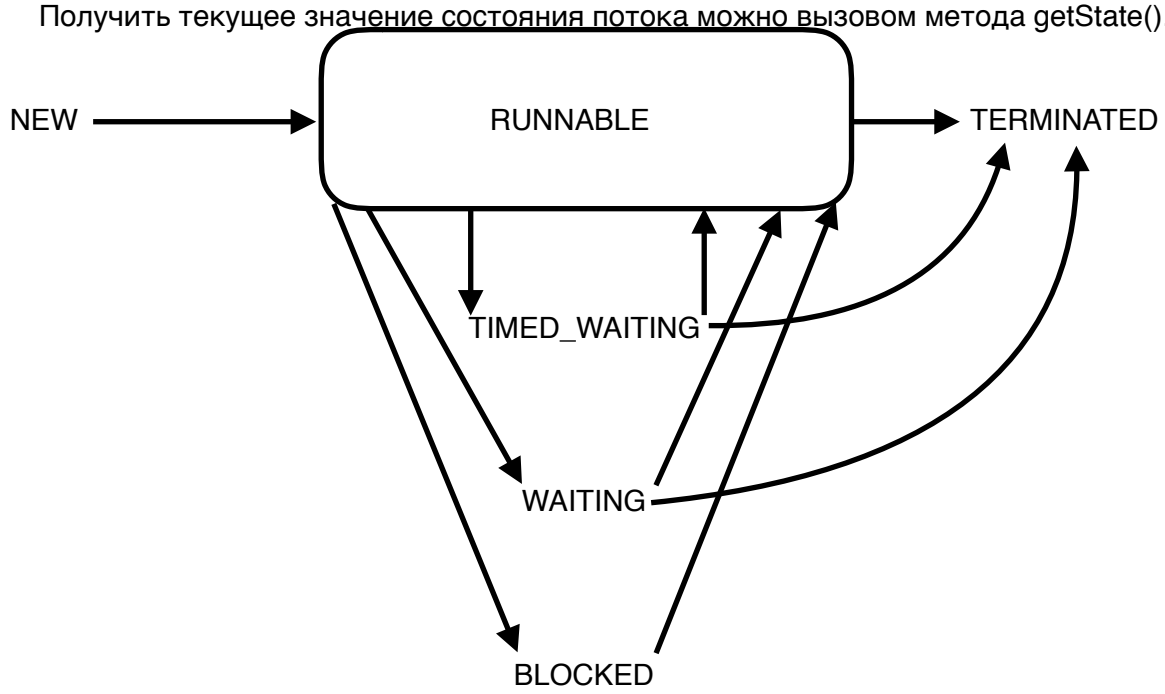
Для перевода потока из состояния «новый» в состояние «работоспособный» (**RUNNABLE**) следует выполнить метод start(), который вызывает метод run() — основной метод потока.

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. Абстрактные классы и абстрактные методы.

Абстрактные классы объявляются с ключевым словом abstract и содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Объекты таких классов создать нельзя с помощью оператора new, но можно создать объекты подклассов, которые реализуют все эти методы. При этом допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

```
abstract class AnyClass {
	int someField;
	void doAction();
	void print() {
		System.out.println(someField);
	}
}

public class Main {
	public static void main(String[ ] args) {
		AnyClass anyField;
		//anyField = new AnyClass();		нельзя создать объект!
		anyField = new AnySubClass();	//создаем объект подкласса абстрактного
класса
		anyField.print();
	}
}
```

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. JRE - определение, назначение.

Java Runtime Enviroment (JRE) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из Java Virtual Machine и библиотеки Java-классов.

Java Virtual Machine (JVM) – виртуальная машина Java, является основной частью JRE. JVM интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором (javac). JVM может также использоваться для выполнения программ, написанных на других языках программирования (Kotlin, Scala и т.п.).