1. Полиморфизм, механизмы Java, его обеспечивающие

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 (связывание производится во время работы программы). Тело метода выбирается в зависимости от фактического типа объекта.

Если связывание производится перед запуском программы (например, компилятором), то оно называется ранним связыванием. Тело методы выбирается в зависимости от типа ссылки. Оно применяется при вызове static и final методов (private методы по умолчанию является final): статические методы существуют на уровне класса, а не на уровне экземпляра класса, а final методы нельзя переопределять.

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

    public static void testStatic() {
        System.out.println("Shape.testStatic");
    }
}


public class Circle extends Shape {
    private double radius;

    public static void testStatic() {
        System.out.println("Circle.testStatic");
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.testStatic();
    }
}
```

```
/* Output:
Shape.testStatic
```

При вызове метода testStatic был задействован код класса Shape - того класса, какого типа ссылка использовалась для вызова. Это ранее связывание.

С понятием полиморфизма тесно связано переопределение метода (Overriding) - это создание в производном класса метода, который полностью совпадает по имени и сигнатуре с методом базового класса. Переопределение необходимо для работы механизма позднего связывания и используется для изменения поведения метода базового класса в производных классах. Для переопределения методов используется аннотация @Override, которая сообщает компилятору о попытке переопределись метод, а компилятор, в свою очередь, проверяет, правильно ли метод переопределен. Когда предопределенный метод вызывается из своего производного класса, он всегда ссылается на свой вариант. Если же требуется получить доступ к методу базового класса, то используется ключевое слово super:

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. Создание, модификация и сравнение объектов класса String. Пул литералов.

String - это один из наиболее широко используемых классов в Java, который находится в пакете java.lang. Это неизменяемый (immutable) и финализированный тип данных, представляющий последовательность символов.

Объект класса String можно создать несколькими способами:

1. Используя последовательность символов в двойных кавычках:

```
String catName = "Barsik";
```

2. С помощью конструкторов:

```
String emptyString = new String();
String dogName = new String("Jack");
String copyOfCatName = new String (catName);
String copyOfDogName = dogName;
```

Конструкторы могут формировать объект строки с помощью массива символов или массива байтов:

```
char[] arrayOfChars = {'M', 'u', 'r', 'z', 'i', 'k'};
String newCatName = new String (arrayOfChars);
```

Строки являются неизменными, поэтому при попытке изменить объект String создается новый объект String. Но ссылку на объект можно изменить так, чтобы она указывала на другой объект.

Методы сравнения строк:

boolean equals(Object obj)	сравнение строк
boolean equalsIgnoreCase(String str2)	сравнение строк без учета регистра символов
int compareTo(String str2)	лексиграфическое сравнение строк, вернет 0, если строки равны
int compareToIgnoreCase(String str)	лексиграфическое сравнение строк без учета регистра символов
boolean contentEquals(CharSequence cs)	сравнивает строку с объектом типа

boolean contentEquals(StringBuffer sb)	CharSequence сравнивает строку с объектом типа StringBuffer

Сравнение объектов класса String происходит при помощи метода equals:

```
System.out.println(catName.equals(copyOfCatName)); //true
```

Строки нельзя сравнивать оператором ==, хотя он и работает со строками. Дело в том, что оператор == сравнивает адреса объектов в памяти. То есть если сравниваемые переменные String указывают на один и тот же объект в памяти, то результат сравнения будет true. В противном случае - false, даже если строки будут равны посимвольно:

```
System.out.println(catName == copyOfCatName); //false!
```

Пул литералов - это коллекция ссылок на строковые объекты в Java Heap. Когда мы используем двойные кавычки для создания строки, сначала в пуле ищется строка с таким же значением. Если такая строка находится, то просто возвращается ссылка на эту строку, иначе создается новая строка в пуле, а затем возвращается ссылка:

```
String cat1 = "Tom";
String cat2 = "Tom";
System.out.println(cat1 == cat2); //true, так как ссылки cat1 и cat2 указывают на одну и
                                  ту же строку в пуле
```

Мы можем "заставить" класс String создать новый объект, независимо от того, есть строка с таким значением в пуле или нет:

```
String cat3 = new String ("Tom");
System.out.println(cat1 == cat3); //false, так как в пуле была создана еще одна строка
                                  с содержимым "Tom".
```

И наоборот: мы можем "заставить" класс String проверять наличие строки в пуле, даже если используется new при создании. Для этого существует метод intern:

```
String cat4 = new String ("Tom").intern();
System.out.println(cat1 == cat4); //true, так как новый объект в пуле не был создан, то
                                  есть ссылки cat1 и cat4 указывают на одну
                                  область памяти
```

Пул строк возможен благодаря неизменяемости строк. Он помогает экономить большой объем памяти, но с другой стороны создание строки занимает больше времени.

---

3. Определение понятия исключения и исключительной ситуации, оператор try-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. При этом ссылка на объект-исключение передается обработчику исключений, который пытается решить возникшую проблему и продолжить выполнение программы. Исключение — это источник дополнительной информации о ходе выполнения приложения. Такая информация позволяет выявить ошибки на ранней стадии.

Каждой исключительной ситуации поставлен в соответствие некоторый класс, экземпляр которого инициируется при исключительной ситуации. Если подходящего класса не существует, то он может быть создан разработчиком.

Управление обработкой исключений в Java осуществляется с помощью пяти ключевых слов: try, catch, throw, throws и finally. Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово try. В этом блоке и генерируется объект исключения, если возникает исключительная ситуация, после чего управление передается в соответствующий блок catch. Блок catch помещается сразу после блока try, в нем задается тип исключения, которое требуется обработать.

После блока try может быть несколько блоков catch для разной обработки разных исключений, может быть один блок catch, объединяющий разные исключения для их одинаковой обработки, а может и вообще не быть блока catch, но в этом случае обязательно должен быть блок finally. Общая форма обработки выглядит так:

```
try {
   \\код, в котором может возникнуть исключительная ситуация
} catch (тип_исключения_1 e) {
   \\обработка исключения 1
} catch (тип_исключения_2 e) {
   \\обработка исключения 2
} finally {
   \\действия, которые должны быть выполнены независимо от того, возникла
исключительная ситуация или нет
}
```

---

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. Интерфейсы в Java 8.

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.

Начиная с Java 8 появилась возможность использовать в интерфейсах дефолтные и статические методы.

Default-методы - это неабстрактные реализации методов. Классы, которые реализуют интерфейс с default-методами, обязаны переопределять только абстрактные методы, а переопределение default-методов необязательно. То есть классы, которые реализуют интерфейс с default-методами, могут пользоваться уже реализованными методами. Default-методы следует использоваться с осторожностью, чтобы не столкнуться с проблемой ромба множественного наследования.

Static-методы - это статические неабстрактные методы, они являются частью интерфейса и не могут быть использованы в классах, реализующих такой интерфейс. Они не могут быть предопределены - если в классе есть метод с такой же сигнатурой, он будет являться самостоятельным методом. Используются как вспомогательные методы для default-методов.

Функциональный интерфейс - это интерфейс с только одним абстрактным методом, для обозначения такого интерфейса используется аннотация @FunctionalInterface. Аннотация не является обязательной, но предостерегает программиста от случайного добавления абстрактных методов в функциональный интерфейс. Функциональные интерфейсы могут содержать сколько угодно default и static методов, но абстрактный метод должен быть только один. Функциональные интерфейсы позволяют использовать лямбда-выражения для создания анонимных классов, реализующих этот интерфейс.

---

6. Что такое сортировка, зачем она нужна, какие виды сортировок бывают, какие у них есть свойства

Сортировка – это алгоритм упорядочивания элементов по какому-либо признаку. Сортировка необходима для обеспечения удобства поиска информации. Например, у нас есть список из 1000 людей разного возраста. В данный момент нам может быть необходимо найти 20 самых младших людей, можно проходить каждый раз по всему массиву данных о людях, находить самого младшего, записывать в отдельную структуру

данных, потом проходить вновь, искать следующего младшего, проверяя, что такого уже не записано в нашу результирующую структуру данных. Это не очень удобно. В отсортированных данных по возрасту нам нужно было бы просто отобрать первые 20 (или последние – в случае убывающей сортировки) человек и все. К тому же это в данный момент нам нужно было отобрать 20 самых младших людей, а спустя пять минут будет нужно найти 4 старших человека или людей определенного возраста. Поиск информации в беспорядочно расположенных данных трудоемок. В отсортированных данных поиск информации происходит гораздо быстрее, тем более для отсортированных данных могут применяться специальные виды поиска (например, бинарный поиск), которые ищут информацию с еще большей производительностью.

Основные свойства сортировок:

- **время работы** – это свойство зачастую считается наиболее важным. Оценивается худшее, среднее и лучшее время работы алгоритма. У большинства алгоритмов временные оценки бывают O(n*log n), O(n^2);
- **дополнительная память** – свойство сортировки, показывающее, сколько дополнительной памяти требуется алгоритму. Сюда входят дополнительный массив, переменные, затраты на стек вызовов. Обычно доп. затраты памяти составляют O(1), O(log n), O(n);
- **устойчивость** – устойчивой называется сортировка, не меняющая порядок объектов с одинаковыми ключами. Ключ – поле элемента, по которому проводим сортировку;
- **количество обменов** – этот параметр может быть важен в том случае, если обмениваемые объекты имеют большой размер, т.к. при большом количестве обменов время работы сильно увеличивается;

Некоторые виды сортировок:

- **Пузырьковая сортировка**
  - время работы: лучшее – O(n), среднее – O(n^2), худшее – O(n^2);
  - затраты памяти – O(1);
  - устойчивая;
  - количество обменов – обычно O(n^2) обменов;
- **Сортировка выбором**
  - время работы: лучшее – O(n^2), среднее – O(n^2), худшее – O(n^2);
  - затраты памяти – O(1);
  - неустойчивая;
  - количество обменов – обычно O(n) обменов;
- **Сортировка вставками**
  - время работы: лучшее – O(n), среднее – O(n^2), худшее – O(n^2);
  - затраты памяти – O(1);
  - устойчивая;
  - количество обменов – обычно O(n^2) обменов;
- **Быстрая сортировка**
  - 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;
- **Сортировка слиянием**
  - 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;
- **Поразрядная сортировка**
  - время работы: лучшее – O(nk), среднее – O(nk), худшее – O(nk);
  - затраты памяти – O(n + k);
  - устойчивая;
  - количество обменов – обычно O(nk) обменов;

7. 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.