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. Работа с объектами типа StringBuilder и StringBuffer

Строки в Java являются неизменяемыми объектами, но часто необходимо получить такой строковый объект, который можно будет изменять. Для этого используются классы StringBuilder и StringBuffer, объекты которых являются изменяемыми. Единственным отличием StringBuilder от StringBuffer является потокобезопасность StringBuffer. В однопоточном использовании StringBuilder практически всегда значительно быстрее, чем StringBuffer. Для этих классов не переопределены методы equals() и hashCode(), то есть сравнить содержимое двух объектов невозможно, а хэш-коды всех объектов этого типа вычисляются так же, как и для класса Object.

Конструкторы StringBuilder:

StringBuilder(String str) – создает StringBuilder, значение которого устанавливается в передаваемую строку, плюс дополнительные 16 пустых элементов в конце строки

StringBuilder(CharSequence charSeq) – создает StringBuilder, содержащий те же самые символы, что в CharSequence, плюс дополнительные 16 пустых элементов, конечных CharSequence

StringBuilder(int length) – создает пустой StringBuilder с указанной начальной вместимостью

StringBuilder() – создает пустой StringBuilder из 16 пустых элементов

Чтение и изменение объекта StringBuilder:

int length() – возвращает количество символов в строке

char charAt(int index) – возвращает символьное значение, расположенное на месте index

void setCharAt(int index, char ch) – символ, расположенный на месте index, заменяется символом ch

CharSequence subSequence(int start, int end) – возвращает новую подстроку

---

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. 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. Вложенные классы: объявление и правила работы.

Классы могут взаимодействовать друг с другом путем организации структуры с определением одного класса в теле другого. Это делает код более эффективным и понятным, а также позволяет сокрыть реализацию, так как внутренний класс может быть не виден вне основного класса. Вложенные классы могут быть статическими и нестатическими. Статические классы могут обращаться к членам включающего класса только через его объект. А нестатические классы имеют доступ ко всем членам включающего класса.

Нестатические вложенные классы называются внутренними (inner) классами. Доступ к элементам внутреннего класса из внешнего возможен только через объект внутреннего класса, который создается в методе внешнего класса.

```
public class Ship {
	private class Engine {
		public void launch() {
			System.out.println("Start engine");
		}
	}
	public void initShip() {
		Engine engine = new Engine();
		engine.launch();
	}
}
```

Методы внутреннего класса имеют доступ ко всем полям и методам внешнего класса, а внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутренного класса, при этом доступ будет разрешен и к private полям и методам.

Внутренние классы могут быть объявлены как final, abstract, private, protected, public. Объявление внутреннего класса как private обеспечивает полную недосягаемость вне внешнего класса. Внутренние классы могут быть объявлены внутри методов или логических блоков внешнего класса. В этом случае видимость внутреннего класса определяется видимостью того блока, где он объявлен.

Внутренние классы могут наследовать другие классы, могут реализовывать интерфейсы, а так же сами могут быть базовыми классами для других классов. То есть этот механизм может решить проблему множественного наследования.

Внутренние классы не могут содержать статические поля и методы, кроме final static, но могут наследовать.

Статические вложенные классы называются вложенными (nested) классами. Вложенный класс логически связан с внешним классом, однако может быть использован независимо от него. Такие классы объявляются с ключевым словом static. Если класс вложен в интерфейс, то он является статическим по умолчанию.

Для доступа к нестатическим полям и методам внешнего класса, статический вложенный класс должен создать объект внешнего класса, а к статическим полям и методам он имеет доступ напрямую. Из этого следует, что для создания объекта статического вложенного класса нет необходимости создавать объект внешнего класса.

Вложенные классы могут наследовать другие классы, могут реализовывать интерфейсы, а так же сами могут быть базовыми классами для других классов. Производный класса вложенного класса теряет доступ к членам внешнего класса при наследовании.

Статические методы вложенного класса при вызове требуют указания полного пути к нему. То есть, если внешний класс это Car, а вложенный статический класс это Driver, в котором есть статический метод park(), то для вызова требуется такая строка:

```
Car.Driver.park();
```

Безымянные вложенные классы, которые объявляются при помощи оператора new называются анонимными классами. Применяются для придания уникальной функциональности отдельно взятому экземпляру. То есть можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Основное требование к анонимным классами - они должны наследовать или расширять уже существующий класс или интерфейс.

Анонимные классы как правило используются для переопределения нескольких методов и/или создания собственных методов объекта, при этом в создании нового полноценного класса нет необходимости из-за узкой направленности или одноразового применения.

Невозможно определить и переопределить конструктор анонимного класса. Анонимные классы не могут содержать статические поля и методы, кроме final static, но могут наследовать. Допускают вложенность, однако это не рекомендуется из-за усложнения читаемости кода.

Локальные классы.

Объявляются и могут быть использованы только внутри методов внешнего класса. Имеют доступ к членам внешнего класса. Имеют доступ как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры, используемые локальным классом, должны быть объявлены final. Локальные классы не могут содержать статические поля и методы, кроме final static, но могут наследовать.

---

6. Insertion Sort, принцип и краткое описание алгоритма.

Принцип

Есть часть массива (в роли этой части может выступать первый элемент в массиве), которая уже отсортирована, требуется вставить остальные элементы в отсортированную часть, сохранив упорядоченность.

Алгоритм

1) выбирается один из элементов входных данных и вставляется на нужную позицию в уже отсортированной части массива;
2) шаг 1 повторяется до тех пор, пока весь массив не будет отсортирован;

Обычно в качестве отсортированной части массива изначально выступает первый элемент (элемент с индексом 0), порядок выбора очередного элемента для вставки в отсортированную часть произволен, но обычно с целью достижения устойчивости алгоритма элементы вставляются по порядку их появления в исходном массиве.

Свойства

- 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;

Плюсы

- проста в реализации;
- не нужна дополнительная память;
- ускоряется на частично отсортированных массивах;
- хороша на малом количестве элементов (<100);
- устойчивая;

Минусы

- медленная в среднем и худшем случае;

---

7. Компиляция и запуск в консольном режиме.

Сначала нужно скомпилировать класс, то есть перевести его в байт-код. Для этого в состав JDK входит компилятор Java кода javac:

E:\EJC>javac -sourcepath src -d out src/tasks/task_01/Main.java

-sourcepath задает каталог исходного кода, что позволяет компилятору найти не только тот класс, который мы отправили на компиляцию, но и найти в каталоге исходного кода другие классы, в т.ч. классы из других пакетов, которые используются в классе, отправленном на компиляцию.

С помощью **-d** мы отметили директорию, в которой будут лежать скомпилированные классы. Именно в **out** компилятор автоматически создает структуру каталогов идентичную структуре каталогов в **src**. Далее компилятор находит класс, который мы отправили на компиляцию **Main.java**, и компилируется он и все классы, которые он использует, в байткод в соответствующие каталоги в каталоге **out**.

Теперь мы можем запустить нашу программу с помощью java:

E:\EJC>java -cp out tasks.task_01.Main

С помощью **-cp** (или **-classpath**) указываем из какого каталога мы загружаем файлы, указываем класс, который запускаем.

Чтобы не прописывать полный путь к **java** и **javac** в консоли, можно задать переменную среды окружения. Если этого не делать, то придется полностью прописывать путь размещения **javac** и **java**.