**Корреляция и параметризация**

Корреляция – поиск, сохранение, передача динамических значений(токенов авторазации)

Параметризация – использование данных для параметров из файла/случайная генерация(у нас есть JSON запрос для покупки, а в нем нужно использовать динамическое значение суммы)

**Pacing**

Шаг нагрузки (Pacing)- такой интервал времени в течение которого гарантировано выполняется один проход сценария

**Виды тестов**

|  |  |  |
| --- | --- | --- |
| Тестирование стабильности | Stability Testing | Достаточно ли надежно работает система на долгом интервале времени? |
| Тестирование отказоустойчивости | Failover Testing | Сможет ли система переместиться сама на другой сервер в случае сбоя основного сервера? |
| Тестирование восстановления | Recovery Testing | Как быстро восстановится система? |
| Стрессовое тестирование | Stress Testing | Что произойдет при незапланированной нагрузке? |
| Тестирование объемов | Volume Testing | Как будет работать система, если объем базы данных увечится в 100 раз? |
| Тестирование масштабируемости | Scalability Testing | Как будет увеличиться нагрузка на компоненты системы при увеличении числа пользователей? |
| Тестирование потенциальных возможностей | Capacity Testing | Какое количество пользователей может работать? |
| Конфигурационное тестирование | Configuration Testing | Как заставить систему работать быстрее? |
| Тестирование сравнения | Compare Testing | Какое оборудование и ПО выбрать? |

* Достаточно ли быстро работает приложение? (Performance Testing)
* Как заставить приложение работать быстрее? (Configuration Testing)
* Какое количество пользователей может работать? (Capacity Testing)
* Что произойдет при незапланированной нагрузке? (Stress Testing)
* Достаточно ли надежно работает приложение? (Stability Testing)
* Какое оборудование и ПО выбрать? (Compare Testing)

**Java**

**Сборка мусора (Garbage Collection) в Java** — это механизм автоматического управления памятью, встроенный в виртуальную машину Java (JVM). Этот процесс позволяет разработчикам сосредоточиться на логике приложений, не беспокоясь об освобождении памяти вручную.

Java-программы компилируются в байт-код, который исполняется JVM. При выполнении программы объекты создаются в куче (heap) — области памяти, выделенной для динамически создаваемых объектов.

Куча разделена на несколько поколений:

* **Young Generation** — используется для новых объектов. Она делится на три области: **Eden** (зона создания новых объектов) и два **Survivor Spaces** (для объектов, которые пережили одну или несколько сборок мусора). Здесь работает алгоритм **Minor GC**, который быстро очищает память, удаляя недолговечные объекты.
* **Old Generation** (или Tenured) — сюда перемещаются объекты, которые пережили несколько сборок в **Young Generation**. Эти объекты, как правило, имеют более долгий срок жизни. Для очистки **Old Generation** используется более сложный и медленный процесс — **Major GC** или **Full GC**.
* **Metaspace** — начиная с Java 8, эта область хранит метаинформацию классов и заменяет прежнюю **PermGen**.

**Сборщик мусора (GC)** работает по принципу поиска объектов, которые больше не имеют активных ссылок. Он освобождает память, устраняя "мусор" (ненужные объекты), и перераспределяет её для новых данных. Основные алгоритмы GC включают:

* Serial GC — однопоточный, подходит для небольших приложений.
* Parallel GC — многопоточный, оптимизирован для пропускной способности.
* G1 GC — использует региональную модель памяти, эффективен для больших куч.
* ZGC и Shenandoah — низкая задержка, предназначены для современных высоконагруженных систем.

Эффективное управление памятью и сборка мусора играют ключевую роль в производительности Java-приложений. Разработчики могут настраивать параметры JVM (например, -Xms, -Xmx, -XX:NewRatio и другие), чтобы оптимизировать использование памяти и поведение GC под конкретные задачи.

**ООП**

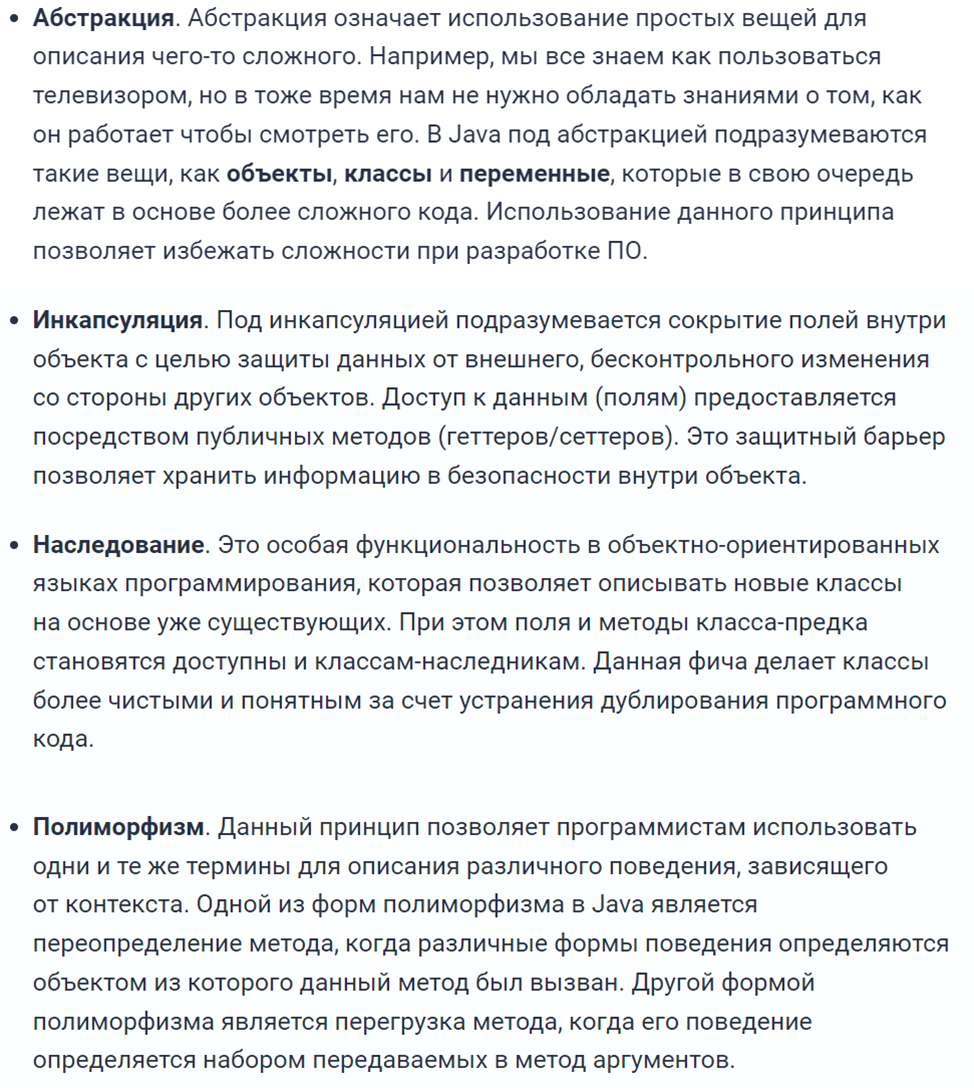
Это подход к разработке, где программы строятся из объектов, объединяющих данных и методы для работы с ними

**Абстракция** скрывает сложные детали, предоставляя только необходимые свойства и поведение (например, интерфейс Animal с методом makeSound() для любых животных).

**Инкапсуляция** защищает данные объекта, предоставляя доступ только через методы (например, приватное поле balance в классе BankAccount с методами deposit() и withdraw()).

**Наследование** позволяет создавать новые классы на основе существующих, повторно используя код (например, класс Dog наследует Animal).

**Полиморфизм** обеспечивает использование объектов разных типов через единый интерфейс (например, метод makeSound() вызывает разное поведение для Dog и Cat)



**Класс –** основная структура, предоставляющая шаблон для создания объектов. Он содержит данные(поля) и методы(функции), которые определяют поведение объекта

public class Person {  
 String name;  
 int age;  
  
 void sayHello() {  
 System.out.println(“Hello, my name is “ + name);  
 }  
}

**Объект**  - экземпляр класса, созданный с помощью оператора new  
Person person = new Person();  
person.name = “John”;  
person.sayHello();  
  
**Конструктор** – специальный метод, используемый для инициализации объекта. Имя конструктора совпадает с именем класса, он не имеет возвращаемого значения.  
public class Person {  
 String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
}  
  
**Интерфейс** – это контракт, который определяет набор методов без их реализации. Класс, реализующий интерфейс, обязан предоставить реализацию всех его методов. Интерфейсы используются для достижения абстракции и полиморфизма  
public interface Animal {  
 void makeSound():  
}  
  
public class Dog impements Animal {  
 public void makeSound() {  
 System.out.println(“Woof”);  
 }  
}  
  
**Наследование** – механизм, позволяющий одному классу(подклассу) наследовать свойства и поведение другого(родительского) класса. Для этого используется ключевое слово extends  
public class Animal {  
 void eat() {  
 System.out.println(“This animal eats food”);  
 }  
}  
  
public class Dog extends Animal {  
 void bark() {  
 System.out.println(“Woof”);  
 }  
}  
  
**Перегрузка методов(Overloading)** – определение методов с одинаковым именем, но с разными параметрами  
  
**Переопределение методов(Overriding)** – предоставление новой реализации метода родительского класса в подклассе  
  
**Статические члены** – поля и методы, принадлежащие классу, а не объекту, и доступные через имя класса  
public class MathUtils {  
 public statin int square(int x) {  
 return x \* x;  
 }  
}  
int result = MathUtils.square(4);  
  
**Модификаторы доступа** – определяют уровень доступа к членам класса:

* Public – доступен всем
* Private – доступен только внутри класса
* Protected – доступен внутри пакета и подклассов
* (без модификатора) – доступен только внутри пакета