**1. Понятие класса и объекта. Статический и нестатический контекст класса. Члены класса. Методы, поля, конструкторы, блоки инициализации. Ключевые слова abstract и final**

**Класс — это шаблон или описание, по которому создаются объекты. Объект — это конкретный экземпляр класса, обладающий своими значениями полей и доступом к методам. Статический контекст связан с самим классом, а нестатический — с конкретным объектом. В статическом контексте нельзя напрямую обращаться к нестатическим членам, так как они принадлежат конкретным экземплярам.**

**Члены класса делятся на:**

* **Поля (переменные) — данные, которые хранят состояние объекта или класса.**
* **Методы — функции, определяющие поведение объекта.**
* **Конструкторы — специальные методы, которые вызываются при создании объекта и инициализируют его.**
* **Блоки инициализации — участки кода, выполняющиеся при создании объекта или загрузке класса (есть статические и нестатические).**

**Ключевое слово abstract помечает класс или метод как абстрактный. Абстрактный класс нельзя создать напрямую, а абстрактный метод не имеет реализации — его должны реализовать подклассы.**

**Ключевое слово final запрещает дальнейшие изменения:**

* **final-класс нельзя унаследовать.**
* **final-метод нельзя переопределить.**
* **final-переменной можно присвоить значение только один раз.**

**2. Основополагающие принципы ООП. Инкапсуляция. Средства реализации инкапсуляции. Модификаторы доступа**

**Есть два распространенных понятия — инкапсуляция и сокрытие. И под словом «инкапсуляция» авторы понимают то одно, то другое.**

**Изначальное значение слова «инкапсуляция» в программировании — объединение данных и методов работы с этими данными в одной упаковке («капсуле»). В Java в роли упаковки-капсулы выступает класс. Класс содержит в себе и данные (поля класса), и методы для работы с этими данными.**

**Сокрытие реализации осуществляется через простой и удобный интерфейс.**

* **private — доступен только внутри текущего класса.**
* **default (отсутствие модификатора) — доступен внутри пакета.**
* **protected — доступен внутри пакета и в наследниках.**
* **public — доступен отовсюду.**

**Доступ к приватным полям обычно обеспечивается через методы геттеры и сеттеры.**

**3. Основополагающие принципы ООП. Наследование. Управление наследованием**

**Наследование — это механизм ООП, позволяющий одному классу (подклассу или производному классу) заимствовать поля и методы другого класса (базового или родительского класса). Это позволяет избежать дублирования кода, упростить сопровождение программ и выстроить иерархическую структуру классов.**

**Ключевые особенности наследования:**

* **Подкласс автоматически получает доступ ко всем public и protected членам базового класса.**
* **Подкласс может расширять или изменять поведение родительского класса.**
* **Один класс может наследовать только от одного класса (в Java — одиночное наследование классов).**

**Управление наследованием включает в себя:**

* **Ограничение наследования:**
  + **Класс, объявленный с ключевым словом final, нельзя наследовать.**
  + **Метод, объявленный как final, нельзя переопределить в подклассе.**
* **Ограничение доступа к членам класса:**
  + **Модификатор private полностью закрывает доступ.**
  + **protected делает члены доступными только в пределах пакета и в подклассах.**
* **Переопределение методов:**
  + **Используется для изменения логики унаследованных методов.**
  + **Аннотация @Override помогает явно указать, что метод должен переопределять метод суперкласса. Это даёт компилятору возможность проверить правильность переопределения.**

**4. Основополагающие принципы ООП. Полиморфизм. Средства реализации полиморфизма**

**Полиморфизм (от греч. «много форм») позволяет объектам одного интерфейса иметь различную реализацию. Это означает, что один и тот же код (например, вызов метода) может работать по-разному в зависимости от конкретного типа объекта, к которому он применяется.**

**Различают два вида полиморфизма:**

1. **Времени компиляции (статический) — реализуется через перегрузку методов (overloading).**
2. **Времени выполнения (динамический) — реализуется через переопределение методов (overriding) и использование интерфейсов или абстрактных классов.**

**Средства реализации:**

* **Переопределение методов: подкласс предоставляет свою реализацию метода, уже определённого в базовом классе. Это позволяет объектам подкласса вести себя специфическим образом при вызове метода через ссылку базового типа.**
* **Перегрузка методов: в пределах одного класса можно определить несколько методов с одинаковым именем, но разными параметрами. Это обеспечивает разнообразие поведения в зависимости от переданных аргументов.**
* **Интерфейсы и абстрактные классы: позволяют задать общую форму (контракт), которую реализуют разные классы. Благодаря этому, код может работать с абстракцией, а конкретное поведение определяется реализацией.**

**Полиморфизм усиливает гибкость кода и позволяет создавать расширяемые, масштабируемые архитектуры.**

**5. Понятие класса и интерфейса: абстрактные классы, абстрактные методы. Отличие абстрактного класса от интерфейса**

**Класс — это описание структуры объекта: его состояния (через поля) и поведения (через методы). Класс можно инстанцировать, создавая объекты.**

**Интерфейс — это форма контракта, который определяет набор методов, которые должен реализовать класс, но не содержит реализации (до Java 8).**

**Абстрактный класс:**

* **Это класс, который не может быть создан как объект, и может содержать как реализованные, так и абстрактные методы (т.е. методы без тела).**
* **Может содержать поля, конструкторы, обычные методы.**
* **Используется, когда нужно задать частичную реализацию с возможностью расширения.**

**Абстрактный метод:**

* **Метод без тела, содержащий только сигнатуру.**
* **Должен быть реализован в подклассе.**

**Интерфейс:**

* **До Java 8 — только абстрактные методы, без реализации.**
* **С Java 8 — можно определять default методы (с реализацией по умолчанию) и static методы.**
* **С Java 9 — также можно определять private методы для использования внутри интерфейса.**
* **Не может содержать состояние в виде обычных полей (разрешены только public static final константы).**

**Ключевые отличия:**

* **Наследование: класс может наследовать только один абстрактный класс, но реализовывать несколько интерфейсов.**
* **Содержимое: абстрактный класс может содержать поля, конструкторы, частичную реализацию. Интерфейс в классическом виде — только сигнатуры методов.**
* **Назначение: интерфейс — это чистый контракт: что должно быть сделано. Абстрактный класс — основа поведения, частично реализующая функциональность.**

**Выбор между интерфейсом и абстрактным классом зависит от архитектурных задач: интерфейсы хороши для описания возможностей, абстрактные классы — для выстраивания иерархий с общим поведением.**

**6. Интерфейсы: определение, реализация, наследование**

**Интерфейс в Java — это абстрактный тип, который определяет набор методов, не предоставляя их реализации. Интерфейсы описывают *что должно быть сделано*, а не *как* это делать. Это своего рода контракт, который обязуется выполнить любой класс, реализующий данный интерфейс.**

**Реализация интерфейса:**

* **Класс реализует интерфейс с помощью ключевого слова implements.**
* **Если интерфейс содержит методы, класс обязан реализовать их все (если он не является абстрактным сам по себе).**
* **Интерфейс не может содержать конструктора, потому что он не описывает поведение объекта, а лишь его внешний интерфейс.**

**Наследование интерфейсов:**

* **Интерфейсы могут наследовать друг друга с помощью ключевого слова extends.**
* **Один интерфейс может расширять сразу несколько интерфейсов.**
* **Это позволяет объединять поведение нескольких интерфейсов в один.**

**Ключевая особенность интерфейсов — это множественная реализация:**

* **Один класс может реализовывать несколько интерфейсов, что позволяет обойти ограничение одиночного наследования в Java.**
* **Это особенно полезно для организации архитектуры, основанной на ролях и возможностях (например, интерфейсы Serializable, Comparable, Runnable и т.п.).**

**С Java 8 интерфейсы получили возможность содержать default методы с реализацией и static методы. Это сделало интерфейсы более мощным инструментом, но не отменило их основной контрактной природы.**

**7. Дженерики: Определение, реализация, примеры**

**Дженерики (Generics) — это механизм обобщённого программирования в Java, позволяющий создавать классы, интерфейсы и методы, способные работать с различными типами данных без указания конкретного типа заранее.**

**Цель дженериков:**

* **Повысить типовую безопасность (ошибки типов выявляются на этапе компиляции).**
* **Уменьшить количество приведения типов (casting).**
* **Улучшить читаемость и переиспользуемость кода.**

**Синтаксис:**

* **Используются угловые скобки <T>, где T — параметр типа.**
* **Пример объявления: class Box<T> { T value; } Здесь T может быть любым ссылочным типом (не примитивом).**

**Примеры использования:**

* **Коллекции: List<String> или Map<Integer, User>.**
* **Метод, который работает с любым типом: <T> void print(T value).**
* **Ограниченные дженерики: <T extends Number> — только числовые типы.**

**Преимущества:**

* **Один универсальный код для разных типов.**
* **Отсутствие необходимости создавать отдельные классы для IntegerBox, StringBox, и т.д.**
* **Повышение надёжности и читаемости.**

**Важно: дженерики в Java реализованы через механизм *type erasure* — все параметры типов удаляются на этапе компиляции, и во время выполнения работают обычные классы без знания о типе T.**

**8. Интерфейсы Comparator и Comparable. Описание, различия, примеры использования**

**Оба интерфейса предназначены для сравнения объектов, но используются в разных ситуациях и имеют различную философию:**

**Comparable:**

* **Интерфейс с одним методом int compareTo(T o).**
* **Реализуется классом, объекты которого нужно сравнивать.**
* **Позволяет задать естественный порядок объектов (natural ordering).**
* **Пример: класс String реализует Comparable<String>, сравнивая строки по алфавиту.**
* **Используется напрямую в Collections.sort() и Arrays.sort() без внешнего компаратора.**

**Минус: можно задать только один способ сравнения, встроенный в класс.**

**Comparator:**

* **Интерфейс с методом int compare(T o1, T o2).**
* **Реализуется отдельно, не связан с классом сравниваемых объектов.**
* **Позволяет задавать альтернативные способы сравнения (например, по имени, по дате и т.д.).**
* **Можно создавать сколько угодно компараторов для одного типа.**
* **Используется в Collections.sort(list, comparator) и других методах сортировки с внешним критерием.**

**Сравнение:**

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Comparable** | **Comparator** |
| **Реализуется** | **Самим классом** | **Внешним объектом** |
| **Метод** | **compareTo()** | **compare()** |
| **Число реализаций** | **Только одна** | **Сколько угодно** |
| **Где используется** | **sort() без аргумента** | **sort() с аргументом** |

**Пример ситуации:**

* **У класса Person реализован Comparable для сравнения по возрасту.**
* **Для сортировки по фамилии можно создать Comparator<Person>.**

**Эти интерфейсы — фундаментальные инструменты для организации сортировки и поиска в структурах данных, особенно при работе с коллекциями.**

**9. Коллекции типа List. Описание, представители. Механизм работы, различия реализаций**

**List — это интерфейс коллекции в Java, представляющий упорядоченное множество элементов, в котором допускаются дубликаты. Элементы имеют индексацию, начиная с 0, и сохраняют порядок добавления. Это делает List особенно удобным для последовательного хранения и обработки данных.**

**Основные реализации интерфейса List:**

1. **ArrayList:**
   * **Внутренне реализован как динамический массив.**
   * **Поддерживает быстрый доступ по индексу (время доступа O(1)).**
   * **Вставка или удаление элементов в середине списка требует сдвига элементов, что делает эти операции медленными (время — O(n)).**
   * **Хорошо подходит для сценариев с частым чтением и редким изменением.**
2. **LinkedList:**
   * **Реализован как двусвязный список: каждый элемент хранит ссылки на предыдущий и следующий.**
   * **Обеспечивает быструю вставку и удаление в любом месте списка (O(1) при наличии ссылки на нужное место).**
   * **Доступ к элементам по индексу происходит линейно (время O(n)), так как необходимо пройти список от начала или конца.**
   * **Полезен в случаях, когда важна производительность операций вставки/удаления.**

**Ключевые отличия:**

* **Производительность: ArrayList быстрее при случайном доступе к элементам; LinkedList быстрее при вставке и удалении из начала или середины.**
* **Память: ArrayList хранит только данные, LinkedList — данные и ссылки, поэтому занимает больше памяти.**

**Также есть неизменяемые списки (List.of(...)), которые появились в Java 9 — они компактны, но не поддерживают модификации.**

**10. Коллекции типа Map. Описание, представители. Механизм работы, различия реализаций**

**Map — это структура данных, представляющая отображение "ключ → значение", где ключи уникальны, а значения могут повторяться. Ключи используются для быстрого доступа к связанным с ними значениям.**

**Основные реализации интерфейса Map:**

1. **HashMap:**
   * **Основан на хеш-таблице.**
   * **Использует хеш-функцию для определения позиции ключа.**
   * **Обеспечивает быстрый доступ, вставку и удаление (в среднем O(1)), если хеш-функция хорошая.**
   * **Не гарантирует порядок элементов.**
   * **При высоком числе коллизий производительность может ухудшаться (внутренне с Java 8 используется сбалансированное дерево при большом числе коллизий в одной корзине).**
2. **LinkedHashMap:**
   * **Наследует HashMap, но дополнительно поддерживает двусвязный список элементов.**
   * **Сохраняет порядок вставки или порядок доступа (если включена соответствующая опция).**
   * **Чуть медленнее HashMap из-за дополнительной структуры, но порядок может быть критичен.**
3. **TreeMap:**
   * **Реализован на основе красно-чёрного дерева (сбалансированного бинарного дерева поиска).**
   * **Сохраняет элементы в отсортированном порядке по ключам.**
   * **Обеспечивает логарифмическое время операций (O(log n)) для доступа, вставки и удаления.**

**Ключи должны реализовывать Comparable или быть предоставлены с Comparator.**