1. ООП – объектно-ориентированное программирование. Этот подход к написанию программ основан на таких понятиях как класс, экземпляр класса. Класс – шаблон, чертёж по которому будут создаваться объекты. В классе могут быть поля и методы. Созданные на основе класса объекты – экземпляры класса. Они располагаются в динамической памяти (куче).
2. Наследование – это по сути возможность расширения существующего класса за счёт создания подкласса. При этом можно использовать код, написанный для исходного класса. Подкласс наследует все поля и методы расширяемого класса. Все классы по умолчанию наследуются от класса Object.
3. В Java операции над классом осуществляются с помощью методов класса, в которых часто используется код, скрытый от остальных объектов. В таком процессе сокрытия данных и их недоступности для прочих объектов заключается смысл инкапсуляции данных. Чтобы пользоваться классом, необходимо знать его интерфейс, т е данные и функции, предоставляемые для взаимодействия с классом извне, а не то, как он реализован внутри.
4. Полиморфизм означает возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования. Понятие полиморфизма, в свою очередь, опирается на два других понятия: совместное использование *overloading* (перегрузку, доопределение, *совместное использование*) и переопределение *overriding*.
5. Пакетом (пространством имен) в Java называется структура вложенных по какому-то признаку папок с размещенными в них классами (интерфейсами, перечислениями, аннотациями), необходимыми проекту.
6. Интерфейс — не что иное, как именованное множество абстрактных полей и методов, которые лишь объявляются, но не определяются. Интерфейсы предназначены для объявления типов, состоящих только из абстрактных методов и констант; они позволяют задать для этих методов произвольную реализацию. Интерфейс является выражением чистой концепции проектирования, тогда как класс представляет собой смесь проектирования и конкретной реализации. Разработчик интерфейса решает, какие методы должны поддерживаться в классах, *реализующих* данный интерфейс, и что эти методы должны делать.
7. B классе могут присутствовать статические поля, относящиеся к классу в целом, а не к его конкретным экземплярам. По аналогии с ними могут существовать и статические методы, также относящиеся ко всему классу, которые часто называют *методами класса*. Статические методы обычно предназначаются для выполнения операций, специфичных для данного класса, и работают со статическими полями, а не с конкретными экземплярами класса. Для объявления таких методов также используется ключевое слово *static*. Методы могут быть объявлены статическими, так как они работают не с каким-то определенным объектом, но составляют внутри класса группу со сходными функциями. Статический метод не может напрямую обращаться к не статическим членам. При вызове статического метода не существует ссылки на конкретный объект, для которого вызывается данный метод. Это ограничение можно обойти, передавая ссылку на конкретный объект в качестве параметра статического метода. Тем не менее, в общем случае статические методы выполняют свои функции на уровне всего класса, а нестатические методы работают с конкретными объектами.
8. Исключительно полезную концепцию объектно-ориентированного программирования представляют собой **абстрактные классы**. С их помощью можно объявлять классы, реализованные лишь частично, поручив полную реализацию расширенным классам. Абстракция оказывается полезной, когда некоторое поведение характерно для большинства или всех объектов данного класса, но некоторые аспекты имеют смысл лишь для ограниченного круга объектов, не составляющих суперкласса. В Java такие классы объявляются с ключевым словом **abstract**, и каждый метод, не реализованный в классе, также объявляется abstract. Если все, что вам требуется, — это определить набор методов, которые будут где-то поддерживаться, но не предоставлять для них реализации, то вместо абстрактных классов лучше воспользоваться интерфейсами.
9. Суперкласс – класс родитель или любой класс, стоящий в иерархии наследования выше.
10. This – ссылка на текущий экземпляр класса.
11. Super – ссылка на класс родитель.
12. Модификаторы доступа нужны для управления доступом к членам класса.
13. Открытый(**Public**): к членам класса всегда можно обращаться из любого места, в котором доступен сам класс; такие члены наследуются в подклассах. Закрытый (**Private**): доступ к членам класса осуществляется только из самого класса. Защищенный (**Protected**): к данным членам разрешается доступ из подклассов и из функций, входящих в тот же пакет. Такие члены наследуются подклассами. Пакетный: доступ к членам, объявленным **без указания атрибута доступа**, осуществляется только из того же пакета. Такие члены наследуются подклассами пакета.
14. Для интерфейсов. Интерфейс может расширить один или несколько других интерфейсов, добавить к ним новые константы и методы, которые должны быть реализованы в классе, реализующем расширенный интерфейс. Множественное наследование оказывается полезным в тех случаях, когда требуется наделить класс новыми возможностями и при этом сохранить большую часть (или все) старых свойств. Однако при наличии нескольких суперклассов возникают проблемы, связанные с двойственным наследованием.
15. Сборщик мусора - это **программа, которая запускается на виртуальной машине Java и избавляется от объектов, которые больше не используются приложением Java**. Это форма автоматического управления памятью. Когда запускается обычное Java-приложение, оно создает новые объекты, такие как строки и файлы, но по истечении определенного времени эти объекты больше не используются.
16. finalize () – **это специальный метод, который вызывается у объекта перед тем, как сборщик мусора его уничтожит**. Основная цель этого метода – освободить используемые внешние не-Java ресурсы: закрыть файлы, потоки ввода-вывода и т.п. К сожалению, этот метод не оправдывает возложенных на него надежд. Java-машина может отложить уничтожение объекта, как и вызов метода finalize на сколько угодно.
17. Основная цель этого метода – освободить используемые внешние не-Java ресурсы: закрыть файлы, потоки ввода-вывода и т.п. К сожалению, этот метод не оправдывает возложенных на него надежд. Java-машина может отложить уничтожение объекта, как и вызов метода finalize на сколько угодно. Более того, она вообще не гарантирует, что этот метод будет вызван. В куче ситуаций ради «оптимизации» он не вызывается.

|  |
| --- |
| 1. finalize() можно использовать только в двух случаях:    1. Проверка/подчистка ресурсов с логированием.    2. При работе с нативным кодом, который не критичен к утечке ресурсов. 2. finalize() замедляет работу GC по очистке объекта в 430 раз 3. finalize() может быть не вызван |