Делал на отъебись ибо впадлу XD

1. Оболочки

Хранение примитивных типов вместо объектов объясняется стремлением увеличить производительность. Применение объектов для хранения этих значений привело бы к добавлению неприемлемых накладных расходов даже к самым простым вычислениям. Таким образом, примитивные типы не являются частью иерархии объектов и не наследуются от Object. Несмотря на преимущество в производительности, обеспечиваемое примитивными типами, бывают случаи, когда может понадобиться объектное представление: передавать примитивный тип по ссылке в метод нельзя. Чтобы справиться с этими ситуациями, в языке Java предусмотрены оболочки типов, которые представляют собой классы, инкапсулирующие примитивный тип внутри объекта. К оболочкам типов относятся Double, Float, Long, Integer, Short, Byte, Character и Boolean. Все оболочки числовых типов унаследованы от абстрактного класса Number. type typeValue() - получить примитивное значение объекта. В настоящее время для получения объекта оболочки настоятельно рекомендуется использовать один из методов valueOf().
Класс Character является оболочкой для значения char. Вот конструктор класса Character: Character (char ch)
В аргументе ch указывается символ, который будет помещен в оболочку создаваемого объекта Character. Однако, начиная с версии JDK 9, конструктор Character стал нерекомендуемым к употреблению, а начиная с JDK 16, он объявлен устаревшим и подлежащим удалению. В настоящее время для получения объекта Character настоятельно рекомендуется применять статический метод valueOf(char ...).
Процесс инкапсуляции значения внутри объекта называется упаковкой. Процесс извлечения значения из оболочки типа называется распаковкой. Современные версии Java включают два важных средства: автоупаковку и автораспаковку. Автоупаковка - это процесс, с помощью которого примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа всякий раз, когда требуется объект такого типа. Нет необходимости явно создавать объект. Автораспаковка - это процесс, при котором значение упакованного объекта автоматически извлекается (распаковывается) из оболочки типа, когда значение необходимо. Не придется вызывать методы вроде intValue() или doubleValue(). В дополнение к простому случаю присваивания автоупаковка происходит всякий раз, когда примитивный тип должен быть преобразован в объект, а автораспаковка - когда объект должен быть преобразован в примитивный тип. Таким образом, автоупаковка/автораспаковка может быть инициирована при передаче аргумента методу или при возвращении методом значения. Автораспаковка также позволяет смешивать в выражении различные типы числовых объектов. После распаковки значений применяются стандартные повышения и преобразования.
Помимо удобства, автоупаковка и автораспаковка также помогают предотвращать ошибки.

```
1 class UnboxingError {
2     public static void main(String[] args) {
3         Integer iOb = 1000;
4         int i = iOb.byteValue();
5         System.out.println(i);
6     }
7 }
```

В общем случае из-за того, что автоупаковка всегда создает надлежащий объект, а автораспаковка всегда производит корректное значение, процесс не может выдать неправильный тип объекта или значения.

2. Потоки ввода-вывода

Поток данных (stream) - это абстракция, которая либо производит, либо потребляет информацию. Поток связан с физическим устройством посредством системы ввода-вывода Java. Все потоки ведут себя одинаково, даже если фактические физические устройства, с которыми они связаны, различаются. Таким образом, одни и те же классы и методы ввода-вывода могут применяться к разным типам устройств, что означает возможность абстрагирования входного потока от множества различных типов ввода: из дискового файла, клавиатуры или сетевого сокета. В Java определены два типа потоков ввода-вывода: байтовые и символьные. Потоки байтовых данных предлагают удобные средства для обработки ввода и вывода байтов. Они используются, например, при чтении или записи двоичных данных. Потоки символьных данных предоставляют удобные средства для обработки ввода и вывода символов. Они применяют Unicode и, следовательно, допускают интернационализацию. Кроме того, в ряде случаев потоки символьных данных эффективнее потоков байтовых данных. На самом низком уровне все операции ввода-вывода ориентированы на байты. Потоки символьных данных просто обеспечивают удобный и эффективный инструмент для обработки символов.
Потоки байтовых данных определяются с применением двух иерархий классов. Вверху находятся два абстрактных класса: InputStream и OutputStream. Каждый из них имеет несколько конкретных подклассов, которые справляются с отличиями между разными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти. В абстрактных классах InputStream и OutputStream определено несколько ключевых методов, реализуемых другими классами потоков. Двумя наиболее важными из них являются read() и write(), которые выполняют, соответственно, чтение и запись байтов данных.
Потоки символьных данных определяются с помощью двух иерархий классов. Вверху находятся два абстрактных

класса: Reader и Writer. Они обрабатывают потоки символов Unicode. Для каждого из них в Java предусмотрено несколько конкретных подклассов. В абстрактных классах Reader и Writer определено несколько ключевых методов, реализуемых другими классами потоков. Двумя наиболее важными методами считаются read() и write(), которые выполняют, соответственно, чтение и запись символов данных.
Все программы на Java автоматически импортируют пакет java.lang, в котором определён класс System, инкапсулирующий ряд аспектов исполняющей среды. Скажем, с помощью некоторых его методов можно получить текущее время и настройки разнообразных свойств, связанных с системой. Класс System также содержит три предопределённые потоковые переменные: in, out и err. Они объявлены в System как поля public, static и final, т.е. могут использоваться в любой другой части программы без привязки к конкретному объекту System. Поле System.out ссылается на стандартный поток вывода. По умолчанию это консоль. Поле System.in ссылается на стандартный поток ввода, в качестве которого по умолчанию выступает клавиатура. Поле System.err ссылается на стандартный поток вывода ошибок, по умолчанию также являющийся консолью. Однако упомянутые потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода. System.in представляет собой объект типа InputStream, а System.out и System.err - объекты типа PrintStream. Это потоки байтовых данных, хотя обычно они применяются для чтения символов из консоли и записи символов на консоль.
Консольный ввод в Java выполняется путем чтения из System.in. Один из способов получения символьного потока, присоединенного к консоли, предусматривает помещение System.in в оболочку BufferedReader. Класс BufferedReader поддерживает буферизованный поток ввода (он читает данные порциями (а не по одному символу), что делает его эффективным). Ниже приведен часто используемый конструктор:
BufferedReader (Reader inputReader)
Здесь inputReader представляет собой поток, связанный с создаваемым экземпляром BufferedReader, а Reader - абстрактный класс. Но BufferedReader работает с символами (Reader), а System.in — это поток байтов (InputStream). Поэтому между ними нужен преобразователь – InputStreamReader.
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.in — это стандартный поток ввода, который принимает данные с клавиатуры.
InputStreamReader(System.in) — это адаптер, который преобразует поток байтов (System.in) в поток символов, так как System.in работает с байтами.
BufferedReader — это класс, который оборачивает InputStreamReader и добавляет возможность читать данные построчно, а также кэшировать ввод для повышения производительности.
Для чтения символа из BufferedReader предназначен метод read(). Вот версия read (), которая будет применяться:
int read() throws IOException
При каждом вызове метод read () читает символ из потока ввода и возвращает его в виде целочисленного значения. Он возвращает -1 при попытке чтения в конце потока. Как видите, он может сгенерировать исключение IOException.
Для чтения строки с клавиатуры предназначена версия метода readLine(), которая является членом класса BufferedReader со следующей общей формой:
String readLine() throws IOException
Также можно читать через
public static Scanner consoleRead = new Scanner(System.in);

Критерий	**BufferedReader**	**Scanner**
Буферизация	Быстрее (8КБ буфер)	Медленнее (без буфера)
Метод чтения	readLine() (строка)	next(), nextInt(), nextDouble() и т. д.
Работа с типами	Только строки, нужно преобразовывать	Читает числа, строки и другие типы данных напрямую
Гибкость	Хорош для построчного чтения	Удобен для работы с разделителями
Обработка ошибок	Требует IOException	Может выбросить InputMismatchException

Класс PrintStream - это именно тот класс, который используется для вывода на консоль. Когда мы выводим на консоль некоторую информацию с помощью вызова System.out.println(), то тем самым мы задействуем PrintStream, так как переменная out в классе System как раз и представляет объект класса PrintStream, а метод println() - это метод класса PrintStream. Но PrintStream полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода. Для этого PrintStream определяет ряд конструкторов: PrintStream(OutputStream outputStream, boolean autoFlushingOn, String charSet) и так далее.
Параметр outputStream - это объект OutputStream, в который производится запись. Параметр autoFlushingOn при значении true позволяет автоматически записывать данные в поток вывода. По умолчанию этот параметр равен false. Параметр charSet позволяет указать кодировку символов. В качестве источника для записи данных вместо OutputStream можно использовать объект File или строковый путь, по которому будет создаваться файл.
PrintWriter — это удобный класс в Java для записи текстовых данных в файлы, потоки и другие объекты. Он упрощает запись строк и текстовых данных по сравнению с низкоуровневыми потоками, такими как FileOutputStream и OutputStreamWriter, и поддерживает автоматическое добавление символа новой строки. В отличие от OutputStream, который работает с байтами, PrintWriter работает с символами и строками; можно использовать autoFlush, чтобы автоматически сбрасывать буфер после вызова println(); в отличие от FileWriter и BufferedWriter, методы PrintWriter не требуют обработки исключений IOException внутри кода (но их можно проверить через checkError()); можно записывать в файлы, OutputStream, Writer и даже в строковый буфер (StringWriter).
у Java IO есть свои недостатки, так что давай поговорим по порядку о каждом из них:

Блокирующий доступ для ввода/вывода данных. Проблема состоит в том, что когда разработчик пытается прочитать файл или записать что-то в него, используя Java IO, он блокирует файл и доступ к нему до момента окончания выполнения своей задачи.
Отсутствует поддержка виртуальных файловых систем.
Нет поддержки ссылок.
Очень большое количество checked исключений.
Java NIO, или Java Non-blocking I/O (иногда — Java New I/O, “новый ввод-вывод”) предназначена для реализации высокопроизводительных операций ввода-вывода. В отличие от потоков, которые используются в Java IO, Channel является двусторонним, то есть может и считывать, и записывать. Канал Java NIO поддерживает асинхронный поток данных как в режиме блокировки, так и в режиме без блокировки.
Средство, иногда называемое автоматическим управлением ресурсами (automatic resource management - ARM), основано на расширенной версии оператора try. Главное преимущество автоматического управления ресурсами связано с тем, что оно предотвращает ситуации, в которых файл (или другой ресурс) по невнимательности не освобождается после того, как он больше не нужен. Ранее уже объяснялось, что игнорирование закрытия файла может привести к утечкам памяти и прочим проблемам. Как правило, спецификация ресурса представляет собой оператор, который объявляет и инициализирует ресурс, скажем, файловый поток. Он состоит из объявления переменной, в котором переменная инициализируется ссылкой на управляемый объект. Когда блок try заканчивается, ресурс автоматически освобождается.

3. Javadoc

Разработан специальный синтаксис для оформления документации в виде комментариев и инструмент для создания из комментариев документации. Этим инструментом является javadoc, который, обрабатывая файл с исходным текстом программы, выделяет помеченную документацию из комментариев и связывает с именами соответствующих классов, методов и полей. Таким образом, при минимальных усилиях создания комментариев к коду, можно получить хорошую документацию к программе. javadoc — это генератор документации в HTML-формате из комментариев исходного кода Java и определяет стандарт для документирования классов Java. При написании комментариев к кодам Java используют три типа комментариев:
// однострочный комментарий;
/* многострочный комментарий */
/** комментирование документации */
С помощью утилиты javadoc, входящей в состав JDK, комментарий документации можно извлекать и помещать в HTML файл. Утилита javadoc позволяет вставлять HTML тэги и использовать специальные ярлыки (дескрипторы) документирования. Дескрипторы javadoc, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ * игнорируется). Дескрипторы, начинающиеся с фигурной скобки, например @code, называются встроенными и могут применяться внутри описания. Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом.

Дескриптор	Применение	Описание
@author	Класс, интерфейс	Автор
@version	Класс, интерфейс	Версия. Не более одного дескриптора на класс
@since	Класс, интерфейс, поле, метод	Указывает, с какой версии доступно
@see	Класс, интерфейс, поле, метод	Ссылка на другое место в документации
@param	Метод	Входной параметр метода
@return	Метод	Описание возвращаемого значения
@exception имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@throws имя_класса описание	Метод	Описание исключения, которое может быть послано из метода
@deprecated	Класс, интерфейс, поле, метод	Описание устаревших блоков кода
{@link reference}	Класс, интерфейс, поле, метод	Ссылка
{@value}	Статичное поле	Описание значения переменной

Документирование класса, метода или переменной начинается с комбинации символов /** , после которого следует тело комментариев; заканчивается комбинацией символов */. В тело комментариев можно вставлять различные дескрипторы. Каждый дескриптор, начинающийся с символа ’@’ должен стоять первым в строке. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Встроенные дескрипторы (начинающиеся с фигурной скобки) можно помещать внутри любого описания. Каждую новую строку комментария принято начинать со звёздочки.

4. Обощения (дженерики)

В своей основе термин обобщения означает параметризованные типы. Параметризованные типы важны, поскольку они позволяют создавать классы, интерфейсы и методы, где тип данных, с которым они работают, указывается в качестве параметра. Например, с использованием обобщений можно создать единственный класс, который автоматически работает с разными типами данных. Класс, интерфейс или метод, который оперирует на параметризованном типе, называется обобщённым. Благодаря обобщениям все приведения становятся автоматическими и неявными.
T ob; - объявить объект типа T
T - это заполнитель для фактического типа, который указывается при создании объекта Gen. Таким образом, ob будет объектом типа, переданного в T. Например, если в T передается тип String, тогда ob получит тип String.
Так выглядит конструктор класса Gen:

```
1 Gen (T o) {
2     ob = o;
3 }
```

Его параметр o имеет тип T, т.е. фактический тип o определяется типом, переданным в T, когда создается объект Gen. Кроме того, поскольку и параметр o, и переменная-член ob относятся к типу T, при создании объекта Gen они будут иметь один и тот же фактический тип. Параметр типа T также можно использовать для указания возвращаемого типа метода.
Компилятор Java на самом деле не создает разные версии Gen или любого другого обобщенного класса. Хотя думать в таких терминах удобно, в действительности происходит иное - компилятор удаляет всю информацию об обобщенном типе, заменяя необходимые приведения, чтобы код вел себя так, как если бы создавалась конкретная версия Gen. Таким образом, в программе действительно существует только одна версия Gen. Процесс удаления информации об обобщенном типе называется стиранием.
В следующей строке кода переменной iOb присваивается ссылка на экземпляр версии класса Gen для Integer:
iOb = new Gen<Integer> (88);
Обратите внимание, что при вызове конструктора класса Gen также указывается аргумент типа Integer. Причина в том, что объект (в данном случае iOb), которому присваивается ссылка, имеет тип Gen<Integer>. Таким образом, ссылка, возвращаемая операцией new, тоже должна иметь тип Gen<Integer>. В противном случае возникнет ошибка на этапе компиляции. Скажем, показанное ниже присваивание вызовет ошибку на этапе компиляции:
iOb = new Gen<Double> (88.0);
Поскольку переменная iOb относится к типу Gen<Integer>, ее нельзя применять для ссылки на объект Gen<Double>. При объявлении экземпляра обобщенного типа передаваемый параметру типа аргумент типа должен быть ссылочным типом. Примитивный тип вроде int или char применять нельзя. Ключевой момент, который необходимо понять относительно обобщенных типов, связан с тем, что ссылка на одну специфическую версию обобщенного типа несовместима по типу с другой версией того же обобщенного типа. В этот момент вы можете задать себе следующий вопрос. Учитывая, что та же функциональность, что и в обобщенном классе Gen, может быть достигнута без обобщений, просто за счет указания Object в качестве типа данных и применения соответствующих приведений, то в чем польза от превращения Gen в обобщенный класс? Ответ заключается в том, что обобщения автоматически обеспечивают безопасность в отношении типов для всех операций с участием Gen. В процессе они избавляют вас от необходимости вводить приведения типов и проверять типы в коде вручную.
В обобщенном типе допускается объявлять больше, чем один параметр типа. Чтобы указать два или более параметров типов, просто применяйте список, разделенный запятыми. Когда определяется параметр типа, вы можете создать верхнюю границу, объявляющую суперкласс, от которого должны быть порождены все аргументы типов. Цель достигается за счет применения конструкции extends при указании параметра типа:
<T extends superclass>
Таким образом, тип T может быть заменен только суперклассом, указанным в superclass, или подклассами этого суперкласса. Для определения границы можно также применять тип интерфейса. На самом деле в качестве границ разрешено указывать несколько интерфейсов. В добавок граница может включать как тип класса, так и один или несколько интерфейсов. В таком случае тип класса должен быть указан первым. Когда граница содержит тип интерфейса, то допускаются только аргументы типов, реализующие этот интерфейс. При указании привязки, которая имеет класс и интерфейс или несколько интерфейсов, для их соединения используйте операцию &, что в итоге создает пересечения типов. Если контейнер объявлен с wildcard ? extends, то можно только читать значения. В список нельзя ничего добавить, кроме null. Также нельзя прочитать элемент из контейнера с wildcard ? super, кроме объекта класса Object.
<?> - подстановочный символ и используется в обобщениях для обозначения неизвестного типа. Он позволяет создавать более универсальный код, который может работать с различными типами данных без жесткого указания конкретного типа. Подстановочный символ применяется, когда точный тип неизвестен или не имеет значения. Запись List<?> означает "список с элементами неизвестного типа". Такой список можно читать, но нельзя добавлять в него элементы (кроме null), потому что компилятор не знает, какой тип фактически используется. Аргументы с подстановочными знаками могут быть ограничены во многом так же, как и параметр типа. В ограниченном аргументе с подстановочным знаком задается верхняя (<?> extends ...) или нижняя граница (<?> super ...) для аргумента типа, что позволяет сузить диапазон типов объектов, с которыми будет работать метод. Методы внутри обобщенного класса могут использовать параметр типа класса и, следовательно, автоматически становятся обобщенными по отношению

к параметру типа. Однако можно объявить обобщенный метод с одним или несколькими собственными параметрами типов. Более того, можно создавать обобщенный метод внутри необобщенного класса.
Обобщенный метод по имени isin(), который выясняет, является ли объект членом массива. Его можно применять с любым типом объекта и массива при условии, что массив содержит объекты, совместимые с типом искомого объекта.
static <T extends Comparable<T>, V extends T> boolean isin (T x, V[] y)
Класс, реализующий Comparable, определяет объекты, которые можно упорядочивать. Таким образом, требование верхней границы как Comparable гарантирует, что метод isin() может использоваться только с объектами, которые обладают способностью участвовать в сравнениях. Интерфейс Comparable является обобщенным, и его параметр типа указывает тип сравниваемых объектов. V ограничен сверху типом T. Соответственно, тип V должен быть либо тем же самым, что и тип T, либо подклассом T. Такое отношение гарантирует, что метод isin() можно вызывать только с аргументами, которые совместимы друг с другом.
Подобно необобщённым, обобщённые классы могут быть частью иерархии классов. Таким образом, обобщённый класс может выступать в качестве суперкласса или быть подклассом. Ключевая разница между обобщёнными и необобщёнными иерархиями связана с тем, что в обобщённой иерархии любые аргументы типов, необходимые обобщённому суперклассу, должны передаваться вверх по иерархии всеми подклассами. Обобщённый класс не может расширять тип Throwable, что означает невозможность создания обобщённых классов исключений.

5. Коллекции и прочее

Итератор предлагает универсальный стандартизированный способ доступа к элементам внутри коллекции по одному за раз. Таким образом, итератор предоставляет средства перечисления содержимого коллекции. Поскольку каждая коллекция поддерживает итератор, доступ к элементам любого класса коллекции можно получить с помощью методов, определённых в интерфейсе Iterator. В итоге код, который циклически проходит по набору, после внесения небольших изменений также может использоваться, скажем, для циклического прохода по списку. Конкретные классы просто предоставляют разные реализации стандартных интерфейсов.

Таблица 20.1. Интерфейсы, поддерживающие коллекции

Интерфейс	Описание
`Collection`	Позволяет работать с группами объектов; находится на вершине иерархии коллекций
`Deque`	Расширяет `Queue` для поддержки двусторонней очереди
`List`	Расширяет `Collection` для поддержки последовательностей (списков объектов)
`NavigableSet`	Расширяет `SortedSet` для поддержки извлечения элементов на базе поиска с наиболее точным совпадением
`Queue`	Расширяет `Collection` для поддержки специальных типов списков, в которых элементы удаляются только из головы
`Set`	Расширяет `Collection` для поддержки наборов, которые должны содержать уникальные элементы
`SortedSet`	Расширяет `Set` для поддержки сортированных наборов

Интерфейс Collection - это основа, на которой построена инфраструктура Collections Framework, т.к. он должен быть реализован любым классом, определяющим коллекцию. Collection представляет собой обобщенный интерфейс с таким объявлением:
interface Collection<E>
С помощью E указывается тип объектов, которые будут храниться в коллекции. Интерфейс Collection расширяет Iterable, т.е. по всем коллекциям можно проходить в цикле for стиля "for-each".

Метод	Описание
`boolean add(E obj)`	Добавляет `obj` в вызывающую коллекцию. Возвращает `true` в случае успешного добавления `obj`. Возвращает `false`, если `obj` уже является членом коллекции, а дубликаты в коллекции не разрешены
`boolean addAll (Collection <? extends E> c)`	Добавляет все элементы коллекции `c` в вызывающую коллекцию. Возвращает `true`, если коллекция изменилась (т.е. элементы были добавлены), или `false` в противном случае
`void clear()`	Удаляет из вызывающей коллекции все элементы
`boolean contains (Object obj)`	Возвращает `true`, если `obj` является элементом вызывающей коллекции, или `false` в противном случае
`boolean containsAll (Collection<?> c)`	Возвращает `true`, если вызывающая коллекция содержит все элементы коллекции `c`, или `false` в противном случае
`boolean equals (Object obj)`	Возвращает `true`, если вызывающая коллекция и `obj` равны. В противном случае возвращает `false`
`int hashCode()`	Возвращает хеш-код для вызывающей коллекции
`boolean isEmpty()`	Возвращает `true`, если вызывающая коллекция пуста, или `false` в противном случае
`Iterator<E> iterator()`	Возвращает итератор для вызывающей коллекции
`default Stream<E> parallelStream()`	Возвращает поток данных, который использует вызывающую коллекцию в качестве своего источника элементов. Если возможно, то поток данных поддерживает параллельные операции
`boolean remove (Object obj)`	Удаляет из вызывающей коллекции один экземпляр `obj`. Возвращает `true`, если элемент был удален, или `false` в противном случае
`boolean removeAll (Collection<?> c)`	Удаляет из вызывающей коллекции все элементы коллекции `c`. Возвращает `true`, если коллекция изменилась (т.е. элементы были удалены), или `false` в противном случае

Интерфейс List расширяет Collection и объявляет поведение коллекции, в которой хранится последовательность элементов. Элементы можно вставлять или получать к ним доступ по позиции в списке, применяя индекс, который начинается с нуля. Список может содержать повторяющиеся элементы.
Интерфейс Set определяет набор. Он расширяет интерфейс Collection и задает поведение коллекции, которое не допускает дублирования элементов, поэтому метод add() возвращает false, если предпринимается попытка добавления в набор повторяющихся элементов. Интерфейс SortedSet расширяет Set и обеспечивает поведение набора, отсортированного в возрастающем порядке.
Интерфейс Queue расширяет Collection и обеспечивает поведение очереди, которая часто представляет собой список, действующий по принципу "первым пришел - первым обслужен". Интерфейс Deque расширяет Queue и обеспечивает поведение двусторонней очереди. Двусторонние очереди могут функционировать как стандартные очереди "первым пришел - первым обслужен" или как стеки "последний пришел - первым обслужен".
Класс ArrayList реализует динамические массивы, которые по мере необходимости могут расти. Стандартные массивы в Java имеют фиксированную длину. После создания массивы не могут увеличиваться или уменьшаться, а потому необходимо заранее знать, сколько элементов будет содержать массив. Доступ по индексу за 1 (элементы хранятся в непрерывной памяти), поиск линейный (массив неупорядоченный), вставка/удаление в середину линейная (требуется сдвиг элементов).
Связанный список (linked list, реализующий два интерфейса — List и Deque) — это структура данных, в которой объекты расположены в линейном порядке. Однако, в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. У каждого элемента, помимо тех данных, которые он хранит, имеется ссылка на предыдущий и следующий элемент. По этим ссылкам можно переходить от одного элемента к другому. Доступ по индексу линейный (требуется пробежать цепочку ссылок), вставка/удаление за единицу, поиск линейный.
Map<K, V> — это интерфейс в Java, представляющий коллекцию пар «ключ-значение». Ключи в Map уникальны, а каждому ключу соответствует одно значение. Доступ к значениям объектов классов имплементирующих Map осуществляется по ключу (но никак не наоборот — ключ нельзя получить по значению, ведь значения могут быть повторяющимися). Примеры имплементирующих классов: HashMap, LinkedHashMap, TreeMap.
HashMap работает на основе массива Node<K, V>[] (где Node — это внутренний класс, представляющий ключ-значение). Каждый элемент Node<K, V> содержит:
Ключ (key)
Значение (value)
Хеш (hash) — результат работы hashCode()
Ссылку на следующий элемент (next) (если произошла коллизия).
V put(K key, V value) добавляет или обновляет значение по ключу.
V get(Object key) получает значение по ключу.
V remove(Object key) удаляет ключ и его значение.
boolean containsKey(Object key) проверяет, есть ли ключ в HashMap.
boolean containsValue(Object value) проверяет, есть ли значение в HashMap.
int size() возвращает количество элементов.
boolean isEmpty() проверяет, пустая ли HashMap.
void clear() удаляет все элементы.
Set<K> keySet() возвращает Set всех ключей.
Collection<V> values() возвращает коллекцию значений.
Set<Map.Entry<K, V» entrySet() возвращает Set пар ключ-значение.
get, remove и put в среднем работают за 1, в х. с. за n.
LinkedHashMap<K, V> — это наследник HashMap, который сохраняет порядок вставки элементов. Основан на HashMap, но дополнительно использует двусвязный список для хранения порядка элементов и гарантирует порядок вставки (элементы выводятся в том порядке, в котором были добавлены).
Множества представлены интерфейсом Set, который входит в пакет java.util.Set представляет собой коллекцию, не содержащую дубликатов. Основные реализации множества в Java:
HashSet – основан на хеш-таблице (быстрый доступ, но порядок элементов не гарантируется).
LinkedHashSet – основан на хеш-таблице + двусвязном списке (гарантирует порядок добавления элементов).
TreeSet – основан на красно-черном дереве (отсортированное множество, можно добавить компаратор).
boolean add(E e) - true, если элемент был добавлен (то есть, если его не было в множестве), и false, если элемент уже существует в множестве.
boolean contains(Object o)
boolean isEmpty()
Iterator<E> iterator()

```
1 Set<String> set = new HashSet<>();
2 set.add("Apple");
3 set.add("Banana");
4 Iterator<String> iterator = set.iterator();
5 while (iterator.hasNext()) {
6     System.out.println(iterator.next());
7 }
```

Object[] toArray()
<T> T[] toArray(T[] array)

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");

String[] array = set.toArray(new String[0]);
for (String s : array) {
    System.out.println(s);
}
```

boolean containsAll(Collection<?> c)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c) - оставляет в множестве только те элементы, которые содержатся в коллекции c. Все остальные удаляются. Возвращает true, если множество было изменено.
Comparable — это интерфейс, входящий в пакет java.lang и используемый для сортировки классов на основе их естественного порядка. Интерфейс Comparable должен быть реализован в классе, который будет использоваться для сортировки. Этот класс можно сортировать на основе отдельных атрибутов, таких как идентификатор, имя, отдел и так далее.

Класс, реализующий интерфейс Comparable, сравнивает себя с другими объектами. Реализованный класс предлагает пользовательскую реализацию int CompareTo(T var1) для пользовательской сортировки. Метод int CompareTo(T var1) должен быть переопределен таким образом, чтобы:
Он должен возвращать целое положительное значение Positive(+ve), если этот объект больше объекта сравнения.
Он должен возвращать целое отрицательное значение Negative(-ve), если этот объект меньше объекта сравнения.
Он должен вернуть Zero(0), если этот и сравниваемый объект равны.
При использовании класса Comparable можно сортировать только по одному атрибуту.

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    @Override
    public int compareTo(Person other) {
        return this.age - other.age;
    }
}
```

Comparator - интерфейс, который позволяет создавать внешние объекты для сравнения двух объектов. Он не зависит от того, реализует ли класс, который вы сравниваете, интерфейс Comparable. В отличие от Comparable, Comparator реализует метод compare(T o1, T o2), который принимает два объекта для сравнения. Это удобно, если вам нужно несколько способов сравнения объектов или если вы не можете изменить исходный класс (например, если это сторонний класс).

```
public class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}
```

Когда класс реализует интерфейс Comparable, он задает один способ, как его объекты должны сравниваться друг с другом. Это называется естественным порядком. То есть, класс сам решает, по какому критерию будут сравниваться его экземпляры.
Теперь, что касается Comparator. Это интерфейс, который предоставляет внешний способ для сравнения объектов. Он позволяет задать несколько разных критериев сортировки для одного и того же класса. То есть, если нужно изменить порядок сортировки или сравнивать объекты по различным признакам (например, по имени, возрасту или другим полям), мы можем создать различные реализации Comparator.

```
class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}

```

```
class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}
\\Sort
List<Person> people = new ArrayList<>();
people.add(new Person("Alice", 30));
people.add(new Person("Bob", 25));
people.add(new Person("Charlie", 35));

Collections.sort(people, new NameComparator());
Collections.sort(people, new AgeComparator());
```

6. Command

Command — это поведенческий паттерн проектирования, который превращает запросы в объекты. Шаблон проектирования Command оборачивает (упаковывает) объекты, которые выполняют запросы пользователя. Таким образом, одна и та же структура кода может использоваться снова и снова. Каждый процесс, который должен выполняться в шаблоне Command, упаковывается как объект. Обработанные объекты команды могут быть разными. Сначала объект для обработки передается классу Command в качестве параметра, а затем над этим объектом выполняются различные операции. Преимуществом создания команды в виде объектов является то, что при необходимости их можно научить отменять свои действия. В результате можно выполнять операции Redo (Повторить) или Undo (Отменить) и создать функцию макроса.
Шаблон Command включает в себя следующие ключевые компоненты:
Command — это интерфейс или абстрактный класс, который объявляет метод execute(). Этот метод определяет операцию, которую необходимо выполнить.
Concrete Command — это реализация интерфейса Command. Каждая конкретная команда инкапсулирует конкретный запрос и привязывает его к получателю, вызывая соответствующую операцию на получателе.
Receiver (Получатель) — это объект, который выполняет фактическую операцию во время выполнения команды. Receiver знает, как выполнить запрос.
Invoker (Вызывающий) — Invoker отвечает за инициирование выполнения команды. Он содержит ссылку на объект команды и может выполнить команду, вызвав её метод execute().
Client — Клиент отвечает за создание объектов команды, настройку их получателей и связывание их с вызывающими.

7. Singleton

Singleton относится к порождающим паттернам. Его дословный перевод – одиночка. Этот паттерн гарантирует, что у класса есть только один объект (один экземпляр класса) и к этому объекту предоставляется глобальная точка доступа. Из описания должно быть понятно, что этот паттерн должен применяться в двух случаях:
когда в вашей программе должно быть создано не более одного объекта какого-либо класса. Например, в компьютерной игре у вас есть класс «Персонаж», и у этого класса должен быть только один объект, описывающий самого персонажа. Когда требуется предоставить глобальную точку доступа к объекту класса. Другими словами, нужно сделать так, чтобы объект вызывался из любого места программы. И, увы, для этого не достаточно просто создать глобальную переменную, ведь она не защищена от записи и кто угодно может изменить значение этой переменной, и глобальная точка доступа к объекту будет потеряна. Это свойства Singleton'а нужно, например, когда у вас есть объект класса, который работает с базой данных, и вам нужно, чтобы к базе данных был доступ из разных частей программы. А Singleton будет гарантировать, что никакой другой код не заменил созданный ранее экземпляр класса.
Поведение Одиночки на Java невозможно реализовать с помощью обычного конструктора, потому что конструктор всегда возвращает новый объект. Поэтому все реализации Singleton'а сводятся к тому, чтобы скрыть конструктор и создать публичный статический метод, который будет управлять существованием объекта-одиночки и «уничтожать» всех вновь-появляющихся объектов. В случае вызова Singleton'а он должен либо создать новый объект (если его еще нет в программе), либо вернуть уже созданный. Для этого:
1. – Нужно добавить в класс приватное статическое поле, содержащее единственный объект:

```
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
}
```

2. – Сделать конструктор класса (конструктор по-умолчанию) приватным (чтобы доступ к нему был закрыт за пределами класса, тогда он не сможет возвращать новые объекты):

```
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
private LazyInitializedSingleton(){}
}
```

3. - Объявить статический создающий метод, который будет использоваться для получения одиночки:

```
if(instance == null) {
        instance = new LazyInitializedSingleton();
    }
    return instance;
}
```

Ленивую инициализацию (Lazy Initialization) еще называют отложенной инициализацией. Это прием в программировании, когда ресурсоемкая операция (а создание объекта – это ресурсоемкая операция) выполняется по требованию, а не заблаговременно. Что, в общем-то, и происходит в нашем коде Singleton'а. Другими словами, наш объект создается в момент обращения к нему, а не заранее.