1. **Типы данных в Java. Переменные, константы.**

Все типы данных делятся на две большие группы - примитивные и ссылочные.

В состав примитивных типов входят 4 подвида и 8 типов данных.

1. Целые числа (byte, short, int, long)

2. Числа с плавающей точкой (float, double)

3. Логический (Boolean)

4. Символьный (char)

К ссылочным типам данных относятся все классы, интерфейсы, массивы, а также тип данных string.

Особняком Java стоит значение типа NULL - это просто конкретное значение с своим специальным смыслом. Значение NULL используется для ссылочных типов, когда значение не определено.

В языке Java имеются только *динамически* создаваемые объекты.  Переменные объектного типа и объекты являются разными сущностями. Переменные объектного типа являются ссылками, т.е. *аналогами указателей* на динамически создаваемые объекты.

**Объявление переменных.**

Для объявлений переменной используется следующая конструкция:

*тип данных переменной [ = значение], [переменная [ = значение],…];*

В языке Java невозможно явное удаление объекта из памяти. Вместо этого реализована сборка мусора. Традиционным приемом, дающим сборщику мусора указание на необходимость освобождения памяти, является присвоение переменной пустого значения NULL.

**Объявление констант.**

Константы объявляются также, как и переменная, только в начале идет ключевое слово final.

final тип данных переменная [ = значение]

Пример, final int N = 5;

Так же существует три специальные константы:

POSITIVE\_INFINITY - обозначает + бесконечность.

NEGATIVE\_INFINITY - обозначает - бесконечность.

Nan - обозначает отсутствие.

1. **Ввод/вывод данных в Java.**

**Вывод данных в Java.**

Самый простой способ взаимодействия с пользователем в Java - это консоль. На нее можно выводить информацию и считывать с нее некоторые данные. Для взаимодействия с консолью в Java применяется класс System. Его функциональность обеспечивает консольный ввод и вывод данных.

Для создания потоков вывода в классе System определен объект out. В этом объекте определены методы print - выводит данные, println - выводит данные и переводит курсор на новую строку.

\n - перевод на новую строку.   
Для конкатенации используется знак ‘+’.

В одном операторе вывода можно многократно использовать переход на новую строку.

**Форматирование вывода в консоли.**

System.out.printf() – функция форматирования вывода. При использовании со спецификаторами она позволяет добиться нужного формата вывода.

**Специфиакторы:**

• %d - для вывода в консоль целочисленных значений;  
• %x - для 16-ричных чисел;  
• %f - выводятся числа с плавающей запятой;  
• %e - для чисел в экспоненциальной форме (1.3е +01);  
• %c - вывод в консоль одиночного символа;  
• %s - вывод в консоль строковых значений;

\*Специфиакторов и соответствующих им аргументов может быть много. Аргументы последовательно подставляются в спецификаторы.

**Ввод данных.**

Чтобы обеспечить ввод с консоли Java в классе System есть объект in(работать не очень удобно). Поэтому для ввода применяют class Scanner, а он, в свою очередь применяет System.in. Для создания объекта Scanner, в его конструктор передается объект System.in. Сам класс Scanner хранится в пакете java.util, поэтому для работы с ним необходимо выполнить его импорт посредством команды import java.util., но для удобства мы импортируем все классы и пишем import java.util.\*.

import java.util.\*;  
Scanner sc = new Scanner(System.in);  
int n = sc.nextInt();  
System.out.println(n);

**Методы класса Scanner**

next() - для считывания введенной строки до первого пробелаnextLine() - для всех введенной строкиnextInt() - считывает введенное число типа intnextDouble() - для doublenextBoolean() - для BooleannextByte() - для bytenextFloat() - для floatnextShort() - для shorthasNext() - проверяет, было ли введено словоhasNextInt() - проверяет, было ли введено число int hasNextDouble() - проверяет, было ли введено double

1. **Классы в Java. Понятие инкапсуляции. Модификаторы доступа.**

Класс в Java - это шаблон для создания объекта, а сам объект - это экземпляр класса.

Класс определяет структуру и поведение, которые будут совместно использоваться набором экземпляров класса. Класс содержит поля(переменные) и методы. Он составляет основу инкапсуляции в java.

Каждый объект данного класса содержит структуру и поведение, которые определены классом.

***Методы*** используются для описания того, что объект класса умеет делать или что можно с ним делать.

***Поля*** - для описания свойств или характеристик объекта.

**Инкапсуляция** - это принцип, согласно которому внутреннее устройство сущностей нужно объединить в специальной оболочке и скрывать от вмешательств извне. Доступ к объектам возможен через специальные открытые методы, а напрямую обратиться к их содержимому нельзя. Обычно содержимое помещается в специальную оболочку, которая закрывает данные от внешних обращений.

<модификатор>class имя\_класса  
{  
 <поля>  
 <методы>  
}  
<модификатор>class имя\_класса  
{  
 <модификатор> тип имя\_поля\_1;  
 //...  
 <модификатор> тип имя\_поля\_N;  
  
 <модификатор> тип имя\_метода\_1(параметры){  
 // тело метода  
 }  
 //...  
 <модификатор> тип имя\_метода\_N(параметры){  
 //тело метода  
 }  
}

**МОДИФИКАТОРЫ ДОСТУПА**

Они позволяют задать допустимую область видимости для членов класса, т.е контекст, в котором можно употреблять данную переменную или метод. В java имеются следующие модификаторы доступа:

1. Public. Это публичный или общедоступный класс или член класса. Поля и методы, объявленные с модификатором Public видны другим классам из текущего пакета и из внешних пакетов.  
2. Private. Это закрытый класс или член класса. Противоположность Public. Доступен из кода только в этом же классе.  
3. Protected. Такой класс или член класса, который доступен из любого места в текущем классе или в производных классах(наследниках), даже если они находятся в других пакетах).  
4. Модификатор по умолчанию. Отсутствие модификатора у поля или метода предполагает использование модификатора по умолчанию. Такие поля и методы видны всем классам в текущем пакете.

**СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА**

Объявление класса создает только шаблон, но не конкретный объект.

Как правило, классы определяются в разных файлах. Сам файл кода должен называться по имени этого класса. Класс представляет собой новый тип, поэтому мы можем определять переменные, которые представляют данный тип - они определяются в основной программе проекта.

Student s = new Student();

Оператор new выделяет память для объекта Student и затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге выполнения данной операции, будет выделен участок в памяти, где будут храниться данные объекта.

1. **Классы в Java. Конструктор.**

Кроме обычных методов классы могут определять специальный метод, который называется конструктор.

Конструкторы вызываются при создании нового объекта данного класса. Они выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то автоматически создаются конструктор по умолчанию, без параметров. Имя конструктора должно совпадать с именем класса, у конструктора нет типа возвращаемого значения.

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещение методов. Она работает и с конструкторами.

Решение о том, какой конструктор вызывать принимается в соответствии с количеством и типом параметров, указанных в операторе new. Недопустимо объявлять в классе конструкторы с одинаковыми параметрами. В данном случае не учитываются имена формальных параметров. Учитывается только их тип и количество.

**Доступ к методам класса.**

<имя экземпляра класса>.<имя метода>(<параметры>);

**Ключевое слово this.**

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Ключевое слово this в Java используется в теле любого метода для ссылки на текущий объект.

Например, в конструкторе, в котором имена входящих параметров, совпадают с именами полей класса. В этом случае параметры перекрывают область видимости полей класса и невозможно напрямую общаться к полям.

1. **Наследование. Полиморфизм.**

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование.

Классы потомки имеют возможность не только создавать свои собственные поля и методы, но и наследовать поля и методы классов предков.

Классы потомки еще называют подклассами (дочерними классами), а непосредственного предка, данного подкласса называют супер классом.

super(<параметры>)-обращается к конструктору суперкласса;

super.method\_name(<параметры>)-обращается к методу суперкласса;

Ключевое слово extends используется для того, чтобы сообщить транслятору о том, что создаются подкласс класса.

В языке Java для обеспечения хорошей производительности и ясности исходного кода множественное наследование не было реализовано (т.е. модем являться наследником только одного класса).

Производный класс (класс наследник) имеет доступ ко всем методам и полям базового класса, кроме тех, которые определены с модификатором private, даже если базовый класс находится в другом пакете.

**Полиморфизм** - это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

1. **Переопределение(override) и перегрузка(overload) методов**

При необходимости реализации унаследованиях методов, их можно переопределить. В примере переопределен метод print.

@Override  
 void print()  
 {  
 super.print();  
 System.out.print("Faculty"+faculty+"Course"+kurs);  
 }

При вызове переопределения метода, виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется динамическое назначение методов.

Перед переопределяемым методом указывается аннотация @override, но она не обязательна.

При переопределении методов он должен иметь уровень доступа не меньше, чем уровень доступа в базовом классе. Например, если в базовом классе метод имеет модификатор public, то и в подклассе он должен иметь модификатор public.

**Перегерузка.**

Совокупность имени методов и набора формальных параметров называется сигнатурой метода. Java позволяет создавать несколько методов с одинаковыми именами, но разными сигнатурами.

Создание нескольких методов с одним и тем же именем, но с разным набором параметров называется перегрузкой (совмещением).

Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров, передаваемых в методы.

**final static**

Все методы и переменные объектов могут быть замещены по умолчанию.

Если необходимо объявить, что подклассы не имеют права замещать какие-либо методы и переменные класса, их нужно объявить как final.

С final используются имена с большими буквами.

Использование final методов ведет к выигрышу в скорости выполнения кода. Поскольку они не могут быть замещены, то транслятор заменяет их вызовы встроенным кодом. Bytecode копируется непосредственно в код вызываемого метода.

Иногда требуется создать метод, который можно было бы использовать вне контекста какого-то объекта класса: для этого необходимо при его объявлении добавить модификатор static, как происходит и с функцией main.

Статические методы могут непосредственно обращаться только к другим статическим методам. В них не допускается использование ссылок this и super. Переменные также могут иметь тип static, они подобны глобальным переменным.

1. **Абстрактные классы. Абстрактные методы.**

Абстрагирование — это процесс, в ходе которого от пользователя скрываются многие детали реализации, а предоставляются только те, которые необходимы. Это позволяет сделать важным то, что делает объект, а не как он это делает.

Бывают ситуации, когда необходимо определить класс, в котором задана структура какой-то абстракции, но полная реализация всех методов отсутствует. В таких случаях можно с помощью модификатора abstract объявить, что некоторые из методов обязательно должны быть замещены в их подклассах.

Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new.

Нельзя объявлять абстрактными конструкторы и статические методы.

Любой подкласс абстрактного класса должен либо предоставить реализацию всех абстрактных методов своего суперкласса, либо так же быть объявлен как абстрактный.

**Абстрактный метод** в Java — это метод, который объявлен, но в нем не описана логика (нет реализации).

**Абстрактный класс** — это базовый класс, который не предполагает создания своих экземпляров.

Абстрактные классы реализуют один из принципов ООП - полиморфизм.

**Полиморфизм** — это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

Абстрактные классы могут содержать как абстрактные, так и не абстрактные методы, так же поля, конструкторы. Они часто применяются для определения общего интерфейса или поведения, которое может использоваться сразу множеством подклассов.

1. **Интерфейсы**

Интерфейс - конструкция, позволяющая описать методы класса, не описывая их реализацию.

В языке Java интерфейсы состоят из набора констант и методов.

Интерфейс не является классом. Он представляет собой множество требований, предъявляемых классу.

Интерфейсы в Java созданы для поддержки динамического выбора методов во время выполнения программы.

В отличие от классов, у интерфейсов нет переменных представителей, а у методов нет реализации.

Класс может иметь любое количество интерфейсов, при этом необходимо реализовать полный набор методов всех интерфейсов.

Сигнатуры методов класса должны точно совпадать с сигнатурами методов реализуемого интерфейса.

Интерфейс обладает своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных не связанных классах. То есть интерфейсы являются аналогом множественного наследования.

interface имя  
{  
тип\_результата имя\_метода1(список параметра);  
...  
тип\_результата имя\_методаN(список параметра);  
  
final тип имя\_переменной1=значение;  
...  
final тип имя\_переменнойN=значение;  
}   
  
class имя\_класса[extends superclass]  
[implements интерфейс0 [, интерфейс1...]]  
{  
//тело класса  
}

У объявленных в интерфейсе методах отсутствуют операторы тела и не явно считается объявленным public. Переменные могут быть только final и их значениями являются константы.

1. **Исключения. Конструкция try, catch, finally.**

В Java исключения - объект, который описывает некоторую ошибочную ситуацию, произошедшую в какой-то части кода. Когда возникает исключительная ситуация, создаются объект и выбрасывается в метод, вызвавшей ошибки, а далее метод может либо обработать ошибку сам, либо передать ее дальше. Исключения захватывается и обрабатывается.

Исключения, которые генерируются средой, касаются фундаментальных ошибок, нарушают условия среды или правила Java. Исключения, генерируемые вручную, позволяют программе сигнализировать о некоторой аварийной ситуации.

Try  
{  
 //операторы для контроля ошибки  
}  
catch(exType1 e1)  
{  
 //Обработка исключений exType1  
}  
catch(exType2 e2)  
{  
 //Обработка исключений exType2  
}  
finally  
{  
 //операторы перед возвратом из try-блока  
}

Те операторы, которые нужно контролировать, находятся в блоке try, если в этом блоке произошла исключительная ситуация, то говорят, что выброшены исключения.

Далее, с помощь catch можно перехватить это исключение и обработать его каким-либо способом.

Любой метод, который выбрасывает исключения необходимо объявить, как throws method.

Все классы исключений являются подклассами встроенного класса Throwable. У него есть 2 подкласса:

1. Exception (которым отлавливаем).

2. Error (автоматически генерируется и разрушается средой).

В некоторых ситуациях на одном участке кода может возникнуть несколько исключений и для их захвата необходимо написать несколько catch блоков, каждый для захвата своего типа исключений. Когда исключения выбрасываются, то catch блоки просматриваются по порядку и первый блок, тип которого совпадет с исключением, будет выполняться, после этого остальные catch блоки обходятся.

Если используется множественный catch, то необходимо учитывать последовательности catch блоков, что подклассы исключений должны следовать всегда перед суперкассами, потому что catch, который использует super класс, будет перехватывать как свои исключения, так и исключения своих подклассов. Соответственно catch с подклассом никогда не будет достигнут, а недостижимый код в Java является ошибкой.

Когда исключение выбрасывается, путь прохождения потока через тело метода становится нелинейным, для того чтобы некоторые операторы в любом случае были выполнены-существует блок finally, операторы в блоке finally выполняются в любом случае, вне зависимости от исключения.  
Этот блок будет выполняться каждый раз, когда метод собирается вернуться к вызывающей программе внутри try-catch. Он выполнится даже если нет catch для соответствующего исключения.

1. **Исключения. Конструкция throw, throws.**

В Java исключения - объект, который описывает некоторую ошибочную ситуацию, произошедшую в какой-то части кода. Когда возникает исключительная ситуация, создаются объект и выбрасывается в метод, вызвавшей ошибки, а далее метод может либо обработать ошибку сам, либо передать ее дальше. Исключения захватывается и обрабатывается.

Исключения могут генерироваться исполнительной системой Java или программный код может генерировать их вручную. Те исключения, которые генерируются средой, касаются фундаментальных ошибок, нарушают условия среды или правила Java. А исключения, генерируемые вручную, позволяют программе сигнализировать о некоторой аварийной ситуации.

Для отработки исключений в Java есть 5 ключевых слов: try, catch, finally, throw, throws.

Для того, чтобы написать свое исключение необходимо написать оператор throw и тип исключения должен быть или классом Throwable или его подклассом.  
После оператора throw поток выполнения останавливается и просматривается ближайший блок try для поиска catch, соответствующего типу исключения. Если соответствие найдено, то управление передается этому оператору. Если не найдено, то просматривается следующее исключение try catch.  
Если соответствующий catch не будет найден, то программа установит обработчик по умолчанию.

Все встроенные классы исключения имеют два конструктора. Один конструктор без параметров, а второй с одним параметром строкового типа, которая (строка) описывает исключения. Эта строка выводится с помощью метода get message() из класса throwable.  
Если метод может выбросить исключение, которое он не обрабатывает, он должен оповестить другие методы об этом. Для этого необходимо включить ключевое слово throws в определении метода и далее перечислить все типы исключений, которые может выбрасывать метод. Это необходимо для всех типов исключений, кроме run time accept ion и его подклассы.

1. **Исключения. Создание собственных классов-исключений.**

В Java исключения - объект, который описывает некоторую ошибочную ситуацию, произошедшую в какой-то части кода. Когда возникает исключительная ситуация, создаются объект и выбрасывается в метод, вызвавшей ошибки, а далее метод может либо обработать ошибку сам, либо передать ее дальше. Исключения захватывается и обрабатывается.

Исключения могут генерироваться исполнительной системой Java или программный код может генерировать их вручную. Те исключения, которые генерируются средой, касаются фундаментальных ошибок, нарушают условия среды или правила Java. А исключения, генерируемые вручную, позволяют программе сигнализировать о некоторой аварийной ситуации.

Для отработки исключений в Java есть 5 ключевых слов: try, catch, finally, throw, throws.

Можно создавать свои исключения для обработки потенциальных ошибок в программе. Для этого необходимо определить подкласс производный от класса Exception, который в свою очередь является подклассом Throwable. При этом, в создаваемый подкласс не обязательно включать какие-либо методы, сам факт его существования позволяет использовать в качестве исключения.  
В классе Exception не определены новые методы, он наследует методы Throwable. Соответственно, все исключения содержат методы класса Throwable, но мы их можем переопределить.

1. **Класс Object, классы оболочки, методы классов-оболочек.**

Фактически, все классы — это наследники класса Object. Все классы, даже те, которые мы добавляем в свой проект являются неявными производными от класса Object. Поэтому, все типы и классы могут реализовывать те методы, которые определены в классе Object.  
**1 метод. toString().**  
Метод toString() служит для получения представления данного объекта в виде строки. При попытке вывести строковое представление какого-либо класса, как правило, будет выводиться полное имя класса. Соответственно, метод toString() стоит переопределять для реализуемого класса.  
**2 метод. getClass()**  
Позволяет получить тип данного объекта.  
**3 метод. equals()**Метод сравнивает два объекта на равенство. Он в качестве параметра принимает объект любого типа, который затем приводим к текущему. Объекты считаются равными, если они принадлежат одному классу и имеют одинаковые значения полей. По умолчанию этот метод сравнивает адреса объектов в памяти. Соответственно, его нужно переопределять.  
**Классы оболочки.**  
Классы оболочки в Java являются объектные представления 8 примитивных типов. Все классы оболочки являются неизменными и finally. Для каждого примитивного значения есть свой класс. Имя класса совпадает с именем примитивного значения. Исключения integer, character.  
Конвертация примитивного типа в объектный называется упаковкой и производится автоматически компилятором, а обратное преобразование называется распаковкой и также происходит автоматически.  
Все классы оболочки (обвертки) числовых типов имеют переопределенный метод equals, сравнивающий примитивные значения этих объектов.  
**Методы класс оболочек**  
**Метод valueOf().**Этот метод представляет собой второй способ создания объектов оболочек. Метод перегруженный и для каждого класса существенно два варианта - на вход значение соответствующего типа, либо значение типа string, а в целочисленные классы byte, short, integer, long добавлен еще один метод, в который можно передать строку в любой системе исчисления, а вторым параметром - основание системы исчисления.  
**Метод parse\*\*\*, toString()**  
Позволяет преобразовать строку в соответствующее примитивное значение.  
В классе double это метод parseDouble()  
И тд.  
Разница с методом valueOf() состоит в том, что valueOf() возвращает объект …… (дописать)

1. **Дженерики. Ограничения дженериков.**

Обобщения - то особые средства языка Java для реализации обобщенного программирования: особого подхода к описанию данных и алгоритма, позволяющего работать с различными типами данных без изменения их описания. Дженерики еще называются параметризированными типами.  
Дженерики позволяют определять классы, интерфейсы и методы, в которых тип данных указывается с помощью параметров. Таким образом, можно написать один параметризированный класс и используя его работать с разными типами данных.  
Дженерики могут работать только с объектами.  
Пример:  
**class имя\_класса <Т>{…}**  
С помощью буквы Т в определении класса указывается, что данный тип Т будет использоваться в классе, он называется универсальным параметром, тк вместо него можно поставить любой тип. Конкретный тип подставляется при создании экземпляра класса. Буква Т выбрана условно, это может быть любая буква или набор символов.  
При определении переменной данного класса и создания объекта необходимо указывать какой тип мы будем использовать. При этом надо учитывать, что Дженерики работают только с объектами, они не работают с примитивными типами. Вместо примитивных типов необходимо использовать классы-обертки.  
В методах параметризированного класса можно использовать параметр типа, а следовательно они становятся параметризированные.  
Можно объявить метод, в котором непосредственно используется один или несколько параметров типа. Более того, можно определить параметризированный метод, входящий в непараметризированный класс.  
**Ограниченные типы.**  
Указывая параметр типа, можно наложить ограничения в виде верхней границы, где объявляется суперкласс, от которого должны быть унаследованы все аргументы типов. В этом случае используется следующая конструкция:  
**class Gen <T extends Superclass>**  
Параметр Т может быть заменен только указанным суперклассом или его подклассом.  
**Ограничения дженериков.**  
1. Нельзя создавать экземпляр по параметру типу, ни обычный объект, ни массив.  
2. Нельзя создавать массив специфических для типа обобщенных ссылок.

3. Нельзя создавать массив из объектов, созданных на основе обобщенного класса. То есть один экземпляр обобщенного класса создать можно, а массив таких классов нельзя.  
4. Нельзя создавать обобщенные статические переменные и методы, но объявить статические обобщенные методы со своими параметрами типа можно.

1. **Коллекции, интерфейсы Iterator, ListIterator.**

Для набора данных в java предназначены массивы, однако у них фиксированная длина и это не удобно. Эту проблему решают коллекции.   
Преимущество коллекции не только в гибком размере, но и в том, что классы коллекций реализуют различные алгоритмы и структуры данных.  
Коллекция – это название для нескольких похожих структур данных, это сложные типы, которые располагаются в пакете java.util.   
В основе всех коллекций лежит применение какого-либо интерфейса, который определяет базовый функционал коллекции.  
Collection – базовый интерфейс для всех коллекций и др интерфейсов коллекций  
Queue – очередь, наследует интерфейс collection, представляет функционал для структур очереди   
Deque – представляет функционал для двунаправленных очередей  
List – тоже наследует интерфейс collection и представляет простые списки  
Set – используется для хранения множества уникальных объектов  
Sortedset – расширяет интерфейс set для сортированных коллекций  
Map – предназначен для создания структур в виде словаря, где каждый эл-т имеет ключ и значение. Он не наследуется от интерфейса collection.

**Интерфейс collection**

Public interface Collection <E> extend Iterable <E>  
{   
 //определение методов  
}

**Методы интерфейса Collection:**

* Boolean add(E item) – добавляет в коллекцию объект item, при удачном добавлении возвращает true, при неудачном false
* Boolean addAll(Collection<?extends E> col) – добавляет в коллекцию все эл-ты из коллекции col
* Void clear() – удаляет все эл-ты из коллекции
* Boolean contains(Object item) – возвращает true если объект item есть в коллекции, иначе false
* Boolean isEmpty() – возвращает true если коллекция пуста, иначе false
* Iterator<E>iterator() – возвращает объект iterator для обхода элементов
* Boolean remove (Object item) – удаляет объект item и возвращает true если объект был удален, иначе false
* Boolean removeAll(Collection<?>col) – удаляет все эл-ты коллекции col из текущей коллекции и возвращает true если получилось, иначе false
* Boolean retainAll(Collection<?> col) – удаляет все эл-ты из текущей коллекции за исключением того, что есть в сol
* Int size() – возвращает размер
* Object[] toArray() – возвращает массив содержащий все эл-ты коллекции

Все эти методы имеются в интерфейсе collection, следовательно реализуются всеми коллекциями, поэтому общий принцип работы с коллекциями одинаковый.

**Итераторы**Одним из ключевых моментов работы с коллекциями является iterator<E>iterator().  
Этот метод возвращает объект реализующий интерфейс iterator.

Public interface Iterator<E>  
{  
 E next();  
 Boolean hasNext();  
 Void remove();  
}

С помощью вызова метода next(); можно получить следующий эл-т и перейти на него.

С помощью hasNext(); можно узнать есть ли следующий эл-т и не достигнут ли конец коллекции. Если он достигнут, то метод next(); выбросит исключение.

Интерфейс iterator представляет ограниченный функционал. Больший набор методов представляет интерфейс Listiterator он используется классами, реализующими интерфейс лист.

**Методы интерфейса Listiterator**

* void add(E obj) – вставляет объект obj перед новым элементов, который должен быть возвращен след. вызовом next().
* Boolean hasNext() – возвращает true, если в коллекции имеется следующий элемент, иначе false
* Boolean hasPrevious() – возвращает true, если в коллекции имеется предыдущий элемент, иначе false
* E next() – возвращает текущий элемент и переходит к следующему, если такого нет, то генерируется исключение NoSuchElementException
* E previous() – возвращает текущий элемент и переходит к предыдущему, если такого нет, то генерируется исключение NoSuchElementException
* int nextIndex() – возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка.
* int previousIndex() – возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1.
* void remove() – удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов next() или previous(), иначе будет сгенерировано исключение illigalStateExceprion

1. **Класс ArrayList и интерфейс List.**

Для создания простых списков используется интерфейс list.

Методы класса ArrayList и интерфейса List.

* Void add(int indext, E obj) – добавляет в список по индексу index объект obj.
* Boolean addAll(int index, Collection<? Extends E> col) – добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе false.
* E get(int index) – возвращает объект из списка по индексу index.
* Int (last)indexOf(Object obj) – возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1.
* ListIterator<E> listIterator() – возвращает объект ListIterator для обхода элементов списка.
* Static <E> List<E> if() – создает из набора элементов объект List.
* E remove(int index) – удаляет объект из списка по индексу index, возвращая при этом удаленный объект.
* E set(int index, E obj) – присваивает значение объекта obj элементу, который находится по индексу index.
* Void sort(Comparator<? Super E> comp) – сортирует список с помощью компаратора comp.
* List<E> subList(int start, int end) – получает набор элементов, которые находятся в списке между индексами start и end.

По умолчанию в java есть встроенная реализация этого интерфейса – ArrayList.

ArrayList – представляет собой список, аналогичный массиву, но не с фиксированным кол-ом эл-ов, он наследует свою функциональность от класса Abstract List и реализует интерфейс List.

Конструкторы ArrayList

* ArrayList() – создает пустой список.
* ArrayList(Collection <? Extends E> col) – создает список, в который добавляются все элементы коллекции col.
* ArrayList(int capacity) – создает список, который имеет начальную емкость capacity.

При добавлении эл-ов фактически происходит перераспределение памяти – создание нового массива и копирование в него эл-ов из старого.

1. **Обобщенный класс LinkedList.**

Обобщенный класс LinkedList представляет структуру данных в виде связанного списка и реализует интерфейсы list, deque, queue и имеет след конструкторы:

* LinckedList () – создает пустой список.
* LinckedList (Collection <? Extends E> col) – создает список, в который добавляются все элементы коллекции col.

LinkedList содержит все методы из интерфейсов list, deque, queue

LIST:

* Void add(int indext, E obj) – добавляет в список по индексу index объект obj.
* Boolean addAll(int index, Collection<? Extends E> col) – добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе false.
* E get(int index) – возвращает объект из списка по индексу index.
* Int (last)indexOf(Object obj) – возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1.
* ListIterator<E> listIterator() – возвращает объект ListIterator для обхода элементов списка.
* Static <E> List<E> if() – создает из набора элементов объект List.
* E remove(int index) – удаляет объект из списка по индексу index, возвращая при этом удаленный объект.
* E set(int index, E obj) – присваивает значение объекта obj элементу, который находится по индексу index.
* Void sort(Comparator<? Super E> comp) – сортирует список с помощью компаратора comp.
* List<E> subList(int start, int end) – получает набор элементов, которые находятся в списке между индексами start и end.

**Методы :**

addFirst()/ offerFirst() – добавляет эл-ты в начало списка

addLast()/ offerLast() – добавляет эл-ты в конец списка

removeFirst()/ pollFirst() – удаляет первый эл-т из начала списка

removeLast()/ pollLast() – удаляет последний эл-т из конца списка

getFirst()/ peekFirst() – получает первый эл-т

getLast()/ peekLast() – получает последний эл-т

1. **Интерфейс Set.**

Интерфейс set расширяет интерфейс collection и представляет набор уникальных эл-ов. Set не добавляет новых методов, а только вносит изменения в унаследованных.

Например, метод add добавляет эл-т в коллекцию, возвращает true если в коллекции еще нет такого эл-та.

Обобщенный класс HashSet() представляет собой хеш-таблицу. Реализует интерфейс set. Хеш-таблица представляет такую структуру данных, в которой все эл-ты имеют уникальный хеш-код, он позволяет идентифицировать объект в таблице.

Имеет след конструкторы:

* HashSet() – создает пустой список.
* HashSet(Collection<? Extends E> col) – создает хеш-таблицу, в которую добавляет все элементы коллекции col.
* HashSet(int capacity) – параметр capacity указывает начальную емкость таблицы, которая по умолчанию равно 16.
* HashSet(int capacity, float koef) - параметр koef или коэффициент заполнения, значение которого должно быть в п ределах 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение.

Еще также есть TreeSet и LinkedHeahSet они также относятся к семейству множеств.

В TreeSet эл-ты отсортированы, а в LinkedHeahSet эл-ты хранятся в порядке их добавления.

1. **Интерфейс Map.**

Представляет собой словарь, где каждый эл-т реализует пару ключ/значение, при этом все ключи уникальны в рамках map. Такие коллекция облегчают поиск эл-ов, если известен ключ. Интерфейс map не расширяет интерфейс collection.

**Методы**:

* Void clear() – очищает коллекцию.
* Boolean containsKey(Object k) – возвращает true, если коллекция содержит ключ k.
* Boolean containsValue(Object v) – возвращает true, если коллекция содержит значение v.
* Set<Map.Entry<K,V>> entrySet() – возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry.
* Boolean equals(Object obj) – возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj.
* Boolean isEmpty() – возвращает true, если колекция пуста.
* V get(Object k) – возвращает значение объекта, ключ которого равен k. если такого элемента не окажется, то возвращает значение null.
* V getOrDefault(Object k, V defaultValue) – возвращает значение объекта, ключ которого равен k. если такого объекта не окажется, то возвращает значение defaultValue.
* V put(K k, V v) – помещает в коллекцию новый объект с ключом k и значением v. если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если ключа еще не было в коллекции, то возвращается значение null.
* V putIfAbsent(K k, V v) – помещает в коллекцию новый объект с ключом k и значенем v, если в коллекции еще нет элемента с подобным ключом.
* Set<K> keySet() – возвращает набор всех ключей отображения.
* Collection<V> values() – возвращает набор всех значений отображения.
* Void putAll(Map<? Extends K, ? extends V> map) – добавляет в коллекцию все объекты из отображения map.
* V remove(Object k) – удаляет объект с ключом k.
* Int size() – возвращает кол-во элементов в коллекции.

Чтобы положить эл-т(объект) в коллекцию используется метод put, чтобы получить по ключу – метод get.

Обобщенный интерфейс Mep.Entry(K,V) также представляет объект с ключом и значением, и имеет след методы:

* Boolean equals(Object obj) – возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему.
* K getKey() – возвращает ключ объекта отображения.
* V getValue() – возвращает значение объекта отображения.
* V setValue(V v) – устанавливает для текущего объекта значение v.
* Int hashCode() – возвращает хеш-код данного объекта.

При переборе объектов данного отображения будем работать с ключами и значениями, базовым классом является abstruct Map, который реализует большую часть методов интерфейса map.

Перебор значений – самая частая операция, все пары хранятся во внутреннем интерфейсе map.entry, а чтобы их получить нужно использовать метод entry.set(), он возвращает множество пар, которые можно перебрать в цикле.

1. **Интерфейсы Comparable, Comparator. Сортировки.**

***Интерфейсы Comparable***

При сравнении эл-тов определенного типа определены операции сравнения, однако сравнить два объекта на больше, меньше, равно – невозможно; объекты имеют свои поля и поэтому сравниваются с помощью след методов.

Interface Comparable<E>  
{  
 Int compareTo(E item);  
}

Интерфейс Comparable содержит метод compareTo, который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отриц число, то текущий объект будет располагаться перед тем, который передается через параметр, если метод возвращает полож число, то наоборот, а если 0, то объекты равны.

***Интерфейс Comparator***

- есть еще более гибкий способ, предполагающий использование интерфейса Comparator, у него метод campare();

Public interface Comparator<E>  
{  
 Int compare(E a, E b);  
 //остальные методы  
}

Метод compare также возвращает числовое значение, если оно отриц, то объект а предшествует объекту b, а иначе наоборот, если метод возвращает 0, то объекты равны.

Для применения интерфейса Сomparator надо сначала создать класс Сomparator, который реализует этот интерфейс

**Сортировка по нескольким критериям**

Можно применить сразу несколько comporator-ов по принципу приоритета.

Интерфейс comporator-а определяет специальный метод по умолчанию thenComparing, который позволяет использовать цепочки comporator-ов для сортировки.

1. **Лямбда-выражения.**

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызывать в различных метах программы.

Основы лябмбда выражения составляет лямбда оператор, который представляет стрелочку, этот оператор разделяет лямбда выражения на две части: левая часть содержит список параметров выражения, а правая представляет тело лямбда-выражения, где выполняются все действия. Лямбда-выражение не выполнится само по себе, оно образует реализацию метода, определенного в функциональном интерфейсе, при этом важно, что функциональный интерфейс должен содержать один метод без его реализации.

Локальные переменные уровня метода мы также можем использовать в лямдах, но изменить их значение нельзя. Если мы попробуем это сделать, то среда разработки(NetBeans) может выдать ошибку и то, что такую переменную надо пометить с помощью ключевого слова final, то есть сделать константой: final int n=70;. Однако это необязательно. Более того мы не сможем изменить значение переменной, которая используется в лямда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути, она является константой.

Лямда-выражение может использовать переменные, которые объявлены в более общей области видимости – на уровне класса или метода, в котором лямда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямдах.

**Блоки кода в лямда-выражениях**

Существуют два типа лямда-выражений: однострочное выражение и блок код. Блочные выражения обрамляются фигурными скобками. В блочных лямда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямда-выражение должно возвращать значение, то явным образом применяется оператор return.

Чтобы объявить и использовать лямбда-выражение основная программа разбивается на этапы:

1. Определение ссылки на функциональный интерфейс: Operationable operation;
2. Создание лямба-выражения: operation = (x,y)->x+y;  
   Причем параметры лямда выражения соответствуют параметрам единственного метода интерфейса Operationable, а результат соответствует возвращаемому результату метода интерфейса. При этом нам не надо использовать ключевое слово return для возврата результата из лямда-выражения.
3. Использование лямда-выражения в виде вызова метода интерфейса: int result = operation.calculate(10,20);
4. **Функциональные интерфейсы.**

Функциональным считается интерфейс с одним не реализованным (абстрактным) методом.

Рассмотрим основные из этих интерфейсов:

* Predicate<T>
* Consumer<T>
* Function<T,R>
* Supplier<T>
* UnaryOperator<T>
* BinaryOperator<T>

**UnaryOperator<T>**

UnaryOperator<T> принимает в качестве параметра объект типа Т, выполняет над ними операции и возвращает результат операции в виде объекта типа Т:

public interface UnaryOperator<T> {  
 T apply(T t);  
}

**Function<T,R>**

Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа Т к объекту типа R:

Public interface Function<T,R>{  
 R apply(T t);  
}

**Predicate<T>**

Функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращает значение true. В качестве параметра лямда-выражение принимает объект типа Т:

Public interface Predicate<T>{  
 Boolean test(T t);  
}

**BinaryOperator<T>**

BinaryOperator<T> принимает в качестве параметра два объекта типа Т, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа Т:

Public interface BinaryOperator<T>{  
 T apply(T t1,T t2);  
}

**Consumer<T>**

Consumer (с англ. — “потребитель”) — функциональный интерфейс, который принимает в качестве входного аргумента объект типа T, совершает некоторые действия, но при этом ничего не возвращает.

public interface Consumer<T> {

void accept(T t);  
}

**Supplier<T>**

Supplier (с англ. — поставщик) — функциональный интерфейс, который не принимает никаких аргументов, но возвращает некоторый объект типа T:

public interface Supplier<T> {

T get();

}

1. **Оконные приложения. Класс JComponent.**

Для создания оконных приложений в java используются библиотеки swing и AWT.

В библиотеке swing хранятся эл-ты для создания пользовательских независимых приложений, а в AWT содержится платформенно-ориентированные средства для работы с окнами.

Окно верхнего уровня в java наз-тся фреймом, для такого окна в AWT содержится класс Frame, для swing JFrame.

Класс JFrame расширяет класс Frame и представляет собой один из немногих компонентов библиотеки swing, которые не отображаются на холсте, а кнопки, строки заголовков и др эл-ты пользовательской системы реализуются с помощью пользовательской оконной системы, а не библиотеки swing.

Классы библиотеки Swing находятся в пакете javax.swing

По умолчанию фрейм имеет размер 0х0 пикс., обязательно нужно определить, что произойдет если пользователь закроет фрейм.

Создание фрейма не приводит к его автоматическому появлению на экране, при создании все фреймы невидимы.

**методы класса JFrame**

- dispose() – закрывает окно и освобождает все системные ресурсы при его создании

- setIconImage(Image image) – позволяет установить иконку окна

- setTitle (String s) – устанавливает заголовок

- setResizable(Boolean b) – дает возможность изменять размер окна если true

- setVisible(Boolean b) – устанавливает видимость окна на экране

- setDefaultCloseOperation(int d) – устанавливает события кот происходят при закрытии окна: DO\_NOTHING\_ON\_CLOSE, HIDE\_ON\_CLOSE, DISPOSE\_ON\_CLOSE, EXIT\_ON\_CLOSE; по умолчанию это значение равно HIDE\_ON\_CLOSE

Большинство методов, которые позволяют изменять размер и форму окна находятся в классах Component и Window

В языке java фреймы предназначены для того, чтобы быть контейнерами для компонентов, например меню или др пользовательского интерфейса. Обычно рисунки выводят в другой компоненте – панели, которые добавляются к фрейму.

Класс JFrame состоит из нескольких слоев, каждый из которых представляет область: корневая область, область слоя и прозрачная область нужны лишь для создания меню и области содержимого. Ключевой здесь является область содержимого и при разработке фрейма компоненты добавляются в нее.

Есть еще один компонент библиотеки Swing это панель – класс JPanel

Панель добавляется к фрейму и так же может служить контейнером либо на ней можно осуществлять рисование.

1. **Оконные приложения. JLabel, JButton, JTextField.**

Для создания оконных приложений в java используются библиотеки swing и AWT.  
В библиотеке swing хранятся эл-ты для создания пользовательских независимых приложений, а в AWT содержится платформенно-ориентированные средства для работы с окнами.  
Окно верхнего уровня в java наз-тся фреймом, для такого окна в AWT содержится класс Frame, для swing JFrame.  
Класс JFrame расширяет класс Frame и представляет собой один из немногих компонентов библиотеки swing, которые не отображаются на холсте, а кнопки, строки заголовков и др эл-ты пользовательской системы реализуются с помощью пользовательской оконной системы, а не библиотеки swing.  
Классы библиотеки Swing находятся в пакете javax.swing  
Все компоненты в библиотеки Swing, такие как JComboBox, JButton и тд унаследованы от класса JComponent, которые можно добавить в классы контейнера.  
Контейнерные классы – это классы, в которые могут быть другие компоненты, поэтому для создания графического интерфейса нам нужен как минимум один контейнерный объект.  
***Есть 3 типа контейнеров:***

1. Панель – это чистый контейнер, а не окно само по себе, единственная цель панели состоит в том, чтобы организовывать компоненты на окне.
2. Рамка – полностью функционирующее окно с его заголовком и значками.
3. Диалог – можно рассматривать как всплывающее окно, кот появляется, когда сообщение должно быть отображено. Это не полностью функционирующее окно как рамка.

Метка JLable размещает статический текст, имеет несколько **конструкторов**, в которых можно задать текст и значок метки:

* JLabel(String text)
* JLabel(Icon image)
* JLabel(String text, Icon image, int align) – LEFT, RIGHT, CENTER

**Методы JLabel:**

* getText() – получить текст.
* setText(String s) – установить текст.
* getIcon() – получить иконку.
* setIcon(Icon image) – установить иконку.
* setVerticalAlignment(int align) – вертикальное выравнивание.
* setHorizonatlAlignment(int align) – горизонательное выравнивание.

**Пример** добавления JLable:

L1=new JLable(“1-ое слогаемое”); //создаем метку

Add(l1); //добавляем на патель

**Кнопка JButton**

**Конструкторы:**

* JButton()
* JButton(String text)
* JButton(Icon icon)
* JButton(String text, Icon icon)

**Пример** добавления JButton:  
JButton button;  
button = new JButton(“Вычислить”);

К кнопке может быть привязан обработчик события – нажатие кнопки.

**Текстовое поле JTextField.** Используется для ввода небольших по объему данных, содержит при себе текст

**Конструкторы:**

* JTextField(int columns)
* JTextField(String text)
* JTextField(string text, int columns)

**Методы JTextField:**

* setText(String text)
* getText()
* getText(int offset, int length)
* setHorizontalAllignment(int align) – LEFT, CENTER, RIGHT

1. **Оконные приложения. JToggleButton, JCheckBox, JRadioButton, JTextArea.**

Для создания оконных приложений в java используются библиотеки swing и AWT.

В библиотеке swing хранятся эл-ты для создания пользовательских независимых приложений, а в AWT содержится платформенно-ориентированные средства для работы с окнами.

Окно верхнего уровня в java наз-тся фреймом, для такого окна в AWT содержится класс Frame, для swing JFrame.

Класс JFrame расширяет класс Frame и представляет собой один из немногих компонентов библиотеки swing, кот не отображаются на холсте, а кнопки, строки заголовков и др эл-ты пользовательской системы реализуются с помощью пользовательской оконной системы, а не библиотеки swing.

Классы библиотеки Swing находятся в пакете javax.swing

**Компаненты JToggleButton, JCheckBox, JRadioButton**

* JToggleButton(String text, Icon icon, Boolean selected)
* setSelected(Boolean selected) – перенос кнопки в требуемое состояние.
* isSelected() - возвращает true если кнопка выбрана и false в обратном случае.

Компонент JToggleButton представляет собой кнопку, которая может находится в двух состояниях – нажатом и отпущенном; когда пользователь щелкает по такой кнопке мышкой, он изменяет ее состояние. Основной конструктор создает кнопку с текстом, иконкой и положением.

От класса JToggleButton унаследован класс JCheckBox – флажок;

Класс имеет те же конструкторы и методы, но выглядит не как кнопка, а как флажок.

Аналогичным образом ведет себя класс JRadioButton – переключатель.

Классы JCheckBox, JRadioButton ведут себя одинаково, но принято их использовать в разных ситуациях.

JRadioButton предлагает выбор единственной альтернативы из нескольких, объекты объединяются в одну группу и при выборе одного эл-та предыдущий выбранный переводится в состояние не выбранный, а JCheckBox предназначен для множественного выбора, для того чтобы получить такое поведение используется ButtonGroup – взаимно исключающая группа. Если добавить в этот контейнер несколько JRadioButton, то выбранный всегда будет только один.

**ButtonGroup:**

* Add(AbstractButton button)
* getElements() Enumeration

**JTextArea** – область для ввода текста, является потомком JTextFild и наследует все его методы, позволяет ввести не одну строку, а несколько

**JTextArea:**

* JTextArea(int rows, int columns)
* append(String text)
* insert(String text, int position)

1. **Оконные приложения. Классы адаптеры.**

Для создания оконных приложений в java используются библиотеки swing и AWT.

В библиотеке swing хранятся эл-ты для создания пользовательских независимых приложений, а в AWT содержится платформенно-ориентированные средства для работы с окнами.

Окно верхнего уровня в java наз-тся фреймом, для такого окна в AWT содержится класс Frame, для swing JFrame.

Класс JFrame расширяет класс Frame и представляет собой один из немногих компонентов библиотеки swing, кот не отображаются на холсте, а кнопки, строки заголовков и др эл-ты пользовательской системы реализуются с помощью пользовательской оконной системы, а не библиотеки swing.

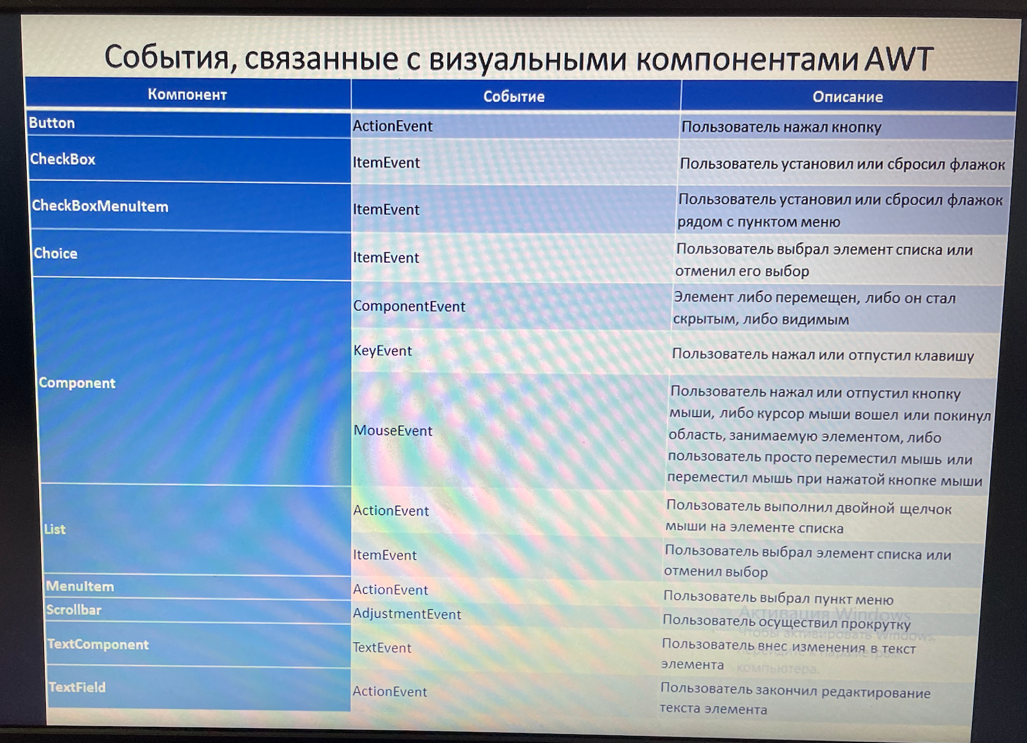
Классы библиотеки Swing находятся в пакете javax.swing.

Когда нужен один или два таких метода иногда проще получить подкласс класса адаптера, чем реализовывать интерфейс самостоятельно, при использовании адаптера требуется лишь определить те методы которые необход, а при прямой реализации интерфейса необходимо опред все методы, в том числе и не нужные.

Заранее определенные классы адаптера называются так же как определеные, но в названиях listener меняется на Adapter.

Изображение выглядит как текст, снимок экрана, доска

Автоматически созданное описание



1. **Обработка событий в Java.**

Обработка любого события состоит в связывании события с методом его обрабатывающим.

Принцип обработки события базируется на модели делегирования событий, в этой модели имеется блок прослушивания события – EventListener, который ждет поступления события определенного типа от источника, после чего обрабатывает его и возвращает управление. Источник — это объект, который генерирует событие, если его состояние изменяется. После генерации объект в событие пересылается для обработки зарегистрированному в источнике блока прослушивания запитав как параметр его методы. Блоки прослушивания Listener представляют собой объекты класса, реализующих интерфейсы прослушивания событий определенных в пакете java.awt.event, соответствующие методы объявленные в используемых интерфейсах необходимо реализовывать при создании собственных классов прослушивания. Эти методы и являются обработчиком события.

**Общий вид регистрации обработчика событий:  
Объект\_источник\_события.add\_событие\_Listener(обраб\_события)**

После этого объект прослушивания Listener будет реагировать именно на это событие и вызывать заданный метод.

Общая схема обработки событий:

1. Создается класс-обработчик, реализующий специальный интерфейс;
2. В источнике события регистрируется обработчик событий;
3. При наступлении события, источник посылает объекты события всем зарегестрированным обработчикам;
4. Обработчики события используют информацию, инкапсулированную в объекте события, для того, чтобы решить, как реагировать на это событие.

Когда событие происходит, то все зарегистрированные блоки прослушивания принимают копии события. В качестве блоков прослушивания на практике исп внутренние классы. Каждый интерфейс, включаемый в блок прослушивания наследуется от интерфейса EventListener и предназначен для обработки события определенного типа, при этом он содержит один или несколько методов, кот всегда принимают объект события в качестве единственного параметра и вызывается в опред обстоятельствах.

**Общий вид реализации**

Class MyListenner implements ActionListener  
{  
 public void actiomPerformed(ActionEvent e)  
 {  
 //при нажатии на кнопку будет выполнен этот код  
 }  
}