



# Lesson 21

07.06.2021

```
public class ex1 {  
    int num = 100;  
    public void calc(int num)    { num = num * 10;  }  
    public void printNum()       { System.out.println(num); }  
  
    public static void main(String[] args)  
    {  
        ex1 obj = new ex1();  
        obj.calc(2);  
        obj.printNum();  
    }  
}
```

```
class First {  
    public First() { System.out.println("a"); }  
}  
  
class Second extends First {  
    public Second() { System.out.println("b"); }  
}  
  
class Third extends Second {  
    public Third() { System.out.println("c"); }  
}  
  
public class ex2 {  
    public static void main(String[] args) {  
        Third c = new Third();  
    }  
}
```

```
public class ex3 {  
    public static void main(String[] args) {  
        var list = List.of(new String[]{"A", "BB", "CCC"},  
                          new String[]{"DD", "E"});  
        list.forEach(x ->  
            System.out.print(x.length));  
    }  
}
```

```
public class ex4 {  
    public static void main(String[] args) {  
        char[][] arr = {  
            {'A', 'B', 'C'},  
            {'D', 'E', 'F'},  
            {'G', 'H', 'I'}  
        };  
        for (int i = 0; i < arr.length; i++) {  
            for (int j = 0; j < arr[i].length; j++) {  
                System.out.print(arr[i][1]);  
            }  
            System.out.println();  
        }  
    }  
}
```

```
class Car {  
    void speed(Byte val) {System.out.println("DARK");}  
  
    void speed(byte... vals) {System.out.println("LIGHT");}  
}  
  
public class ex5 {  
  
    public static void main(String[] args) {  
        byte b = 10;  
        new Car().speed(b);  
    }  
}
```



## Future

Проще говоря, класс *Future* представляет будущий результат асинхронного вычисления - результат, который в конечном итоге появится в *Future* после завершения обработки.

Давайте посмотрим, как писать методы, которые создают и возвращают экземпляр *Future* .

Длительные методы являются хорошими кандидатами для асинхронной обработки и интерфейса *Future* . Это позволяет нам выполнять какой-то другой процесс, пока мы ожидаем завершения задачи, заключенной в *Future* .

Вот некоторые примеры операций, которые используют асинхронную природу *Future* :

- вычислительные интенсивные процессы (математические и научные расчеты)

- манипулирование большими структурами данных (большие данные)

- удаленные вызовы методов (загрузка файлов, очистка HTML, веб-сервисы).

## Исполнители

Concurrency API вводит понятие сервиса-исполнителя (*ExecutorService*) — высокоуровневую замену работе с потоками напрямую. Исполнители выполняют задачи асинхронно и обычно используют пул потоков, так что нам не надо создавать их вручную. Все потоки из пула будут использованы повторно после выполнения задачи, а значит, мы можем создать в приложении столько задач, сколько хотим, используя один исполнитель.

Класс Executors предоставляет удобные методы-фабрики для создания различных сервисов исполнителей. В данном случае мы использовали исполнитель с одним потоком.



Кроме Runnable, исполнители могут принимать другой вид задач, который называется Callable. Callable — это также функциональный интерфейс, но, в отличие от Runnable, он может возвращать значение.

Callable-задачи также могут быть переданы исполнителям. Но как тогда получить результат, который они возвращают? Поскольку метод submit() не ждет завершения задачи, исполнитель не может вернуть результат задачи напрямую. Вместо этого исполнитель возвращает специальный объект Future, у которого мы сможем запросить результат задачи.

После отправки задачи исполнителю мы сначала проверяем, завершено ли ее выполнение, с помощью метода isDone(). Поскольку задача имеет задержку, прежде чем вернуть число, я более чем уверен, что она еще не завершена.

Вызов метода get() блокирует поток и ждет завершения задачи, а затем возвращает результат ее выполнения. Теперь future.isDone() вернет true, и мы увидим на консоли результат.



## CompletableFuture

CompletableFuture используется для асинхронного программирования в Java. Асинхронное программирование — это средство написания *неблокирующего* кода путём выполнения задачи в отдельном, отличном от главного, потоке, а также уведомление главного потока о ходе выполнения, завершении или сбое.

Таким образом, основной поток не блокируется и не ждёт завершения задачи, а значит может параллельно выполнять и другие задания.

Наличие такого рода параллелизма значительно повышает производительность программ.

## Future vs CompletableFuture

CompletableFuture это расширение Future API, представленного в Java 5.

Future используется как ссылка на результат асинхронной задачи. В нём есть метод `isDone()` для проверки, завершилась ли задача или нет, а также метод `get()` для получения результата после его завершения.

Future API был хорошим шагом на пути к асинхронному программированию, но ему не хватало некоторых важных и полезных функций.

**База данных** — совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют в соответствии с правилами средств моделирования данных.

## Классификация по модели данных

Примеры:

Иерархическая

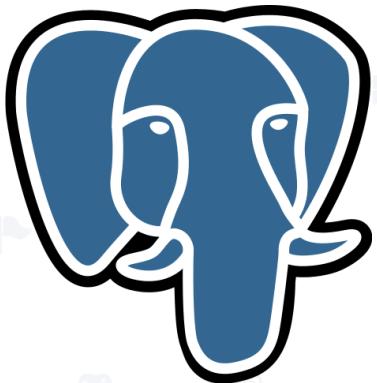
Объектная и объектно-ориентированная

**Объектно-реляционная**

Реляционная

Сетевая

Функциональная.

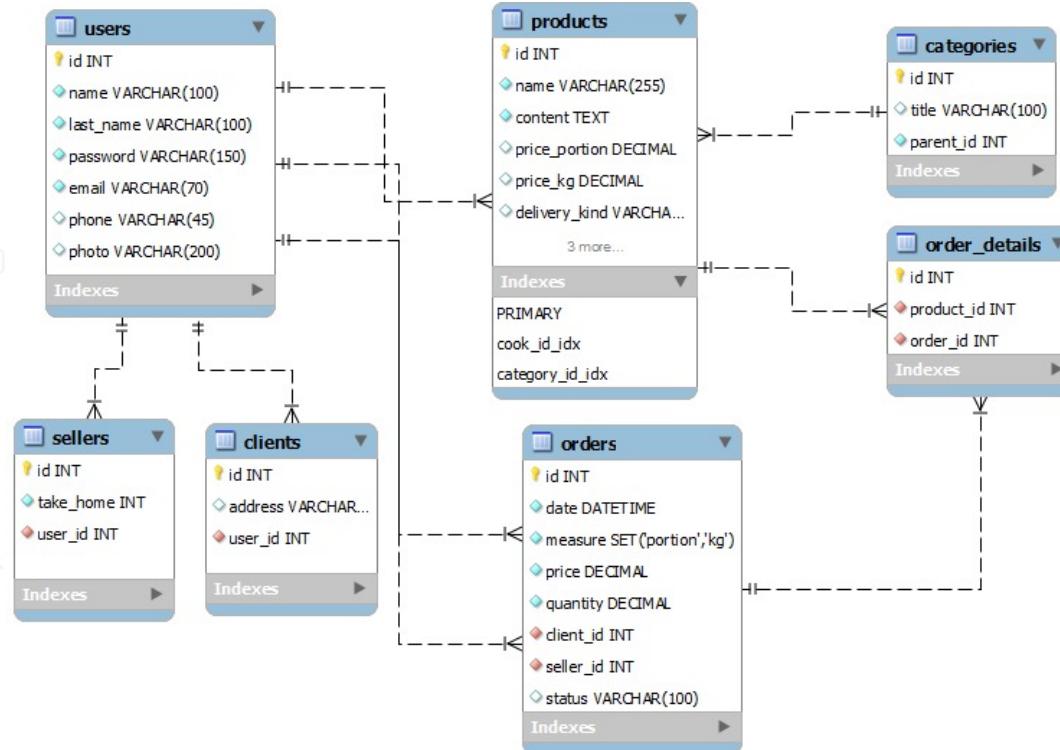




Rank			DBMS	Database Model	Score		
Dec 2019	Nov 2019	Dec 2018			Dec 2019	Nov 2019	Dec 2018
1.	1.	1.	Oracle	Relational, Multi-model	1346.39	+10.33	+63.17
2.	2.	2.	MySQL	Relational, Multi-model	1275.67	+9.38	+114.42
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1096.20	+14.29	+55.86
4.	4.	4.	PostgreSQL	Relational, Multi-model	503.37	+12.30	+42.74
5.	5.	5.	MongoDB	Document, Multi-model	421.12	+7.94	+42.50
6.	6.	6.	IBM Db2	Relational, Multi-model	171.35	-1.25	-9.40
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	150.25	+1.85	+5.55
8.	8.	↓ 7.	Redis	Key-value, Multi-model	146.23	+1.00	-0.59
9.	9.	9.	Microsoft Access	Relational	129.47	-0.60	-10.04
10.	10.	↑ 11.	Cassandra	Wide column	120.71	-2.52	-1.10
11.	11.	↓ 10.	SQLite	Relational	120.36	-0.66	-2.65
12.	12.	12.	Splunk	Search engine	90.53	+1.46	+8.34
13.	13.	↑ 14.	MariaDB	Relational, Multi-model	86.79	+1.22	+9.53
14.	14.	↑ 15.	Hive	Relational	86.05	+1.83	+18.67
15.	15.	↓ 13.	Teradata	Relational, Multi-model	78.49	-1.86	-0.67
16.	16.	↑ 21.	Amazon DynamoDB	Multi-model	61.63	+0.26	+7.33
17.	17.	↓ 16.	Solr	Search engine	57.22	-0.56	-4.13
18.	↑ 19.	↑ 20.	SAP Adaptive Server	Relational	55.55	+0.25	-0.27

<https://db-engines.com/en/ranking>

**Схема БД** (базы данных) -- это набор всех схем её таблиц, т.е. описание всех колонок этих таблиц (их типов, допустимых значений, связей между таблицами типа внешних ключей, индексов и т.д.), *без учета конкретных данных, записанных в таблицы БД*.



Эти инструкции являются  
обязательными

```
CREATE DATABASE [IF NOT EXISTS] имя_базы  
[CHARACTER SET имя_кодировки] [COLLATE collation];
```

Эти инструкции могут быть  
пропущены

Инструкция **IF NOT EXIST** позволяет перед созданием базы данных проверить, существует ли уже база с таким именем. Если она существует, то ничего не произойдет. Если она не существует, то база будет создана согласно всем инструкциям. При пропуске инструкции **IF NOT EXIST** в случае существования базы возникнет ошибка.

Инструкция **CHARACTER SET** устанавливает кодировку, которая будет использоваться для всей базы данных. Если она пропущена, то это не значит, что кодировка не используется. Просто выбирается некоторая кодировка по умолчанию.

Инструкция **COLLATE** устанавливает порядок сортировки принятый для данной кодировки.

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] **имя\_таблицы**

(определение\_поля1, определение\_поля2...) [table\_options]

Инструкция TEMPORARY позволяет создавать таблицу временно.

Инструкция IF NOT EXIST работает точно так же как и аналогичная инструкция команды CREATE DATABASE.

Сам раздел, где указываются определения полей таблицы, является обязательным. Но для определения поля могут использоваться разные инструкции, не все из которых обязательны. Этот раздел имеет следующую структуру

**имя поля** type [NOT NULL | NULL] [DEFAULT значение по умолчанию]  
[AUTO\_INCREMENT] [PRIMARY KEY] [reference\_definition]

**type** – обязательное определение типа поля. Поле может быть числовым, строковым, датой, логическим и т.д.

Инструкция **NULL | NOT NULL** разрешает или запрещает ставить в поле значение **NULL** (то есть оставлять поле пустым)

Инструкция **DEFAULT** позволяет установить в поле значение по умолчанию, которое будет автоматически подставляться, если пользователь не введет другого значения.

Инструкция **AUTO\_INCREMENT** используется для полей целого типа, которые нужно автоматически при появлении следующей записи увеличивать на единицу.

Инструкция **PRIMARY KEY** устанавливается, если данное поле входит в первичный ключ.

## Символьные типы

**CHAR:** представляет строку фиксированной длины. Длина хранимой строки указывается в скобках, например, CHAR(10) - строка из десяти символов. И если в таблицу в данный столбец сохраняется строка из 6 символов, то строка дополняется 4 пробелами и в итоге все равно будет занимать 10 символов

**VARCHAR:** представляет строку переменной длины. Длина хранимой строки также указывается в скобках, например, VARCHAR(10). Однако в отличие от CHAR хранимая строка будет занимать именно столько места, сколько необходимо. Например, если определенная длина в 10 символов, но в столбец сохраняется строка в 6 символов, то хранимая строка так и будет занимать 6 символов плюс дополнительный байт, который хранит длину строки.

Ряд дополнительных типов данных представляют текст неопределенной длины:

**TINYTEXT:** представляет текст длиной до 255 байт.

**TEXT:** представляет текст длиной до 65 КБ.

**MEDIUMTEXT:** представляет текст длиной до 16 МБ

**LARGETEXT:** представляет текст длиной до 4 ГБ

## Числовые типы

**TINYINT**: представляет целые числа от -127 до 128, занимает 1 байт

**BOOL**: фактически не представляет отдельный тип, а является лишь псевдонимом для типа TINYINT(1) и может хранить два значения 0 и 1. Однако данный тип может также в качестве значения принимать встроенные константы TRUE (представляет число 1) и FALSE (предоставляет число 0).

**TINYINT UNSIGNED**: представляет целые числа от 0 до 255, занимает 1 байт

**SMALLINT**: представляет целые числа от -32768 до 32767, занимает 2 байта

**SMALLINT UNSIGNED**: представляет целые числа от 0 до 65535, занимает 2 байта

**MEDIUMINT**: представляет целые числа от -8388608 до 8388607, занимает 3 байта

**MEDIUMINT UNSIGNED:** представляет целые числа от 0 до 16777215, занимает 3 байта

**INT:** представляет целые числа от -2147483648 до 2147483647, занимает 4 байта

**INT UNSIGNED:** представляет целые числа от 0 до 4294967295, занимает 4 байта

**BIGINT:** представляет целые числа от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807, занимает 8 байт

**BIGINT UNSIGNED:** представляет целые числа от 0 до 18 446 744 073 709 551 615, занимает 8 байт

**DECIMAL:** хранит числа с фиксированной точностью. Данный тип может принимать два параметра precision и scale: DECIMAL(precision, scale).

## Типы для работы с датой и временем

**DATE**: хранит даты с 1 января 1000 года до 31 декабря 9999 года (с "1000-01-01" до "9999-12-31"). По умолчанию для хранения используется формат yyyy-mm-dd. Занимает 3 байта.

**TIME**: хранит время от -838:59:59 до 838:59:59. По умолчанию для хранения времени применяется формат "hh:mm:ss". Занимает 3 байта.

**DATETIME**: объединяет время и дату, диапазон дат и времени - с 1 января 1000 года по 31 декабря 9999 года (с "1000-01-01 00:00:00" до "9999-12-31 23:59:59"). Для хранения по умолчанию используется формат "yyyy-mm-dd hh:mm:ss". Занимает 8 байт

**TIMESTAMP**: также хранит дату и время, но в другом диапазоне: от "1970-01-01 00:00:01" UTC до "2038-01-19 03:14:07" UTC. Занимает 4 байта

**YEAR**: хранит год в виде 4 цифр. Диапазон доступных значений от 1901 до 2155.

## Составные типы

**ENUM**: хранит одно значение из списка допустимых значений. Занимает 1-2 байта

**SET**: может хранить несколько значений (до 64 значений) из некоторого списка допустимых значений. Занимает 1-8 байт.

## Бинарные типы

**TINYBLOB**: хранит бинарные данные в виде строки длиной до 255 байт.

**BLOB**: хранит бинарные данные в виде строки длиной до 65 КБ.

**MEDIUMBLOB**: хранит бинарные данные в виде строки длиной до 16 МБ

**LARGELOB**: хранит бинарные данные в виде строки длиной до 4 ГБ



## Primary key (PK)

В каждой таблице БД может существовать первичный ключ. Под первичным ключом понимают поле или набор полей, однозначно (уникально) идентифицирующих запись. Первичный ключ должен быть минимально достаточным: в нем не должно быть полей, удаление которых из первичного ключа не отразится на его уникальности.

## Foreign key(FK)

Обеспечивает однозначную логическую связь, между таблицами одной БД.

```
ALTER TABLE table_name  
ADD column_name datatype;
```

```
INSERT INTO table_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);
```

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

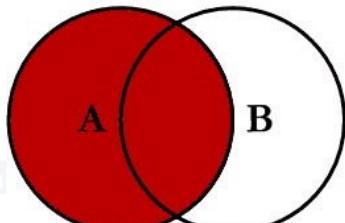
```
SELECT column1, column2, ...  
FROM table_name;
```

```
SELECT COUNT(column_name)  
FROM table_name  
WHERE condition;
```

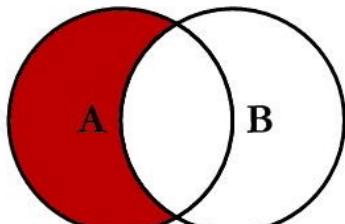
```
SELECT AVG(column_name)  
FROM table_name  
WHERE condition;
```

```
SELECT SUM(column_name)  
FROM table_name  
WHERE condition;
```

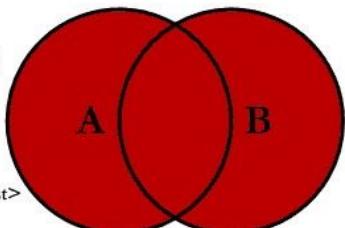
# SQL JOINS



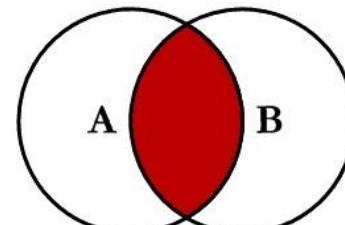
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



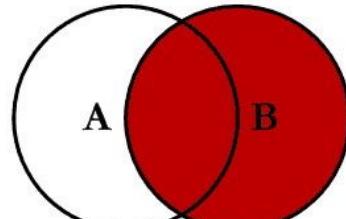
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



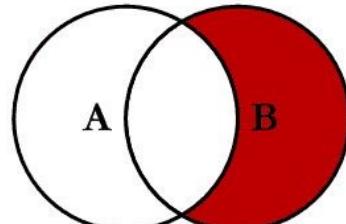
```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



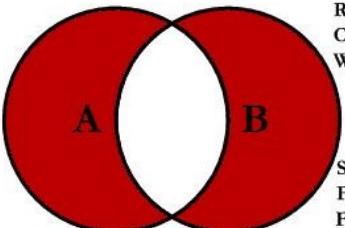
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



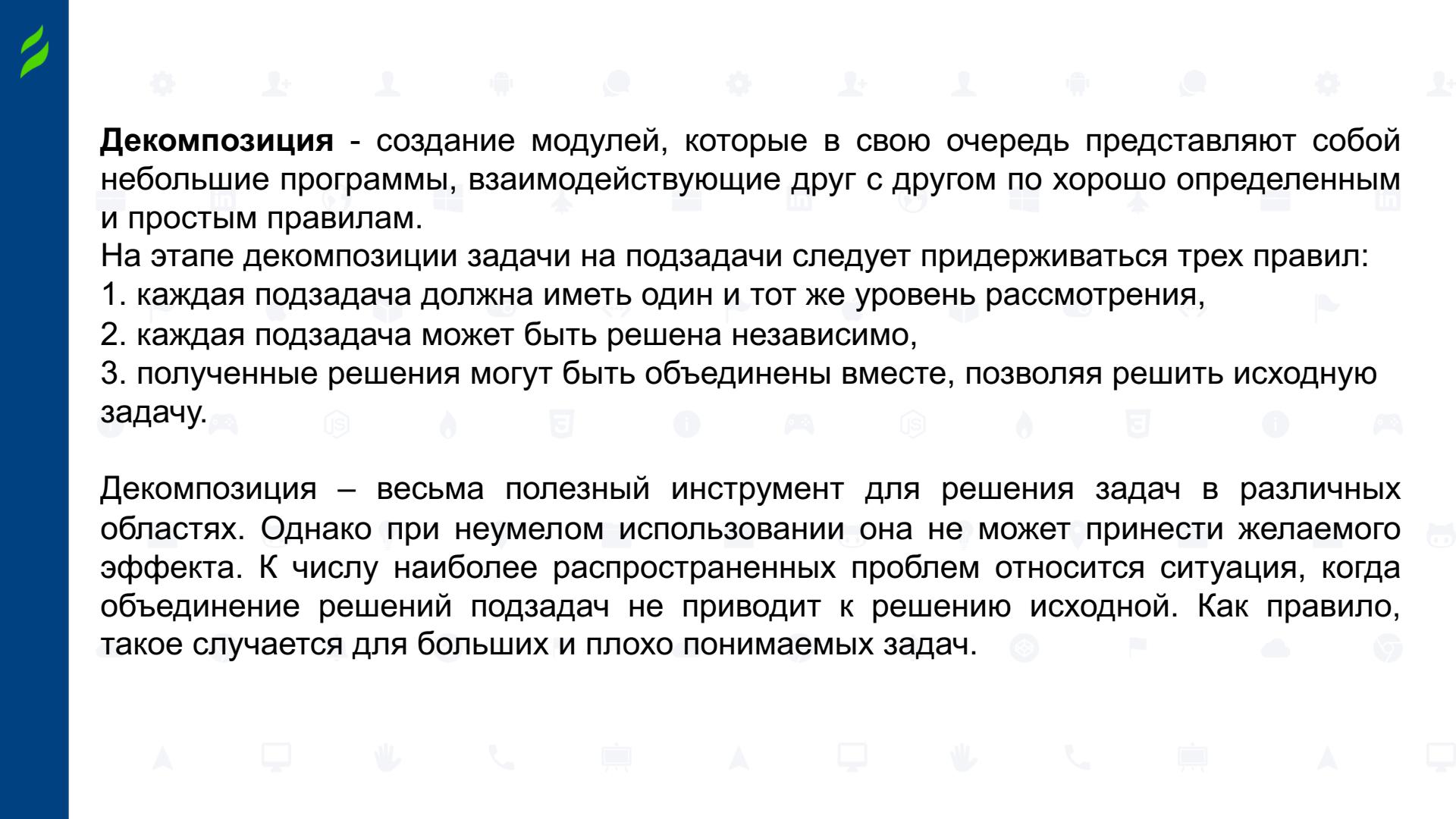
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
WHERE A.Key IS NULL
OR B.Key IS NULL
```



**Декомпозиция** - создание модулей, которые в свою очередь представляют собой небольшие программы, взаимодействующие друг с другом по хорошо определенным и простым правилам.

На этапе декомпозиции задачи на подзадачи следует придерживаться трех правил:

1. каждая подзадача должна иметь один и тот же уровень рассмотрения,
2. каждая подзадача может быть решена независимо,
3. полученные решения могут быть объединены вместе, позволяя решить исходную задачу.

Декомпозиция – весьма полезный инструмент для решения задач в различных областях. Однако при неумелом использовании она не может принести желаемого эффекта. К числу наиболее распространенных проблем относится ситуация, когда объединение решений подзадач не приводит к решению исходной. Как правило, такое случается для больших и плохо понимаемых задач.