

## 5.2 SQL 基礎

- ・データ定義
- ・データ操作
- ・テーブルの結合



# 目 次

- 1 データ定義
- 2 データ操作
- 3 テーブルの結合

# SQL クエリ

- 以下は、最も一般的な SQL 言語のクエリの一覧です：
  - データベース操作：CREATE DATABASE、DROP DATABASE など。
  - テーブル操作（データ定義）：CREATE TABLE、DROP TABLE、ALTER TABLE、TRUNCATE TABLE など。
  - 追加、削除、検索操作（データ操作）：INSERT INTO、DELETE、UPDATE、SELECT など。
  - これに加えて、結果セットの和、積、差分、テーブルの結合などの特殊な操作。
- この授業で扱わないクエリについては、PostgreSQL の公式チュートリアルを参照してください：  
 <https://www.postgresql.org/docs/14/index.html>

# 基本文法

- 識別子の命名は Java に似ています。英文字、数字、アンダースコア「\_」、ドル記号「\$」を使うことができます。ただし:
  - SQL では、キーワードと識別子は**大文字と小文字を区別しない**。select は SELECT と、student は Student と同じ。区別するために、キーワードはすべて大文字、識別子はすべて小文字にすることをお勧めします。
  - 識別子は、blog\_user という**小文字のスネークケース**で書かれる場合は多いが、DBMS によって規格が異なる場合があります。
  - データベース名はすべて単数形 (user など) または複数形 (users など) で記述することがお勧めです。
- 各 SQL クエリはセミコロン「;」で終わること。

## コメント

- 一行コメントを追加する場合は、「**--**」を使用します：

```
-- 一行コメントです。
```

- 複数行コメントを追加するには、「**/\* \*/**」を使用します：

```
/*
 * 多行コメントはこのように
 * かけます。
 */
```

# データベース作成

- **CREATE DATABASE** 文は、新しいデータベースを作成するために使用します。構文：

```
CREATE DATABASE name_of_database;
```

- 次のステートメントは、「hello」という名前のデータベースを作成します。

```
CREATE DATABASE hello;
```

```
postgres=# CREATE DATABASE hello;
CREATE DATABASE
postgres# ■
```

# データベース削除

- **DROP DATABASE** 文は、既存のデータベースを削除するために使用されます。構文：

```
DROP DATABASE name_of_database;
```

- 以下の文は、「hello」というデータベースを削除します：

```
DROP DATABASE hello;
```

Note !

削除操作は元に戻せない！よく  
考えてから行いましょう。

# テーブルの作成

- **CREATE TABLE** 文は、データベースに新しいテーブルを作成するためには使用します。構文:

```
1 CREATE TABLE table_name (
2   column1 type1,
3   column2 type2,
4   column3 type3
5 );
```

- ここで:
  - `table_name` はテーブル名を指定する。
  - `column1`、`column2`、`column3` は表のカラムの名を指定する。
  - `type1`、`type2`、`type3` は各カラムのデータ型を指定する。

# データ型

- PostgreSQL では非常に多くのデータ型が利用可能ですが、ここでは最も一般的なもののみを説明します。利用可能な全てのデータ型は、以下のページで確認できます：

 <https://www.postgresql.org/docs/14/datatype.html>

# 数字

データ型	意味
SMALLINT	小さな整数
INTEGER / INT	整数
BIGINT	大きな整数
DECIMAL / NUMERIC	精度が高い小数
REAL / DOUBLE PRECISION	精度が低い小数
SMALLSERIAL	自動増加する小さな整数
SERIAL	自動増加する整数
BIGSERIAL	自動増加する大きな整数

# 文字列

データ型	意味
VARCHAR( <i>n</i> )	制限ある可変長の文字列
CHAR( <i>n</i> )	制限ある固定長の文字列
TEXT	制限なしの文字列

Note !

文字列の値は、单一引用符「'」で囲む。

# その他のデータ型

データ型	意味
BOOLEAN	ブール型 (true、false、unknown)
ENUM( 'a', 'b', 'c' )	列挙型
MONEY	金額を表す小数
TIMESTAMP	日付と時間
DATE	日付
TIME	時間

# テーブルの例

- 以下の例では、id、name、score の 3 つのカラムを含む「student」というテーブルを作成しています：

```
1 CREATE TABLE student (
2     id      SERIAL,
3     name   VARCHAR(255),
4     score  SMALLINT
5 );
```

- ここで：
  - id の型は整数で、自動増加する。
  - name は文字数が 255 以下の文字列。
  - score は小さな整数。

# 制約

- 制約[Constraint]は、テーブル内のデータが特定の条件を満たすように要求するために使用されます。テーブルを作成する際、データ型の後に制約を書くことができます。構文:

```
1 CREATE TABLE table_name (
2   column1 type1 constraints1,
3   column2 type2 constraints2,
4   column3 type3 constraints3
5 );
```

- ここで、constraints1、constraints2、constraints3は、各カラムの制約を指定します。
- 制約を満たさないデータを挿入しようとすると、エラーが発生してクエリは直ちに終了します。

# 制約の種類

- SQL の一般的な制約は以下の通りです：
  - **NOT NULL**: フィールドが NULL であるべきない。
  - **UNIQUE**: カラム内のすべての値が異なるべき。
  - **PRIMARY KEY**: 主キー。NOT NULL + UNIQUE に相当する。
  - **FOREIGN KEY**: 外部キー。他のテーブルの主キーでリンクすることができます。
  - **DEFAULT**: デフォルト値を設定。
  - **INDEX**: データを取得するために使用するカラムを設定する。このカラムを使用して検索クエリすると高速になります。
- スペースで区切って、複数の制約をカラムに追加することもできます。

# 制約の例

- 先ほどの例と同じような student テーブル:

```
1 CREATE TABLE student (
2     id          SERIAL           PRIMARY KEY,
3     name        VARCHAR(255)    NOT NULL,
4     score       SMALLINT        DEFAULT 0
5 );
```

- ここで:
  - id は主キーであり、一意に決定され、かつ NULL であるべきない。
  - name を NULL にすることはできない。
  - score のデフォルト値は 0。

# テーブルのカラムを変更

- **ALTER TABLE** 文で、テーブルのカラムを追加、削除、修正したり、その制約を変更したり、色々な変更を行えます。構文:
  - カラムの追加:

```
ALTER TABLE table_name ADD column_name type;
```

- カラムの削除:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

◀ 前へ

## ● 構文:

- カラムのデータ型を変更する:

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE type;
```

- カラムに NOT NULL 制約を追加する:

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

# テーブルを削除

- **DROP TABLE** 文は、データベース内の既存の**テーブルを削除**するために使用されます。構文:

```
DROP TABLE table_name;
```

- 以下のクエリは、students テーブルを削除します:

```
DROP TABLE student;
```

- 次のクエリは、テーブルが存在する場合のみ削除できます:

```
DROP TABLE IF EXISTS student;
```



# Q & A

*Question and answer*



- 1 データ定義
- 2 データ操作
- 3 テーブルの結合

# データを挿入

- **INSERT INTO** 文は、テーブルに新しいレコードを挿入するために使用されます。構文:

```
INSERT INTO table_name VALUES (data1, data2, ...);
```

- 各レコードの区切りにはカンマ「,」を使用し、複数行を一度に挿入できるようにする:

```
1 INSERT INTO student
2 VALUES (0, 'Alice' , 90),
3      (1, 'Bob'   , 40),
4      (2, 'Charlie', 70);
```

## Note !

データは**定義された**  
**ときの順番**に書かなければならぬ。

# 指定したカラムのデータを挿入

- また、挿入するデータのカラムを指定することも可能である。この際、指定されないフィールドには自動的に NULL（または DEFAULT で設定されたデフォルト値）に設定されます：

```
1 INSERT INTO table_name (column1, column2)
2 VALUES (data1_column1, data1_column2),
3         (data2_column1, data2_column2);
```

## Tips

この場合、column1 や column2 などの順番は、定義されたものと異なっても構いません。

# 挿入の例

- 次のクエリは、name が「Dave」のレコードを挿入：

```
INSERT INTO student (id, name) VALUES (3, 'Dave');
```

- 結果：

id	name	score
0	Alice	90
1	Bob	40
2	Charlie	70
3	Dave	0

Try  
insert.sql

- ご覧の通り：

- id フィールドと name フィールドに指定された値が設定され、
- score フィールドは自動的にデフォルト値に設定される。

# データの検索

- **SELECT** 文は、テーブルから特定のレコードを検索してそのデータを取得するために使用します。構文：

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

- ここで、column1 と column2 は取得したい列の名前です。
- 返されたデータは、表の形で表示され、**結果セット**[Result Set] と呼ばれます。

# 検索結果の例

- 次のクエリは、name と score のフィールドを検索します：

```
SELECT name, score FROM student;
```

```
hello=# SELECT name, score FROM student;
   name | score
-----+
  Alice |    90
    Bob |     40
Charlie |    70
   Dave |      0
(4 rows)
```



また、一般的な演算子も使える：

```
SELECT name, score * 2 - 10 FROM student;
```

Try   
select.sql

# すべてのカラムを検索する

- すべてのカラムを選択する場合は「\*」を使用します：

```
SELECT * FROM student;
```

```
hello=# SELECT * FROM student;
```

	id	name	score
0	Alice	90	
1	Bob	40	
2	Charlie	70	
3	Dave	0	

# WHERE 句

- あるテーブルのデータに対して特定の条件を満たすレコードだけを検索したい場合があります。この場合、**WHERE** 句が使用されます。構文:

```
1 SELECT column1, column2, ... FROM table_name  
2 WHERE constraints;
```

- この場合、constraints のところで、レコードの中の特定のフィールドが検索されるために満たす必要がある条件を指定することができます。

# 比較演算子

- 最も基本的な条件はデータの比較:

演算子	意味
=	等しい
>	大なり
<	小なり
>=	大なりイコール
<=	小なりイコール
!= または <>	等しくない

# WHERE の例

- 次のクエリを実行すると、テーブルの中で score が 60 より大きいレコードだけ選択されます:

```
1 SELECT * FROM student  
2 WHERE score > 60;
```

```
hello=# SELECT * FROM student WHERE score > 60;  
 id | name      | score  
---+-----+-----  
  0 | Alice     | 90  
  2 | Charlie   | 70
```

# NULL の比較

- フィールドが NULL かどうかを判断するには、比較演算子ではなく、**IS NULL** または **IS NOT NULL** 演算子のみを使用できます。
- 例えば、以下のクエリは name が NULL でないレコードだけ選択します：

```
1 SELECT * FROM student  
2 WHERE name IS NOT NULL;
```

# LIKE 演算子

- **LIKE** 演算子は、**文字列**のフィールドが特定の条件を満たすことを表します。
- 通常、LIKE 演算子は次のような記号と組み合わせて使用されます（このような記号は他にもあります）：
  - 「%」は、任意長の文字列を示す。
  - 「\_」は、任意の 1 文字を表す。
- 例えば、以下のクエリは、name が「A」で始まるテーブル内のレコードを検索：

```
1 SELECT * FROM student  
2 WHERE name LIKE 'A%';
```

- 正規表現の機能とは似ていますが、異なる文法を持っています。

# IN 演算子

- IN 演算子で複数の値を指定します。フィールドがそのうちのどれかであれば、条件は満たします。
- 次のクエリは、点数が 50、60、70 のレコードを選択：

```
1 SELECT * FROM student  
2 WHERE score IN (50, 60, 70);
```

# BETWEEN 演算子

- **BETWEEN-AND** 演算子は、フィールドがあるべきデータの範囲を指定します。数値、文字列、日付（時間）のタイプの値を指定することができます。
- 次のクエリは、score が 30 から 50 までのレコードを選択します：

```
1 SELECT * FROM student  
2 WHERE score BETWEEN 30 AND 50;
```

# 論理演算子

- 上記の演算子は、AND（かつ）、OR（または）、NOT（...ではない）演算子と組み合わせて使用することも可能です。
- 次のクエリは、「name が「e」で終わる、かつ score が 50 より大きい」レコードを検索します：

```
1 SELECT * FROM student  
2 WHERE name LIKE '%e'  
3 AND score > 50;
```



*Question and answer*

# ORDER BY 句

- ORDER BY 句は、結果セットを並べ替えるために使用され、整列の基準になるカラムを選択することもできます。
- ORDER BY はデフォルトでは、レコードを昇順で並べ替えます。降順に並べるには、DESC を使用します。
- 次のクエリはレコードを score の降順で並べ替えます。

```
1 SELECT * FROM student  
2 ORDER BY score DESC;
```

id	name	score
0	Alice	90
2	Charlie	70
1	Bob	40
3	Dave	0

# DISTINCT キーワード

- 検索した結果セットに重複したレコードが含まれる場合があります。
- **DISTINCT** キーワードを使用して、検索結果から重複するデータを削除できます：

```
SELECT DISTINCT score FROM student;
```

# LIMIT と OFFSET

- **LIMIT** キーワードを使用すると、結果の最初の数行だけを保留することができます。
- 次のクエリは、最初の 3 行のみを照会します：

```
SELECT * FROM student LIMIT 3;
```

- クエリ結果の最初の数行をスキップするには、**OFFSET** キーワードを使用します。
- 次のクエリ、5 行目から始まる 3 つの行に検索します：

```
SELECT * FROM student OFFSET 4 LIMIT 3;
```

# 出力列の名前を変更する

- **AS** キーワードで出力列の名前を変更することができます。
- 選択したカラム名の後に「**AS [新し名前]**」と記述すると、SQL が新しい名前でクエリ結果を出力するようになる：

```
1 SELECT name AS student_name, score AS test_score  
2 FROM student;
```

```
hello=# SELECT name AS student_name, score AS test_score FROM student;  
student_name | test_score  
-----+-----  
Alice        | 90  
Bob          | 40  
Charlie      | 70  
Dave         | 0
```

# 集計関数

- **集計関数** [Aggregate Function] を使って、結果セットのデータ全体から特定の結果を計算することができます:
  - MIN()・MAX() 関数は、カラムの最小値・最大値を算出。
  - COUNT() 関数は、データの数をカウント。
  - AVG() 関数は、カラムの平均値を計算。
  - SUM() 関数は、カラムの和を計算。
- 次のクエリは、score の最大値、最小値、平均値を計算します:

```
SELECT MAX(score), MIN(score), AVG(score) FROM student;
```

# GROUP BY

- **GROUP BY** 句は、同じ値を持つレコードをまとめて表示します。例えば、「国・地域別」のような効果を実現できます。
- GROUP BY 句は、先程紹介した集計関数と併用して各グループのデータから何かの統計的な結果を計算するのは一般です。
- 次のクエリは、レコードを国別にグループ化し、各国の平均賃金を計算します：

```
1 SELECT country, AVG(salary)
2 FROM employee
3 GROUP BY country;
```

Try  
group.sql

# HAVING 句

- HAVING 句で、データのグループを選択することができます。
- HAVING 句は、GROUP BY 句の後に置かなければなりません。
- 次の文は、記録を国別にグループ化し、平均賃金が 80 を超える国を出力しています：

```
1 SELECT country FROM employee  
2 GROUP BY country  
3 HAVING AVG(salary) > 80;
```

# データの変更

- **UPDATE-SET** 文で、既存のレコードを変更できます。
- 変更したいレコードを選択するために、 WHERE 句は使用できます。
- 次のクエリは、 name が「Bob」のレコードの score フィールドを、新しいデータに変更します：

```
UPDATE student SET score = 100 WHERE name = 'Bob';
```

```
hello=# SELECT * FROM student;
```

	id	name	score
	0	Alice	90
	2	Charlie	70
	3	Dave	0
	1	Bob	100

# データ削除

- **DELETE** 文は、テーブルの既存のレコードを削除するために使用します。
- WHERE 句で削除する行を選択することもできます：

```
DELETE FROM student WHERE name = 'Bob';
```

- **TRUNCATE** 文で、テーブル内のすべての行を削除することができます。テーブル自体は削除しないことに注意：

```
TRUNCATE TABLE student;
```

# 検索結果の組み合わせ

- 複数の SELECT 文の結果を、いくつかの演算子で組み合わせることができます:
  - UNION は、結果セットの和集合を求める。
  - INTERSECT は、結果セットの積集合を求める。
  - EXCEPT は、ある結果セットと別の結果セットとの差集合を求める。
- 次のクエリは、両方のテーブルにある同じ name を検索:

```
1 SELECT name FROM student  
2 INTERSECT  
3 SELECT name FROM employee;
```



*Question and answer*



- 1 データ定義
- 2 データ操作
- 3 テーブルの結合

# 結合

- 2つのテーブルを、何らかの方法で**結合**[Join]させることができます。これにより、別のテーブルのデータを利用して、それに基づいて今のテーブルからデータを検索できます。
- SQL には、5種類の結合があります：
  - 交差結合：左右のテーブルレコードの「組み合わせ」をすべて列挙
  - 内部結合：左右のテーブルレコードの「組み合わせ」から、特定な条件を満たすレコードを選び出す
  - 左外部結合：右のテーブルのレコードを使用して、左のテーブルのデータを「拡張」する
  - 右外部結合：左のテーブルのレコードを使用して、右のテーブルのデータを「拡張」する
  - 完全外部結合：左と右の外部結合の和集合。

# テーブルの準備

- ここでは、以下のテーブルを例にして、各結合を説明します：

pokemon

<b>id</b>	<b>name</b>	<b>type_id</b>
37	'Vulpix'	10
46	'Paras'	7
133	'Eevee'	1

type

<b>id</b>	<b>name</b>	<b>super_id</b>
2	'Fight'	1
7	'Bug'	12
10	'Fire'	7
11	'Water'	10

Try  joins.sql

# 交差結合

- **交差結合**は、2つのテーブルのデカルト積、つまり、すべての可能なレコードの組み合わせを計算します。
- 交差結合を計算するには、**CROSS JOIN** またはその省略形である「,」を使用します。
- 次のクエリは、pokemon テーブルと type テーブルのデカルト積を計算します：

```
SELECT * FROM pokemon, type;
```

# 交差結合の結果

- このように、計算された結果セットには、pokemon と type の組み合わせの可能性がすべて記載されています：

```
hello=# SELECT * FROM pokemon, type;
```

id	name	type_id	id	name	super_id
37	Vulpix	10	2	Fight	1
37	Vulpix	10	7	Bug	12
37	Vulpix	10	10	Fire	7
37	Vulpix	10	11	Water	10
46	Paras	7	2	Fight	1
46	Paras	7	7	Bug	12
46	Paras	7	10	Fire	7
46	Paras	7	11	Water	10
133	Eevee	1	2	Fight	1
133	Eevee	1	7	Bug	12
133	Eevee	1	10	Fire	7
133	Eevee	1	11	Water	10

# 内部結合

- 内部結合でもレコードの組み合わせが作成されますが、特定の条件を満たす組み合わせのみが選択されます。
- 内部結合を計算するには、**INNER JOIN** またはその省略形の **JOIN** を使用します。**ON** を使って、各テーブルのレコードに同じであるべきカラムを指定します。
- 次のクエリは、pokemon の type\_id が type の id と同じであるようなレコードの組み合わせを選択します（同名問題の回避方法に注意）：

```
1 SELECT *
2 FROM pokemon JOIN type
3 ON pokemon.type_id = type.id;
```

# 内部結合の結果

- このように、pokemon の type\_id と type の id が一致するレコードだけが選択されました：

```
hello=# SELECT *
hello-# FROM pokemon JOIN type
hello-# ON pokemon.type_id = type.id;
+-----+-----+-----+-----+
| id | name | type_id | id | name | super_id |
+-----+-----+-----+-----+
| 37 | Vulpix |      10 | 10 | Fire |        7
| 46 | Paras   |       7 |  7 | Bug  |       12
```

- 次のクエリは実用例として、ポケモンのタイプ名を取得：

```
hello=# SELECT pokemon.name AS pokemon, type.name as type
hello-# FROM pokemon JOIN type
hello-# ON pokemon.type_id = type.id;
+-----+-----+
| pokemon | type |
+-----+-----+
| Vulpix | Fire |
| Paras  | Bug  |
```

# 左外部結合

- **左外部結合**は、右のテーブルで条件を満たすレコードを探し、左のテーブルのレコードを「拡張」します。
- 右のテーブルに対応するレコードがない場合、NULL で補完。
- 左外部結合を計算するには、**LEFT OUTER JOIN** またはその省略形である **LEFT JOIN** を使用します。
- 以下のクエリは、pokemon テーブルを type テーブルで「拡張」しています：

```
1 SELECT *
2 FROM pokemon LEFT JOIN type
3 ON pokemon.type_id = type.id;
```

# 左外部結合の結果

- 対応する type 持つ pokemon のレコードには右にデータが追加されて、対応する type がないレコードには NULL が追加されたことがわかります：

```
hello=# SELECT *
hello# FROM pokemon LEFT JOIN type
hello# ON pokemon.type_id = type.id;
+-----+-----+-----+-----+-----+
| id   | name | type_id | id   | name  | super_id |
+-----+-----+-----+-----+-----+
| 37   | Vulpix | 10    | 10   | Fire  | 7       |
| 46   | Paras   | 7     | 7    | Bug   | 12      |
| 133  | Eevee   | 1     |      |        |          |
+-----+-----+-----+-----+-----+
```

- 左外部結合と呼ばれる理由は、この操作では左のテーブルのすべてのデータが保留されるからです。

# 右外部結合

- 右外部結合は、左右を入れ替わる以外は左の外側の接続と同様です。
- 右外部結合を計算するには、**RIGHT OUTER JOIN** またはその省略形である **RIGHT JOIN** を使用します。
- 以下のクエリは、type テーブルを pokemon テーブルで「拡張」しています：

```
1 SELECT *
2 FROM pokemon RIGHT JOIN type
3 ON pokemon.type_id = type.id;
```

# 右外部結合の結果

- 右のテーブルのレコードは保留され、いくつかは「拡張」されていることがわかります：

```
hello=# SELECT *
hello-# FROM pokemon RIGHT JOIN type
hello-# ON pokemon.type_id = type.id;
 id | name   | type_id | id | name    | super_id
---+-----+-----+---+-----+-----
 37 | Vulpix |      10 | 10 | Fire    |      7
 46 | Paras   |       7 |  7 | Bug     |     12
                |           |       11 | 11 | Water   |     10
                |           |       2  |  2 | Fight   |      1
```

# 完全外部結合

- 完全外部結合は左と右の外部結合の和集合であり、両方のテーブルの全てのデータが保留されます。そして、対応関係があるレコードは、お互いにデータを補完します。
- 完全外部結合を計算するには、**FULL OUTER JOIN** またはその省略形である **FULL JOIN** を使用します。
- 以下のクエリにより、pokemon テーブルと type テーブルが互いに補完し合えるようになります：

```
1 SELECT *
2 FROM pokemon FULL JOIN type
3 ON pokemon.type_id = type.id;
```

# 完全外部結合の結果

- ご覧のように、左右のテーブルに元々あるすべてのレコードがクエリ結果で確認できます：

```
hello=# SELECT *
hello# FROM pokemon FULL JOIN type
hello# ON pokemon.type_id = type.id;
+----+-----+-----+----+-----+-----+
| id | name | type_id | id | name | super_id |
+----+-----+-----+----+-----+-----+
| 37 | Vulpix | 10 | 10 | Fire | 7
| 46 | Paras | 7 | 7 | Bug | 12
| 133 | Eevee | 1 | 11 | Water | 10
|      |        |    | 2 | Fight | 1
+----+-----+-----+----+-----+-----+
```

# 自己結合

- また、テーブルを自己結合 [Self Join] することも可能です。例えば、以下のクエリは左外部結合を使用して、type の super\_id に対応する type を探します（同名テーブルの問題の回避方法に注意）：

```
1 SELECT *
2 FROM type AS a LEFT JOIN type AS b
3 ON a.super_id = b.id;
```

```
hello=# SELECT *
hello-# FROM type AS a LEFT JOIN type AS b
hello-# ON a.super_id = b.id;
 id | name | super_id | id | name | super_id
---+-----+-----+-----+-----+
 2 | Fight |        1 |      |
 7 | Bug   |        12 |      |
10 | Fire  |        7 |      |
11 | Water |        10 |      |
                           | Bug  |        12 |
                           | Fire |        7 |
```



*Question and answer*

# まとめ

## Sum Up

1. SQL の基本構文。
2. データ定義構文：
  - ① データベースの作成と操作。
  - ② テーブルの作成、データ型、制約条件。
3. データ操作構文：
  - ① データ検索： WHERE、 ORDER BY、 GROUP BY など。
  - ② データの挿入・削除・修正の方法。
4. テーブル結合の構文と適用シーン。

**THANK YOU!**