

# Bazy danych: MySQL w PHP

v 1.1

# Plan

- [Wprowadzenie do baz danych](#)
- [Przygotowanie do pracy z MySQL](#)
- [Trochę teorii o MySQL](#)
- [MySQL i PHP](#)

- [Łączenie tabel](#)
- [Relacje między tabelami](#)
- [Zaawansowany SQL](#)

# Łączenie tabel

# Łączenie tabel

Wyniki z dwóch (lub więcej) tabel naraz możemy uzyskać dzięki użyciu wyrażenia kluczowego **JOIN ... ON ....**

Oto cztery możliwości łączenia tabel:

- **INNER,**
- **LEFT,**
- **RIGHT,**
- **FULL.**

```
SELECT column_name(s)
```

```
FROM table1
```

```
JOIN table2
```

```
ON table1.column_name=table2.column_name;
```

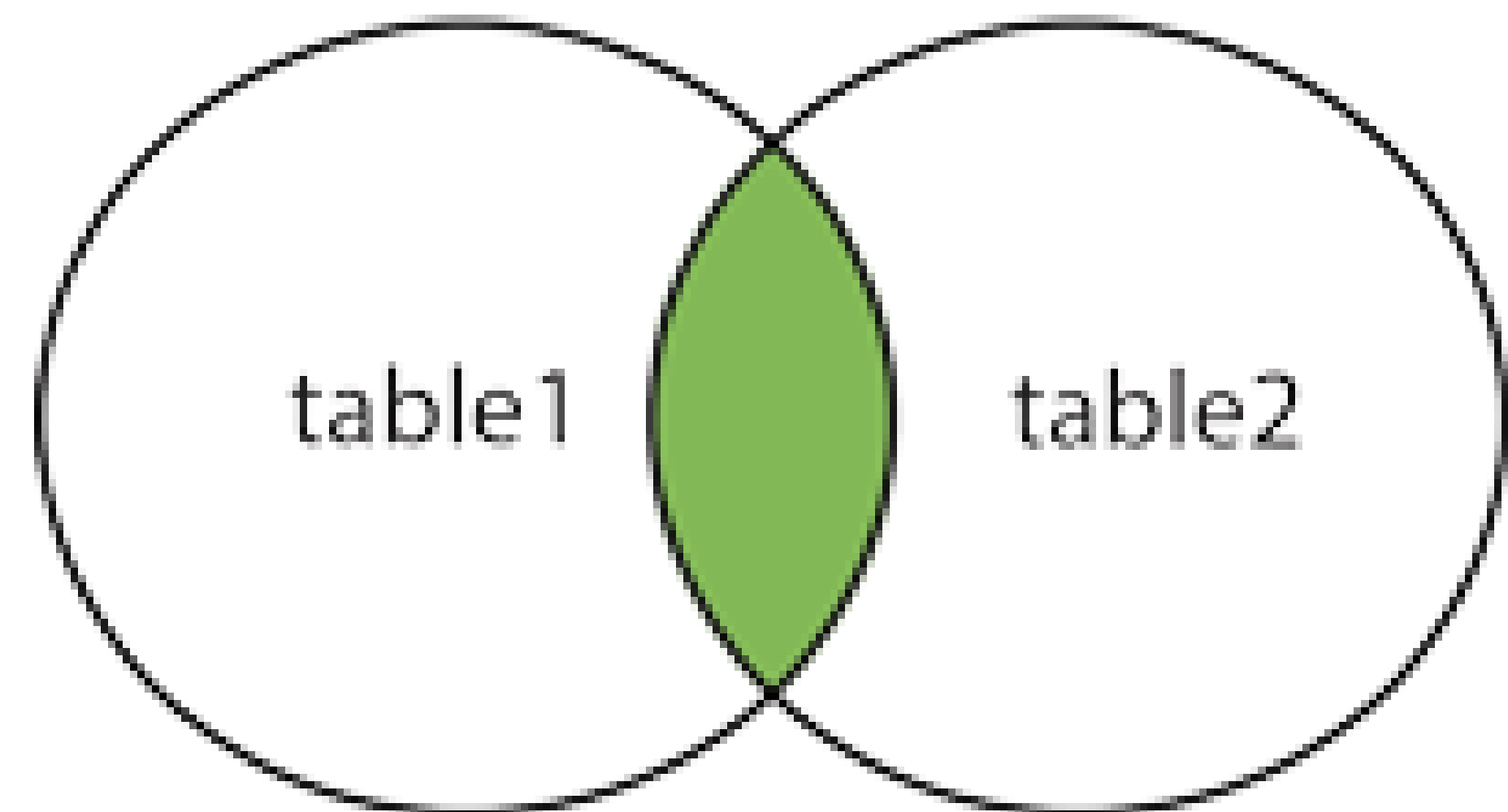
# Łączenie tabel

```
CREATE TABLE customers(  
  customer_id int NOT NULL AUTO_INCREMENT,  
  name varchar(255) NOT NULL,  
  PRIMARY KEY(customer_id)  
);
```

```
CREATE TABLE addresses(  
  address_id int NOT NULL AUTO_INCREMENT,  
  customer_id int,  
  street varchar(255),  
  PRIMARY KEY(address_id)  
);
```

# INNER JOIN

INNER JOIN (lub zwykłe JOIN) jest podstawowym typem łączenia tabel. Jako wynik daje on tylko wiersze, które spełniają podany warunek.



# Join

Założmy że mamy tablice z następującymi danymi:

**SELECT \* FROM customers;**

customer_id	name
1	Jacek
3	Paweł
4	Kuba

**SELECT \* FROM addresses;**

address_id	customer_id	street
1	1	Adres Jacka
2	3	Adres Kuby
3	10	Zły adres

# INNER JOIN

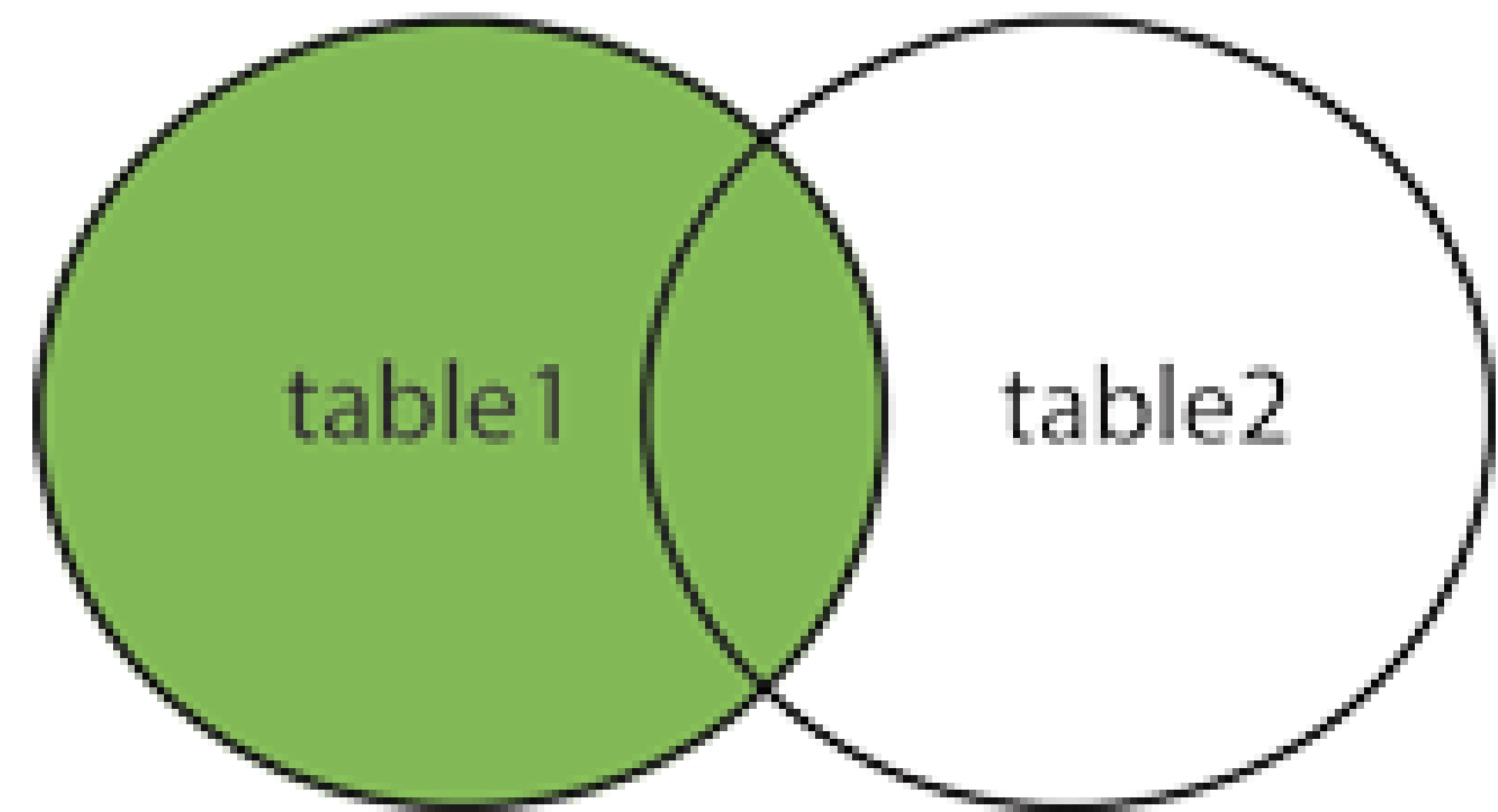
```
SELECT * FROM customers JOIN addresses ON  
customers.customer_id=addresses.customer_id;
```

customer_id	name	address_id	customer_id	street
1	Jacek	1	1	Adres Jacka
3	Paweł	2	3	Adres Kuby



# LEFT JOIN

LEFT JOIN zwraca jako wynik wszystkie wiersze z lewej tabeli. Dane z prawej tabeli zostaną dołączone tylko w rzędach spełniających warunek.



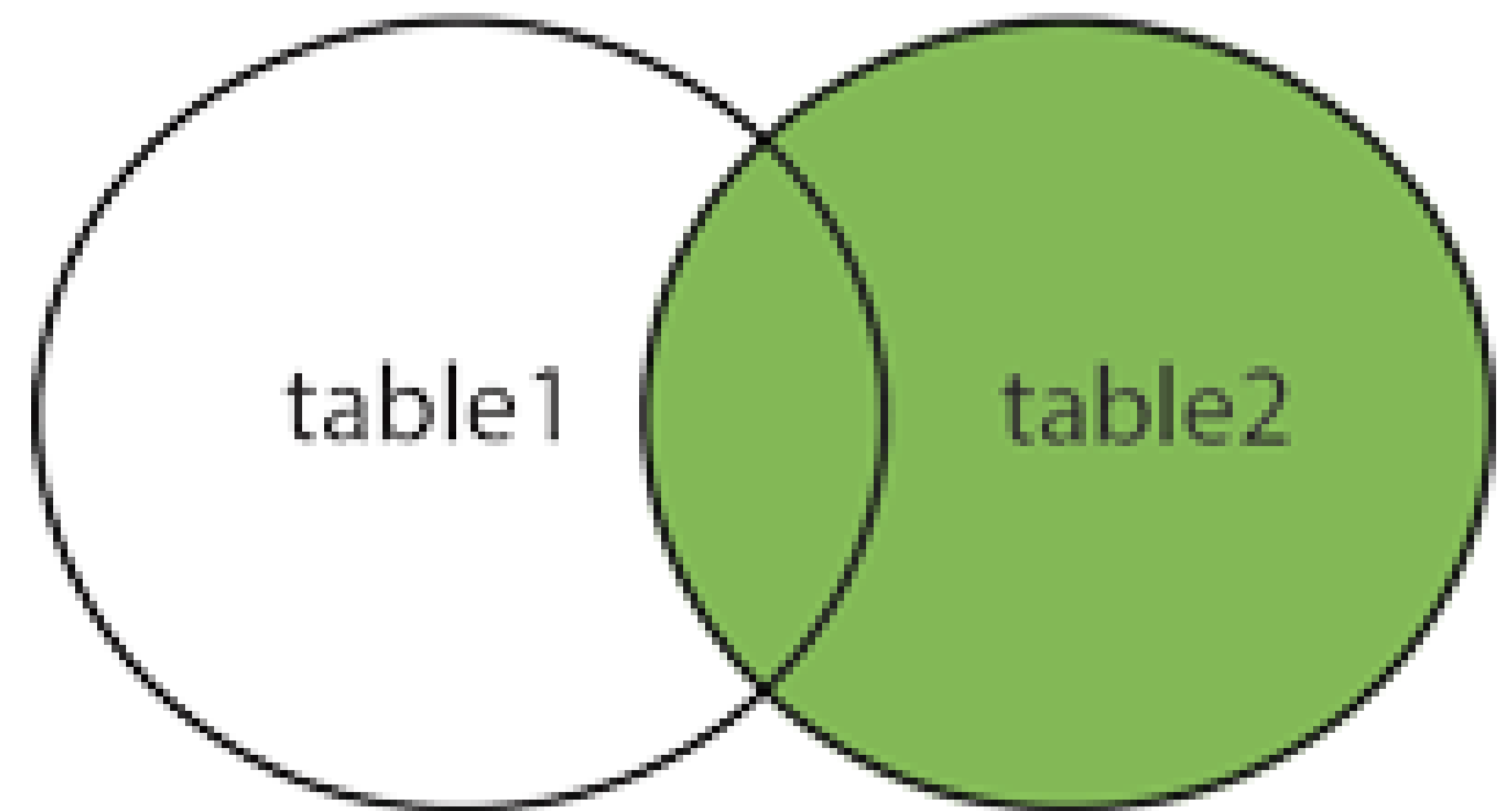
# LEFT JOIN

```
SELECT * FROM customers LEFT JOIN addresses ON  
customers.customer_id=addresses.customer_id;
```

customer_id	name	address_id	customer_id	street
1	Jacek	1	1	Adres Jacka
3	Paweł	2	3	Adres Kuby
4	Kuba	NULL	NULL	NULL

# RIGHT JOIN

RIGHT JOIN zwraca jako wynik wszystkie wiersze z prawej tabeli. Dane z prawej zostaną dołączone tylko w rzędach spełniających warunek.



# RIGHT JOIN

**SELECT \* FROM** customers **RIGHT JOIN** addresses **ON**  
customers.customer\_id=addresses.customer\_id;

customer_id	name	address_id	customer_id	street
1	Jacek	1	1	Adres Jacka
3	Paweł	2	3	Adres Kuby
NULL	NULL	3	10	Adres błędny



# Relacje między tabelami

# Relacja jeden do jednego

Relacja, w której jeden element z danej tabeli może być połączony tylko z jednym elementem z innej tabeli.

Klient może mieć tylko jeden adres.  
Adres musi mieć jednego klienta.




# Relacja jeden do jednego

Relację jeden do jednego tworzymy przez uzależnienie klucza głównego wpisu od klucza głównego drugiego obiektu. Klucz główny nie może istnieć bez tej relacji. W naszym przypadku będzie to wpisywanie klucza głównego klienta jako klucza głównego adresu.

```
CREATE TABLE customers(  
  customer_id int NOT NULL AUTO_INCREMENT,  
  name varchar(255) NOT NULL,  
  PRIMARY KEY(customer_id)  
);
```

```
CREATE TABLE addresses(  
  customer_id int NOT NULL,  
  street varchar(255),  
  PRIMARY KEY(customer_id),  
  FOREIGN KEY(customer_id) REFERENCES customers(customer_id)  
  ON DELETE CASCADE  
);
```



**ON DELETE CASCADE** jest opcjonalnym parametrem. Nie musi go być.

# FOREIGN KEY

- Atrybut FOREIGN KEY dopisany do jakiejś kolumny mówi po prostu, że ta kolumna wskazuje na klucz główny innej tabelki.
- Przyspiesza on pracę naszej bazy danych i powoduje zabezpieczenia przed wprowadzeniem niepoprawnych danych (np. nie pozwoli wpisać tam klucza który nie występuje w drugiej tabeli).



# ON DELETE CASCADE

Podczas budowania relacji możemy dodać jeszcze opcję **ON DELETE CASCADE**. Opcja ta powoduje że usunięcie rzędu w tabelce automatycznie spowoduje usunięcie wszystkich rzędów w innych tabelkach które są z nim połączone jakąś relacją.

Np. Jeżeli usuwamy użytkownika to chcemy żeby wszystkie jego wiadomości w systemie zostały wyrzucone razem z nim.

Jeżeli nie dodamy tej opcji to SQL nie pozwoli nam usunąć rzędu dopóki są z nim powiązane jakiekolwiek wpisy w innych tabelkach.

Np. SQL nie pozwoli nam usunąć użytkownika dopóki w tabelce z wiadomościami znajdują się rzędy przypisane do niego.

# Relacja jeden do jednego

```
INSERT INTO customers(name) VALUES ("Janusz"), ("Kuba"), ("Wojtek");
```

```
INSERT INTO addresses(customer_id, street) VALUES (1, "Ulica Janusza"), (2, "Ulica Kuby");
```

```
SELECT * FROM customers JOIN addresses ON  
customers.customer_id=addresses.customer_id  
WHERE customers.customer_id=2;
```

customer_id	name	customer_id	street
2	Kuba	2	Ulica Kuby

# FOREIGN KEY – dodawanie elementu

```
SELECT * From customers;
```

customer_id	name
1	Jacek
3	Paweł

```
INSERT INTO addresses(customer_id, street) VALUES (5, "xxx");
```

**ERROR 1452 (23000):** Cannot add or update a child row: a foreign key constraint fails ('test'.addresses, CONSTRAINT 'addresses\_ibfk\_1' FOREIGN KEY ('customer\_id') REFERENCES 'customers' ('customer\_id'))

Jeżeli w drugiej tabeli nie ma klucza głównego do którego chcemy się odnieść przez klucz zewnętrzny to SQL zwróci nam błąd

# FOREIGN KEY – usuwanie elementu

**Select \* FROM addresses;**

customer_id	street
3	xxx

**DELETE FROM customers WHERE customer\_id = 3;**

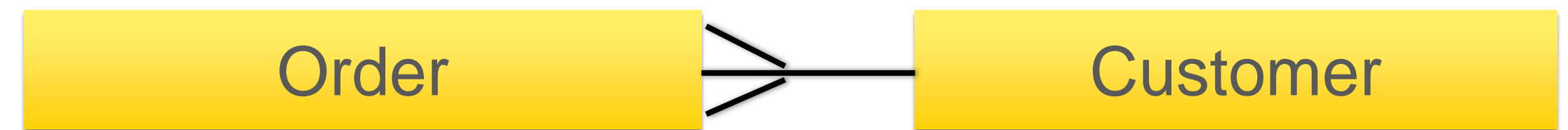
# Czas na zadania

- Przeróbcie ćwiczenia z części E  
Pierwsze dwa ćwiczenia zróbcie z wykładowcą.

# Relacja jeden do wielu

Relacja, w której jeden element z danej tabeli, może być połączony z wieloma elementami z innej tabeli.

Klient może mieć wiele zamówień.  
Zamówienie musi mieć tylko jednego klienta.



# Relacja jeden do wielu

Relację jeden do wielu tworzymy przez dodanie dodatkowej kolumny, w której trzymamy klucz główny obiektu z drugiej tabeli.

```
CREATE TABLE orders(  
  order_id int NOT NULL AUTO_INCREMENT,  
  customer_id int NOT NULL,  
  order_details varchar(255),  
  PRIMARY KEY(order_id),  
  FOREIGN KEY(customer_id)  
  REFERENCES customers(customer_id)  
);
```

# Relacja jeden do wielu

```
INSERT INTO orders(customer_id, order_details) VALUES (3, "Zamowienie1"), (3, "Zamowienie2"),  
                                                       (1, "Zamowienie3");
```

```
SELECT * FROM customers JOIN orders  
ON customers.customer_id=orders.customer_id  
WHERE customers.customer_id=3;
```

customer_id	name	order_id	customer_id	order_details
3	Wojtek	1	3	Zamówienie1
3	Wojtek	2	3	Zamówienie2



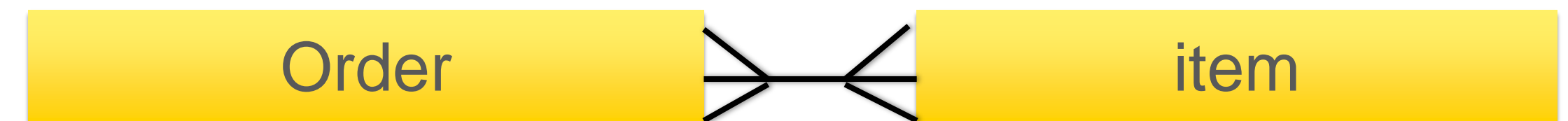
# Czas na zadania

- Przeróbcie ćwiczenia z części F  
Pierwsze dwa ćwiczenia zróbcie z wykładowcą.

# Relacja wiele do wielu

Relacja, w której wiele elementów z danej tabeli może być połączonych z wieloma elementami z innej tabeli.

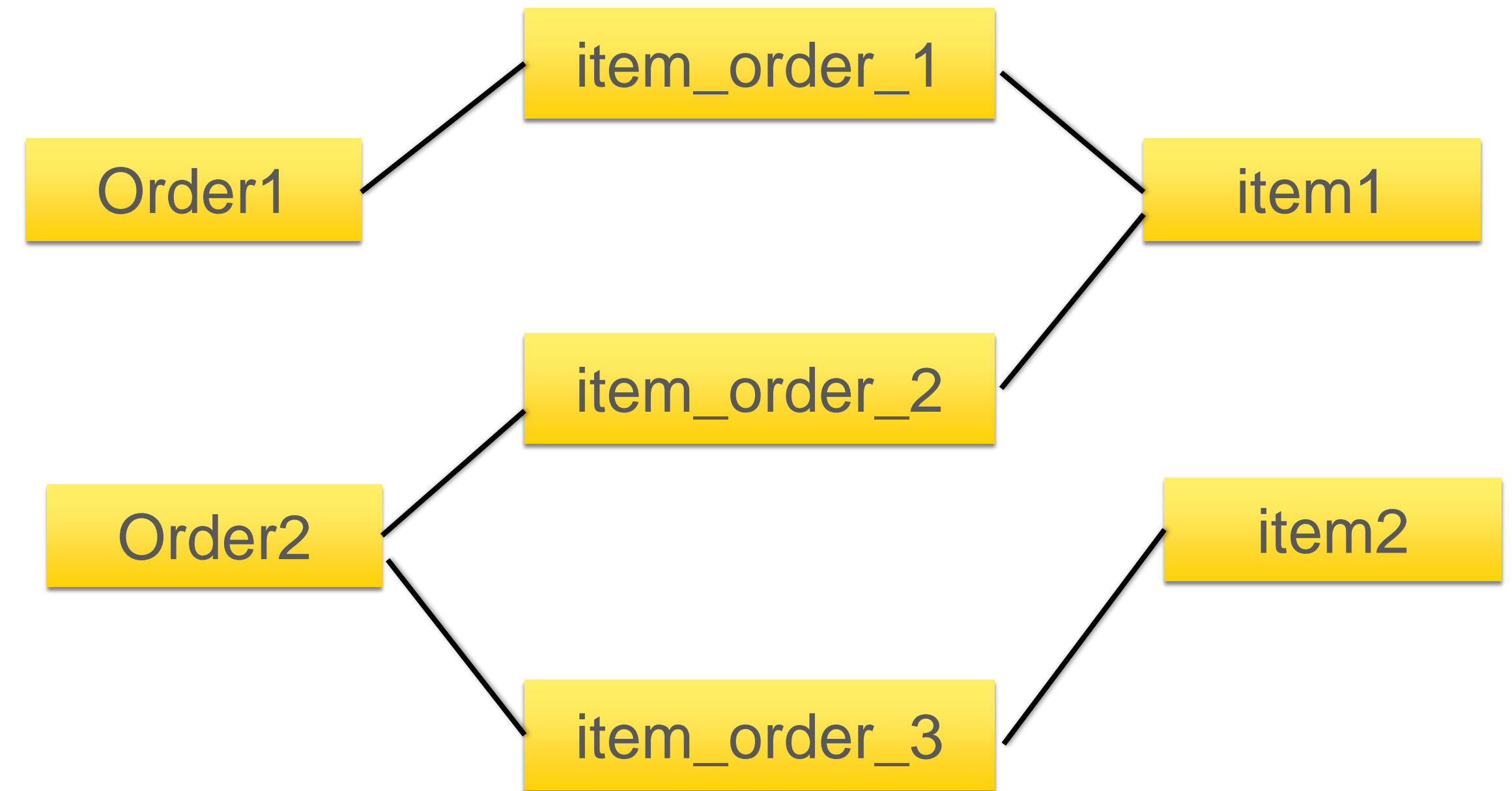
Na przykład zamówienie ma w sobie wiele przedmiotów, przedmiot może być w wielu zamówieniach.



# Relacja wiele do wielu

W SQL nie da się zrobić czegoś takiego jak relacja wiele do wielu. Relację wiele do wielu stworzymy przez dodanie dodatkowej tabeli, która opisuje nam taką relację i która posiada dwie relacje jeden do wielu.

Tabela ta może trzymać więcej informacji niż tylko kucze główne swoich relacji.



# Relacja wiele do wielu

```
CREATE TABLE items(  
    item_id int NOT NULL AUTO_INCREMENT,  
    description varchar(255),  
    PRIMARY KEY(item_id)  
);
```

```
INSERT INTO items(description) VALUES ("Item 1"), ("Item 2"), ("Items 3");
```

# Relacja wiele do wielu

```
CREATE TABLE items_orders(  
  id int AUTO_INCREMENT,  
  item_id int NOT NULL,  
  order_id int NOT NULL,  
  PRIMARY KEY(id),  
  FOREIGN KEY(order_id) REFERENCES orders(order_id),  
  FOREIGN KEY(item_id) REFERENCES items(item_id)  
);
```

# Relacja wiele do wielu

```
INSERT INTO items_orders(order_id, item_id) VALUES (1,1), (2,1), (2,2);
```

```
SELECT * FROM orders  
  JOIN items_orders ON orders.order_id=items_orders.order_id;  
  JOIN items ON items.item_id=items_orders.item_id;
```

order_id	customer_id	order_details	item_id	order_id	item_id	description
1	3	Zamówienie1	1	1	1	Item 1
2	3	Zamówienie2	1	2	1	Item 1
2	3	Zamówienie2	2	2	2	Item 2

# Czas na zadania

- Przeróbcie ćwiczenia z części G  
Pierwsze dwa ćwiczenia zróbcie z wykładowcą.

# Zaawansowany SQL



# SQL Injection

SQL Injection jest bardzo częstym atakiem na bazy danych. Polega on na przekazaniu danych z formularza, które mają w sobie zapytanie SQL.

Przeanalizujmy taki kod:

```
$userName = _POST["name"];  
$sql = "SELECT * FROM users WHERE name=".$userName;  
$result = $conn->query($sql);  
if ($result->num_rows > 0) {  
    ...  
}
```

# SQL Injection

Co się stanie, jeżeli ktoś wpisał do naszego formularza taką wartość:

**"xxx"; DROP TABLE users;**

Nasze zapytanie SQL będzie wyglądać następująco:

**SELECT \* FROM users WHERE name="xxx";  
DROP TABLE users;**

# Zabezpieczenie przed SQL Injection

Istnieją dwa sposoby zabezpieczania się przed tego typu atakami:

- Używanie funkcji czyszczących specjalne znaki z naszego inputu.
- Używanie **prepared statements**.

# Czyszczenie specjalnych znaków

Czyszczenie znaków specjalnych dla SQL odbywa się przez wywołanie metody **real\_escape\_string()** na obiekcie połączenia.

Zmienna zwrócona przez tę metodę może być bezpiecznie użyta w zapytaniach SQL.

```
$userName = $conn->real_escape_string($userName);
```

# Prepared statements

**Prepared statements** to funkcjonalność PHP, dzięki której możemy przygotować szablon zapytania SQL. Następnie taki szablon wypełniamy odpowiednimi danymi i uruchamiamy.

Zalety **prepared statements**:

- dużo szybsze niż normalne zapytania,
- bezpieczne na ataki SQL Injection.

# Przygotowanie – prepared statements

Obiekt **prepared statements** tworzymy przez użycie metody **prepare()** na obiekcie połączenia. Metoda ta zwróci FALSE, gdy nie powiedzie się stworzenie takiego zapytania. W miejsca w które potem podepniemy dane wstawiamy znak zapytania.

Tutaj pojawią się dane które będziemy bindować do zapytania

```
$statement = $mysqli->prepare("INSERT INTO customers(name) VALUES (?");
```

```
if ($statement === FALSE) {  
    echo "Błąd: (" . $mysqli->errno . ") " . $mysqli->error;  
}
```

# Używanie prepared statements

Do wcześniej przygotowanego obiektu **prepared statement** wpisujemy dane (jest to bindowanie danych). Następnie wywołujemy zapytanie.

Bindowanie danych do **prepared statements** polega na wywołaniu metody **bind\_param()** na obiekcie zapytania.

```
bool mysqli_stmt::bind_param(string $types, mixed &$var1 [, mixed &$... ])
```

```
$stmt = $mysqli->prepare(  
    "INSERT INTO customers(name, age) VALUES (?,?)");
```

```
$stmt->bind_param('sd', $name, $age);
```

Bindowane parametry pojawiają się  
w naszym zapytaniu



# Bindowanie danych do prepared statements

Bindowanie danych do **prepared statements** polega na wywołaniu metody **bind\_param()** na obiekcie zapytania.

```
bool mysqli_stmt::bind_param(string $types, mixed &$var1 [, mixed &$... ])
```

```
$stmt = $mysqli->prepare(  
    "INSERT INTO customers(name, age) VALUES (?,?)");
```

```
$stmt->bind_param('sd', $name, $age);
```



# Bindowanie danych – typy

i	Wartość przekazana będzie zmienną liczbową całkowitą
d	Wartość przekazana będzie zmienną liczbową zmiennoprzecinkową
s	Wartość przekazana będzie napisem
b	Wartość przekazana będzie BLOB (będzie wysyłana w paczkach)

# Wywołanie – prepared statements

Wywołanie wcześniej przygotowanego wyrażenia polega na użyciu metody **execute()** na obiekcie **prepared statement**.

Metoda ta zwraca TRUE lub FALSE.

```
if (!$statement->execute()) {  
    echo "Błąd: (" . $stmt->errno . ") " . $stmt->error;  
}
```

# Funkcje wbudowane w SQL

Język SQL implementuje również wiele funkcji ułatwiających pracę na napisach, liczbach i datach.

Pełną listę tych funkcji (wraz z opisami) możecie znaleźć tutaj:

[http://www.w3schools.com/sql/sql\\_functions.asp](http://www.w3schools.com/sql/sql_functions.asp)

# Indeksy

Indeksy są specjalnymi tabelami przeszukań przyspieszającymi przeszukiwanie tabeli względem jednej z kolumn.

Powinniśmy ich używać, gdy wiele klauzur **WHERE** zależy właśnie od tej kolumny.

Zbyt duża ilość indeksów może spowolnić działanie bazy danych – dlatego trzeba na nie uważać i dodawać je po dogłębnej analizie bazy danych.

```
CREATE INDEX index_name  
ON table_name (column_name);
```

# Wyzwalacze (triggers)

W SQL jest możliwość stworzenia wyzwalaczy (**triggerów**).

Są to funkcje, które zostaną automatycznie uruchomione, jeżeli zajdzie określona przez nas sytuacja (zazwyczaj DELETE, INSERT, DELETE).

Więcej o wyzwalaczach:

- [http://www.tutorialspoint.com/plsql/plsql\\_triggers.htm](http://www.tutorialspoint.com/plsql/plsql_triggers.htm)
- <http://www.sqlteam.com/article/an-introduction-to-triggers-part-i>

# Transakcje

SQL pozwala też na transakcje (**transactions**). Jest to grupa zapytań SQL wywoływana w całości na bazie danych. Jeżeli któreś z tych zapytań nie powiedzie się, baza wraca do stanu sprzed takiej transakcji.

Jest to zaawansowany mechanizm, który występuje np. w systemach bankowych.

Więcej o transakcjach znajdziecie tutaj:

- <http://www.tutorialspoint.com/sql/sql-transactions.htm>
- <http://www.sqlteam.com/article/introduction-to-transactions>