# Core Data Relationships (iOS) – Explained with Diagrams

## Slide 1: Title

**Core Data Relationships**
What they are, why they matter, and how to create them (Visual & Explanatory Guide)

---

## Slide 2: What is Core Data? (In Simple Words)

Core Data is Apple's framework used to **store, manage, and relate data** inside an iOS app.

Think of Core Data as: - A **local database** (like tables & rows) - PLUS an **object graph** (objects connected to each other)

It uses: - **Entities** → Like database tables - **Attributes** → Data fields (name, age, price) - **Relationships** → Connections between entities

---

## Slide 3: Real-World Analogy

Before understanding relationships, think about real life:

```
Person → owns → Car
Author → writes → Books
Customer → places → Orders
```

In apps, data is also connected like real life.
These connections are called **relationships** in Core Data.

---

## Slide 4: What is a Relationship in Core Data?

A **relationship** defines how **one entity is connected to another entity**.

Example: - A `User` can have multiple `Orders` - An `Order` belongs to only one `User`

```
User -------- places --------> Order
```

Relationships allow Core Data to understand **ownership and navigation** between objects.

---

## Slide 5: Why Do We Need Relationships?

Without relationships: - Data becomes duplicated - Hard to track connections - Complex queries

With relationships: - Clean data structure - Easy navigation between objects - Real-world modeling

```
User → Orders → OrderItems
```

One object can naturally reach another object.

---

## Slide 6: Types of Relationships (Overview)

Core Data supports three main types:

1️⃣One-to-One (1:1) 2️⃣One-to-Many (1:N) 3️⃣Many-to-Many (M:N)

Each type matches a real-life scenario.

---

## Slide 7: One-to-One Relationship (1:1)

One object is connected to **exactly one** other object.

Example: - `User ↔️ Profile`

```
User -------- Profile
```

✔️One user has one profile
✔️One profile belongs to one user

---

## Slide 8: One-to-Many Relationship (1:N)

One object is connected to **multiple objects**.

Example: - `User → Orders`

```
User
 ├── Order 1
 ├── Order 2
 └── Order 3
```

✔️One user can place many orders
✔️Each order belongs to one user

---

## Slide 9: Many-to-Many Relationship (M:N)

Many objects are connected to many objects.

Example: - `Student ↔️ Course`

```
Student A ─┬─ Course 1
Student B ─┼─ Course 2
Student C ─┴─ Course 3
```

✔️One student can enroll in many courses
✔️One course can have many students

---

## Slide 10: Inverse Relationships (Very Important)

Every relationship should have an **inverse relationship**.

Example: - `User.orders` ↔ `Order.user`

```
User ─── orders ──> Order
User <── user  <─── Order
```

Why inverse matters: - Keeps data consistent - Updates both sides automatically - Prevents memory & data bugs

---

## Slide 11: How to Create Relationships in Xcode

Steps in **.xcdatamodeld**: 1. Select an entity (e.g., User) 2. Add a new **Relationship** 3. Set destination entity (Order) 4. Choose relationship type (To-One / To-Many) 5. Set the inverse relationship

Repeat for the other entity.

---

## Slide 12: Relationship Settings Explained

Key settings you must understand:

- **Optional** → Can this relationship be nil?
- **To-Many** → One or many objects?
- **Ordered** → Maintain order or not
- **Delete Rule** → What happens on deletion

These control data safety and behavior.

---

## Slide 13: Delete Rules (With Meaning)

When a parent object is deleted:

- **Nullify** → Child remains, link removed
- **Cascade** → Child objects are deleted
- **Deny** → Prevent deletion if child exists
- **No Action** → Unsafe (avoid using)

☸️Most common: **Cascade**

---

## Slide 14: Accessing Relationships in Code

Core Data allows direct navigation:

```
// From User to Orders
let orders = user.orders

// From Order to User
let user = order.user
```

No SQL, no joins — Core Data handles everything.

---

## Slide 15: Final Summary (One-Glance Understanding)

- Relationships connect entities
- Match real-world logic

- Always set inverse relationships
- Choose correct delete rules
- Use diagrams to design first

✅Good relationships = clean architecture