

TypeDB Fundamentals Lecture 4

17th January 2024

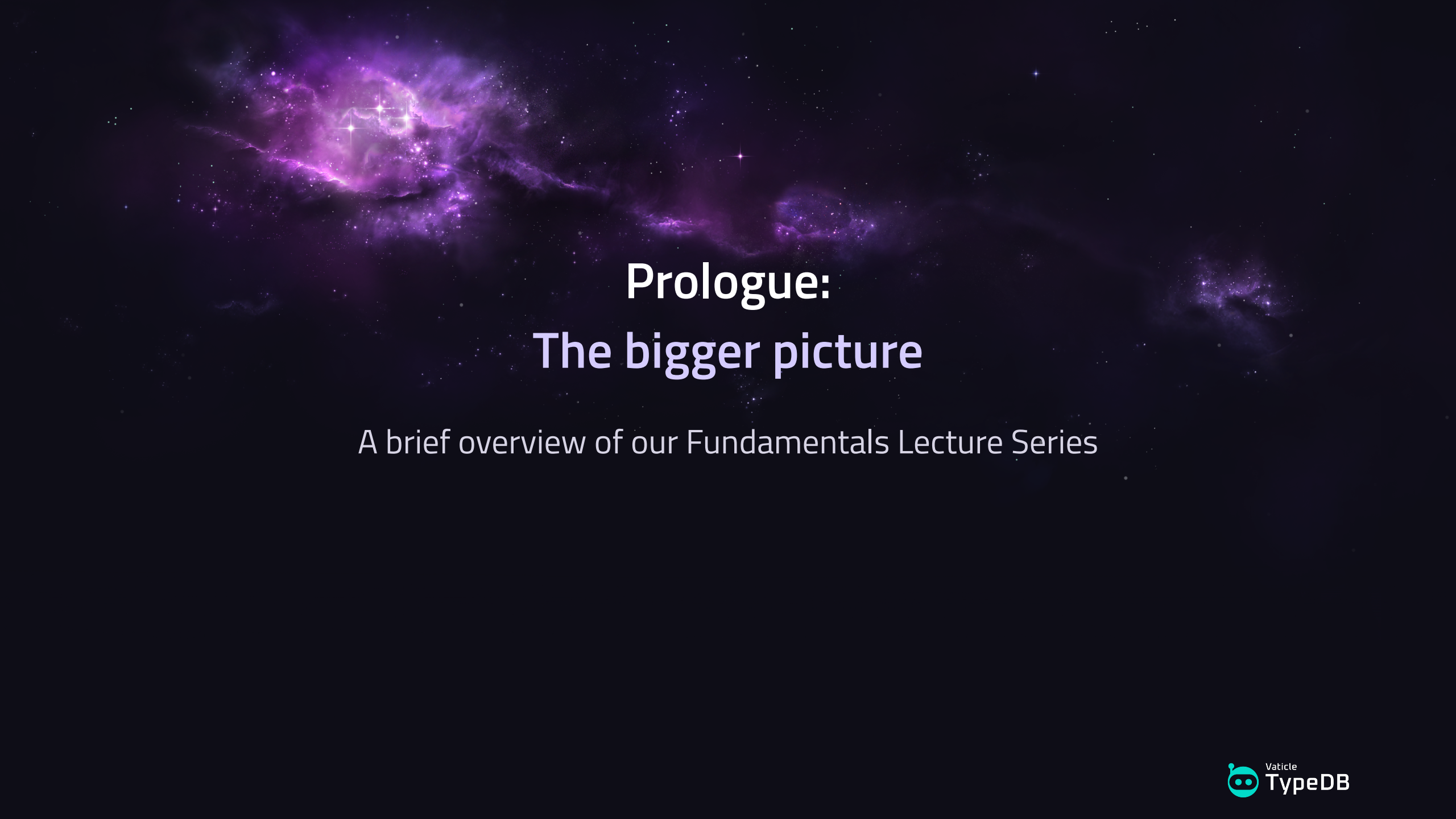
# The Polymorphic Data Model With Types



**Dr. Christoph Dorn**

Head of Research, Vaticle

Previously: Theoretical Computer Scientist in Category Theory  
@ Oxford University

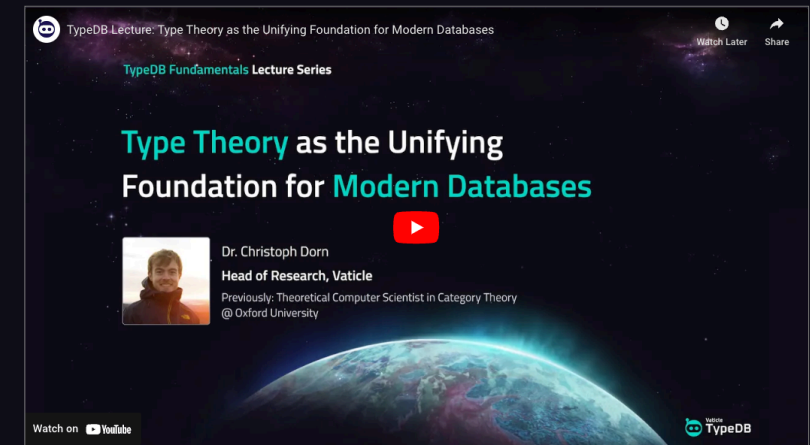


# Prologue: The bigger picture

A brief overview of our Fundamentals Lecture Series

# Lecture #1: The **theoretical** motivation

- Databases lack behind in providing *high-level “zero-cost” abstractions* as now found in many modern programming languages
- Type systems force developers to *“think before they write”*, making applications not only more robust, but also more **composable** and **maintainable**
- With type theory succeeding predicate logic, *“types as queries”* is an ideal modern querying paradigm
- ... and it provides a framework for *polymorphism*



Watch at [TypeDB.com/lectures](https://TypeDB.com/lectures)

# Lecture #2: The **pragmatic pain points**

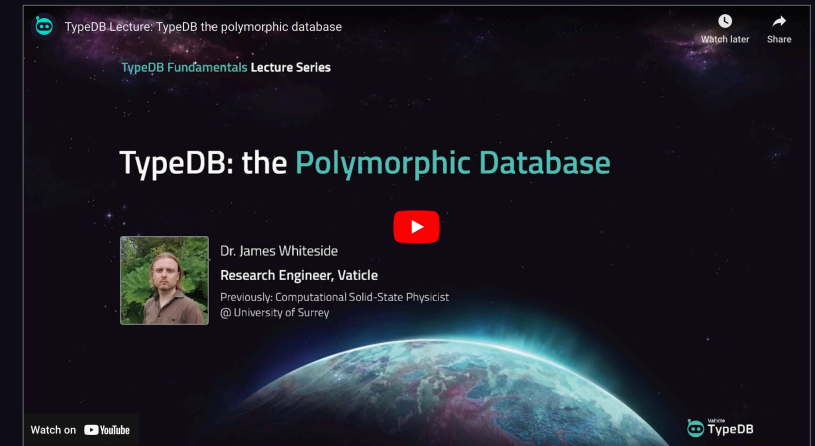
- Regular **mismatches** between aged (or domain-specific) data modeling paradigms and modern programming language paradigms
  - Object-relational mismatch
  - Object-graph mismatch
  - Object-document mismatch
- **Data Integrity**, lack of **Polymorphic Querying**...
- Wrappers, like ORMs, often yield **costly abstractions** (non-optimal queries, additional object representations)



Watch at [TypeDB.com/lectures](https://typedb.com/lectures)

# Lecture #3: The **polymorphic database**

- **TypeDB** builds directly on a novel *polymorphic* data model, equipped with type system that enables:
  - an expressive query language that can leverage **inheritance and interface polymorphism**
  - principled database engineering, which is **maintainable and robust** by design
  - complex applications that are **modular and composable**, continuously modifiable



Watch at [TypeDB.com/lectures](https://TypeDB.com/lectures)

*Today's lecture:* a deep dive into TypeDB's data model!

## Lecture #4 overview

- Key **modeling terminology**: *concepts, instances, dependencies*
- *Polymorphic entity-relation-attribute (PERA)* model via TypeQL
- **Comparison** of PERA to other data model
- Pattern-based **querying** and **reasoning**



# Part I: Concepts and instances

Illustrated guide to key terminology in conceptual modeling

# No meaning without **concepts**

“qwerty”

*Literal value*

Data defined in an abstract  
mathematical framework



# No meaning without **concepts**

“qwerty”

could represent a

Plain-text **password** of a user

**Message** between Ana and Bob

**Keyboard layout** in our system

*Literal value*

Data defined in an abstract  
mathematical framework

# No meaning without **concepts**

“qwerty”

could represent a

- Plain-text **password** of a user
- Message** between Ana and Bob
- Keyboard layout** in our system

*Literal value*

Data defined in an abstract  
mathematical framework

*Concept*

Meaningful categorization of data  
in the modeler’s framework

# No meaning without **concepts**

“qwerty”

could represent a

- Plain-text **password** of a user
- Message** between Ana and Bob
- Keyboard layout** in our system

*Literal value*

Data defined in an abstract  
mathematical framework

*Concept*

Meaningful categorization of data  
in the modeler’s framework

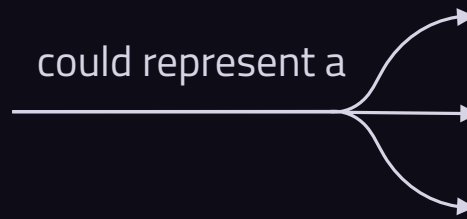
**Definition.** *Concept formation* is the process of sorting data *or* data structures into meaningful classes a.k.a. *types*.

—Hunt, Earl B.. "concept formation". *Encyclopedia Britannica*

# No meaning without **concepts**

“qwerty”

could represent a



Plain-text **password** of a user

**Message** between Ana and Bob

**Keyboard layout** in our system

*Literal value*

Data defined in an abstract  
mathematical framework

*Concept*

Meaningful categorization of data  
in the modeler's framework

**Definition.** *Concept formation* is the process of sorting data or data structures  
into meaningful classes a.k.a. *types*.  
*instances* of types

—Hunt, Earl B.. "concept formation". *Encyclopedia Britannica*

# Two kinds of type instances

# Two kinds of type instances

person	
name	birthday
“Ana”	31-May-1997
“Bob”	01-Feb-1990



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S
3. `birthday` S

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S  $\longrightarrow$  instances are *strings*
3. `birthday` S



# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S  $\longrightarrow$  instances are *strings*
3. `birthday` S  $\longrightarrow$  instances are *dates*

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

*Are persons the sum of names and birthdays?* ←

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

*Are persons the sum of names and birthdays?* ←

**No.**

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

*Are persons the sum of names and birthdays?* ←

**No.**

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990
NULL	NULL

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

*Are persons the sum of names and birthdays?* ←

**No.**

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990
NULL	NULL

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

*Are persons the sum of names and birthdays?* ←

**No.** Instances of `person` are *rows*!

# Two kinds of type instances



Tables are a *specific* example...  
Definitions apply to data models  
in general as we will later see!

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990
NULL	NULL

Have concepts (i.e. types) of:

1. `person` S
2. `name` S → instances are *strings*
3. `birthday` S → instances are *dates*

Are persons the sum of names and birthdays? ←

**No.** Instances of `person` are *rows*!

**Definition.** Two kinds of types:

1. Those collecting *literal values* → *attribute types*
2. Those collecting *freely constructible objects* → *object types*

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
NULL	01-Feb-1990



# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

creating **new name** *requires* a **person** instance (i.e. row) to "have that name"

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

creating **new name** requires a **person** instance (i.e. row) to "have that name"

employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1

NOT NULL FKs

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

creating **new name** requires a **person** instance (i.e. row) to "have that name"

employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1

creating **new team\_membership** requires an **employee** and a **team**

NOT NULL FKs

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

creating **new name** requires a **person** instance (i.e. row) to "have that name"

employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1

creating **new team\_membership** requires an **employee** and a **team**

NOT NULL FKs

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

**Definition.** A concept *depends* on another concept if instantiating the former requires references to instances of the latter.

creating **new name** requires a **person** instance (i.e. row) to "have that name"

employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1

creating **new team\_membership** requires an **employee** and a **team**

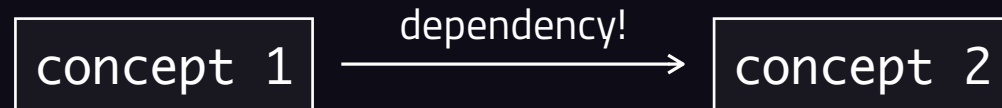
NOT NULL FKs

# The **dependency** hierarchy of concepts

person	
name	birthday
"Ana"	NULL
<del>NULL</del> "Bob"	01-Feb-1990

creating **new name** requires a **person** instance (i.e. row) to "have that name"

**Definition.** A concept *depends* on another concept if instantiating the former requires references to instances of the latter.



employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1

creating **new team\_membership** requires an **employee** and a **team**

NOT NULL FKs

# Conceptualizing dependencies

Schema 1

employee	
emp_id	name
1	"Ana"
2	"Bob"

team	
team_id	team_name
1	"Engineering"
2	"Marketing"

team_membership	
employee	team
1	1
2	1



# Conceptualizing dependencies

Schema 1


employee		team	
emp_id	name	team_id	team_name
1	"Ana"	1	"Engineering"
2	"Bob"	2	"Marketing"

team_membership	
employee	team
1	1
2	1

Schema 2

employee		team	
emp_id	name	team_name	employees
1	"Ana"	"Engineering"	[1,2]
2	"Bob"	"Marketing"	[...]

  
 NOT NULL

# Conceptualizing dependencies

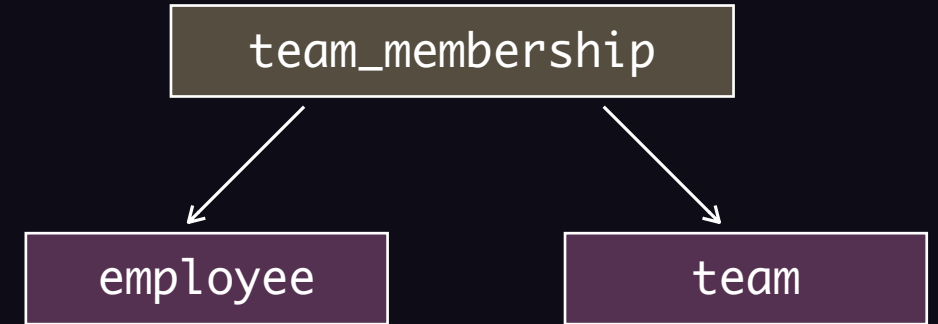
Schema 1

employee		team	
emp_id	name	team_id	team_name
1	"Ana"	1	"Engineering"
2	"Bob"	2	"Marketing"

team_membership	
employee	team
1	1
2	1

*Dependency hierarchy*



Schema 2

employee		team	
emp_id	name	team_name	employees
1	"Ana"	"Engineering"	[1,2]
2	"Bob"	"Marketing"	[...]

}  
 NOT NULL



# Conceptualizing dependencies

Schema 1

employee		team	
emp_id	name	team_id	team_name
1	"Ana"	1	"Engineering"
2	"Bob"	2	"Marketing"

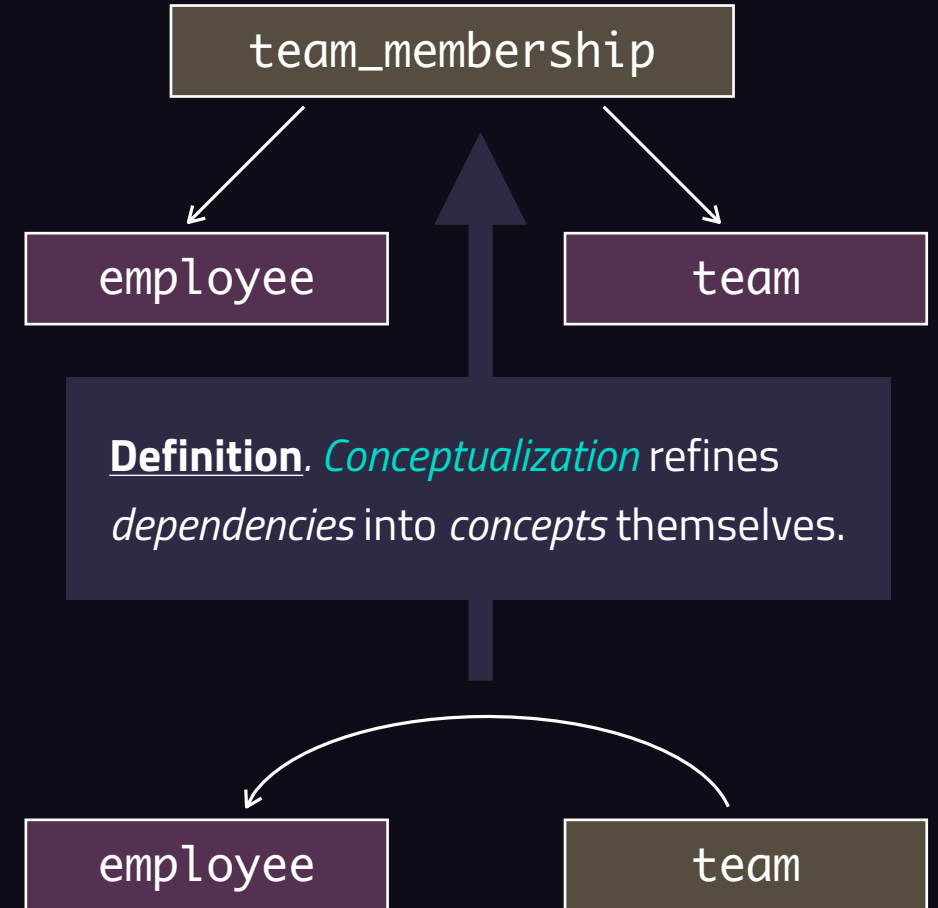
team_membership	
employee	team
1	1
2	1

Schema 2

employee		team	
emp_id	name	team_name	employees
1	"Ana"	"Engineering"	[1,2]
2	"Bob"	"Marketing"	[...]

NOT NULL

## Dependency hierarchy



# Conceptualizing dependencies

Schema 1

employee		team	
emp_id	name	team_id	team_name
1	"Ana"	1	"Engineering"
2	"Bob"	2	"Marketing"

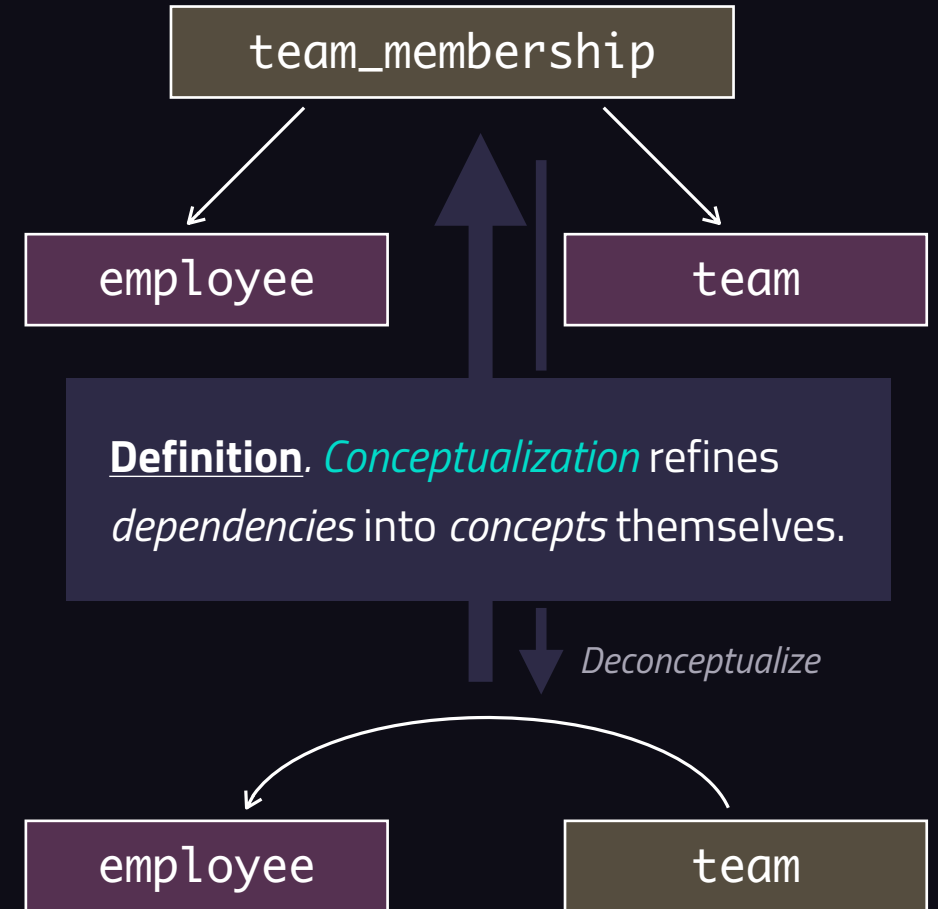
team_membership	
employee	team
1	1
2	1

Schema 2

employee		team	
emp_id	name	team_name	employees
1	"Ana"	"Engineering"	[1,2]
2	"Bob"	"Marketing"	[...]

NOT NULL

## Dependency hierarchy



# Summary

# Summary

- *Concept formation* sorts our data into meaningful *types*

person

name

birthday

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

# Summary

- *Concept formation* sorts our data into meaningful *types*
- Some types can be freely instantiated (*objects types*),  
Others collect pre-defined values (*attribute types*)

person

name

birthday

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

# Summary

- *Concept formation* sorts our data into meaningful *types*

*model allows us to "invent" new instances!*

- Some types can be freely instantiated (**objects types**), Others collect pre-defined values (**attribute types**)

person

name

birthday

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

| "Bob" | 01-Feb-1990 |



# Summary

- *Concept formation* sorts our data into meaningful *types*

- Some types can be freely instantiated (**objects types**),  
Others collect pre-defined values (**attribute types**)

*model allows us to  
"invent" new instances!*

*cannot invent new instances!  
value creation is idempotent*

person

name

birthday

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

| "Bob" | 01-Feb-1990 |  
"Bob"

# Summary

- *Concept formation* sorts our data into meaningful *types*

- Some types can be freely instantiated (*objects types*), Others collect pre-defined values (*attribute types*)

*model allows us to "invent" new instances!*

*cannot invent new instances!  
value creation is idempotent*

- *Dependencies* indicate instantiation of types must reference instances of other types

person

name

birthday

person	
name	birthday
"Ana"	31-May-1997
"Bob"	01-Feb-1990
"Bob"	01-Feb-1990

| "Bob" | 01-Feb-1990 |  
"Bob"

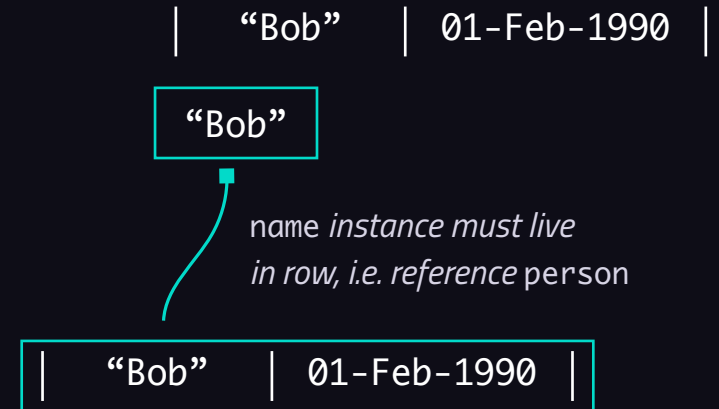
# Summary

- *Concept formation* sorts our data into meaningful *types*
- Some types can be freely instantiated (*objects types*), Others collect pre-defined values (*attribute types*)
- *Dependencies* indicate instantiation of types must reference instances of other types

*model allows us to "invent" new instances!*

*cannot invent new instances!  
value creation is idempotent*

person	person	
name	name	birthday
birthday	"Ana"	31-May-1997
	"Bob"	01-Feb-1990
	"Bob"	01-Feb-1990



# Summary

- *Concept formation* sorts our data into meaningful *types*

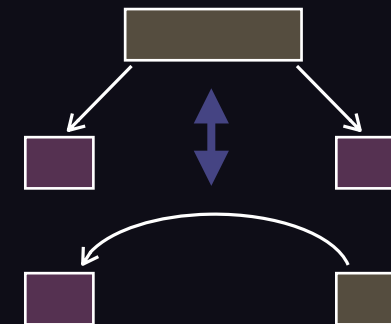
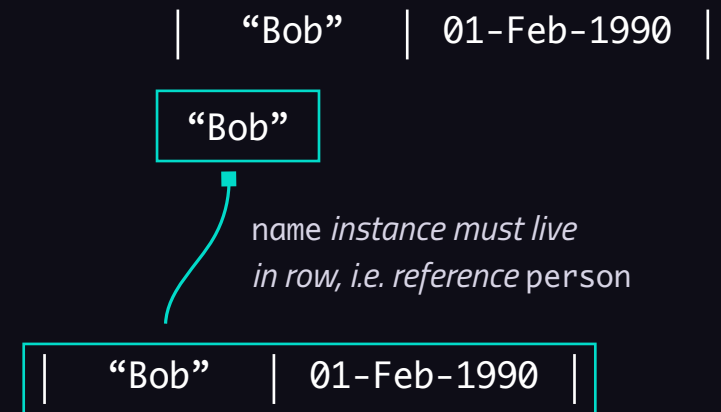
- Some types can be freely instantiated (*objects types*), Others collect pre-defined values (*attribute types*)


*cannot invent new instances!  
value creation is idempotent*

- *Dependencies* indicate instantiation of types must reference instances of other types

- *Conceptualization* turns a dependency into a concept itself

person	person	
name	name	birthday
birthday	"Ana"	31-May-1997
	"Bob"	01-Feb-1990
	"Bob"	01-Feb-1990





# Part II: The polymorphic data model

Exploring the PERA model with TypeQL

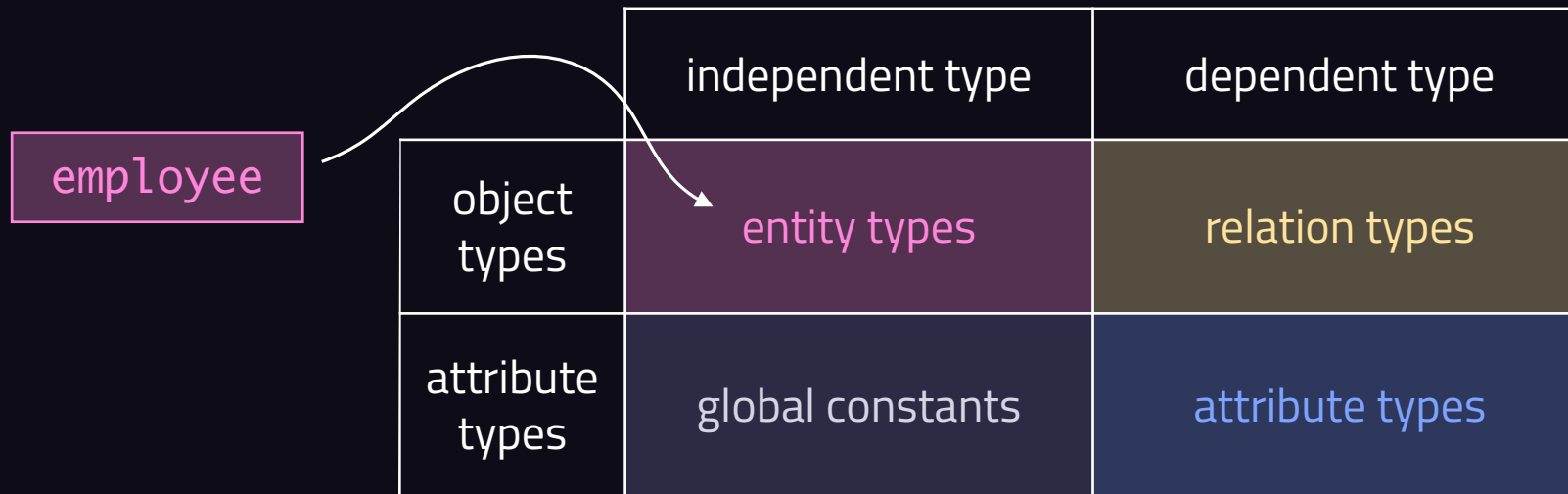
# Introducing the **PERA model** and its **types**

- The **polymorphic entity-relation-attribute** (PERA) builds directly on our previous definitions of types, instances, dependencies in conceptual modeling, with key **kinds of types**:

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	attribute types

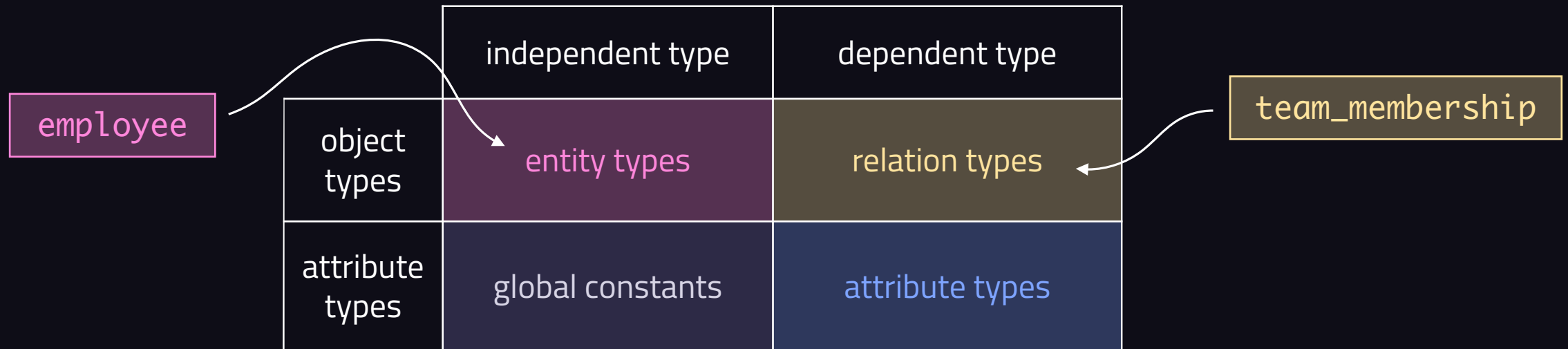
# Introducing the PERA model and its types

- The **polymorphic entity-relation-attribute** (PERA) builds directly on our previous definitions of types, instances, dependencies in conceptual modeling, with key **kinds of types**:



# Introducing the PERA model and its types

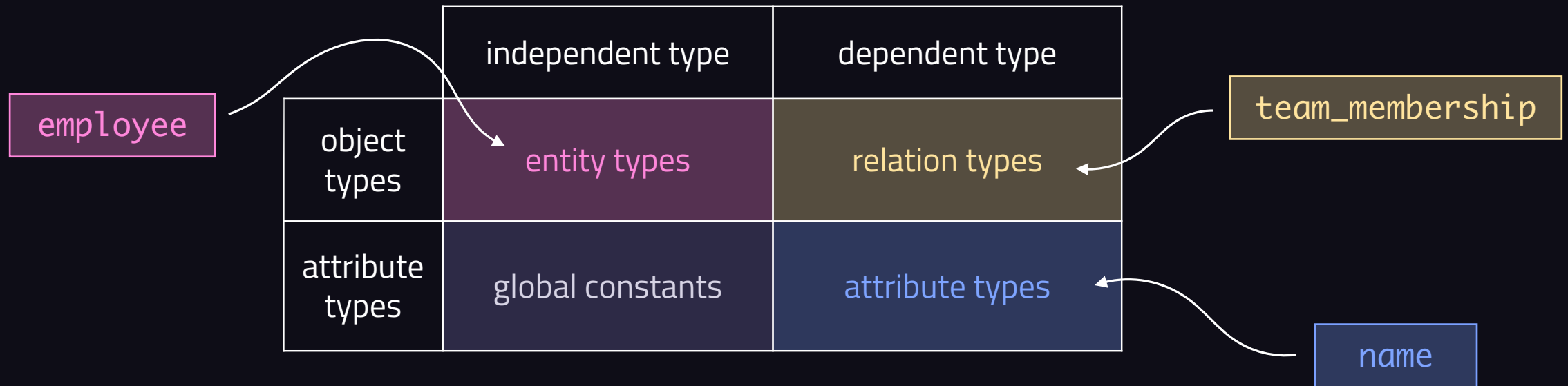
- The **polymorphic entity-relation-attribute** (PERA) builds directly on our previous definitions of types, instances, dependencies in conceptual modeling, with key **kinds of types**:





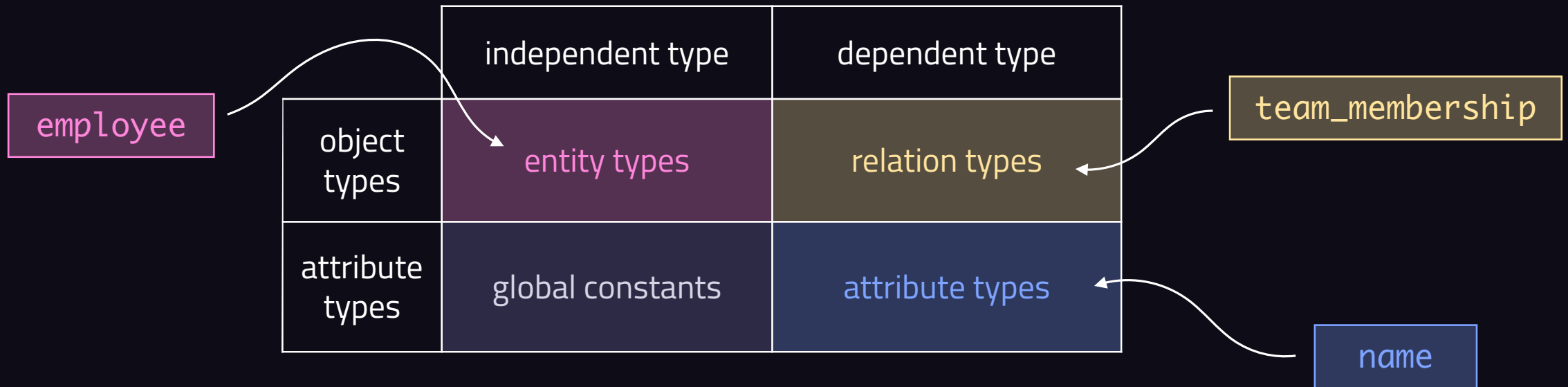
# Introducing the PERA model and its types

- The **polymorphic entity-relation-attribute** (PERA) builds directly on our previous definitions of types, instances, dependencies in conceptual modeling, with key **kinds of types**:



# Introducing the PERA model and its types

- The **polymorphic entity-relation-attribute** (PERA) builds directly on our previous definitions of types, instances, dependencies in conceptual modeling, with key **kinds of types**:



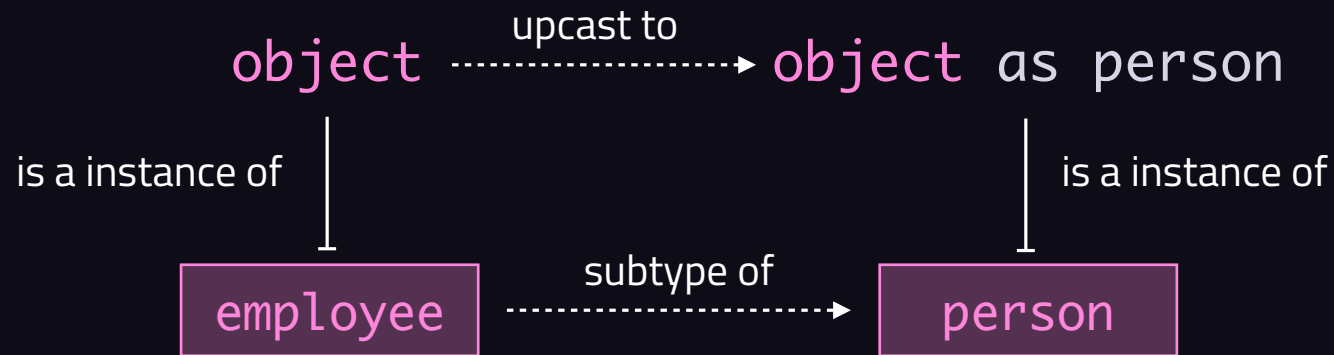
- Incidentally, the above 2 x 2 classification closely matches classical ERA modeling, *however*, for us, it fundamentally derives from simple type-theoretic principles.

# PERA type system: **subtyping**

- Type system defines *types* and *type operations* to pass between types
- A fundamental operation is **subtyping**, allowing to cast type instances:

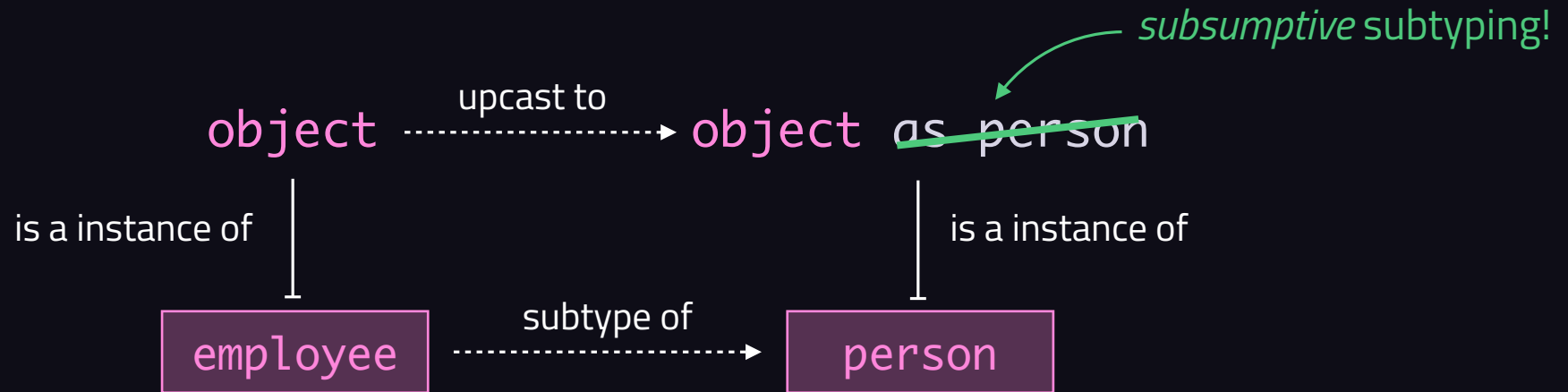
# PERA type system: **subtyping**

- Type system defines *types* and *type operations* to pass between types
- A fundamental operation is **subtyping**, allowing to cast type instances:



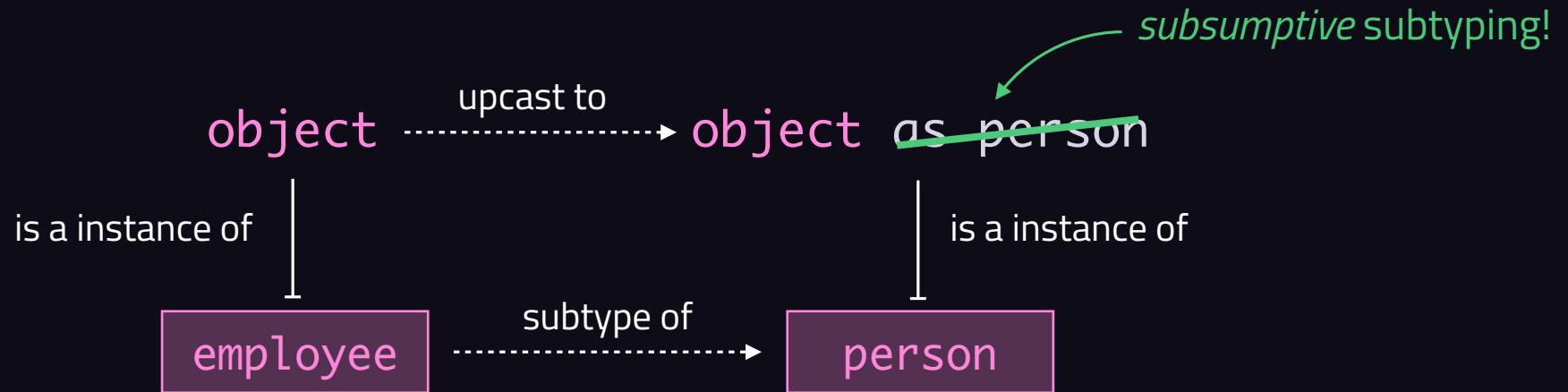
# PERA type system: **subtyping**

- Type system defines *types* and *type operations* to pass between types
- A fundamental operation is **subtyping**, allowing to cast type instances:



# PERA type system: **subtyping**

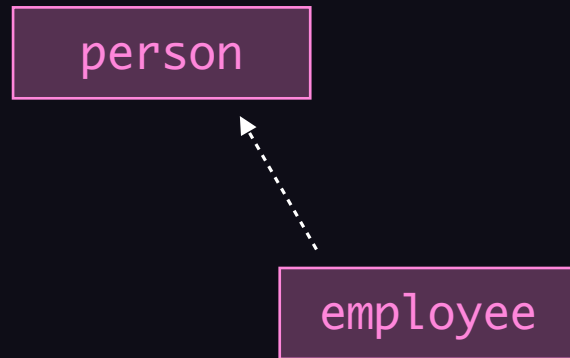
- Type system defines *types* and *type operations* to pass between types
- A fundamental operation is **subtyping**, allowing to cast type instances:



- A name "Ana" in type `name` can be cast into the *bare string* "Ana" in type `string`

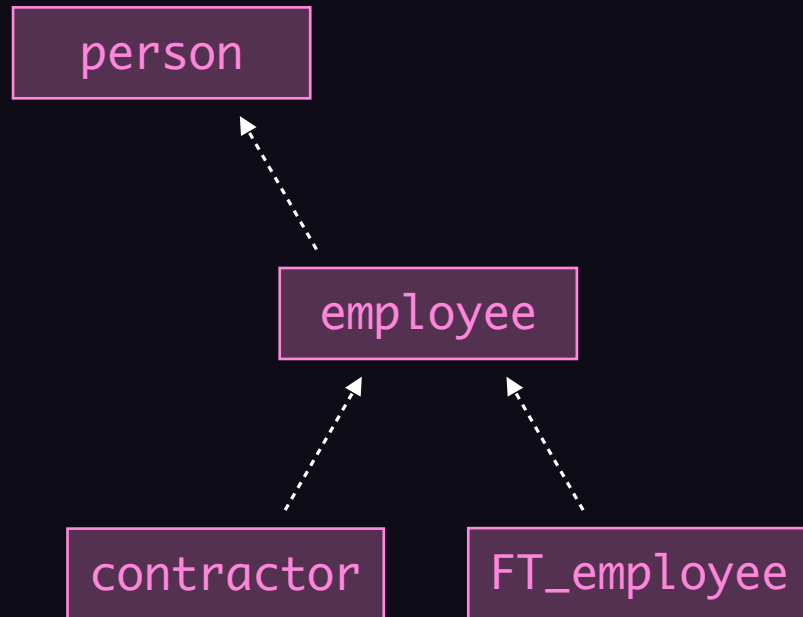
# PERA type system: inheritance polymorphism

- The PERA model supports *inheritance type hierarchies* to describe concept specialization



# PERA type system: inheritance polymorphism

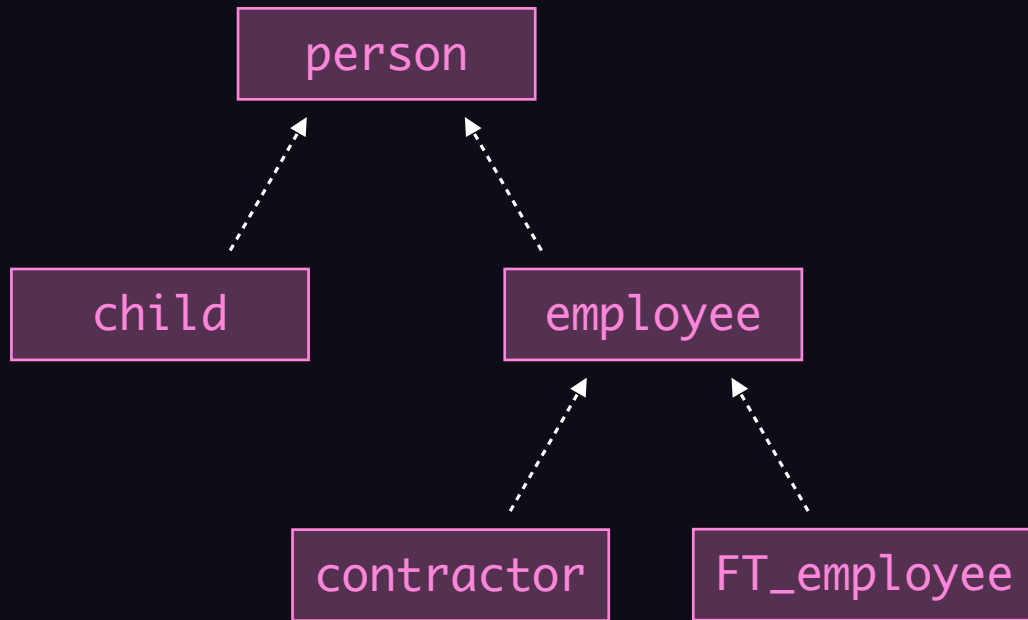
- The PERA model supports *inheritance type hierarchies* to describe concept specialization





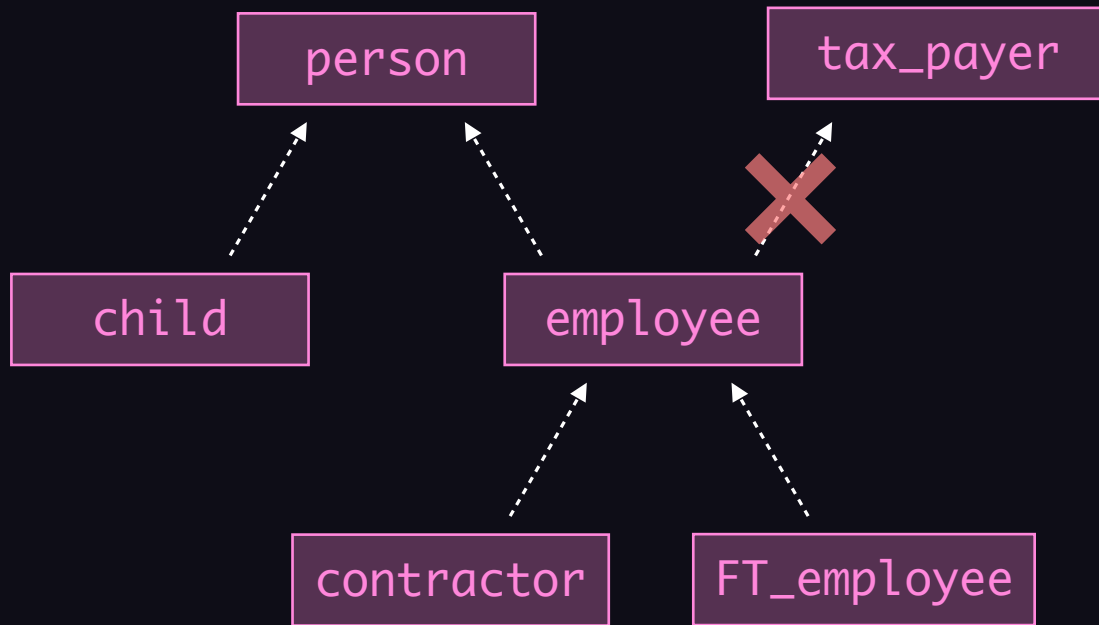
# PERA type system: inheritance polymorphism

- The PERA model supports *inheritance type hierarchies* to describe concept specialization



# PERA type system: inheritance polymorphism

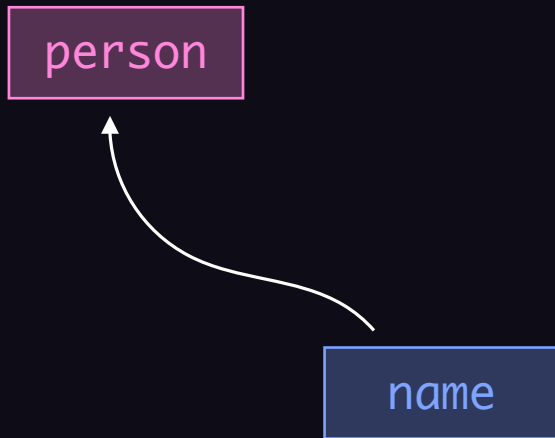
- The PERA model supports *inheritance type hierarchies* to describe concept specialization





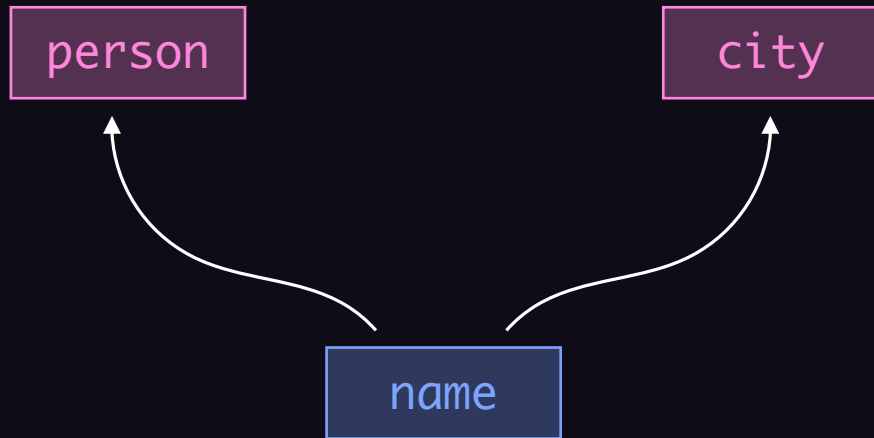
# PERA type system: interface polymorphism

- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



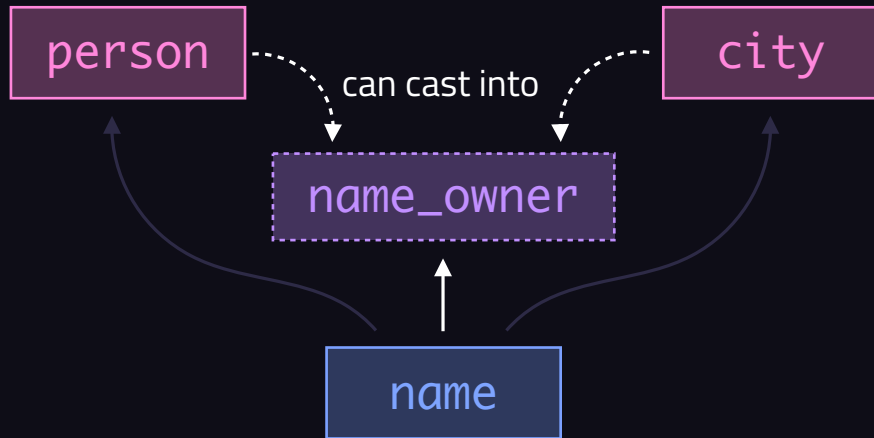
# PERA type system: interface polymorphism

- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



# PERA type system: interface polymorphism

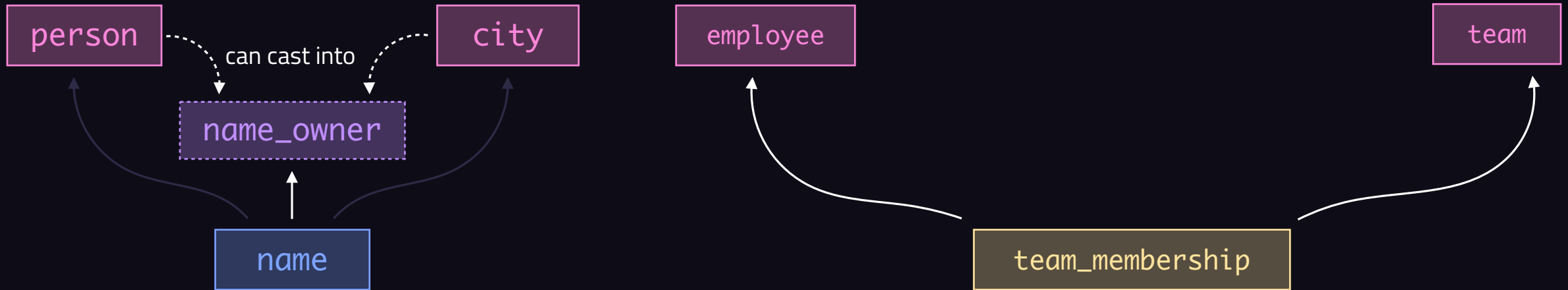
- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



*"cities, like persons, have the capability to be name owners"*

# PERA type system: interface polymorphism

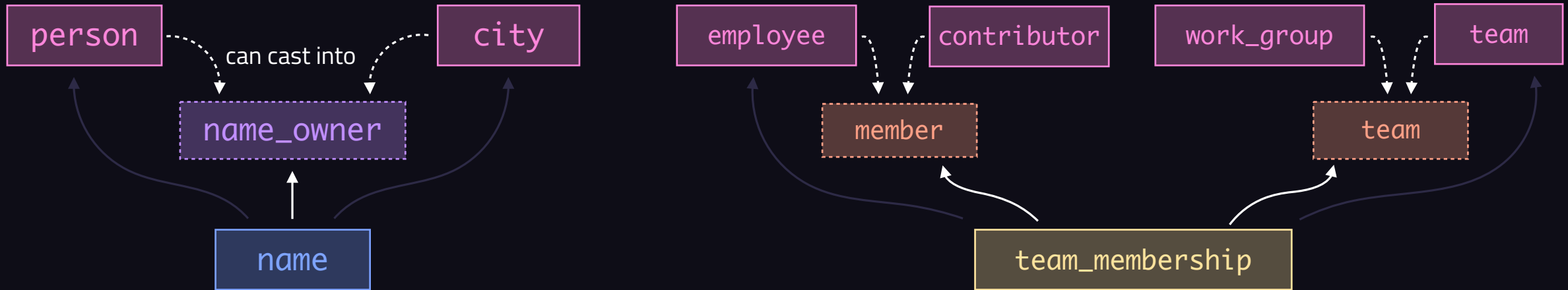
- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



*"cities, like persons, have the capability to be name owners"*

# PERA type system: interface polymorphism

- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



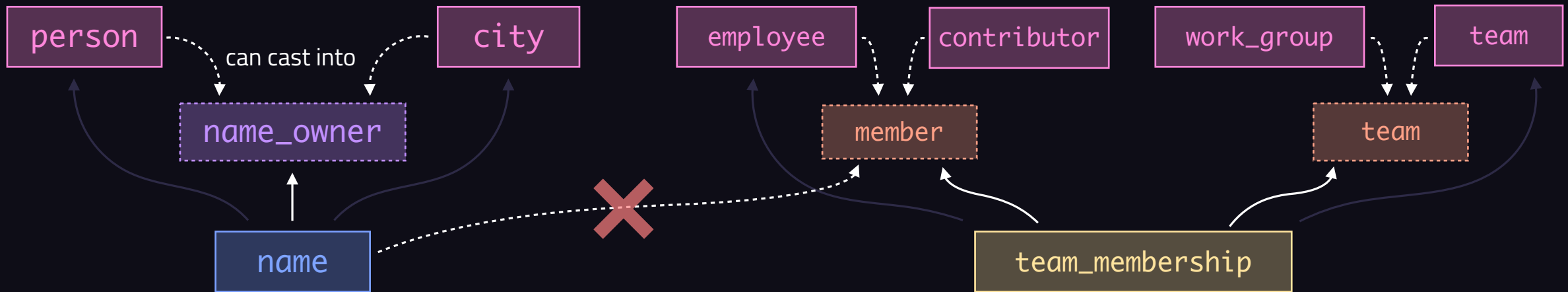
*"cities, like persons, have the capability to be name owners"*

*"contributors, like employees, have the capability to be team members"*



# PERA type system: interface polymorphism

- The PERA model *abstracts type dependencies* as *type capabilities implementable by other types*



*"cities, like persons, have the capability to be name owners"*

*"contributors, like employees, have the capability to be team members"*

**Condition:** the PERA model enforces that

*only object types* can be given capabilities.

*Object types may implement many interfaces!*

→ avoids ambiguity due to idempotent value creation

→ c.f. "single-inheritance"

# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:

```
define person sub entity;
```

# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:

```
define person sub entity;
```

keyword for a *define query*  
(think CREATE TABLE)

# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:

```
define person sub entity;
```

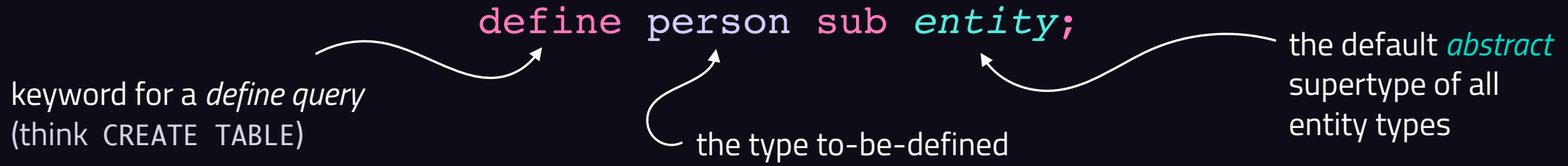
keyword for a *define query*  
(think CREATE TABLE)

the type to-be-defined

# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

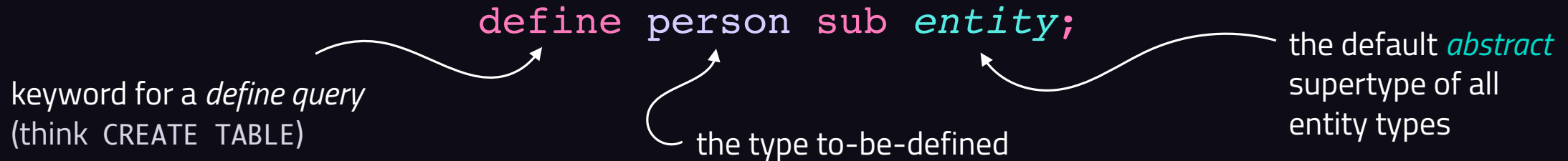
- A new **person** entity type in our schema is defined via:



# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:



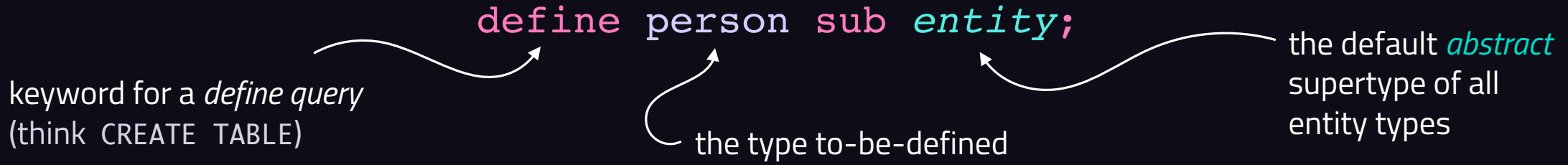
- A new **employee** entity type subtyping **person** is defined via:

```
define employee sub person;
```

# TypeQL practice: **entity types**

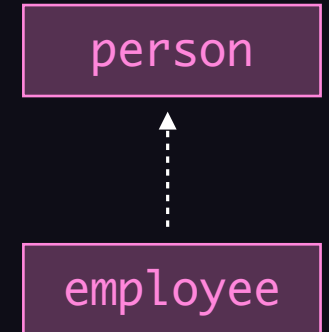
	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:



- A new **employee** entity type subtyping **person** is defined via:

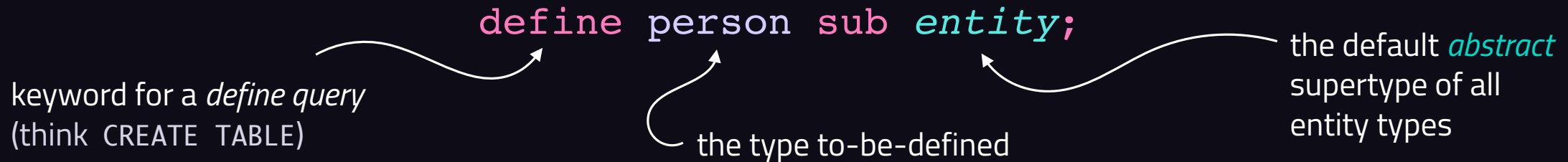
```
define employee sub person;
```



# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:

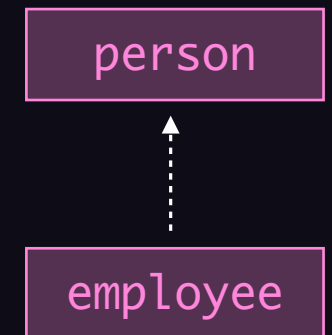


- A new **employee** entity type subtyping **person** is defined via:

```
define employee sub person;
```

- *Abstractness* ensures no instances can be created directly:

```
define person abstract;
```

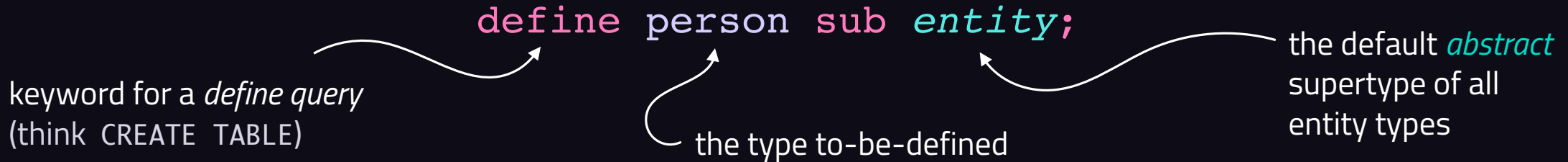




# TypeQL practice: **entity types**

	independent type	dependent type
object types	<b>entity types</b>	relation types
attribute types	global constants	attribute types

- A new **person** entity type in our schema is defined via:

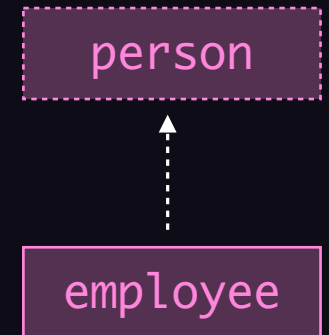


- A new **employee** entity type subtyping **person** is defined via:

```
define employee sub person;
```

- *Abstractness* ensures no instances can be created directly:

```
define person abstract;
```



# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new `marriage` relation type...

```
define
marriage sub relation,
    relates spouse;
```

# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new **marriage** relation type...

```
define
marriage sub relation,
  relates spouse;
```



interfaces of relations are called *roles*

# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new `marriage` relation type...

```
define
marriage sub relation,
  relates spouse;
```

and several subtypes:

```
# case 1: role overwriting
define
hetero_marriage sub marriage,
  relates husband as spouse,
  relates wife as spouse;
```



interfaces of relations are called *roles*

# TypeQL practice: relation types

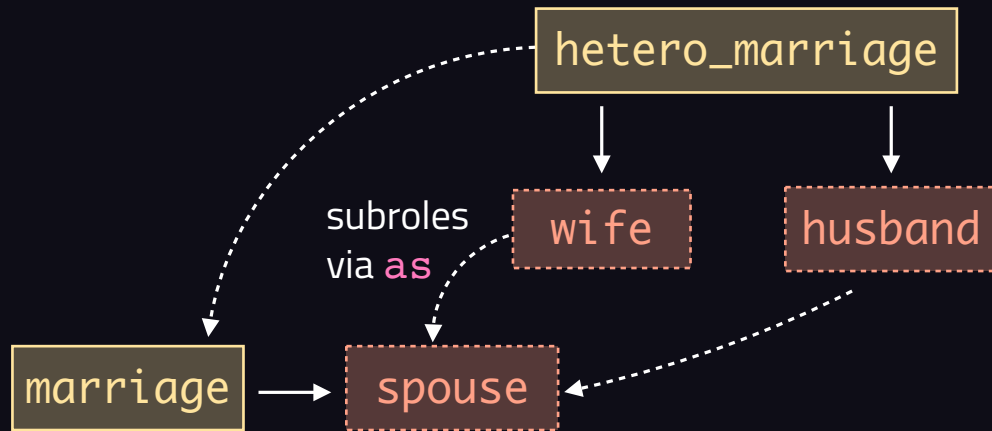
	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new **marriage** relation type...

```
define  
marriage sub relation,  
  relates spouse;
```

and several subtypes:

```
# case 1: role overwriting  
define  
hetero_marriage sub marriage,  
  relates husband as spouse,  
  relates wife as spouse;
```



interfaces of relations are called *roles*

# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

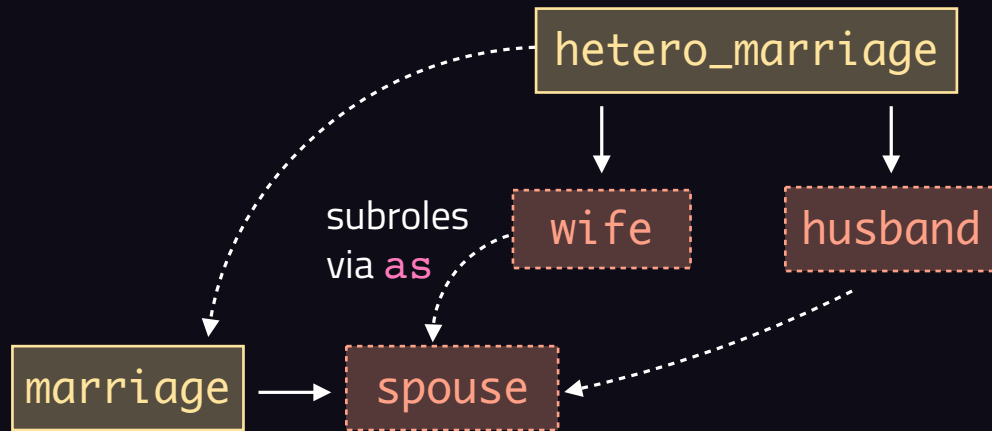
- Define new **marriage** relation type...

```
define  
marriage sub relation,  
  relates spouse;
```

and several subtypes:

```
# case 1: role overwriting  
define  
hetero_marriage sub marriage,  
  relates husband as spouse,  
  relates wife as spouse;
```

```
# case 2: role inheritance  
define  
religious_marriage sub marriage;
```



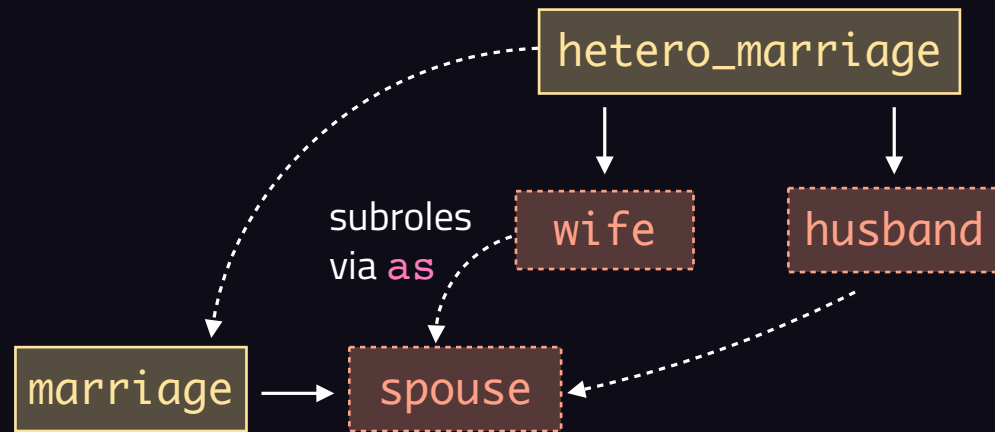
interfaces of relations are called *roles*

# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new **marriage** relation type...

```
define
marriage sub relation,
  relates spouse;
```



interfaces of relations are called *roles*

and several subtypes:

```
# case 1: role overwriting
define
hetero_marriage sub marriage,
  relates husband as spouse,
  relates wife as spouse;
```

```
# case 2: role inheritance
define
religious_marriage sub marriage;
```

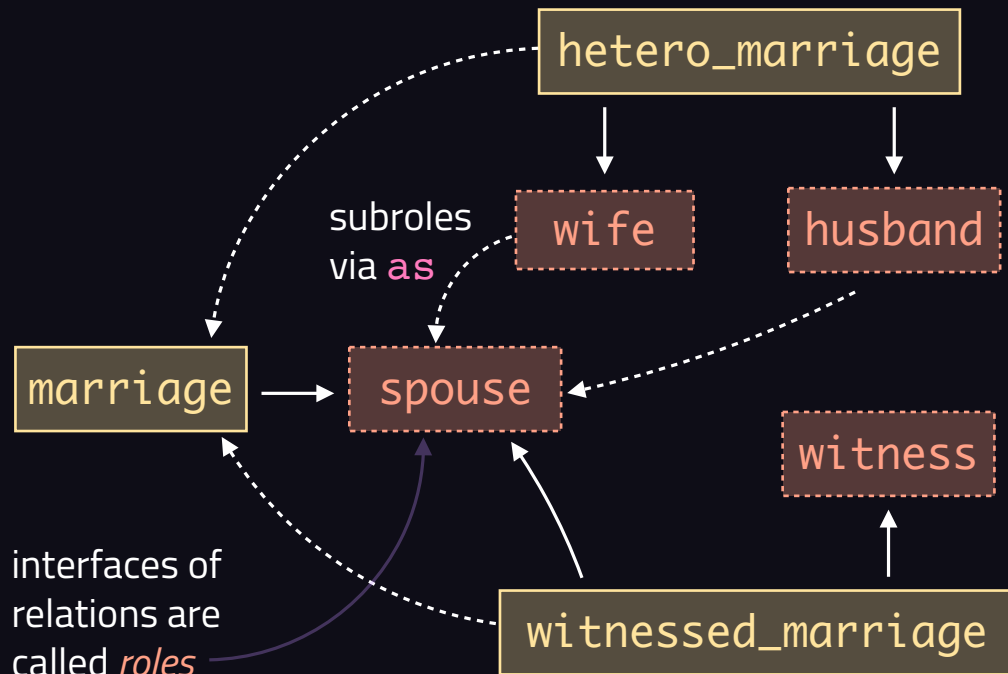
```
# case 3: role extension
define
witnessed_marriage sub marriage,
  relates witness;
```

# TypeQL practice: relation types

	independent type	dependent type
object types	entity types	<b>relation types</b>
attribute types	global constants	attribute types

- Define new `marriage` relation type...

```
define
marriage sub relation,
  relates spouse;
```



and several subtypes:

```
# case 1: role overwriting
define
hetero_marriage sub marriage,
  relates husband as spouse,
  relates wife as spouse;
```

```
# case 2: role inheritance
define
religious_marriage sub marriage;
```

```
# case 3: role extension
define
witnessed_marriage sub marriage,
  relates witness;
```



# TypeQL practice: attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	<b>attribute types</b>

- Define new `name` attribute type, with pre-defined `value` type:

```
define name sub attribute, value string;
```

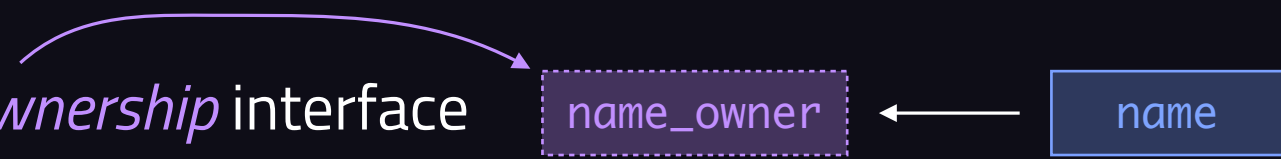
# TypeQL practice: attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	<b>attribute types</b>

- Define new `name` attribute type, with pre-defined `value` type:

```
define name sub attribute, value string;
```

**Condition:** Attribute have a single<sup>1</sup> *ownership* interface  
*...in TypeQL this interface is **implicit!***



<sup>1</sup>Note, *n*-ary attribute types can be modeled of unary attribute types of *n*-ary relation types

# TypeQL practice: attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	<b>attribute types</b>

- Define new `name` attribute type, with pre-defined `value` type:

```
define name sub attribute, value string;
```

**Condition:** Attribute have a single<sup>1</sup> *ownership* interface  
*...in TypeQL this interface is **implicit!***



- Define `first_name` as subtype of `name`

```
define first_name sub name;
```

<sup>1</sup>Note, *n*-ary attribute types can be modeled of unary attribute types of *n*-ary relation types

# TypeQL practice: attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	<b>attribute types</b>

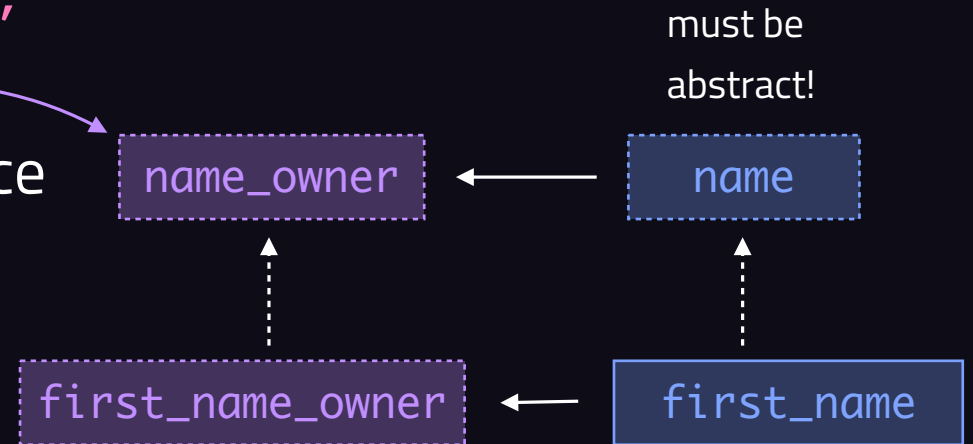
- Define new `name` attribute type, with pre-defined `value` type:

```
define name sub attribute, value string;
```

**Condition:** Attribute have a single<sup>1</sup> *ownership* interface  
*...in TypeQL this interface is **implicit!***

- Define `first_name` as subtype of `name`

```
define first_name sub name;
```



<sup>1</sup>Note, *n*-ary attribute types can be modeled of unary attribute types of *n*-ary relation types

# TypeQL practice: attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	global constants	<b>attribute types</b>

- Define new `name` attribute type, with pre-defined `value` type:

```
define name sub attribute, value string;
```

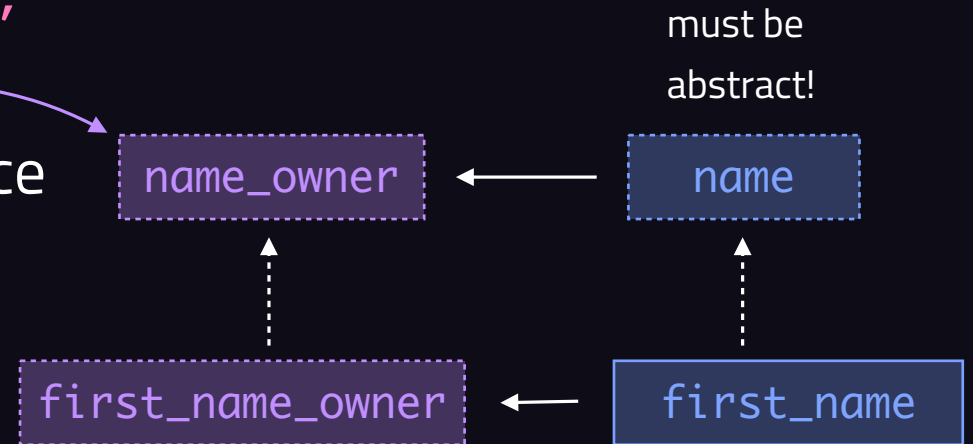
**Condition:** Attribute have a single<sup>1</sup> *ownership* interface  
...in TypeQL this interface is **implicit!**

- Define `first_name` as subtype of `name`

```
define first_name sub name;
```

**Condition:** attribute supertypes must be abstract!

→ avoids ambiguity due to idempotent value creation



<sup>1</sup>Note, *n*-ary attribute types can be modeled of unary attribute types of *n*-ary relation types

# The TypeQL practice: **implementing roles**

- Object types can implement (i.e. "play") roles, defined via:

```
define person plays marriage:spouse;
```

## The TypeQL practice: implementing roles

- Object types can implement (i.e. "play") roles, defined via:

```
define person plays marriage:spouse;
```

the scoped `marriage:spouse` is used here since role identifiers need only be unique *within* relation type hierarchies

# The TypeQL practice: implementing roles

- Object types can implement (i.e. "play") roles, defined via:

```
define person plays marriage:spouse;
```

the scoped `marriage:spouse` is used here since role identifiers need only be unique *within* relation type hierarchies



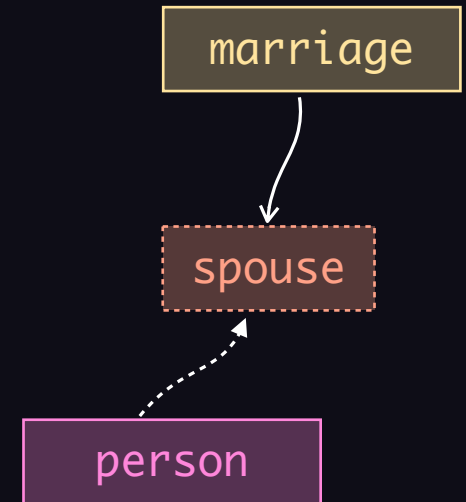


# The TypeQL practice: **implementing roles**

- Object types can implement (i.e. "play") roles, defined via:

```
define person plays marriage:spouse;
```

the scoped `marriage:spouse` is used here since role identifiers need only be unique *within* relation type hierarchies



# The TypeQL practice: implementing roles

the scoped `marriage:spouse` is used here since role identifiers need only be unique *within* relation type hierarchies

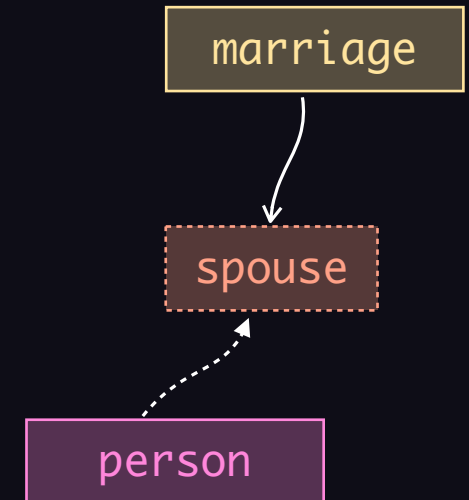
- Object types can implement (i.e. "play") roles, defined via:

```
define person plays marriage:spouse;
```



- Role capabilities are *inherited*

```
define
civil_servant sub person;
registry_entry sub relation,
  relates registrar,
  relates event;
civil_servant plays registry_entry:registrar;
marriage plays registry_entry:event;
```



# The TypeQL practice: implementing roles

- Object types can implement (i.e. "play") roles, defined via:

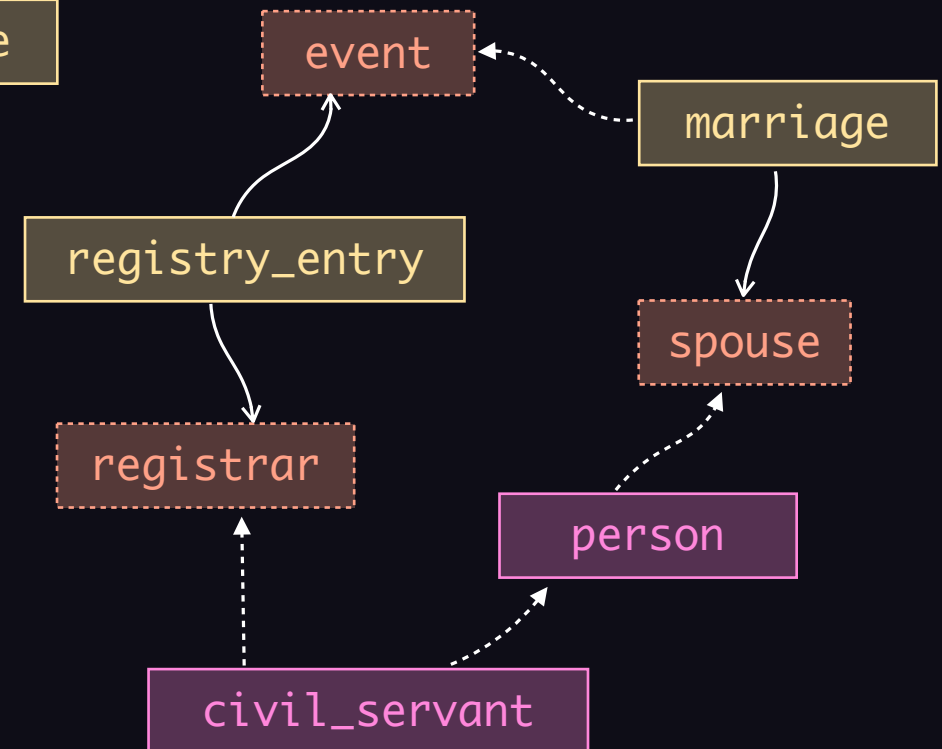
```
define person plays marriage:spouse;
```

the scoped `marriage:spouse` is used here since role identifiers need only be unique *within* relation type hierarchies



- Role capabilities are *inherited*

```
define
civil_servant sub person;
registry_entry sub relation,
  relates registrar,
  relates event;
civil_servant plays registry_entry:registrar;
marriage plays registry_entry:event;
```



# The TypeQL practice: **implementing ownerships**

- Object types can implement ownership of (i.e. "own") attribute types:

```
define person owns name;
```

# The TypeQL practice: implementing ownerships

- Object types can implement ownership of (i.e. "own") attribute types:

```
define person owns name;
```



# The TypeQL practice: implementing ownerships

- Object types can implement ownership of (i.e. "own") attribute types:

```
define person owns name;
```



```
define  
date sub attribute, value datetime;  
marriage owns date;
```



# The TypeQL practice: implementing ownerships

- Object types can implement ownership of (i.e. "own") attribute types:

```
define person owns name;
```



```
define  
date sub attribute, value datetime;  
marriage owns date;
```

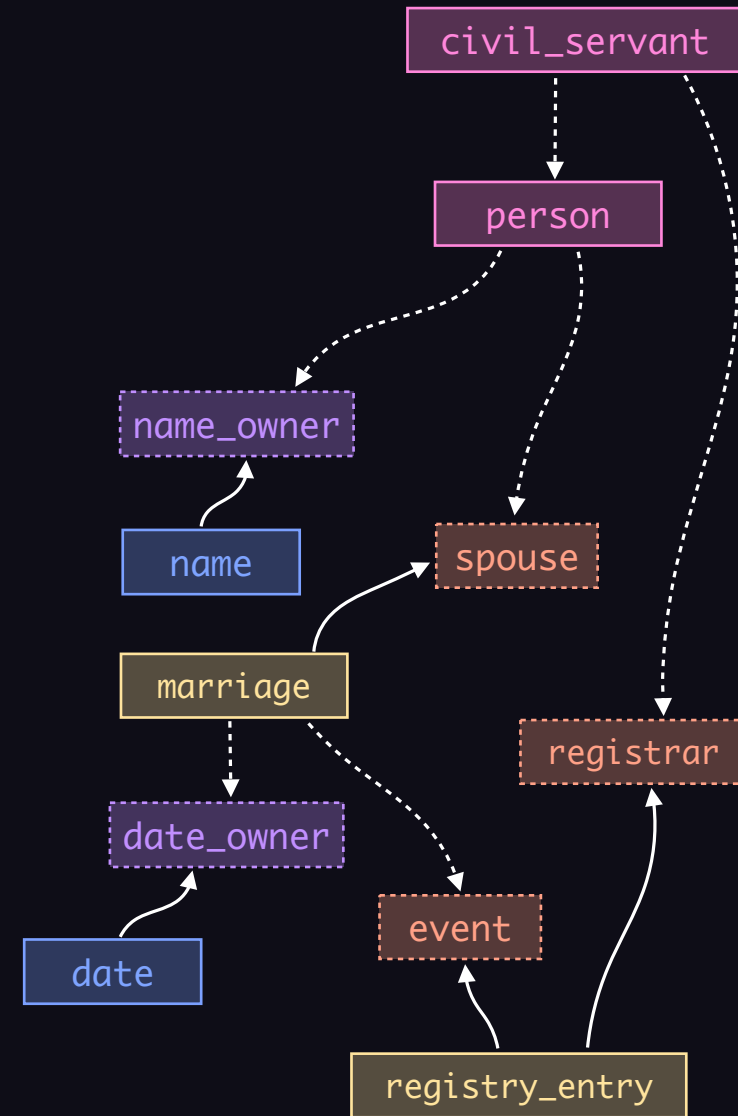


- Like for roles, ownership capabilities are *inherited*

# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```

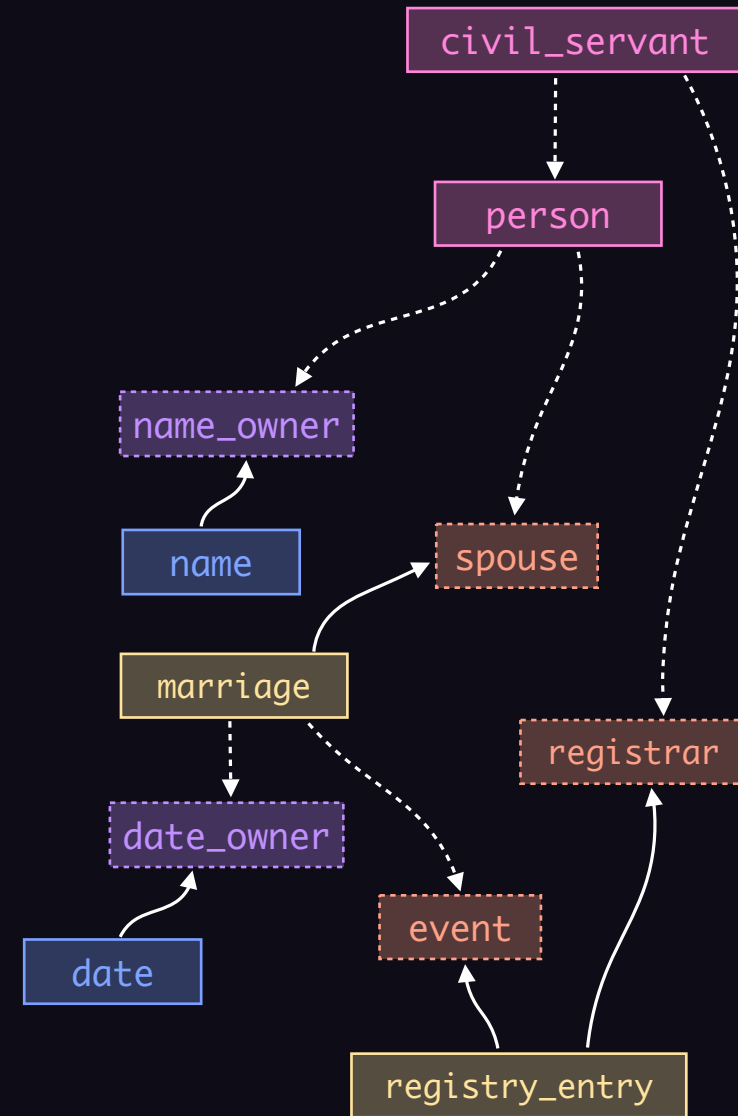




# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

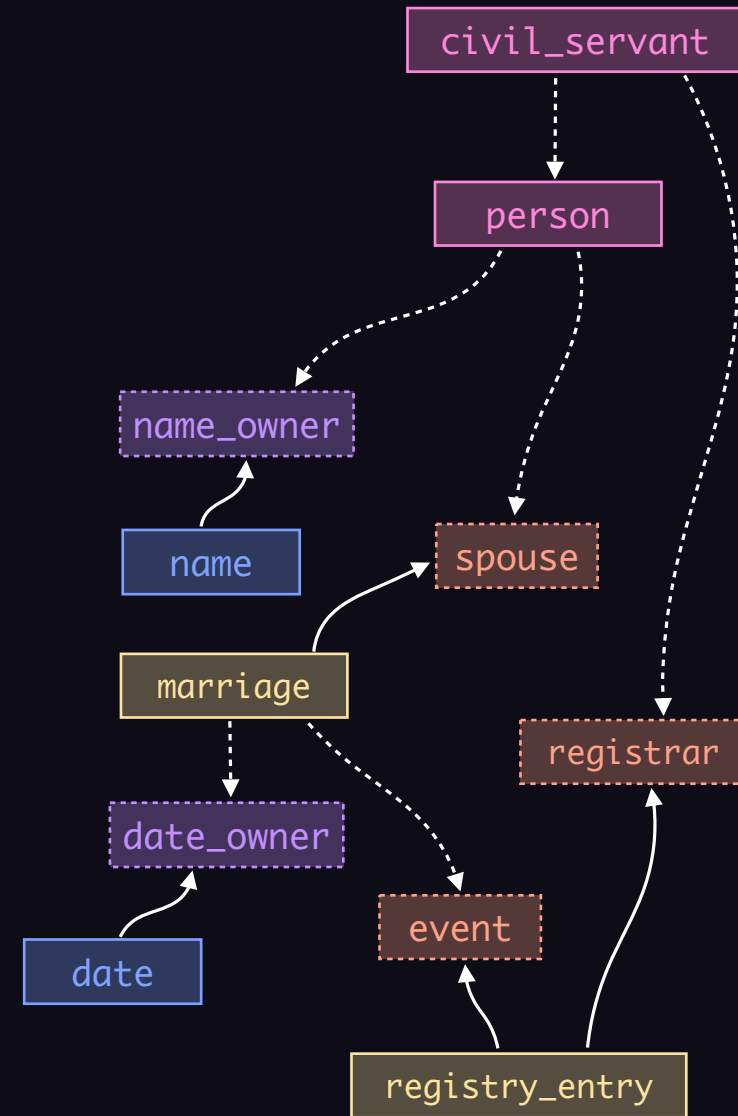
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
    isa marriage,
    has date 17-05-2004;
(event: $m, registrar: $a)
    isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

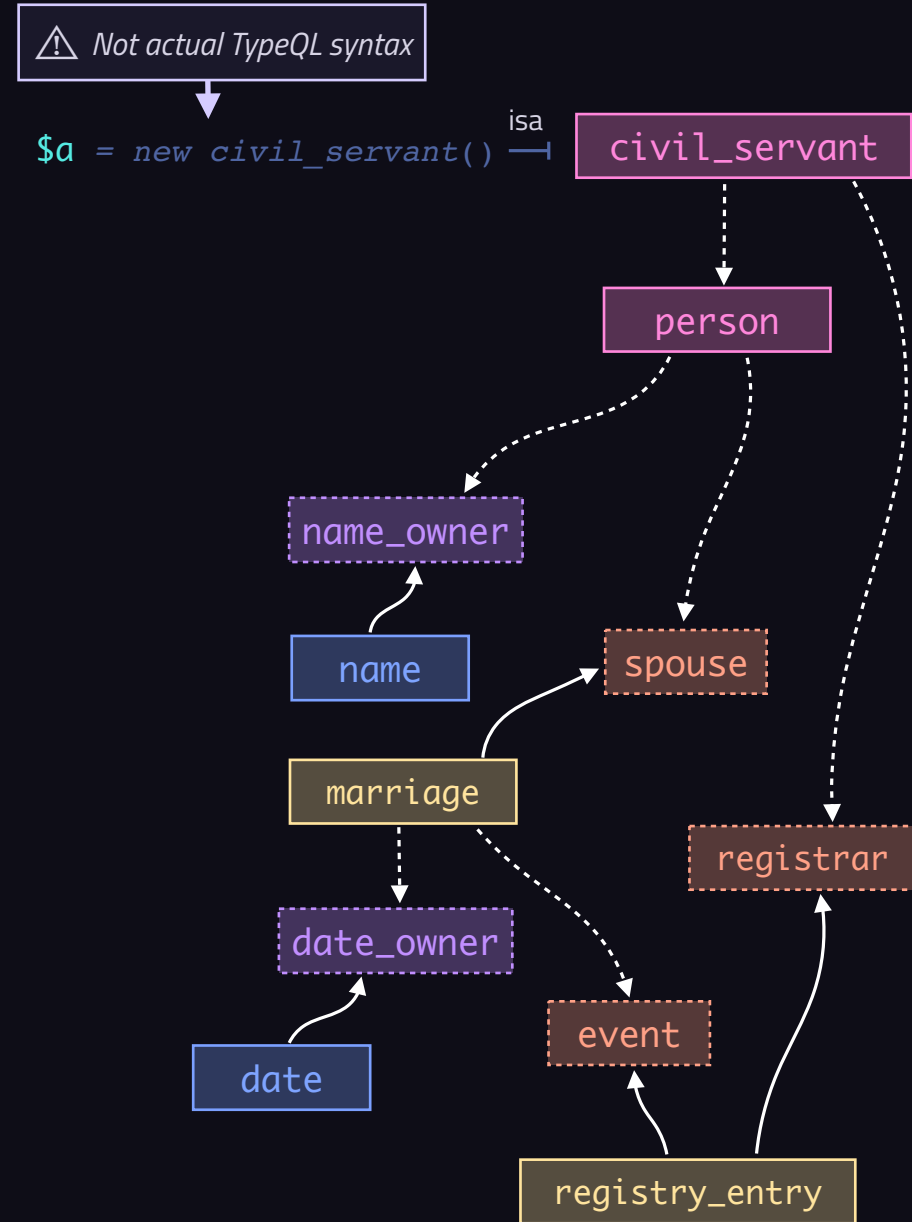
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
    isa marriage,
    has date 17-05-2004;
(event: $m, registrar: $a)
    isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

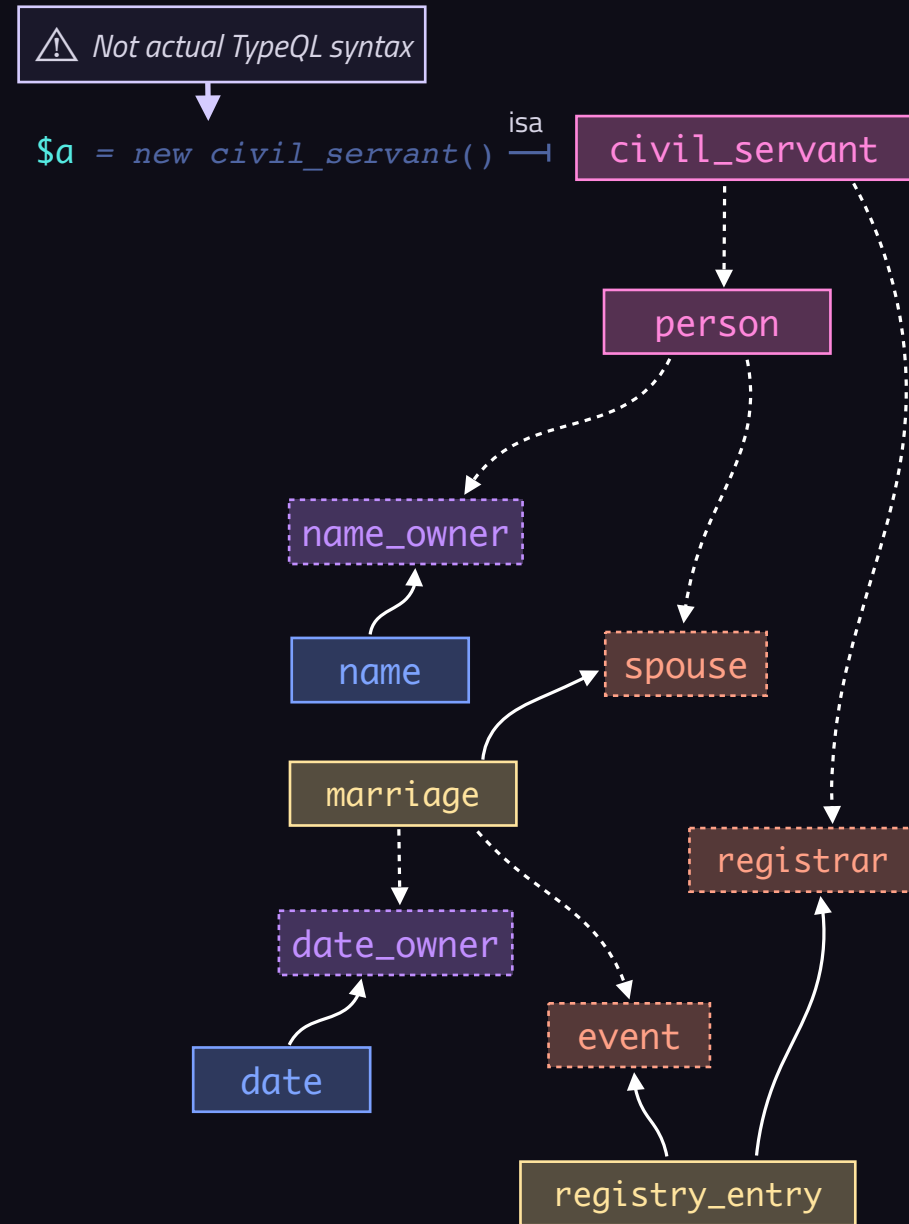
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
    isa marriage,
    has date 17-05-2004;
(event: $m, registrar: $a)
    isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

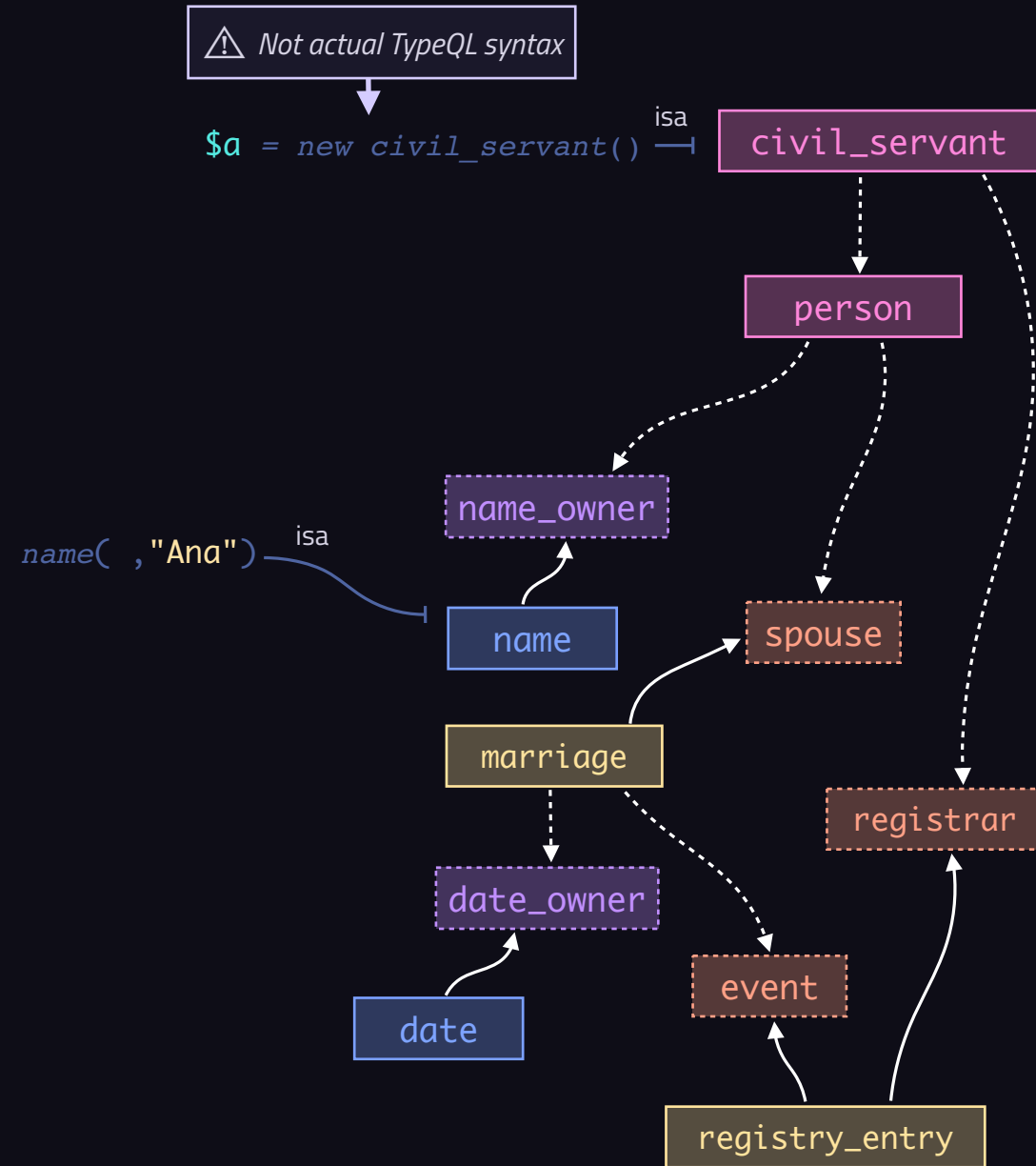
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

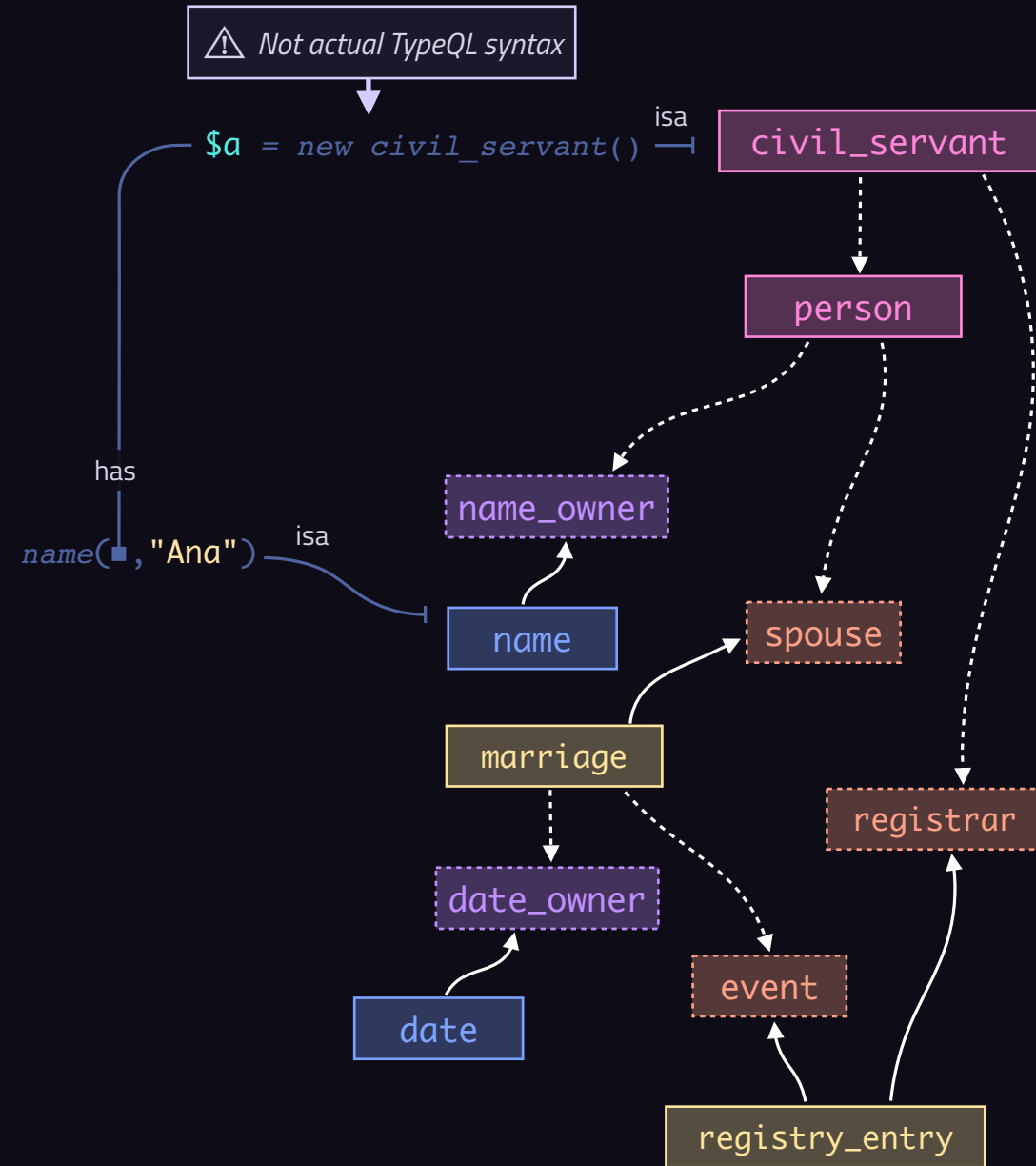
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

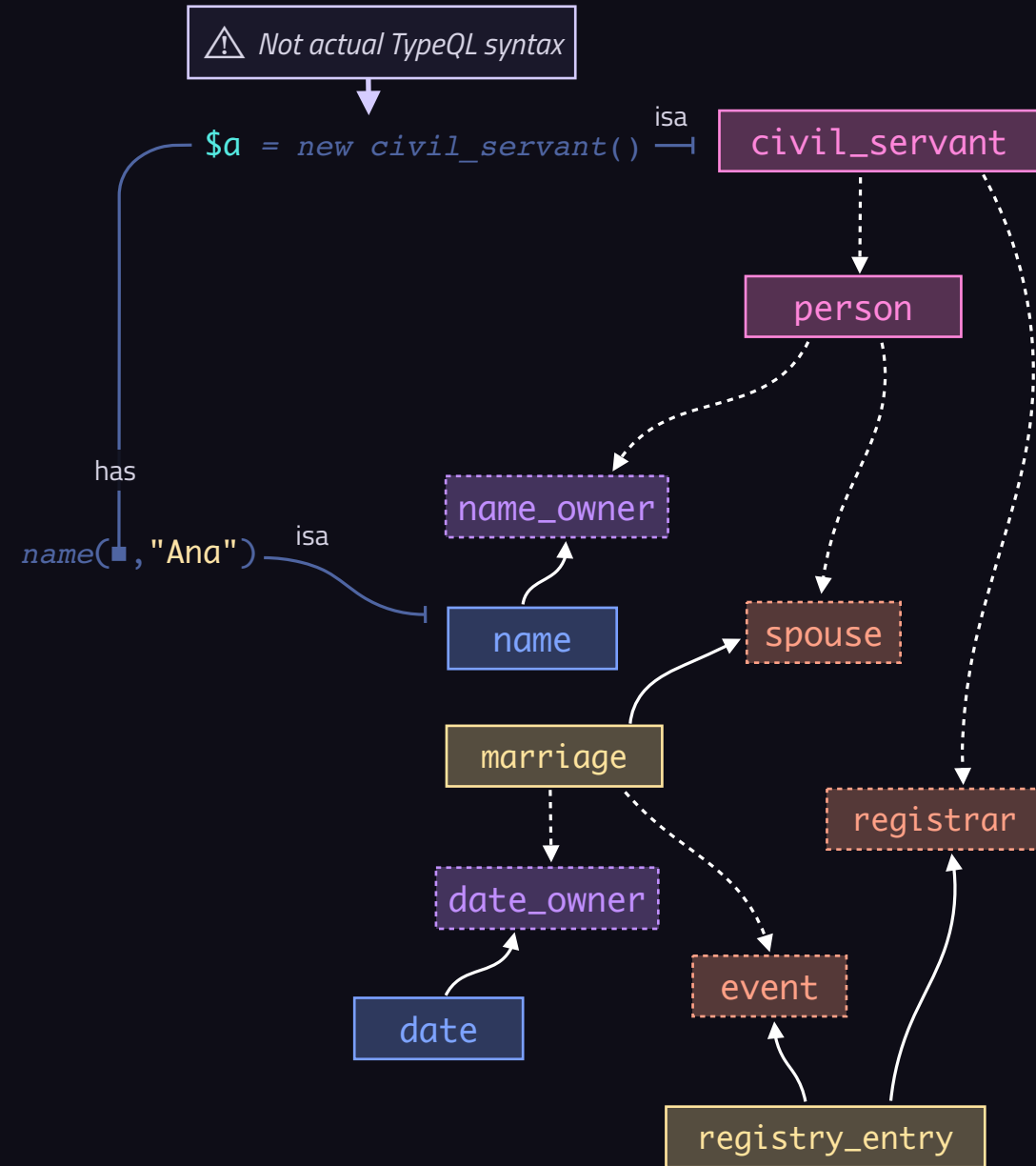
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

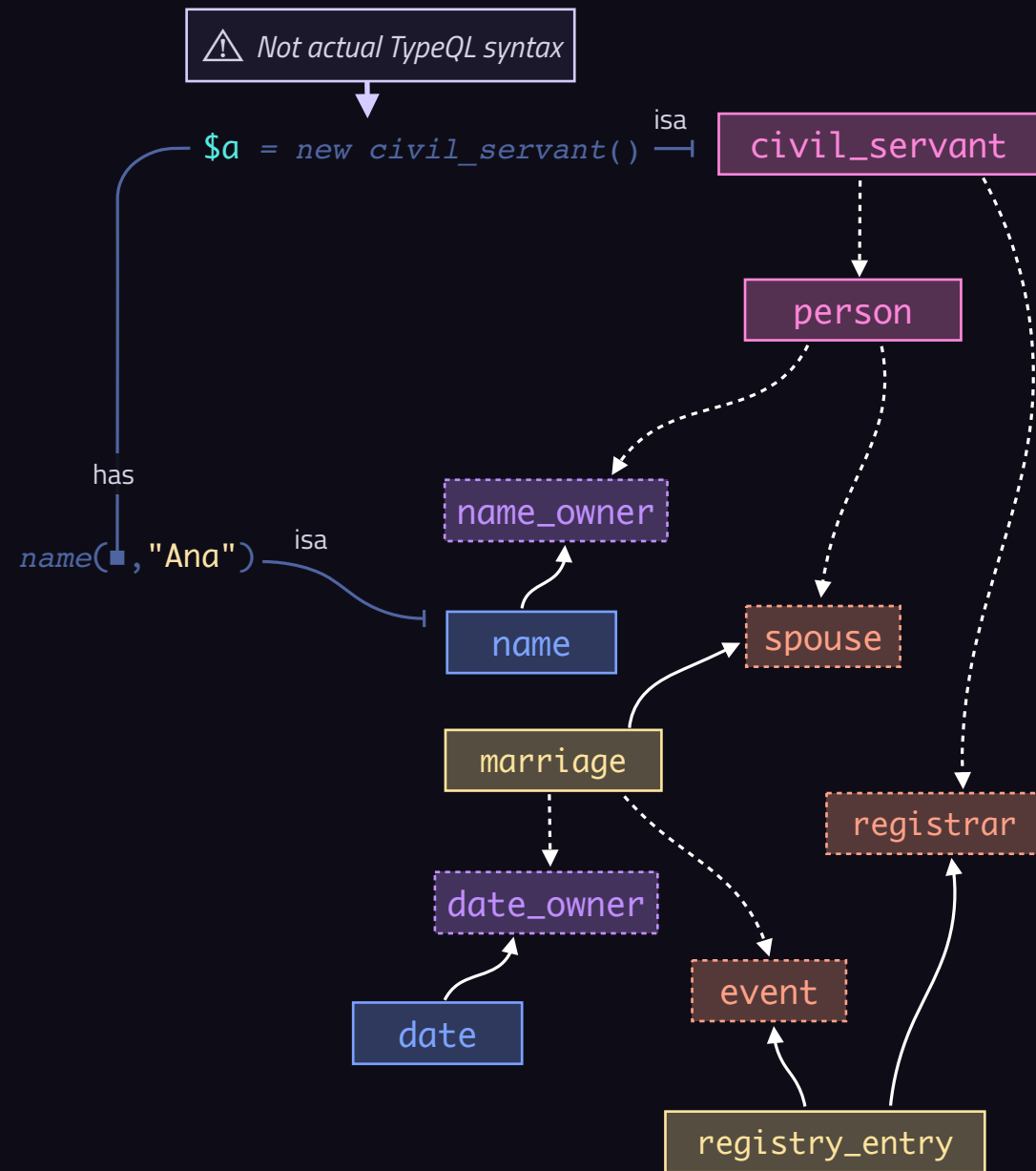
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can `insert` data instances
- Must account for value types and dependencies

```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```

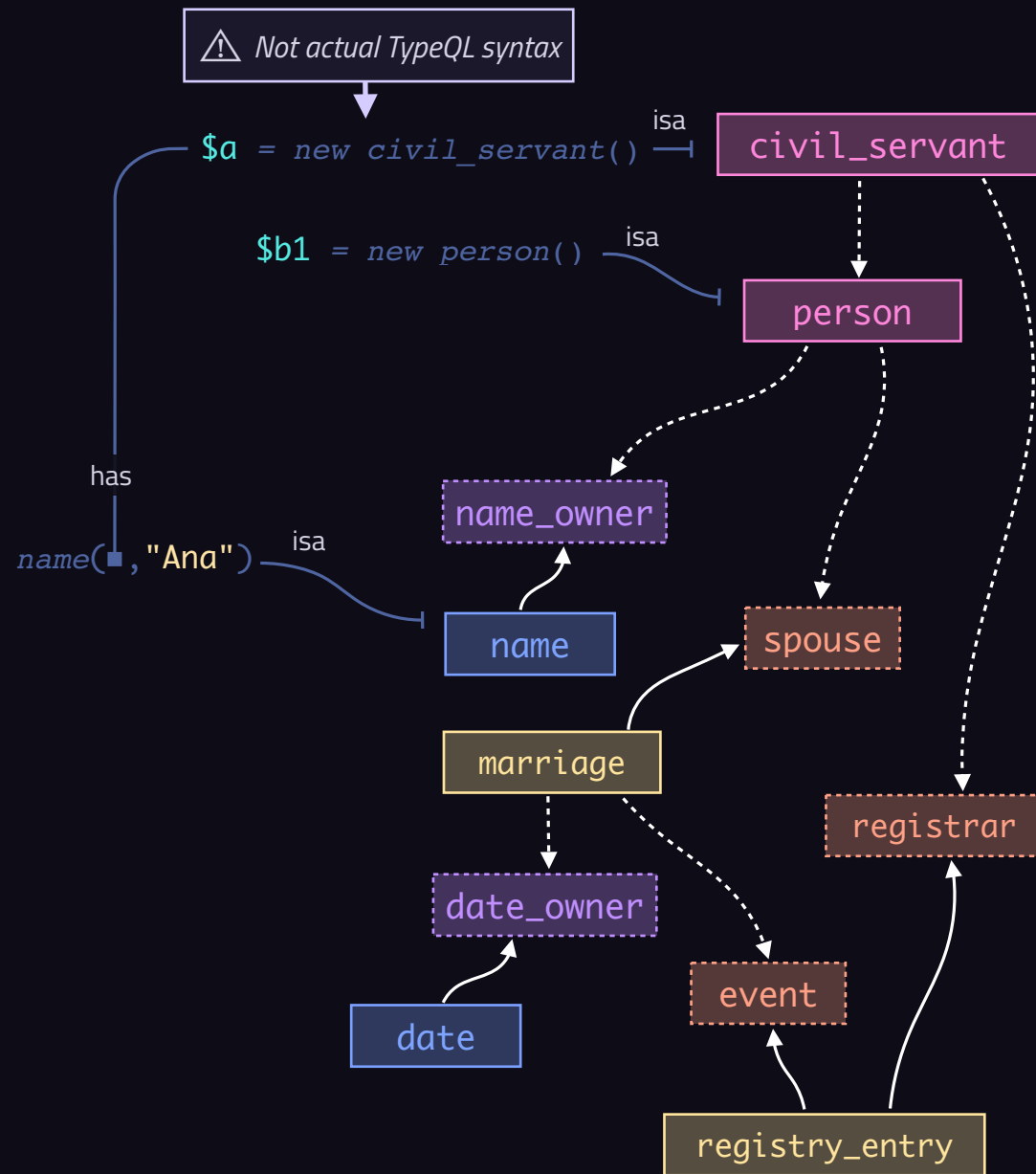




# TypeQL practice: data instances

- For given type schema can insert data instances
- Must account for value types and dependencies

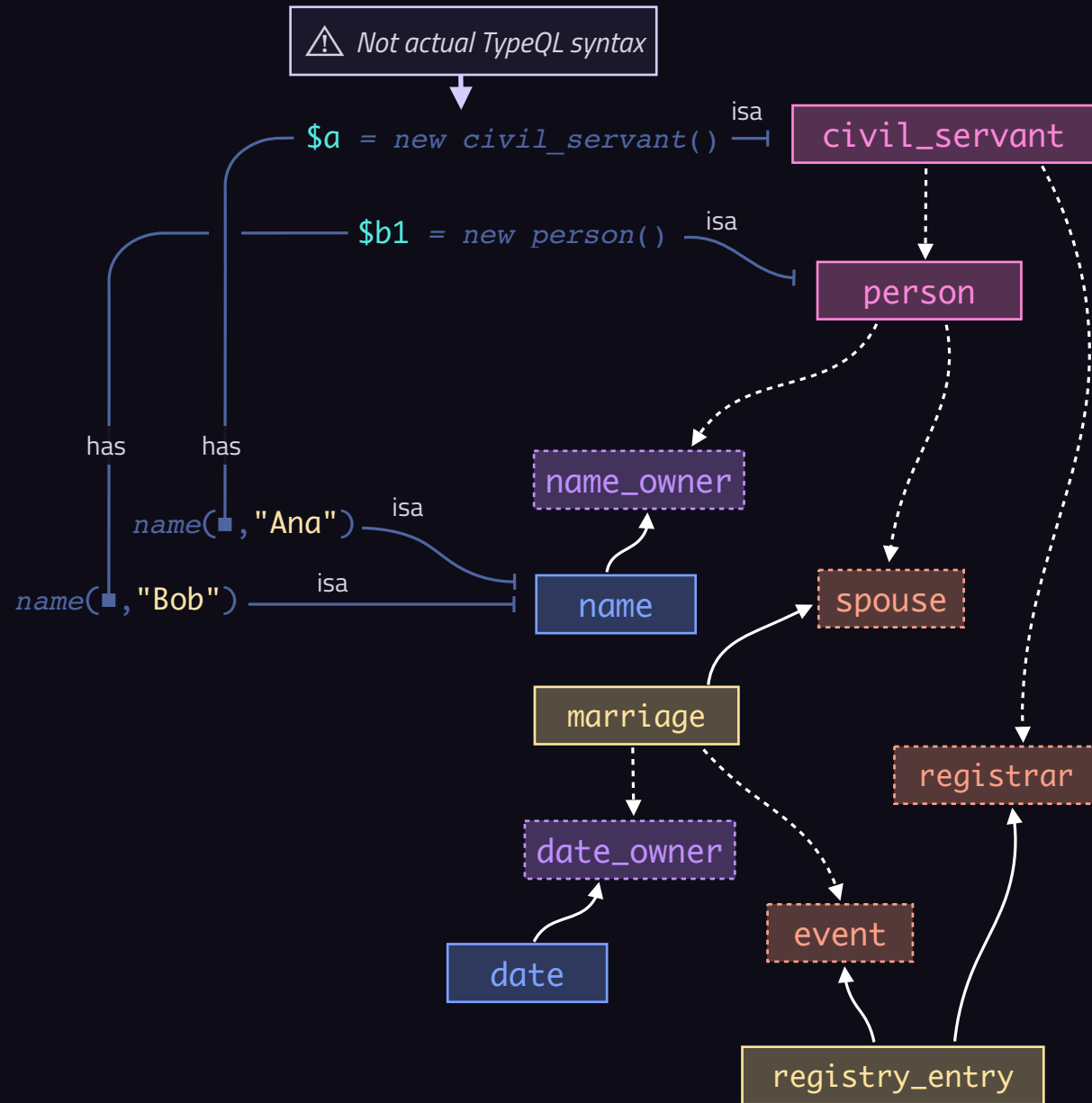
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can *insert* data instances
- Must account for value types and dependencies

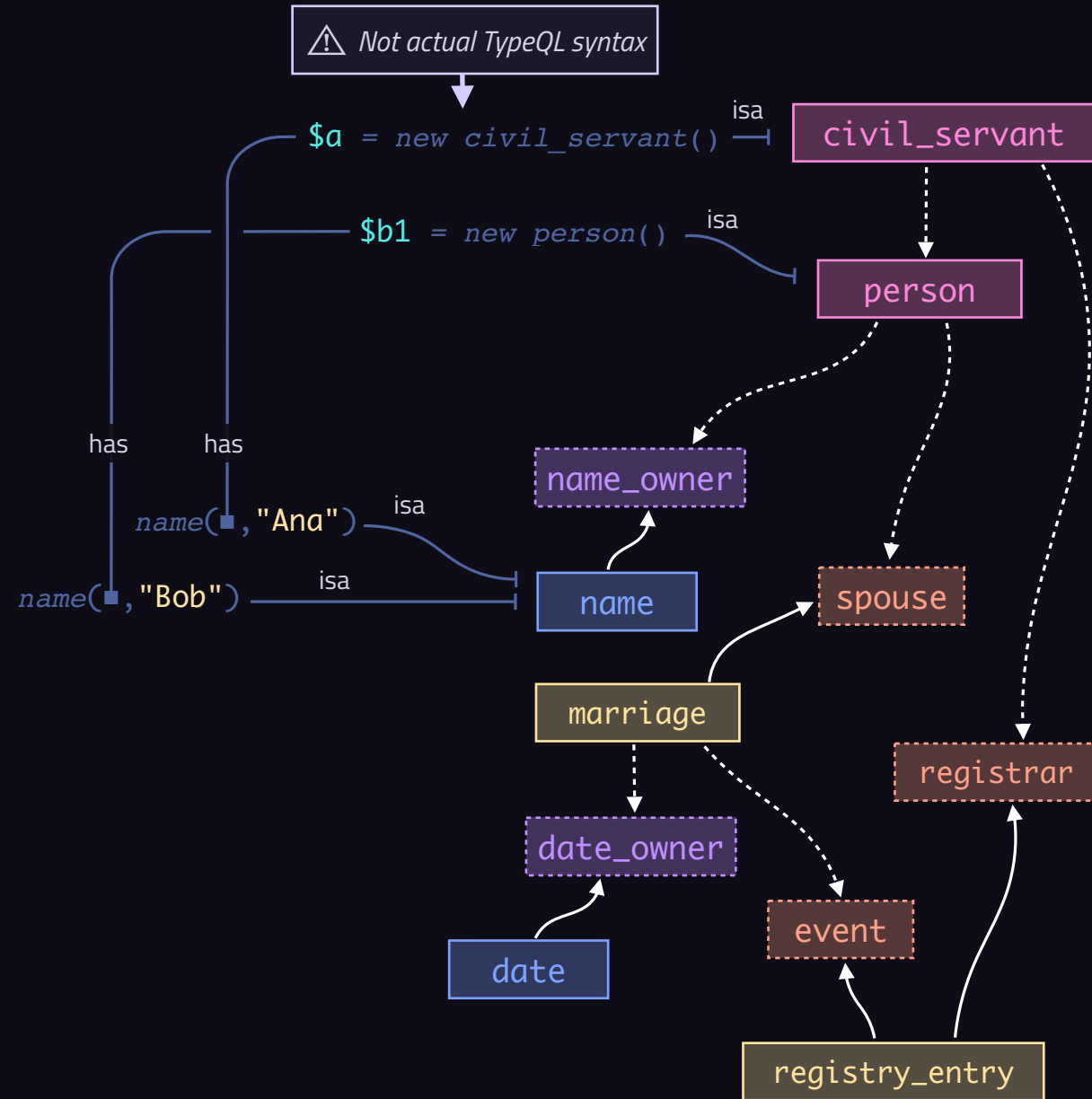
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can insert data instances
- Must account for value types and dependencies

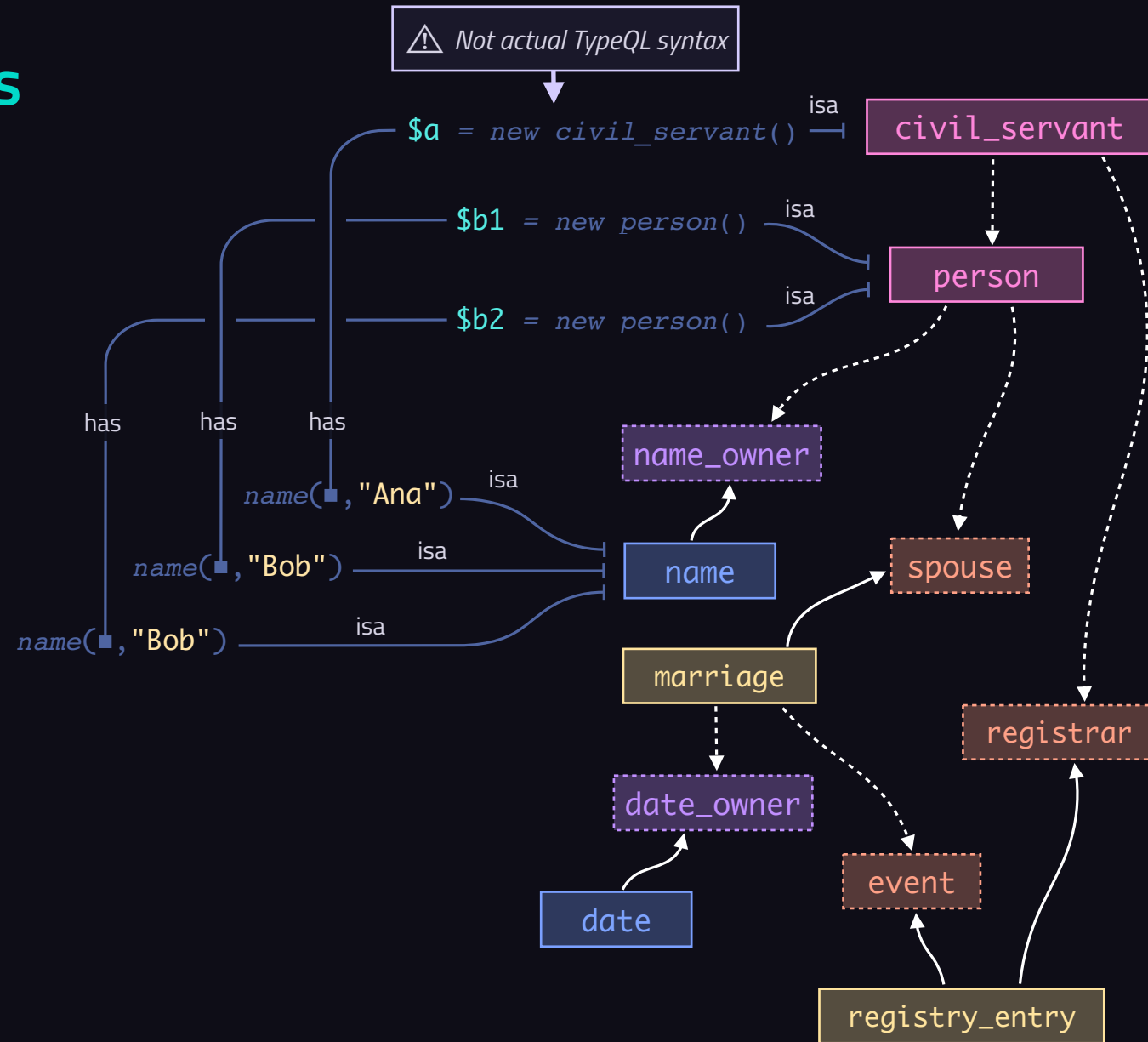
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

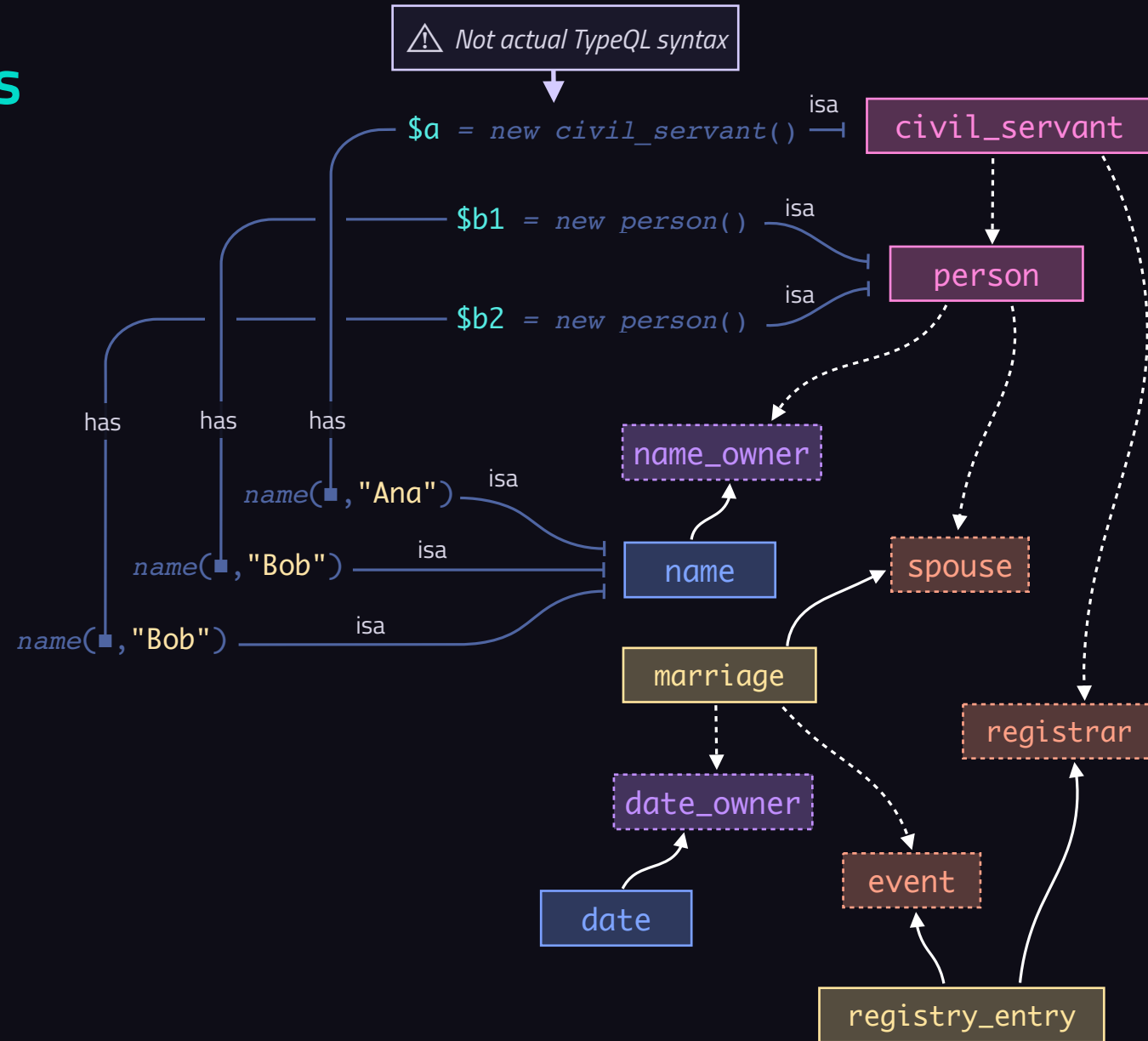
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

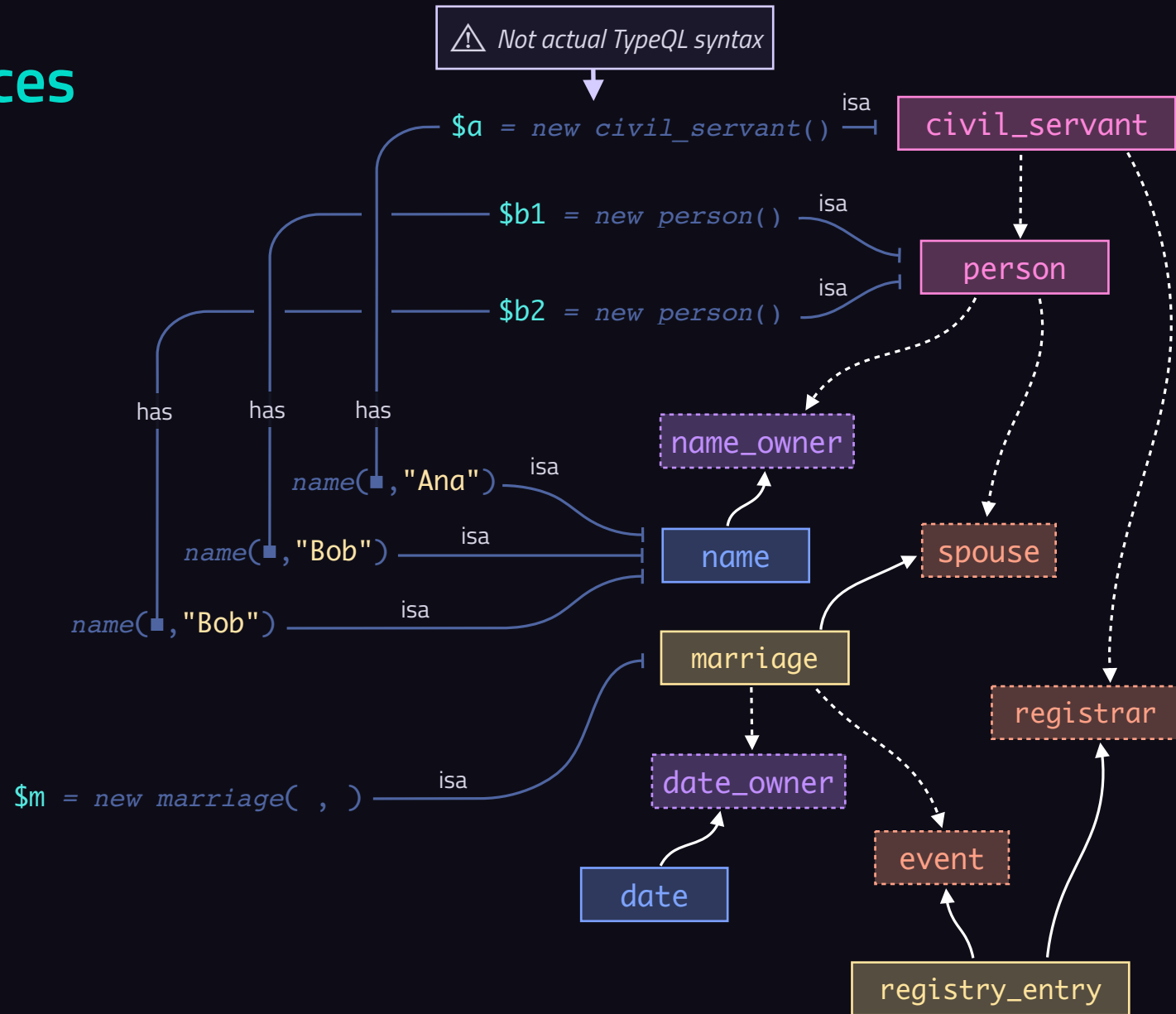
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```

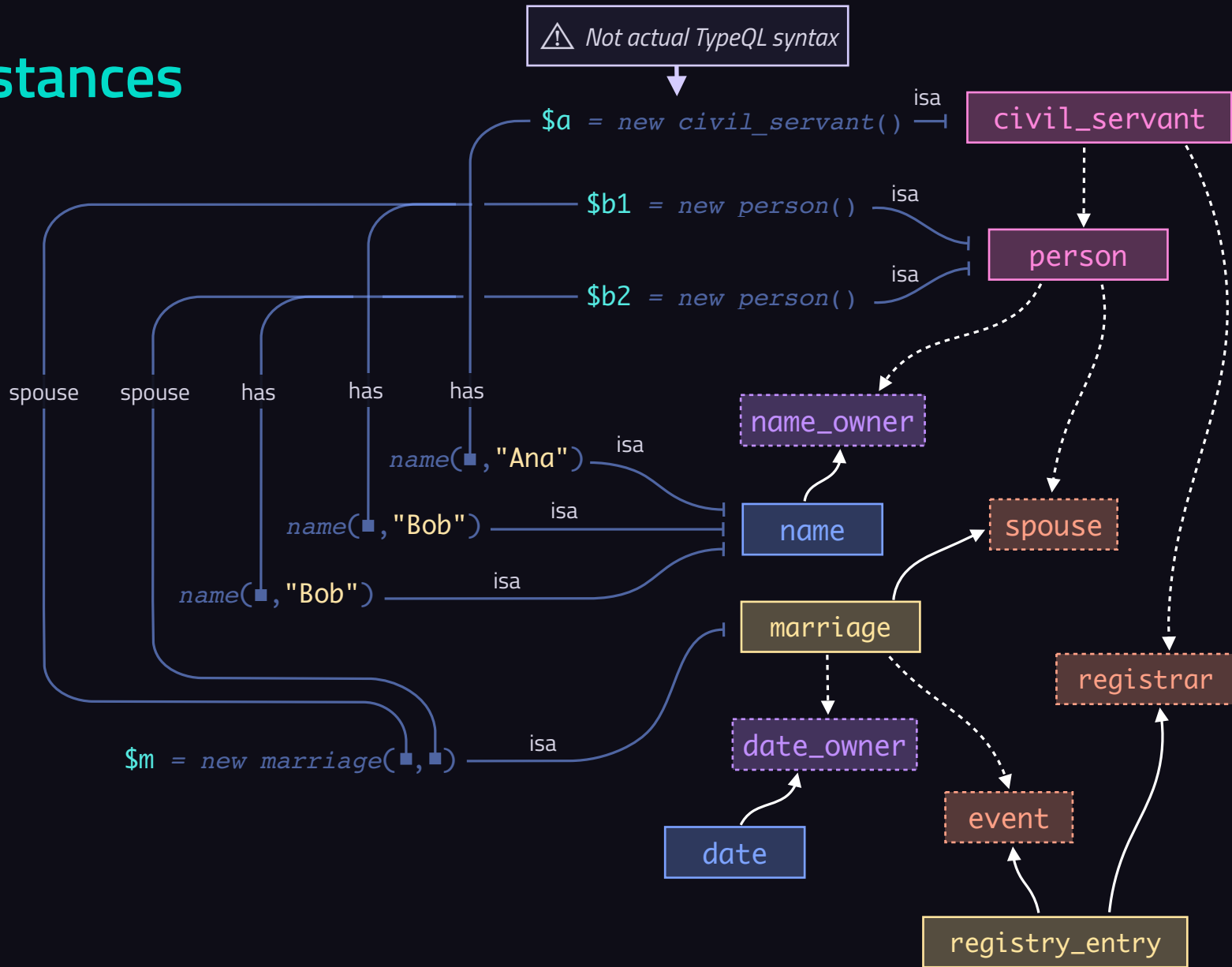


# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

```

insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
  
```



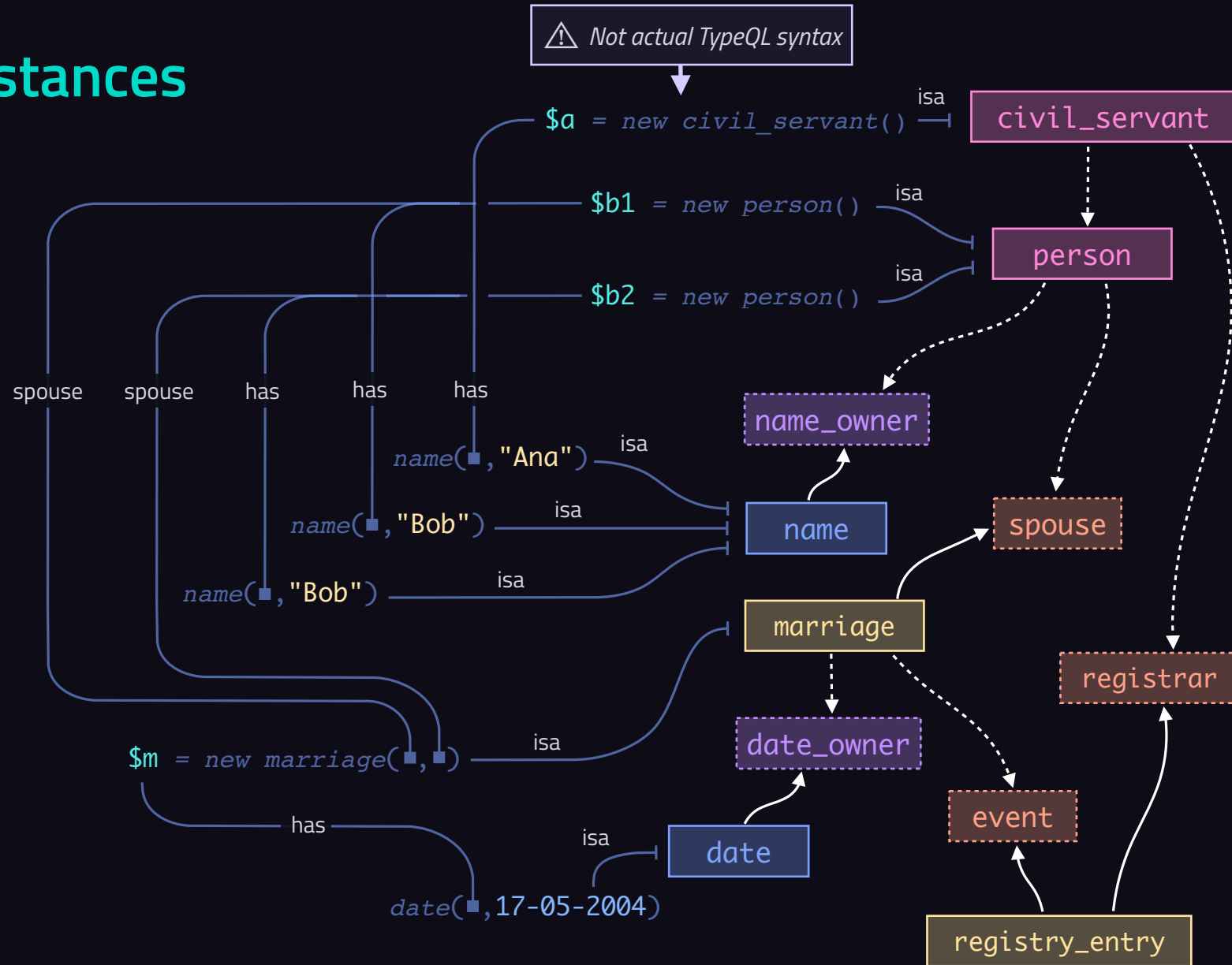




# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

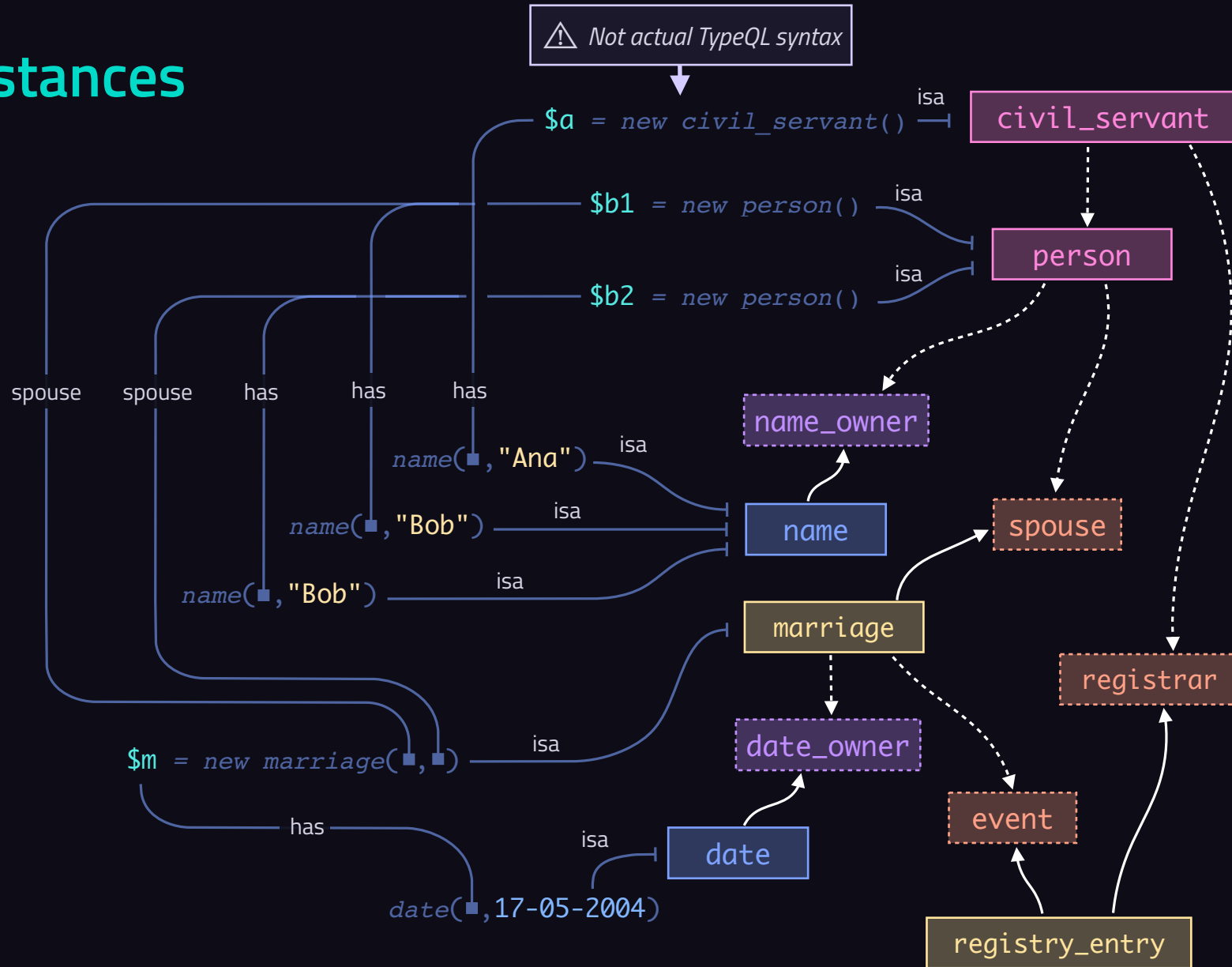
```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```

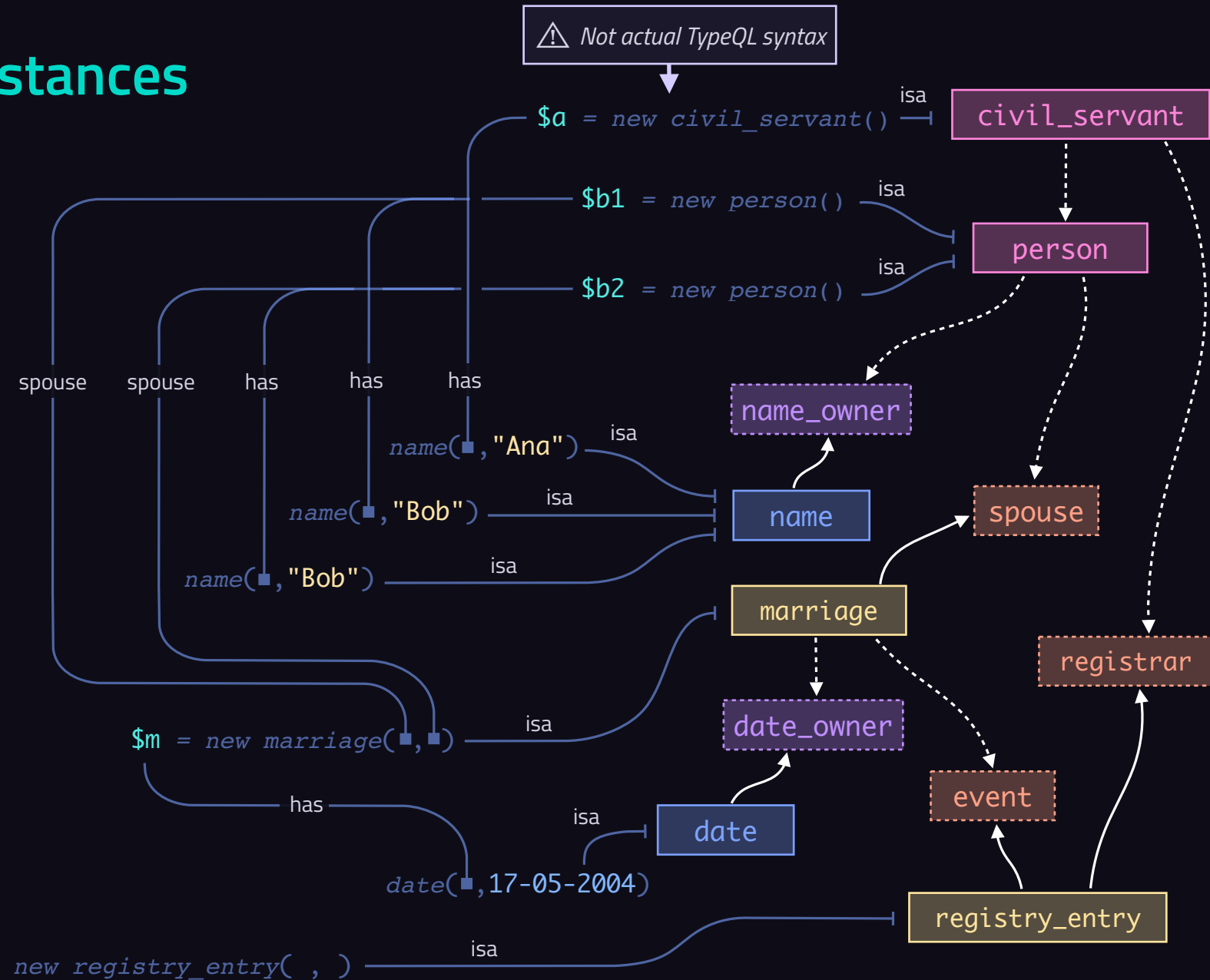


# TypeQL practice: data instances

- For given type schema can *insert* data instances
- Must account for value types and dependencies

```

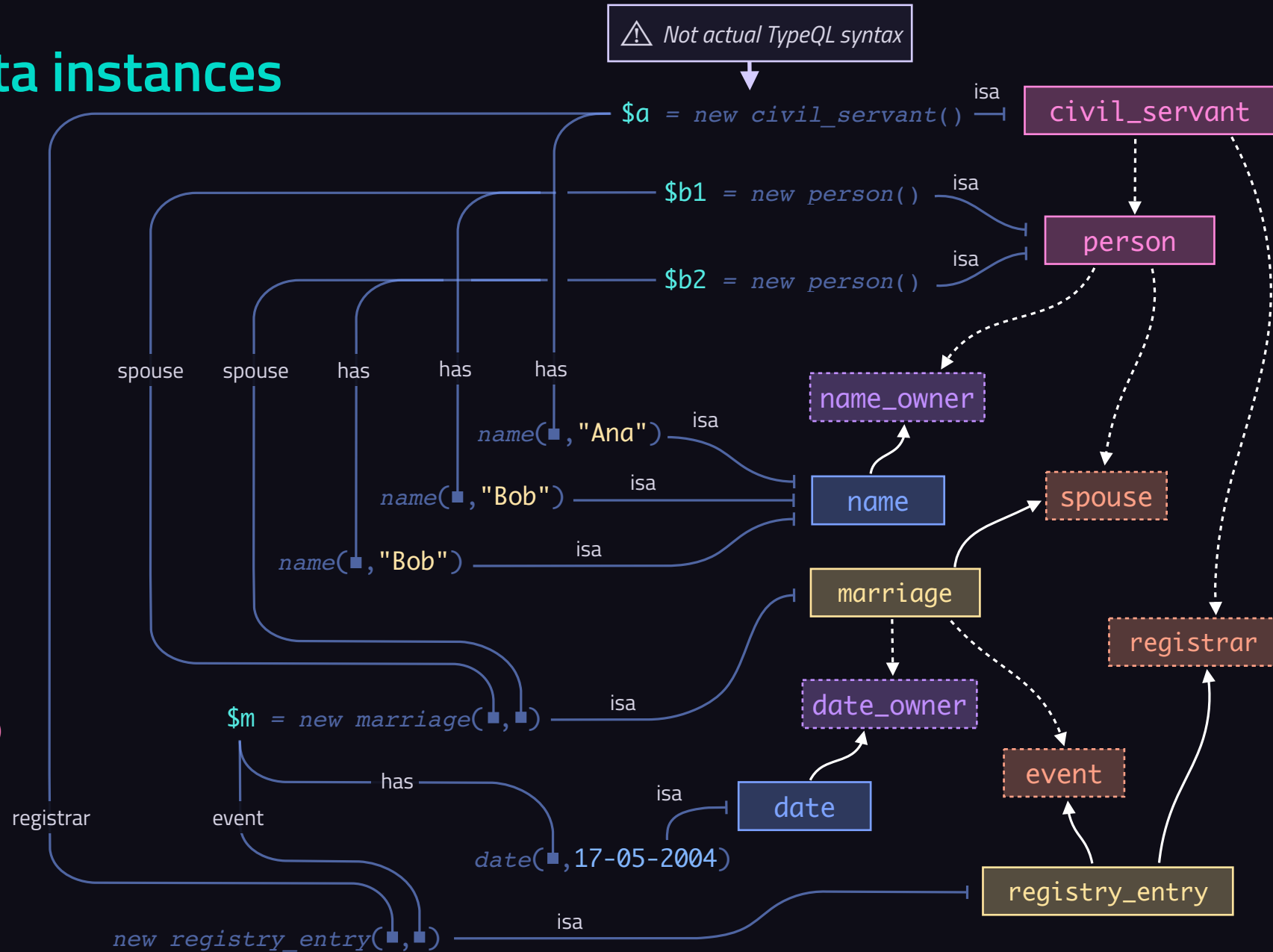
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
    has name "Bob";
$b2 isa person,
    has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
  
```



# TypeQL practice: data instances

- For given type schema can **insert** data instances
- Must account for value types and dependencies

```
insert
$a isa civil_servant;
$a has name "Ana";
$b1 isa person,
  has name "Bob";
$b2 isa person,
  has name "Bob";
$m (spouse: $b1, spouse: $b2)
  isa marriage,
  has date 17-05-2004;
(event: $m, registrar: $a)
  isa registry_entry;
```



# TypeQL practice: data instances—*the details*

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
isa marriage,
```

# TypeQL practice: data instances—*the details*

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
isa marriage,
```

number of spouses is *variadic*

# TypeQL practice: data instances—*the details*

*fully addressed in TypeQL 3.x!*

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
  relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
  isa marriage,
```

number of spouses is *variadic*

# TypeQL practice: data instances—*the details*

fully addressed in TypeQL 3.x!

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
  relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
  isa marriage,
```

number of spouses is *variadic*

## 2. *Globality* of attributes

```
insert  
  "Chloe" isa name;
```



# TypeQL practice: data instances—*the details*

fully addressed in TypeQL 3.x!

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
  relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
  isa marriage,
```

number of spouses is *variadic*

## 2. *Globality* of attributes

```
insert  
  "Chloe" isa name;
```

yields an *unowned attribute*:  
*a.k.a. 'global constant'*

```
name( _ , "Chloe") — isa —> name
```

# TypeQL practice: data instances—*the details*

fully addressed in TypeQL 3.x!

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
  relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
  isa marriage,
```

number of spouses is *variadic*

## 2. *Globality* of attributes

```
insert  
  "Chloe" isa name;
```

yields an *unowned attribute*:  
*a.k.a. 'global constant'*

```
name( _ , "Chloe") — isa —> name
```

## 3. *Intentionality* of castings

*In all instantiations, objects must cast into interfaces without having to pass through sub-interfaces.*



*cannot instantiate with adult*

# TypeQL practice: data instances—*the details*

fully addressed in TypeQL 3.x!

## 1. *Variadicity* of roles

defined

```
marriage sub relation,  
relates spouse;
```

then inserted

```
$m (spouse: $b1, spouse: $b2)  
isa marriage,
```

number of spouses is *variadic*

## 2. *Globality* of attributes

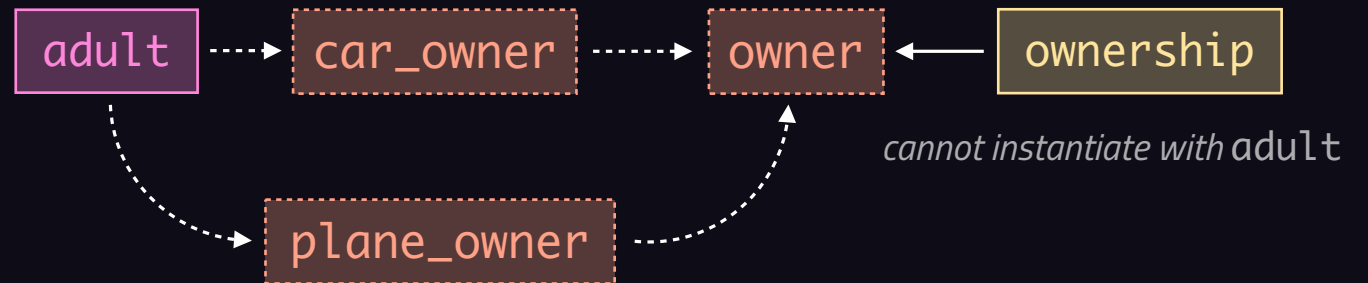
```
insert  
"Chloe" isa name;
```

yields an *unowned attribute*:  
*a.k.a. 'global constant'*

```
name( _ , "Chloe") — isa —> name
```

## 3. *Intentionality* of castings

*In all instantiations, objects must cast into interfaces without having to pass through sub-interfaces.*



# Summary: the PERA model in a nutshell

# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:

	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*

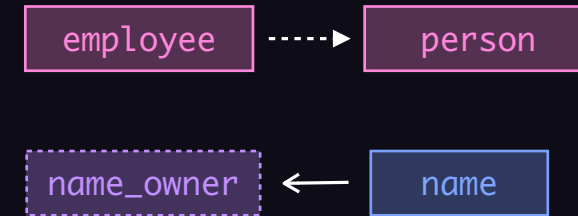
	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types



# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*
- All dependencies are abstracted by *interface types*

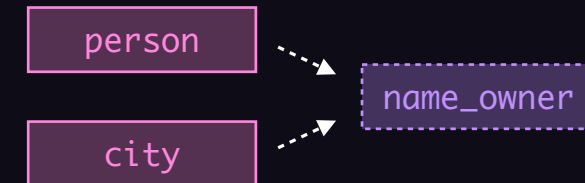
	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types



# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*
- All dependencies are abstracted by *interface types*
- *Object types can implement* these interfaces

	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types

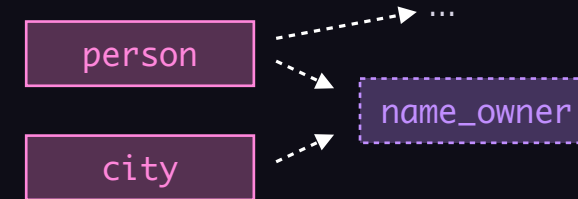




# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*
- All dependencies are abstracted by *interface types*
- *Object types can implement* these interfaces
- *Single-inheritance* but *multi-capability* via interfaces

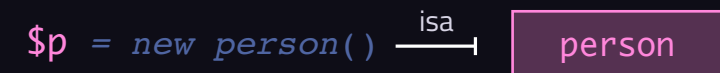
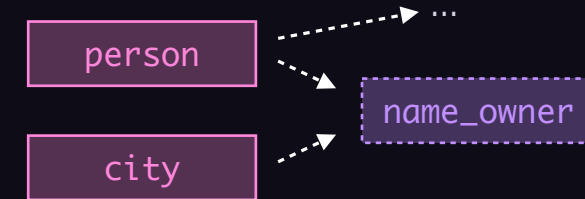
	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types



# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*
- All dependencies are abstracted by *interface types*
- *Object types* can implement these interfaces
- *Single-inheritance* but *multi-capability* via interfaces
- *Objects* in object types can be freely created

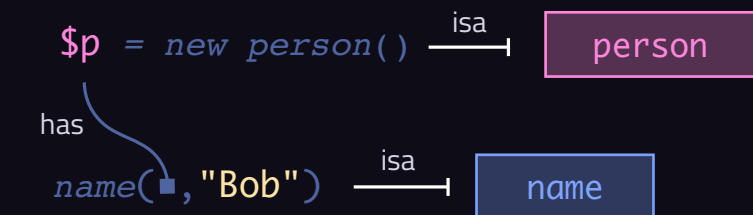
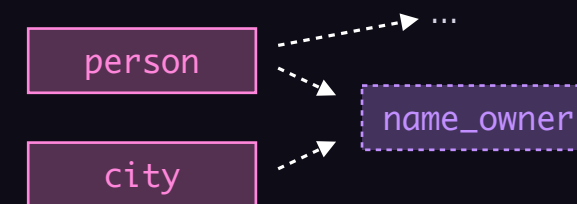
	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types



# Summary: the PERA model in a nutshell

- The PERA type system as *three kinds* of types:
- Type kinds are organized by *inheritance hierarchies*
- All dependencies are abstracted by *interface types*
- *Object types* can implement these interfaces
- *Single-inheritance* but *multi-capability* via interfaces
- *Objects* in object types can be freely created
- *Values* can be (idempotently) added to attribute types

	independent type	dependent type
object types	entity types	relation types
attribute types	(global constants)	attribute types





# Part III:

## Comparison to other data models

...and how PERA brings order to modeling

# The **bottomline**

## The **bottomline**

The PERA model is a *simple and principled* model, built on basic and fundamental ideas found across *all* data models. From the PERA perspective, capturing other data models is *easy*.

# The relational model

*Types:*

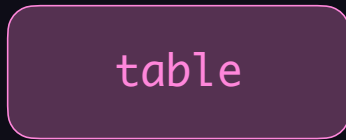
*Instances:*

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"

# The relational model

*Types:*



*Instances:*

row

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"



# The relational model

Types:

table

FK column

Value column

Instances:

row

reference

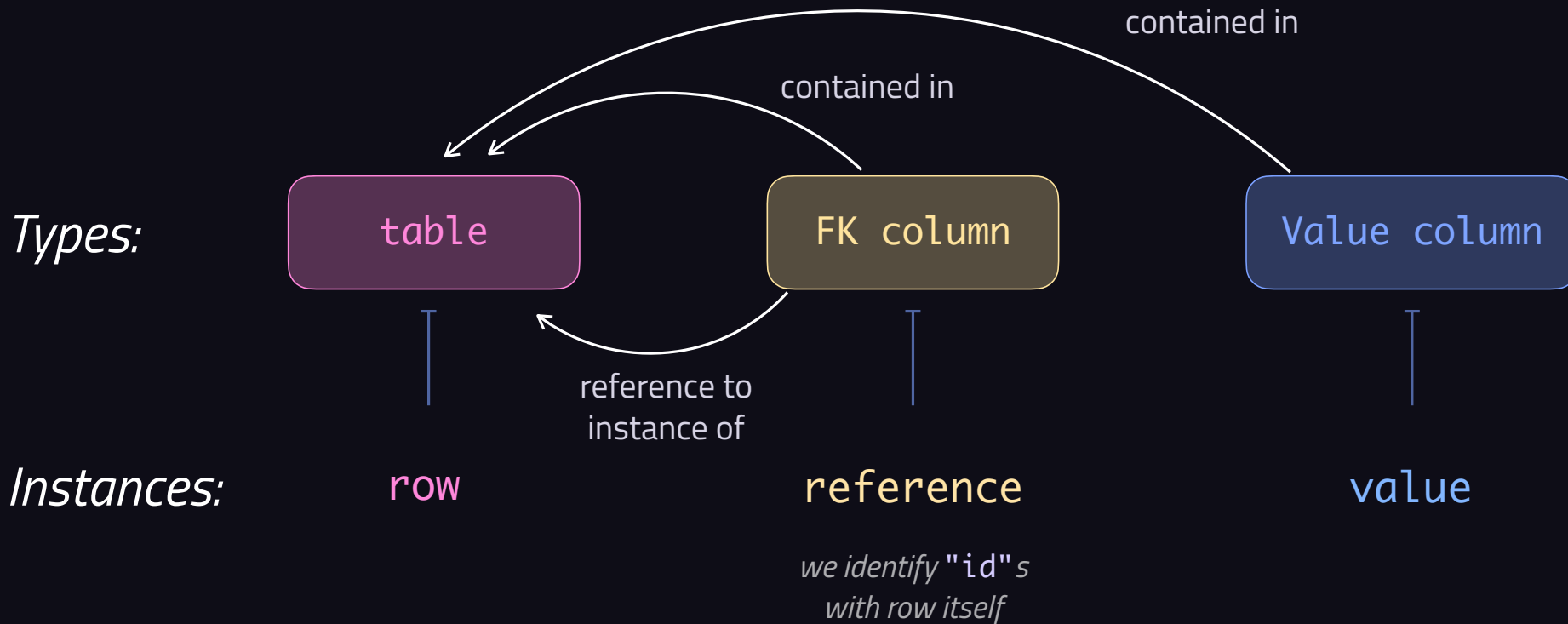
value

*we identify "id"s  
with row itself*

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"

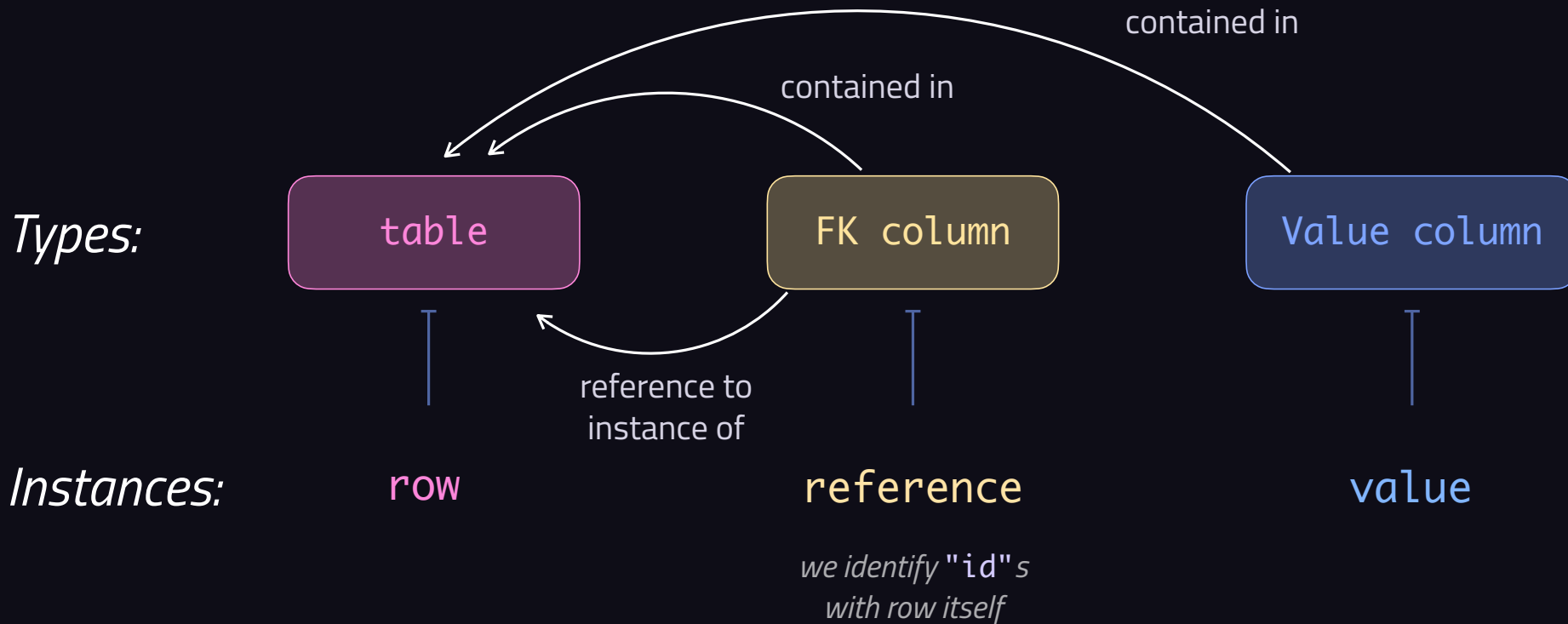
# The relational model



employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"

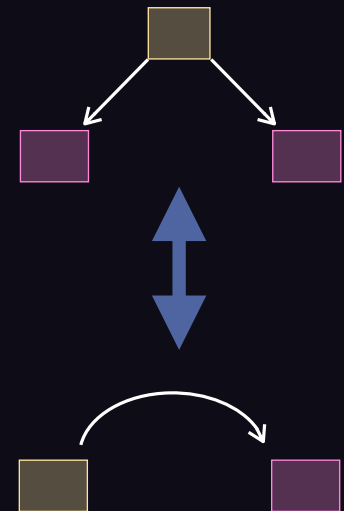
# The relational model



employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"

*Remark.*  
Constraints can modify dependencies!  
e.g. NOT NULL FK:



# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

PERA analog

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...

# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

PERA analog

name

member\_of

team\_name

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...

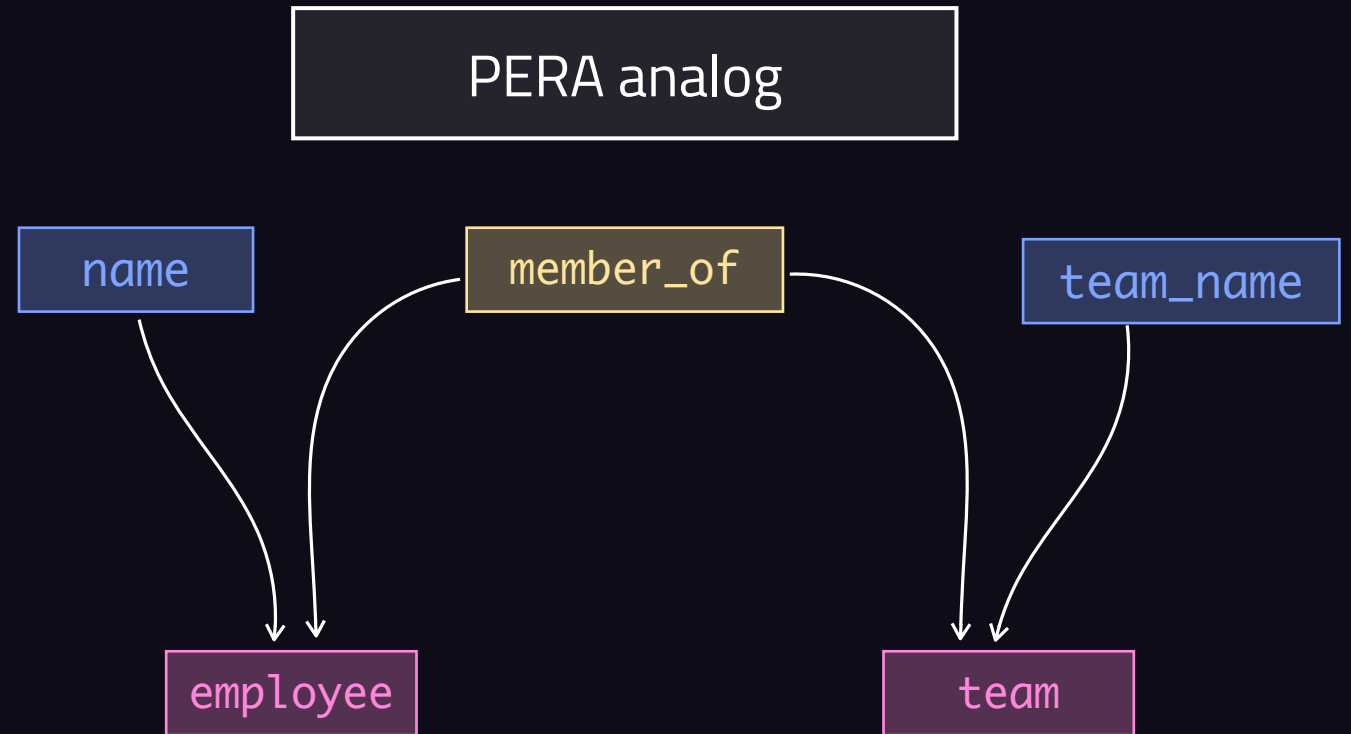
employee

team

# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

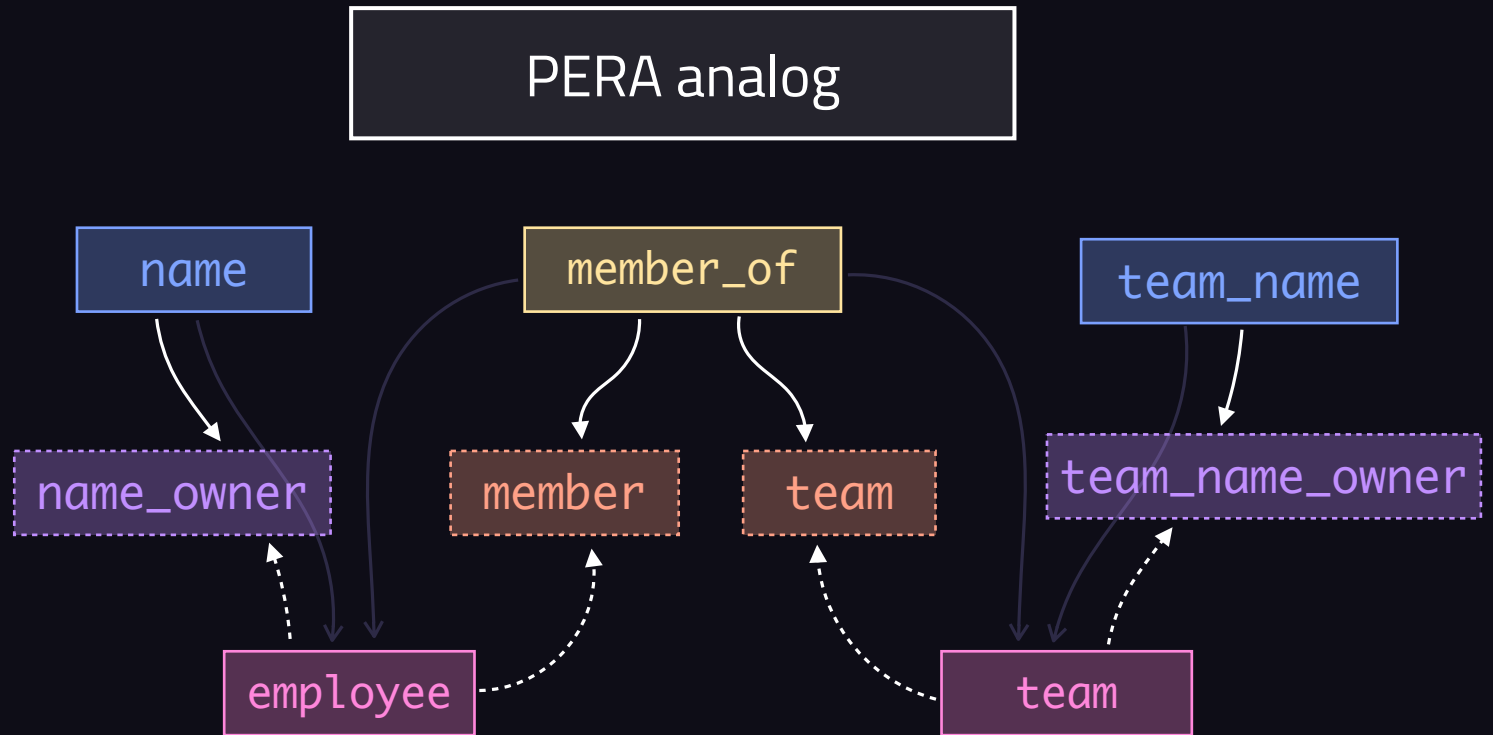
team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...



# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

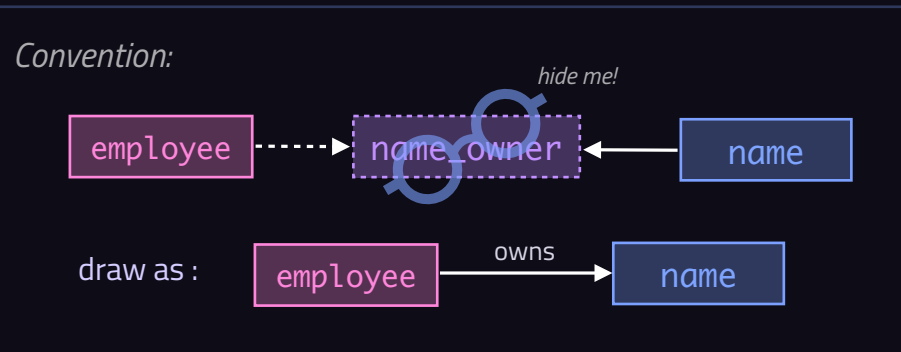
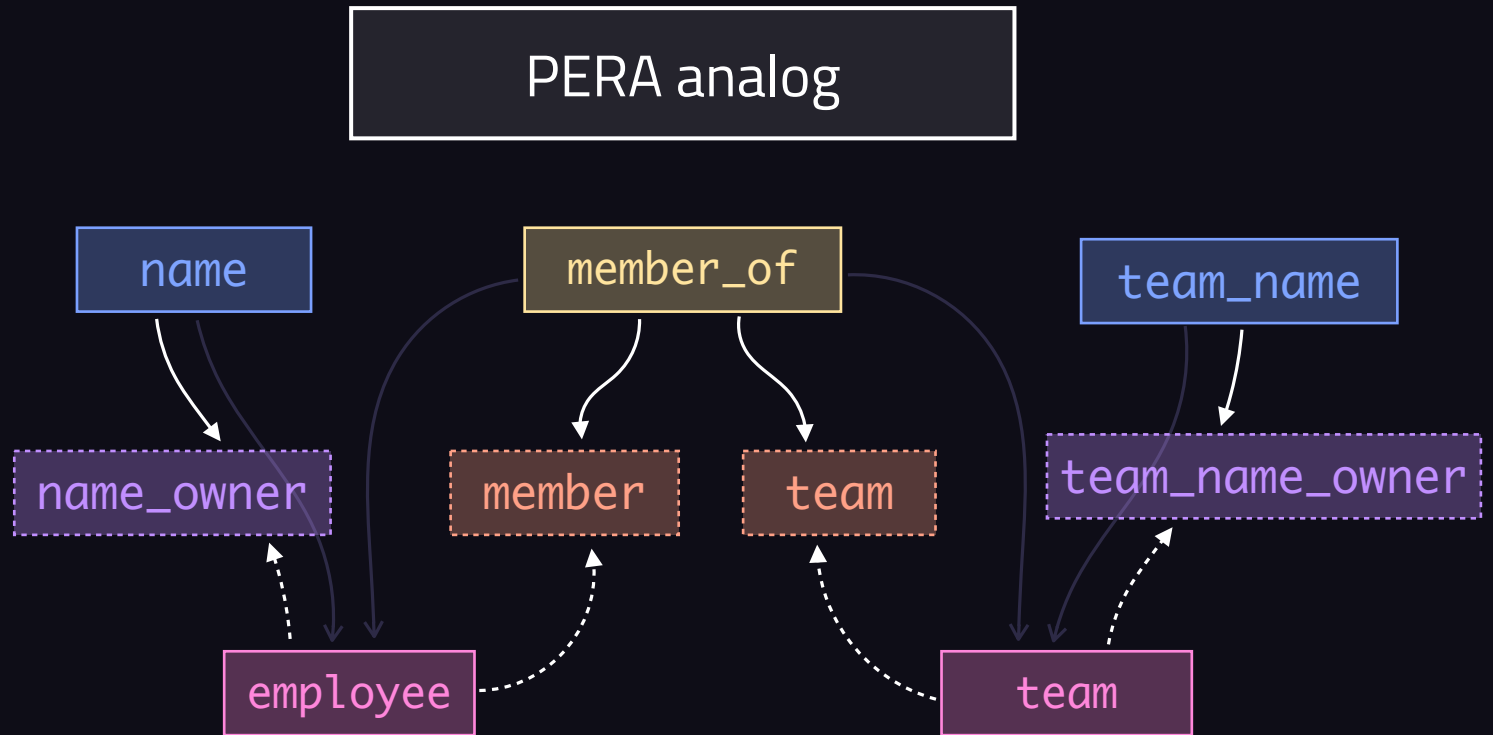
team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...



# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...

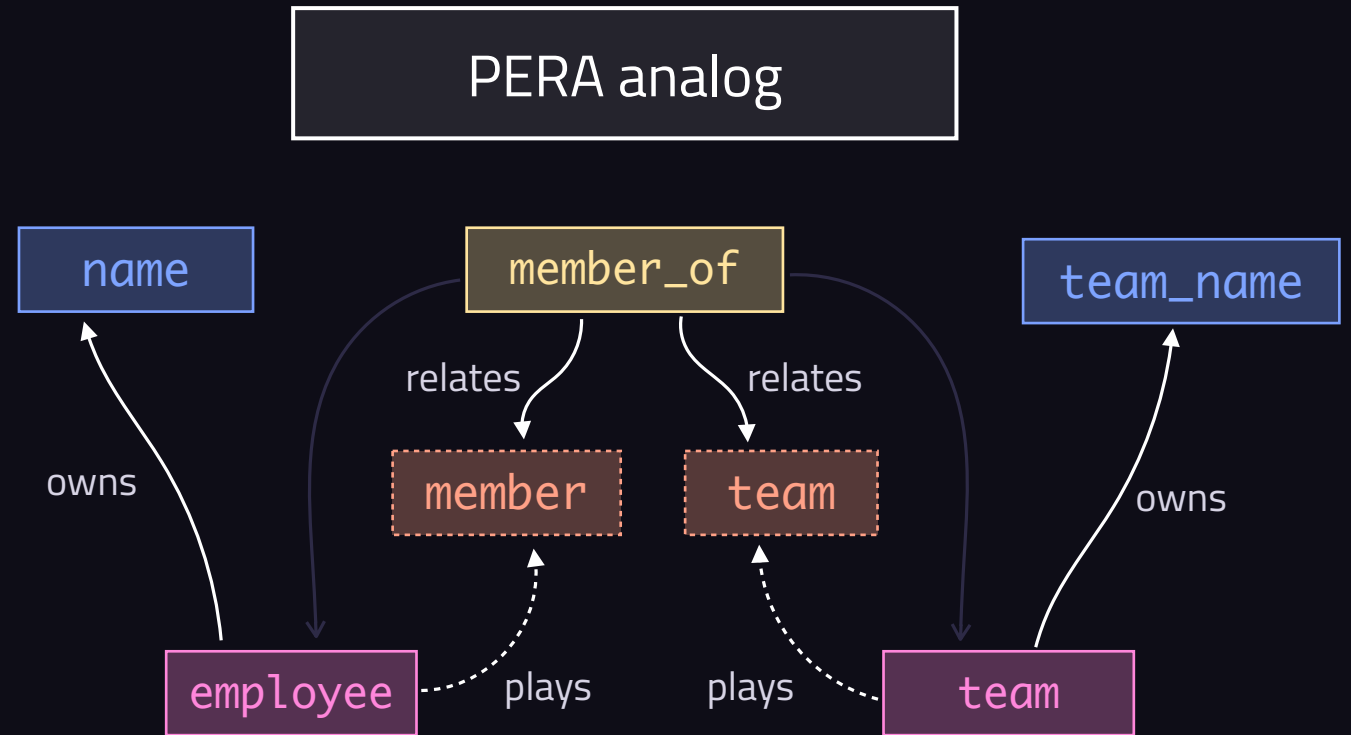




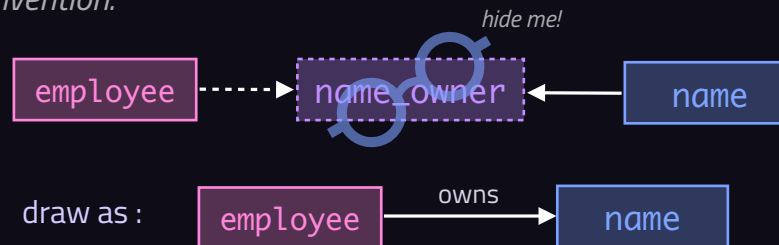
# The relational model: a simple example

employee	
name	member_of
"Ana"	1
"Bob"	1
"Chloe"	3
...	...

team	
id	team_name
1	"Engineering"
2	"Marketing"
3	"Research"
...	...



Convention:



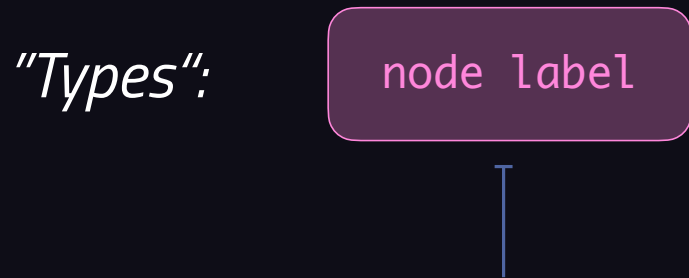
# The graph model

*"Types":*

*Instances:*



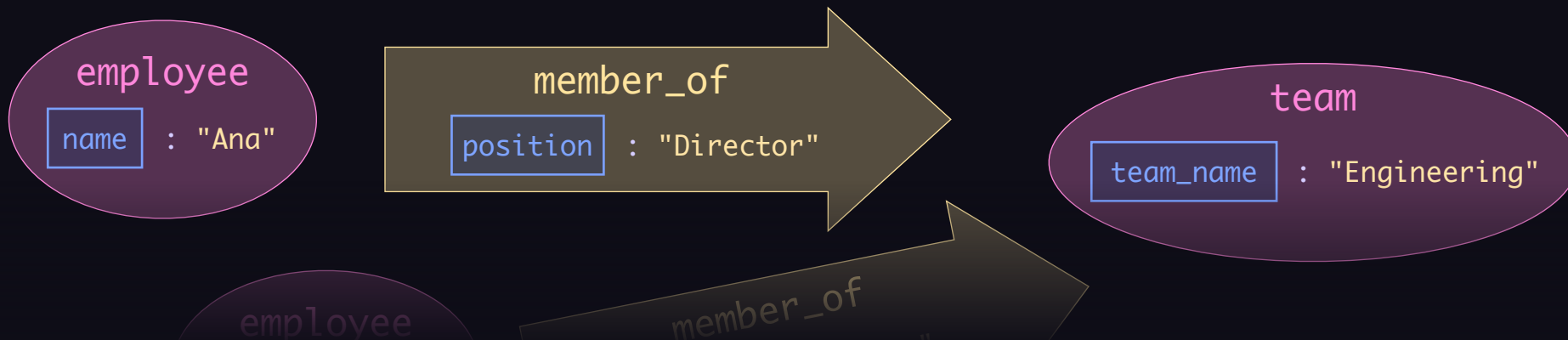
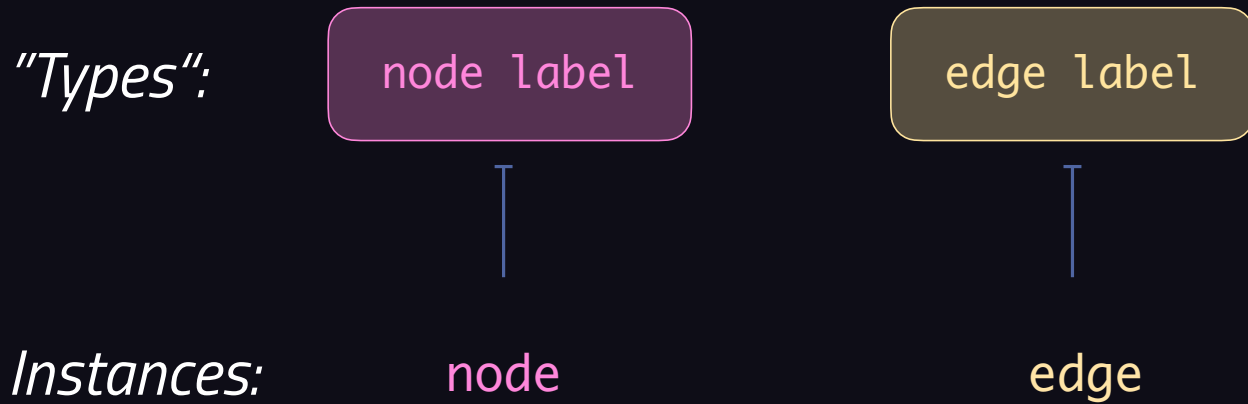
# The graph model



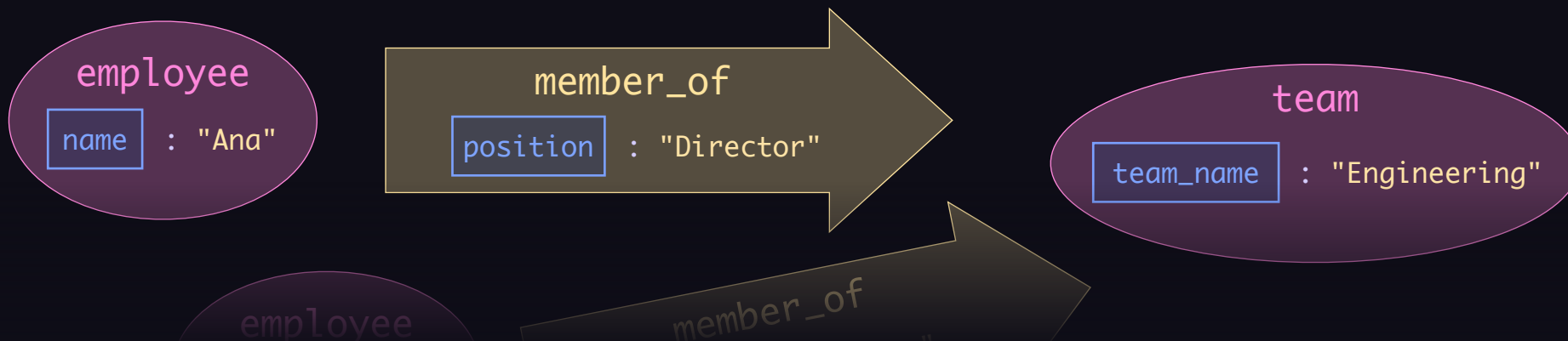
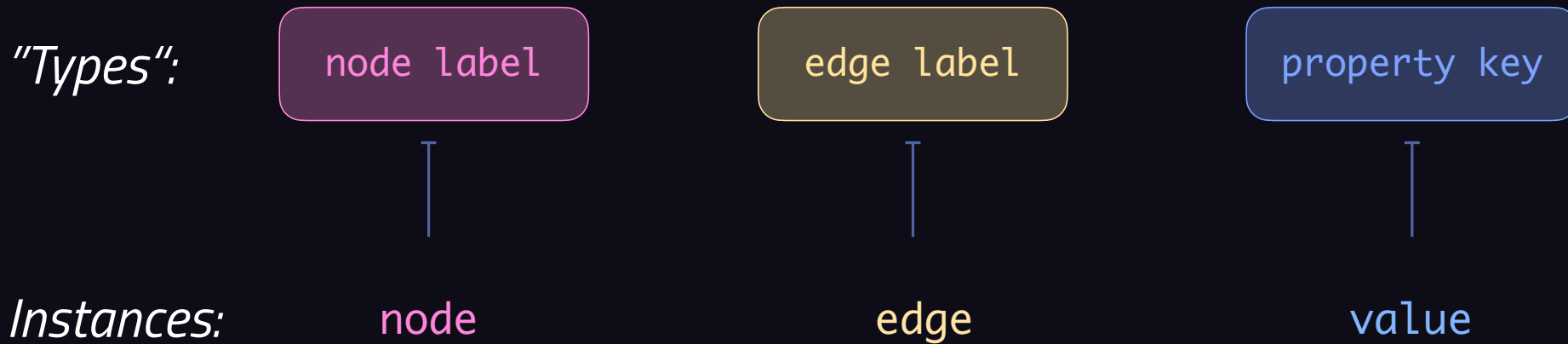
*Instances:* node



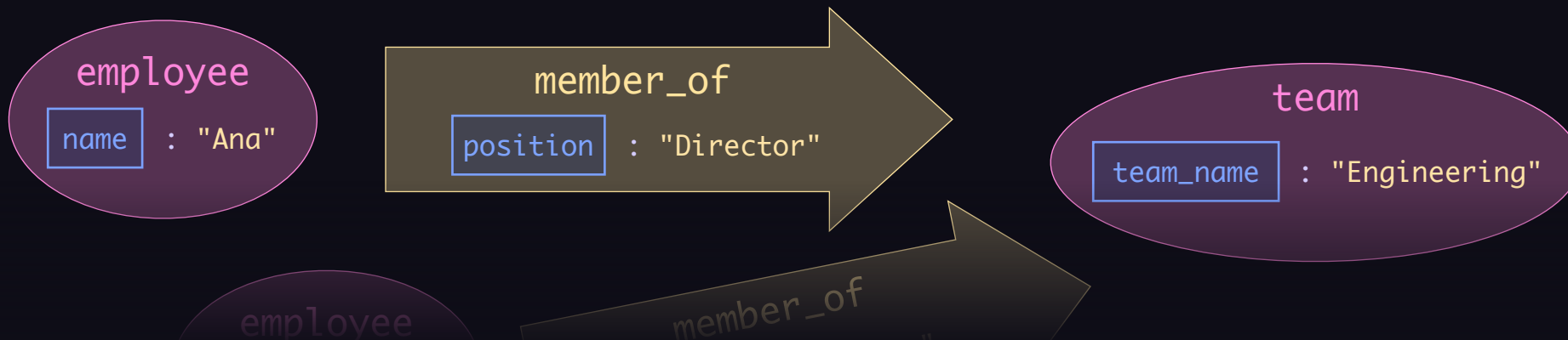
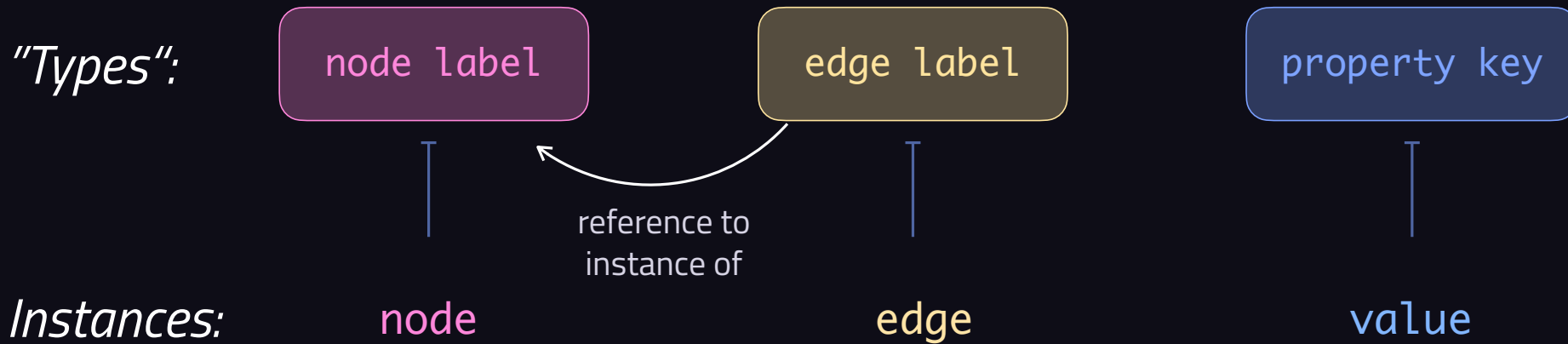
# The graph model



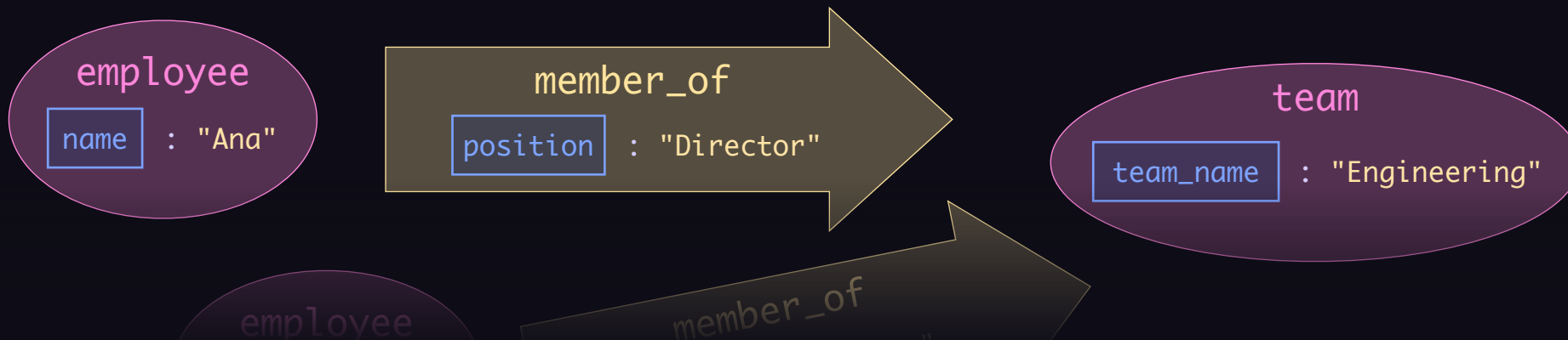
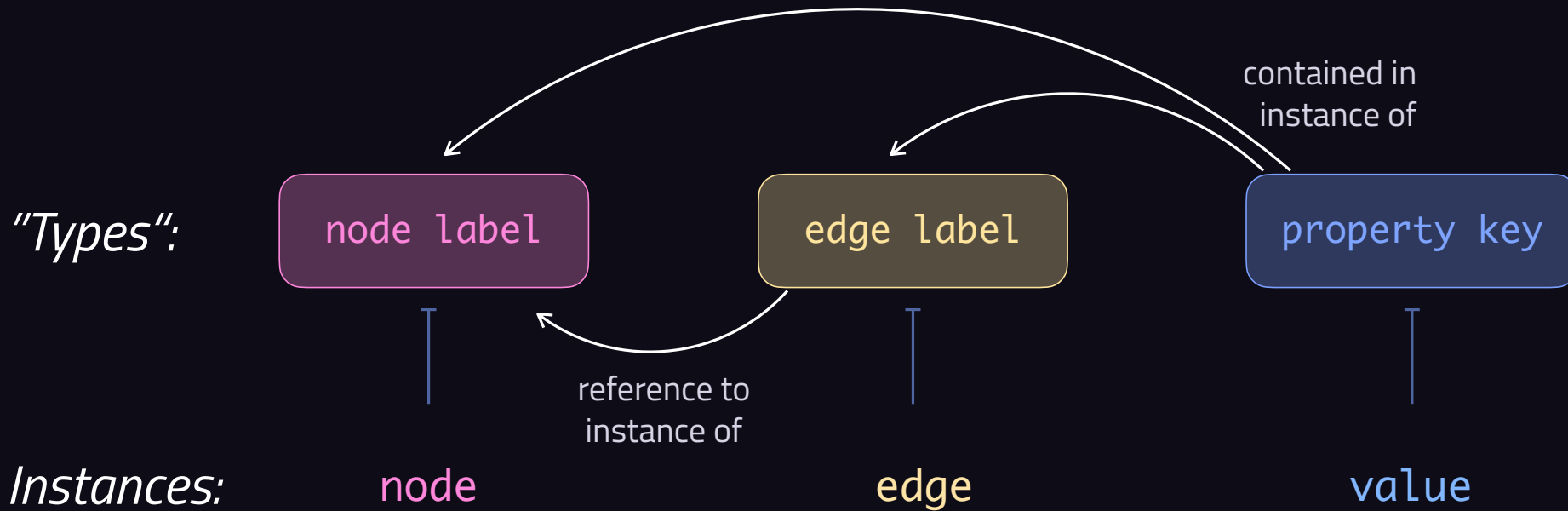
# The graph model



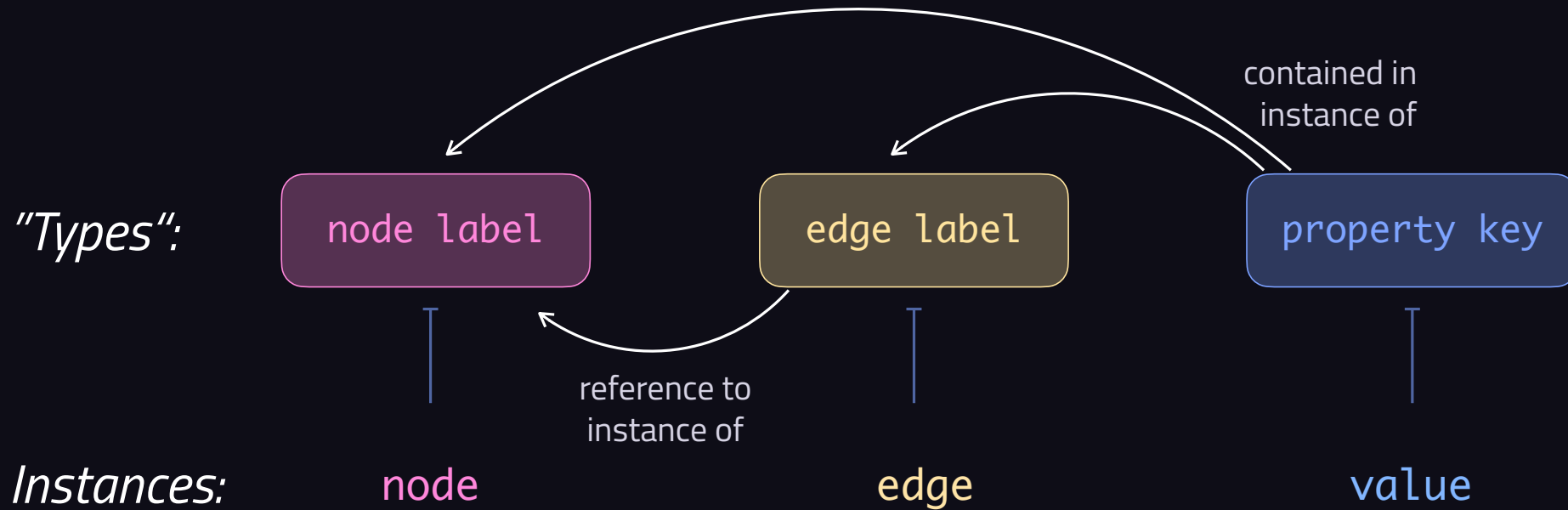
# The graph model



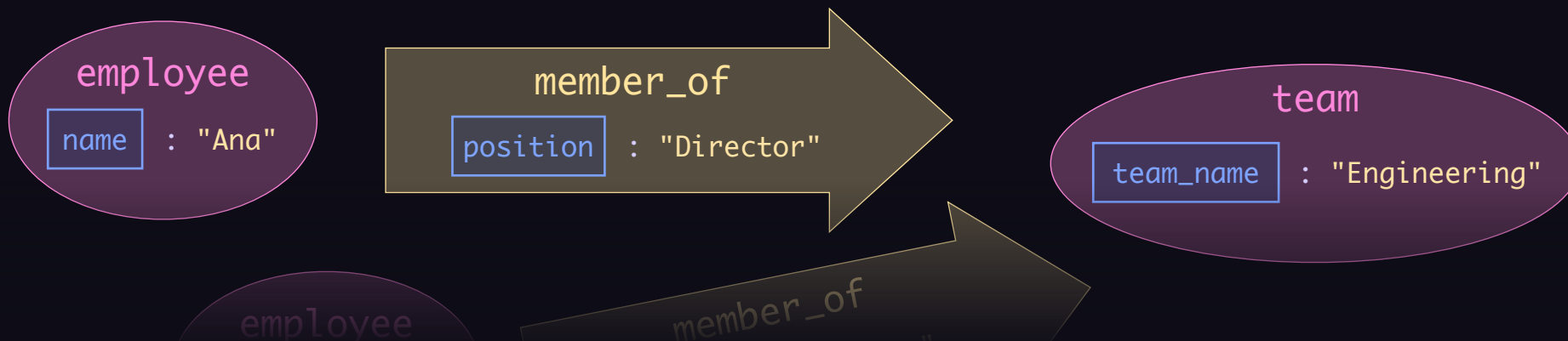
# The graph model



# The graph model

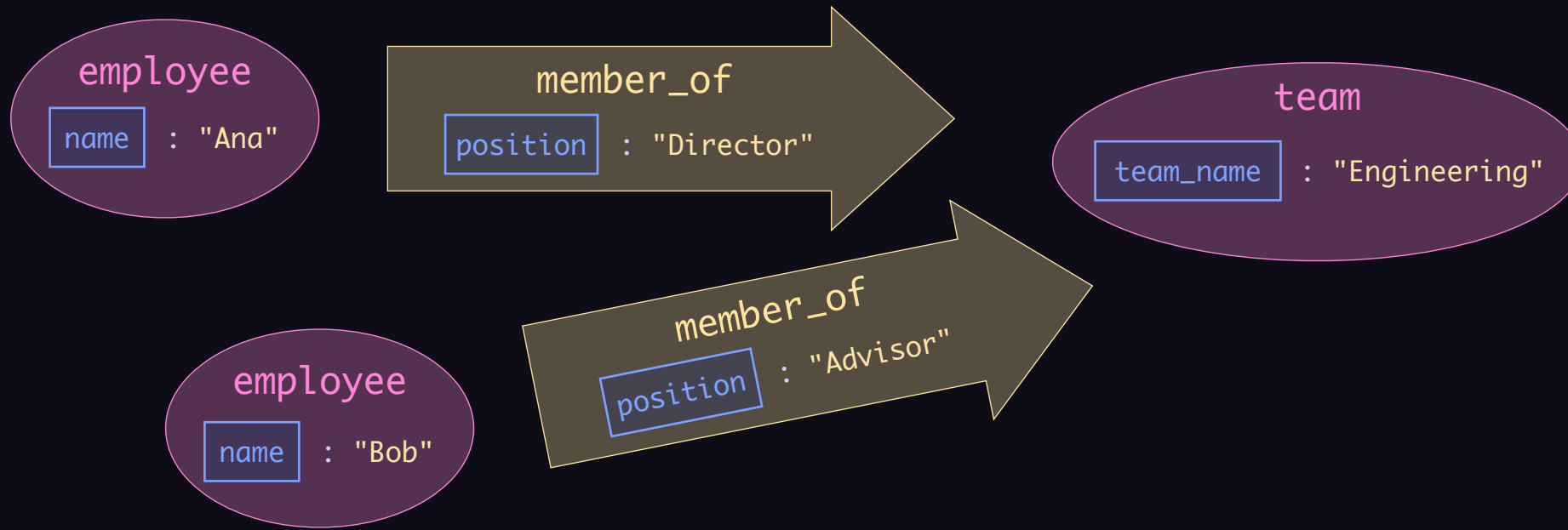


*Remark.*  
In the LPG, nodes and edges can be assigned multiple labels, which emulates "subtype hierarchies". However, this provides no meaningful way to model inheritance of type behaviors.



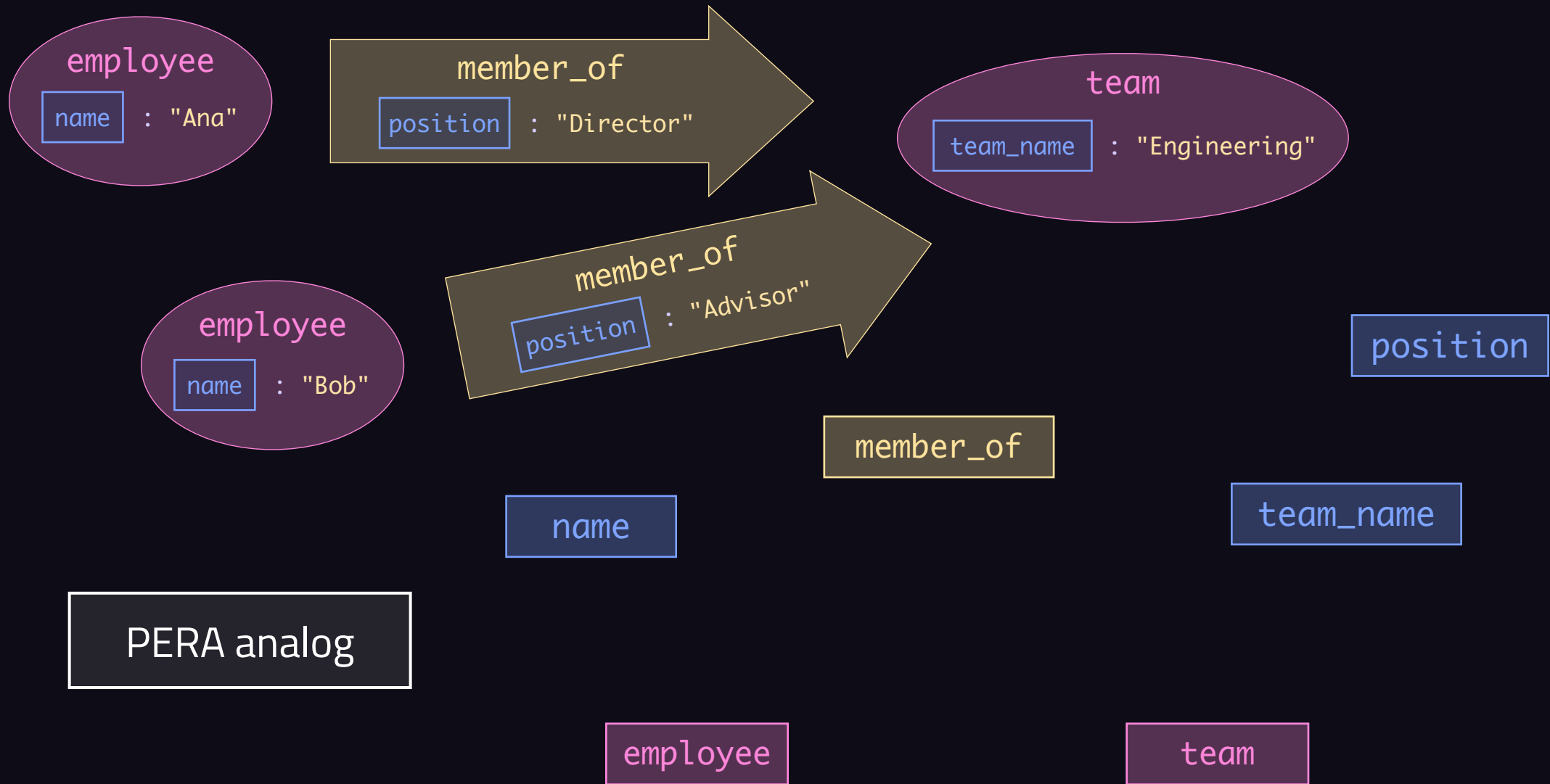


# The graph model: a simple example

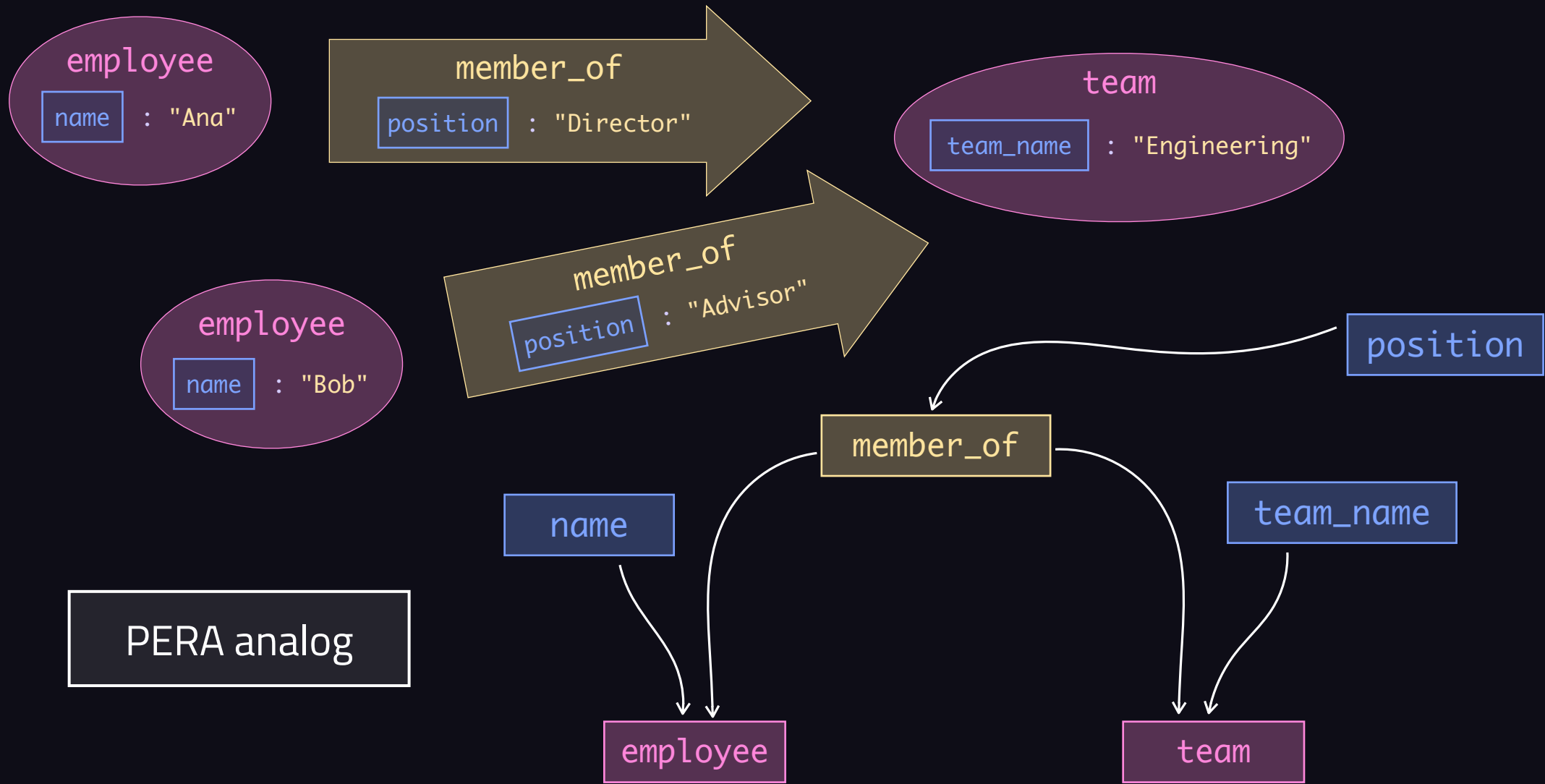


PERA analog

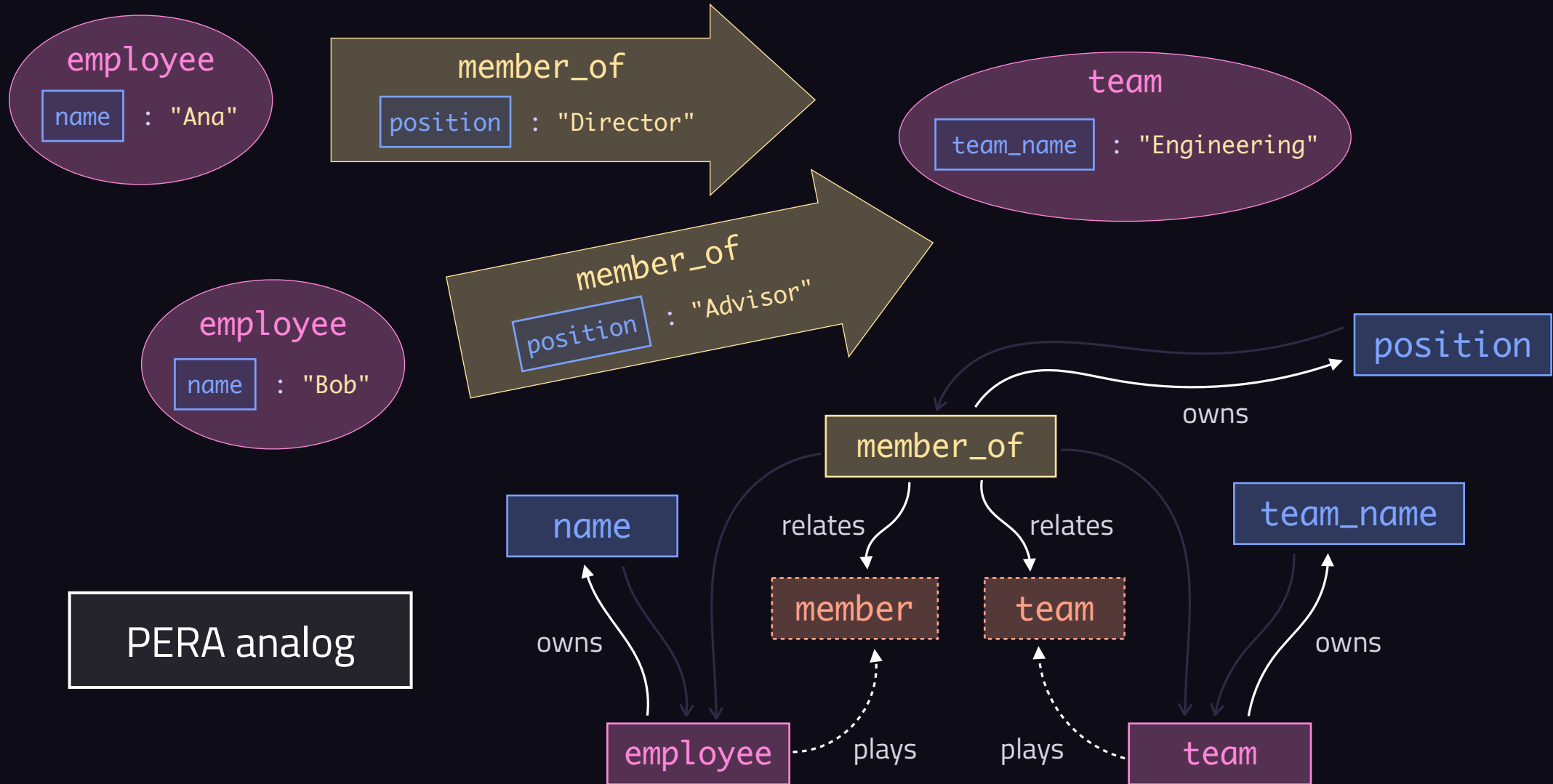
# The graph model: a simple example



# The graph model: a simple example



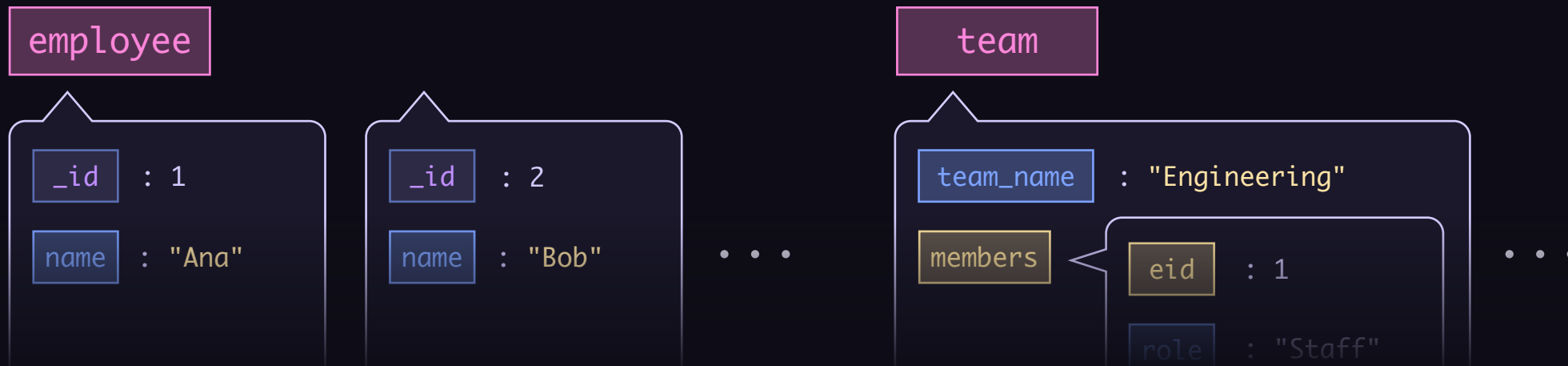
# The graph model: a simple example



# The document model

*"Types":*

*Instances:*



# The document model

*"Types":*

collection  
of docs



*Instances:*

document

employee

`_id` : 1

`name` : "Ana"

`_id` : 2

`name` : "Bob"

...

team

`team_name` : "Engineering"

`members`

`eid` : 1

`role` : "Staff"

...

# The document model

*"Types":*

collection  
of docs

key to  
subdoc

key to  
\_id

key to  
value

*Instances:*

document

employee

\_id : 1

name : "Ana"

\_id : 2

name : "Bob"

...

team

team\_name : "Engineering"

members

eid : 1

role : "Staff"

...

# The document model

*"Types":*

collection  
of docs

key to  
subdoc

key to  
\_id

key to  
value

*Instances:*

document

document

reference

value

employee

\_id : 1

name : "Ana"

\_id : 2

name : "Bob"

...

team

team\_name : "Engineering"

members

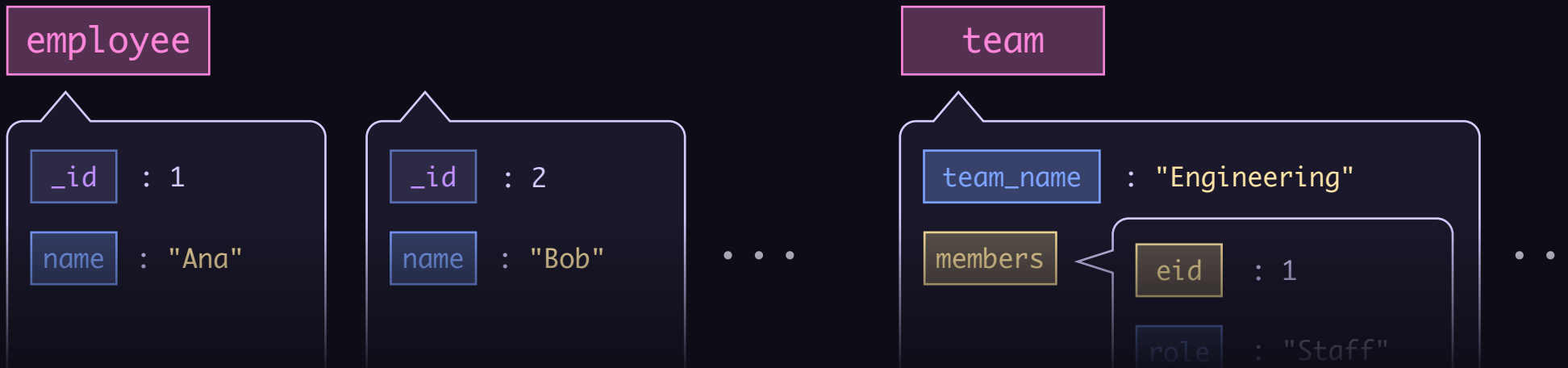
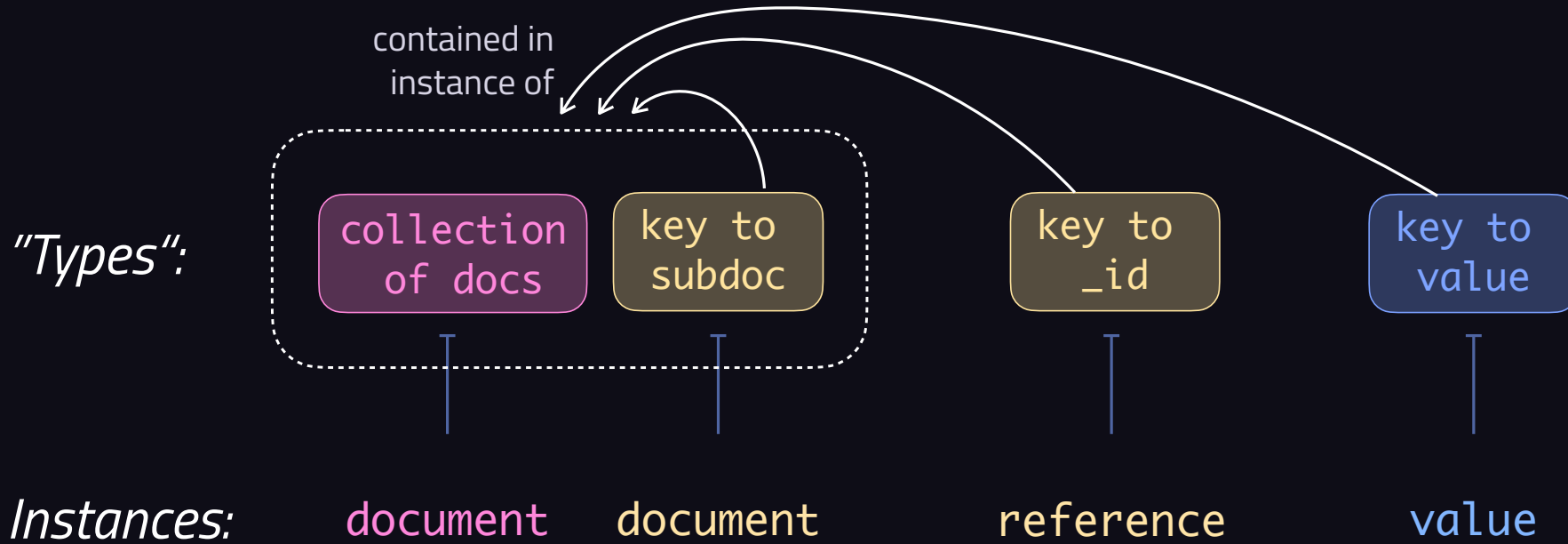
eid : 1

role : "Staff"

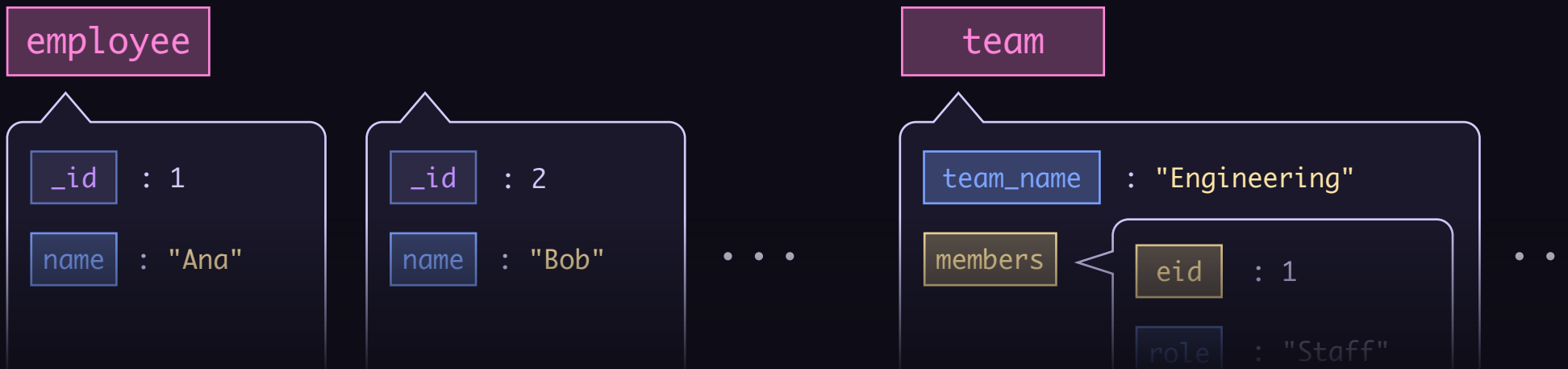
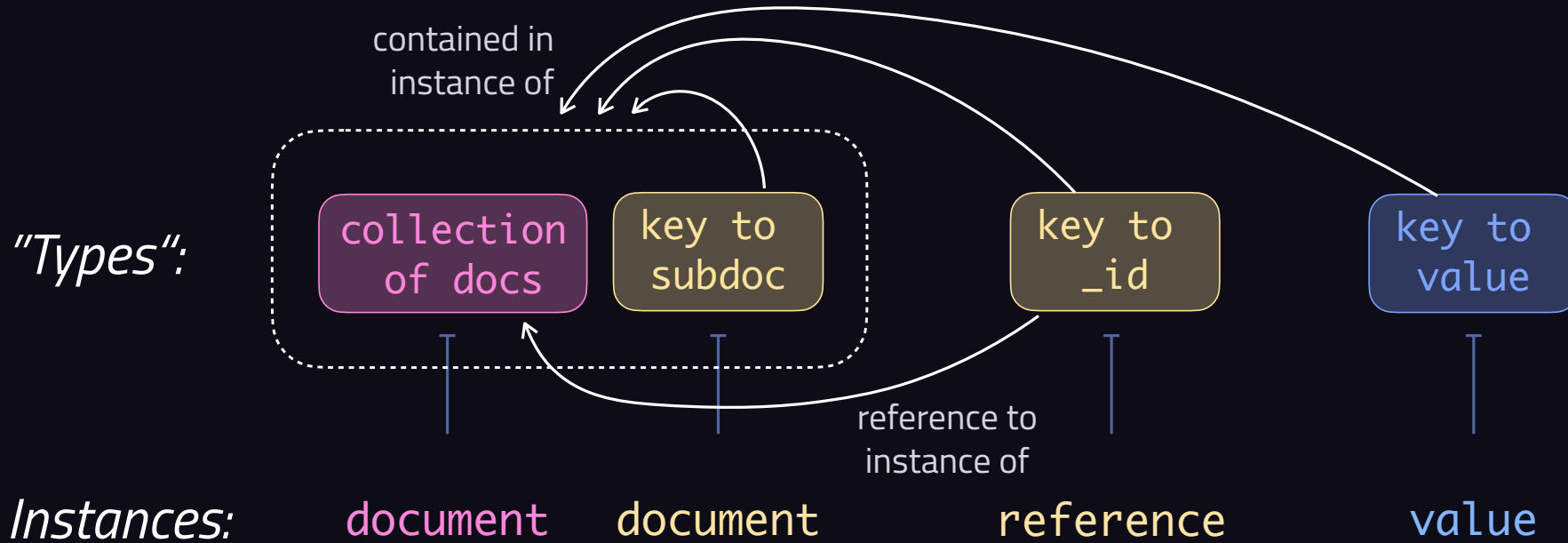
...



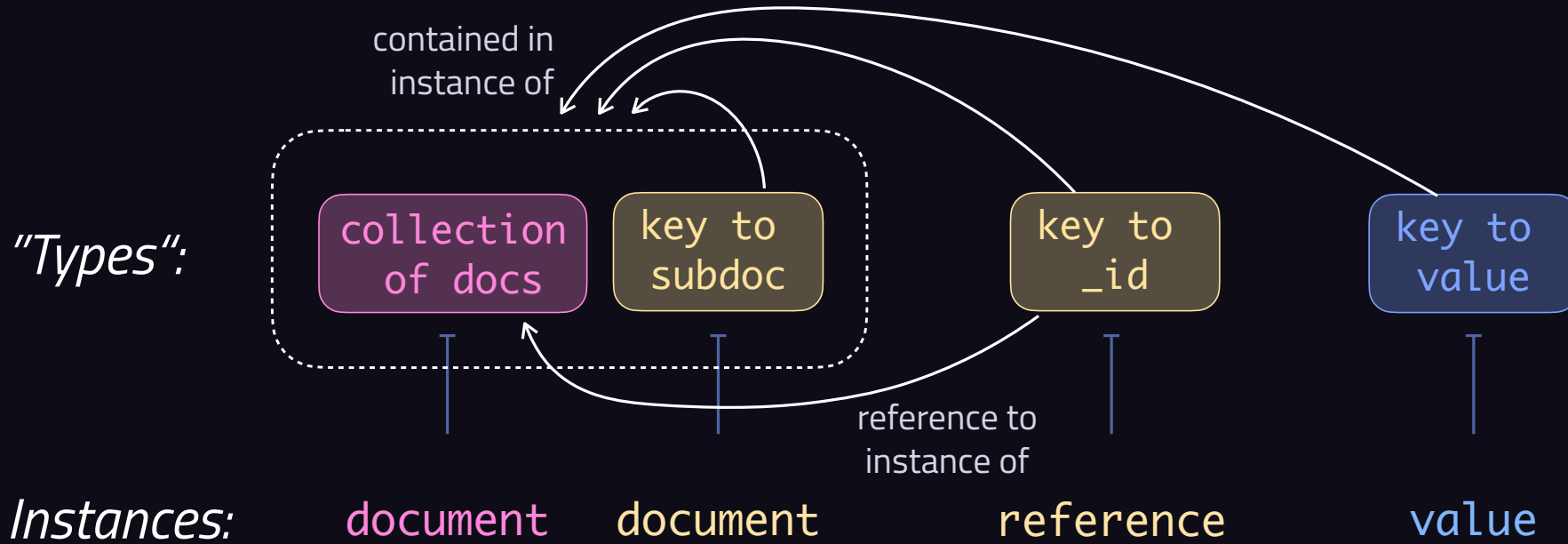
# The document model



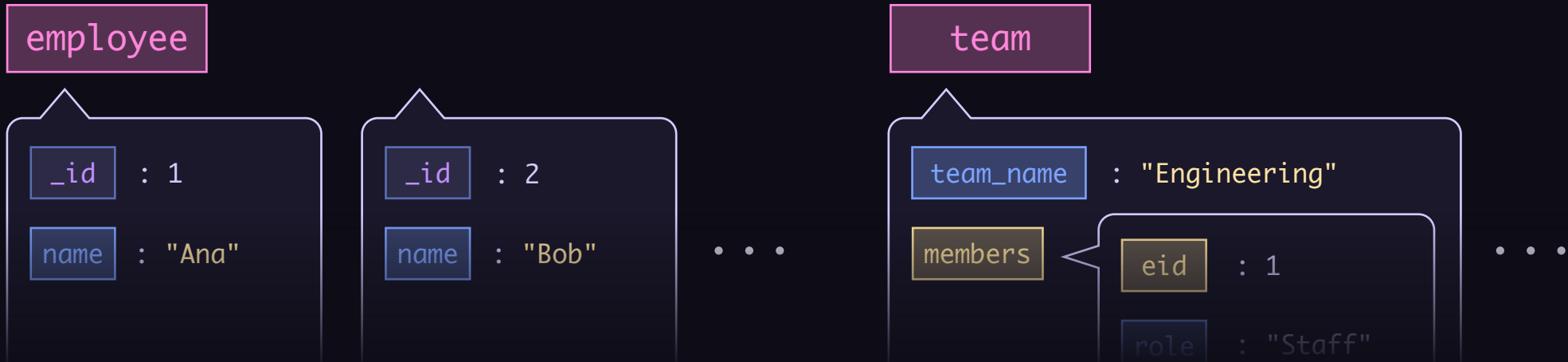
# The document model



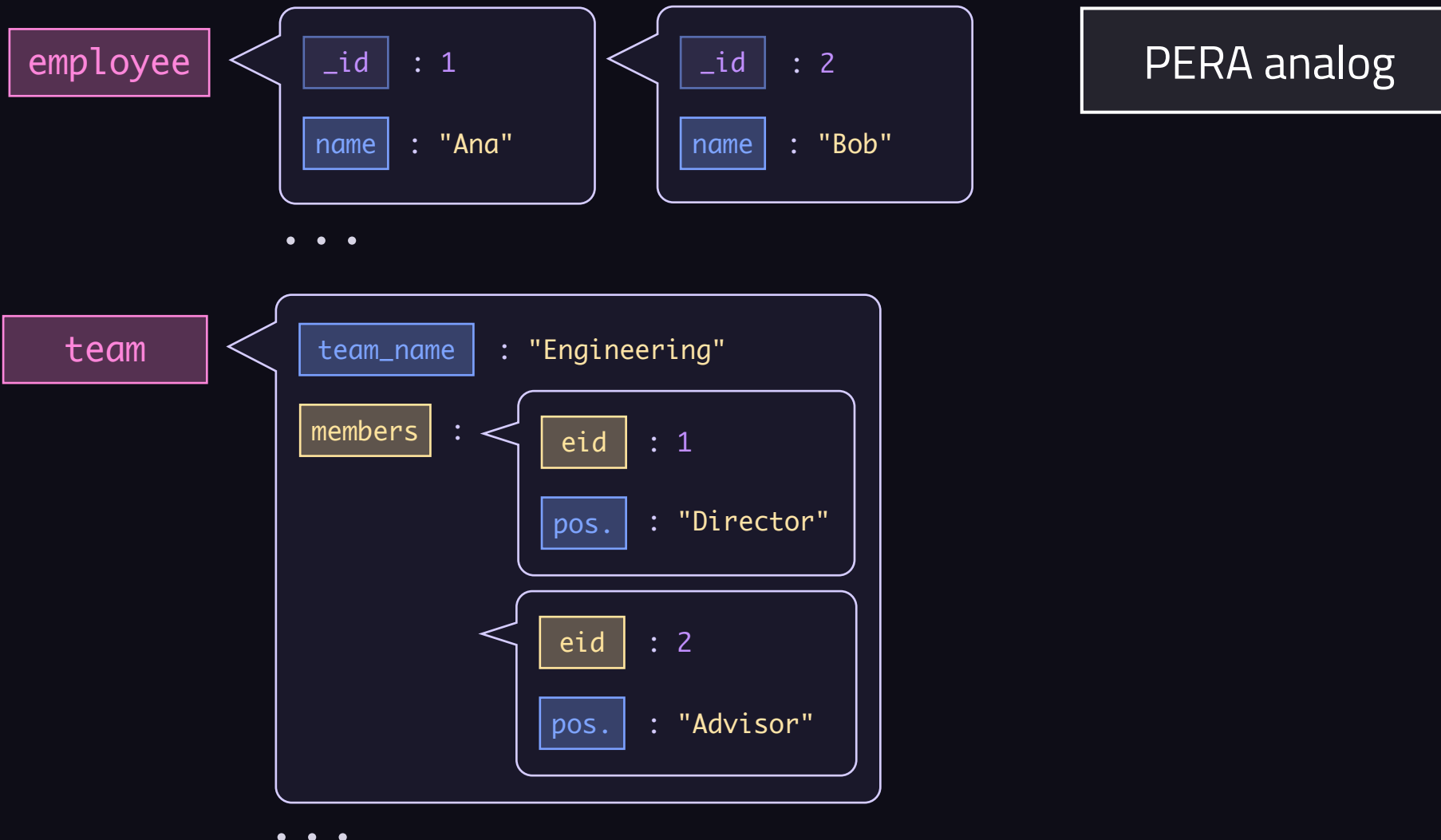
# The document model



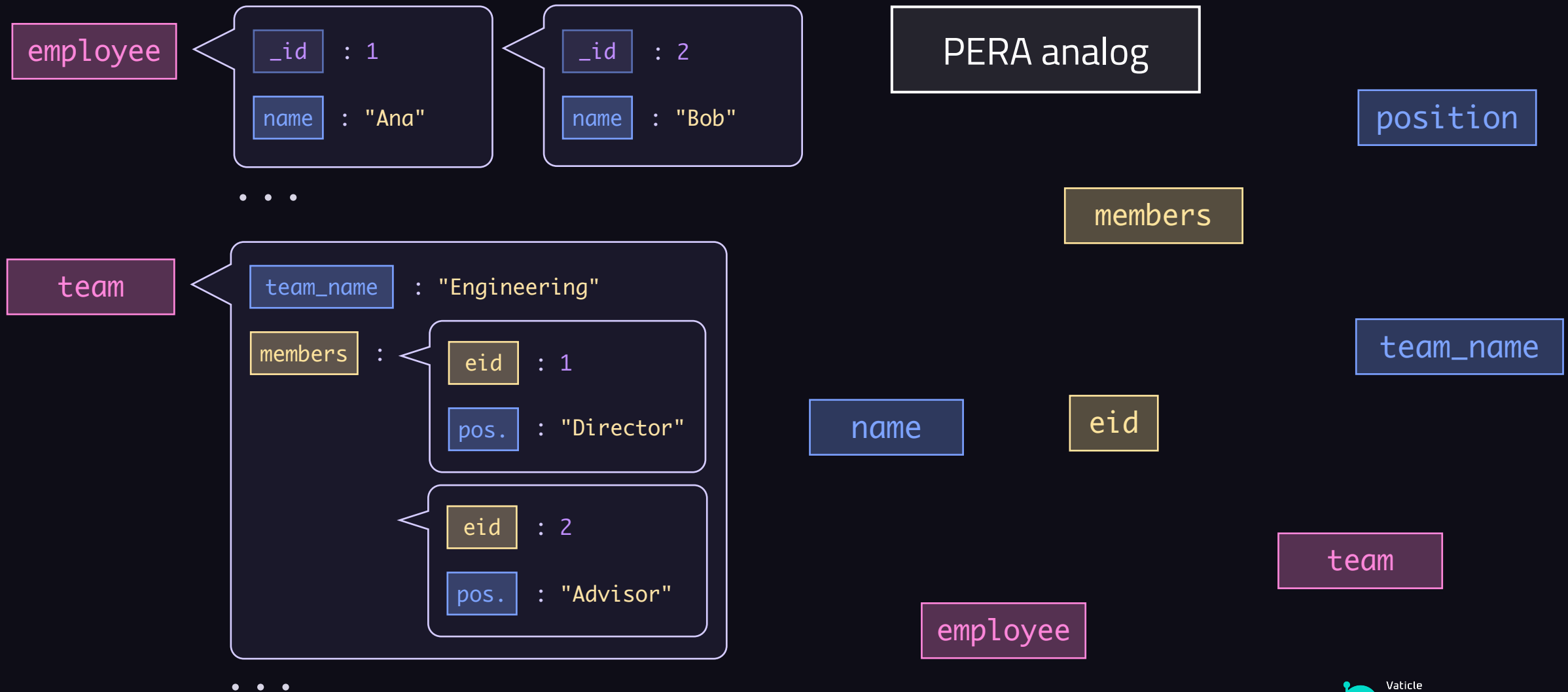
*Remark.*  
Semantically, some embedded sub-documents represent *duplicates* of other documents, but the document model does not relate these logically independent data instances.



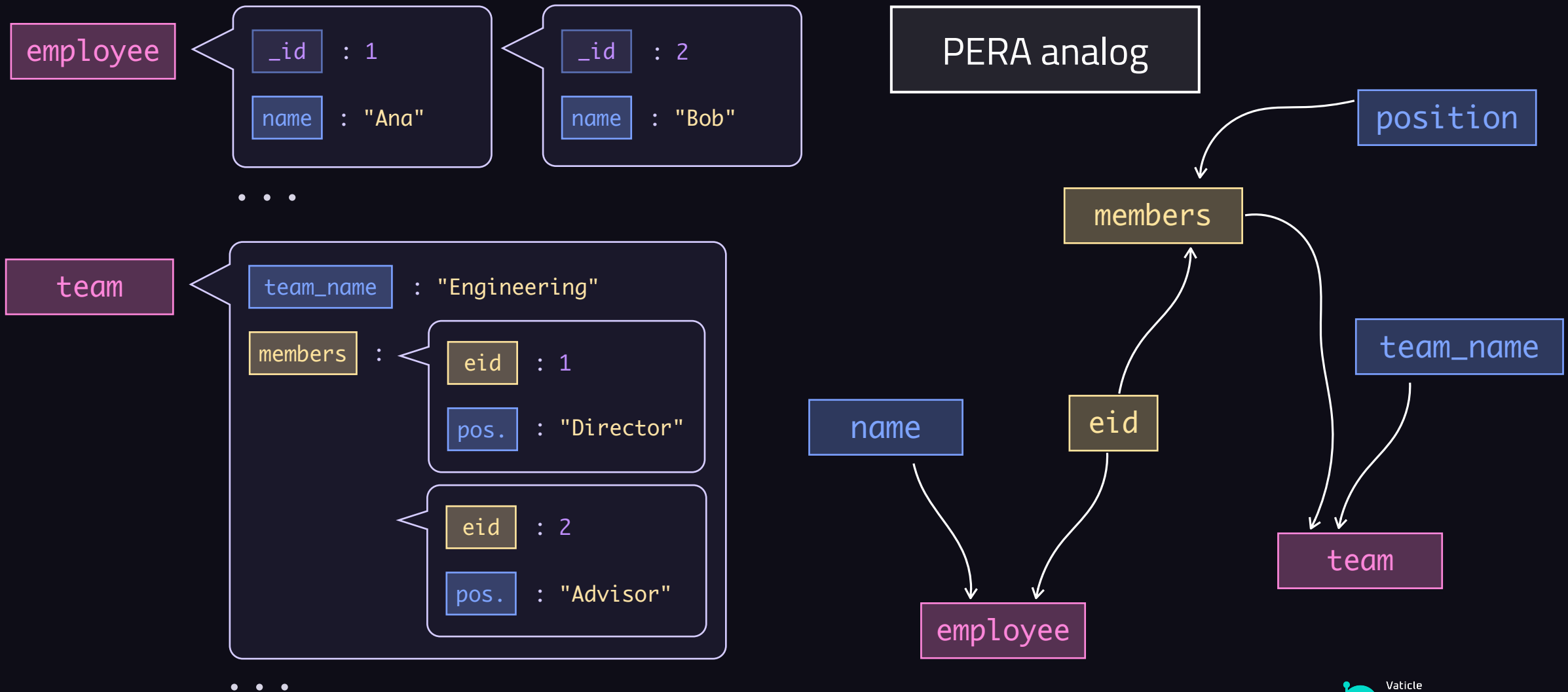
# The document model: a simple example



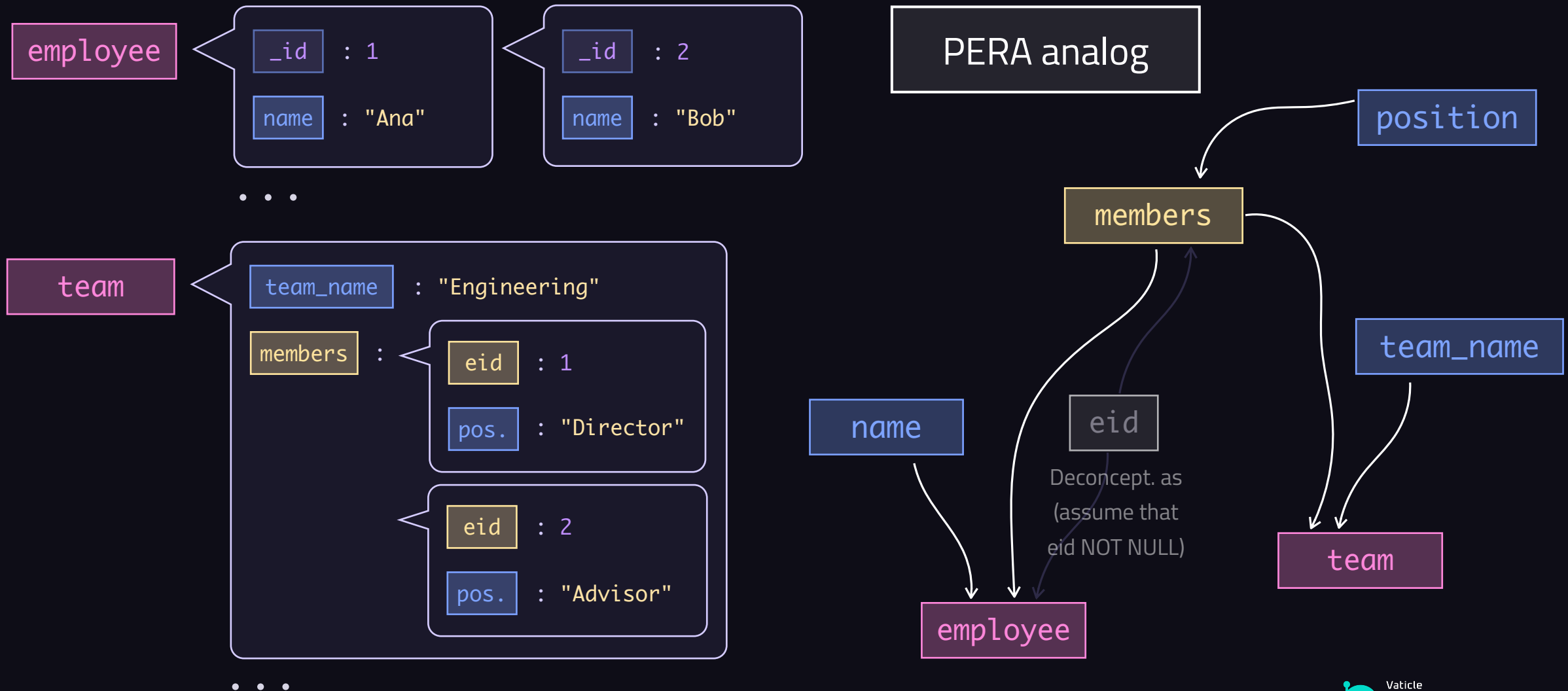
# The document model: a simple example



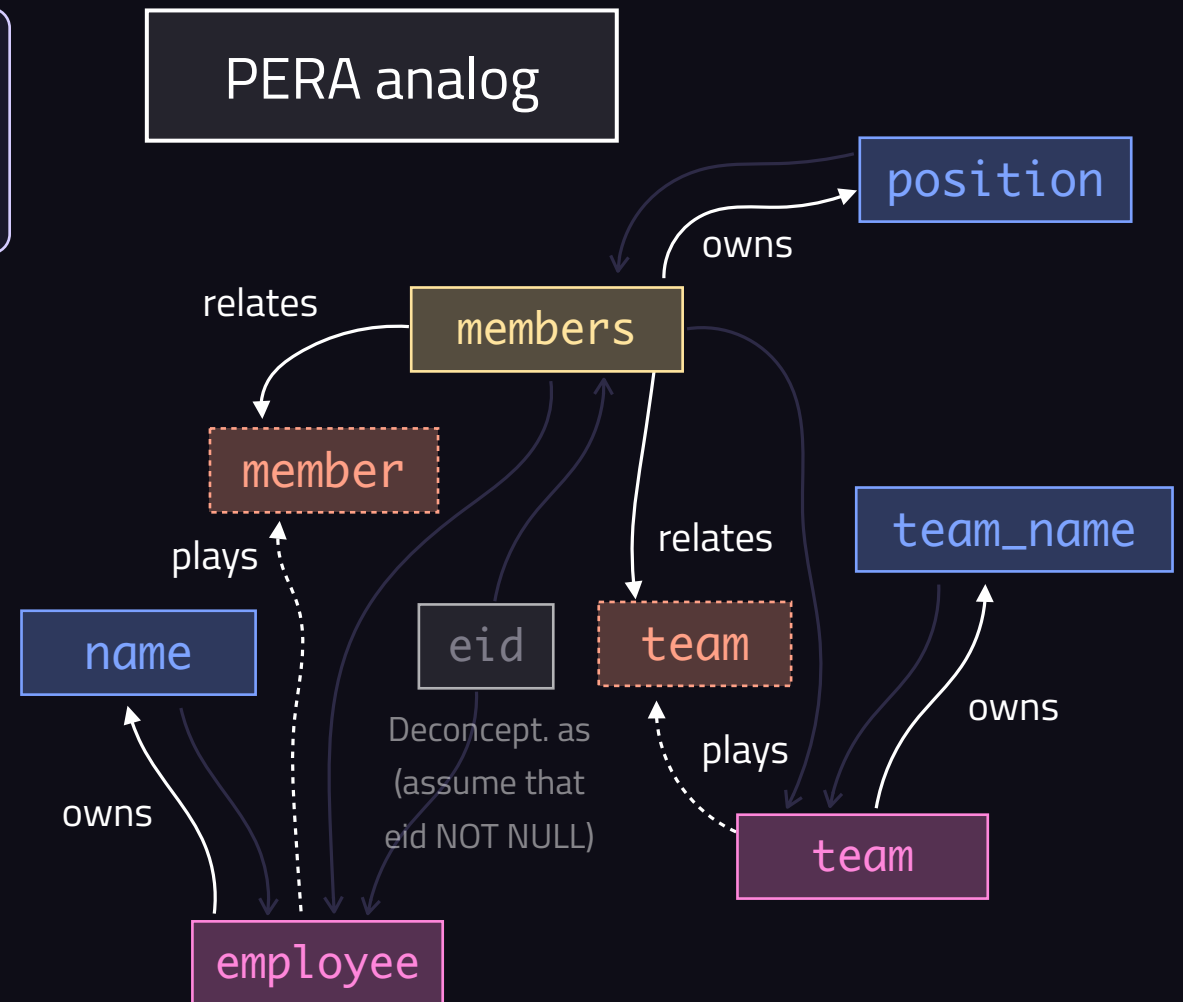
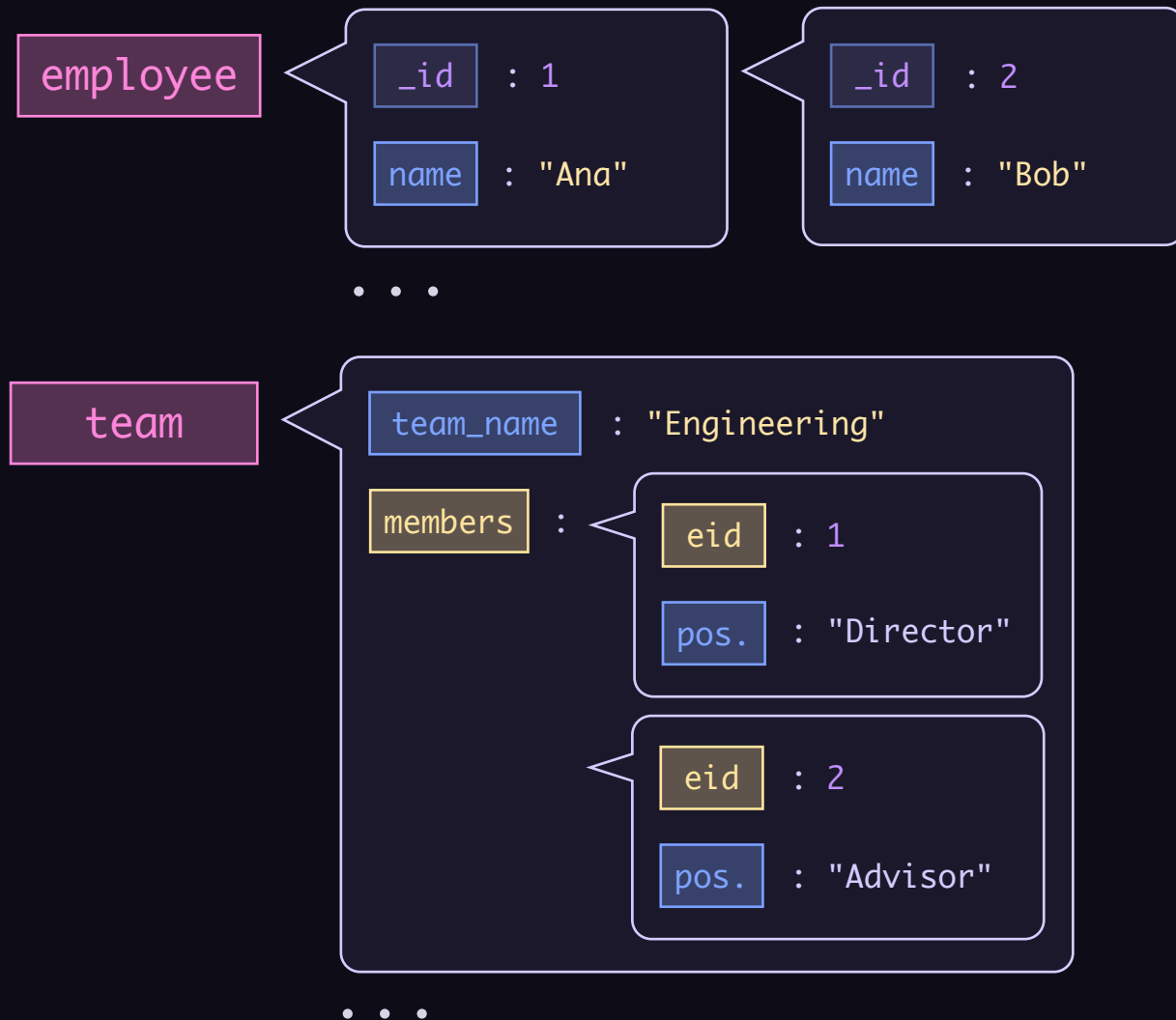
# The document model: a simple example



# The document model: a simple example

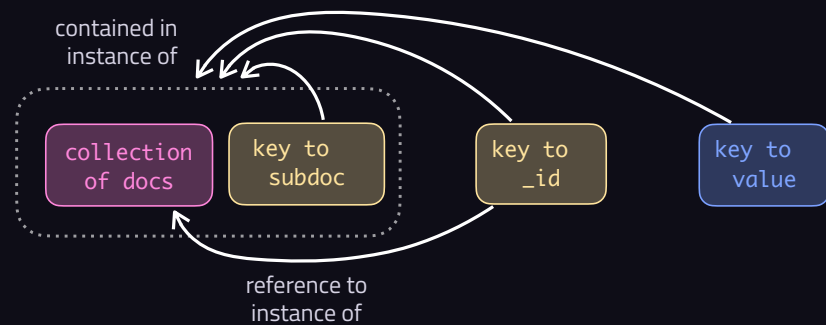
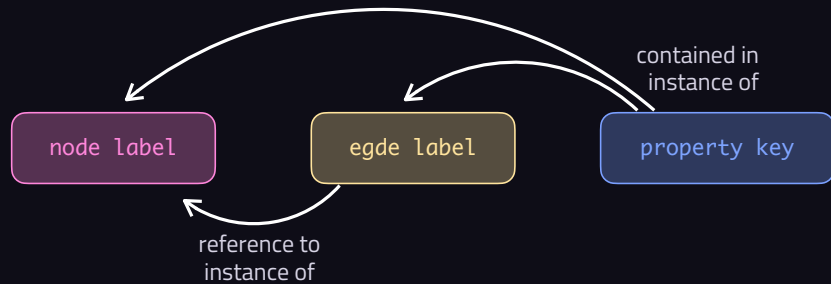
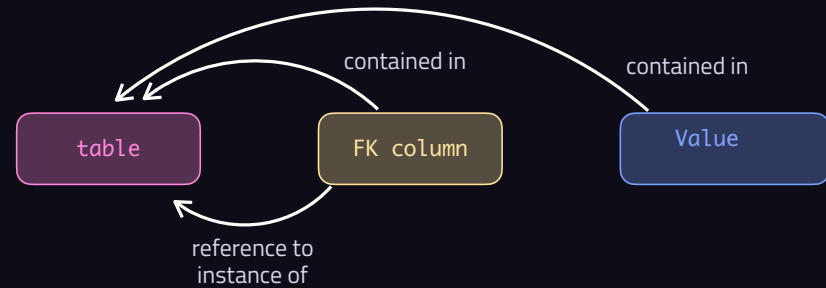


# The document model: a simple example

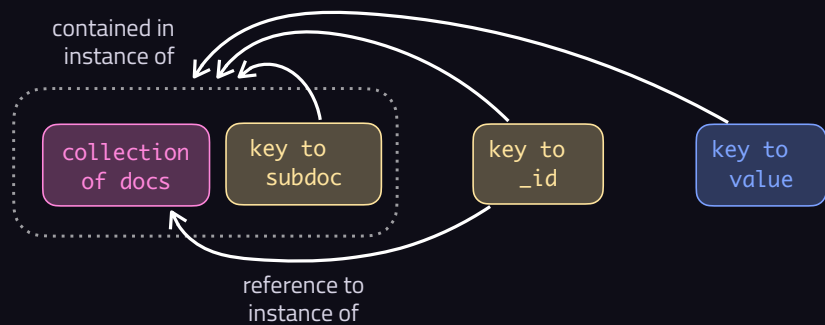
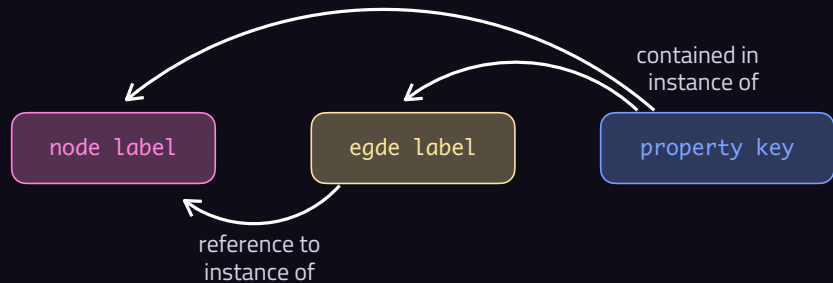
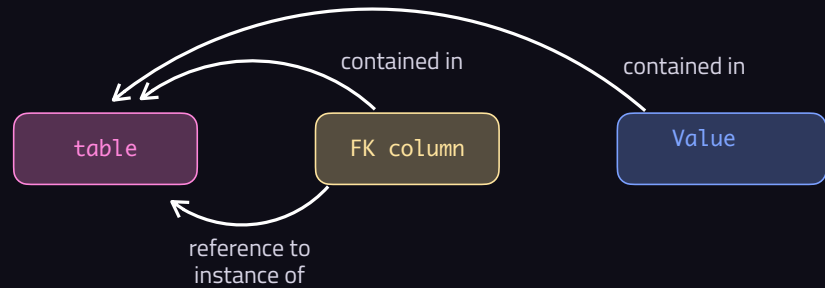




# Adding **polymorphism** to the picture



# Adding polymorphism to the picture

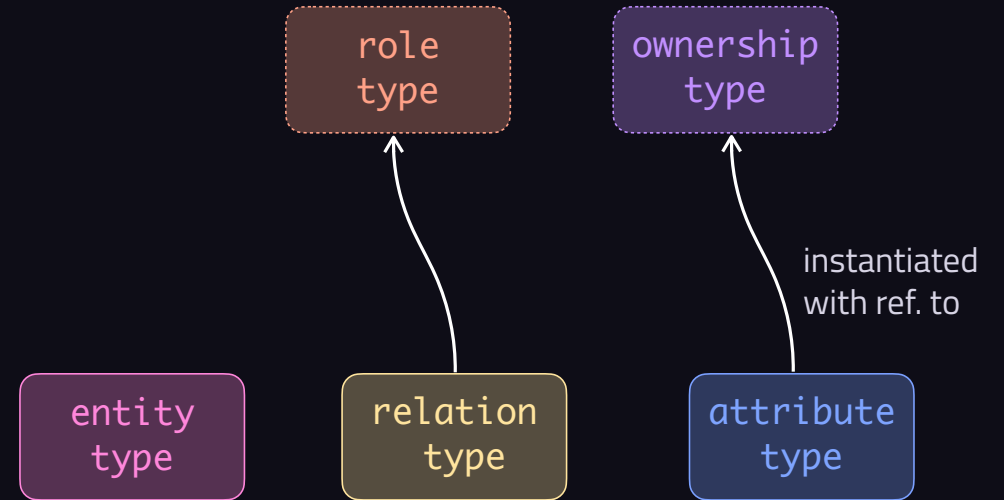
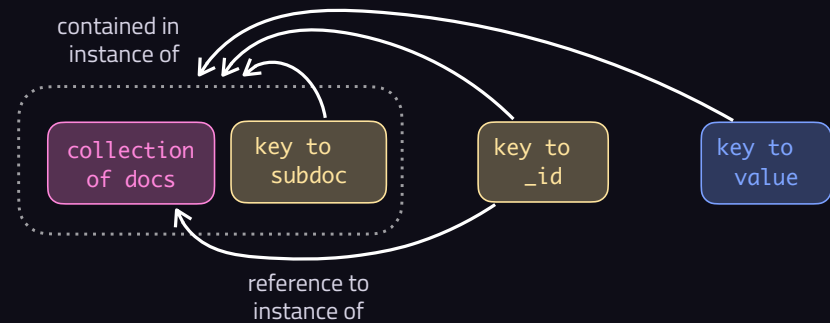
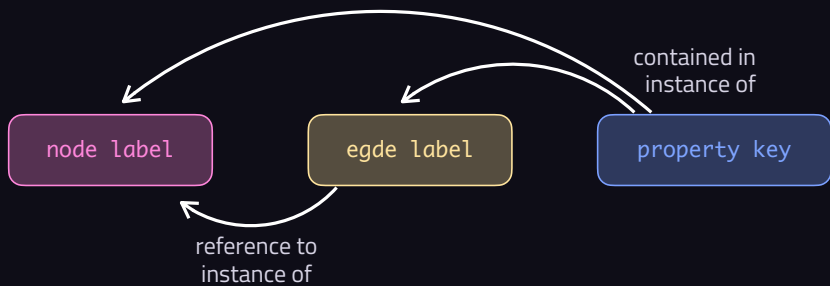
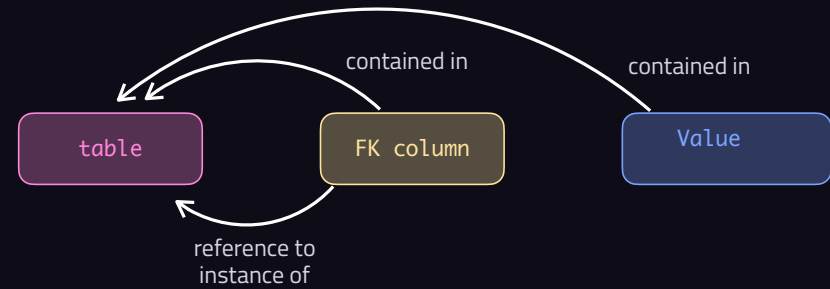


entity type

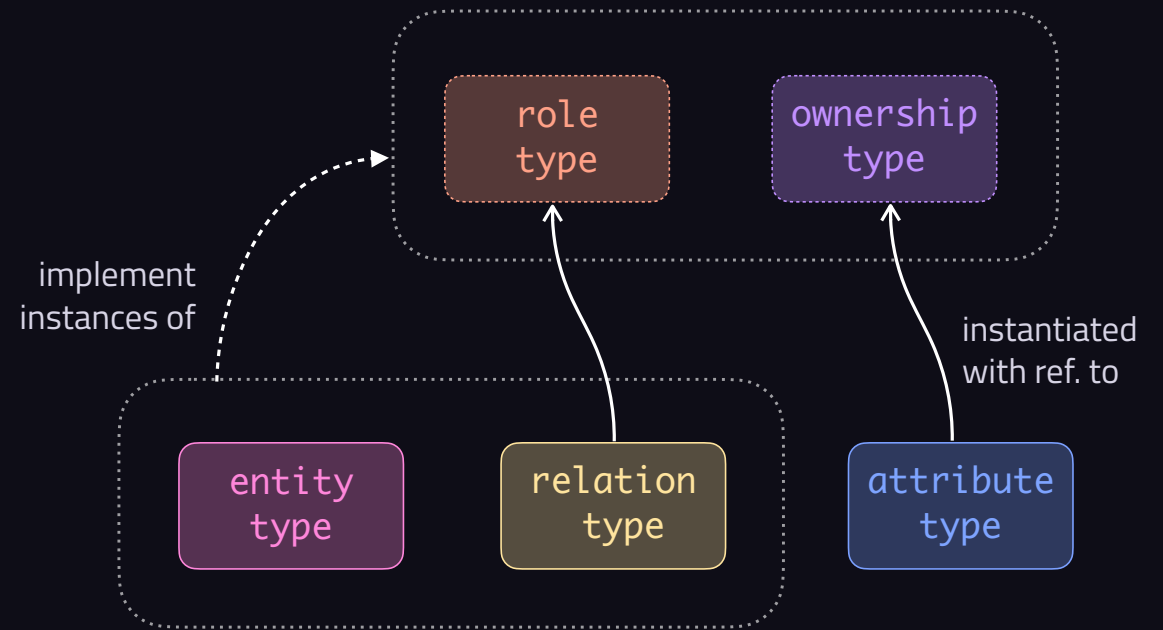
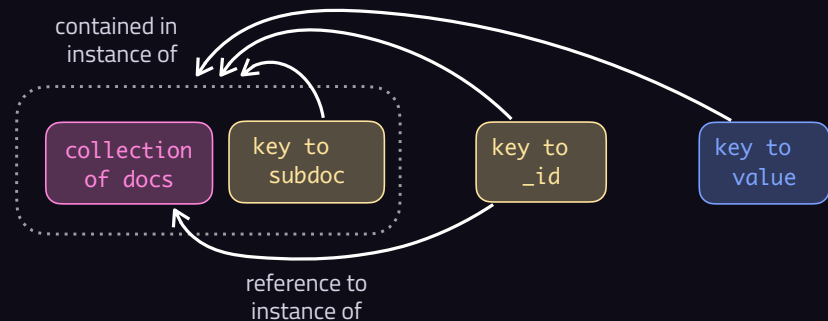
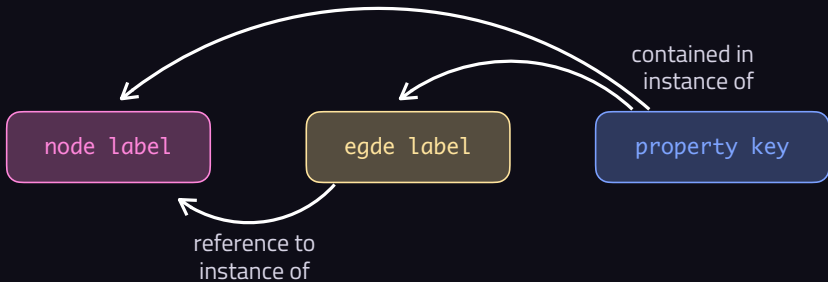
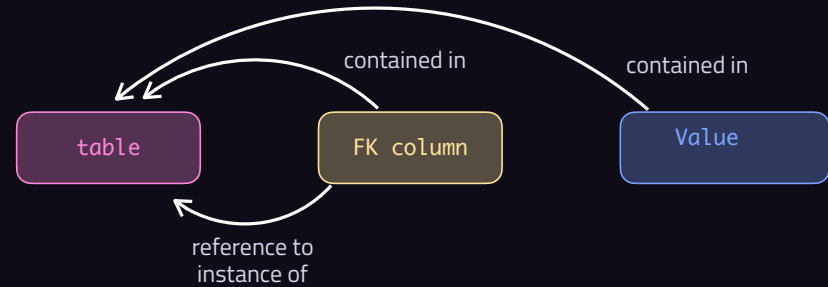
relation type

attribute type

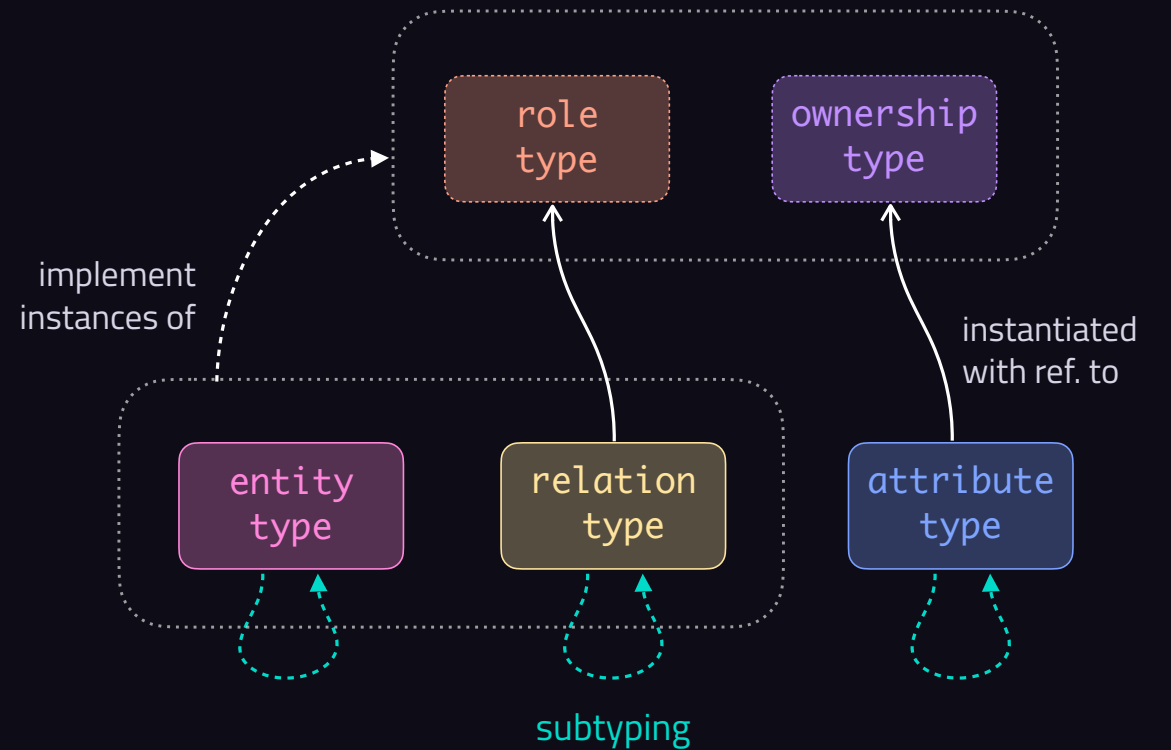
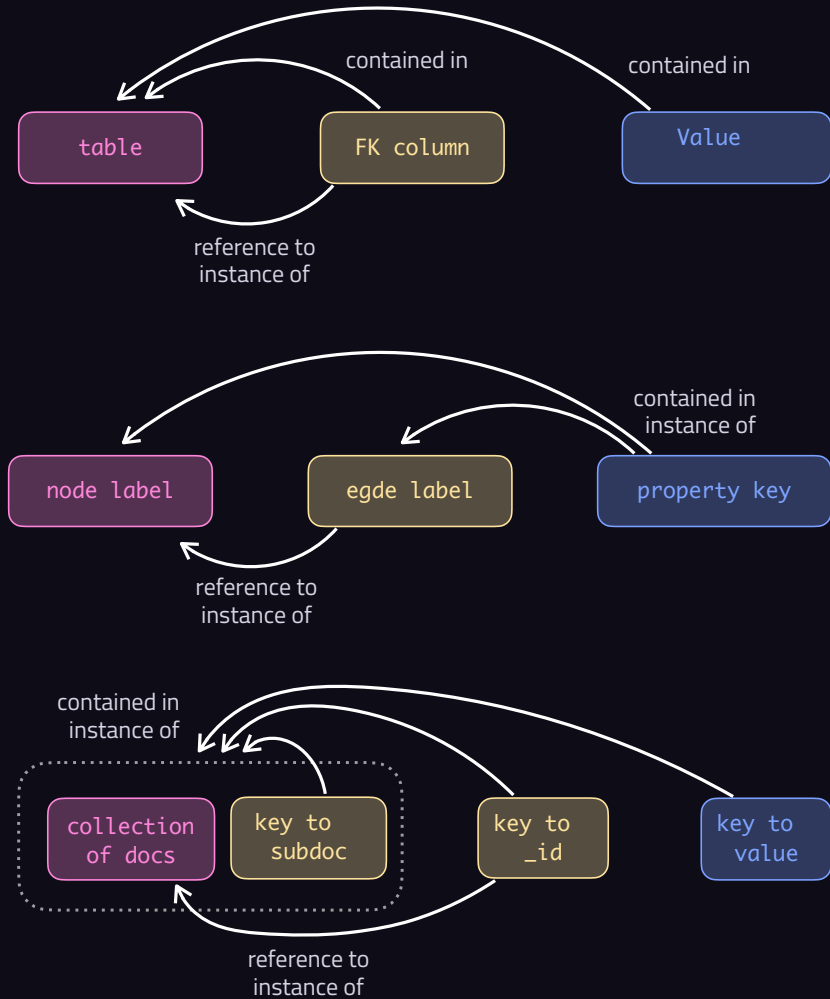
# Adding **polymorphism** to the picture



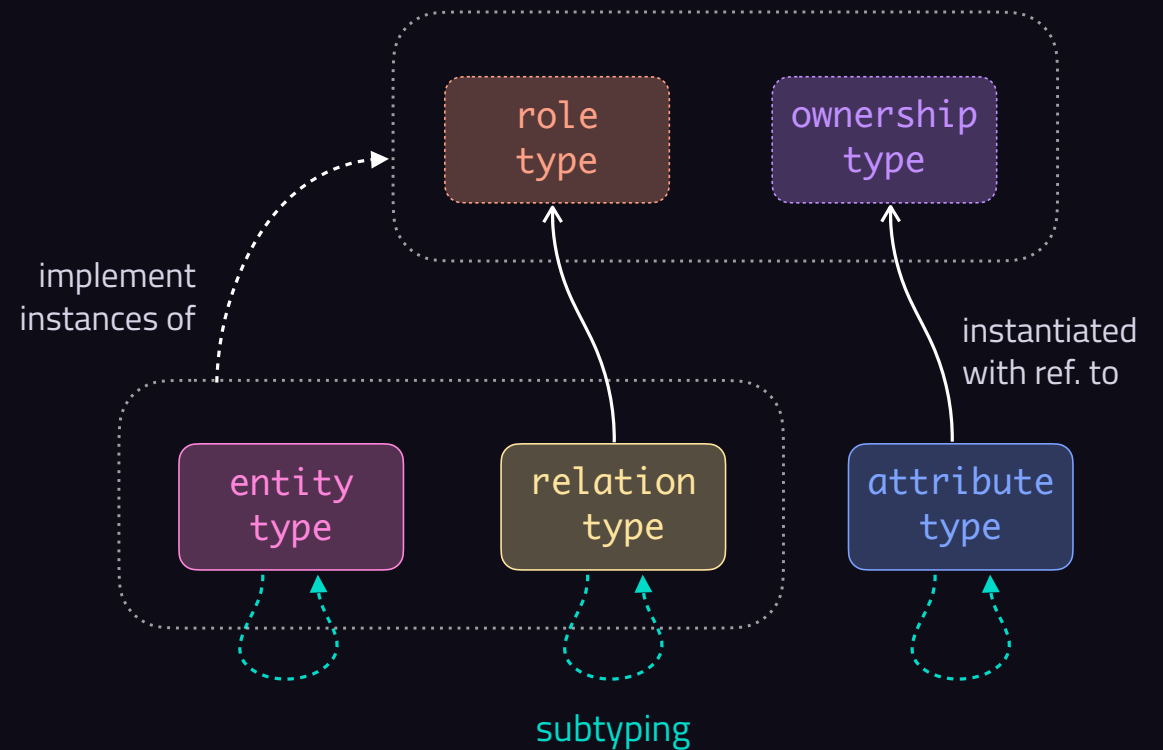
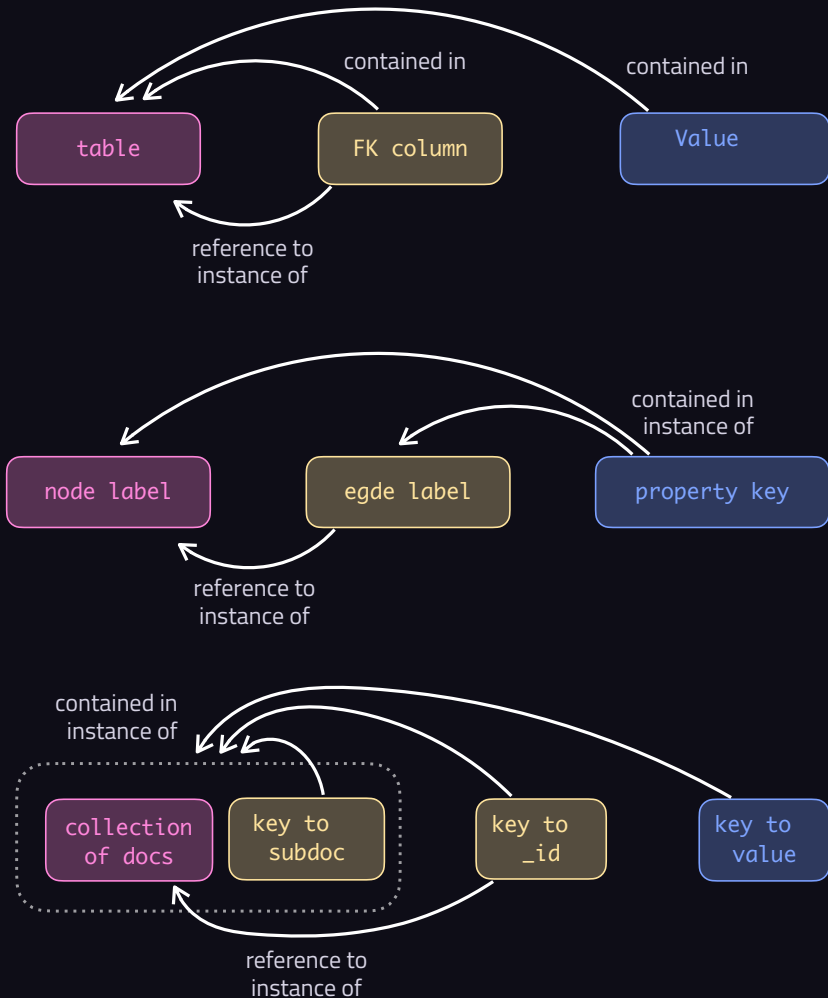
# Adding polymorphism to the picture



# Adding **polymorphism** to the picture



# Adding polymorphism to the picture



principled + modular + polymorphic

# Summary and further remarks

## Summary and further remarks

- In comparison to other data models, the PERA model is

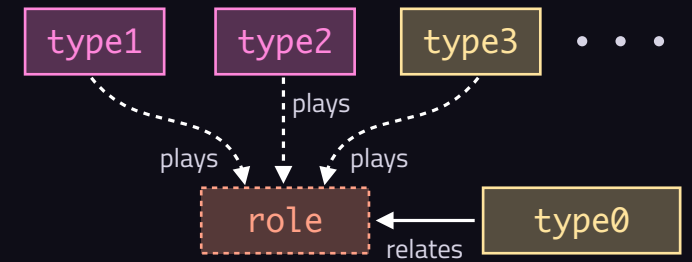


## Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function

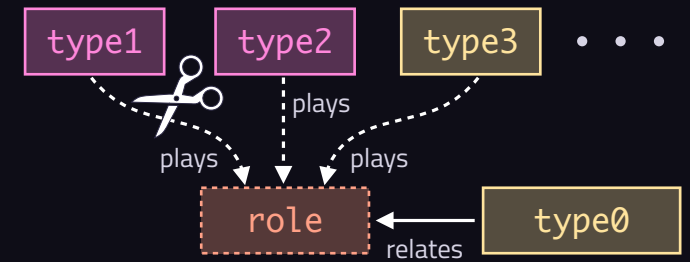
# Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function
  - **Modular:** **safely modify** implementations of dependencies



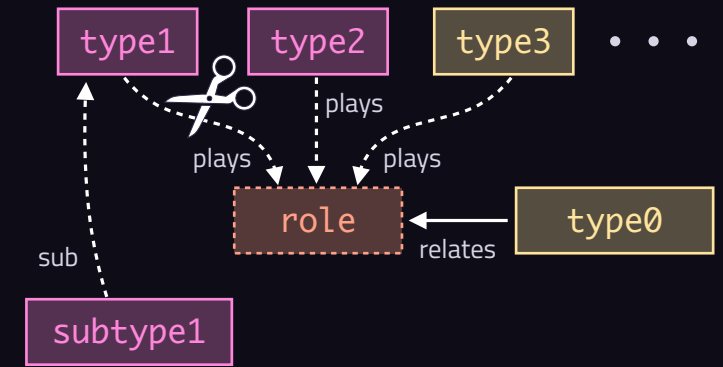
# Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function
  - **Modular:** **safely modify** implementations of dependencies



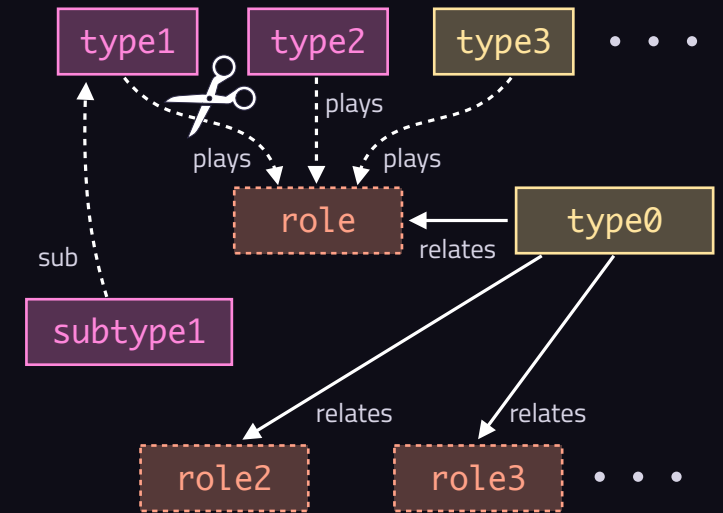
# Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function
  - **Modular:** **safely modify** implementations of dependencies
  - **Polymorphic:** directly express type **capabilities** and type **inheritance**



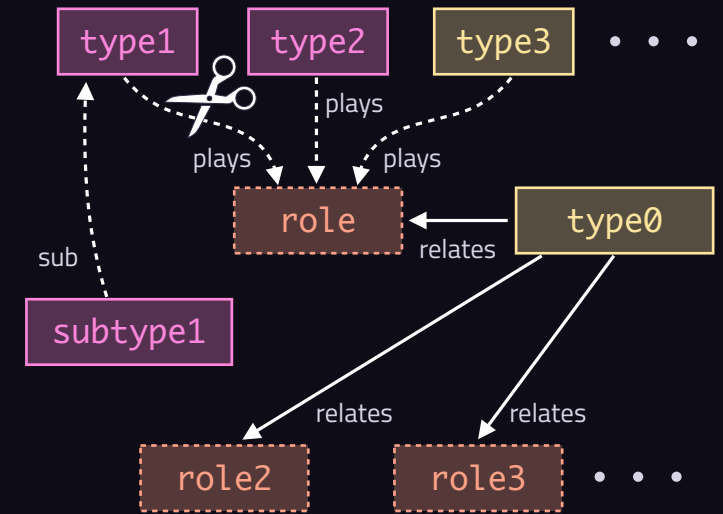
# Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function
  - **Modular:** **safely modify** implementations of dependencies
  - **Polymorphic:** directly express type **capabilities** and type **inheritance**
  - ... and more: variadicity, globality, ...



# Summary and further remarks

- In comparison to other data models, the PERA model is
  - **Principled:** types have **explicit and unambiguous** function
  - **Modular:** **safely modify** implementations of dependencies
  - **Polymorphic:** directly express type **capabilities** and type **inheritance**
  - ... and more: variadicity, globality, ...
- The PERA model follows a *type-first approach*
  - In schema-less approaches types are an *after-thought*
  - Requires us to think about types *ahead of time*, but modularity makes *continuous development* painless



second\_favorite\_music\_instrument: "Guitar"

no surprise fields!



# Epilogue: Beyond the basics

Polymorphic *pattern-based* querying and reasoning

# Polymorphic querying and reasoning



# Polymorphic querying and reasoning

- *Type-theoretic querying*
  - Queries via **patterns** ... patterns via **types**
  - Elegant and **intuitive querying** paradigm

# Polymorphic querying and reasoning

- *Type-theoretic querying*
  - Queries via **patterns** ... patterns via **types**
  - Elegant and **intuitive querying** paradigm

*"data type of solutions"*

→ *fully declarative QL*

# Polymorphic querying and reasoning

- *Type-theoretic querying*
    - Queries via **patterns** ... patterns via **types**
    - Elegant and **intuitive querying** paradigm
  - *Type-theoretic reasoning*
    - Type system is a collection of rules (e.g.: subtyping)
    - **User-defined rules** can extend this to derive new data
- “data type of solutions”*  
→ *fully declarative QL*

# Polymorphic querying and reasoning

- *Type-theoretic querying*
    - Queries via **patterns** ... patterns via **types**
    - Elegant and **intuitive querying** paradigm
  - *Type-theoretic reasoning*
    - Type system is a collection of rules (e.g.: subtyping)
    - **User-defined rules** can extend this to derive new data
- "data type of solutions"*  
→ *fully declarative QL*

Both are key functions of the PERA model ...let's give a brief preview!

# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

## Pattern

*statements with variables that can be "solved for" by substituting objects and attributes for variables*

# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

## Pattern

*statements with variables that can be "solved for" by substituting objects and attributes for variables*

- `match` clause, line 1
  - `$eng_team` is some existing team object
  - that object must be the owner of the `team_name` "Engineering"

# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

## Pattern

statements with variables that can be "solved for" by substituting objects and attributes for variables

- `match` clause, line 1
  - `$eng_team` is some existing team object
  - that object must be the owner of the `team_name` "Engineering"
- `match` clause, line 2
  - there exists a `team_membership` object, left unassigned, but required to have a team roleplayer `$eng_team` and a member roleplayer `$m`



# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

## Pattern

statements with variables that can be "solved for" by substituting objects and attributes for variables

- `match` clause, line 1
  - `$eng_team` is some existing team object
  - that object must be the owner of the `team_name` "Engineering"
- `match` clause, line 2
  - there exists a `team_membership` object, left unassigned, but required to have a team roleplayer `$eng_team` and a member roleplayer `$m`

What concept (i.e. type) does `$m` belong to?

The answer is "polymorphic": `$m` can belong to any type that implements the member role.

# Pattern-based querying with interface polymorphism

```
match
  $eng_team isa team, has team_name "Engineering";
  (team: $eng_team, member: $m) isa team_membership;
fetch
  $m as engineer: name;
```

## Pattern

statements with variables that can be "solved for" by substituting objects and attributes for variables

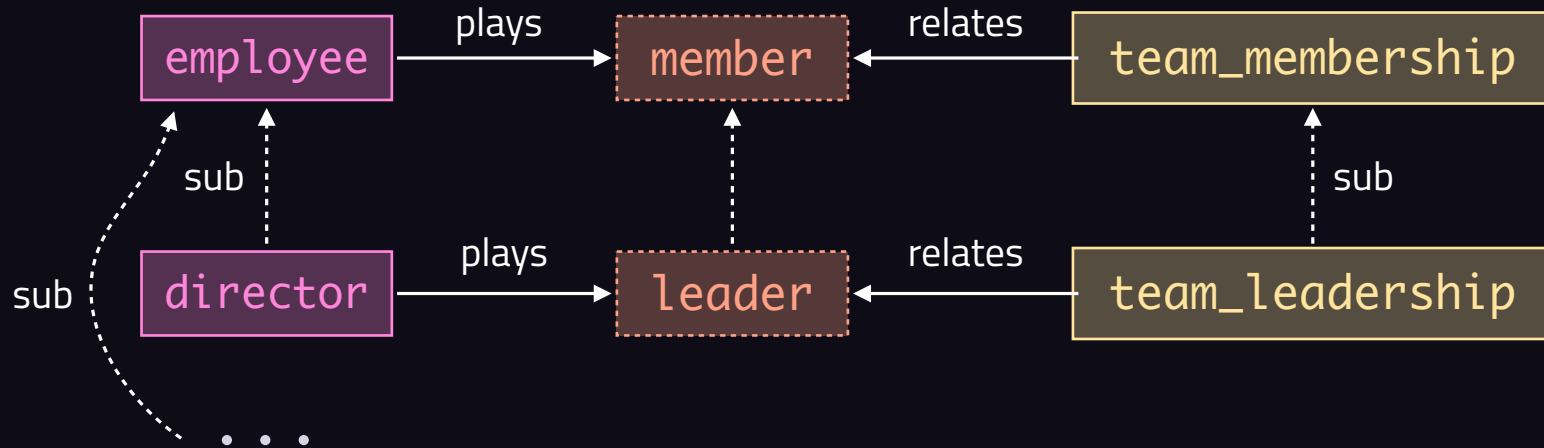
- `match` clause, line 1
  - `$eng_team` is some existing team object
  - that object must be the owner of the `team_name` "Engineering"
- `match` clause, line 2
  - there exists a `team_membership` object, left unassigned, but required to have a `team` roleplayer `$eng_team` and a `member` roleplayer `$m`
- `fetch` clause, line 1
  - for each "engineer" object `$m` return the `name` attribute(s) of `$m`

What concept (i.e. type) does `$m` belong to?

The answer is "polymorphic": `$m` can belong to any type that implements the `member` role.

# Pattern-based querying with inheritance polymorphism

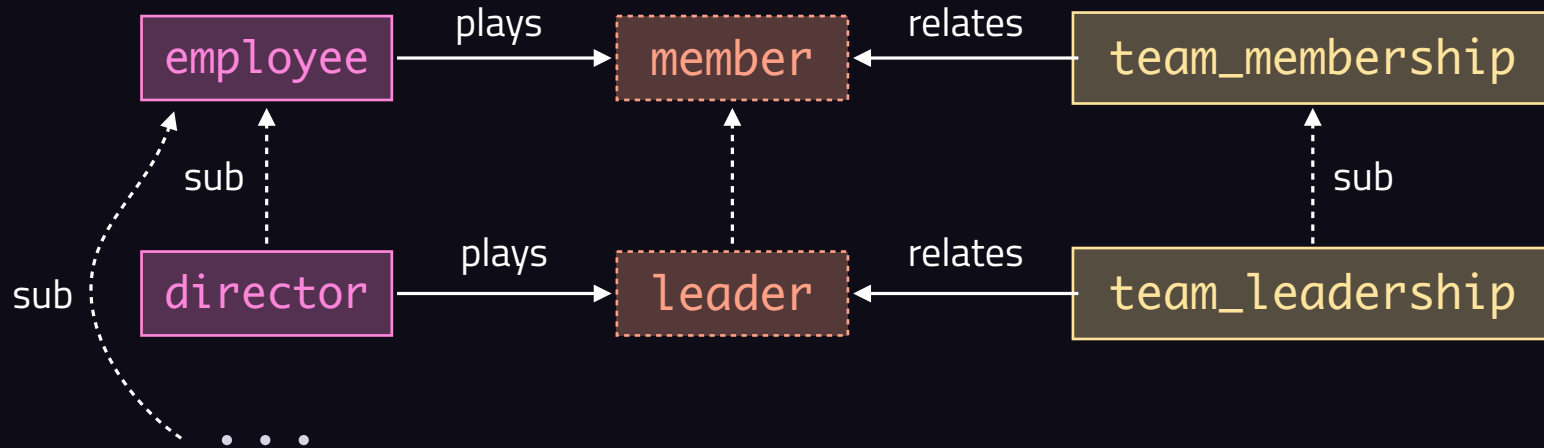
```
match
  $e isa employee;
  (team: $t, leader: $e) isa team_leadership;
fetch
  $e as leader: name;
  $t as team: team_name;
```



# Pattern-based querying with inheritance polymorphism

```
match
  $e isa employee;
  (team: $t, leader: $e) isa team_leadership;
fetch
  $e as leader: name;
  $t as team: team_name;
```

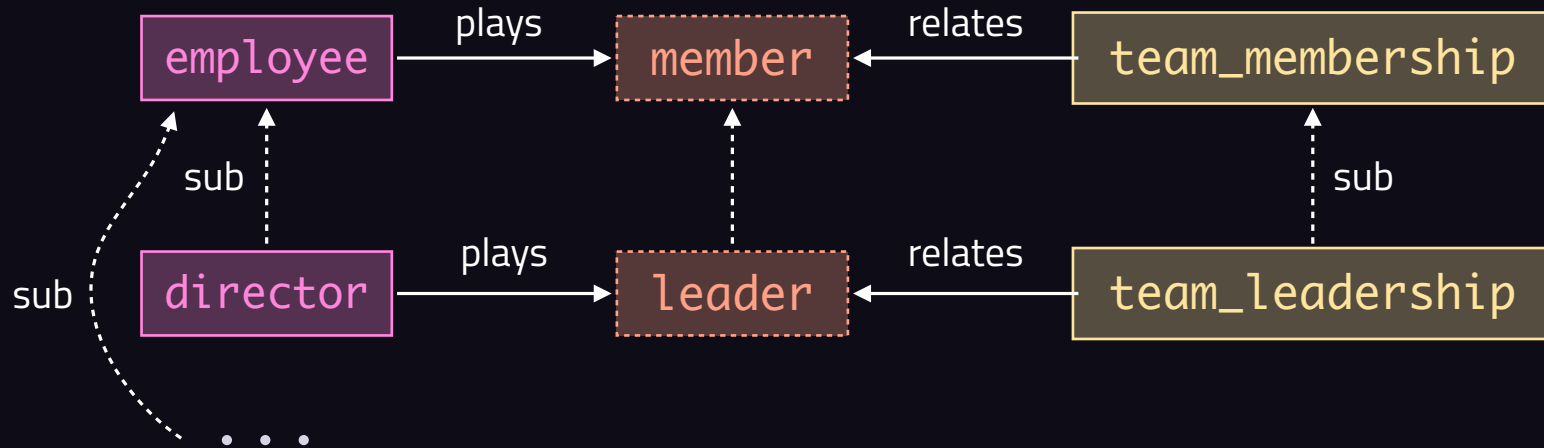
- `match` clause, line 1
  - States `$e` is a `employee` object
  - In particular, `$e` could be a `director` object!



# Pattern-based querying with inheritance polymorphism

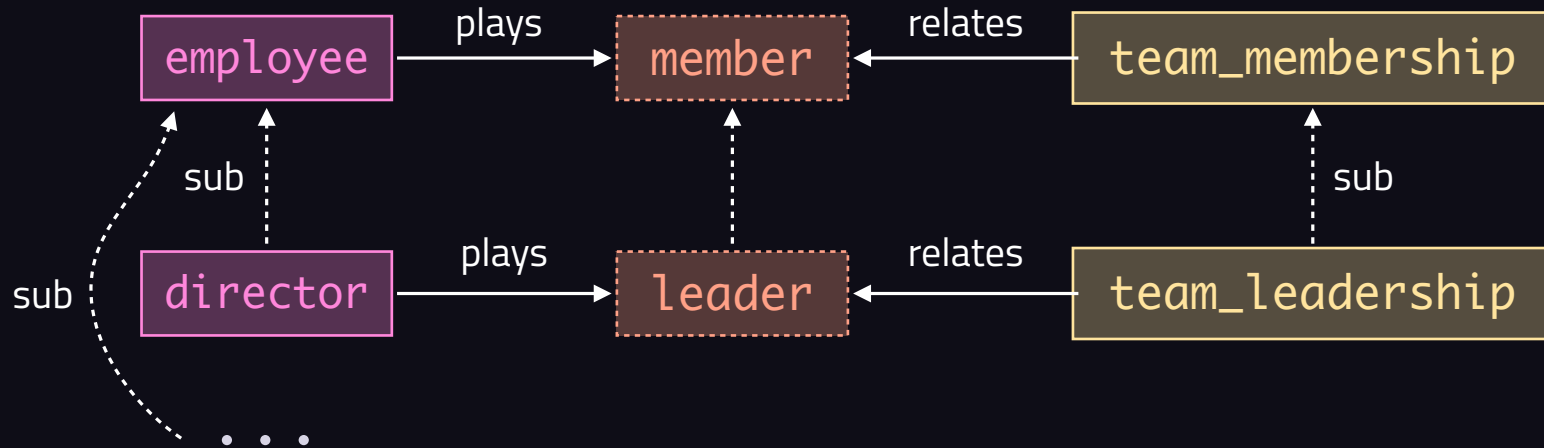
```
match
  $e isa employee;
  (team: $t, leader: $e) isa team_leadership;
fetch
  $e as leader: name;
  $t as team: team_name;
```

- `match` clause, line 1
  - States `$e` is a `employee` object
  - In particular, `$e` could be a `director` object!
- `match` clause, line 2
  - States a `team_leadership` exists with leader `$e`
  - Can infer `$e` is a `director`



# Pattern-based querying with inheritance polymorphism

```
match
  $e isa employee;
  (team: $t, leader: $e) isa team_leadership;
fetch
  $e as leader: name;
  $t as team: team_name;
```



- **match** clause, line 1
  - States **\$e** is an employee object
  - In particular, **\$e** could be a director object!
- **match** clause, line 2
  - States a **team\_leadership** exists with leader **\$e**
  - Can infer **\$e** is a director
- **fetch** clause:
  - The time we return tuples of employees **\$e** (with names) and teams **\$t** (with team\_names)

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

*When clause*

*contains a pattern stating  
the rule's assumptions*



# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

## *When clause*

*contains a pattern stating  
the rule's assumptions*

## *Then clause*

*statement of data to be derived*

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

## *When clause*

*contains a pattern stating  
the rule's assumptions*

## *Then clause*

*statement of data to be derived*

- **when** clause
  - States `$d` is a director object
  - States `$d` is a member in a team `$t`

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

## *When clause*

*contains a pattern stating  
the rule's assumptions*

## *Then clause*

*statement of data to be derived*

- **when** clause
  - States `$d` is a director object
  - States `$d` is a member in a team `$t`
- **then** clause
  - States `$d` is a leader of team `$t`

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

## When clause

contains a pattern stating  
the rule's assumptions

## Then clause

statement of data to be derived

- **when** clause
  - States `$d` is a director object
  - States `$d` is a member in a team `$t`
- **then** clause
  - States `$d` is a leader of team `$t`

In other words:

*"directors are automatically  
designated as leaders in all  
teams they are members in."*

# Reasoning in complex domains

```
define rule director-team-members-are-leader: when {  
  $d isa director;  
  (team: $t, member: $d) isa team_membership;  
} then {  
  (team: $t, leader: $d) isa team_leadership;  
};
```

## When clause

contains a pattern stating  
the rule's assumptions

## Then clause

statement of data to be derived

- **when** clause
  - States `$d` is a director object
  - States `$d` is a member in a team `$t`
- **then** clause
  - States `$d` is a leader of team `$t`
- *Note:* same pattern-based QL used throughout!

In other words:

*"directors are automatically  
designated as leaders in all  
teams they are members in."*

# Summary

# Summary

- **Patterns** in the PERA model:
  - express **combined statement** about type instances and dependencies
  - are fully **composable** and **declarative**
  - can be **type-checked!**

# Summary

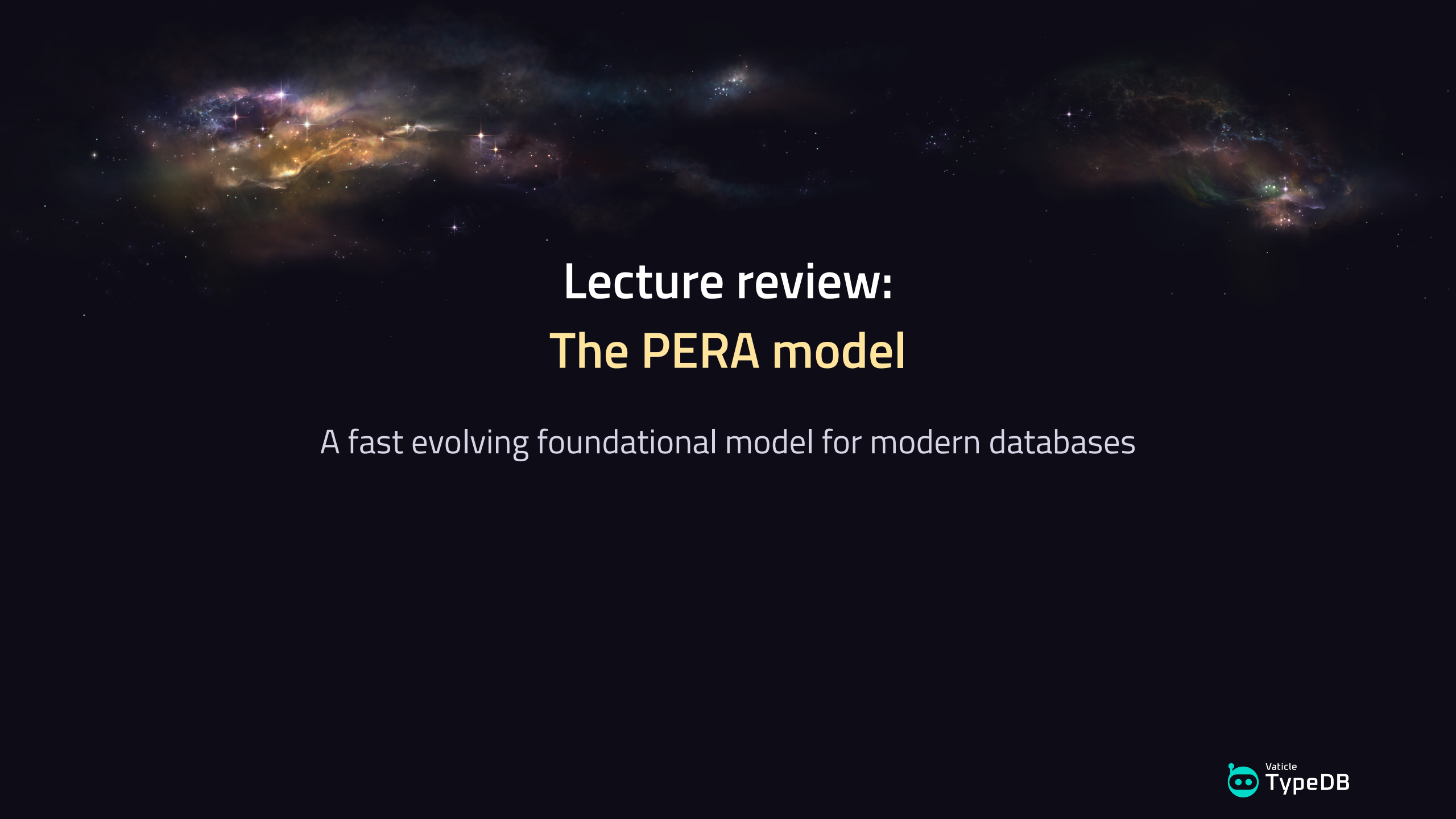
- **Patterns** in the PERA model:
  - express **combined statement** about type instances and dependencies
  - are fully **composable** and **declarative**
  - can be **type-checked!**
- Patterns underlie two *key functions* to the PERA model:
  - type-theoretic query language: **TypeQL**
  - rule-based **Reasoning Engine**



# Summary

- **Patterns** in the PERA model:
  - express **combined statement** about type instances and dependencies
  - are fully **composable** and **declarative**
  - can be **type-checked!**
- Patterns underlie two *key functions* to the PERA model:
  - type-theoretic query language: **TypeQL**
  - rule-based **Reasoning Engine**

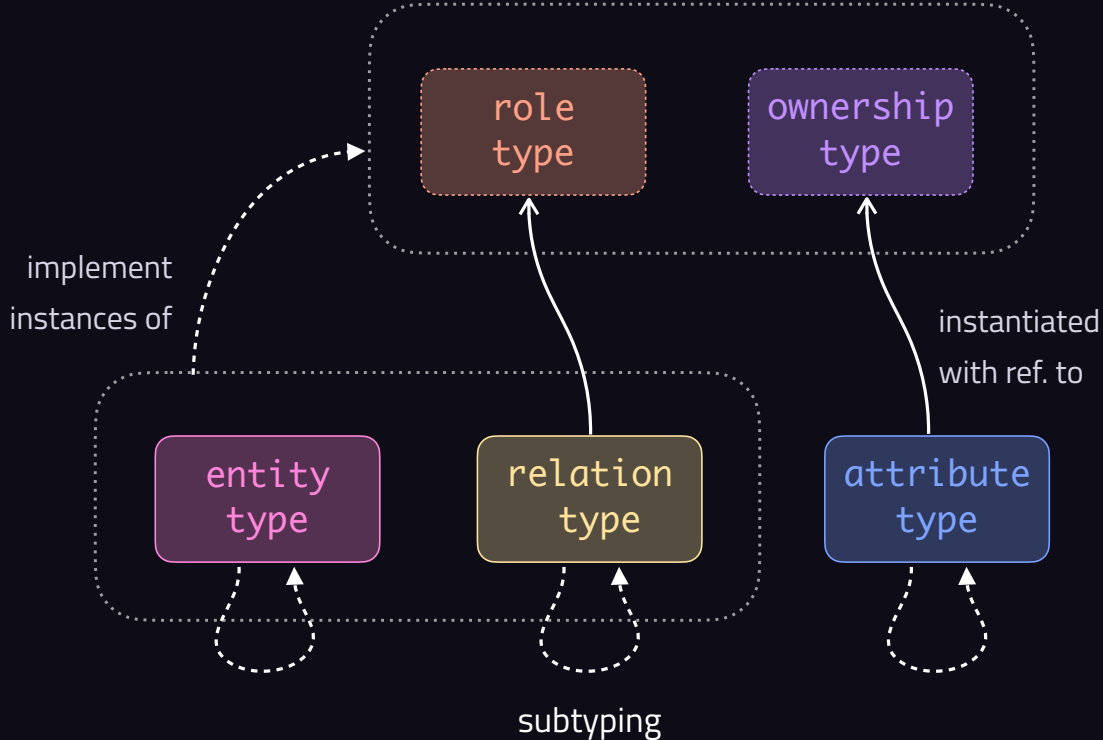
Stay tuned for more!



# Lecture review: **The PERA model**

A fast evolving foundational model for modern databases

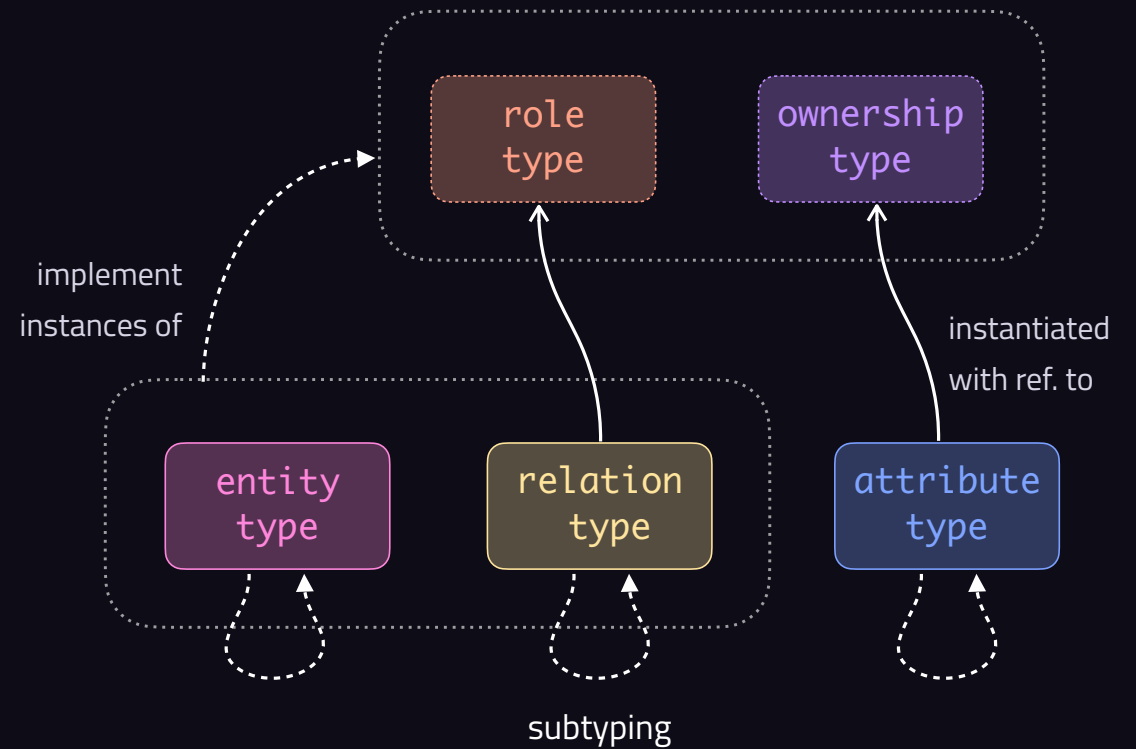
# Lecture #4 conclusion



	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

# Lecture #4 conclusion

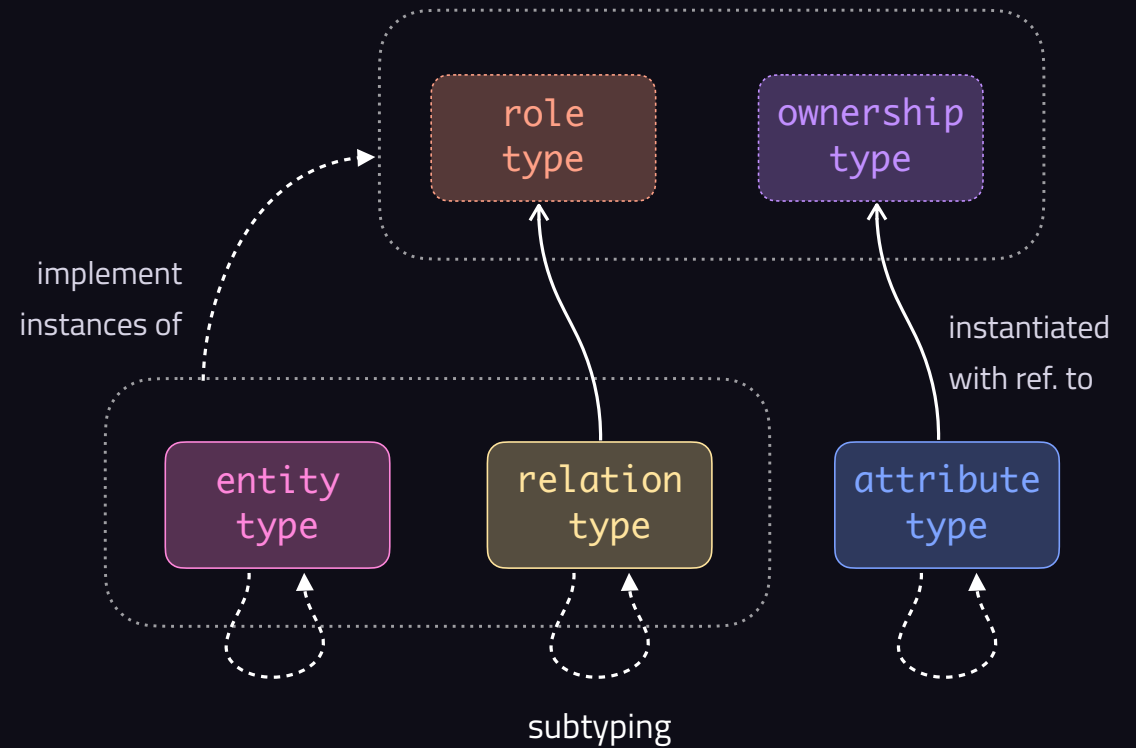
- We defined *concepts*, *instances*, and *concept dependencies* abstractly



	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

# Lecture #4 conclusion

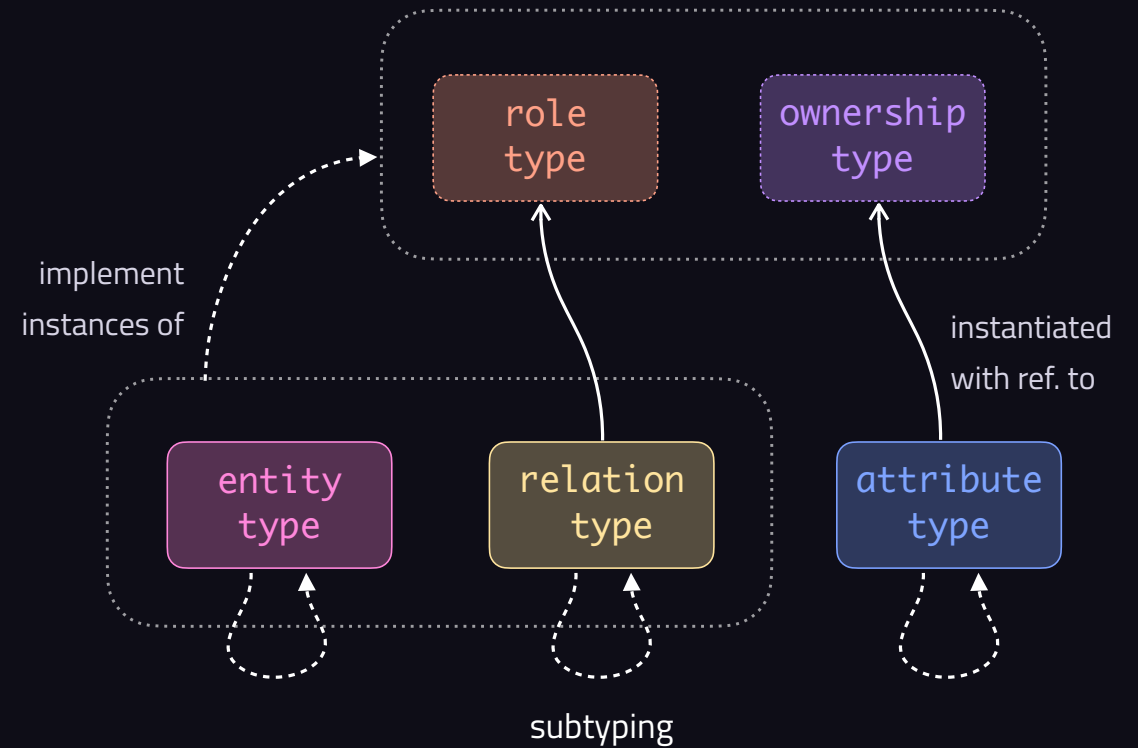
- We defined *concepts*, *instances*, and *concept dependencies* abstractly
- Used these simple conceptual principles to develop the **PERA model**



	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

# Lecture #4 conclusion

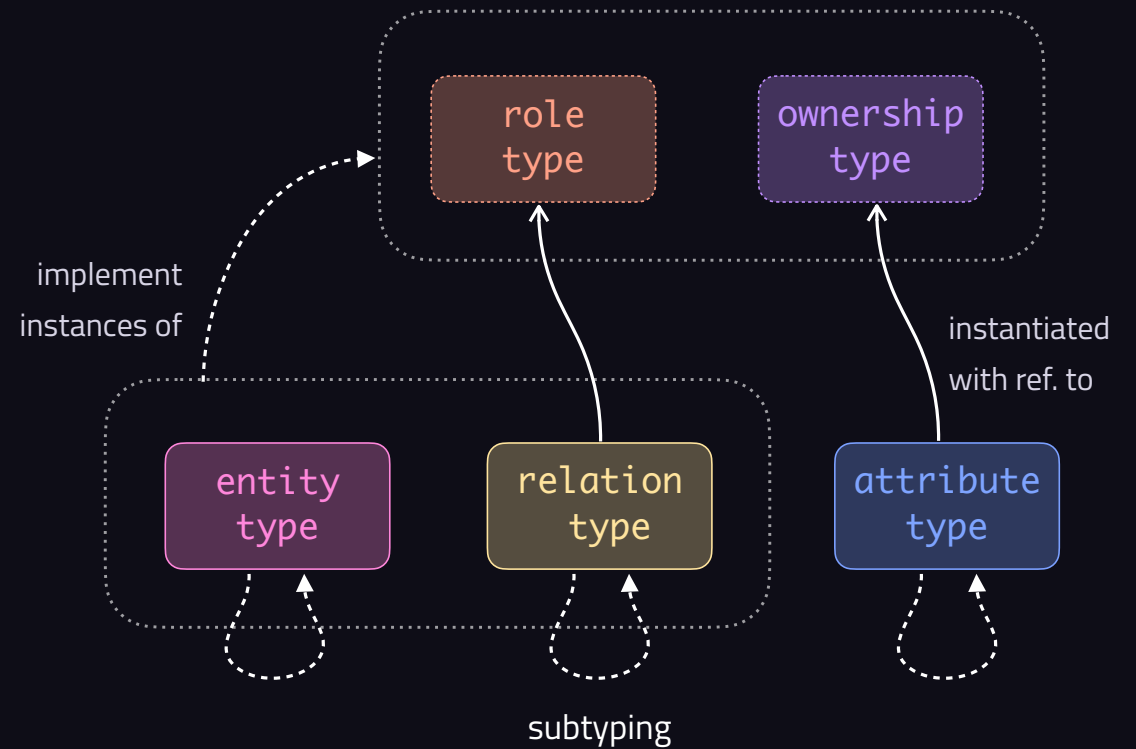
- We defined *concepts*, *instances*, and *concept dependencies* abstractly
- Used these simple conceptual principles to develop the **PERA model**
- Reverse-engineered concepts and dependencies of **existing data models**



	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

# Lecture #4 conclusion

- We defined *concepts*, *instances*, and *concept dependencies* abstractly
- Used these simple conceptual principles to develop the **PERA model**
- Reverse-engineered concepts and dependencies of **existing data models**

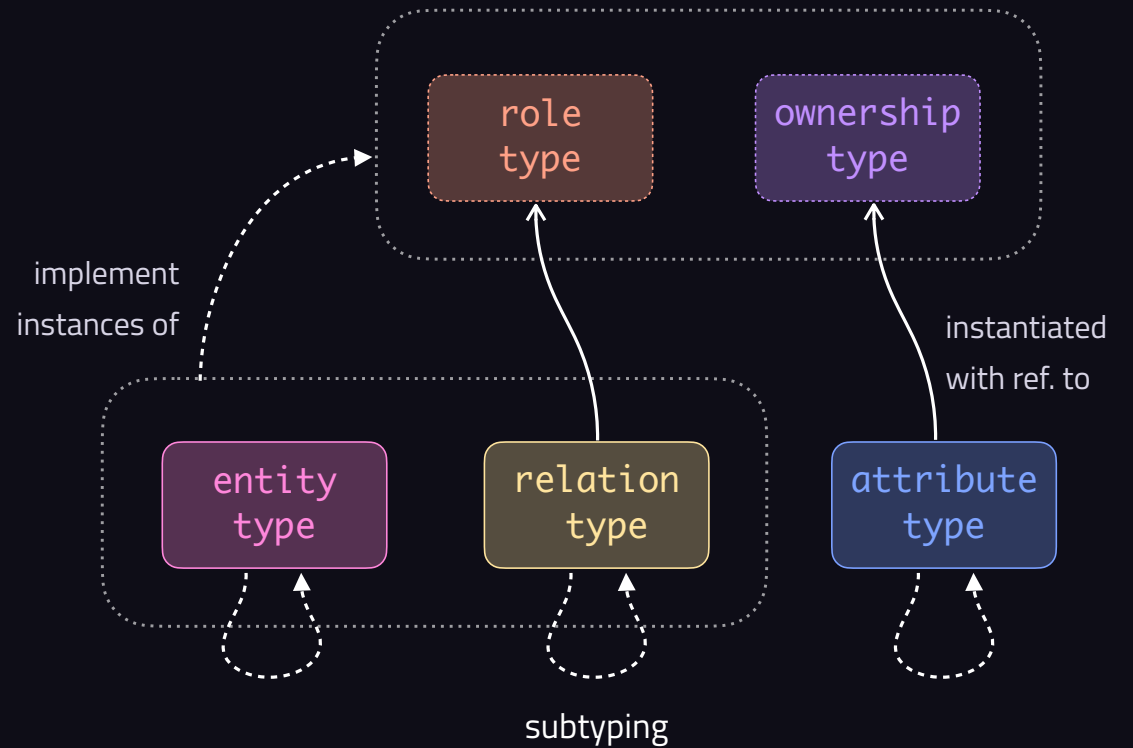


	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	(global constants)	<b>attribute types</b>

- In comparison, the PERA excels as a **principled**, **modular**, **polymorphic**, and *types-first model*

# Lecture #4 conclusion

- We defined *concepts*, *instances*, and *concept dependencies* abstractly
- Used these simple conceptual principles to develop the **PERA model**
- Reverse-engineered concepts and dependencies of **existing data models**



	independent type	dependent type
object types	<b>entity types</b>	<b>relation types</b>
attribute types	<i>(global constants)</i>	<b>attribute types</b>

- In comparison, the PERA excels as a **principled**, **modular**, **polymorphic**, and *types-first model*
- Integrates **type-theoretic querying** and **reasoning**



# Check out our **previous lectures**

*Replay now available*

TypeDB Lecture: TypeDB the polymorphic database

TypeDB Fundamentals Lecture Series

## TypeDB: the Polymorphic Database

Dr. James Whiteside  
Research Engineer, Vaticle  
Previously: Computational Solid-State Physicist @ University of Surrey

Watch on YouTube

TypeDB

*Replay now available*

TypeDB Lecture: Why We Need a Polymorphic Database

TypeDB Fundamentals Lecture Series

## Why We Need a Polymorphic Database

Dr. James Whiteside  
Research Engineer, Vaticle  
Previously: Computational Solid-State Physicist @ University of Surrey

Watch on YouTube

TypeDB

*Replay now available*

TypeDB Lecture: Type Theory as the Unifying Foundation for Modern Databases

TypeDB Fundamentals Lecture Series

## Type Theory as the Unifying Foundation for Modern Databases

Dr. Christoph Dorn  
Head of Research, Vaticle  
Previously: Theoretical Computer Scientist in Category Theory @ Oxford University

Watch on YouTube

TypeDB

Watch at [TypeDB.com/lectures](https://typedb.com/lectures)



Vaticle

TypeDB

Thank you!



Q & A

# More TypeDB Resources



TypeDB Learning Center  
[typedb.com/learn](https://typedb.com/learn)



Download TypeDB  
[typedb.com/deploy](https://typedb.com/deploy)



TypeDB Cloud  
[cloud.typedb.com](https://cloud.typedb.com)



Vaticle

TypeDB

Join us at [typedb.com/discord](https://typedb.com/discord)