# LionWeb Introduction for C# Developers

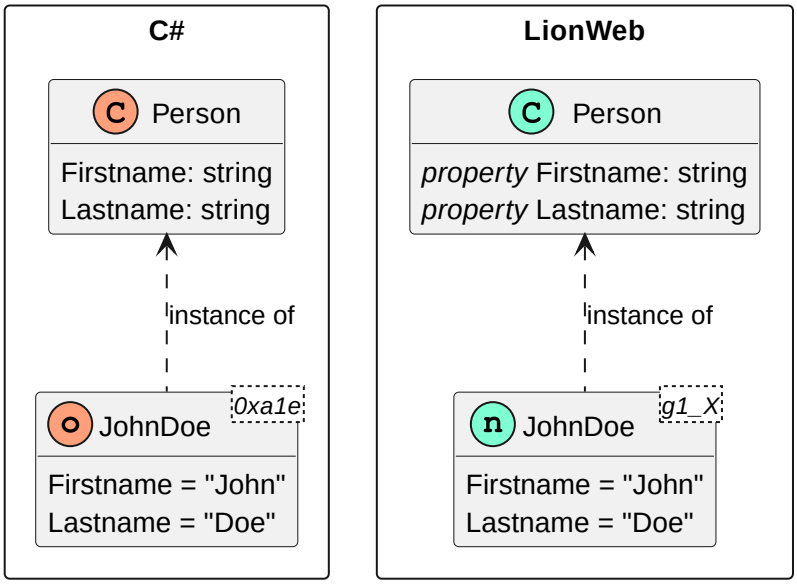
## Table of Contents



# Everything is an Object / Node

In object-oriented languages like C#, everything is an *object*. Objects have an *identity*, *state*, and a *type* — the object's *class*. The object's identity is its memory location, the object's state is the value of all its *members*. Relevant members are *fields* and *C# properties*[1]

Similarly, everything in LionWeb is a *node*. Nodes have an *id*, state, and a *classifier* — the node's *concept*. The id is a string comprised of uppercase or lowercase `A` to `Z`, `1` to `9`, `-` (dash) and `_` (underscore). A node id has no "meaning" and must be unique, similarly to the memory location of an object. A node's state is the value of all its *features*. Possible features are *LionWeb properties*[1], *containments*, and *LionWeb references*[1].

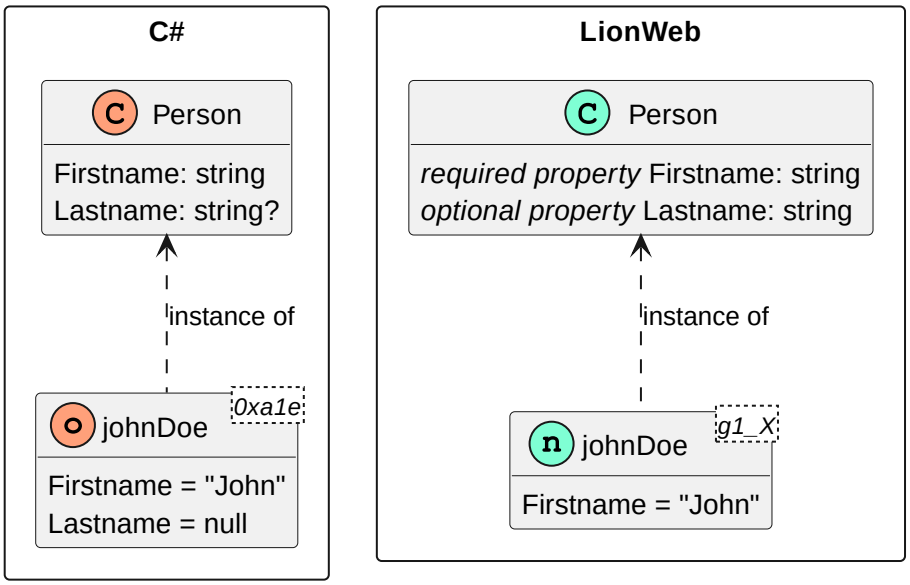


| C# | LionWeb |
|---|---|
| object | node |
| object identity | node id |
| object type | node classifier |
| class | concept |
| class member | concept feature |
| field<br>C# property | LionWeb property<br>containment<br>LionWeb reference |

# … except Value Types / LionWeb Properties

C# has *value types* that are not objects. Examples include `int` and `bool`. They don't have any identity, they are purely defined by their value. We cannot tell apart `15` from `15`, but we can tell apart the two objects `var a = new Person { Firstname="John", Lastname="Doe" };` and `var b = new Person { Firstname="John", Lastname="Doe" };`. Value types cannot be `null`, unless we declare them *nullable*: `int age = 23; int? numberOfAunts = null;`. In the example, we always know the person's age, but we might not know how many aunts they have.

LionWeb properties have value type semantics[2] — they are purely defined by their value. LionWeb properties can have type `integer`, `boolean`, enumeration, and `string`. LionWeb properties are either *required* or *optional*. If required, the LionWeb property must have a value, otherwise we can omit the value.




# Mapping to C#

LionWeb properties become C# properties with proper getters and setters. In C# we also have a method like `Person SetFirstname(string value)`. They form a fluent interface.

Optional LionWeb properties have nullable types in C#. They may return `null`, and can be set to `null`.

Required LionWeb properties have non-nullable types in C#. They may never return `null`. If no value has ever been assigned to this C# property, the C# property getter throws an `UnsetFeatureException`. If set to `null`, the C# property setter, and the `SetProperty()` method, throw an `InvalidValueException`.

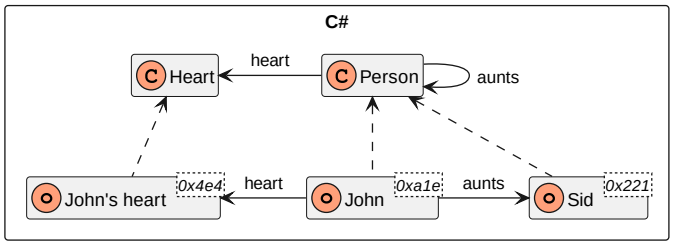
```
class Person {
  ...
  public string Firstname { get; set; }
  public Person SetFirstname(string value);

  public string? Lastname { get; set; }
  public Person SetLastname(string? value);
}

...

Person johnDoe = new Person("g1_X");

johnDoe.Firstname;           // throws UnsetFeatureException

johnDoe
  .SetFirstname("John")
  .SetLastname("Doe");

johnDoe.Firstname = null;    // throws InvalidValueException
johnDoe.SetFirstname(null);  // throws InvalidValueException
```

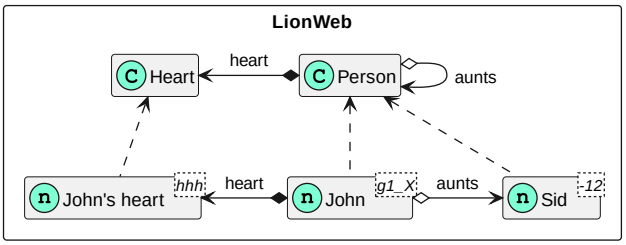
| C# | LionWeb |
|---|---|
| `int` type | `integer` property type |
| `bool` type | `boolean` property type |
| `string` type | `string` property type |
| enum | enumeration |
| nullable type | optional feature |
| non-nullable type | required feature |

# Object / Node Members

C# always uses *C# references* to connect objects. A `Person` object uses C# references to connect to both the person's `Heart` object and their aunts' objects. We cannot "delete" an object, but we can cut all C# references to it — eventually the garbage collector will delete the object once all C# references to it are unset.

In LionWeb we have two different ways to connect nodes. A `Person` node *contains* its `Heart` node, but uses a *LionWeb reference* to connect to the persons' aunts' nodes. Both *containment* and *LionWeb reference* are a *link*.

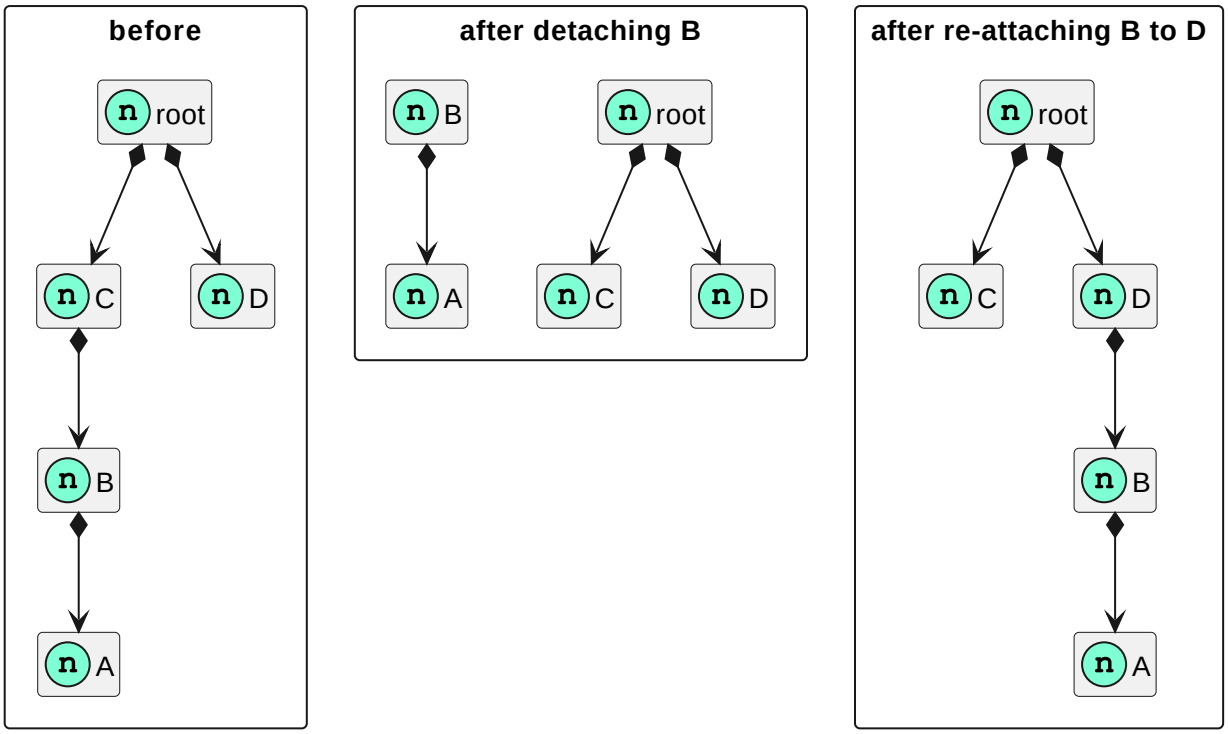



| C# | LionWeb |
|---|---|
| C# reference | Link<br>Containment<br>LionWeb Reference |

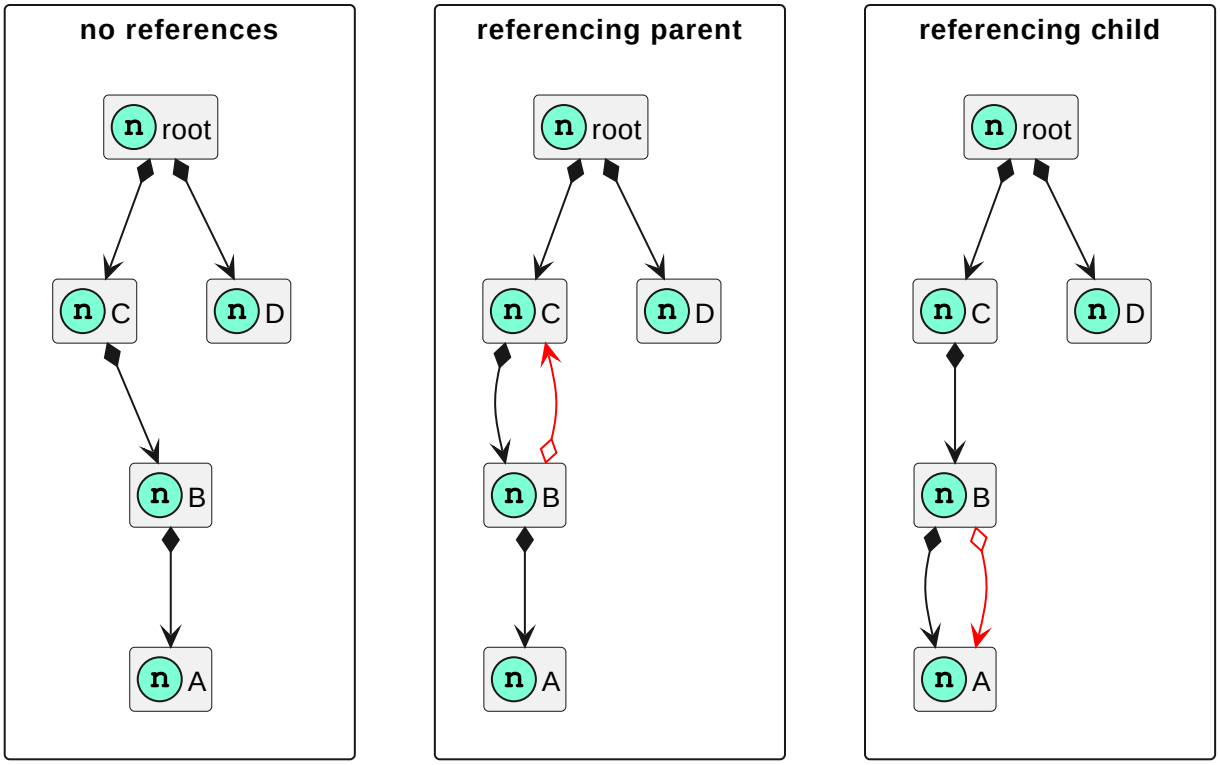
# Containments

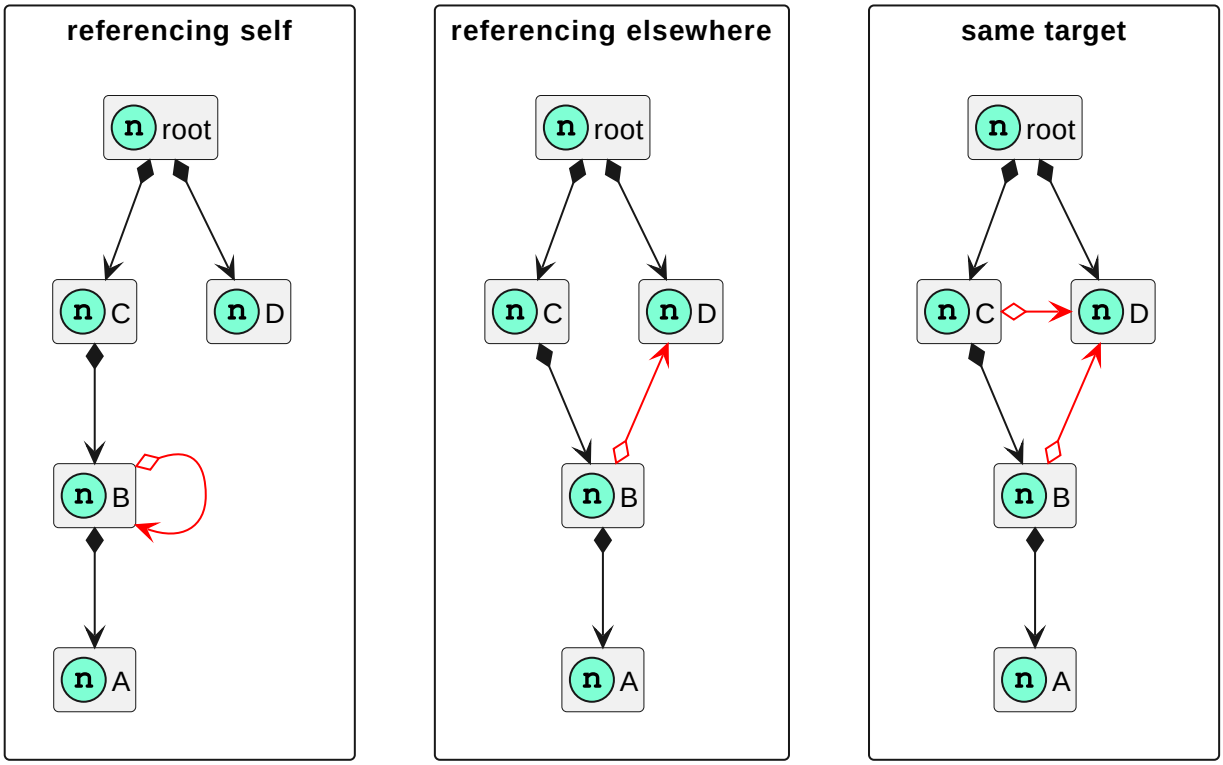
Every node `A` is contained in exactly one other node `B`. `B` is contained in `C`, and so on, until we arrive at the *root node*. The root node is the only node that is not contained anywhere. Thus, all nodes form a tree. We usually look at the tree from the top: *root node* contains `C`, which contains `B`, which contains `A`. Each node has one *parent* and zero or more *children*. We cannot "delete" a node, but we can *detach* the node from its parent. If we detach node `B` from its parent `C`, both `B` and its child `A` are *orphaned* — unless we re-attach them to `D`. If not, these nodes stay orphans, and the garbage collector claims them eventually.



# LionWeb References

LionWeb references behave very similar to C# references. We can refer to any other node, no matter where any of the two belongs to. Removing a LionWeb reference also does not affect anything besides that reference. LionWeb references turn the strict containment tree into a graph with interconnections.

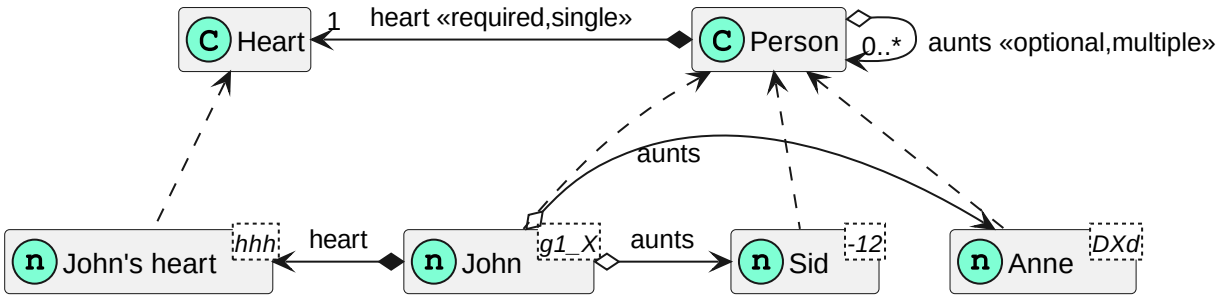





# Link Cardinality

Links (i.e. containments and LionWeb references) are either *required* or *optional*, just as LionWeb properties. Links are also either *singular* or *multiple*, i.e. the link can point to one or several other nodes. This maps nicely to cardinalities, as known from UML:

| | **singular** | **multiple** |
|---|---|---|
| **optional** | 0..1 | 0..* |
| **required** | 1..1 | 1..* |




# Mapping to C#

*Singular Links*

LionWeb singular links become C# properties with proper getters and setters. Singular links have a method like `Person SetHeart(Heart value)` in C#. They form a fluent interface.

Their method parameter type and C# property type for a singular link is nullable for optional links. They may return `null`, and can be set to `null`.

Required singular links have non-nullable types in C#. They may never return `null`. If no value has ever been assigned to this C# property, the C# property getter throws an `UnsetFeatureException`. If set to `null`, the C# property setter, and the `SetLink()` method, throw an `InvalidValueException`.

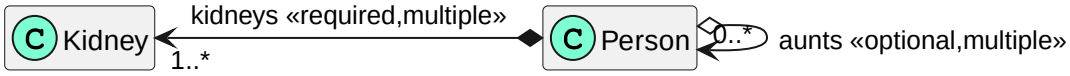

```
class Person {
  ...
  public Heart OwnHeart { get; set; }
  public Person SetOwnHeart(Heart value);

  public Person? BestFriend { get; set; }
  public Person SetBestFriend(Person? value);
}

...

Person john = new Person("g1_X") { OwnHeart = new Heart("hhh") };
Person sid = new Person("-12");

john.SetBestFriend(sid);
var friend = john.BestFriend;

john.OwnHeart = null;  // throws InvalidValueException

sid.OwnHeart;          // throws UnsetFeatureException
```

*Multiple Links*

LionWeb multiple links become C# properties with only getters. They always return `IReadOnlyList<LinkType>`, never `null`. The resulting enumerable cannot be modified — it doesn't even offer appropriate methods. Instead of setters or direct manipulation of the result, we have several methods for each multiple link: `AddLink(IEnumerable<LinkType>)`, `InsertLink(int index, IEnumerable<LinkType>)`, `RemoveLink(IEnumerable<LinkType>)`.

Optional multiple links may return empty `IReadOnlyList`, and all existing elements can be removed.

Required multiple links never return `null` or an empty list. If the list is empty, the C# property getter throws an `UnsetFeatureException`. Trying to remove all entries from a required multiple link throws an `InvalidValueException`.



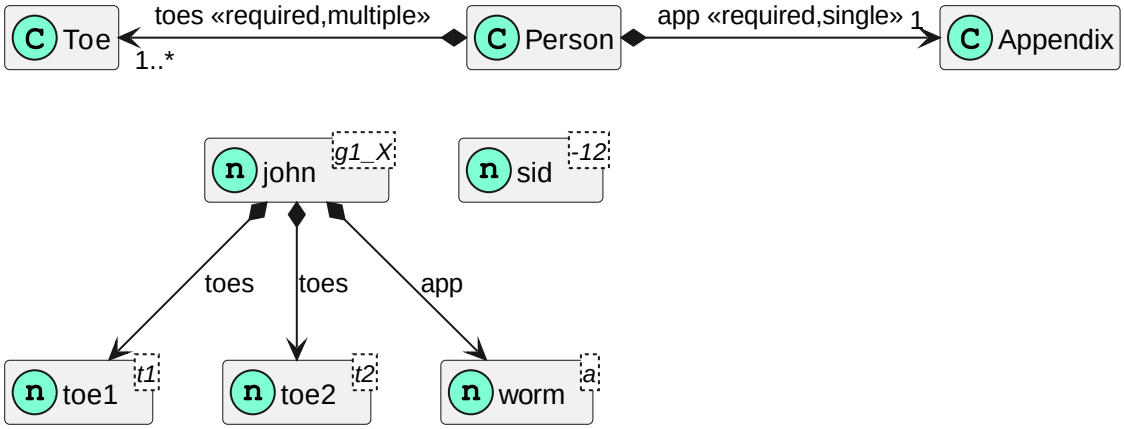

```
class Person {
```

```
  ...
  public IReadOnlyList<Kidney> Kidneys { get; }
  public Person AddKidneys(IEnumerable<Kidney> nodes);
  public Person InsertKidneys(int index, IEnumerable<Kidney> nodes);
  public Person RemoveKidneys(IEnumerable<Kidney> nodes);

  public IReadOnlyList<Person> Aunts { get; }
  public Person AddAunts(IEnumerable<Person> nodes);
  public Person InsertAunts(int index, IEnumerable<Person> nodes);
  public Person RemoveAunts(IEnumerable<Person> nodes);
}

...

Person john = new Person("g1_X") { Kidneys = [new Kidney("s3S")] };
Person sid = new Person("-12");

john.AddAunts([sid]);
var onlyKidney = john.Kidneys.First();

john.Kidneys = [];                 // compilation error
john.Kidneys.Remove(onlyKidney);   // compilation error

john.RemoveKidneys([onlyKidney]);  // throws InvalidValueException

sid.Kidneys;                       // throws UnsetFeatureException
```

# Maintain Node Tree first, then Required Flag

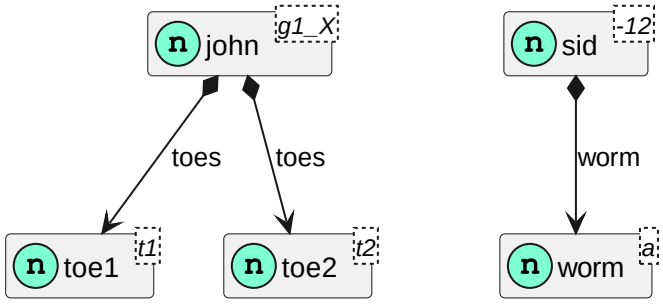
The LionWeb C# framework always keep the nodes in a tree. This means that every node has zero or one parents, and this parent contains the node. As a consequence, simple assignments of containments can have side effects.

**NOTE** This only concerns *containments*. We can freely assign *LionWeb references* without side effects.



```
class Person {
  ...
  public IReadOnlyList<Toe> Toes { get; }
  public Person AddToes(IEnumerable<Toe> nodes);
  public Person InsertToes(int index, IEnumerable<Toe> nodes);
  public Person RemoveToes(IEnumerable<Toe> nodes);

  public Appendix App { get; set; }
  public Person SetApp(Appendix value);
}

...

Toe toe1 = new Toe("t1");
Toe toe2 = new Toe("t2");
Appendix worm = new Appendix("a");
Person john = new Person("g1_X") { Toes = [toe1, toe2], App = worm };
Person sid = new Person("-12");

sid.App = john.App  ;               ①
john.App;                           // throws UnsetFeatureException

Toe firstToe = john.Toes.First();   ②
Toe lastToe = john.Toes.Last();

sid.AddToes([firstToe, lastToe]);   ③
joh.Toes;                           // throws UnsetFeatureException
```

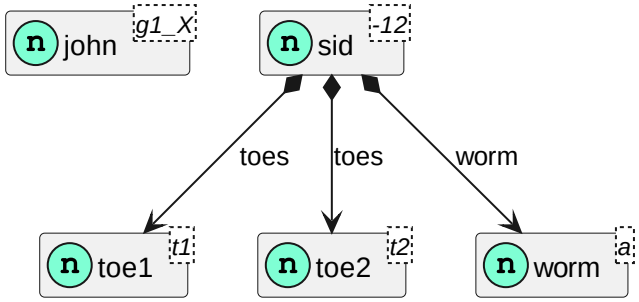
① We assign John's App to Sid. For C#, that's just a C# reference — several places can refer to the same C# object. But for LionWeb, that's a containment, and we *must not* have two parents for `worm`! Thus, we detach `worm` from John, and attach it to Sid:




Now, John's App is `null`, even though it's *required*. Consequently, we'd get a `UnsetFeatureException` in the next line if we tried to get John's App.

② We get John's first Toe, i.e. `toe1`. That's ok, as we only store it in a local variable — no effect on the tree.

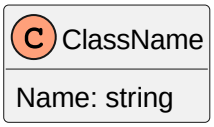
③ We add several of John's Toes to Sid. Again, we *must not* have two parents for the same toe, so we detach them from John, and attach them to Sid:



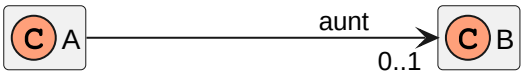

Now, John's Toes are empty, even though the link is *required*. Consequently, we'd get a `UnsetFeatureException` in the next line if we tried to get John's Toes.

To summarize, the LionWeb framework always keeps the tree, even if it has to violate *required* constraints. It helps the developer to adhere to required flags by throwing specializations of `LionWebExceptionBase` on direct attempts to violate the constraints.

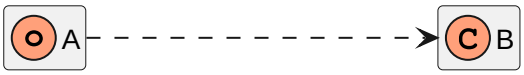
# Appendix A: Diagram Legend



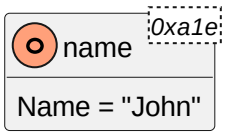

C# class
with C# property "Name" of type string



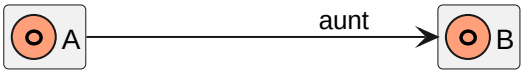

C# class A declares C# Reference
"aunt" of type B with cardinality 0..1



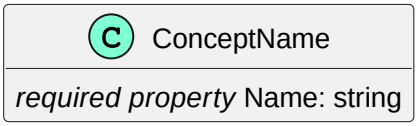

C# object A is instance of C# class B



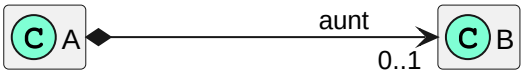

C# object at memory location 0xa1e
with "Name" C# property set to "John"



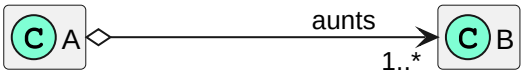

C# object A's C# Property "aunt"
references object B



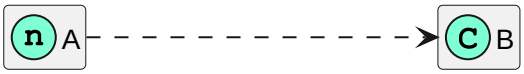

LionWeb concept
with required LionWeb property
"Name" of type string



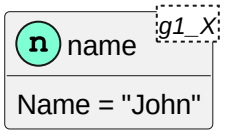

LionWeb concept A declares
singlar, optional containment
"aunt" to concept B



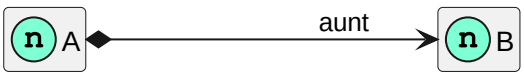

LionWeb concept A declares
multiple, required LionWeb reference
"aunts" to concept B



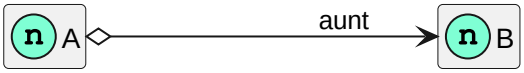

node A is instance of concept B




LionWeb node with node id g1_X
with "Name" LionWeb property
set to "John"




LionWeb node A contains node B
in containment "aunt"




LionWeb node A references node B
in containment "aunt"

[1] We use the term *property* both in C# context and LionWeb context. Thus, we'll always qualify it with C# or LionWeb prefix. The same applies to the term *reference*.

[2] Value type *semantics* because C# `string` is not a value type.