|  |
| --- |
| Circle Construct Drafts |

## Relationships Construct Drafts

### Relationships Between Classes

The\* class-relationship structure is the\* bone structure of a program.

### Bidirectional Relationships

The notation suggested here was moved away from the Circle Language Spec in favor of another notation.

Starting with:



A suggested notation would be for the two class lines to merge together to form the picture below:



This, however, might be a quite ambiguous notation. It would suggest that the two symbols joined by the line would have the same class. But that would not be what is intended. The circle inside **Class A** would have **Class B** and the circle inside **Class B** would have **Class A**.

A solution to this ambiguity may be proposed.

Fortunately, the\* notation can\* be disambiguated using the\* rules of automatic containment. Automatic containment is explained in the\* article *Automatic Containment*. Before explaining how automatic containment leads to the\* eventual notation, here is the\* disambiguated notation of a relationship between two classes:



The\* notation is accomplished by first taking the\* original picture with one class refering to another and the\* other refering back to the\* first class:



Then\*, an imaginary reference to each class is added to the\* diagram



Next, the\* class lines are merged, but\* also the\* class symbols are merged:



The\* notation would\* still be ambiguous, if\* it weren’t for the\* double dashed line of the\* merged class symbols. So a double dashed circle symbolizes a relationship between classes.

Examples with different multiplicities:







A symbol merge in a relationship that has nonagons in it also results in a double dashed circle, because\* the\* imaginary reference to the\* classes, that are put on a higher level, are represented by circles, not\* a nonagons.

There is an idea where one related list might contain items from multiple classes. **Class A** would have a **n 🡪 1** relationship to items of **Class B** and **Class C**, which the picture below aims to express:



#### Counterpart out of Sight

If the counterpart of a relationship would be out of sight, a line might point out of the diagram. A catch there might be that you might not see whether the relationship counterpart would have multiplicity of **1** or **n**. A possible solution for this, might be to express multiplicity at the end of that line that might point out of the diagram.

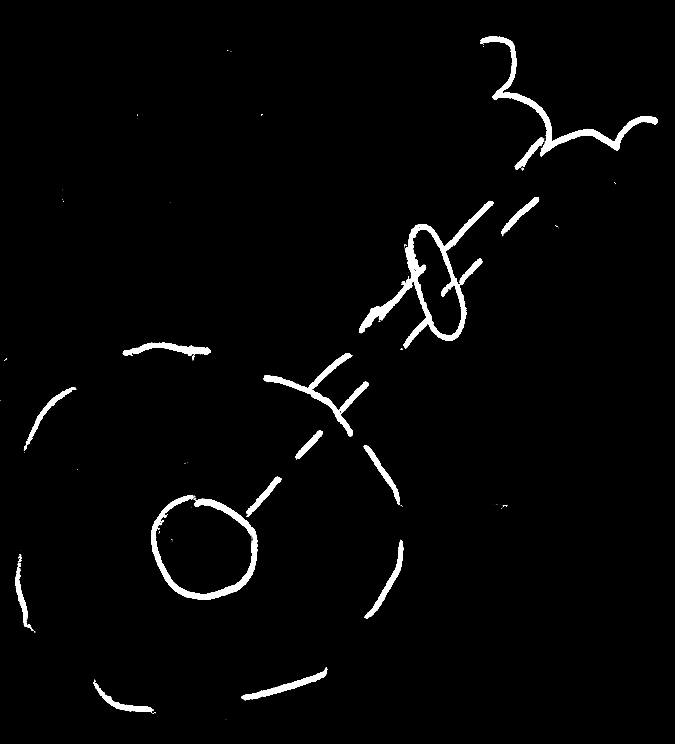
This might look as follows with the double dashed border notation:

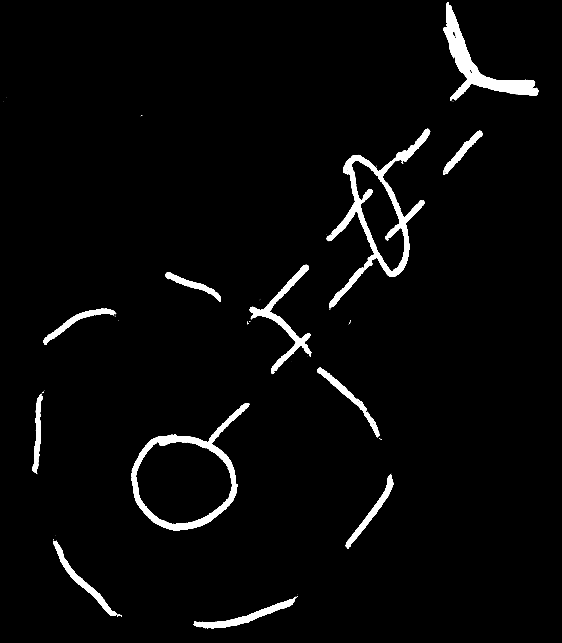




But then again, if something is out of sight, it might just be out of sight and you cannot see things out of sight. Perhaps there is no problem here.

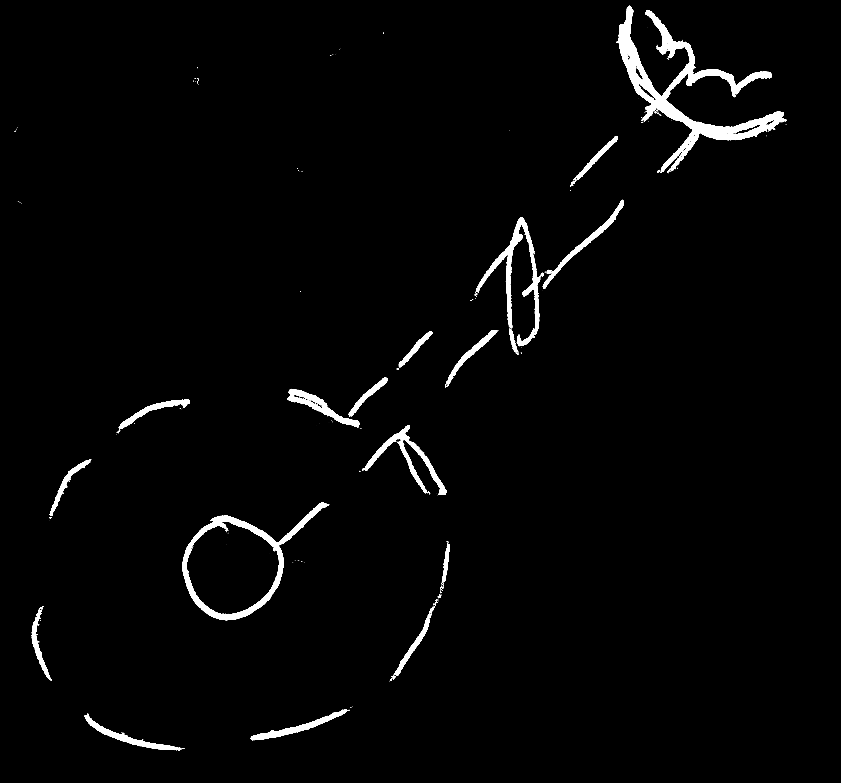
Here are examples of what it could look like in case of relational ring notation, with explicit relationship counterparts:

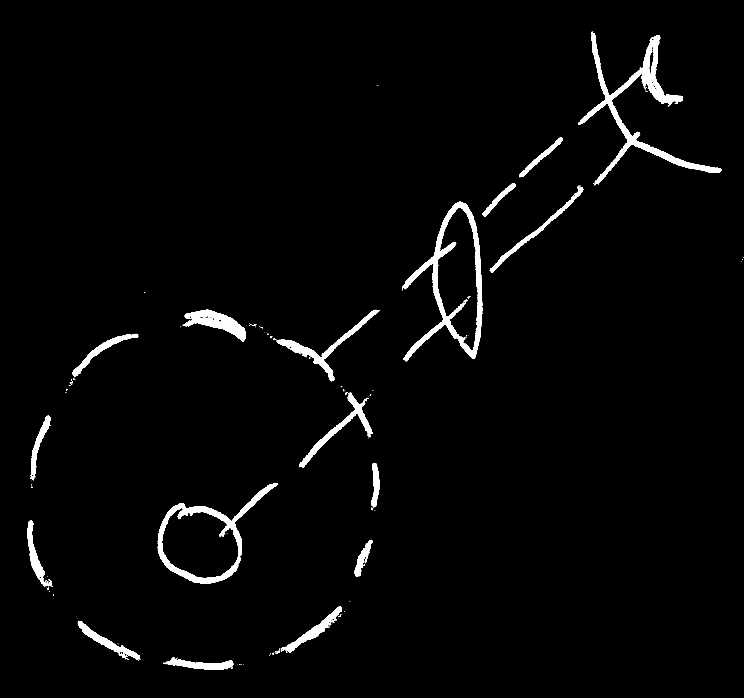




That proposal might be a problem, because it seems to clash with a proposed notation for optional. And also it seems to not reflect the containment structure: there would be a container in between the half shape and the other parts of the diagram. Seems unfortunate.

In another proposal:

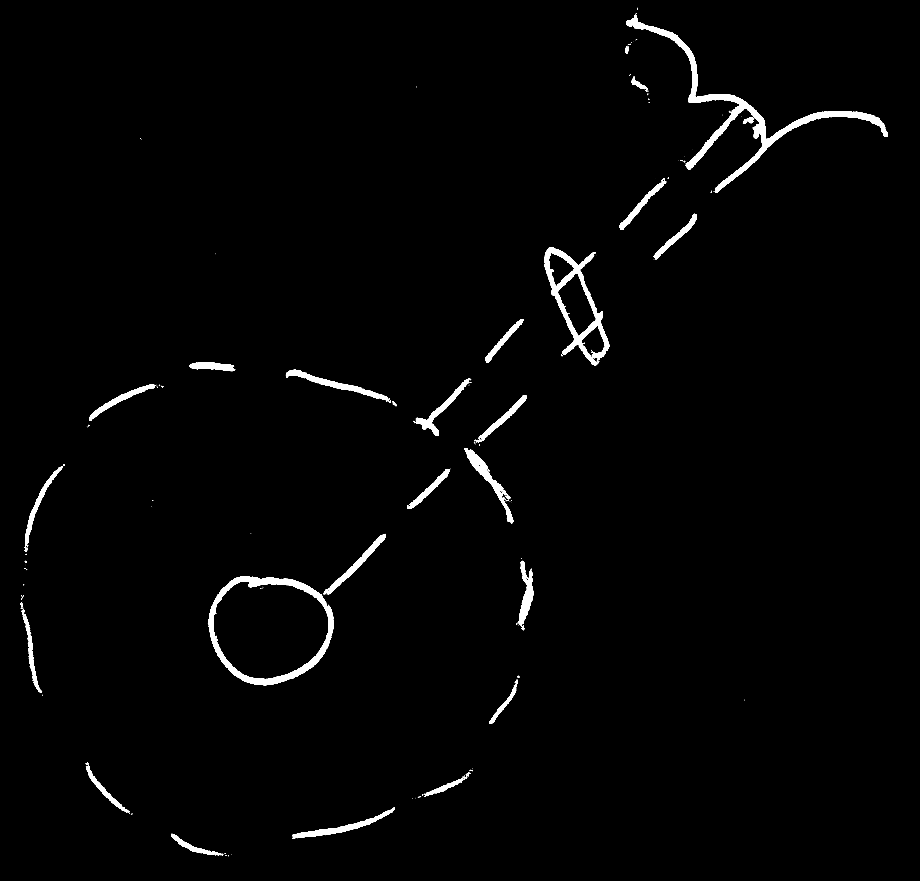




It may be going a bit far. It seems to draw out part of the diagram, that is out of sight. But then it might actually have a container in between, that is not drawn.

The argument "What is out of sight, is out of sight." might be good enough to not solve this 'problem'. A proposal might be that might not be a real problem.

Here a variation that might also be dubious:



In that image both relationship counterparts' class lines are connected to the same half shape. It might go a bit far in suggesting something it's not.

#### No Reuse of Merged Imaginary References

If\* two imaginary references have merged, to become a relationship symbol, then\* other references to the\* same classes won’t connect to an imaginary reference that has merged to become a relationship symbol. Relationships create their own imaginary references, that aren’t reused. This is displayed in the\* article *Relationships Between Objects in a Diagram*, but\* may also apply to the\* notation of relationships between classes.



#### Example



### Relations Between Objects

#### Diagram Notation

The\* relationship symbol is a double circle. The\* reason behind this notation, was already explained in the\* article *Relationships in a Diagram*. The\* notation is accomplished by first taking the\* original picture with one class refering to another and the\* other refering back to the\* first class:



Then\*, an imaginary reference to each class is added to the\* diagram



Next, the\* class lines are merged, but\* also the\* class symbols are merged:



The\* notation would\* still be ambiguous, if\* it weren’t for the\* double line of the\* merged object symbols. So a double circle symbolizes a relationship between objects.

For relationships between classes the\* relationship symbol is a double *dashed* circle. For relationships between objects, the\* relationship symbol is a double circle drawn with *solid* lines.

#### Counterpart out of sight

When\* the\* counterpart of the\* relationship is out of sight, a line should point out of the\* diagram. A catch there is, that you\* can’t see if\* the\* relationship counterpart is part of a multiplicity of **n** or not\*. Therefore, the\* multiplicity is expressed at the\* end of th line pointing out of the\* diagram as follows:



#### No reuse of merged imaginary references

If\* two imaginary references have merged, to become a relationship symbol, then\* other references to the\* same objects won’t connect to an imaginary reference that has merged to become a relationship symbol. Relationships create their own imaginary references, that aren’t reused.

Here is a relationship between two objects:



The\* two objects refer to eachother. This originally consisted of two distinct references:



Imaginary references were put on one level higher:



If\* other references to the\* same objects were also displayed in the\* diagram, then\* they would\* connect to the\* same imaginary references, put on a higher level:



When\* you\* merge the\* imaginary references to display that two references are part of a single relationship, you\* will not\* connect all

references to the\* merged imaginary reference:



You\* will keep separate imaginary references for the\* other unidirectional relationships to the\* objects:



### Bidirectional Relationship Synchronization Implementation Details

When\* you\* let one of the\* object references link to another object, then\* the\* link from the\* old counterpart to the\* object should be broken and a link from the\* new counterpart to the\* object should be established. The\* object synchronizes the\* old counterpart *out*, and it synchronizes the\* new counterpart *in*.

In the\* new computer language you\* are still able to create just a unidirectional relationship. In a programming environment you\* should be able to automatically turn a unidirectional relationship to a bidirectional relationship. In a programming environment you\* should be able to turn two already defined unidirectional relationships into a single bidirectional relationship.

So whenever a **Jar** changes its **Lid**, the\* original **Lid**’s reference to the\* **Jar** is annulled, and the\* new **Lid** get a reference to its new **Jar**. This actually explains synchronization between **1 => 1** related objects. Synchronization happens when\* assigning a related item. When\* assigning a related item, the\* related item gets a reference back to the\* first item.

Relationship synchronization happens when\* you\* assign a related object. When\* you\* assign **Lid . Jar**, then\* synchronizing the\* relationship, consists of, in turn, assigning **Jar . Lid**.

For every relationship type it works in a different way.

There are three relationship types:

**1 <=> 1**

**1 <=> n**

**n <=> n**

But\* synchronization is managed separately for each end of the\* relationship, so synchronization is managed in the\* following four of ways:

**1 => 1 synchronization**

**1 => n synchronization**

**n => 1 synchronization**

**n => n synchronization**

#### Synchronization Types

There are four synchronization types:

**1 => 1 synchronization**

**1 => n synchronization**

**n => 1 synchronization**

**n => n synchronization**

Every synchronization type follows a slightly different procedure, to make sure that on assignment of one relationship counterpart, the\* other relationship counterpart goes along with it.

#### 1 => 1 Synchronization

**1 => 1** synchronization is quite easy. In a **Jar =>** **Lid** relationship, when\* assigning **Lid . Jar**, The\* old **Jar . Lid** is set to **Nothing**, while the\* new **Jar . Lid** is set to **This**.

#### Risk of infinite loop 1 => 1

When\* a relationship is synchronized, you\* may have a risk to an infinite loop.

When\* you\* assign **Jar** to **Lid**, then\* **Lid** is assigned to **Jar**, upon which **Jar** is assigned to **Lid** again, and so on. Fortunately, when\* a **Jar** is assigned a **Lid** it already has, the\* whole assignment is not\* executed. So it only goes as far as: **Jar** is assigned to **Lid**, upon which **Lid** is assigned to **Jar** again, upon which **Jar** is assigned to **Lid** again, but\* **Jar** already had that **Lid**, so that assignment is never executed.

#### 1 => n Synchronization

In a **1 => n** relationship between **Parents** and **Children**, every time you\* assign a **Child** to a **Parent**, **Child . Parent** is overwritten. The\* old **Child** is assigned **Nothing** as the\* **Parent**, and the\* new **Child** is assigned its new **Parent**.

Also, the\* original **Child**’s **ID In Parent** is yielded over to the\* new **Child. The\*** original **Child**’s **ID In Parent** is reset.

There used to be a misunderstanding, that one **Parent** could\* reference the\* same **Child** multiple times. But\* that idea was abolished, because\* when\* a **Parent** relates to a **Child** twice, the\* **Child** has to relate back to the\* **Parent** twice. A **Child** can\* only have one **Parent**, so it can\* never relate back to the\* same **Parent** twice. Something like that would\* require an **n => n** relationship, for the\* **Child** to be able to hold multiple references to the\* same **Parent**.

#### Risk of infinite loop 1 => n

Infinite loops for **n => 1** synchronization the\* are prevented the\* same way as for **1 => 1** synchronization. When\* you\* assign a **Parent** to a **Child**, The\* **Child** is added to the\* **Parent**, upon which the\* **Parent** is again assigned to the\* **Child**. But\* the\* **Child** already had that **Parent**, so the\* assignment is never executed. So that prevents an infinite loop there.

#### n => 1 Synchronization

In a **Child n => 1 Parent** relationship, when\* you\* change **Child . Parent**, the\* **Child** is removed from its original **Parent** and added to the\* new **Parent**. So you\* can\* never have the\* same **Child** in several **Parents**.

A **Child** can\* not\* appear multiple times in the\* same **Parent**, because\* that, in turn, should give a **Child** multiple references back to the\* **Parent**, but\* a **Child** holds only one reference to a **Parent**.

#### Risk of infinite loop n => 1

When\* a **1 => n** relationship is synchronized, you\* may have a risk to an infinite loop. When\* you\* add a **Child** to a **Parent**, then\* the\* **Parent** is assigned to the\* **Child**, upon which the\* **Child** is added to the\* **Parent** *again*.

An earlier solution proposed for this, is that in synchronizing the\* relationship, you\* never boldly **Add** the\* **Child** to the\* **Parent**, but\* you\* execute a **Find Or Add**, which prevents the\* **Child** from being added again, when\* it is already in the\* **Parent**’s list. This would\* have worked, but\* n => n synchronization already required a different solution, that will be more efficient for n => 1 synchronization as well.

When\* you\* assign an item to a list for synchronization purposes, no synchronization is to be executed on the\* other side again.

You’d\* have to call a special **List Item Set** procedure, accessible only to the\* related class, that simply won’t synchonize back again.

#### n => n Synchronization

One *related item* in one object always creates *one related item* inside the\* other object.

Two items, related to eachother in an **n => n** relationship, are always connected to eachother, by connecting two specific list positions to eachother.

In **n => n** synchronization, **Object A**’s reference to **Object B** will be replaced by a reference to **Object C**. When\* **Object B** is removed from **Object A**’s list, then\* **Object A** is also be removed from **Object B**’s list. After that, **Object A** is added to **Object C**’s list.

An item in one list is aware of its position in the\* other list. That makes it easy for an item in one list, to remove itself from the\* other list.

#### Risk of infinite loop n => n

But\* when\* you\* add **Object A** to **Object C**’s list of related items, then\* **Object C** will try to add itself to **Object A**’s list of related items, upon which **Object A** will add itself to the\* list of **Object C** again. An infinite loop should be prevented here.

When\* synchronizing the\* relationship between two objects in an **n => n** relationship, you\* will add a position to the\* list of the\* referrer, and next assign an item to this position.

When\* you\* assign an item to a list for synchronization purposes, no synchronization is to be executed on the\* other side again.

You’d\* have to call a special **List Item Set** procedure, accessible only to the\* related class, that simply won’t synchonize back again.

Another solution opted at first, was to execute a **Find Or Add** for synchronization, instead of executing an normal **Add** command. That would\* prevent a related item from being added and added again. But\* then\* you\* have the\* problem: maybe the\* same item *should* be added twice to the\* list, because\* one item can\* relate to another item multiple times, which also requires the\* other item to relate back to the\* first item multiple times. For each reference to an item, the\* item must have a reference back to the\* referrer.

So the\* new option is better: you\* have a special **List Item Set** procedure, possibly called by a special **Add** procedure, used solely for relationship synchronization, that won’t synchronize *back* again.

#### The\* abolished multiplicity of x

Earlier I had invented a multiplicity of **x**, which is plural, but\* then\* a fixed set of items, for instance three items. But\* **x** can\* be replaced by three separate **=> 1** relationships. **X** was abolished in particular, because\* it would\* cause a lot of unpredictable behavior when\* trying to synchronize the\* two relationship counterparts, especially in **n/x => n/x** synchronization.

#### Confusions about relationship synchronization

There used to be two points at which there was confusion about the\* workings of relationship synchronization.

##### Confusion 1

What can\* be confusing is that, when\* a **1 => n** relationship is synchronized, it can\* never be used as an **n => n** relationship. In the\* relationship **Parent 1 => n Child**, every time you\* add a **Child** to a **Parent**, **Child . Parent** is overwritten. When\* you\* change **Child  .  Parent**, the\* **Child** is removed from its original **Parent** and added to the\* new **Parent**. So you\* can\* never have the\* same **Child** in several **Parents**. If\* you\* want to use multiple **Parents**, you\* can\*’t.

It is often easier to define something in **1 => n** relationships, without thinking about it, that the\* backward relationship might be **=> n** too. But\* when\* a relationship is synchronized, the\* system falls apart when\* a **1 => n** is actually intended as **n => n** and you’re\* trying to *use* it that way.

This can\* be misconceived as an error in the\* new computer language, or an inability of it, while it’s really just a wrongly defined relationship.

So then\* it becomes really important to define the\* exact relationship type of something, even when\* it’s more difficult.

If\* you\* do want to use the\* **1 => n** relationship as an **n => n** relationship, you\* should change the\* relationship type, but\* you\* could\* also choose to split the\* bidirectional relationship into two unsynchronized unidirectional relationships. However, you\* will be loosing out on functionality and loosing integrity and coherence of the\* system.

A system in which all relationships are bidirectional and given the\* correct relationship type, functions in perfect harmony and everything is logical, correct and solid.

##### Confusion 2

Synchronization could\* cause confusion in older versions of the\* computer language, where you\* have two bidirectional relationships to the\* same class, that are given the\* same **Item Object Name**. In that case, one relationship’s counterpart synchronizing back to the\* related object could\* affect the\* other relationship. By default it is not\* allowed to have one class **A** being **1 =>** related to multiple classes, in which **A** has the\* same **Item Object Name**. That is only allowed if\* the\* other relationships are made unidirectional, and not\* bidirectional, or if\* *melding* is enabled. *Melding* is a topic, which makes it possible for multiple relationship classes or for instance *progressed objects (*article *Progression)*, to be referenced as a single related item, but\* it has to be stated explicitly that this is the\* intention. See the\* article *Melding*.

## Loose Ideas

Stereotyping relationships

Perhaps another typing can be assigned to a relation, instead of containment. For instance: *ownership* or *usage*. Perhaps a few standard ones, and it may be possible to define your own typing by specifying a String.

JJ

Relationships,

2008-11

The\* roles that symbols get when\* connecting them with lines.

JJ

### ± 2004

(Specific implementation in Creator 0.9)

The\* main unit in a relational structure is the\* *class*. There is a list of *classes*.

The\* classes are tied together with *relationships*. One class is tied to another.

The\* main object that defines a relational structure is the\* Structure object. The\* Structure object contains a Classes collection and a Relationships collection. If\* you\* want to add a class, you\* do that in the\* Classes collection. If\* you\* want to add a relationship, you\* do that in the\* Relationships collection.

The\* Classes collection contains objects of class Class.

The\* Relationships collection contains objects of class Relationship.

Every Class has an Attributes collection. A Person Class, for instance, could\* have a Name Attribute and an EmailAddress Attribute and more attributes such as Street, HouseNumber, ZipCode, etcetera. A Class also contains a RelatedClasses collection, which reflects all of the\* class’s related classes. You\* can’t add RelatedClasses to this collection. You\* have to define relationships in the\* Structure.Relationships collection and they will be *reflected* in the\* Class.RelatedClasses collection. Other members of the\* Class class are explained in other sections, covering different concepts. However, all members are briefly explained in a sub section below.

A **Relationship** consists of two **RelationClasses**. It contains two **RelationClass** objects that define the\* two classes of the\* relationship and how they relate to one another.

A **Relationship** also defines whether the\* relationship is **Bidirectional** or **Unidirectional**, by the\* **Boolean** **Bidirectional** member. If\* a **Relationship** is **Bidirectional**, then\* both classes are aware of eachother and refer to one another. If\* a **Relationship** is **Unidirectional** then\* only **RelationClassA** is aware of **RelationClassB** and refers to it, but\* **RelationClassB** is unaware of **RelationClassA** and doesn’t refer to it.

Even though a **Bidirectional** relationship would\* seem to make **RelationClassA** and **RelationClassB** equal opponents, **RelationClassB** in many cases is the\* inferior one. For instance, in writing XML files, **RelationClassB** is seen as contained in **RelationClassA** and not\* the\* other way around. In that case *direction* of the\* relationship does matter. However, still in many cases **RelationClassA** and **RelationClassB** are technically equal opponents. When\* you\* keep in mind which **RelationClass** is inferior and which one is superior, things like XML writing go well automatically. If\* you\* are sloppy with choosing if\* something is **RelationClassA** or **RelationClassB**, you\* could\* get trouble that makes you\* obliged to switch the\* two relationship classes within the\* relationship, but\* usually you\* won’t notice anything going wrong. So relax, but\* beware.

The\* two **RelationClass** objects define the\* relationship furtherly.

The\* member **Class** of **RelationClass** is very important to set, and it defines which class makes part of the\* relationship. Define the\* **Class** in both **RelationClasses** of the\* **Relationship** and you’ve\* made a relationship between the\* two classes.

A very important member of a **RelationClass** is the\* **AbstractNumber**. This defines whether a relationship class is **1**, **x** or **n**. If\* you\* define the\* **AbstractNumber** for both of the\* two relationship classes, you\* can\* for instance make a 1<=>n relationship between the\* two classes or a x<=>n relationship or whatever. **AbstractNumber** is **1** by default.

To define the\* quantity of x, you\* set **ExactNumber**. For instance, in a   
**Line n<=>2 Point** relationship, you\* define for the\* **Point** **RelationClass** that its **AbstractNumber = x** and its **ExactNumber = 2**.

Two other important members of a **RelationClass** are **CreateObjects** and **EnsureSubObjects**. **CreateObjects** is by default **True**, exceptions not\* regarded. **CreateObjects** says that when\* a new position is created within the\* related list, an object is instantly *created* in that position (See *Objects and Object Positions*). In many cases you\* want that to happen. Sometimes you\* don’t want objects to be created, because\* you’d\* want to assign an object to that position yourself. Then\* you\* set **CreateObjects** to **False**. Furtherly, **EnsureObjects** will see to it that you\* can’t\* assign **Nothing** to the\* object position. It is **True** by default, but\* can\* be set to **False**. For more information see the\* sections *Create Objects* and *Ensure Objects*.

And then\* there’s another member of **RelationClass** that is important to mention. And that is **ListType**. **ListType** is usually set to **NormalListType**, but\* can\* also be set to **ReferenceCountedListType**, **RegistrationListType**, **SharedListType** or **SelectionListType**. The\* list then\* gets very special behavior. For more information see the\* *Specialized Lists* section. Although I don’t explain them here very thoroughly, **ListType** does mean a lot for the\* general structure of the\* system.

You\* can\* see that inside a **RelationClass** object, much more is defined than just the\* **Class**. That’s why **RelationClass** is a separate class. Many times I will speak of a relationship class, and you\* shouldn’t confuse it with just a class then\*, because\* it will be defining a class as it is in the\* context of a relationship.

The\* elements as I’ve described them in this section, form the\* following structure of Classes, Attributes and Relationships.

Structure

|

|-- Classes

| |

| |-- Class ()

| |

| |-- Attributes

| | |

| | |-- Attribute ()

| |

| |-- RelatedClasses

| |

| |-- RelationClass ()

|

|-- Relationships

|

|-- Relationship ()

|

|-- Bidirectional

|

|-- RelationClassA and RelationClassB

|

|-- Class

|-- AbstractNumber

|-- ExactNumber

|-- CreateObjects

|-- EnsureObjects

|-- ListType

Every other concept of J Data is hung up on this main structure of **Classes**, **Attributes** and **Relationships**.

For instace, the\* physical appearance in the\* user interface of a list defined in a **RelationClass** with **AbstractNumber = n**, is defined inside that **RelationClass** object. The\* whole appearance of a **List Control** is defined inside a **RelationClass**. That is an example of how the\* user interface is defined right inside the\* general structure above. The\* same way *all* of the\* application’s features are defined in the\* context of the\* relational structure of **Classes**, **Attributes** and **Relationships**.