UML 2.5.1 Overview

This document provides an overview of syntax elements (and occasionally their semantics) from UML 2.5.1. THIS DOCUMENT WILL NOT TELL YOU HOW TO CONSTRUCT A GOOD MODEL: you need to understand material from lectures on abstraction, purpose, and complexity to hope to arrive at good models. Furthermore, THERE ARE VARIOUS DETAILS HERE THAT YOU MIGHT NOT EVEN USE; avoid the temptation to cram in all syntax when it is not appropriate!

The formal UML specifications are all big, complex and focus on something called its metamodel (a model of its models); the notation is an afterthought. They have also vacillated about whether the defined notation is required or a suggestion. In UML 2.5.1, it seems to be required unless flexibility in some details is explicitly mentioned; at least, this will be the interpretation in the course.

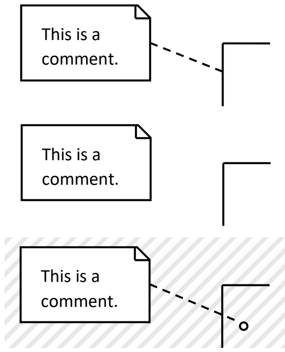
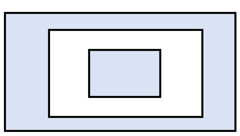
UML has vacillated about whether specific diagram kinds are supported and what those kinds are. The end result as of 2.5.1 is confusion and inconsistencies. In this course, we will talk about/use certain kinds of diagrams as though those were inflexible.

Note: any notation with a shaded background is unofficial (not conforming to the official specification) but acceptable for use in the course.

Table of Contents

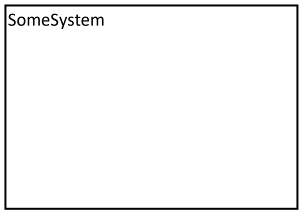
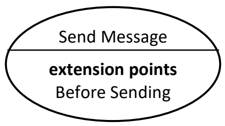
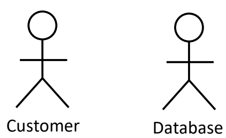
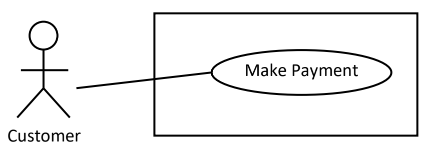
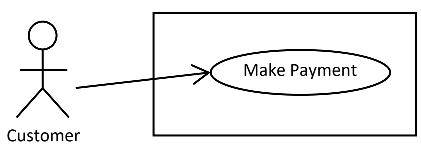
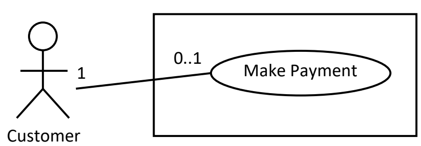
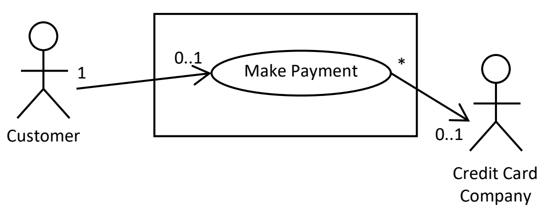
General

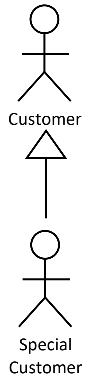
Some details apply to all diagrams.

Name	Notation	Notes
comment		Comments provide human-readable, additional information. They have no formal meaning otherwise. Officially, two forms are available: (1) with a dashed line connecting it to some other element to which the comment refers; or (2) floating in space, presumably as a general comment about the model and not specifically about one element therein. **Unofficially**, the non-comment end of the line can be attached to a small, empty circle placed next to some detail in the element being referred to (e.g., a particular operation in a class), suggesting that the comment applies to that detail. Only add comments if there is something too complicated to model or otherwise unobvious. Comments don't appear in every model; in fact, they are rather uncommon.
keyword (stereotype)	«interface» «interface» «foo» «interface, foo»	UML predefines various special keywords that can be used in models to specify more precisely what kind of model element is being used. In past versions of UML, these were call stereotypes, but now "stereotype" is used to mean a more narrow, special case of some other model element. For us, there are only a few places where keywords MUST appear and a few where they (or stereotypes) MAY appear if you wish to specify certain details. In general, avoid making up your own keywords or stereotypes except where I explicitly tell you that you can. The keyword appears in *guillemets* (doubled angle brackets can be used if you don't have access to guillemets); if more than one keyword is needed in the same place, you can either place each in its own guillemets or provide a comma-separated list within a single set of guillemets. With the exception of certain required keywords (like "interface"), you are not likely to use them.
additional properties	{<*property*>} {<*property*>=true, <*property*>, <*property*>=foo}	A programming language like Java contains lots of additional details. Officially, every kind of model element is defined in what is called the UML *metamodel*: this specifies all the properties that each kind has; sometimes these have official notation/symbols to use and sometimes not. (Go figure why they don't bother to provide notation for the full metamodel; they probably couldn't agree about it all!) In cases without official notation, you really need to remember to ask yourself, "Is this detail important to my model?" Remember: the goal is not a perfect representation of every detail: that is what the source code is for! But if you are sure that you ought to represent one or more such details, here is what is possible. **In many cases, you are either officially permitted to use the property notation shown here, or for practicality, I will tell you where that is permissible.** When the properties are connected to the usual <*name*> : <*type*> kind of syntax that is mentioned various places below, the properties (between the curly braces) come after the <*type*>.
shading and colour		Shading and colour can sometimes make things clearer, when the number of overlapping boxes and lines start to be overwhelming. Judicious use of shading or colour is acceptable in the course; don't overdo them: this isn't a paint-by-numbers activity!
presentation options		The UML specification states that the standard notation can be replaced with other symbols (e.g., a computer icon in place of the stick figure for non-human actors in use case diagrams). **In the course, you are required to adhere to the standard notation or the options that I have detailed for you.** In particular, using the wrong kind of arrows (or certain kinds of arrows in the wrong kind of diagram) can be meaningless or misleading. **You are responsible to ensure that the syntax you use represents what you intend it to mean.** Using an arrow when you should have used a rectangle is rather unlikely, but it is easy to change a solid line to a dashed line or vice versa, or to use the wrong kind of arrowhead. You will be graded, in part, on correct use of syntax and correct semantics.
guard	[<*boolean-expression*>]	A guard is a common construct that is used in many diagrams wherever tests of values are needed. The key identifying syntax is the presence of the enclosing square brackets. The contents of the Boolean expression depend on what other names make sense in the context where the guard is found and the level of formality that the modeller is trying to achieve. A typical example could be "[foo < 10]". The locations permitted for guards is quite limited and it is important not to just slap them down arbitrarily: you would be risking total confusion.

Use Case Diagrams

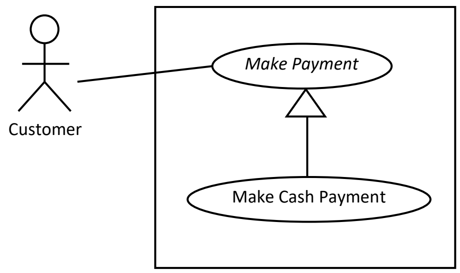
A use case diagram is not a complete description of use cases; they must be augmented with other information, which in the course, we represent with use case descriptions. In official UML, use case descriptions would be represented by other diagrams or other annotations.

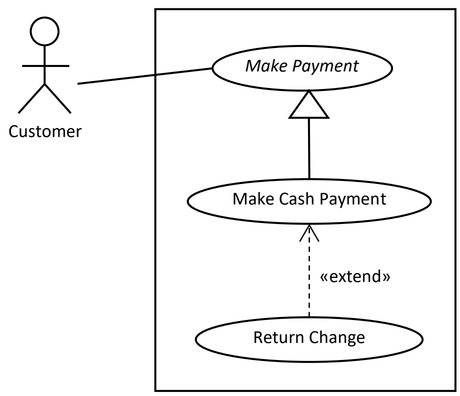
Name	Notation	Notes
system boundary (officially, subject)		The system boundary (officially, the *subject*) differentiates between what is inside the system being modelled and what is outside of it. It is simply represented as a rectangle, with the name of the system in the top left corner. Use cases can be placed inside the system boundary, and actors can be placed outside. Note: It is always an error to place an actor inside the system boundary or a use case outside it.
use case	 	"Each UseCase specifies some behavior that a subject can perform in collaboration with one or more Actors. UseCases define the offered Behaviors of the subject without reference to its internal structure. These Behaviors, involving interactions between the Actors and the subject, may result in changes to the state of the subject and communications with its environment. A UseCase can include possible variations of its basic behavior, including exceptional behavior and error handling." The notation for a use case is simple: an ellipse containing the name of the use case. One variation exists in which the ellipse is divided by a horizontal line: the name goes above the line, and any *extension points* defined by the use case can be listed below the line. The variation with the extension points can become awkward when there are multiple extension points or their names are long. The UML specification provides alternative notation to cope with such cases, but we will not accept that in this course.
actor		"An Actor models a type of role played by an entity that interacts with the subjects of its associated UseCases (e.g., by exchanging signals and data). Actors may represent roles played by human users, external hardware, or other systems. "**NOTE**. An Actor does not necessarily represent a specific physical entity but instead a particular role of some entity that is relevant to the specification of its associated UseCases. Thus, a single physical instance may play the role of several different Actors and, conversely, a given Actor may be played by multiple different instances." The stick figure icon can be used for all actors, human or otherwise. Officially, alternative notations are provided in the UML specification, but these will not be accepted in the course. Note: It is always an error to represent the system being modelled as an actor, since that is what the system boundary delineates and the system cannot be outside itself.
association	   	An association can be placed only between one actor and one use case. The interpretation is that one instance of the actor interacts with one instance of the use case (at a time). The unadorned line can be interpreted this way; it also does not specify whether the system or the actor initiates the interaction. Adornments are also possible. An arrowhead indicates navigability: the entity (actor or use case) at the other end of the arrow can send messages (or perform any other interaction, which we model simply as "sending a message"). Without navigability, the entity receiving the message can only respond, not initiate messages. In the second example, a Customer instance can send messages to Make Payment, but Make Payment can only send responses to those messages. Multiplicities indicate how many instances of the actor are involved with how many instances of the use case (simultaneously); a multiplicity is either a single integer greater than or equal to 1, or a range "*x*..*y*" where *x* can be as small as 0 and *y* can be as large as unbounded, shown by "*"; *x* has to be smaller than *y*. The range "0..*" is equivalent to "*". By default, a set of use cases will all have their association ends adorned with "0..1", meaning that zero or one instance of the use case could be occurring at any moment, and so we don't bother to provide this multiplicity. Also by default, one instance of each of the associated actors is involved in the interaction, so the set of actors will have their association ends adorned with "1". Other situations are far less common but possible in principle Multiple actors (or actor instances) can be involved in a single use case instance. In the example, an instance of Credit Card Company can be involved in Make Payment, but the Credit Card Company could be involved in multiple instances of Make Payment at the same time. The fourth example illustrates this.

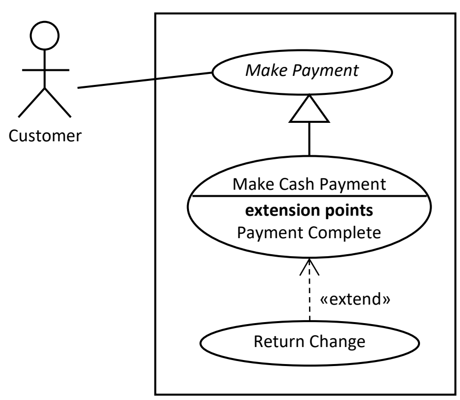
<table>
<tr><th>Name</th><th>Notation</th><th>Notes</th></tr>
<tr><td>actor generalization</td><td>
</td><td>The only kind of relationship permitted directly between two actors on this diagram is generalization. This indicates that one actor is a more general form of the other actor, or conversely, that one actor is a more specialized form of the other actor.

This is strictly an anti-symmetric relationship.

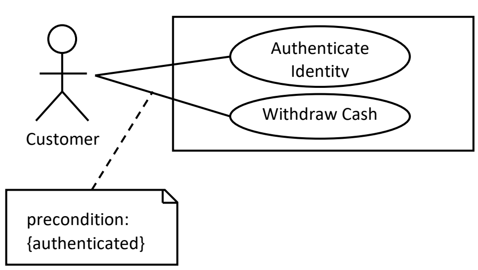
The notation is an arrow with a solid line and a triangular but closed arrowhead. The arrowhead points at the more general actor.

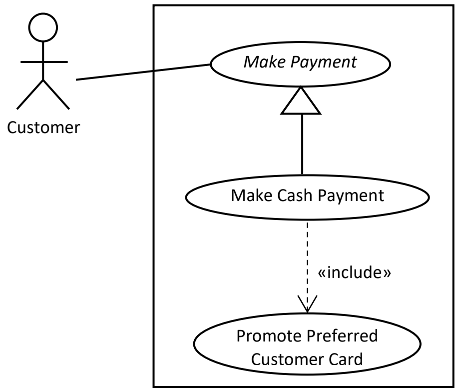
The more specialized actor inherits all associations from the more general actor. The more specialized actor can add other associations as well, but care must be taken to remember that the inherited associations are still there.</td></tr>
<tr><td>use case generalization</td><td>
</td><td>Some use cases can be designated as abstract, meaning that they represent a general kind of task but without being specific on the details. This is indicated with slanted or italic text for the label. One or more specialized versions of the use case can be provided, which will usually be concrete. The specialized versions inherit all associations of the general version.

In the example, Make Cash Payment is a concrete specialization of Make Payment, which is abstract. Each instance of Make Cash Payment will involve a single instance of Customer, adhering to our default interpretation of an unadorned association. The notation for use case generalization is identical to that for actor generalization.</td></tr>
<tr><td>extend</td><td>


</td><td>“An Extend is a relationship from an extending UseCase (the extension) to an extended UseCase (the extendedCase) that specifies how and when the behavior defined in the extending UseCase can be inserted into the behavior defined in the extended UseCase. The extension takes place at one or more specific extension points defined in the extended UseCase. Extend is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in one or more UseCases.
The extended UseCase is defined independently of the extending UseCase and is meaningful independently of the extending UseCase. On the other hand, the extending UseCase typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending UseCase defines a set of modular behavior increments that augment an execution of the extended UseCase under specific conditions.”

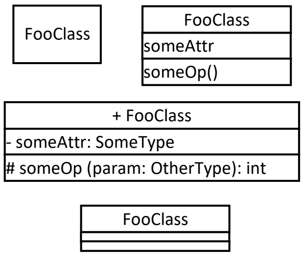
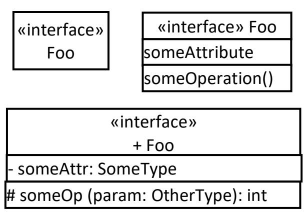
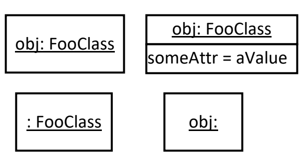
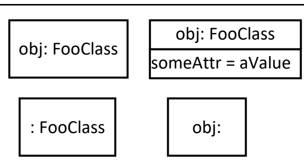
The first example shows that the Return Change use case extends Make Cash Payment. This means that sometimes Return Change will occur at some point during the execution of Make Cash Payment. This example is not more specific, but we can specify an explicit extension point from which Return Change extends and we can then use our use case descriptions to specify what happens there.

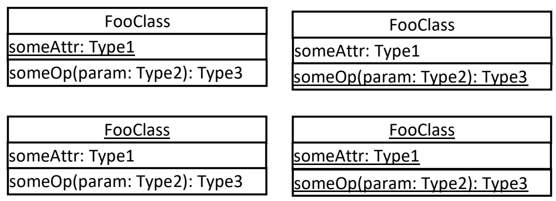
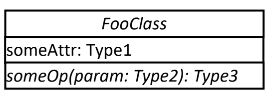
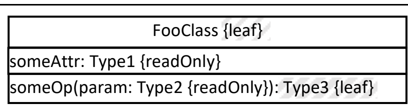
The notation is a dashed arrow with an open arrowhead plus the keyword “extend” as a label on the arrow; the arrow points from the extending use case to the extended use case.</td></tr>
<tr><td>precondition</td><td>
</td><td>When some property has to be true before an interaction or an extension is permitted, this can be indicated with a note stating “precondition: {<<i>details</i>>}” attached to the constrained relationship. The use case description should also conform to this precondition.</td></tr>
</table>

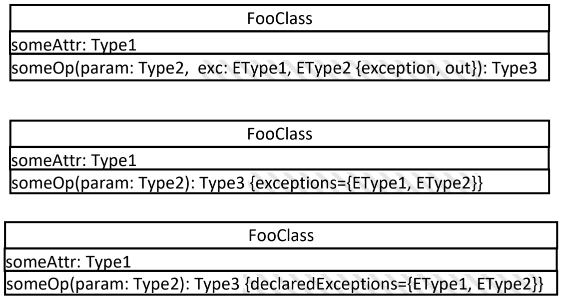
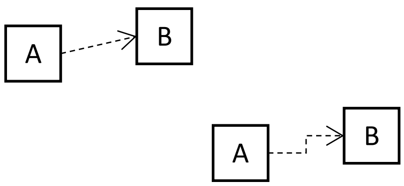
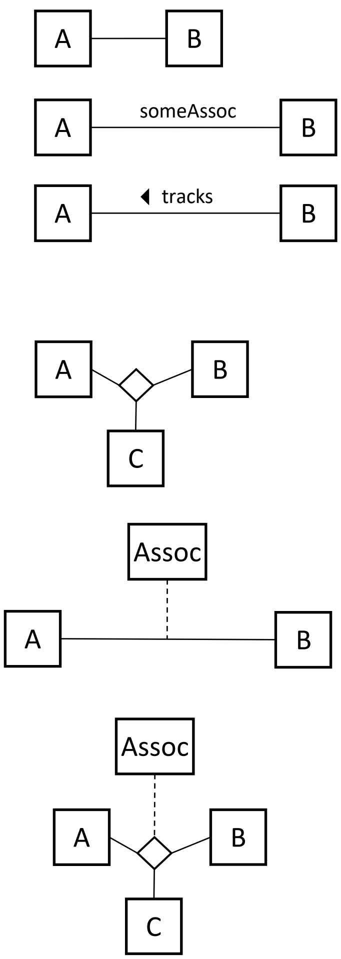
Name	Notation	Notes
include		"Include is a DirectedRelationship between two UseCases, indicating that the behavior of the included UseCase (the addition) is inserted into the behavior of the including UseCase (the includingCase). It is also a kind of NamedElement so that it can have a name in the context of its owning UseCase (the includingCase). The including UseCase may depend on the changes produced by executing the included UseCase. The included UseCase must be available for the behavior of the including UseCase to be completely described. The Include relationship is intended to be used when there are common parts of the behavior of two or more UseCases. This common part is then extracted to a separate UseCase, to be included by all the base UseCases having this part in common. As the primary use of the Include relationship is for reuse of common parts, what is left in a base UseCase is usually not complete in itself but dependent on the included parts to be meaningful. This is reflected in the direction of the relationship, indicating that the base UseCase depends on the addition but not vice versa. All of the behavior of the included UseCase is executed at a single location in the included UseCase before execution of the including UseCase is resumed." **The notation goes in the opposite direction from that for extend**: from the including use case to the included one. Using include only makes sense if there is non-trivial behaviour shared by two or more use cases; often, generalization is more appropriate to use. The example here shows the syntax, but the fact that only one use case includes Promote Preferred Customer Card calls inti question its presence in the diagram.

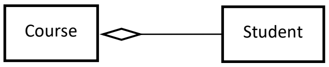
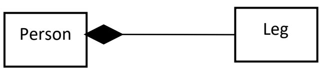
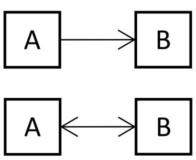
Structure Diagrams

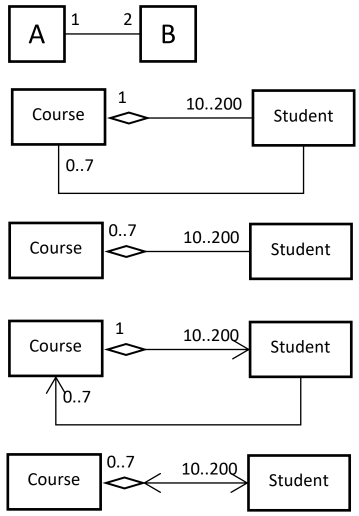
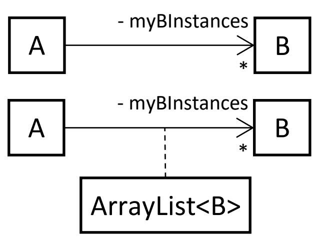
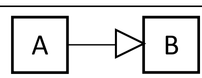
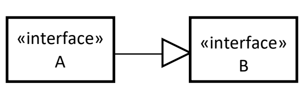
Our structure diagrams are a combination of what were known as class diagrams, object diagrams, and package diagrams, when people realized that they (sometimes) needed to intermix them. UML 2.5.1 in its Annex A again explicitly mentions various kinds of diagrams, as well as treating structure diagrams as the superclass of class, object, package, and some other kinds of diagrams, but the rest of the specification is inconsistent with the descriptions there. In the end, this nitpicking only matters if you are worried about conformance with other tools (beyond the course).

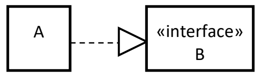
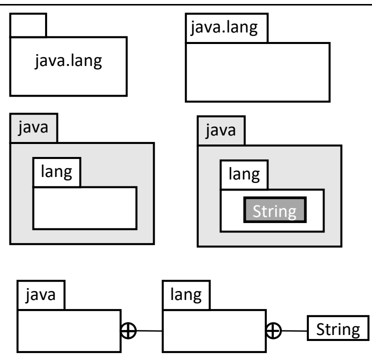
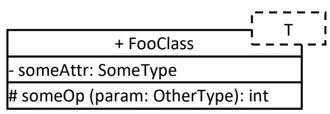
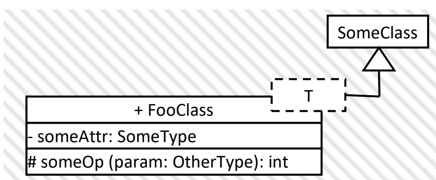
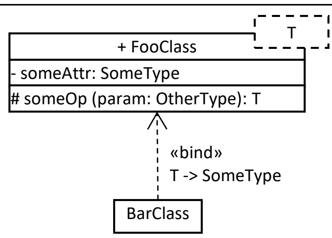
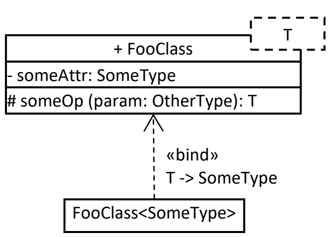
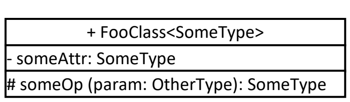
Name	Notation	Notes
Class		A rectangle (without rounded corners, and not boldface!) with a name inside it (no colon ":" shown!) represents a class. Small variations on this **mean other things**, so it is easy to confuse them. In practice, the four variations shown here show the range of common use: (1) [top left] only the class name is shown (the class is named "FooClass" in this case), with all other details suppressed or non-existent (you can't tell which); (2) [top right] three compartments are shown: topmost is the name, middle are the attributes, bottom are the operations; (3) [middle] the same as #2, but with additional details shown: visibility, parameters for the operation, type/return type for the attributes and operations; (4) [bottom] this is identical to #1 but the compartments have not been suppressed; this is less tidy than #1, though legal. **Additional** decorations are possible as discussed later. Attributes are often referred to as fields or member variables, depending on the programming language. Attributes can possess various properties, like visibility; these are detailed below. Operations are often referred to (somewhat incorrectly) as methods, procedures, functions, or member functions, depending on the programming language. Operations can possess various properties, like visibility; these are detailed below. In some languages, constructors and destructors can be named arbitrarily; UML thus includes the stereotype "Create" (yes, capitalized) that can be used to adorn operations to indicate which is a constructor; it goes in front of the method name, though they don't explain if it should come before or after the visibility notation, when present. (The specification does not say how to represent destructors.) In Java, constructors must have the same name as their class, so they are unambiguous: **don't use the "Create" keyword for Java-based models**, therefore. And Java does not possess destructors. **Note that constructors do not have return types!** *Java also possesses things called* initializers *and* static initializers. *These have no useful equivalent in UML; you can treat them as nullary methods named "*<init>*" and "*<clinit>*" respectively if you* <u>*really*</u> *think you need to represent them, and number them if there is more than one of either. Or you can reconsider if they are really important! Hint!*
interface		Basically the same *syntax* as a class but with the keyword "interface". Interfaces are *semantically* significantly different from classes, so we will treat the "interface" keyword as **non-optional**, i.e., if the rectangle doesn't contain «interface», it must be a class, *whether or not that is your intent and whether or not that would make sense*.
object (also called instance) (formally, instance specification)		It is rare though not impossible to want to model a specific object in a structure diagram; a typical case would be to provide a concrete example of objects at run-time, their contained values, and their links amongst themselves. *In most cases though, you are doing something wrong.* Objects are distinguished from classes by the presence of the colon ":" that separates the name from the type and the presence of the <u>underlining</u>. If either of these is absent, you have made a mistake. The label consists of *<name> ":" <type>*. Either the name or the type (or both) can be suppressed. Without a name, the object is anonymous (meaning: without name). In even more rare cases, you can also model the specific values that the object holds in its versions of the attributes (these are called *slots*), as in the version at the top right. The value held in the slot "someAttr", here, is "aValue", whatever that might be.
role		A role represents a placeholder into which different objects can be slotted at different times. If we were to model SENG 300 and the people involved, the instructor could be a role as could be a student whereas "Robert Walker" or "Jane Smith" would be specific objects that could play either of those roles at some time in their lives. Syntactically, a role looks like an object except for the **lack of underlining**.
public visibility	+	Usually shown preceding the name of a type, attribute, or operation. Equivalent to the modifier "public" in a programming language like Java.
private visibility	-	Usually shown preceding the name of a type, attribute, or operation. Equivalent to the modifier "private" in a programming language like Java.

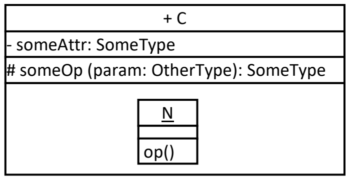
Name	Notation	Notes
protected visibility	#	Usually shown preceding the name of a type, attribute, or operation. Equivalent to the modified “protected” in a programming language like Java.
package visibility	~	Usually shown preceding the name of a type, attribute, or operation. Equivalent to contexts where none of the visibility modifiers “public”, “protected”, or “private” are given, in a programming language like Java.
colon separator	:	The name or signature (if shown) comes to the left and the type (if shown) comes to the right. For attributes and operations, the colon and the following type can be suppressed, but the name/signature cannot be unless the whole attribute or operation is itself suppressed. For formal parameters, the name or type can be suppressed (sometimes I have seen the name by itself without anything else, but this is not correct UML). **There is no official equivalent of the “…” construct of Java: you can just use “…” where needed.** Objects and roles are distinguishable by underlining the whole label in the case of objects.
static modifier		The “static” modifier can be found on attributes, operations, classes, and interface types in Java; it is a signal that each can be accessed/called/referenced without reference to any particular instance of a class. The name/signature and following type (when not suppressed) is underlined. The examples illustrate different elements of the class (or the class itself) being designated static; these mean different things. The fourth example shows all the elements being static. Note that classes and interfaces can only be designated static if they are nested (not shown here). Usually, it is attributes and/or operations that are static. **Be careful not to confuse this** with the underlining to indicate objects.
abstract modifier		The “abstract” modifier can be found on operations, classes, and interface types in Java; it is a signal that each can is not complete and hence cannot be called or instantiated. The name/signature and following type (when not suppressed) is italicized or slanted. In the example, the “someAttr” attribute is not abstract, while the class and the “someOp” operation are abstract. Note that the opposite of abstract is *concrete*. Operations and classes are all concrete unless explicitly abstract. Interface types in Java are implicitly abstract, though it is legal to explicitly declare them as abstract; you can use the slanted/italic text or not in specifying interface types, as you find more convenient. Annotation types and enumeration types are always concrete; annotation types are a special case of interface types and enumeration types are a special case of classes.
final modifier		In Java, the “final” modifier is used to mean different things in different contexts. On a variable (field, local variable, formal parameter), it means that the initial value, once assigned, cannot be altered. On a method, it means that its implementation cannot be overridden. On a class, it means that that it cannot be subclassed. In UML, these ideas are represented in different ways. For attributes, we can append the property “readOnly”, as shown. For classes and operations, there are two properties in the metamodel: “isLeaf” and “isFinalSpecialization”. (Why two? Don’t they achieve the same thing? Good questions; no answers.) Problem: there is no official notation for either! Solutions: I have seen examples on the web where they use “leaf” as a property, which is what I do in the example here, but it is unofficial. You could also just say, “Hey, enough already! I’ll just make up a ‘final’ property and use it everywhere.” **In the context of the course, this is not acceptable.** If you decide that it is important to model the presence of the “final” modifier, you have to use “leaf” on classes and operations but “readOnly” on variables of whichever kind.
other modifiers	{<*modifier*>}	Java has many other, more rarely used modifiers: “default”, “native”, “strictfp”, “synchronized”, “transient”, and “volatile”. As far as I know, none of these is supported even inside the metamodel. **Are you really sure that they matter to your model?** If so, your only options are to use a custom, unofficial property or a comment.

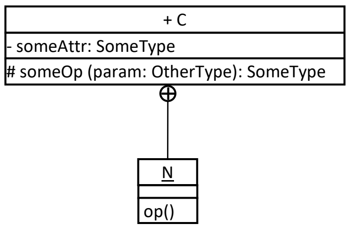
Name	Notation	Notes
declared exceptions		Dealing with exceptions is thoroughly unclear in the UML specification. What follows is how you should handle this in the course, in the context of structural diagrams, but it is unofficial; we will look into notation for behavioural diagrams afterwards. Formal parameters are usually thought of as passing arguments into a method, but they can also receive return values, because some programming languages allow multiple return values or for arguments to be pass-by-reference, so the underlying variables are changed. We are going to exploit this feature. Every operation that you want to model as <u>*declaring*</u> the potential to throw an exception, you can show with an extra formal parameter (you can make up the name; here I used “exc”) with the type being a list of all the types that might be thrown as per the method signature. Here I show two types on the “exc” formal parameter; it also has the property “exception” which corresponds to the presence of “isException” in the metamodel (but without notation specified); this extra parameter will also have the “out” property. Alternatively, I have also seen models where they specified the property “exceptions={<*exceptionClass*>, <*exceptionClass*>, …}” or “declaredExceptions={…” but this can be even more messy, in my opinion. You may use either of these alternatives **This all gets complicated really fast. Avoid dealing with the exceptions if you can suppress them without ignoring important details!**
dependency		This diagram says that “the class A depends on the class B”. (One can add more details inside of A and B as needed.) The formal specification states: “A Dependency signifies a supplier/client relationship between model elements where the modification of a supplier may impact the client model elements.” In particular, the absence of B will prevent A from executing or even compiling. **Be careful with the kind of arrow**: a dependency is dashed with an open arrowhead. Change either of those properties and you change the relationship represented. **Putting bends in the arrow does not change its meaning, and the individual line segments have no significance. (This is true for all relationships in all diagrams.)** While you might be interested in showing dependencies of or on particular attributes or operations, the standard notation does not support that.
association		According to the specification: “An Association classifies a set of tuples representing links between typed instances.” Well, OK, but that’s not really helpful. (A *link* is not really defined explicitly by the specification: it is simply a relationship between two or more objects.) In practice, the presence of an association between two types means that the implementation of one or both possesses an attribute whose type is closely related to the other. (Either the type itself, or some kind of collection of instances of that type, in the case of multiplicities greater than 1.) In practice, an association implies a dependency as well, so we don’t show both. (Formally, you might be able to have an association without a dependency, but this doesn’t make much sense to me, so we can assume that an association is also one dependency or both; the presence of navigability makes this clearer.) In the first example we see an association with no labels, etc. This is extremely vague, but syntactically permissible. Classes A and B are associated “somehow”, meaning that there are relationships between certain instances of A and certain instances of B. To be more specific, we can add adornments to the association, as discussed later: navigability, multiplicity, visibility, ownership, names. In the second example, the association (not one of its ends) is named “someAssoc”. (When we need to start also adding labels to the ends, the label for the association itself really gets in the way, so it tends not to be used anymore.) In the third example, I have added a tiny triangle to represent the order in which you should read the association; here: B tracks A. It is uncommon to see these little “reading order” triangles. Note that the association has two *association ends*. It is important to be able to differentiate a label on the association from labels on the association ends! (This starts to be a failing of the notation when the association starts to be more complex, but “less is more” is an important maxim for modelling.) In general, an association can have multiple ends, but this is <u>*very rare*</u>, and I have personally never used it nor seen it used except in contrived examples. The notation is as shown in the fourth example. “Any Association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each Association memberEnd connecting the diamond to the Classifier that is the end’s type. An Association with more than two ends can only be drawn this way.” In the course, showing the diamond notation for an association is **ONLY PERMITTED FOR ASSOCIATIONS WITH MORE THAN TWO ENDS**. And since it is unlikely that you will encounter the need for such an association, it is unlikely that you will encounter the need for this notation. In some cases, it makes sense to represent the association itself as a class! This is called an *association class* and can be represented as in the fifth and sixth examples. This is uncommon (outside pedagogical examples), and if you are tempted to do this, you are probably

Name	Notation	Notes
		modelling too much. An association class is both an association and a class; you can add all the usual details to the class.
aggregation		(Aggregation is now officially referred to as the "shared aggregation kind". Here is what the specification says about it: "Indicates that the Property has shared aggregation semantics. Precise semantics of shared aggregation varies by application area and modeler." Great, thanks for the help.) Here is my interpretation (based on historical definitions) that I want you to adopt. Aggregation refers to a **weak whole/part relationship** in which the parts might also be parts of other wholes. So we might model the students in SENG 300 as being instances of the Student class, aggregated into a Course class, but you are shared amongst different instances of Course and in other aspects of your life if we needed to model those too. It is a stronger relationship than normal association, as a course without students cannot function, and how well it functions depends in part on the specific students. If you are unsure whether a given situation qualifies as a real whole/part relationship, fall back to simple association. The class representing the whole has an **open** diamond at its association end (much smaller than the association diamond talked about above). Note that aggregation is necessarily an asymmetrical relationship; **if you put the diamond at both ends, you are plain wrong, and if you have two relationships where the diamond is at different ends, you are equally wrong.**
composition		(Composition is now officially referred to as the "composite aggregation kind". Here is what the specification has to say: "Indicates that the Property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects." That feels a bit better to me than the mess for aggregation.) Here is my interpretation that I want you to adopt. Composition refers to a **strong whole/part relationship** in which the parts cannot be shared between different wholes (though they might be interchanged) and where the parts generally do not survive the destruction of the whole. Composition is a stronger form of aggregation. Only use it if you are sure that it is appropriate. Again, the class representing the whole has a diamond at its association end but now it is **FILLED**. Again, **it makes no sense to have diamonds at both ends of the association (or two associations with the diamonds at different ends)**, whether they are filled or not. (The issues regarding the diamond end and the navigability arrowhead are the same here as with aggregation.)
navigability		Navigability is a property representing the ability to arrive at an instance given a different instance. Navigability can be indicated at any or all association ends. The only kind of marking permitted in the course is the open arrowhead (meaning "navigable"). (The UML specification also provides an explicit indication for non-navigability, but you may not use that in the course, and outside the course, don't be surprised if no one knows it.) At one point in the specification, it says, "An association with neither end marked by navigability arrows means that the association is navigable in both directions." This contradicts details I see elsewhere, as well as the general convention, "lack of information cannot be interpreted to mean something". Ah, wonderful consistency! However, we will stick to that definition: no arrowheads will be interpreted as navigability in both directions, but it would be better to put arrowheads at both ends.

Name	Notation	Notes
multiplicity		Multiple instances can be involved in a single relationship. For an association (and thus, also for aggregation and composition), a specific kind of small label can be placed at each association end. There are two forms possible: a number and a range. A number can be any positive integer or the special symbol "*", meaning "unbounded". A range has the syntax <*lower-bound*> "." "." <*upper-bound*>, where both bounds are a non-negative integer (or "*" for the upper) and the lower bound must be smaller than the upper bound. The special case "0..*" is shortened to "*" as being synonymous. If multiplicities are absent from one or both ends of an association, the default interpretation is "1". A composition *must* have (explicitly or implicitly) a multiplicity of "1" at the association end of the "whole"; any other multiplicity is wrong. Note that a normal association with multiplicities "1" and "*" (for example) is not the same as an aggregation with those multiplicities. In the first example, instances of the association involve one instance of A and 2 instances of B; there is no guarantee that those class instances are uniquely involved in that association instance. In the second example, between 10 and 200 instances of Student are involved with an instance of Course and one instance of Student is involved with between 0 and 7 courses. The third example is subtly different from the second. It says that instances of the relationship involve between 0 and 7 instances of Course and between 10 and 200 instances of Student. In this case, the association represents the enrollments in the courses and the courses being taken by each student; since that is a potentially complex situation, an association class might be called for, to explicitly model the relationship. These are clearer with explicit navigability arrowheads, so I have added them in the last two examples.
association end name, visibility, and other properties		An attribute and an association are effectively the same thing; the one is shown in the attribute compartment while the other is shown as the arrow. The name and other properties of the attribute can be placed as a label at the appropriate association end. **In the course, you have to choose between using the attribute notation vs. the association notation for each relationship: you cannot have both for a single relationship.** You may, however, choose to use an attribute in one case and an association in another (either-or). In the example, the association end adjacent to B is labelled "-myBInstances"; it also has a multiplicity of "*" and it is marked as navigable. This means that A tracks two instances of B which can be referred to as "myBInstances" and that these are visible only privately. Other attribute properties can be inserted after the name, but realize that space will quickly be a problem. Often, the implementation of the class will contain a field whose type is not a second class, but some sort of arbitrarily large Collection of that second class. There are two options: ignore the Collection and just show the second class with multiplicity "*" (usually the preferred option), or show the Collection as an association class on an association with the second class. Note that it would be wrong to model this as an association with ArrayList<B>, almost always, as that is almost certainly an implementation choice that shouldn't be modelled: ArrayList could be replaced by various other data structures that would achieve the same idea (perhaps performance would differ) and thus the concepts to be modelled don't include the presence of ArrayList. In such a case, we have two reasons not to show the association class!
ownership		The UML specification spends a lot of energy trying to differentiate between ownership, navigability, visibility, etc. The ownership notation (a small dot at one or both ends of an association) is <u>*extremely uncommon*</u> (i.e., no one uses it). In this course, it is <u>**not permitted**</u> to use the ownership notation and it will be treated as wrong in all cases. We will use the convention that the type at the opposite association end owns the association end if it is navigable (explicitly or implicitly). Thus, in the examples in the previous part, A owns "myBInstances" but B doesn't own the other end (B instances don't know about any A instances). If we eliminated the arrowhead, the unnamed end would also be navigable, and B would own it (this would presumably be a mistake, so ensure that you label the end or place the arrowheads appropriately.)
generalization	 	In the top example, A is a subclass of B; B is a superclass of A. The relationship is called generalization and **points to the more general class, i.e., the superclass**. In the bottom example, A is a subinterface of B; B is a superinterface of A. The relationship is also called generalization and **points to the more general interface**. Both of these relationships are implemented in Java with the "extends" keyword in the declaration of the subclass or subinterface. If you are tempted to put the triangle at both ends, you have severely misunderstood something.

Name	Notation	Notes
interface realization		In Java, this relationship is implemented with the “implements” keyword. In the example, the class A implements the interface B, meaning that A provides implementations for the operations declared by B, or that one or more of them are abstract. To be a concrete class, A must define or inherit every method declared by B. In this example, all such details are suppressed. **Note that this is the only relationship for which the line is dashed while the arrowhead is a complete triangle (usually empty, though this doesn’t matter here).** If you are tempted to put the triangle at both ends, you have severely misunderstood something.
package		Packages are a traditional mechanism for organizing a system, corresponding to the Java notion of packages. The notation is a rectangle with a little tab symbol on it. The name of the package can go either in the tab (usually used when you plan on showing more details inside the package, e.g., classes) or in the main rectangle. You can show them in a nested, hierarchical fashion too; I’ve added shading to make this clearer, but shading is unofficial (though common). All four of these examples mean exactly the same thing, with the exception of the last one, where it says that String is inside lang is inside java. The notation at the bottom is an alternative that is sometimes more convenient; the circle with the plus sign in it indicates the *containment* relationship. You can also mix-and-match, sometimes showing the physical nesting and sometimes showing the containment relationship in the same diagram; but be careful about too many inconsistencies leading to confusion: *legal* is different than *clear*. To represent the class java.lang.String in your model, you could do it like either of these ways or just show the fully-qualified name of the class in the name compartment, or ignore the “java.lang.” part. It depends on what matters in your model. Cleave to that KISS principle!
template class		This is equivalent to a generic class in Java. Templates are both more general and less general than Java generics, coming from C++ ideas. In the top example, T represents the type parameter for the FooClass class; thus, FooClass is generic. I will describe how you parameterize a generic class in the next part. One thing you can do in Java is to specify the supertype of the type parameter (or the subtype, which is useful more rarely). There is no simple way to achieve this in official UML. I have seen two ways used: sticking a colon followed by the supertype’s name after the type parameter’s name. But the UML specification explicitly says that this is how the *metaclass* would be specified, and the supertype is not a metaclass! Instead, I like the style I show here where “SomeClass” must be a supertype of whatever type you put in place of the type parameter; **use this style for the course**. This also allows you to model more details of “SomeClass” where desired. (You could also represent the rare cases where the type parameter is something like “T super SomeClass” in which case you could just have the generalization arrow point the other way.)
template binding		You have a generic class and it is parameterized by a type argument: this is what template binding means, making the generic class into a parameterized class. The first example shows that the BarClass class is really the FooClass class with its type parameter T bound to SomeType. This is only really worthwhile in programming languages where the resulting class can (or must) be aliased to a different name. You can add the implied attributes and operations that result, within the BarClass class; for clarity, you can add “/” in front of any visibility marker in the bound class: this means “derived”. Java is not one of these classes. The second example shows the kind of redundant noise that this syntax can cause in Java. FooClass<SomeType> is the result of binding T to SomeType. But if this is the only binding that happens with FooClass, simplifying this into one class often makes more sense. The third example shows this kind of simplification. Note that all occurrences of T have been replaced with SomeType (there was only one, aside from the type parameter in the dashed rectangle).

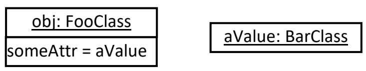
<table>
<tr><th>Name</th><th>Notation</th><th>Notes</th></tr>
<tr><td>nested type</td><td>

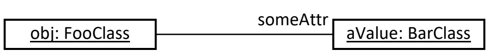

</td><td>Java allows for classes and interfaces to be nested inside other ones; consider a class C that has a nested class N (either one could instead be an interface):

if each instance of C is supposed to keep track of instances of N and each instance of N can access its enclosing instance of C, the N is called an inner class (respectively, inner interface or simply inner type to mean either);
inner classes may be local, meaning that they are declared inside a method, which is the only place they can be used;
inner classes may be <i>anonymous</i>, meaning that they have no name; this is only possible in a class instance creation expression, i.e., “new SomeType(…) { <<i>body of anonymous class</i>> }”, and is the only situation where an interface type can be mentioned in such an expression;
otherwise, a nested class is not inner: this is indicated on the declaration of N via the “static” keyword.

The rules around nested and inner types in Java are complex, even excessively so to the point of becoming hard to deal with; fortunately, you don’t need to deal with all that here.

There are officially two ways to indicate a nested class in UML: (1) literally place its rectangle inside its enclosing type’s rectangle, often in another compartment added for the purpose (although I talked about there being three compartments, there can actually be arbitrarily many, but you usually only see three; look back at my comments about shading and colour for clarity); or (2) use the containment relationship that I mentioned in conjunction with packages.

Personally, I find the approach using the containment relationship to be simpler to deal with.</td></tr>
<tr><td>link</td><td>

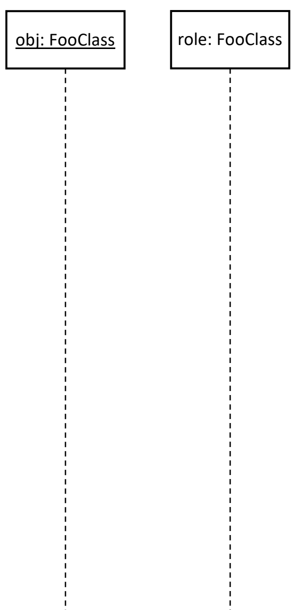
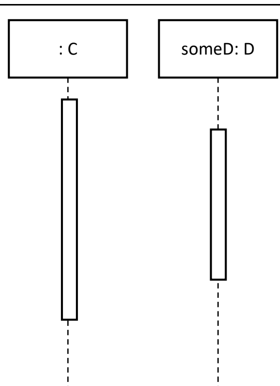

</td><td>In the unlikely event that you want to model individual objects in your structure diagram, one and only one kind of relationship is available to you: link. If two types have an association relationship, instances of those two types will have a link relationship. The link could be shown either with the slot notation or with a line/arrow notation. The annotations on the line/arrow are similar to those on the association: a name at a link end, equivalent to the name of a slot; navigability arrows identical to those on its generalizing association. Since we are dealing with objects, a multiplicity greater than 1 on an association would have to be elaborated into the equivalent number of objects, each linked to the object(s) at the other association end. Since this starts to be complicated to show in detail, official notation schemes are scarce, and you likely want to rethink what you are trying to achieve.

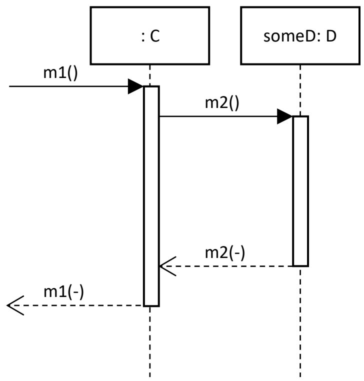
I repeat: it is rare that you would need to show objects in a structure diagram!

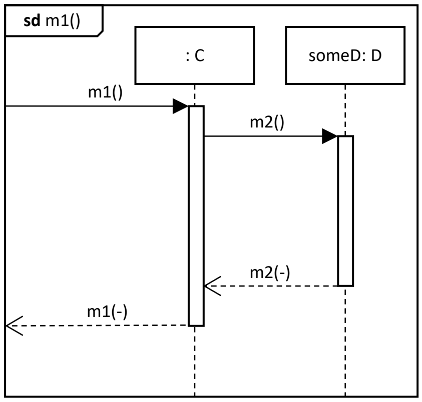
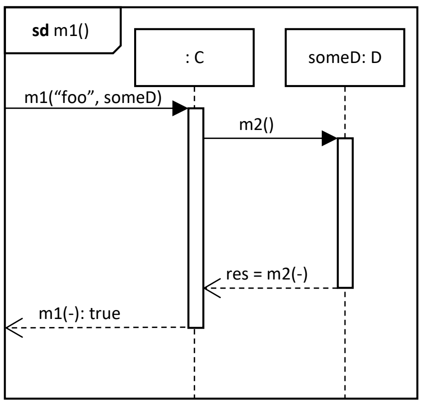
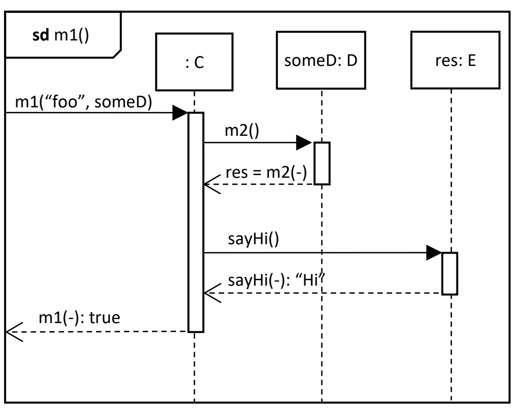
In the example, I repeat an object that we saw earlier: it is named “obj”, it is of type “FooClass”, and it has a slot named “someAttr” with value “aValue”: we can now see an example of what “aValue” could be. It turns out that “aValue” is an instance of “BarClass” (I haven’t shown any slots for it). The “aValue” in “obj” refers explicitly to this other instance. The link between them is implicit.

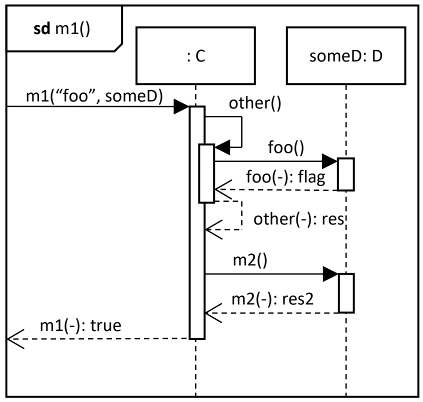
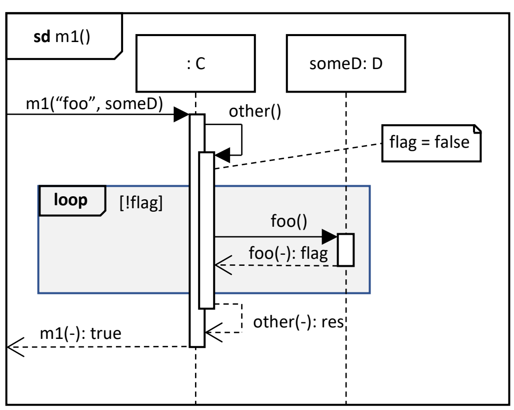
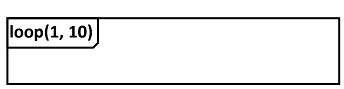
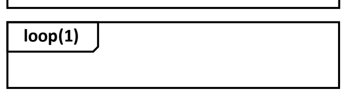
We can also use “graphical notation” (as it is called in the specification), drawing the link as a line and name its end. These two alternatives are exactly the same. We could also place navigability arrows on one or both ends; this would be extra information, though, and the top example doesn’t include that information (strictly speaking, though this interpretation seems reasonable).</td></tr>
</table>

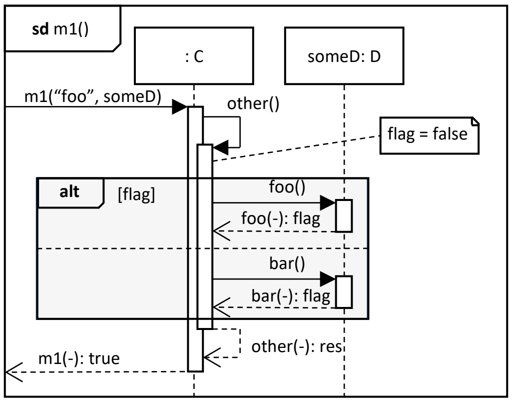
Sequence Diagrams

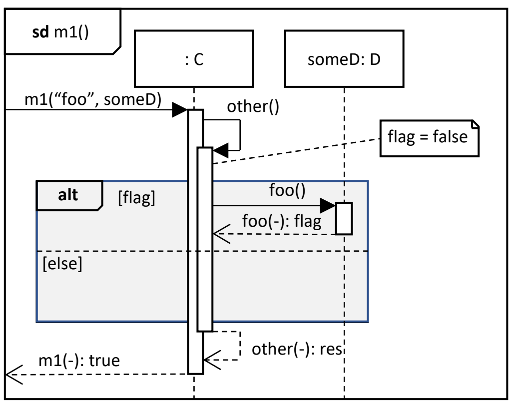
Name	Notation	Notes
class	FooClass	It is **uncommon but not impossible** to need to represent classes on your sequence diagrams, typically when you need to call a static method. This is a class, not an object and not a role. For instance methods (i.e., ones not marked "static"), the method call has to be made on a receiving object (usually inside of a variable of some kind, so you see things like "a.foo()"). A class is not (normally) a receiving object: you are likely doing something wrong. The notation is the same as what can go in the name compartment of a class in a structure diagram. (Note that interfaces don't have static methods, so their presence makes no sense.)
interface	«interface» Foo «interface» Foo	Basically the same *syntax* as a class but with the keyword "interface". It is **very uncommon but not impossible** to need to represent interfaces on your sequence diagrams.
object (also called instance)	obj: FooClass : FooClass obj:	It is rare though not impossible to want to model a specific object, even in a sequence diagram: usually (but not always) you want a *role*. Objects are distinguished from classes by the presence of the colon ":" that separates the name from the type and the presence of the underlining. If either of these is absent, you have made a mistake. The label consists of *<name> ":" <type>*. Either the name or the type can be suppressed. The notation is the same as for objects in a structure diagram, except that **you can only have the name compartment**.
role	role: FooClass : FooClass obj:	A role represents a placeholder into which different objects can be slotted at different times. If we were to model SENG 300 and the people involved, the instructor could be a role as could be a student whereas "Robert Walker" or "Jane Smith" would be specific objects that could play either of those roles at some time in their lives. Syntactically, a role looks like an object except for the **lack of underlining**.
lifeline		Since a sequence diagram represents events occurring over time and since objects can be created or destroyed during the events represented therein, objects and/or roles are shown as explicitly existing over a certain period; this is called a lifeline. Formally, a UML lifeline is the rectangle (representing the object, role, or [rarely] class) AND the dotted line; informally, we tend to think of only the dotted line as being the lifeline. (I don't know that it makes practical difference outside formal definitions.) *Relative time* proceeds down the lifeline. Given two events e_1 and e_2 on the lifeline with e_1 above e_2, e_1 occurs some time before e_2; there is no way to tell the exact amount of time between two events as their physical distance apart **means nothing**. (There are additional details that allow the time interval to be made somewhat precise, but these only matter in contexts beyond the course like for real-time systems with parallelism and timing constraints.) If the two events happen on *different* timelines, we still consider their relative vertical position as indicative of their ordering. (In systems with parallelism, there is no guarantee of relative order and so the UML specification becomes very complex in worrying about the semantics here, but these are complications beyond the course.) In most cases, the lifeline extends from the top of the diagram to the bottom. Visually, this means that some objects/roles/classes lie at the top (or as close thereto as is visually acceptable); these exist at the start of the sequence depicted and the diagram is SILENT about when and why they were created. Likewise, those objects/roles/classes typically still exist at the end of the sequence depicted and the diagram is SILENT about when and why they will be destroyed. Any object/role/class shown further down the diagram must be coming into existence at that point; there is specific syntax required to show WHY that object/role was created; **if you don't use this, your diagram is wrong**. The examples here are what you would typically expect, assuming the frame of the table cell is the edge of the diagram: the objects/roles near the top and the dotted line extended nearly to the edge. The few millimeters above and below are for visual comfort.
activation bars (formally, execution specifications)		An activation bar indicates when an object/role/class is actively doing something (or waiting for a response to a call to another method). In most (all?) programming language implementations, method calls are placed in what is known as the call stack. In Java, a "main()" method is called first; imagine that this then calls "m1()", which then calls "m2()". Before "m2()" returns, the call stack has "main()" on the bottom, with "m1()" above it, and "m2()" on top. The method on top is the one actively computing something; the others are waiting for a response. On the diagram, each of these methods would be shown as a separate activation bar. But this is an incomplete description: unless the methods are static, they have to be called on a receiver object, which has to be an instance of the class implementing the method. Consider two classes C and D, where C implements main() and m1() and D implements m2() (it is very rare that you would want to represent the "main()" method, but it is always static). The example here shows what you can expect the lifelines and activation bars to look like once the call to C.m1() occurs; we will look at adding the syntax of messages in the next part, but I have aligned the starts and ends of the activations bars to account for the nesting effect of the calls and returns. There are two roles, one anonymous and one not; the diagram indicates that they were created before this sequence started and they still existed at its end. It isn't clear what cause the activations to start and end; we need more syntax for that.

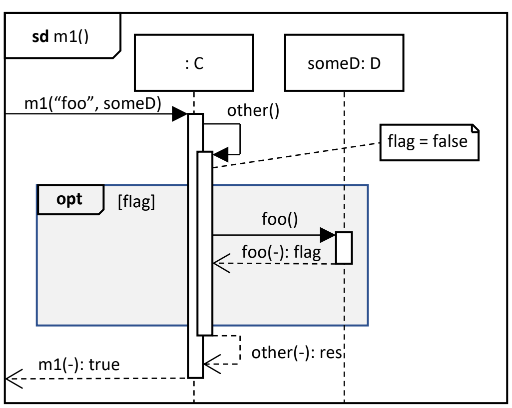
Name	Notation	Notes
		According to the UML specification, the activation bars are optional. Eliminating them easily leads to confusion and then nonsensical models, so in this course: **activation bars are REQUIRED**.
method call and method return (also known as message passing and message return)		Method calls and method returns are shown as arrows between activation bars. For a method call, the arrow must consist of a solid line with a filled, triangular arrowhead. (This is a synchronous method call, the only kind we will deal with in this course.) Details of the method call are placed as a label above the arrow; it is important to spread out the details enough to avoid ambiguity about which label belongs to which arrow. Here, we see the method "m1()" being called on the anonymous role of type C. Who calls it? The diagram doesn't say, which is why the arrow seems to come from nowhere; of course, the call comes from somewhere, but we want to ignore that because the source does not alter the behaviour. (Formally, it comes from a *gate* on the edge of the sequence diagram which in turn is also connected to the caller, but that is too much detail for us to bother with.) We also see that the method call arrow points EXACTLY to the corner of the activation bar on the anonymous role; this is important: the reception of the method call is what CAUSES the activation of that anonymous role (i.e., whatever object is playing that role). **A common mistake is to have the method call point to the middle of the activation bar; this would mean that the role/object was *magically* activating in anticipation of the arrival of the message: WRONG!** Note that the second message (to m2()) starts awhile after the start of the activation of the anonymous role, i.e., the start of the second arrow is vertically below the end of the first arrow: it takes some time to do "something" before the second call is made; it cannot be instantaneous! Notice that the return messages show the name of the called method, but with the argument list replaced with a wildcard "-" which means "whatever is appropriate". Furthermore, note that the method call is simply shown as the name of the method. What about the receiver object? It is *implied* by which object/role actually receives the method call: that's the point. **It is wrong to use the "someVariable.methodName()" kind of syntax!** Next, we see that each of these method calls returns after some time. This is important. When the method is finished doing whatever it does, it is no longer active (so the activation bar ends) and the method return comes EXACTLY from the bottom corner; this is important: the end of the method's computation CAUSES a method return. Strictly speaking, it takes some time between when a message is sent and when it is received, but we generally ignore this and make the arrows perfectly horizontal. It is paramount that you not misorder each of the method call sends, method call receptions, method return sends, and method return receptions (formally, these are known as *occurrence specifications*). Notice the staggered effect that results: the called method's activation bar clearly occupies an interval of time strictly inside the interval of time occupied by the calling method's activation bar. The messages in this example are simplistic: to add in arguments and a return value, we need more syntax, but first I will add another detail that can be helpful. Note that, although the message return is optional according to the specification, **in this course, the message return arrow is always REQUIRED and its label must conform to the syntax specified here below.** (For those of you who dig around and find "lost" and "found" messages [represented as black dots on the end of the "dangling" arrow, realize that those are different concepts than not showing where a call comes from. The method call here is not found; we just are not modelling the sender. **Do not use the lost/found notation for this.**) In the course, we expect that a method call label will consist of <*method-name*> "(" <*arguments*> ")", where the <*arguments*> are zero or more arguments separated by commas. An argument can be a literal or a reference to a name accessible from the sending lifeline. In the case that you don't care to model a specific value, you can use "-" as a wildcard for any specific argument, meaning "arbitrary legal value". In the course, we expect that a (normal) method return label will consist of [<*assignment-target*> "="] <*message-name*> "(" ")" [":" <*value*>]. The <*assignment-target*> and following equals sign is an optional means of assigning the return value to a variable of some kind; this may be a name of convenience for you to use later in the interaction or it can be a reference to an object/role within the diagram or it can be a reference to a field within the current object/role represented by the lifeline receiving the reply. You don't have to show an explicit value if you don't want/it doesn't make sense; **"void" is not a value!** We will see later how to represent thrown exceptions.

Name	Notation	Notes
interaction (interaction frame)		Sometimes it is unclear where the “top”, “sides”, and “bottom” of the diagram are. We can draw a frame around it for clarity. At the top left corner, you are supposed to place a pentagon (irregular) containing “sd” (for “sequence diagram”, usually boldfaced) “followed by the Interaction name and parameters”, according to the specification. Usually, you can just give it a name that makes sense for the context. In the course, interaction frames are optional, but you are always evaluated on the clarity of your models.
argument passing and return values	 	Often, methods require that formal parameters be bound to arguments, and that return types be bound to values. On our sequence diagrams, we can make use of the names of objects/roles for this purpose as well as pragmatically using literals (strings, numbers) or other notes when filling in the details would be overly complicated and not relevant to what we are trying to achieve with our model. **Be careful: when you use a name on one of these arrow labels and it is the same as the name of one of your objects or roles, it means THAT object or role (i.e., whatever object is playing the role)!** I have seen many times where that Is not what someone intended. Sometimes, it makes sense to add roles just so they can be passed, even if they do not send/receive messages (i.e., the object playing the role is passive in the sequence being modelled). (Note also: primitive values are not objects; although UML supports dealing with primitive objects I have yet to see a case where it made sense to do more than just provide a literal value.) In these examples, I have just added the new syntax without worrying about maintaining consistency with other examples; as another interpretation: similar names in different examples are mere coincidence. I have modified the m1() method to now take two arguments: here, I pass it a string literal and reference to the named role someD. Presumably (but not certainly), this is how the anonymous role comes to have a reference (a link) to the object that it then uses as the receiver object for the call to m2(). We also see that m1(..) also now returns a boolean value, “true” in this case. There are two alternatives for representing return values: (1) showing a colon after the method name and empty parentheses, followed by the return value; and (2) assigning the return value to a variable. (Having both is also possible, thereby defining what the value of the variable is.) The syntax in the second case usually doesn’t tell us what the actual return value is; rather, it assigns whatever the return value is to a name, “res”, which is presumably clear from the context or is explained elsewhere. The name “res” can now (i.e., in events below the message return reception) be used by the anonymous role to make other calls. The second example shows this: the anonymous role cannot communicate with res until res is returned from the call from m2(), but notice that res already existed at the start of the sequence. Whether two objects can communicate with each other or not cannot be represented directly in this diagram. Note that the position of the label on *these* arrows (unlike on associations or links) is arbitrary; we choose it so that it fits while not letting it be crossed by other lines (if practicable). **And if a method is declared as returning “void”, it means that it DOES NOT return a value**; the method return label cannot show a value in such a case, nor will the variable assignment syntax be correct!

Name	Notation	Notes
self-call		A self-call is a method that an object calls a method on itself, either the currently active method or a different one. Often, in Java source code, this looks like a method call without an apparent receiver (hence, it is implicitly "this"). In the example, I have added a self-call from the anonymous role (to itself, of course); the method involved is simply called "other" and takes no arguments. The "other()" method has a return value, which I have assigned to "res". Since this is what the return value from "m2()" was formerly assigned to, I have updated it to make the assignment to "res2". Notice that the activation bar involved in the self-call is REQUIRED: it is slightly offset so you can see both the original activation bar and the one for the self-call. If "other()" then made a self-call (to itself or any other method on the anonymous role), this pattern would continue: another slightly offset activation bar, with the method call and method return arrows bent back in this fashion. If the self-called method then makes a non self-call, we would see the method call and method return connected between the nested activation bar and the receiver object's activation bar, as I have added to the example. It is important to understand what this diagram says. There is some object of type C that the modeller didn't want to name (for whatever reason); this may be held in some variable of type C or of a supertype of C, but that detail didn't matter for the model's purpose. A call is made to "C.m1(..)" with that object as the receiver; in that call, two arguments are passed: a string literal "foo" and a reference to an instance of type D labelled "someD". It doesn't matter what the variable holding "someD" was called in the source code: the modeller decided to give it a simple name here. The anonymous role becomes active when it receives this call; after some time, it makes a self-call to "other()"; this nested call then makes a call to "foo()" on the receiver labelled as "someD". When "foo()" is finished its computation, it returns from the method call, passing a value here labelled as "flag". After some time, "other()" finishes its computation and returns from its call, passing the value here labelled as "res". After some time, "m1(..)" makes a second call, this time, to "m2()". When "m2()" is finished its computation, it returns from the call, passing the value here labelled as "res2". After some time, "m1(..)" is finished its computation, so it returns to its caller (outside the interaction), passing the value "true".
other control structures		Up to now, we have only seen modelling of the control structure known simply as "sequence", i.e., one operation performed after another. We also need ways of representing other, standard control sequences. There is a standard syntax form used for these called an *interaction fragment*: it looks much like the interaction frame we talked about earlier, but it is placed inside the interaction frame, with lifelines and messages entering and leaving it. To understand the specific syntax and semantics used for different control structures, we need to look at them individually. Realize that these fragments sometimes need to be nested, at which point, visual noise gets to be a problem. Ah yes, the KISS principle rears its ugly head yet again.
loop fragment	   	Continuing to use the previous example, I have added in an interaction fragment representing a loop (I also eliminated the call to "m2()" for the sake of saving some space, but you can imagine it still being in the interval between when "res" is returned and when "true" is returned). For the sake of making it more obvious, I have shaded the background of the loop frame. The idea is that there are some interactions that occur before the loop is encountered, then the loop, and then some other interactions. In this simple example, the two lifelines that were shown before the loop frame are shown as continuing during the loop and also after the loop. Notice also that the activation bars that were started before the loop continue through the loop and onwards after the loop. Since a particular loop many run an unknown number of times (0 times or forever or anywhere in between), it is important to be clear about allowing for behaviour that skips the loop altogether. In this syntax, I am using a guard to specify when the loop should be entered (and exited): "[!flag]". What is "flag"? A boolean variable I made up. How can you tell it is boolean? Because otherwise it would make no sense. Before the first call to "foo()" happens, it is not clear what value "flag" would have had, so I inserted a comment before the start of the loop. (This is an example that there is no formal interpretation of this comment. It is apparently pointing to the nested activation bar, but before loop entry. Given the context, a human being ought to realize that this is a specification of the initial state of "flag".) If and only if the guard evaluates to true (in this case, when "flag" is false) will the execution continue into the loop body; otherwise, execution skips the loop and continues below it. When can "flag" become true? Only if that is the return value from "foo()", which is why the value of "flag" is changed to the value that is returned from it. At the bottom of the loop, execution jumps back to the beginning, the guard is evaluated, and execution continues either inside the loop or outside it. Loops can also work for a fixed number of times like in the second example. Here, the loop happens at least once, but at most ten times. Why the uncertainty? Something might happen inside the loop to prevent it from iterating more than once. In the third example, the loop will occur at exactly once. In the fourth example, we see mixed syntax. The specification states, "After the minimum number of iterations have executed and the Boolean expression is false the loop will terminate"; apparently, the maximum number of iterations only matters if the Boolean expression is not false before that. Note that it would make no sense to have only the lower bound plus a guard.

<table>
<tr><th>Name</th><th>Notation</th><th>Notes</th></tr>
<tr><td></td><td></td><td>What variable name can you use to reference the loop counter? The specification says nothing about it. If you want a loop counter, you would have to set it up and increment it manually.</td></tr>
<tr><td>alternative fragment</td><td>






</td><td>An alternative fragment is designated by the presence of the keyword “alt” (usually boldface). It requires the presence of a guard at the start.

An alternative fragment is divided vertically into two compartments (officially, these are called interaction operands) separated by a dashed line. The top compartment is executed if the guard evaluates to true; the bottom compartment is executed if the guard evaluates to false.

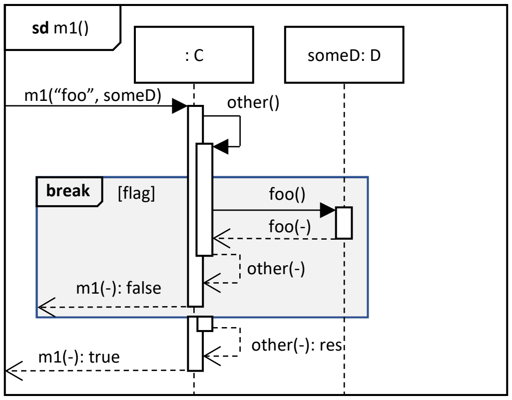
In the first example, we again see the use of “flag”. If it is true, “foo()” will be executed (the return value is assigned to “flag” at that point, but that doesn’t change anything and the return value could be dropped from the diagram); if it is false, “bar()” will be executed (the return value is assigned to “flag” at that point, but that doesn’t change anything and the return value could be dropped from the diagram). I left the note stating that “flag” is false before the fragment, so the top compartment will never be executed! It is still legal syntax, but to avoid this issue, “flag” could be treated as some global variable or a field on C or added as a parameter.

“Should the shading go over the activation bars, or under, and likewise with the various edges?” Good question; no answers. (I show the borders of the fragment on top, but the shading behind. I prefer that because it is clearer where the fragment’s boundary intersects the activation bars.)

Also in the second example, I have emptied out the bottom compartment. If “flag” is false, nothing happens, and execution skips to after the bottom of the alternative fragment. Since this seems like a waste of space, you can use an option fragment instead like in the third example; it’s a bit more concise.

Officially, the bottom-most compartment can be guarded by “[else]”, but the lack of a guard there means exactly the same thing.

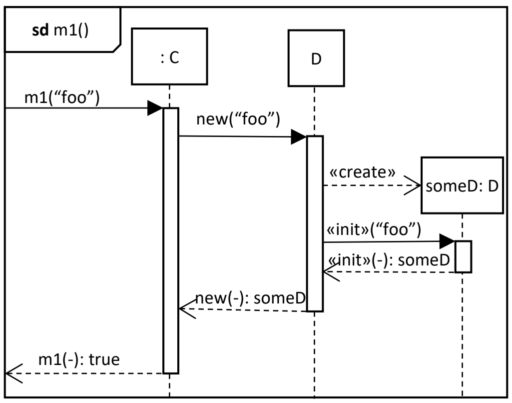
Although the specification is not clear on this point, the metamodel suggests that arbitrarily many compartments are possible, which would be useful for modelling a nested if-then-else structure or switch statement.</td></tr>
</table>

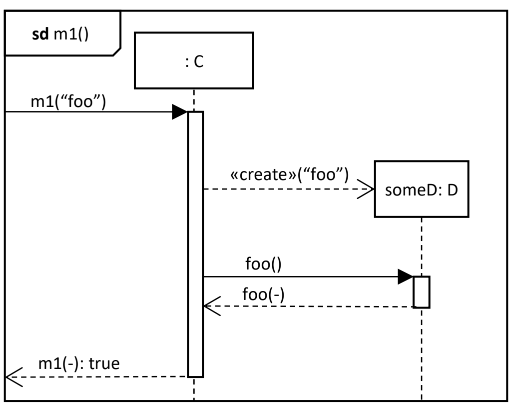
<table>
<tr><th>Name</th><th>Notation</th><th>Notes</th></tr>
<tr><td>break fragment</td><td>


</td><td>
The break fragment is used to stop the execution of the containing fragment and perform some alternative as a replacement. This can lead to complications in presenting differing lifelines and activation bars as in the example.

Here the break fragment is again guarded by a test of the value of “flag”; it is not clear where “flag” comes from or how it is set, but presumably it would be clear from context. If “flag” is true, the body of the break fragment is executed: “foo()” is called, the nested activation bar representing “other()” returns, and the outer activation bar representing “m1(..)” returns; as a result, we see the ends of those two activation bars.

But, if “flag” evaluates as false, the break fragment is skipped, and we see that the nested activation bar is still present along with the outer one; the calculation proceeds as before.
</td></tr>
<tr><td>instance creation

(also known as object creation, class instantiation)</td><td>




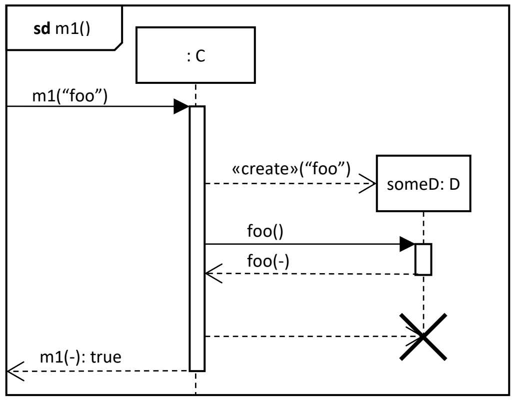
</td><td>
Creating an object dynamically can be an important thing, which should thus be modelled correctly. You must realize that this happens in two phases: memory allocation, and object initialization. In a language like Java, the details are often largely ignored … until you run into a bug you struggle to understand.

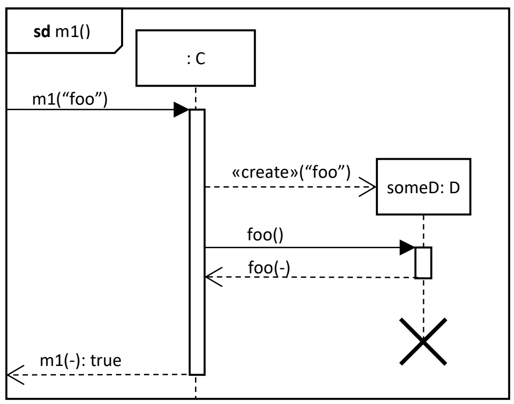
In UML 2.5.1, you need to differentiate the two phases, unless you are confident that the details don’t matter for the model and you are trying to simplify. We’ll consider both situations. Imagine that you are trying to model the instantiation of a class C through the use of a constructor that takes one argument of type String.

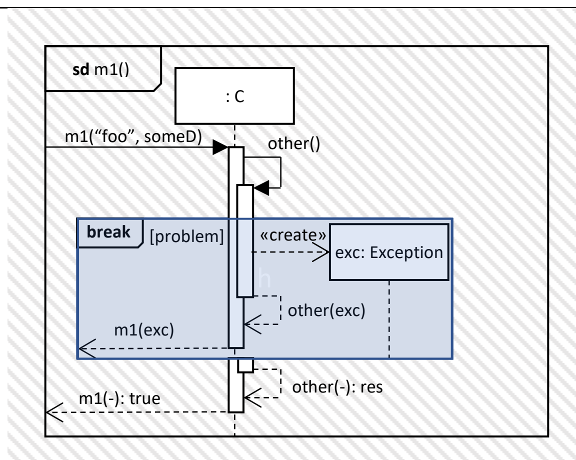
The first example shows the situation fully. A “method call” is made to the class D on the “method” “new(..)”, passing a string literal as argument. This is not really a method call, because allocating memory is an internal operation, but we can model that step like this. Next, class D itself creates an instance of itself (again, this isn’t really how it works, but the modelling is simpler this way); then it calls a special method known as “«init»(..)”, which represents the object initialization (this part is fairly accurate, though it can be complicated with multiple calls to constructors, super-constructors, instance initializers, vary and sundry). Once the initialization call returns, the class can return the newly allocated and initialized instance, which I have labelled “someD” for convenience.

That was rather complicated and it is rare that you would want to see all that detail just to represent an instantiation. More typically, you would skip almost all of it, as in the second example. Here, we compress all of that into a single “«create»” message, after which interaction can occur with the new object like any other. In this example, the object continues to exist after the end of the interaction. Is it a memory leak? Has a reference to the object been stored somewhere? Either is possible; the example doesn’t say.

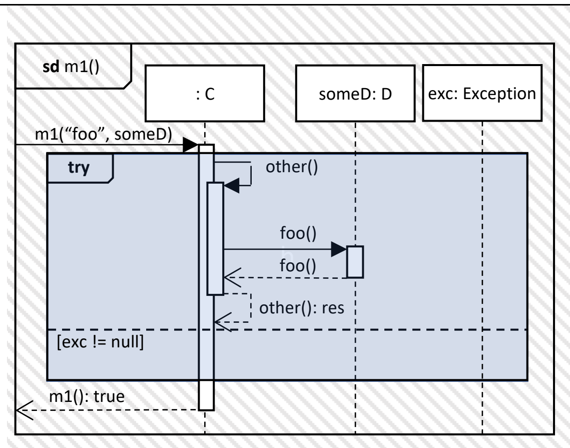
To be clear, in either case, the object creation is modelled by a “message” labelled “«create»” (I have added arguments where needed, but this is unofficial). This is a dashed arrow with an open arrowhead pointing to the object role being created. Hence: **THIS IS THE ONLY TIME YOU SHOULD HAVE OBJECTS/ROLES NOT AT THE TOP OF THE DIAGRAM.** Note also the difference between an object obtaining a reference to another object versus that second object being created; these are not at all the same thing.
</td></tr>
</table>

<table>
<tr><th>Name</th><th>Notation</th><th>Notes</th></tr>
<tr><td>object destruction</td><td>




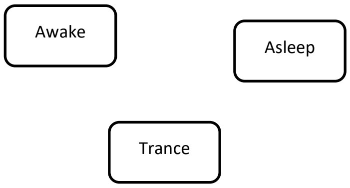
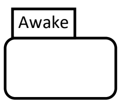
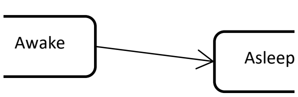
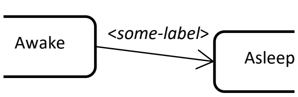
</td><td>
Just as objects can be created, they can be destroyed, and sometimes this is an important thing to model. In Java, object destruction happens “magically” for you. In UML, if you have modelled creating an object, you would typically want to also model its destruction or its explicit storage/communication to other parts of the system. Otherwise … what happened to it?
In this first example, we see that “someD” is explicitly destroyed at some point by the anonymous role. This is shown by ending the lifeline of “someD” in a big X, caused by the reception of a “message” indicated with a dashed arrow and an open arrowhead; there is no label on this “message”.
You would almost never destroy objects that you did not create in the same diagram. (Yes, there are situations where it could be possible, but it is far more likely that you are doing something wrong, or at least modelling too much detail.)
It is also possible to have objects be implicitly destroyed. You eliminate the explicit message, and just show the big X ending the lifeline, as in the second diagram. Presumably, the modeller wants to show here that the created object is destroyed but without active participation by the anonymous role. In fact, this is how Java works: you can’t explicitly destroy objects. Thus, the second example is the only way you should model object destruction when modelling Java code.
</td></tr>
<tr><td>exception throw</td><td>


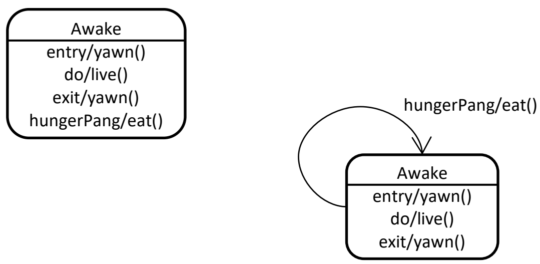
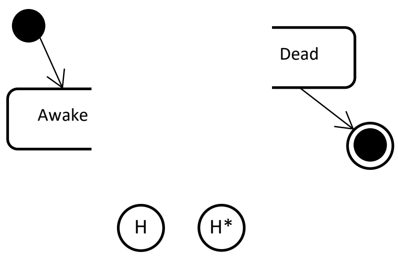
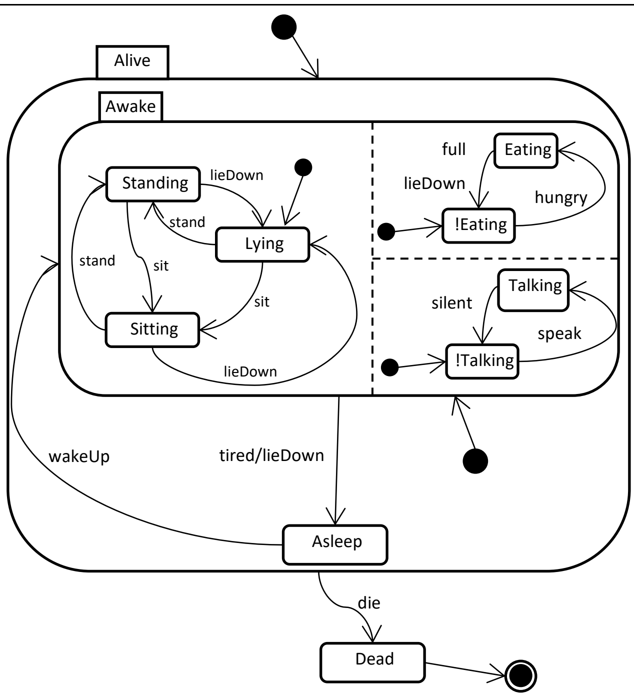
</td><td>
If you are trying to model some source code that involves throwing an exception, the first question you should consider is: “Do I really need to show this detail?” In the case that your answer is “yes” and you have not simply made a mistake in that decision, the question becomes: “How can I model this?”
Bad news: UML was mostly developed before exceptions became a standard construct, so it mostly does not consider them.
Good news: We can adapt what UML provides to arrive at a solution that works. This is required syntax for the course.
To achieve this, we can make use of the break fragment plus modify the message return label’s syntax a bit. The break fragment halts the normal execution, and the specialized message return allows the exception to be passed to the caller.
The specialized message makes use of an out parameter, an actual part of the UML specification that allows more than one return value to be passed. Here, I have set the (unique) out parameter to the name referencing a newly created exception. The guard checks for “problem” which would normally be a bit more specific about the nature of the problem, but this suffices for this example. Note that the exception is not shown as being destroyed since we don’t know what happens to it; also, in a real situation, a type more specific than “Exception” is likely to be used.
(Strictly speaking, the “isException” property of the parameter should be “true”, but showing that would be a waste of space.)
</td></tr>
</table>

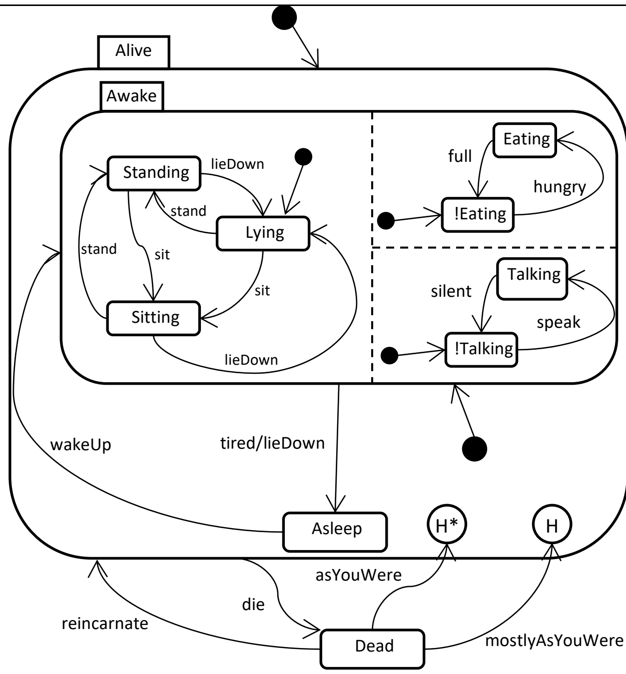
Name	Notation	Notes
exception handling		Exception handling, provided in Java through its try/catch construct, is not well supported in UML. **To deal with it, I will introduce some strictly unofficial syntax that is required in the course.** I define a try fragment, which has two or more compartments. The top compartment executes by default, but if an exception occurs therein, execution stops; the other compartments (one or more) specify which kinds of exception each handles. When the exception type matches the guard of the compartment, that compartment then executes; if no exceptions occur therein, execution then jumps to right after the try fragment. In this specific example, the catch compartment ignores the exception, but arbitrary behaviour could be placed therein just like in Java code. I show the exception role as though it always exists, whether or not an object is assigned to it. (Note: showing this with official syntax would require a break fragment after every method call, in case *that* call ended up causing an exception. With a non-trivial piece of source code to model, that would quickly become a redundant mess.)

State Machine Diagrams

Students have a lot of difficulty with state machine diagrams. State machines are all about state, the events that occur to cause them to change state, and the reactions they have as a result. They are not a kind of “flowchart”; they do not represent the steps in a computation. And the ones we will tend to encounter run without end, and thus they have no “final state”.

Name	Notation	Notes
state machine frame	 	Just like with sequence diagrams, it is sometimes useful to embed a state machine inside a frame that explicitly shows the boundaries of the machine. The pentagon now contains “stm” (for “state machine”, usually boldface) and some sort of name for the state machine contained therein.
state	  	The specification states: “A State models a situation in the execution of a StateMachine Behavior during which some invariant condition holds. In most cases this condition is not explicitly defined, but is implied, usually through the name associated with the State.” A state is an invariant property or set of properties about something. **A state is <u>NOT</u> a step in a computation; if you model states incorrectly like this, you can expect a D-range grade on the grading item in question.** A state is represented as a rectangle with rounded corners. It is not a class; it is not an object. The rounded corners are the only signal that this is a state, without examining the rest of the context of the diagram. We give states names to represent what the invariant property is while the state machine is in that state. If we think the state needs to change sometimes, we need to show what causes the state to change (the *event*), how the state changes (the *transition*), and anything that happens as a result of the change of state (the *reaction*). Given multiple states, the state machine keeps track of which state it is currently in. You can think of this as a single token (a coin, a poker chip, a peanut, or however you like to think of it) that moves around the state machine. There is only one such token, normally, and so the state machine is only in one state in any moment or in the act of transitioning between two states along one specific *transition*, which we will examine later. States may optionally be shown with a “tab” at their top left (just back from the curve of the rounded corner) in which is shown its name; this is usually only useful if other details (like *regions* in a *composite state*, q.v.) occupy the space inside the rounded rectangle, but it can be used at any other time too.
event (officially: trigger)	alarmRings()	An event is something that causes our state machine to change state, to react in some way, or both. What events get modelled depends on the nature and purpose of the model, but they are generally something that happens *outside* the state machine. An event will typically be named with some meaningful label; there is no formal syntax involved, but you need to avoid using labels that will confuse the reader with the syntax around them (see the details of transition labels). When modelling events in a software system, we will often be interested in modelling when method calls occur. If a method is called “foo” and takes three arguments, we don’t want to waste a bunch of space saying, “When foo is called with argument #1, argument #2, and argument #3.” Typically, we would shorten this to something like “foo(a, b, c)” to allow us to add a guard regarding the value of the arguments, or simply “foo()” or “foo(-)” if we are not interested in those details. You will find that lack of physical space, in which to represent your model, quickly becomes a problem, so being succinct is desirable.
reaction (officially: behaviour expression)	shutOffAlarm()	A reaction is the response of a state machine to an event, in addition to transitioning to another state. Something causes the state machine to react (an *event* [or *trigger*]) but only if the right conditions are satisfied (the *guard*); in addition, the state machine’s state may change (via a *transition*) but again only if the right conditions are satisfied. Just like with events, there is no formal syntax for reactions. Be succinct here too.
[state] transition	  	A transition is the unique means for a state machine to move between states. Something causes the transition to occur (an *event*) but only if the right conditions are satisfied (the *guard*); as a result of the transition being taken, the state machine may do something beyond simply transitioning (the *reaction*). A transition is shown as a solid arrow with an open arrowhead. Unofficially, multiple, sometimes complex, labels can adorn each transition; a transition WITHOUT a label is immediately taken by the state machine, and that is rarely what you want to say. When a transition has one or more labels, the transition is taken WHEN an event occurs that is specified in any of its labels AND the guard specified in the *same* label evaluates to true; the reaction specified in that label is then performed and the transition completes.

Name	Notation	Notes
transition label	*blank* <*event-descriptor*> "[" <*guard*> "]" "/" <*reaction*> <*event-descriptor*> "[" <*guard*> "]" <*event-descriptor*> "/" <*reaction*> "[" <*guard*> "]" "/" <*reaction*> <*event-descriptor*> "[" <*guard*> "]" "/" <*reaction*>	Every transition is labelled with one of the eight forms shown here (including the blank one, which means that there is no label). In all cases, the transition is taken by the state machine if and only if the label on it contains the event descriptor of an event that has just happened AND it contains a guard that evaluates to true; a missing event is taken to occur constantly; a missing guard is taken to always be true. Once the machine has recognized that the transition is to be taken, the reaction specified in the label (if present) is performed and the transition is taken. Officially, a transition only has one label, with zero or more triggers (comma-separated), zero or one guard, and zero or one behaviour expression. However, this can lead to excessively complex guards in an attempt to narrow the context to which they each apply. **In this course, you are permitted to use arbitrarily many labels on a single transition, but you have to be careful to avoid visual ambiguity.**
activity		An activity is a behaviour that is not necessarily instantaneous, as opposed to an event which occurs at an instant in time. (Note that UML contains Activity Diagrams for modelling the modern equivalent to the classic flowchart, but we will not be looking at these in this course.) In this course, the use of activities will be strictly limited; each state can (but does not have to) define up to three special activities, labelled "entry", "exit", and "do". The "entry" activity occurs when the state is first entered; "exit", when the state is left; and "do", while the state machine remains in that state. A given state can define any or all of these three activities, depending on what makes sense there; don't overuse them. The only other situation where an activity designation would be acceptable is as a replacement for a self-loop transition. In this example, the trigger "hungerPang" doesn't cause a transition but it does cause a reaction. This could also be modelled as in the second example. In both cases, a guard can be added too; they cannot be added to "entry", "exit", or "do" activities.
pseudostate		A pseudostate is not a real state ("pseudo-" comes from the Greek word meaning "false"), but an annotation to control various aspects of a state machine. For us, every state machine will possess a start pseudostate (a solid black circle), with a transition to a real state that is the initial state of the state machine. (More than one transition is possible from the start pseudostate, but only if these are guarded unambiguously.) A final pseudostate is also possible (a solid black circle surrounded by an empty black circle), but of less interest to us because things we will model will not generally stop: if the final pseudostate is reached, the state machine stops (or the portion thereof, in the case of *compound states*). History pseudostates are also possible, but they are only useful in conjunction with complex state machines using regions; we look at this next.
composite state		Sometimes our states are actually compound, representing the juxtaposition of multiple properties. For example: awake, lyingDown, eating, talking are largely independent properties that could be used to model a person. Imagine that each of these properties has two possible values; the total combinations would be 2^4=16 and so 16 independent states would be needed to show them. Instead, we can show how the state machine works with the four properties as "mini-state machines" running in parallel. These parallel, mini-state machines are each represented by a *region.* Either a whole state machine can have a set of regions (2 or more, to be meaningful) or an individual state can itself be *compound*, meaning that it internally contains regions (1 or more). Regions are separated from each other with a dashed line. The whole space of the enclosing state or state machine is to be *tiled* with the regions, meaning that the regions occupy all the space therein (their sizes can be adjusted for convenience). The text compartment for the whole state can be shown in a "tab" at the top left of the state symbol; for a whole state machine. In principle, each region can possess a name, but this is not usually useful. In this example, we see two states ("Alive" and "Dead"), one of which is compound ("Alive", with exactly one region), being composed of two other states ("Awake", also compound, with three regions; and "Asleep", simple). Imagine that our state machine is currently in the "Alive" state; we need to know which of its internal states is the current one: let's say "Asleep" and then the "wakeup" event occurs; this causes a transition to the compound "Awake" state. Each of the regions of "Awake" then moves into its local start state ("Lying", "!Eating", and "!Talking", respectively). Additional events can then take place (specifically, "sit", "stand", "lieDown", "full", "hungry", "silent", "speak") causing internal transitions within those regions. If, at any point, a "tired" event occurs, our state machine leaves the "Awake" state, transitioning to the "Asleep" state, but first reacting by lying down (if it was already in the "Lying" state, it is not clear that it makes sense to lie down again, but we could avoid the redundant behaviour by either adding a guard, adding an entry activity to "Asleep" which would test if the system is lying down, or similar mechanisms). Likewise, if at any time the "die" event occurs, the state machine moves to the "Dead" state, which is final: no reincarnation here. Notice that the transition from "Eating" to "!Eating" can occur due to either of two events: "full" or "lieDown". You shouldn't eat when lying down, but notice if you stay lying down and you are hungry, you start eating again anyways!

Name	Notation	Notes
		Notice that none of the regions of "Awake" possesses an end state in this example, and each region supports infinite looping. The next time that the state machine enters the "Awake" state, it does not remember where it had been before, but its regions each move to its local start state again. In this particular model, that behaviour probably makes sense, but in some cases it would not. To remember the previous configuration, we can use a mechanism called *history*.
history pseudostate		History pseudostates allow us to remember, for a compound state, "where we were" the last time that state was exited. There are two variations, *shallow history* (marked with "H") and *deep history* (marked with "H*"). Consider the example shown here. "Alive" is a compound state that contains "Awake", another compound state. We might want to remember the current state within "Alive" if we leave it as well as the state of "Awake"; deep history achieves this, and this is the effect when the transition labelled "asYouWere" is taken. On the other hand, we might want to remember the current state within "Alive" but not within "Awake"; shallow history achieves this, and this is the effect when the transition labelled "mostlyAsYouWere" is taken. Notice the difference when taking the transition labelled "reincarnate": there is no memory of the previous state, and so the initial states are used to configure the compound states.
other constructs		**In this course, you MAY NOT use the various other constructs provided for state machines by the UML specification; they will be treated as INCORRECT in all circumstances.** This is to avoid the kinds of mistakes that I see most students making; the permitted syntax should suffice for your needs. The forbidden constructs include *submachine state*, *state list*, *fork*, *join*, *choice*, *junction*, *entry point*, *exit point*, *terminate pseudostate*, *action symbol*, *signal receipt symbol*, *signal sent symbol*, *choice point symbol*, *merge symbol*, *deferred trigger*, and *state machine redefinition*. Similarly, our state machines are strictly behaviour state machines, so all the syntax and semantics of protocol state machines are not permitted.