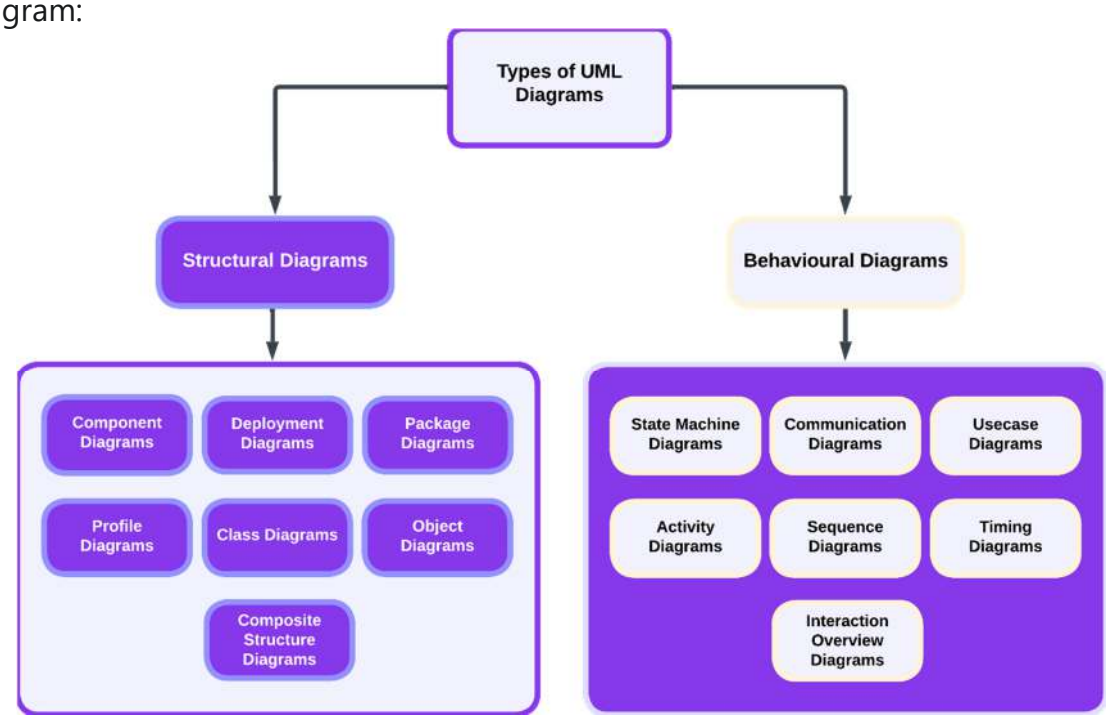
Unified Modelling Language Diagrams

Unified Modelling Language Diagrams (UML for short) are a way to illustrate complex design concepts. Whether you're trying to talk about something as complex as enterprise systems architecture or something as simple as a class in a Java program, UML can help you do that. UML helps us define what entities are involved in a system, what details we need to know about them, and how they relate to each other through the use of a simple visual language. You've seen quite a few of them throughout the camp here, but let's get into the details before we move on.

#Types of UML Diagrams

Here are all of the currently recognized types of UML Diagrams displayed as a UML diagram:



A UML Diagram about UML Diagrams

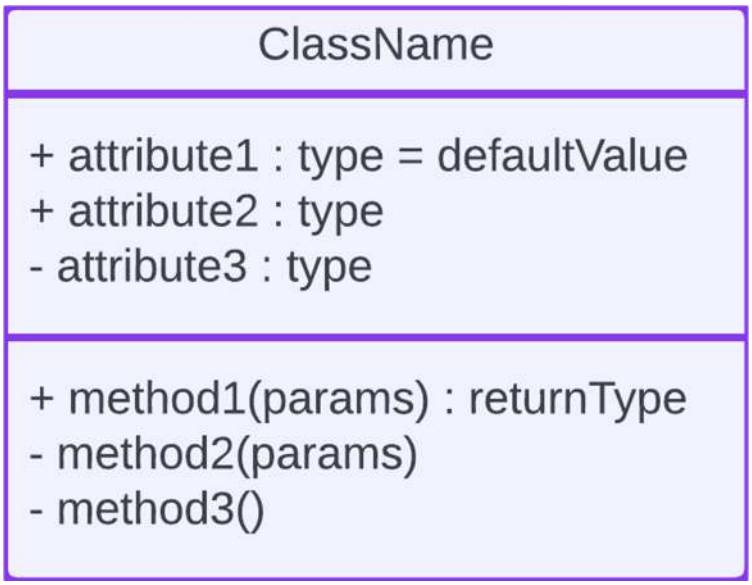
There's a lot to see here, but you can breathe easy: we're only going to focus on the Class Diagram for now. If you'd like to read more about any of the other types, go here for a great visual breakdown of each one.

#The Class Diagram

As you've recently learned, classes are an integral part of the OOP process. They aid us with all four of the pillars of OOP and form the backbone of our Java applications by defining the parameters that our objects will follow.

While classes are super helpful, they can also get a little confusing. Remembering which class is responsible for which attributes or methods can cause some issues, especially in a large project with many developers. When you start to think about how each object instantiated from a class might interact with other objects from other classes, you can see how the problem might get compounded. Now try to imagine yourself explaining all of these intricate interconnections to a new teammate on their first day. Our IDEs can help us keep track in some ways (try holding `ctrl` and clicking a class name), but sometimes it's best just to have a diagram that lays it all out for you.

The Class Diagram does that for us. Here's a lightly modified default Class Diagram from LucidChart:



A Class Diagram

Let's break it down:

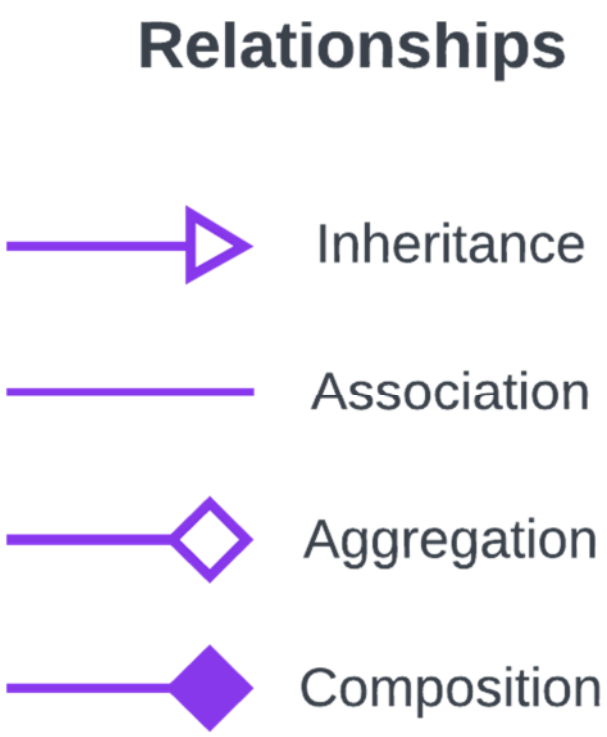
- `ClassName` – This is simply where you put the name of the class.
- `attribute1 : type = defaultValue` – This is the most complex version of an attribute that you will put in a Class Diagram. When you make your own, you'll replace `attribute1` with the name of the attribute, then define the `type` of the

variable after a colon `:`, and lastly list the default value following the assignment operator `=`. Each of the lines that follows for `attribute2` and `attribute3` are equally valid–we simply use them when we don't need as much information. If there's no default value to speak of, simply don't write one.

- `+` or `-` – Along with # and ~, these symbols represent the visibility of the attribute:
  - `+`: a visibility setting of public means that any other class can access this attribute.
  - `-`: a visibility setting of private means that these attributes are only available to this class itself.
  - `#`: a visibility setting of protected is somewhere in between private and public–it means that these attributes can be accessed by this class or its subclasses, but nothing else.
  - `~`: a visibility setting of package/default means that any other class within the same package can access this attribute.
  - Of the visibility options above, `+` and `-` are the most commonly used, but don't be surprised if you see the others out in the wild.
- `method1(params) : returnType` – As with attribute1 above, this is the most complex version of a method you might see in a Class Diagram. This version features the `(params)`, short for parameters, that the class is expected to take in, as well as the `returnType` for whatever data it expects to return. Note that the params list can be as long as it needs to be, and is sometimes listed with both the type and the name of the parameter, as in `(num1 : int)`.
- Methods make use of the same visibility settings, though they're typically set to public `+` because most methods need to be exposed to the system in order to be used.

#Relationships

Once you've made a few classes, you're going to want to start drawing out the ways that they're related. Here are the UML diagram relationships for Class Diagrams:

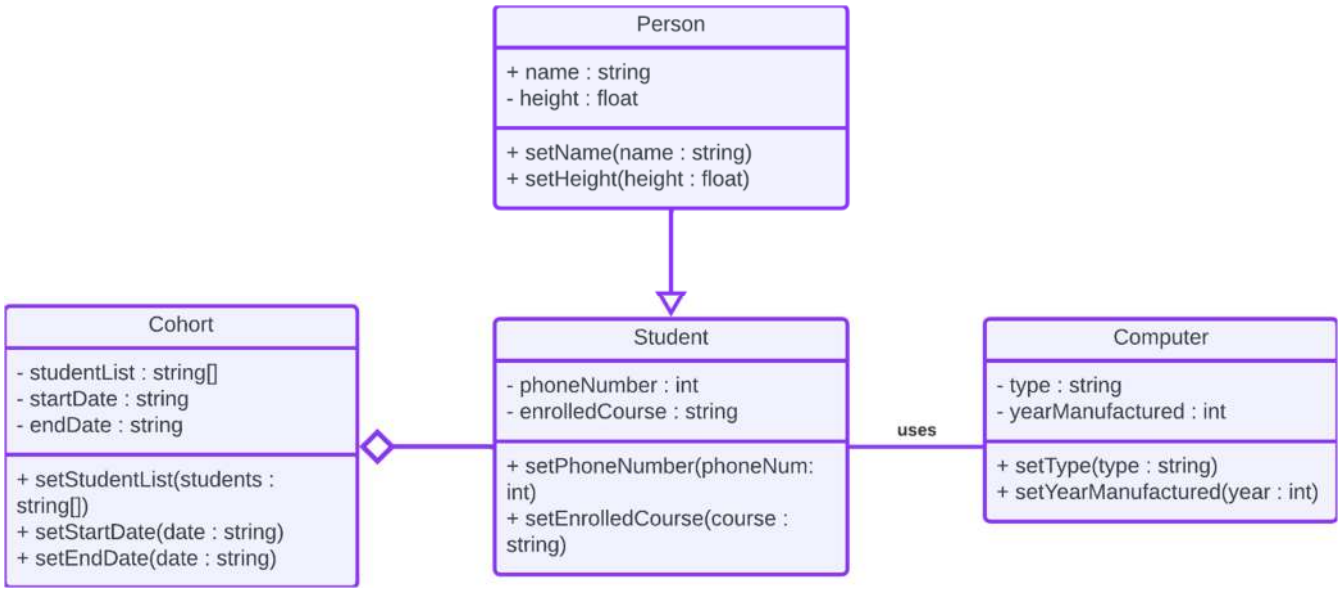


The four relationships

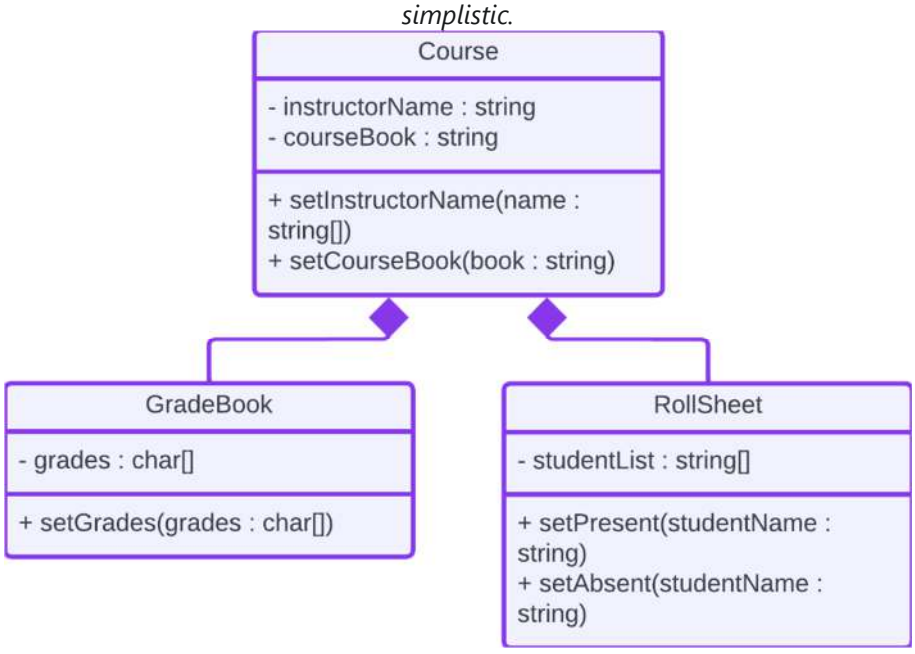
Let's go through them one at a time:

- **Inheritance** is exactly what it sounds like: it shows which subclasses (aka child classes) inherit from a superclass (aka parent class). This keeps us from having to write out lists of attributes/methods that a group of classes share. Note that this is an empty arrow, not a filled in one, just as aggregation is an empty diamond. These differences are often very important.
- **Association** is used to connect classes based on other kinds of relationships, such as connecting a `student` class to a `computer` class to show which kind of computer they use. Without a `student` to own the `computer`, we may still need to know about the computer class for our `lessonCreation` class or something similar. There's no inheritance involved here, but we can connect the two with a simple line to show association. We can also write the relationship on the line we draw, such as "uses", if we so choose.
- **Aggregation**, which is just a fancy word for "adding up", is used to show a relationship between more singular entities and what they combine to form. For example, you can have a `student` class without a `cohort`, but a `cohort` can't exist without a number of students. In this case, the open diamond would be drawn on the `cohort` class, with a simple line leading to the `student` class.

- **Composition** is the opposite of aggregation, and it's easiest to think about in terms of "these parts compose the whole, but they can't exist without the whole." To continue with our student example, without a `course` class, a `grades` class, or a `rollSheet` class wouldn't exist.



Inheritance, aggregation, and association are shown in the image above. Note: examples are kept simplistic.



Composition is shown in the UML diagram above. Note: examples are kept simplistic.

- Class diagram (class name at top, instance variables, methods… return types for methods, **the constructor**… model that. That's the base level.

#Extra Resources

When you have some extra time, feel free to watch this video about how to make your own diagrams. It's a bit of an ad for LucidChart, but there's a free tier of that service, and it works pretty well for the purpose.

Open in new tab
https://youtu.be/UI6lqHOVHic

If you'd like to learn more, including a few other relationship lines, check out this tutorial here.

Back
Unsubmit
Next