**UML TUTORIAL**

Inheritance:

We have the “Animals” superclass and Tortoise, Otter and Slow Loris as child classes. Instead of writing the same attributes to all child classes we just write them once at the superclass and we inherited the child classes with the Parent class with that white arrow.



Association:

So, if we had a class for Sea Urchin we could draw an association which is just a simple line between Otter and Sea Urchin. And we could say Otter eats Sea Urchin. There is no dependency between them, it’s just a basic association relationship and it’s simple.



Aggregation:

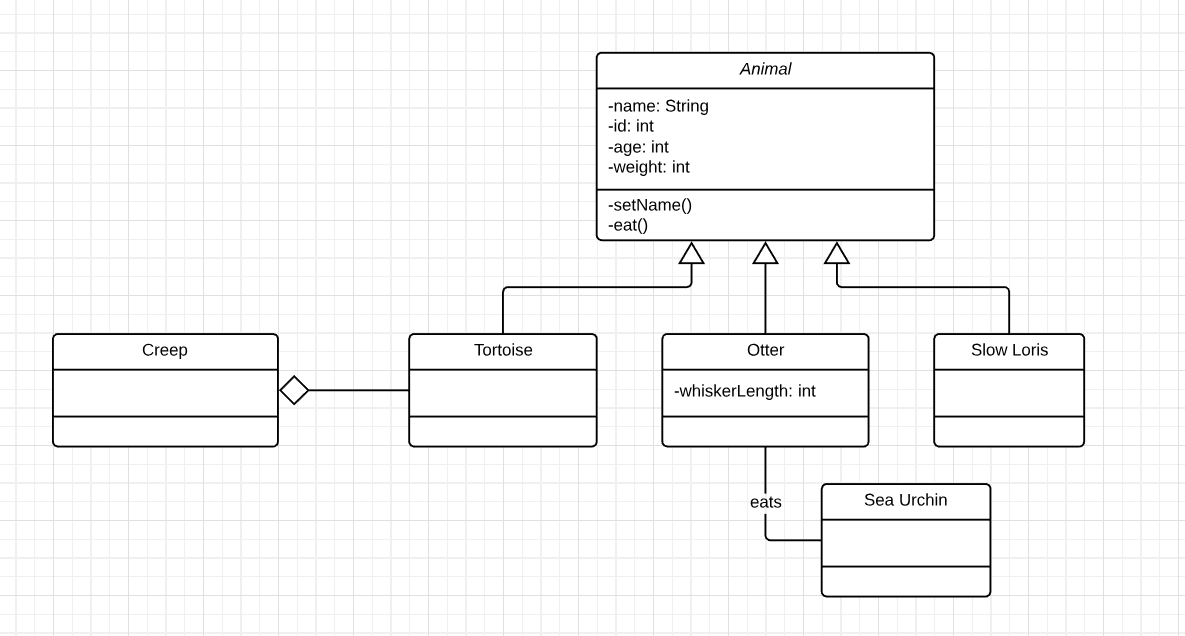
A new class for group of tortoises “Creep”, got a relationship with Tortoise, any of our Tortoises could be a part of a creep, but they don’t have to be. A tortoise could leave the creep at any point and still exist on its own. That type of relationship, where a part can exist outside the whole, is aggregation.



Composition:

There is also a relationship where a part can’t exist outside the whole. It called composition. Let’s say we have a superclass “Visitor Center” which it has 2 subclasses “Lobby” and “Bathroom”. Now if one of the “Visitor Center” was torn down, the lobby and the bathroom of that visitor center would be destroyed as well. Those rooms couldn’t exist apart from the Visitor Center that they’re in. That’s composition. When a child object wouldn’t be able to exist without its parent object.

Multiplicity:

Our Visitor Centers are going to have just one lobby, we simply write the number “1”, near Lobby table, meaning that can be one and only one lobby per visitor center. For bathrooms there’s at least one bathroom per Visitor Center, but leave the option to have as many as you’d like (1..\*) one to many bathrooms.

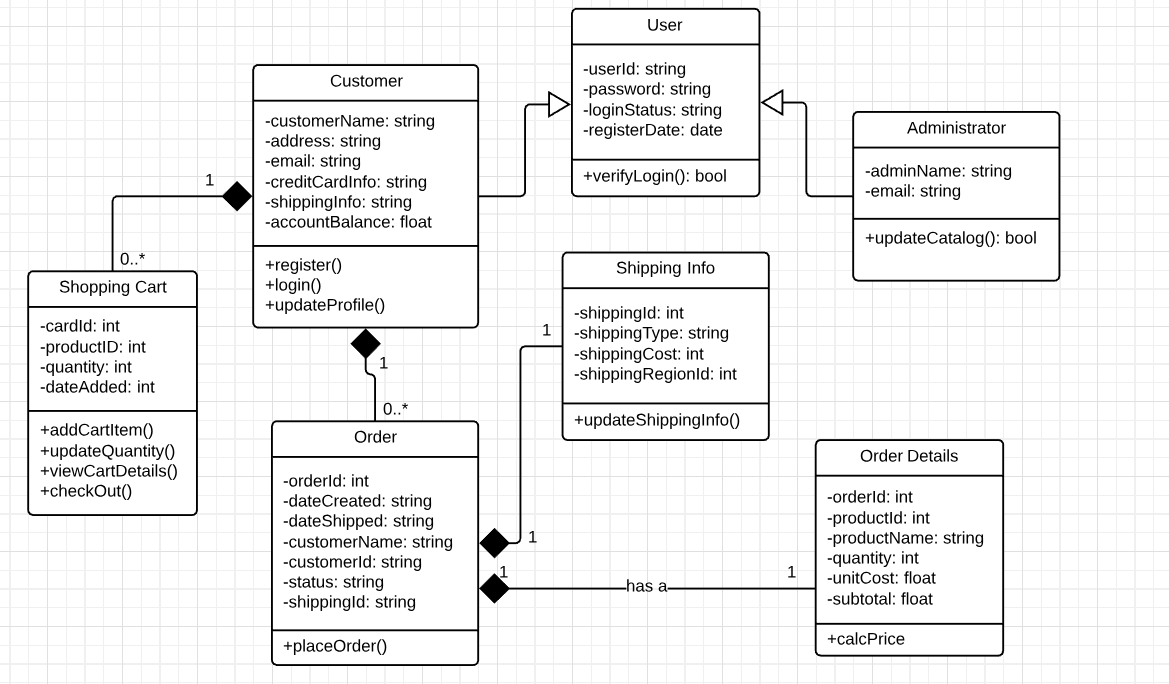
0..1 zero to one (optional)

n specific number

0..\* zero to many

1..\* one to many

m..n specific number range



You can see that a customer can have a zero or many orders. You could create a customer account for an online store but never buy anything or you could be a frequent customer and place several different orders. And on the flip side an order can belong to only one customer. It’d be confusing if a specific order with a unique order ID was duplicated across several different customers.

Order #3500302 - > Customer A, Customer B, Customer C, Customer D… ????? Wrong

Order #3500302 -> Customer A

Order #3500303 -> Customer B

Order #3500304 -> Customer C

Order #3500305 -> Customer D …….

**USE CASE DIAGRAMA**

We are going to break down Use Case diagrams into four different elements:

Systems

Actors

Use Cases

Relationships

System:

A system is whatever you are developing. It could be a website, a software component, a business process, an app, or any number of other things. You represent a system with a rectangle. We are going to build a use case diagram for a very simple Banking Application. The rectangle helps define the scope of this system. Anything within the rectangle happens within the Banking App.

Actor:

The next element is an actor, which is represent with a stick figure. An actor is someone or something that uses our system to achieve a goal. That could be a person, an organization, another system, or an external device. First, actors are external objects, they always need to be placed outside of our system. Second, there are 2 types of actors. Primary and Secondary Actors. A primary actor initiates the use of the system while a second actor is more reactionary. The Primary Actor of this example is Customer, because the customer is going to initiate the use of our system. They are going to pull out their phone, open our Banking App, and do something with it. Bank on the other hand is a secondary actor, the Bank is only going to act once the Customer does something. Primary actors should be to the left of the system, and secondary actors should be to the right.

Use Case:

A Use Case is depicted with an oval shape and it represents an action that accomplishes some sort of task within the system. They are placed within the rectangle because they’re actions that occur within the Banking App. So, the banking app does:

Log In

Check Balance

Transfer Funds

Make Payment

Relationships:

Our Customer is going to Log In to our Banking App. So, we draw a solid line between the actor and Use Case to show this relationship. This called association and it just signifies a basic communication or interaction. We don’t need to draw a line to Log In (from Bank actor) because that process happens within the Banking App. There’s no need for the Bank to get involved with the login process. Other types of Relationships is Include, Extend and Generalization.

When a customer types in their login information, our Banking App is going to verify the password before completing the login process. But if the password is incorrect, the Banking App is going to display an error message. Every time a Customer Logs In, our Banking App will automatically Verify Password. This Log In use case won’t be complete unless Verify Password is complete. To an extended use case out Banking App won’t display a Log In Error Message every time a Customer logs in. This will only happen once in a while when a Customer accidently enters an incorrect password. Generalization, also known as inheritance, when you Make a Payment from our Banking App, you can do so from either your checking account or your savings account.

