# OOPS

OBJECT ORIENTED PROGRAMMING

# Object ORIENTED DESIGN

TOPICS

OBJECT-ORIENTED

OBJECT-ORIENTED ANALYSIS

## OBJECT-ORIENTED

they are models of somethings that can do certain things and have certain things done to them. Formally, an object is a collection of data and associated behaviors.

So knowing what an object is, what does it mean to be object-oriented? Oriented simply means directed toward. So object-oriented simply means, "functionally directed toward modeling objects.

It is one of many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior

## OBJECT-ORIENTED ANALYSIS

Object-oriented Analysis (OOA) is the process of looking at a problem, system, or task that somebody wants to turn into an application and identifying the objects and interactions between those objects. The analysis stage is all about what needs to be done. The output of the analysis stage is a set of requirements. If we were to complete the analysis stage in one step, we would have turned a task, such as, "I need a website", into a set of requirements, such as: Visitors to the website need to be able to (italic represents actions, bold represents objects):

- *review our* **history**
- *apply for* **jobs**
- *browse, compare, and order our* **products**

## Object-oriented Design (OOD)

It is the process of converting requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify what objects can activate specific behaviors on other objects. The design stage is all about how things should be done. The output of the design stage is an implementation specification. If we were to complete the design stage in one step, we would have turned the requirements into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language

## Object-oriented Programming (OOP)

It is the process of converting perfectly defined design into a working program.

In the fast-paced modern world, most development happens in an iterative development model. In iterative development, a small part of the task is modeled, designed, and programmed, then the program is reviewed and expanded to improve each feature and include new features in a series of short cycles.

## Objects and classes

An **object** is a collection of data with associated behaviors

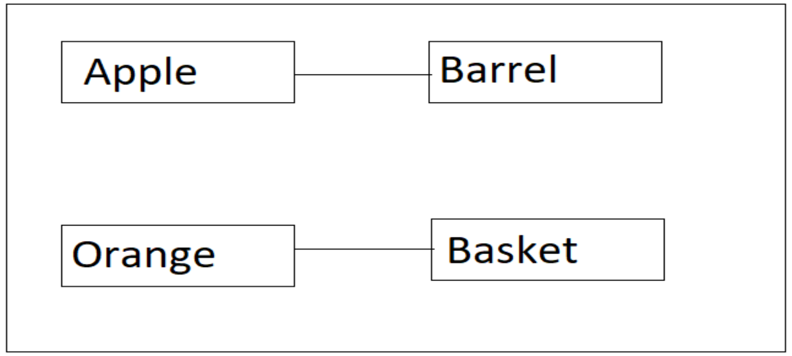
**Classes** describe objects. They are like **blueprints** for creating an object.

Example:

Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm! As an example, we can assume that apples go in barrels and oranges go in baskets. We have four objects such as Apple, Barrel, Orange and Basket.

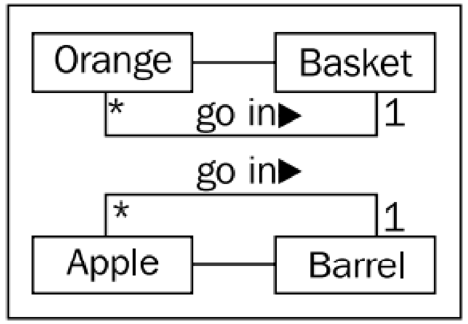
| Apple | Barrel | Orange | Basket |
|---|---|---|---|
| | | | |

The relationship between the four classes of objects in our inventory system can be described using **a Unified Modeling Language class diagram**. Here is our first class diagram



This diagram simply shows that an Orange is somehow associated with a Basket and that an Apple is also somehow associated with a Barrel. Association is the most basic way for two classes to be related.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in.



In the case of apples and barrels, we can be fairly confident that the association is, "many apples go in one barrel", but just to make sure nobody confuses it with, "one apple spoils one barrel", we can enhance the diagram as shown above:

## Specifying attributes and behaviors

We now have a grasp on some basic object-oriented terminology. Objects are instances of classes that can be associated with each other. An object instance is a specific object with its own set of data and behaviors; a specific orange on the table in front of us is said to be an instance of the general class of oranges. That's simple enough, but what are these data and behaviors that are associated with each object?

# Data describes objects

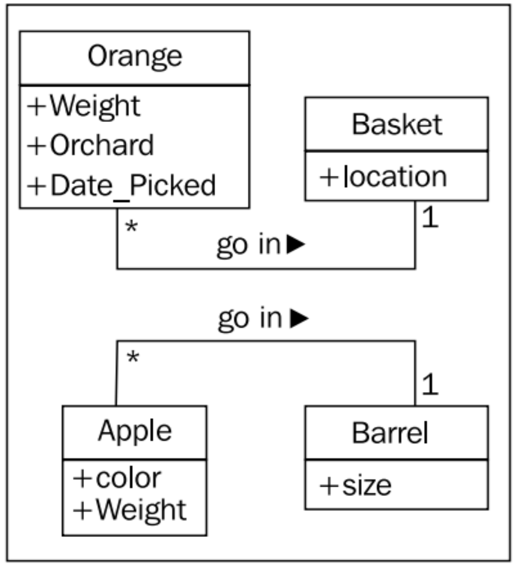
Data typically represents the individual characteristics of a certain object.

A class of objects can define specific characteristics that are shared by all instances of that class.

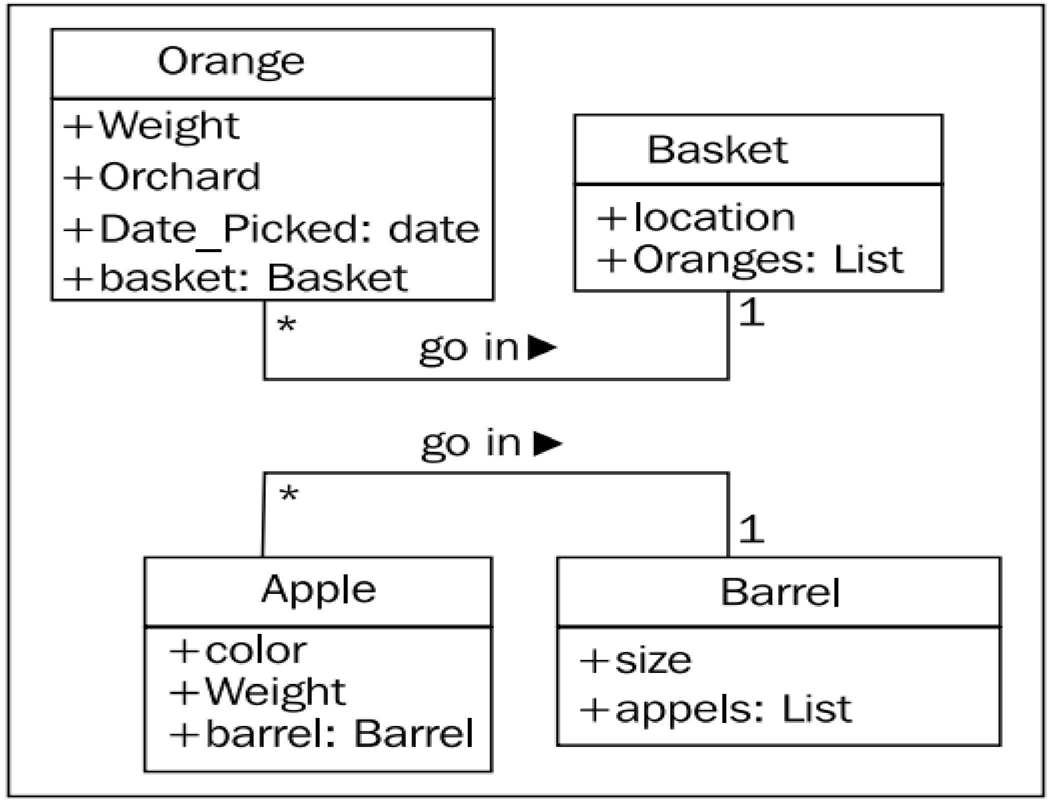
The orange class could then have a weight attribute. All instances of the orange class have a weight attribute, but each orange might have a different value for this weight.

Attributes are frequently referred to as properties

We have refine our class diagram and added attributes:



There are implicit attributes that we can make explicit: the associations. For a given orange, we might have an attribute containing the basket that holds that orange. Alternatively, one basket might contain a list of the oranges it holds. The next diagram adds these attributes as well as including type descriptions for our current properties:



## Behaviors are actions

The behaviors that can be performed on a specific class of objects are called methods.

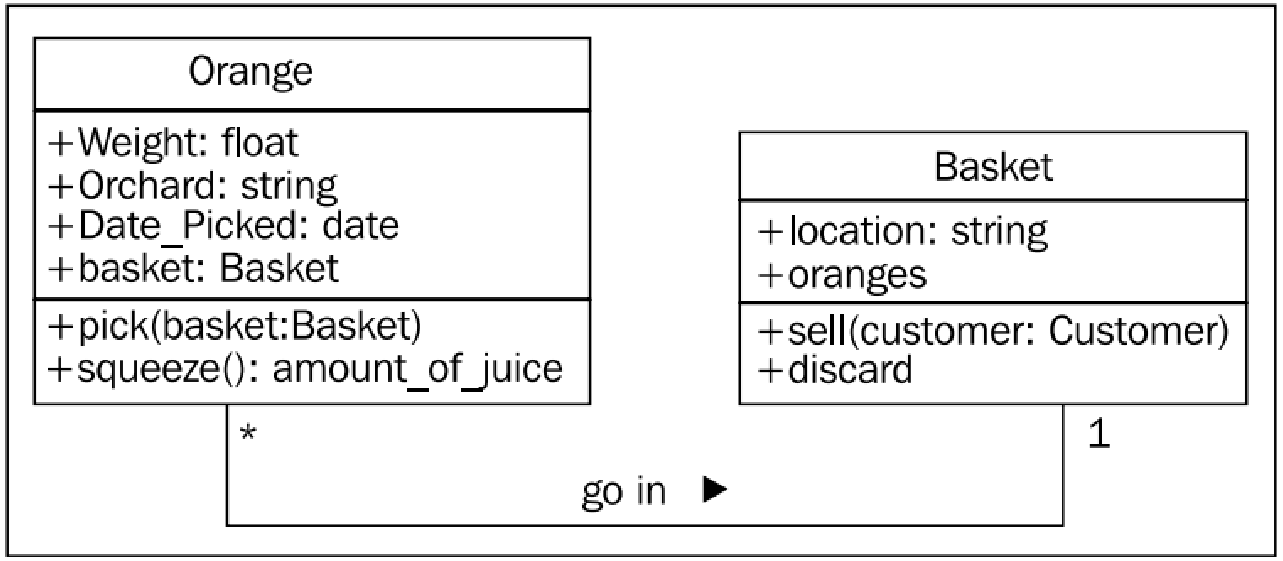
At the programming level, methods are like functions in structured programming, but they magically have access to all the data associated with that object. Like functions, methods can also accept parameters, and return values.

One action that can be associated with oranges is the pick action. If you think about implementation, pick would place the orange in a basket by updating the basket attribute on the orange, and by adding the orange to the oranges list on the Basket.

We can add a squeeze method to Orange. When squeezed, squeeze might return the amount of juice retrieved, while also removing the Orange from the basket it was in.

Basket can have a sell action. When a basket is sold, our inventory system might update some data on as-yet unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we add a discard method. Let's add these methods to our diagram

Adding models and methods to individual objects allows us to create a system of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other objects of the same class, and each object may react to method calls differently because of the differences in state



Object-oriented analysis and design is all about figuring out what those objects are and how they should interact. The next section describes principles that can be used to make those interactions as simple and intuitive as possible

## Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public interface of that object will be. The interface is the collection of attributes and methods that other objects can use to interact with that object.

A common real-world example is the television. Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object.

This process of hiding the implementation, or functional details, of an object is suitably called **information hiding.**

It is also sometimes referred to as **encapsulation**, but encapsulation is actually a more all-encompassing term. Encapsulated data is not necessarily hidden.

The public interface, however, is very important. It needs to be carefully designed as it is difficult to change it in the future. Changing the interface will break any client objects that are calling it.

## Abstraction

Abstraction is another object-oriented buzzword that ties in with encapsulation and information hiding. Simply put, abstraction means dealing with the level of detail that is most appropriate to a given task. It is the process of extracting a public interface from the inner details.

Abstraction is the process of encapsulating information with separate public and private interfaces. The private interfaces can be subject to information hiding.

Note: *When designing the interface, try placing yourself in the object's shoes and imagine that the object has a strong preference for privacy. Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give them an interface to force you to perform a specific task unless you are certain you want them to be able to do that to you.*

# Composition and Inheritance

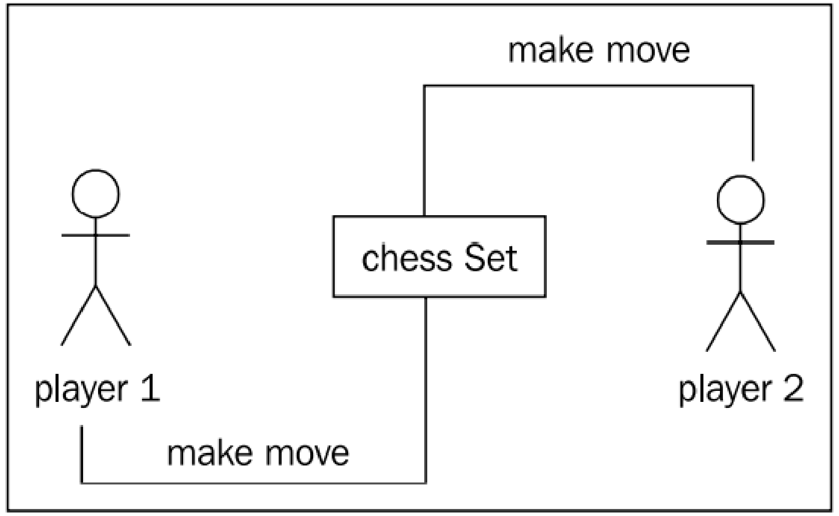
Composition is the act of collecting together several objects to compose a new one. Composition is usually a good choice when one object is part of another object.

Let's try modeling to see composition in action.

As a basic, high-level analysis: a game of chess is played between two players, using a chess set featuring a board containing sixty-four positions in an 8x8 grid. The board can have two sets of sixteen pieces that can be moved, in alternating turns by the two players in different ways. Each piece can take other pieces. The board will be required to draw itself on the computer screen after each turn

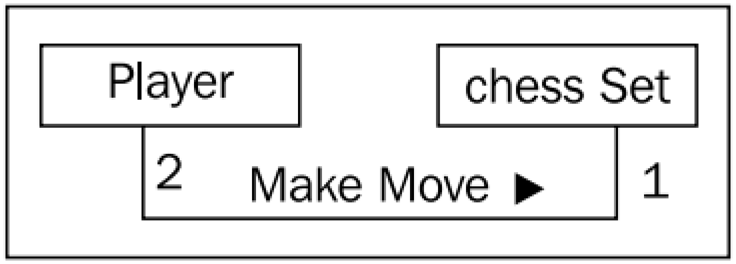
## At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

Let's start at the highest level of abstraction possible. We have two players interacting with a chess set by taking turns making moves.



This is an object diagram, also called an instance diagram. It describes the system at a specific state in time, and is describing specific instances of objects, not the interaction between classes.

Remember, both players are members of the same class, so the class diagram looks a little different:



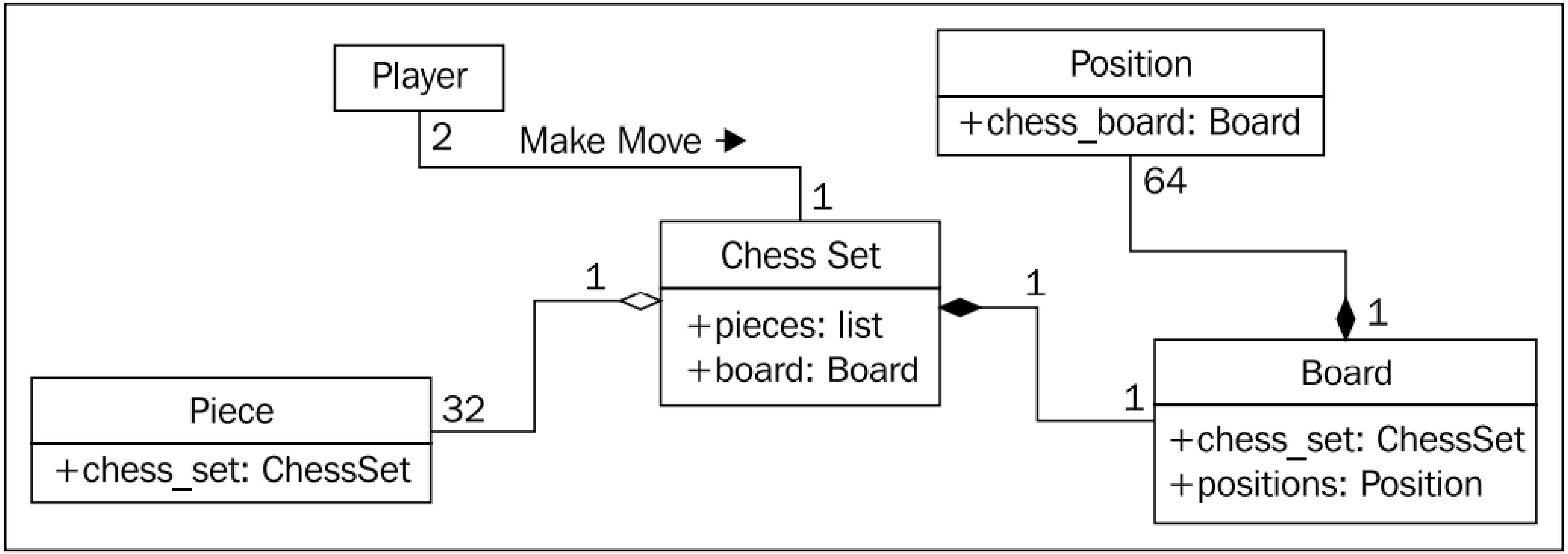
## Composition example

The chess set, then, is composed of a board and thirty-two pieces. The board is further comprised of sixty-four positions. You could argue that pieces are not part of the chess set because you could replace the pieces in a chess set with a different set of pieces.

## Aggregation

Aggregation is almost exactly like composition. The difference is that aggregate objects can exist independently. It would be impossible for a position to be associated with a different chess board, so we say the board is composed of positions. But the pieces, which might exist independently of the chess set, are said to be in an aggregate relationship with that set.

Also keep in mind that composition is aggregation; aggregation is simply a more general form of composition. Any composite relationship is also an aggregate relationship, but not vice versa.



THE COMPOSITION RELATIONSHIP IS REPRESENTED IN UML AS A SOLID DIAMOND

THE HOLLOW DIAMOND REPRESENTS THE AGGREGATE RELATIONSHIP.

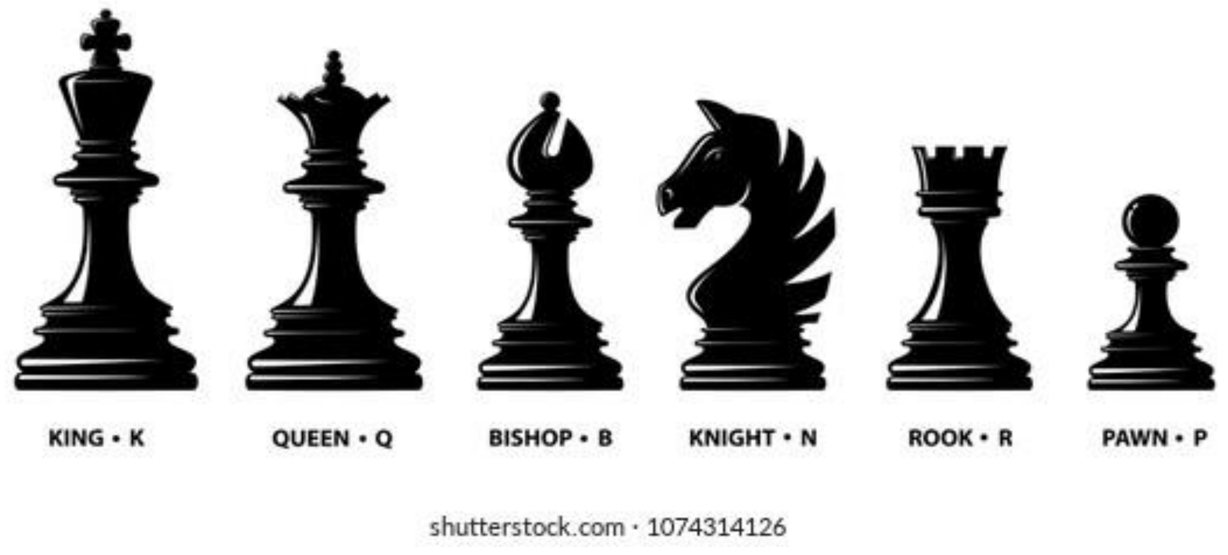
## Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

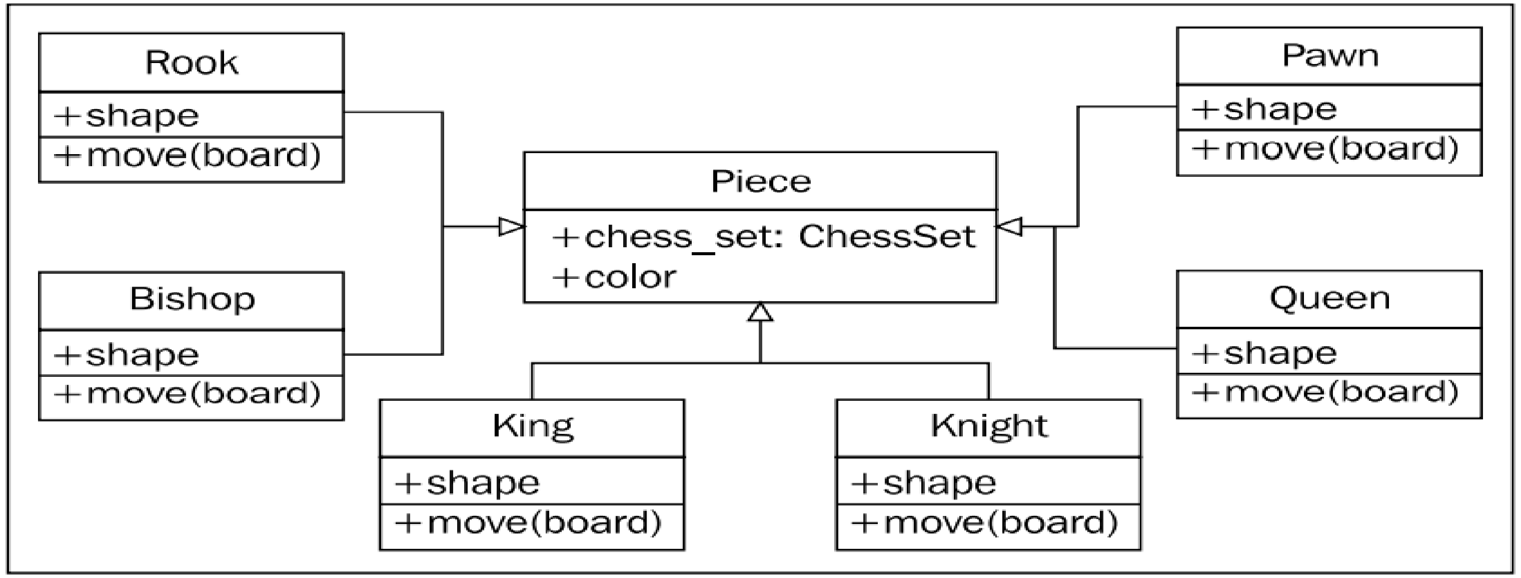
Inheritance is the most famous, well-known, and over-used relationship in object-oriented programming. Inheritance is sort of like a family tree. My grandfather's last name was Phillips and my father inherited that name. I inherited it from him. In object-oriented programming, instead of inheriting features and behaviors from a person, one class can inherit attributes and methods from another class.

For example, there are thirty-two chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen), each of which behaves differently when it is moved. All of these classes of piece have properties, like color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves. See how the six types of pieces can inherit from a Piece class

## Chess pieces



## Class diagram



The hollow arrows, of course, indicate that the individual classes of pieces inherit from the Piece class. All the subtypes automatically have a chess set and color attribute inherited from the base class. Each piece provides a different shape property (to be drawn on the screen when rendering the board), and a different move method to move the piece to a new position on the board at each turn.