Objects

Introduction

In this reading, objects will be discussed. Objects are the building blocks of all code and play a particularly important role in object-orientated programming (OOP). In this reading, the benefits of using objects will be outlined, as well as some important terminology relating to the use of objects.

Definition

An object is a programming concept that means that a structure has both state and behavior. Here, behavior relates to the object's ability to perform some action. As you progress through this course, you will witness many instances of objects exhibiting different behaviors. Concretely it relates to calling the methods of an object. An example of this might be calling the sort method of an array. This has the result of re-organizing the items of an array so that they are organized in relation to one another.

State pertains to the information about an object. Another word for this that you may be familiar with is object attributes. If you create a person class and you instantiate it to an object, an example of a state might be the person's age or name. The behavior then might relate to an action that is required of this person-object, like run or tackle.

Classes are commonly described as blueprints for an object. By extension, an object can be described as an instance of a class. The most common use of objects is in OOP, where code is encapsulated into objects, and these objects then interact with one another.

Example

One of the considerable strengths of using objects to instantiate a class is that you only need to create one template for how an object will act. Then you are free to create multiple instances of this class that can interact with one another. Consider a football team with 11 players. Only one class needs to be outlined that can hold varying speed, agility and work rate characteristics. This can seriously reduce the amount of code overhead in creating a football game. Characteristics like speed and agility are known as instance variables and can be said to relate to the state of the object. No unit outside of the object will share this instance. Two players can have a fitness instance. Changing this in one will not affect the fitness instance found within the other objects. You can change the instances of a class through the constructor or an internal method of the object. In Java, a typical instantiation of a class is as follows:

```
Player player1 = new Player(agility = 54, speed = 88, fitness = 90);

Player player2 = new Player(agility = 90, speed = 64, fitness = 83);
```

In this example, the player class creates two objects, `player1` and `player2`. The variables within the class are set differently depending on the values found within the soft brackets. There also needs to be a method to change the variable. Over the course of a match, a player may become tired. A method can be called to reduce the variable instance to reflect this.

```
player1.set_fitness(80)
```

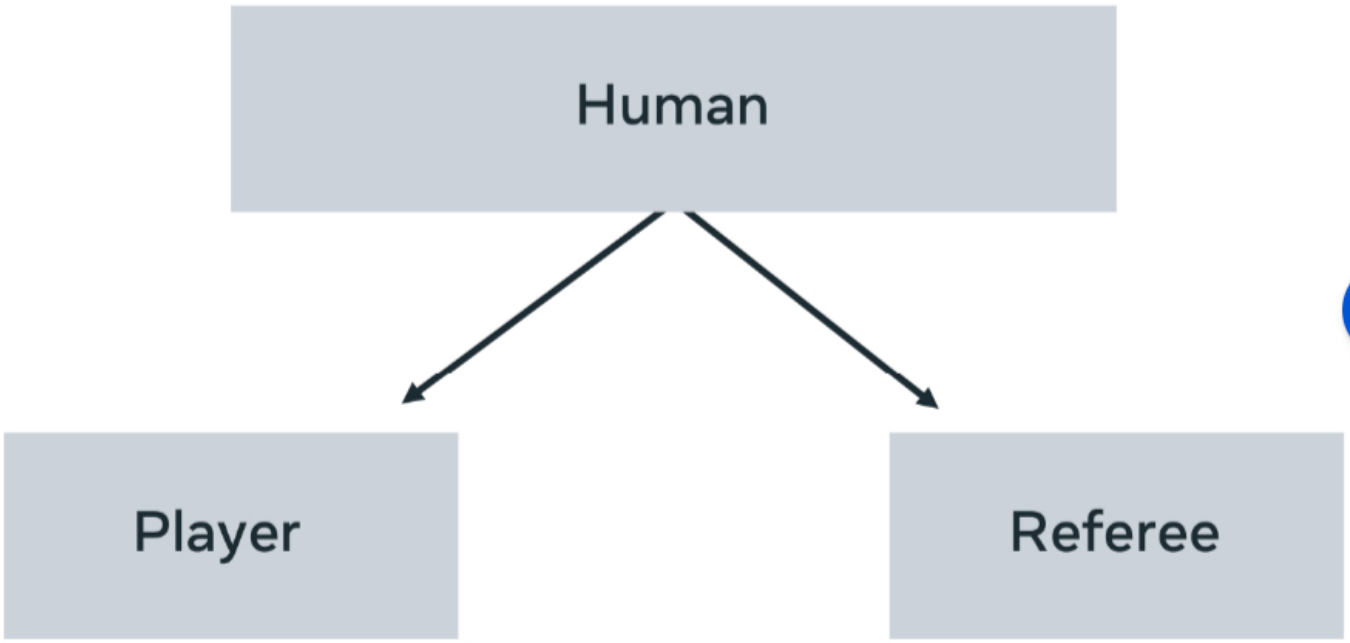
This will alter the fitness level of `player1` but will not affect the other players. The methods used to change attribute instances are generally called getters and setters.

Further control of the objects is available through the other methods found with the object. A command such as

```
player1.kick()
```

would cause the object assigned to `player1` to enact the procedure of kicking the ball.

The player class allows for the creation of a team with varying abilities that can play. Another concept related to the use of objects is inheritance. Imagine there is a need to create another human on the pitch, such as a referee, that does not play but still performs actions relating to the game, such as running and general appearance. You may want to reuse some of the code found in the player without creating a player. Here you would create a class that holds all common attributes you wish to retain, and each object can inherit them, as shown.



Thus, the standard methods such as run, get-tired, and follow the ball may all be found within `Humans`, but how these relate to the class `player` and class `referee` can be differentiated. This notion of having one general concept(`human`) that can manifest in different forms (players, managers, referees) is known as polymorphism. One shape, many forms. The classic example is the use of shapes. One overarching shape will contain the area, diameter, and height concepts. Then each instance of shape (square, triangle, circle) will apply the actual implementations of how these attributes look in their given state.

Conclusion

In this reading, the concept of objects was explored, with a particular focus on its usefulness when used in an object-orientated programming approach.