A Simplified Look At Object-Oriented Programming

Richard Hanxu, Jefferson Liu, Jason Li

When teenagers think of video games, one of the first games that comes to mind is Mario. This Italian, red-sporting plumber has made its way into the hearts of children, and even some adults, everywhere. Since his first game, Donkey Kong, in 1981, Mario has starred in hundreds of games, which have been played and enjoyed by all around the world.

Despite the numerous titles that he’s appeared in, the structure of Mario’s games have remained consistent over the years. With this observation, we can create a relationship between the formation of Mario games, to the process of object-oriented programming.

This explanation will consist of three levels, in increasing complexity. Readers who are new to the concept of object-oriented programming, and just want to be able to talk about it in a conversation, should read the words in pink, which in plain English, provide the thought process of OOP. Those looking for deeper understanding can go on to read the examples outlined in red, which aim to incorporate a code format. If you are planning to work directly with object-oriented programming in the future, we suggest reading the green explanations, which consist of code that would function in a real program.

Let’s examine this by starting with a common object in every Mario game: levels.

|  |
| --- |
| In every Mario game, there are always levels that Mario must progress through in order to rescue the princess. These levels have many individual variations, but some characteristics that all levels possess are enemies, items, platforms, a location, a state of whether or not it has been cleared, and flag poles. For example, a standard level would start off uncompleted, and possess one flag pole. |

This description sets the stage for a typical Mario level. It identifies the key components that are constant in all levels. As well, it listed more specifically the value of some things that would be found in a standard level.

|  |
| --- |
| MarioLevel  contains a certain number of enemies  contains a certain number of items  contains a certain number of platforms  is located somewhere  is either completed or uncompleted  contains a certain number of flag poles  a standard MarioLevel would possess the following qualities:  it is not completed  it contains one flag pole. |

Though this passage still contains English, it’s meaning may be a little more difficult to understand. The first line states that the author is defining the characteristics of a standard Mario level. As well, the words on the first line are not separated by a space. This is a rule that is present in programming. The following lines describe the characteristics, or **variables** of a level that are found in all **classes** of Mario levels. Lastly, a description of a normal level is provided with some characteristics already **constructed**.

|  |
| --- |
| public class MarioLevel  {  int enemies;  int items;  int platforms;  String location;  boolean isCompleted;  int flagPoles;  public MarioLevel ()  {  isCompleted = false;  flagPoles = 1;  }  } |

In this example, a **class** of a Mario level is being defined. However, it is important to realize here that an **object** is not being created, which in this case would be a level with specific characteristics (i.e. 25 enemies, 5 items, 12 platforms, etc.).

This code is also **declaring** certain **variables** that will exist in the level, by assigning them names (i.e. platforms) and the type of value they represent. Most of the values are represented by either a number, in this case, **int**, or by a string of letters, or **String**s. **Boolean** is a value that can only be either true or false, in this case, whether the level has been completed or not.

When writing code, curly braces “{“ are used to indicate different sections of code in a program. Classes include all their statements between a pair of “{“ “}”.

When a level is being created (**instantiated**) out of this class, or when an **object** of this class is being created, the **constructor** “public MarioLevel” will create a level **object** with some of the variable values already filled in. The brackets can contain **arguments**, or further specifications, but as they are empty, these are the actions that would occur if the **object** was **instantiated** without specifying any values.

Next, once we have defined the blueprints for a level, one can be finally created.

|  |
| --- |
| Level 8 - 10  This level takes place in Bowser’s Castle. The level will contain 20 enemies, 5 items, and 16 platforms. |

A level, namely Level 8 - 10, is being defined with specific information. The previous values that were left blank before in the blueprint are now being filled in these statements.

|  |
| --- |
| Main Level  MarioLevel level8-10 is a new MarioLevel  the location of level8-10 is in Bowser’s Castle  the number of enemies in level8-10 is twenty.  the number of items is five.  the number of platforms is sixteen. |

Here, an **object** of MarioLevel is being **instantiated**, and given the name “level8-10”. Next, the **variables** that were only **declared** in the class **fields** are now being **assigned** values

|  |
| --- |
| public class MarioLevel  {  public static void main (String [] args){  MarioLevel level8-10 = new MarioLevel ();  level8-10.location = “Bowser’s Castle”;  level8-10.enemies = 20;  level8-10.items = 5;  level8-10.platforms = 16;  }  } |

The line, “public static void main (String [] args)” , has many different utilities, but for now, it can be treated as the **main routine** where all the code is being read and run in a program.

While each of the **variables** are being assigned values, **dot notation**, with the object’s name appearing before the **variable**’s, lets the program know which **object** is receiving the changes to its values.

We have just now created a **class** for levels, and instantiated a level **object** in a **main routine** to be performed. However, those of us who have played Mario know that there is more to a game than just the levels we have to play through; it’s also about the characters, which can be playable, act as helpers, or stand as enemies in your way.

|  |
| --- |
| All characters in Mario have certain qualities. They have a name, can have eyes, a nose, legs, and/or a mouth, a name, belong to a species, and are either good or bad, |

These variables are needed in order to differentiate characters from each other. For example, a “Toad” is good, while a “Goomba” is bad. Some characters are missing eyes, noses, mouths, and even legs (i.e. Piranha Plant, Blooper), which is why each of those variables are required (there is no gore here, not to worry).

|  |
| --- |
| Characters  have a name  have a certain number of eyes.  have a certain number of noses.  have a certain number of mouths.  have a certain number of legs.  belong to a species.  are either good or bad.  A standard character would  have two eyes.  have one nose.  have one mouth.  have two legs.  When one meets a character,  the character tells them their name. |

Now, our characters are also able to interact, unlike our levels previously, by being able to give out information of their name. In programming, this ability to interchange is called a **method.**

|  |
| --- |
| public class Character  {  String name;  int eyes;  int noses;  int mouths;  int legs;  String species;  String goodOrBad;  public Character ()  {  eyes = 2;  noses = 1;  mouths = 1;  legs = 2;  }  public String whatIsYourName () {  return name;  }  } |

The section, “public String areYouGoodOrBad ()” is a **method**, which gives objects **instantiated** with this class a **behaviour**, allowing it to perform an action. In this case, it returns a value of the character’s name.

In the world of Mario though, “characters” isn’t quite specific enough. After all, there are plumbers, princesses, and enough enemies to fill an encyclopedia. Thus, further analysis is required. There are tons of different characters we could choose to go into further detail, but a strong example, literally, is bosses.

|  |
| --- |
| Bosses are a type of character. They share many similar traits, but also have a few more, such as the “world” they are guarding, a temperament, the number of hits they can sustain on the head before collapsing, a special power, and a princess they have captured. |

This is an example of a **subclass**, which **extends** from a **parent class**. Here, the concept of bosses is taking pre existing information about characters, and building onto it to make it more diverse.**Subclasses** are not the same as **objects**. **Subclasses** can be viewed as a more specific blueprint (i.e. going from what makes a building to what makes a house or an apartment), while **objects** are created directly from classes (Making a house from the blueprints).

|  |
| --- |
| Bosses extend the idea of Characters.  Bosses  have a “world” they are guarding.  have a number of hits on the head they can take before collapsing.  have a temperament.  have a special power.  have captured a princess    For a given Boss,  they would start calm and cocky.  they would start with three hits on the head they could take before collapsing. |

|  |
| --- |
| public class Boss extends Character  {  String world;  int numberOfHitsBeforeCollapsing;  int temper;  String specialPower;  Character princess;  public Boss ()  {  int temper = 0;  int numberOfHitsBeforeCollapsing = 3;  }  } |

The reason that “Character princess;” is present, is to allow a Character **object**, or **instance** **variable**, to be used in the Boss **class**. This is necessary so that we can know which princess the boss has captured.

Bosses don’t just look intimidating, they should be able to do some things to threaten you as well. Beforehand, we saw our characters being able to perform **methods**, such as giving whether they were good or evil. We can also give our bosses **behaviours**.

|  |
| --- |
| When a boss suffers a hit to the head, the number of hits they can sustain before collapsing decreases by one, and their level of temperament, or anger, will increase by one as well. |

This statement is only addressing how the boss **object** will act. It is not providing any extra information or characteristics about a boss. This is often how a method usually works, by accessing and changing class values. We can see that there is one defined method with two outputs: Their number of sustainable hits will decrease by one, and their level of anger will increase by one when they are stomped on.

|  |
| --- |
| becomeStompedOn  number of sustainable hits before collapsing decreases by one  level of anger increases by one |

The first line describes the **method**, while the following lines describe its output or effect. This is similar to the concept of **variables**, where they are first declared, and then assigned a value.

|  |
| --- |
| public void becomeStompedOn (){  {  int temper = temper ++;  int numberOfHitsBeforeCollapsing --;  } |

“++” and “--” indicate that the variables they were assigned two increased and decreased by one respectively.

Bosses also have a lot of confidence in themselves, partly due to the special powers that put them above the rest. Thus, whenever they meet someone who challenges them, they always reveal their special powers at the beginning of the fight. We can also represent this through **methods**.

|  |
| --- |
| When a boss meets someone who challenges them to a fight, they will always reveal to them their special power. |

|  |
| --- |
| whatIsYourSpecialPower  tell them their special power |

|  |
| --- |
| public String whatIsYourSpecialPower (){  {  return specialPower;  } |

Last time, the method had “**void**” in its name, instead of “**String**”. The first method’s description indicated that a **return statement** was not necessary. **String** in the description indicates that one of the purposes of the method will be to **return** a **String** value, in this case, the boss’ special power.

Further adding to a boss’ confidence, they will also reveal who they’ve captured (not a good idea to flaunt to someone you’ve kidnapped their possibly romantic partner, but okay).

|  |
| --- |
| If a boss meets the romantic partner of the princess they’ve captured, they will reveal the name of the princess they captured to the other character. Then, a statement, reading “Oh no! (the boss’ name) has captured (the princess’ name)!” |

|  |
| --- |
| flauntConfidence (princess name)  Print “Oh no! The enemy has captured (the princess’ name)!” |

|  |
| --- |
| public void flauntConfidence (Character princess){  {  this.princess = princess;  System.out.print(“Oh no! The enemy has captured “ + princess.whatIsYourName() + “!”);  } |

The words “Character princess” inside the brackets on the first line form an **argument**. Before, we glossed over them, but essentially, they can make an action more specific. These arguments are also **variables**, but they are local, meaning that they can only be used inside its current method, and disappear when it finishes its process. In this case, by allowing an input of a princess Character while calling the method, it allows us to perform the next statement in the code.

The statement “this.princess = princess” may seem confusing, so here is an explanation that will hopefully increase your understanding. If we examine the class for bosses, we can see that we have declared a variable named “princess”. This means that inside our **method**, we have two variables with the same name (one is a **local variable** from the method **argument**, and the other was declared creating our extended **class**). Thus, what the “this.” does is it references the variable declared from the extended **class**. As a result, the value of the “princess” variable *declared in our class* is being assigned the same value as the “princess” variable *given in the parameter*.

Finally, the code “System.out.print” indicates that the text between the quotation marks is to be outputted directly, in this case, on the game’s graphics. The statement “princess.whatIsYourName()”, upon review, returns the name of the character, but is not put inside the quotation marks. This is because the text between the quotation marks is treated literally (called a **literal**), meaning whatever is located between will be printed out. We don’t want to literally print the letters “princess.whatIsYourName ()” onto the screen, which is why it is not surrounded by quotation marks. Instead, this statement accesses the Character **class**, finds the princess **object**, and then runs the **method** “whatIsYourName ()” to return the value of the **object**’s name. The “+” means a joining of two statements.

Finally, let’s review what we’ve done to define our boss character class.

|  |
| --- |
| Bosses are a type of character. They share many similar traits, but also have a few more, such as the “world” they are guarding, a temperament, the number of hits they can sustain on the head before collapsing, a special power, and a princess they have captured. A typical boss would appear calm and cocky, and start with three hits on the head they could sustain before collapsing.  When a boss suffers a hit to the head, the number of hits they can sustain before collapsing decreases by one, and their level of temperament, or anger, will increase by one as well. Also, when a boss meets someone who challenges them to a fight, they will always reveal to them their special power. Finally, if a boss meets the romantic partner of the princess they’ve captured, they will reveal the name of the princess they captured to the other character. Then, a statement, reading “Oh no! (the boss’ name) has captured (the princess’ name)!” |

|  |
| --- |
| Bosses extend the idea of Characters.  Bosses  have a “world” they are guarding.  have a number of hits on the head they can take before collapsing.  have a temperament.  have a special power.  have captured a princess    For a given Boss,  they would start calm and cocky.  they would start with three hits on the head they could take before collapsing.  becomeStompedOn  number of sustainable hits before collapsing decreases by one  level of anger increases by one  whatIsYourSpecialPower  tell them their special power  flauntConfidence (princess name)  Print “Oh no! The enemy has captured (the princess’ name)!” |

|  |
| --- |
| public class Boss extends Character  {  String world;  int numberOfHitsBeforeCollapsing;  int temper;  String specialPower;  Character princess;  public Boss ()  {  int temper = 0;  int numberOfHitsBeforeCollapsing = 3;  }  public void becomeStompedOn (){  {  int temper = temper ++;  int numberOfHitsBeforeCollapsing --;  }  public String whatIsYourSpecialPower (){  {  return specialPower;  }  public void flauntConfidence (Character princess){  {  this.princess = princess;  System.out.print(“Oh no! The enemy has captured “ + princess.whatIsYourName() + “!”);  }  } |

Pardon the spacing in that method there.

Now, let’s see how this would look after designing an entire level, complete with a level, characters, and a boss.

|  |
| --- |
| In every Mario game, there are always levels that Mario must progress through in order to rescue the princess. These levels have many individual variations, but some characteristics that all levels possess are enemies, items, platforms, a location, a state of whether or not it has been cleared, and flag poles. For example, a standard level would start off uncompleted, and possess one flag pole. There is also a boss character named Bowser. Bowser is a Koopaling. Bowser is evil. Bowser is guarding World 8. He can take three hits on the head before collapsing. He is generally calm and cocky. His special power is being able to shoot fireballs from his mouth. He has captured a princess named, Princess Peach. Princess Peach is female. She is human, and good. In this level, Bowser reveals his special power, the princess he has kidnapped, and consequently becomes stomped on three times. |

|  |
| --- |
| Main Level  MarioLevel level8-10 is a new MarioLevel  the location of level8-10 is in Bowser’s Castle  the number of enemies in level8-10 is twenty.  the number of items is five.  the number of platforms is sixteen.  There is a boss character named Bowser  His name is Bowser  He is male.  He is evil  He is the boss of World 8  He is generally calm and cocky  He can take three hits to the head before collapsing  His special power is being able to shoot fireballs  There is a princess named Princess Peach  She is female  She is human  She is good  Bowser reveals his special power  Bowser reveals he has captured Princess Peach  Bowser gets stomped on the head three times |

His name is Bowser

He is male.

He is evil

He is the boss of World 8

He is generally calm and cocky

|  |
| --- |
| public class MarioLevel  {  public static void main (String [] args){  MarioLevel level8-10 = new MarioLevel ();  level8-10.location = “Bowser’s Castle”;  level8-10.enemies = 20;  level8-10.items = 5;  level8-10.platforms = 16;    Boss Bowser = new Boss();  Bowser.name = “Bowser”;  Bowser.species = “Koopaling”;  Bowser.goodOrBad = “Bad”;  Bowser.world = “World 8”;  Bowser.specialPower = “Shoots Fireballs”;  Bowser.temper = 0;  Bowser.numberOfHitsBeforeCollapsing = 3;    Character princessPeach = new Character();  princessPeach.gender = “Female”;  princessPeach.species = “Human”;  princessPeach.goodOrBad = “Good”;  System.out.println(Bowser.whatIsYourSpecialPower());  Bowser.flauntConfidence(princessPeach);  Bowser.becomeStompedOn();  Bowser.becomeStompedOn();  Bowser.becomeStompedOn();    }  } |

This “**main routine**” may seem uninteresting, but if we take a closer examination, we can see how it incorporates everything we learned before to create a simple, elegant level.

First, an **object** of a Level is **instantiated** from its **class**, and given the name Level8-10. Some characteristics are added to it to give it the difficulty of a real challenging level one might find in an actual Mario game.

Next, our boss, Bowser, is also initialized from our Bosses, which **extended** or expanded upon the idea of a regular Character **class**. He is given characteristics belonging to a Boss, as well as an ordinary character (i.e. specialPower, goodOrBad).

We also have another Character, princessPeach, who also plays a large part in our level. After all, she is the one captured by Bowser. As she is a Character but not a Boss, she is given characteristics only belonging to the Character **class**.

Finally, we can get to our **behaviours** or **methods**, which dictate the events that will occur in this level. Bowser will get cocky, reveal his special powers, show the princess he has kidnapped, and ultimately, be defeated by getting stomped on thrice.

When we use the format of objects and classes, it is easier to foster interaction within our level. By applying this same concept to the programming language Java, we hope you learned a little bit more about **object-oriented programming**.