What is Object Oriented Programming?

In classic, procedural programming you try to make the real world problem you're attempting to solve fit a few, pre-determined data types: integers, floats, Strings, and arrays perhaps. In object oriented programming you create a model for a real world system. Classes are programmer-defined types that model the parts of the system.

A \textit{class} is a programmer defined type that serves as a blueprint for instances of the class. You can still have ints, floats, Strings, and arrays; but you can also have cars, motorcycles, people, buildings, clouds, dogs, angles, students, courses, bank accounts, and any other type that's important to your problem.

Classes specify the data and behavior possessed both by themselves and by the objects built from them. A class has two parts: the \textit{fields} and the \textit{methods}. Fields describe what the class is. Methods describe what the class does.

Using the blueprint provided by a class, you can create any number of objects, each of which is called an \textit{instance} of the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. For example, all humans have eye color but the color of each human's eyes can be different from others.

On the other hand, objects have the same methods as all other objects in the class except in so far as the methods depend on the value of the fields and arguments to the method.

\textbf{Example 1: The Car Class}

Suppose you need to write a traffic simulation program that watches cars going past an intersection. Each car has a speed, a maximum speed, and a license plate that uniquely identifies it. In traditional programming languages you'd have two floating point and one string variable for each car. With a class you combine these into one thing like this.

class Car {
  String licensePlate; // e.g. "New York 543 A23"
  double speed;      // in kilometers per hour
  double maxSpeed;   // in kilometers per hour
}

These variables (licensePlate, speed and maxSpeed) are called the \textit{member variables}, \textit{instance variables}, or \textit{fields} of the class. Fields tell you what a class is and what its properties are.

An \textit{object} is a specific instance of a class with particular values for the fields. While a class is a general blueprint for objects, an instance is a particular object.

\textbf{Constructing objects with “new”}

class Car {
  String licensePlate; // e.g. "New York 543 A23"
  double speed;      // in kilometers per hour
  double maxSpeed;   // in kilometers per hour
}

To instantiate an object in Java, use the keyword \texttt{new} followed by a call to the class's constructor. Here's how you'd create a new \texttt{Car} variable called \texttt{c}:

\begin{verbatim}
Car c;
    c = new Car();
\end{verbatim}
The first word, Car, declares the type of the variable c. Classes are types and variables of a class type need to be declared just like variables that are ints or doubles.

The equals sign is the assignment operator and new is the construction operator.

Finally notice the Car() method. The parentheses tell you this is a method and not a data type like the Car on the left hand side of the assignment. This is a constructor, a method that creates a new instance of a class. You'll learn more about constructors shortly. However if you do nothing, then the compiler inserts a default constructor that takes no arguments.

Using a Car object in a different class

class Car {
  String licensePlate; // e.g. "New York 543 A23"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour
}

The next program creates a new car, sets its fields, and prints the result:

class CarTest {
  public static void main(String args[]) {
    Car c = new Car();
    c.licensePlate = "New York A45 636";
    c.speed = 70.0;
    c.maxSpeed = 123.45;
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
  }
}

This program requires not just the CarTest class but also the Car class. To make them work together put the Car class in a file called Car.java. Put the CarTest class in a file called CarTest.java. Put both these files in the same directory. Then compile both files in the usual way. Finally run CarTest. For example,

% javac Car.java
% javac CarTest.java
% java CarTest
New York A45 636 is moving at 70.0 kilometers per hour.

Note that Car does not have a main() method so you cannot run it. It can exist only when called by other programs that do have main() methods.

Methods

Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class is. Methods say what a class does. The fields and methods of a class are collectively referred to as the members of the class.

The classes you've encountered up till now have mostly had a single method, main(). However, in general classes can have many different methods that do many different things. For instance the Car class might have a method to make the car go as fast as it can. For example,
class Car {
    String licensePlate = ""; // e.g. "New York 543 A23"
    double speed = 0.0; // in kilometers per hour
    double maxSpeed; = 120.0; // in kilometers per hour

    // accelerate to maximum speed
    // put the pedal to the metal
    void floorIt() {
        this.speed = this.maxSpeed;
    }
}

The fields are the same as before, but now there's also a method called floorIt(). It begins with the Java keyword void which is the return type of the method. Every method must have a return type which will either be void or some data type like int, byte, float, or String. The return type says what kind of the value will be sent back to the calling method when all calculations inside the method are finished. If the return type is int, for example, you can use the method anywhere you use an int constant. If the return type is void then no value will be returned.

There is one statement inside the method

this.speed = this.maxSpeed;

Notice that within the Car class the field names are prefixed with the keyword this to indicate that I'm referring to fields in the current object.

Invoking Methods

class Car {
    String licensePlate = ""; // e.g. "New York 543 A23"
    double speed = 0.0; // in kilometers per hour
    double maxSpeed; = 120.0; // in kilometers per hour

    // accelerate to maximum speed
    // put the pedal to the metal
    void floorIt() {
        this.speed = this.maxSpeed;
    }
}

Outside the Car class, you call the floorIt() method just like you reference fields, using the name of the object you want to accelerate to maximum and the . separator as demonstrated below

class CarTest3 {
    public static void main(String args[]) {
        Car c = new Car();
        c.licensePlate = "New York A45 636";
        c.maxSpeed = 123.45;
        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
        c.floorIt();
        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
    }
}

The output is:
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 123.45 kilometers per hour.
Implied this

class Car {
    String licensePlate = ""; // e.g. "New York 543 A23"
double speed        = 0.0;   // in kilometers per hour
double maxSpeed;    = 120.0; // in kilometers per hour

    void floorIt() {
        speed = maxSpeed;
    }
}

Within the Car class, you don't absolutely need to prefix the field names with this. like this.licensePlate or this.speed. Just licensePlate and speed are sufficient. The this. may be implied. That's because the floorIt() method must be called by a specific instance of the Car class, and this instance knows what its data is. Or, another way of looking at it, the every object has its own floorIt() method.

Member Variables vs. Local Variables

class Car {
    String licensePlate = ""; // member variable
double speed;       = 0.0;   // member variable
double maxSpeed;    = 120.0; // member variable

    boolean isSpeeding() {
        double excess;    // local variable
        excess = this.maxSpeed - this.speed;
        if (excess < 0) return true;
        else return false;
    }
}

Until now all the programs you've seen quite simple in structure. Each had exactly one class. This class had a single method, main(), which contained all the program logic and variables. The variables in those classes were all local to the main() method. They could not be accessed by anything outside the main() method. These are called local variables.

This sort of program is the amoeba of Java. Everything the program needs to live is contained inside a single cell. It's quite an efficient arrangement for small organisms, but it breaks down when you want to design something bigger or more complex.

The licensePlate, speed and maxSpeed variables of the Car class, however, belong to a Car object, not to any individual method. They are defined outside of any methods but inside the class and are used in different methods. They are called member variables or fields.

Member variable, instance variable, and field are different words that mean the same thing. Field is the preferred term in Java. Member variable is the preferred term in C++.

A member is not the same as a member variable or field. Members include both fields and methods.
Passing Arguments to Methods

It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

To do this you need to be able to send information into the Car class. This is done by passing arguments. For example, to allow other objects to change the value of the speed field in a Car object, the Car class could provide an accelerate() method. This method does not allow the car to exceed its maximum speed, or to go slower than 0 kph.

```java
void accelerate(double deltaV) {
    this.speed = this.speed + deltaV;
    if (this.speed > this.maxSpeed) {
        this.speed = this.maxSpeed;
    }
    if (this.speed < 0.0) {
        this.speed = 0.0;
    }
}
```

The first line of the method is called its signature. The signature

`void accelerate(double deltaV)`

indicates that accelerate() returns no value and takes a single argument, a double which will be referred to as deltaV inside the method. deltaV is a purely formal argument.

Java passes method arguments by value, not by reference.

Passing Arguments to Methods, An Example

class Car {
    String licensePlate = "";    // e.g. "New York 543 A23"
    double speed        = 0.0;   // in kilometers per hour
    double maxSpeed;    = 120.0; // in kilometers per hour

    // accelerate to maximum speed
    // put the pedal to the metal
    void floorIt() {
        this.speed = this.maxSpeed;
    }

    void accelerate(double deltaV) {
        this.speed = this.speed + deltaV;
        if (this.speed > this.maxSpeed) {
            this.speed = this.maxSpeed;
        }
        if (this.speed < 0.0) {
            this.speed = 0.0;
        }
    }
}
class CarTest4 {
    public static void main(String[] args) {
        Car c = new Car();
        c.licensePlate = "New York A45 636";
        c.maxSpeed = 33.45;

        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");

        for (int i = 0; i < 4; i++) {
            c.accelerate(10.0);
            System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
        }
    }
}

Here's the output:
utoopia% java CarTest4
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.

Setter Methods

Setter methods, also known as mutator methods, merely set the value of a field to a value specified by the argument to the method. These methods almost always return void.

One common idiom in setter methods is to use this.name to refer to the field and give the argument the same name as the field. For example,

class Car {
    String licensePlate; // e.g. "New York A456 324"
    double speed;       // kilometers per hour
    double maxSpeed;    // kilometers per hour

    // setter method for the license plate property
    void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    // setter method for the maxSpeed property
    void setMaximumSpeed(double maxSpeed) {
        if (maxSpeed > 0) this.maxSpeed = maxSpeed;
        else this.maxSpeed = 0.0;
    }

    // accelerate to maximum speed
    // put the pedal to the metal
    void floorIt() {
        this.speed = this.maxSpeed;
    }
}
void accelerate(double deltaV) {
    this.speed = this.speed + deltaV;
    if (this.speed > this.maxSpeed) {
        this.speed = this.maxSpeed;
    }
    if (this.speed < 0.0) {
        this.speed = 0.0;
    }
}
}

Using Setter Methods, an Example

class CarTest5 {
    public static void main(String args[]) {
        Car c = new Car();
        c.setLicensePlate("New York A45 636");
        c.setMaximumSpeed(33.45);

        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");

        for (int i = 0; i < 5; i++) {
            c.accelerate(10.0);
            System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
        }
    }
}

Here's the output:

utopia% java CarTest5
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.

Returning Values From Methods

It's often useful to have a method return a value to the class that called it. This is accomplished by the return keyword at the end of a method and by declaring the data type that is returned by the method at the beginning of the method.

For example, the following getLicensePlate() method returns the current value of the licensePlate field in the Car class.

    String getLicensePlate() {
        return this.licensePlate;
    }
A method like this that merely returns the value of an object's field or property is called a *getter* or *accessor* method.

The signature `String getLicensePlate()` indicates that `getLicensePlate()` returns a value of type `String` and takes no arguments. Inside the method the line

```
return this.licensePlate;
```

returns the `String` contained in the `licensePlate` field to whoever called this method. It is important that the type of value returned by the return statement match the type declared in the method signature. If it does not, the compiler will complain.

### Using Getter Methods, an Example

```java
class CarTest6 {
    public static void main(String args[]) {
        Car c = new Car();
        c.setLicensePlate("New York A45 636");
        c.setMaximumSpeed(33.45);

        System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() + " kilometers per hour.");

        for (int i = 0; i < 5; i++) {
            c.accelerate(10.0);
            System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() + " kilometers per hour.");
        }
    }
}
```

There's no longer any direct access to fields!

Here's the output:

```
utopia% java CarTest6
New York A45 636 is moving at 0.0 kilometers per hour.
New York A45 636 is moving at 10.0 kilometers per hour.
New York A45 636 is moving at 20.0 kilometers per hour.
New York A45 636 is moving at 30.0 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.
New York A45 636 is moving at 33.45 kilometers per hour.
```

### Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line

```
Car c = new Car();
```

`Car()` is the constructor. A constructor has the same name as the class.
If no constructor exists Java provides a generic one that takes no arguments (a noargs constructor), but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called Car().

Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.

The following method is a constructor that initializes license plate to an empty string, speed to zero, and maximum speed to 120.0.

```java
Car() {
    this.licensePlate = "";
    this.speed = 0.0;
    this.maxSpeed = 120.0;
}
```

Better yet, you can create a constructor that accepts three arguments and use those to initialize the fields as below.

```java
Car(String licensePlate, double speed, double maxSpeed) {
    this.licensePlate = licensePlate;
    this.speed = speed;
    if (maxSpeed > 0) this.maxSpeed = maxSpeed;
    else this.maxSpeed = 0.0;
    if (speed > this.maxSpeed) this.speed = this.maxSpeed;
    if (speed < 0) this.speed = 0.0;
    else this.speed = speed;
}
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified:

```java
Car(String licensePlate, double maxSpeed) {
    this.licensePlate = licensePlate;
    this.speed = 0.0;
    if (maxSpeed > 0) this.maxSpeed = maxSpeed;
    else this.maxSpeed = 0.0;
}
```
Constructors

Here's the complete class:

class Car {

    String licensePlate; // e.g. "New York A456 324"
    double speed;       // kilometers per hour
    double maxSpeed;    // kilometers per hour

    Car(String licensePlate, double maxSpeed) {
        this.licensePlate = licensePlate;
        this.speed = 0.0;
        if (maxSpeed > 0) this.maxSpeed = maxSpeed;
        else this.maxSpeed = 0.0;
    }

    // getter (accessor) methods
    String getLicensePlate() {
        return this.licensePlate;
    }

    double getMaxSpeed() {
        return this.maxSpeed;
    }

    double getSpeed() {
        return this.speed;
    }

    // accelerate to maximum speed
    // put the pedal to the metal
    void floorIt() {
        this.speed = this.maxSpeed;
    }

    void accelerate(double deltaV) {
        this.speed = this.speed + deltaV;
        if (this.speed > this.maxSpeed) {
            this.speed = this.maxSpeed;
        }
        if (this.speed < 0.0) {
            this.speed = 0.0;
        }
    }
}

Notice that several things are taken out:

- the initialization of the fields
- the setter methods
Using Constructors

The next program uses the constructor to initialize a car rather than setting the fields directly.

```java
class CarTest7 {
    public static void main(String args[]) {
        Car c = new Car("New York A45 636", 123.45);

        System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() + " kilometers per hour.");

        for (int i = 0; i < 15; i++) {
            c.accelerate(10.0);
            System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() + " kilometers per hour.");
        }
    }
}
```

You no longer need to know about the fields `licensePlate`, `speed` and `maxSpeed`. All you need to know is how to construct a new car and how to print it.

You may ask whether the `setLicensePlate()` method is still needed since it's now set in a constructor. The general answer to this question depends on the use to which the `Car` class is to be put. The specific question is whether a car's license plate may need to be changed after the `Car` object is created.

Constraints

One of the reasons to use constructors and setter methods rather than directly accessing fields is to enforce constraints. For instance, in the `Car` class it's important to make sure that the speed is always less than or equal to the maximum speed and that both speed and maximum speed are greater than or equal to zero.

You've already seen one example of this in the `accelerate()` method which will not accelerate a car past its maximum speed.

```java
void accelerate(double deltaV) {
    this.speed = this.speed + deltaV;
    if (this.speed > this.maxSpeed) {
        this.speed = this.maxSpeed;
    }
    if (this.speed < 0.0) {
        this.speed = 0.0;
    }
}
```
You can also insert constraints like that in the constructor. For example, this `Car` constructor makes sure that the maximum speed is greater than or equal to zero:

```java
Car(String licensePlate, double maxSpeed) {
    this.licensePlate = licensePlate;
    this.speed = 0.0;
    if (maxSpeed >= 0.0) {
        this.maxSpeed = maxSpeed;
    } else {
        maxSpeed = 0.0;
    }
}
```

**Access Protection**

Global variables are a classic cause of bugs in most programming languages. Some unknown function can change the value of a variable when the programmer isn't expecting it to change. This plays all sorts of havoc.

Most OOP languages including Java allow you to protect variables from external modification. This allows you to guarantee that your class remains consistent with what you think it is as long as the methods of the class themselves are bug-free. For example, in the `Car` class we'd like to make sure that no block of code in some other class is allowed to make the speed greater than the maximum speed. We want a way to make the following illegal:

```java
Car c = new Car("New York A234 567", 100.0);
c.speed = 150.0;
```

This code violates the constraints we've placed on the class. We want to allow the compiler to enforce these constraints.

A class presents a picture of itself to the world. (This picture is sometimes called an *interface*, but the word *interface* has a more specific meaning in Java.) This picture says that the class has certain methods and certain fields. Everything else about the class including the detailed workings of the class's methods is hidden. As long as the picture the class shows to the world doesn't change, the programmer can change how the class implements that picture. Among other advantages this allows the programmer to change and improve the algorithms a class uses without worrying that some piece of code depends in unforeseen ways on the details of the algorithm used. This is called encapsulation.

Another way to think about encapsulation is that a class signs a contract with all the other classes in the program. This contract says that a class has methods with unambiguous names which take particular types of arguments and return a particular type of value. The contract may also say that a class has fields with given names and of a given type. However the contract does not specify how the methods are implemented. Furthermore, it does not say that there aren't other private fields and methods which the class may use. A contract guarantees the presence of certain methods and fields. It does not exclude all other methods and fields. This contract is implemented through access protection. Every class, field and method in a Java program is defined as either public, private, protected or unspecified.

You're closer to your immediate family (your parents and your children) than you are to your cousins. You're closer to your cousins than to the general public at large, but there are some things you don't tell anybody. Furthermore, your family is not my family.
Examples of Access Protection

This is how the Car class would probably be written in practice. Notice that all the fields are now declared private, and they are accessed only through public methods. This is the normal pattern for all but the simplest classes.

```java
public class Car {
    private String licensePlate; // e.g. "New York A456 324"
    private double speed;        // kilometers per hour
    private double maxSpeed;     // kilometers per hour

    public Car(String licensePlate, double maxSpeed) {
        this.licensePlate = licensePlate;
        this.speed = 0.0;
        if (maxSpeed >= 0.0) {
            this.maxSpeed = maxSpeed;
        } else {
            maxSpeed = 0.0;
        }
    }

    // getter (accessor) methods
    public String getLicensePlate() {
        return this.licensePlate;
    }

    public double getMaxSpeed() {
        return this.speed;
    }

    public double getSpeed() {
        return this.maxSpeed;
    }

    // setter method for the license plate property
    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    // accelerate to maximum speed
    // put the pedal to the metal
    public void floorIt() {
        this.speed = this.maxSpeed;
    }

    public void accelerate(double deltaV) {
        this.speed = this.speed + deltaV;
        if (this.speed > this.maxSpeed) {
            this.speed = this.maxSpeed;
        }
        if (this.speed < 0.0) {
            this.speed = 0.0;
        }
    }
}
```
In many cases there will also be private, protected and default access methods as well. Collectively these are called non-public methods.

In many cases, the fields may be protected or default access. However public fields are rare. This allows programmers to change the implementation of a class while still maintaining the same contract with the outside world.

Examples of Access Protection

Now let's try to directly access the fields from another class and see what happens:

class CarTest8 {
    public static void main(String args[]) {
        Car c = new Car("New York A45 636", 100.0);
        c.licensePlate = "New York A45 636";
        c.speed = 0.0;
        c.maxSpeed = 123.45;
        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
        c.floorIt();
        System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
    }
}

Here's what happens when you try to compile it against the revised Car class:

% javac Car.java
% javac CarTest8.java
CarTest8.java:7: Variable licensePlate in class Car not accessible from class CarTest8.
    c.licensePlate = "New York A45 636";
         ^
CarTest8.java:8: Variable speed in class Car not accessible from class CarTest8.
    c.speed = 0.0;
         ^
CarTest8.java:9: Variable maxSpeed in class Car not accessible from class CarTest8.
    c.maxSpeed = 123.45;
         ^
CarTest8.java:11: Variable licensePlate in class Car not accessible from class CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
         ^
CarTest8.java:11: Variable speed in class Car not accessible from class CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
         ^
CarTest8.java:11: Variable speed in class Car not accessible from class CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
         ^
CarTest8.java:16: Variable licensePlate in class Car not accessible from class CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
         ^
CarTest8.java:16: Variable speed in class Car not accessible from class CarTest8.
    System.out.println(c.licensePlate + " is moving at " + c.speed + " kilometers per hour.");
         ^
7 errors
%
The Three Benefits of Access Protection

Access protection has three main benefits:

1. It allows you to enforce constraints on an object's state.
2. It provides a simpler client interface. Client programmers don't need to know everything that's in the class, only the public parts.
3. It separates interface from implementation, allowing them to vary independently. For instance consider making the licensePlate field of Car an instance of a new LicensePlate class instead of a String.

Changing the Implementation

Suppose the Car class needs to be used in a simulation of New York City traffic in which each actual car on the street is represented by one Car object. That's a lot of cars. As currently written each car object occupies approximately 60 bytes of memory (depending mostly on the size of the license plate string. We can knock off eight bytes per car by using floats instead of doubles, but the interface can stay the same:

```java
public class Car {
  private String licensePlate; // e.g. "New York A456 324"
  private float speed;        // kilometers per hour
  private float maxSpeed;     // kilometers per hour

  public Car(String licensePlate, double maxSpeed) {
    this.licensePlate = licensePlate;
    this.speed = 0.0F;
    if (maxSpeed >= 0.0) {
      this.maxSpeed = (float) maxSpeed;
    } else {
      maxSpeed = 0.0F;
    }
  }

  public String getLicensePlate() {
    return this.licensePlate;
  }

  public double getMaxSpeed() {
    return this.speed;
  }

  public double getSpeed() {
    return this.maxSpeed;
  }

  // setter method for the license plate property
  public void setLicensePlate(String licensePlate) {
    this.licensePlate = licensePlate;
  }

  // accelerate to maximum speed
  // put the pedal to the metal
  public void floorIt() {
    this.speed = this.maxSpeed;
  }
}
```
public void accelerate(double deltaV) {
    this.speed = this.speed + (float) deltaV;
    if (this.speed > this.maxSpeed) {
        this.speed = this.maxSpeed;
    }
    if (this.speed < 0.0) {
        this.speed = 0.0F;
    }
}

**What should be public? What should be private?**

As a rule of thumb:

- Classes are public.
- Fields are private.
- Constructors are public.
- Getter and setter methods are public.
- Other methods must be decided on a case-by-case basis.

All of these rules may be freely violated if you have a reason for doing so. These are simply the defaults that handle 90% of the cases.