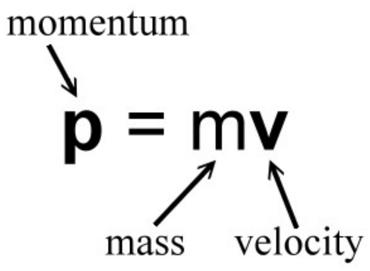
# Collision Calculator

One of the core concepts you'll learn in AP Physics I is the **conservation of momentum**. In physics, **momentum** is defined by the following equation:



In any closed system, the net initial momentum will always equal the net final momentum. Thus, we can use this equation to predict the outcomes of collisions.

There are two types of collisions discussed in AP Physics I:

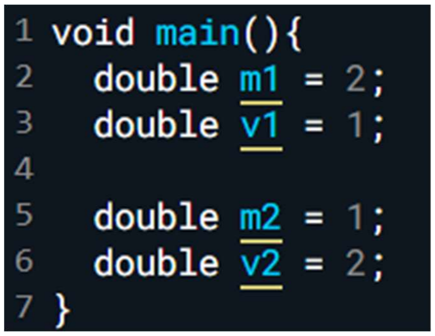
**Elastic Collisions** are collisions in which two objects bounce off of each other on impact.

**Inelastic Collisions** are collisions in which two objects stick together on impact.

That being said, let's write a calculator to compute the outcomes of collisions.

---

First, let's define our given values. In order to compute the outcomes of collisions, you need both the initial and final mass values. Thus, let's define these values like so:



```
void main(){
  double m1 = 2;
  double v1 = 1;

  double m2 = 1;
  double v2 = 2;
}
```

**No Output**

Recall that a "double" is the same thing as a rational number.

Before continuing, let's rewrite the conservation of momentum so we can actually use it:

$$\rho_i = \rho_f$$

Here, we have the simplified version, but let's expand it for our use:

$$m_f v_f = m_1 v_1 + m_2 v_2$$

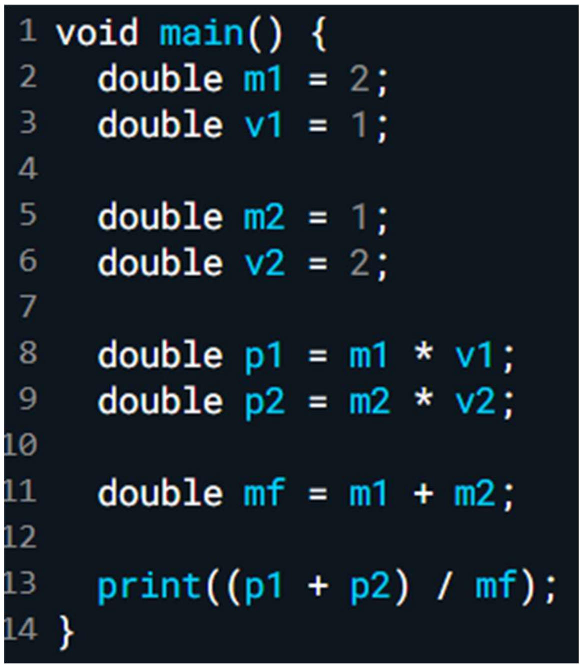
Here, we've just redefined momentum as mass times velocity, and accounted for both objects in this collision. We know from the conservation of mass that:

$$(m_1 + m_2) v_f = m_1 v_1 + m_2 v_2$$

Now, we're really interested in learning the final velocity here, so we can rearrange the equation to finally get:

$$v_f = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2}$$

Knowing this, we can calculate the final velocity like so:



```
void main() {
  double m1 = 2;
  double v1 = 1;

  double m2 = 1;
  double v2 = 2;

  double p1 = m1 * v1;
  double p2 = m2 * v2;

  double mf = m1 + m2;

  print((p1 + p2) / mf);
}
```

**Output**: 1.33333333

Here, I separated the computation across several variables: momentum 1, momentum 2, and mass final. Then, I just plugged them into our equation, and now we have a functioning calculator.

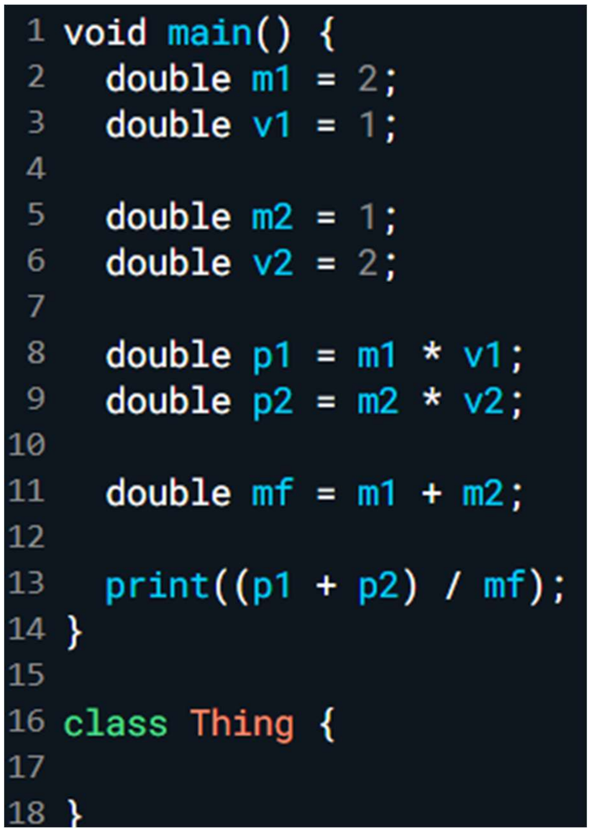
But that's too easy! Let's learn something new!

Recall from our last lesson that we briefly talked about objects. **Objects** are like variables in programming which have multiple different attributes or values and multiple different functions attached to them—called **instance variables** and **object functions** respectively.

One example of an object we've already been working with is a List. Remember that a list has multiple different values and functions we can use them for.

The reason I'm talking about objects is because, ideally, we'd like to create an object to store our mass and velocity values, but also to hold the functions to do the calculations for us. But how do we do that?

**Classes** are also collections of values and functions which serve the main purpose of being blueprints for creating objects. Think of classes as organized collections of variables. Thus, before we can define our object, we'll need to define a class. Classes are defined very similarly to functions, e.g:

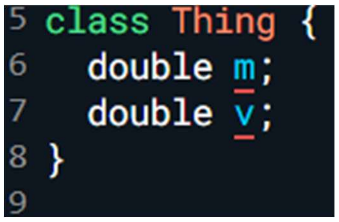


```dart
void main() {
  double m1 = 2;
  double v1 = 1;

  double m2 = 1;
  double v2 = 2;

  double p1 = m1 * v1;
  double p2 = m2 * v2;

  double mf = m1 + m2;

  print((p1 + p2) / mf);
}

class Thing {

}
```

**No Output**

Generally, you want to name classes in UpperCamelCase format.

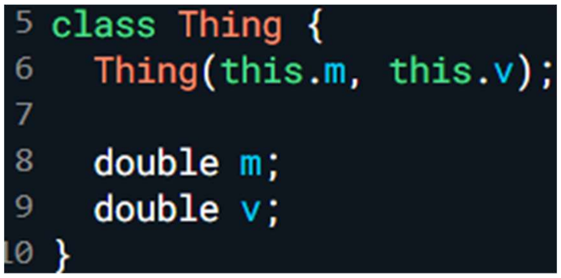
Anything within the curly braces of the class will be a part of the class. Knowing this, let's go ahead and define the instance variables of our object in the class:



```dart
class Thing {
  double m;
  double v;
}
```

Here, we didn't assign any values to the variables, as they're going to vary among instances. However, we get an error because we're not telling the computer that these variables need to be defined for each object. To do that, we need to create a constructor.

A **Constructor** is simply a function in the code that tells the computer how to create an object from a class. Dart oversimplifies the constructor, so all we have to write is this:



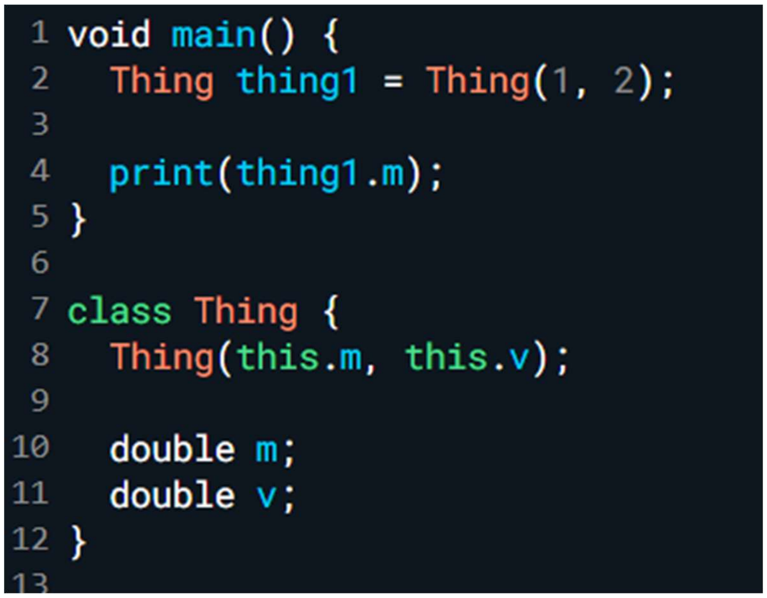
```
class Thing {
  Thing(this.m, this.v);

  double m;
  double v;
}
```

Notice how the constructor is defined a lot like a method. However, here “Thing” does not represent the name of the constructor, rather the type of object it creates.

Additionally, you can see that we specify some parameters of the constructor within the parameters. To reference an instance variable within the instance itself, you use the keyword “this,” treating it like an ordinary object. By specifying *this.m* and *this.v* as parameters of the constructor, the computer automatically assigns any value provided towards their respective variables.

Optionally, you could also throw some curly braces at the end to run additional code on the creation of the object, but that’s not required in Dart.

Now that we have the class and constructor all set up, we can go ahead and create and reference an object:

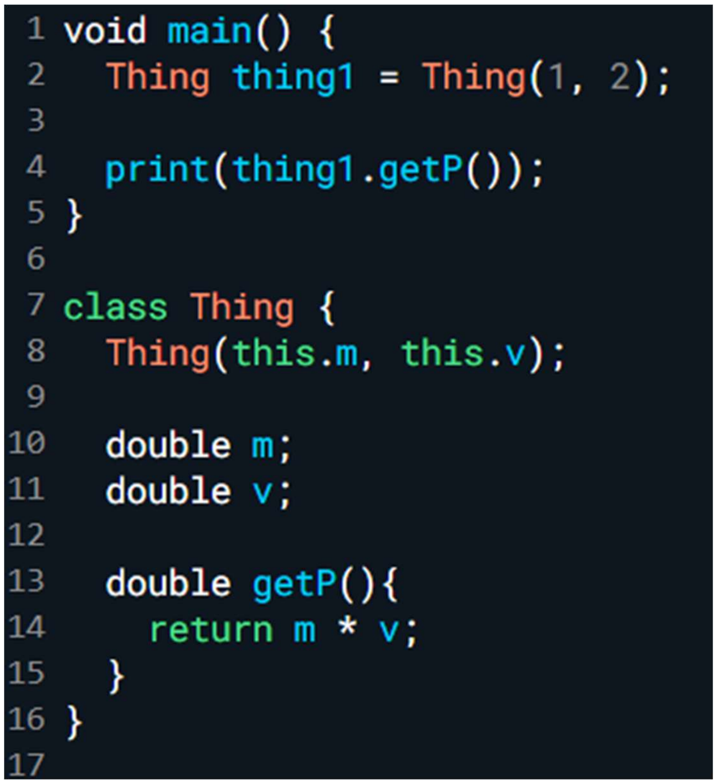


```
void main() {
  Thing thing1 = Thing(1, 2);

  print(thing1.m);
}

class Thing {
  Thing(this.m, this.v);

  double m;
  double v;
}
```

**Output:** 1

As you can see, we can now treat the name of our class as a type of variable (because it is), and we can now create an object using the constructor we specified.

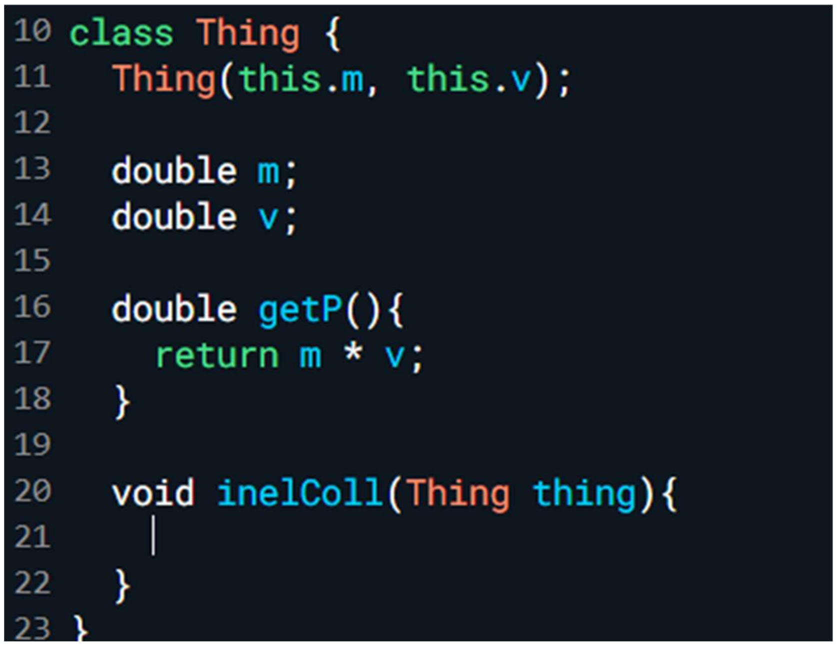
Now, let’s go ahead and create a function within our method to calculate the momentum:



```
void main() {
  Thing thing1 = Thing(1, 2);

  print(thing1.getP());
}

class Thing {
  Thing(this.m, this.v);

  double m;
  double v;

  double getP(){
    return m * v;
  }
}
```

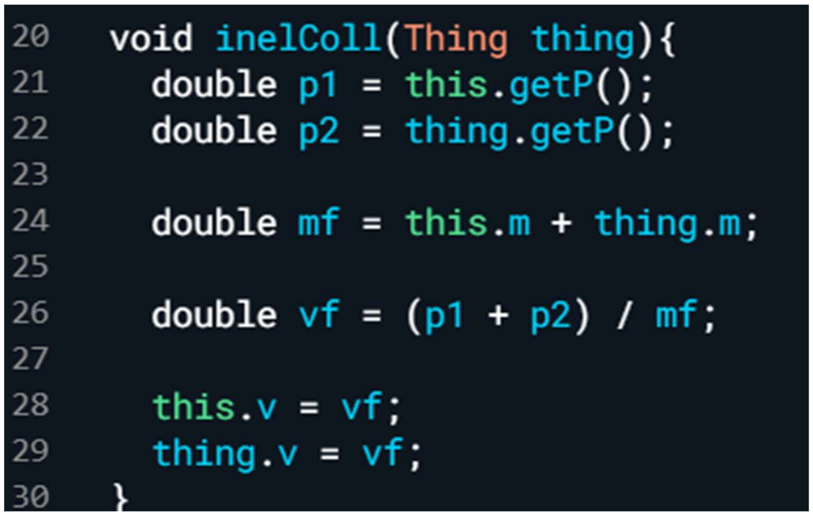
**Output**: 2

With this function defined, we can reference it through any object created through the class, just as we did above. Now, let's start making a method to handle inelastic collisions.



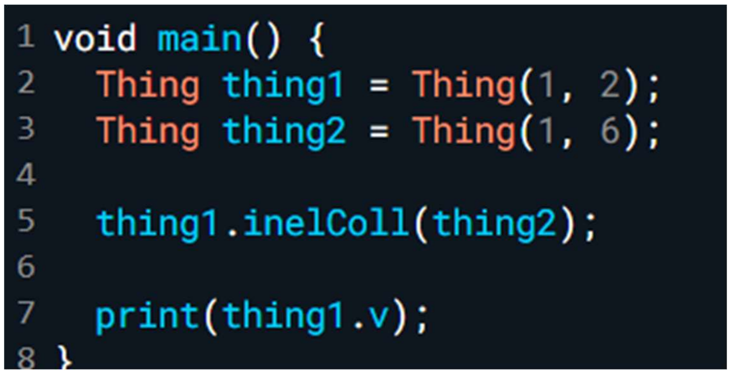
```
class Thing {
  Thing(this.m, this.v);

  double m;
  double v;

  double getP(){
    return m * v;
  }

  void inelColl(Thing thing){
    
  }
}
```

We’ve defined the function “inelColl” (inelastic Collision), and we’ve set our only parameter as another instance of a Thing. Let’s treat the values from this object as “1” and the values from parameter *thing* as “2,” and then plug in our code from earlier:



```
void inelColl(Thing thing){
  double p1 = this.getP();
  double p2 = thing.getP();

  double mf = this.m + thing.m;

  double vf = (p1 + p2) / mf;

  this.v = vf;
  thing.v = vf;
}
```

Here, I’ve added a portion which reassigns the values of our objects at the end. “this” and “thing,” since their collision in inelastic, would share the same velocity at the end. To better understand and test this function, let’s go ahead and collide to things in our code:



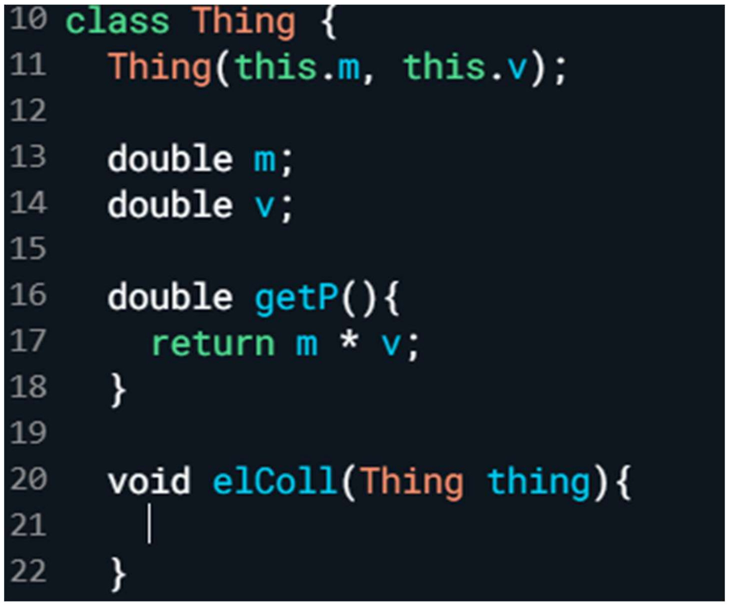
```
void main() {
  Thing thing1 = Thing(1, 2);
  Thing thing2 = Thing(1, 6);

  thing1.inelColl(thing2);

  print(thing1.v);
}
```

**Output:** 4

If you did the math yourself, the velocity would finish at 4, so our program works.

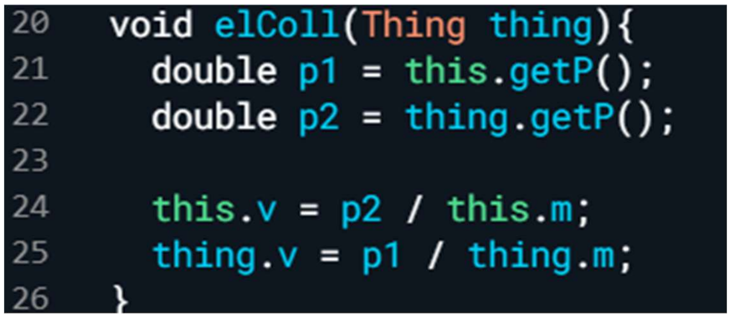
Now, let’s do elastic collisions.

There are many different types of elastic collisions, but in our program we’ll simply have the objects swap momentum. Knowing this, let’s define a new function:



```
class Thing {
  Thing(this.m, this.v);

  double m;
  double v;

  double getP(){
    return m * v;
  }

  void elColl(Thing thing){
    
  }
```

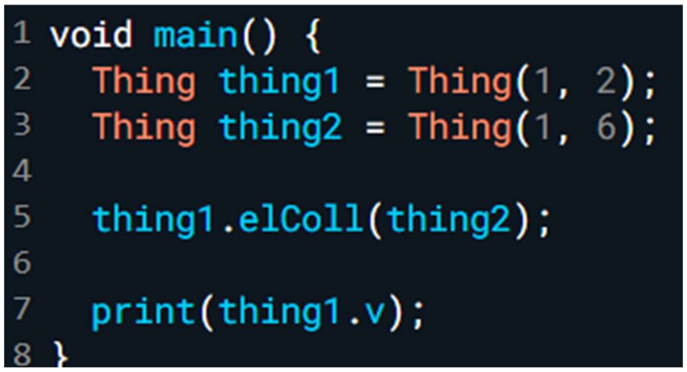
Here, we've defined "elColl" (elastic Collision" just like *inelColl*, and we're ready to begin writing the computations. If we first define the momentums of each, then swap them by reassigning their velocities as their swapped momentum divided by mass, we can get our code to work as intended:



```
  void elColl(Thing thing){
    double p1 = this.getP();
    double p2 = thing.getP();

    this.v = p2 / this.m;
    thing.v = p1 / thing.m;
  }
```

Note: technically, inserting "this" before referencing the *m* or *v* of this object is fully optional in both *elColl* and *inelColl*. I've just done it here for clarity.

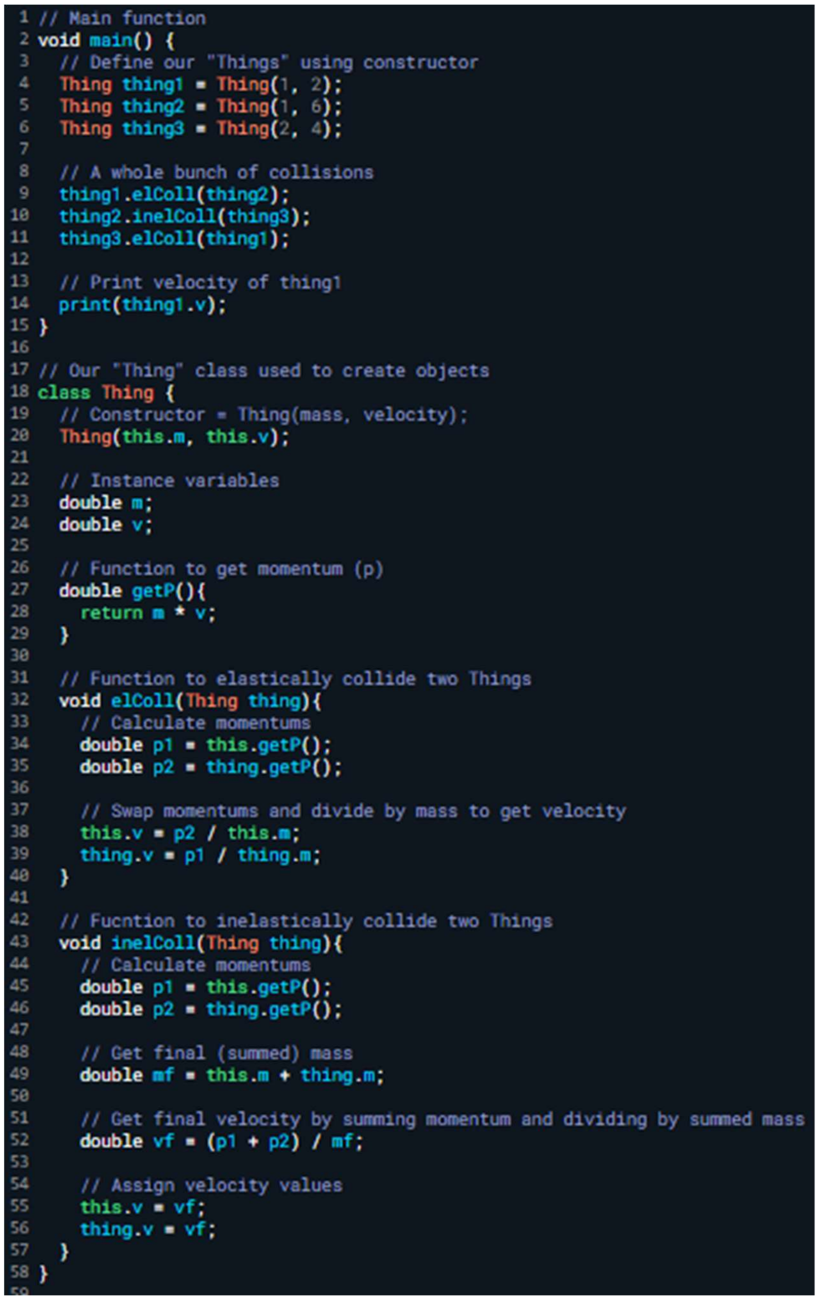
Now that we have this defined, let's go ahead and test it:



```
void main() {
  Thing thing1 = Thing(1, 2);
  Thing thing2 = Thing(1, 6);

  thing1.elColl(thing2);

  print(thing1.v);
}
```

**Output**: 6

This works perfectly. Now, we've got a fully functioning collision calculator—hold on to this to use it in AP Physics! All that's left to do is comment our code for clarity and to have fun colliding different Things:



```
// Main function
void main() {
  // Define our "Things" using constructor
  Thing thing1 = Thing(1, 2);
  Thing thing2 = Thing(1, 6);
  Thing thing3 = Thing(2, 4);

  // A whole bunch of collisions
  thing1.elColl(thing2);
  thing2.inelColl(thing3);
  thing3.elColl(thing1);

  // Print velocity of thing1
  print(thing1.v);
}

// Our "Thing" class used to create objects
class Thing {
  // Constructor = Thing(mass, velocity);
  Thing(this.m, this.v);

  // Instance variables
  double m;
  double v;

  // Function to get momentum (p)
  double getP(){
    return m * v;
  }

  // Function to elastically collide two Things
  void elColl(Thing thing){
    // Calculate momentums
    double p1 = this.getP();
    double p2 = thing.getP();

    // Swap momentums and divide by mass to get velocity
    this.v = p2 / this.m;
    thing.v = p1 / thing.m;
  }

  // Fucntion to inelastically collide two Things
  void inelColl(Thing thing){
    // Calculate momentums
    double p1 = this.getP();
    double p2 = thing.getP();

    // Get final (summed) mass
    double mf = this.m + thing.m;

    // Get final velocity by summing momentum and dividing by summed mass
    double vf = (p1 + p2) / mf;

    // Assign velocity values
    this.v = vf;
    thing.v = vf;
  }
}
```

**Output**: 6.7

# And we're done! Good job!