Fields

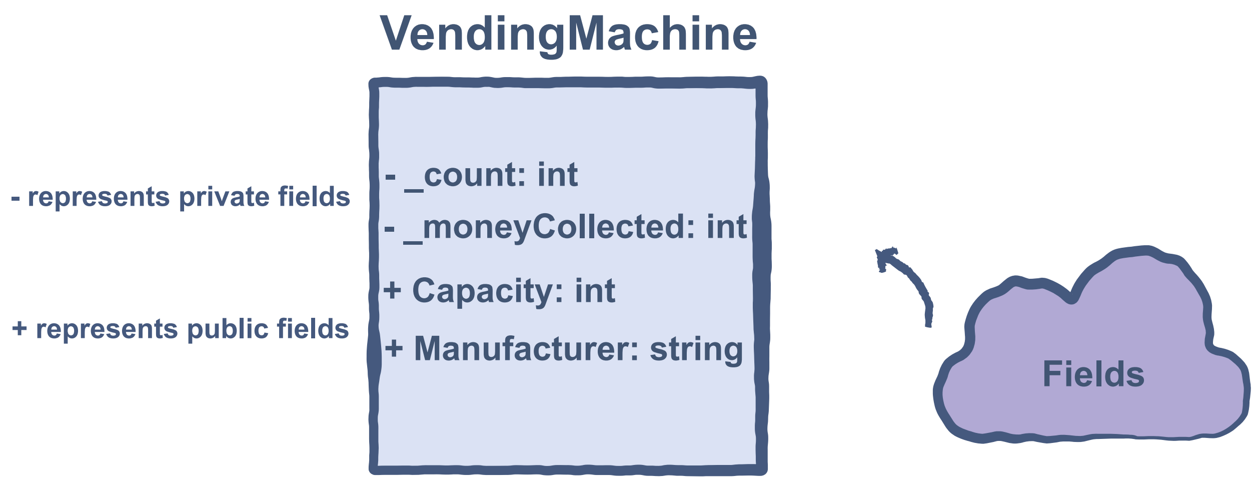
This lesson will go into the details of the member variables or fields of a class.

WE'LL COVER THE FOLLOWING

Fields

As discussed earlier, fields are actually the *member variables* inside a class. For instance, in a class representing a vending machine, the `VendingMachine` class might contain the following fields:

- A `count` of type `int` that stores the count of products available in the machine.
- A `capacity` of type `int` that stores the maximum number of products the machine can have.
- A `moneyCollected` of type `int` to store the money it has collected.
- A `manufacturer` of type `string` to store the name of the manufacturer of the machine.



Class Diagram

The fields in the `VendingMachine` class could be defined like this:

```
public class VendingMachine
{
    private int _count;
    private int _moneyCollected;
    public int Capacity;
    public string Manufacturer;
}
```

As it has been already mentioned that declaring `public` fields is not a good idea when it comes to the security of the code but for now we are doing so just for understanding the concept.

Types of Fields

Now apart from the access modifiers, three other things can determine the type of a field:

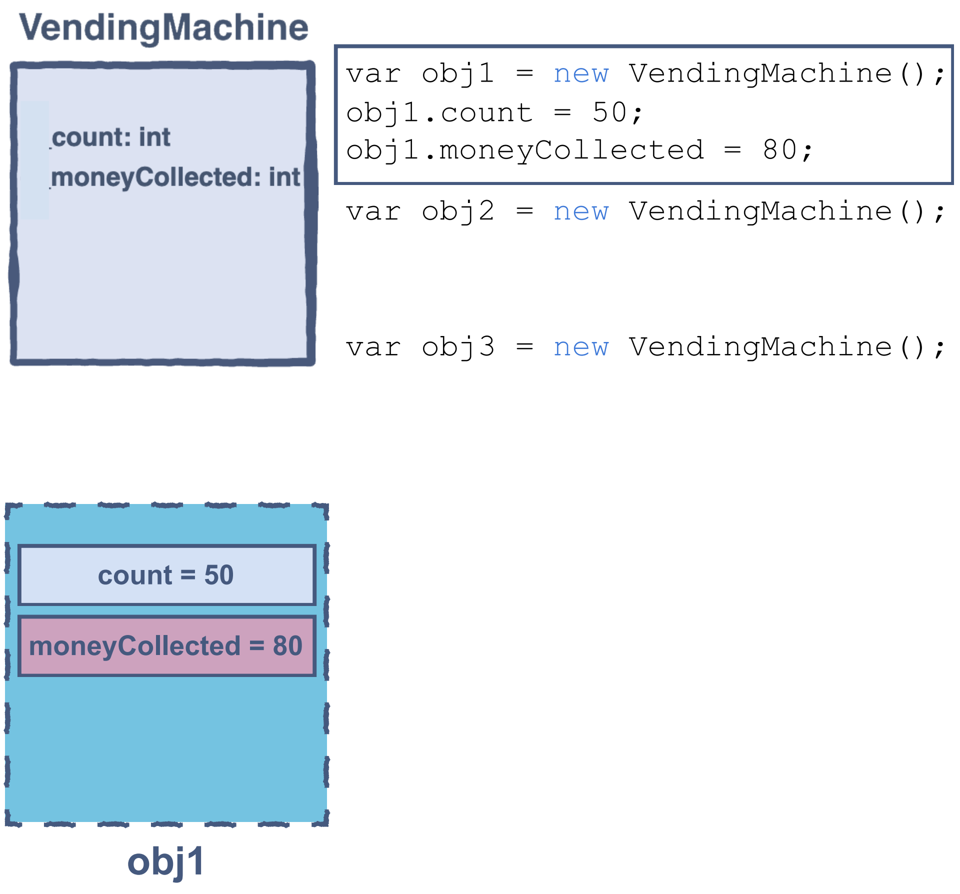
- *Non-Static*
- *Static*
- *Read Only*

Non-Static Fields

We have already discussed that whenever we need to create an object, we use the `new` keyword. This lets the compiler know about the creation of an object and it allocates the required memory to that object.

The question with the creation of every `new` object is, “What does the newly allocated memory location hold?” The answer is that every object has its own copy of the non-static class members. These non-static members reside in the memory allocated to that object.

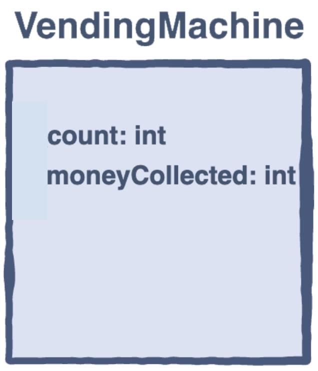
Each object of the class can have its own values for non-static fields. Let’s try to grasp this concept with the help of the following illustrations:



```
var obj1 = new VendingMachine();
obj1.count = 50;
obj1.moneyCollected = 80;
var obj2 = new VendingMachine();


var obj3 = new VendingMachine();
```

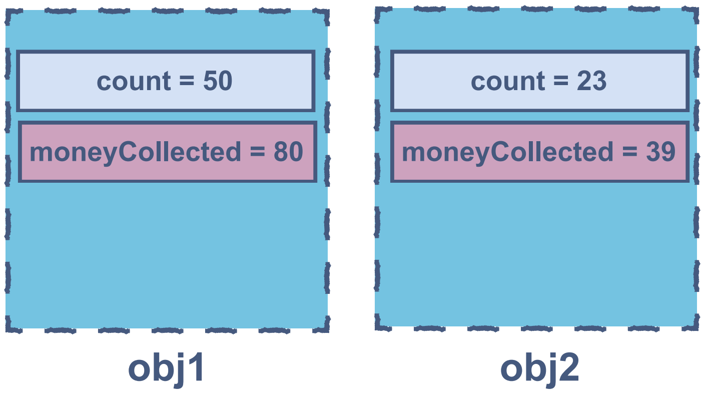
1 of 3

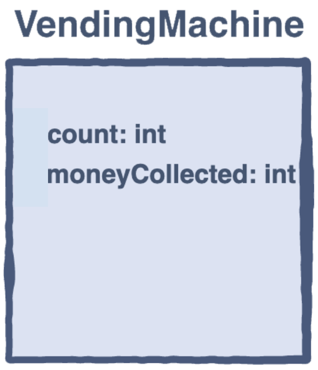


```
var obj1 = new VendingMachine();
obj1.count = 50;
obj1.moneyCollected = 80;

var obj2 = new VendingMachine();
obj2.count = 23;
obj2.moneyCollected = 39;

var obj3 = new VendingMachine();
```

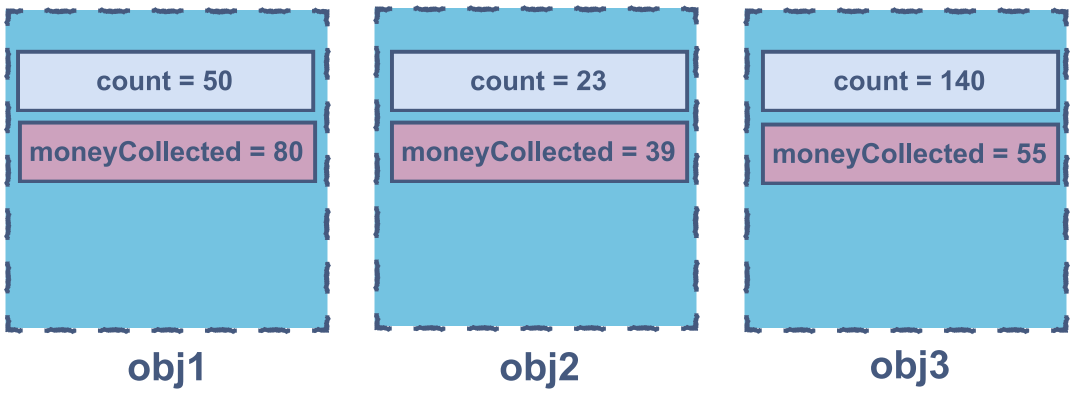




```
var obj1 = new VendingMachine();
obj1.count = 50;
obj1.moneyCollected = 80;
var obj2 = new VendingMachine();
obj2.count = 23;
obj2.moneyCollected = 39;

var obj3 = new VendingMachine();
obj3.count = 140;
obj3.moneyCollected = 55;
```

All the objects have their own copies of non-static fields.
All the objects can set their own values of non-static fields.



Since we are simulating a vending machine by implementing its class, it wouldn't have made sense if each newly created vending machine couldn't have its own count of available products and collected amount of money. That's where non-static fields come in handy.

Until now, we have been declaring `public` non-static fields in our class implementations. The declaration is very simple.

```
class VendingMachine {

  // Non-Static Fields
  public int count;
  public int moneyCollected;

}
```

> The fields are intentionally declared `public` for now.

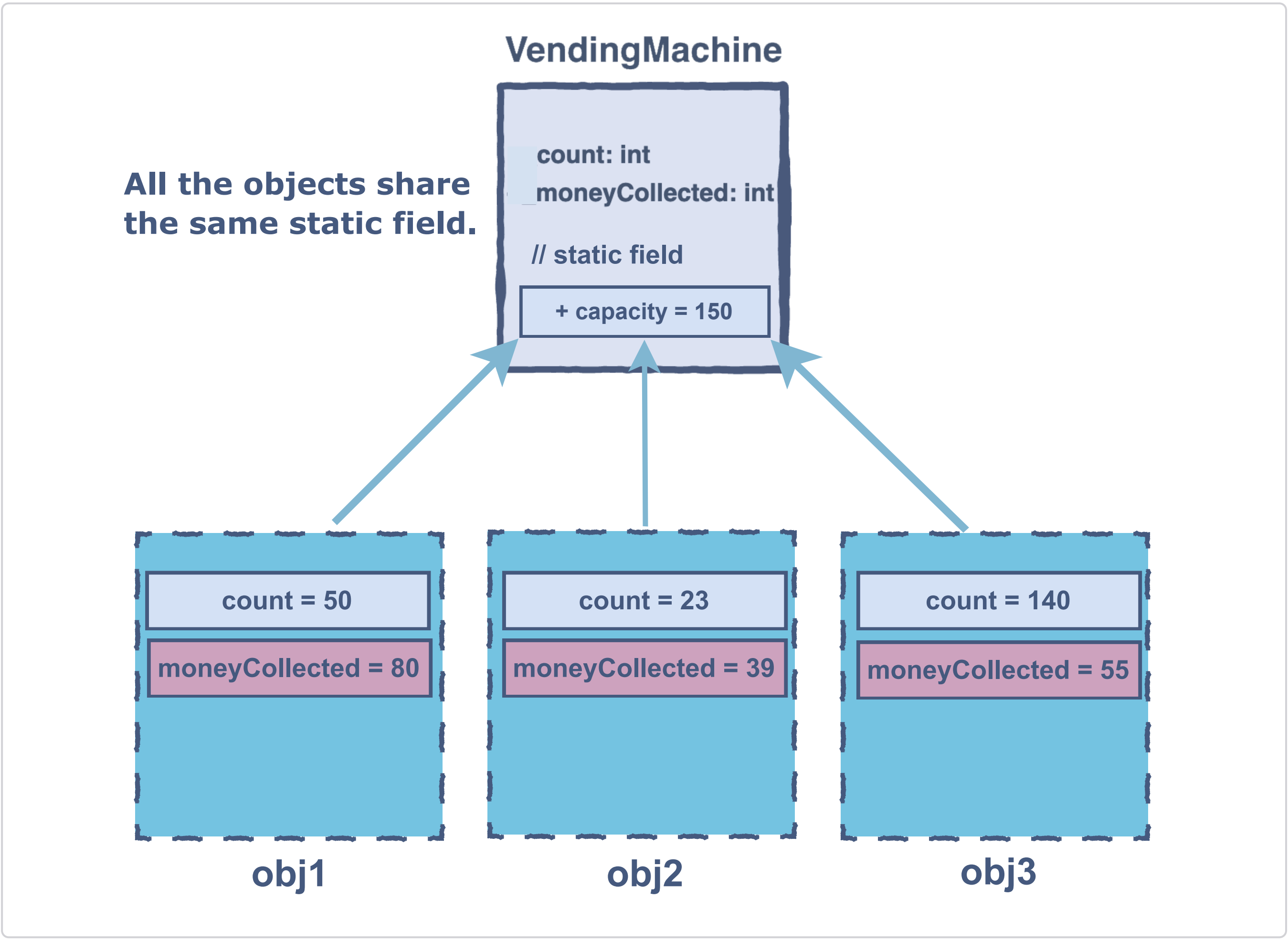
As every object has its own copy of non-static fields, we need an object of the class to access non-static fields.

```
var obj1 = new VendingMachine();

Console.WriteLine(obj1.count);
Console.WriteLine(obj1.moneyCollected);
```

Static Fields

`static` is a keyword in C#. A static field resides in a class. All the objects/instances of this class will share this field and its value.



Now talking about creating the objects of `VendingMachine` class, a similar model will have a similar capacity which means that every machine doesn't need to have its own `Capacity` variable.

You can define a static field by using the `static` keyword in C#:

```
class VendingMachine {

  // static field
  public static int capacity;

  // Non-Static Fields
  public int count;
  public int moneyCollected;

}
```

Static fields reside in the class. We don't need an instance of the class to access static fields. We can access the static fields of a class by just writing the class name before the field:

```
// Static fields are accessible in the main without creating an object
```

```
Console.WriteLine(VendingMachine.capacity); //called using class' name
```

`readonly` Fields

A `readonly` field must be assigned a value in the constructor of the class or during initialization. It cannot have its value changed once it is assigned. We can make a field read only by using the keyword `readonly`.

Here is an example in C#:

```
class VendingMachine {
  // readonly field of manufacturer = "Vendy Inc."
  // Now manufacturer of the machine can nerver be changed
  // to some other value throught the program
  public readonly string Manufacturer;

}
```

`VendingMachine` has a fictitious manufacturer company named *Vendy Inc.* which can’t be changed. If you try to do so, you will get a compilation error.

You can check this on your own in the following code widget:

```
class VendingMachine {

  // Readonly variable manufacturer
  public readonly string Manufacturer;

  public VendingMachine(string s){
     Manufacturer = s; // Assigning a value to readonly member
     Console.WriteLine("Manufacturer is {0}", Manufacturer);
  }

}

class Demo {

   public static void Main() {

      var vendingMachine = new VendingMachine("Vendy"); // Manufacturer is Vendy
      vendingMachine.Manufacturer = "Some other company"; // This will give an error
   }

}
```

The `readonly` fields, once declared, should always be initialized otherwise the compiler will show a warning message.

`const` Fields

`const` fields are assigned values during their initialization. Unlike `readonly` fields, values are not assigned in the constructor.

Once a value has been assigned to a `const`, it cannot be modified. `const` fields are `static` by default.

```
class VendingMachine {

  // Readonly variable manufacturer
  public const string Manufacturer = "Vendy";

}
```

> **Note**: The value of a `const` field is known at compile time, whereas the value of a `readonly` field is computed when an instance is made.

In the next lesson, we'll discuss methods in C#.