



## The "this" Keyword

A method invocation, like "kayla.introduce()", implicitly passes an object (in this example, kayla) as a parameter called "this". So we can rewrite our last constructor as follows without changing its meaning.

```
public Human() {
    this.age = 0;
    this.name = "Untitled";
}
```

In this case, "this" is optional. However, if the parameters or local variables of a method have the same name as the fields of an object, then the former have priority, and the "this" keyword is needed to refer to the object's fields.

```
public void change(int age) {
    String name = "Meng";

    this.age = age;
    this.name = name;
}
```

When we call "kayla.change(8)", "this" is assigned the same value as "kayla" before the change() method begins execution.

<pre>kayla  .+----&gt;         -----          age  12           -----          name   -+--+----&gt;  Kayla          .+----&gt;         ----- </pre>	<pre>Parameters &amp; local variables        -----          age   8           -----          name   -+-----&gt;  Meng          ----- </pre>
---	---

Now, when Java executes "this.age = age", it overwrites the 12 with an 8. When Java executes "this.name = name", it overwrites kayla's name as below.

<pre>kayla  .+----&gt;         -----          age   8           -----          name   -+--+----&gt;  Kayla          .+----&gt;         ----- </pre>	<pre>Parameters &amp; local variables        -----          age   8           -----          name   -+-----&gt;  Meng          ----- </pre>
---	---

-----  
IMPORTANT: You CANNOT change the value of "this"!

A statement like "this = kayla;" will trigger a compile-time error.

## The "static" Keyword

A `_static_field_` is a single variable shared by a whole class of objects; its value does not vary from object to object. For example, if "numberOfHumans" is the number of Human objects that have been constructed, it is not appropriate for each object to have its own copy of this number; every time a new Human is created, we would have to update every Human.

If we declare a field "static", there is just one field for the whole class. Static fields are also called `_class_variables_`.

```
class Human {
    public static int numberOfHumans;

    public int age;
    public String name;

    public Human() {
        numberOfHumans++; // The constructor increments the number by one.
    }
}
```

If we want to look at the variable numberOfHumans from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object.

```
int kids = Human.numberOfHumans / 4; // Good.
int kids = kayla.numberOfHumans / 4; // This works too, but has nothing to
// do with kayla specifically. Don't
// do this; it's bad (confusing) style.
```

System.in and System.out are other examples of static fields.

Methods can be static too. A `_static_method_` doesn't implicitly pass an object as a parameter.

```
class Human {
    ...
    public static void printHumans() {
        System.out.println(numberOfHumans);
    }
}
```

Now, we can call "Human.printHumans()" from another class. We can also call "kayla.printHumans()", and it works, but it's bad style, and kayla will NOT be passed along as "this".

The main() method is always static, because when we run a program, we are not passing an object in.

-----  
IMPORTANT: In a static method, THERE IS NO "this"!

Any attempt to reference "this" will cause a compile-time error.

## Lifetimes of Variables

- A local variable (declared in a method) is gone forever as soon as the method in which it's declared finishes executing. (If it references an object, the object might continue to exist, though.)
- An instance variable (non-static field) lasts as long as the object exists. An object lasts as long as there's a reference to it.
- A class variable (static field) lasts as long as the program runs.