The “`static`” Modifier

Java’s **`static`** modifier, when used appropriately, can simplify the design of certain aspects of a program. Used inappropriately, however, it can result in disaster. In this brief addendum I hope to clear up any confusion you might have about the **`static`** modifier. Feel free to send me an email if you have any questions or comments.

What does “static” mean?

Generally speaking, when a construct is declared as being **`static`** it means there is only one copy of that construct regardless of how many instances are created of the class that contains the **`static`** construct. I’ll explain why this can be useful later.

What can be declared “`static`”?

The **`static`** modifier can be used in the declaration of the following Java constructs:

1. Fields (i.e., attribute variables and constants)
2. Methods (except constructors)
3. Nested classes (but not inner classes)
4. Static Initializers (code blocks present in a class that are not part of a method or constructor)

The last two constructs listed above may not be familiar to you. You will see examples of nested classes in this course, so I will cover them later. Static initializers are rarely used, and we won’t be using them in this course, so I won’t talk about them here.

The **`static`** modifier **cannot** be used in the declaration of the following Java constructs:

1. Constructors
2. Classes
3. Interfaces
4. Inner classes (distinguished from nested classes)

Static fields and methods

If a class-scope field (i.e., a variable or a constant declared within a class but outside any code blocks) is declared as being static, it means there is only one copy of that field regardless of how many instances of the enclosing class are created. If there are multiple instances of the enclosing class, they all reference the single copy of the field.

To get a better feel for this, consider the following example. Suppose you have a class representing a poker player called `Player`, with an attribute called `hand`, which represents a hand of cards. If `hand` is declared as non-static, each instance of the class `Player` will have its own copy of `hand`, as shown in Figure 1a. If `hand` is declared as `static`, however, every instance of `Player` that is created will refer to the same copy of `hand`, as shown in Figure 1b.

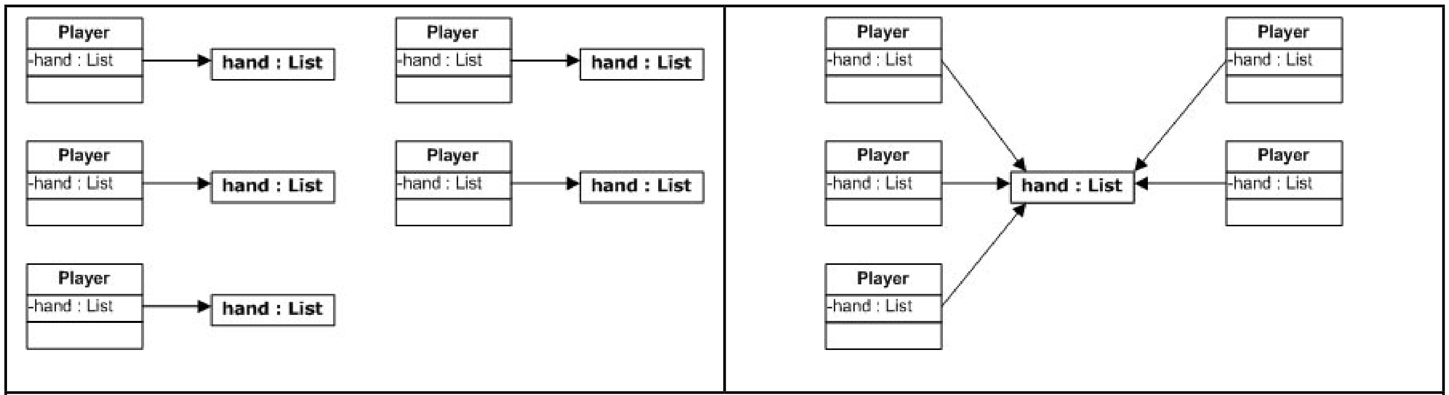

Figure 1. Illustration of the difference between a static and non-static field. Panel A shows a field called size, declared as non-static. Panel B shows the same field declared as static.

What is the significance of this? When `hand` is declared as being non-`static`, each instance of `Player` possesses its own copy of `hand`, and can change the value of its copy of `hand` independently of the other instances of `Player`. In other words, each player has its own `hand`, which makes sense for most games of poker. When `hand` is declared as being `static`, all instances of `Player` reference the single copy of `hand`. In this case, all the players share a common hand, which is not the norm for most variations of poker, although some variations such as Texas Hold'em do have community cards that all players may use.

So, one of the main advantages to declaring a field `static` is if it makes sense to do so in the design of a project. If all the instances of a particular class need to use a common field, it may be a good idea to make that field `static`. It enforces the concept of using shared data, and eliminates redundant copies of that data. There is one potential drawback to declaring a field `static`, however, and that is that `static` fields can only be accessed via `static` methods. For example, the getter and setter methods for the `static` field must be declared as `static` methods, as does any other method that references the `static` field in its implementation. If the `static` field is widely used, it can result in too many `static` methods, many of which will not work correctly if they are declared `static`.

The reverse is also true: static methods cannot reference non-`static` fields. The reason why is fairly straightforward. As with `static` fields, only one copy of a `static` method exists for all instances of the method's enclosing class. If this method were to attempt to reference a non-`static` field, it would have no idea which copy of the non-`static` field it should use. Suppose for the example in Figure 1a above the `Player` class also has a static method called `getHand()`, which simply returns the value of hand. All instances of `Player` would use the one copy of `getHand()`. However, when the method is invoked, which player's hand should it return? There is no way for it to determine which hand it should return, since it could be any of the five hands shown in the Figure 1a. If `hand` is `static`, on the other hand, as in Figure 1b, the ambiguity is resolved since there is only one hand that can be returned.

A more common use of static fields is in the declaration of constants such that they can be accessed without the need for an explicit instance of the class that contains the constant. In Figure 1b, assuming the `hand` field is declared as public, can be accessed using the syntax:

```
Player.hand
```

as opposed to creating an instance of `Player` and invoking a getter method, as in:

```
Player p = new Player();
List h = p.getHand();
```

Although this avoids the necessity of invoking methods, the `static` modifier should not be overused. Proper consideration must be given to software design. Attributes should only be declared `static` if one or more of the following are true:

1. The attribute must be accessed globally. Widespread use of global attributes is generally considered dangerous, and not part of good programming practice. However, there can be instances where a global attribute makes sense; it depends on the project and the design.

2. The attribute will likely need to be accessed in the absence of an explicit instance of the constant's class.

Likewise, methods should only be declared as `static` under appropriate conditions, such as:

1. The method must access a static attribute that cannot be accessed globally (e.g., a `private static` attribute).

2. The method will likely need to be accessed in the absence of an explicit instance of the constant's class.

The `main` method

The first **`static`** method everyone learns when learning Java is the `main` method used to launch a Java program. The basic method signature is:

```
public static void main (String[] args){}
```

Since the method is **`static`**, there can only be one `main` method in use for a particular program. Even though it is possible to include multiple `main` methods in a project (but only one `main` method per class), one of those methods must be chosen to be the one that launches a program just prior to runtime. Any other `main` methods are ignored.

One common problem many Java programmers encounter occurs when they include the `main` method in one of their project-specific classes. The error that appears is along the lines of:

"Cannot make a static reference to a non-static field"

The immediate notion is to convert the offending non-**`static`** field into a **`static`** field. Unfortunately, this results in virtually every method and field in the entire project being declared **`static`**, which is not good.

The best solution to this problem is to place the program's `main` method into a separate class all by itself, with no other methods or attributes. From a design perspective this is reasonable, since the `main` method isn't really related to any particular project, so there is no good reason to incorporate it into any project-specific class. The `main` method is simply an artifact of the Java language, used to instruct the Java Run-time Environment where to begin execution when a program is launched.

The following is an example of how to use a separate class for the `main` method. You should choose an appropriate name for the class that easily distinguishes it from the other classes in your project. Some common class names I have seen are: "`Driver`", "`Runner`", and "`MainClass`".

```
public class MainClass
{
      public static void main(String[] args)
      {
            // here, create instances of whatever classes are necessary to launch
            // the program

            // next, invoke the method that launches the program
      }
}
```

As you can see, there isn't much there. The comments indicate what needs to happen inside the `main` method. First, create instances of any classes that must be present prior to launching the program. These will include the class that contains the first method that must be invoked, along with any supporting classes needed. Once those class instances have been created, invoke the method that launches the program.

If you use a separate class to store your `main` method, like the one shown above, you shouldn't have any more problems with static references trying to access non-static fields.