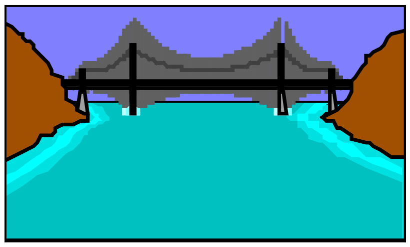
Unit 4: Client View of a Component — Methods

Preview of Coming Attractions

In this unit be sure to look for

- method/operation
- parameters/formal parameters
- arguments/actual parameters
- method header/method signature/method prototype
- updates parameter mode
- method contract
- requires clause/ensures clause
- distinguished parameter/distinguished argument/receiver
- procedure and function
- tracing tables
- how to call functions

Where We've Been, Where We're Going



We are learning about a proper way to *describe* components to component clients. Here's what we know so far:

- Clients need to understand *what* a component can do and *how* to use it, and do not need to understand how the component does what it does.
- To use a component, clients declare *variables* described by a component type, initialize the variables with a value of the component type, and then use the variables.
- To understand a variable and how to use it, clients need to understand the interface; that is, the *values* the variable can assume and the *methods*, sometimes called *operations* that can be applied to the variable/value.
- To describe the values a variable can assume, we define a *mathematical model* for the abstract state space of the component type (the abstract values the variable can store).

That's where we've been, using an am-pm-clock component as a working example. What we'll do next is to learn about a proper way to describe the methods that can be applied to a variable, and we will continue to use the clock example. When that's done, we'll have a good understanding of how components are described to clients.

The Before and the After

When components come from a catalog, they come equipped with a set of methods that can be applied to variables described by the component. Generally, methods are used to change the values of variables, so what clients need to understand is how a method changes the values of variables. (It should be as plain as day that there is no way that clients can understand the changes methods make to the values of variables if they don't understand what values the variables can assume! That's why we first carefully define the mathematical type that is used to model the programming type for the component.)

The essential idea of a method is simple:

- A method has a fixed number of *parameters*, sometimes called *formal parameters* or just *formals*.
- Clients supply a method with *"incoming*" values (often in the form of variables), called *arguments*, sometimes called *actual parameters*, which correspond to the parameters.
- Depending on the incoming values, the method performs its task and may change the values of the variables resulting in *"outgoing"* values for some of the client's variables.

That's it. So, our descriptions of methods will describe:

- the number and types of the formal parameters
- the allowable incoming values of the client-supplied variables
- the resulting outgoing values of the client-supplied variables.

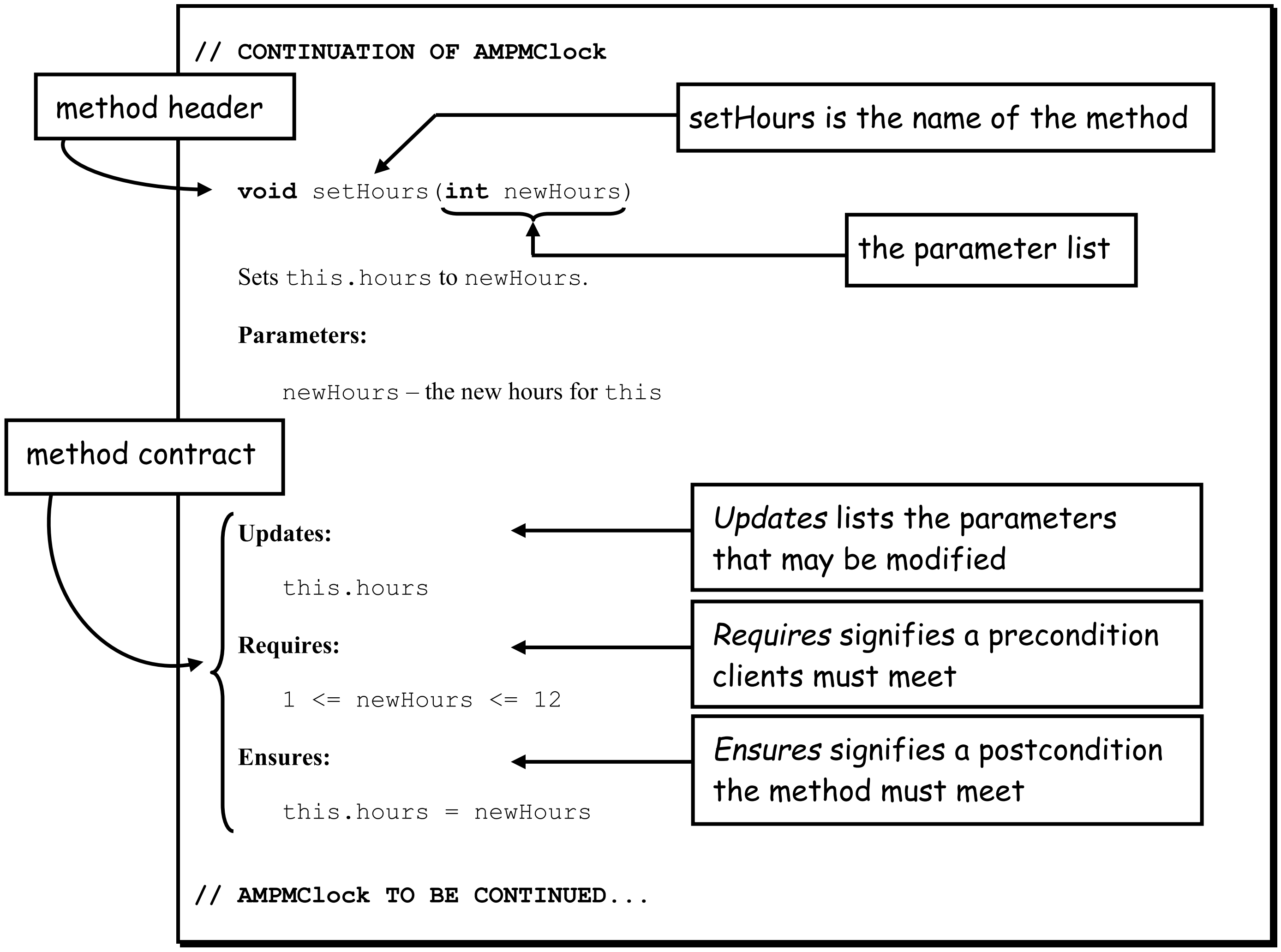
Still Awake?

4-1. Carefully consider what kinds of methods you'd like to have available for a clock variable. What do you think a client should be able to do with a clock variable?

Describing Methods – An Example

The first method we'll look at is a method that can be used to set the hours of a clock variable. The figure titled "Client Description of `setHours` for `AMPMClock`" provides the description for this method.

Client Description of setHours for AMPMClock



```
// CONTINUATION OF AMPMClock

    void setHours(int newHours)

    Sets this.hours to newHours.

    Parameters:

        newHours – the new hours for this

    Updates:

        this.hours

    Requires:

        1 <= newHours <= 12

    Ensures:

        this.hours = newHours

// AMPMClock TO BE CONTINUED...
```

method header
setHours is the name of the method
the parameter list
method contract
Updates lists the parameters that may be modified
Requires signifies a precondition clients must meet
Ensures signifies a postcondition the method must meet

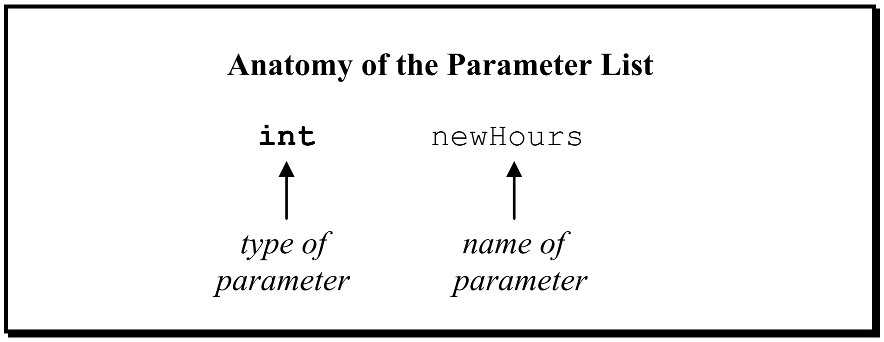
There is quite a bit to understand in this little method description, so we'll take our time and discuss it carefully.

The Method Header

The line

```
void setHours(int newHours)
```

is called the ***method header*** or ***method signature*** or ***method prototype***. It specifies the name of the method and the parameter list. The name is `setHours`, appearing just after the keyword **`void`**. (The keyword **`void`** indicates that this method is a ***procedure***, i.e., it does not return a value explicitly.) In this case the parameter list contains a single parameter and has the following structure:



Note that there is no code here. The actual implementation of the method `setHours` will appear in some other component (a class). This is just what we want, since the component `AMPMClock` presents a client view of the method `setHours`, and clients do not need to know how methods do what they do. (The code that actually implements a method is called the ***method body***.)

The Method Contract

The second crucial piece of the client description of `setHours` is the specification

Updates:

```
  this.hours
```

Requires:

```
  1 <= newHours <= 12
```

Ensures:

```
  this.hours = newHours
```

describing a ***contract*** between clients of `setHours` and the `setHours` method itself. To get you into the spirit of things, here is another example of a contract:

requires

students show up for class, stay awake, and ask questions

ensures

the instructor delivers an incredible lecture

Contracts are agreements between clients of a service and providers of the service. Contracts have two parts: the ***requires clause*** specifies obligations that clients must meet and the ***ensures clause*** specifies obligations providers must meet. There is a little catch, however. Providers are obligated to meet their obligations *only if* clients meet their obligations. In the second example, the obligation of the students (the client) is to come to the class, to stay awake, and to ask questions. *If* students meet their obligations, the instructor (the provider) is then obligated to deliver an incredible lecture. On the other hand, if the students do not show up for class or fall asleep in class or sit there like bumps on logs, the instructor can do anything she wants, since she is no longer bound to her obligations. So, the instructor might deliver an incredible lecture, or she might tell the story of Goldilocks and the Three Bears, or she might go to aerobics.

In the case of methods, the *requires clause* specifies conditions that *incoming* parameter values must satisfy. Since clients provide incoming values, this is an obligation of the client. The *ensures clause* specifies conditions that *outgoing* parameter values must meet. This is the obligation of the method. Of course, if a client provides incoming parameter values that do not satisfy the requires clause, then the method can supply anything it wants for outgoing values, or it might even fail to terminate its execution!

As you might expect, the operation `setHours` can be used to set the hours of an am-pm clock. Clients supply the desired new value for the hours through the formal parameter `newHours`. This explains the client obligation for the incoming value of `newHours`:

Requires:

```
1 <= newHours <= 12
```

Assuming that a client supplies a good value for `newHours`, method `setHours` should change the value of the clock so that its hours are the same as the incoming value of `newHours`. *But what clock variable is to be changed? There is no clock parameter in the parameter list!* Well, at least not explicitly. The operation `setHours` is specified in the interface `AMPMClock` and is considered to be an *instance method*. Instance methods must be applied to variables of a component type. This means that clients must invoke the operation `setHours` with a statement like `myClock.setHours(myHours)` or like `yourClock.setHours(yourHours)`, where `myClock` and `yourClock` are variables of a component type, described by `AMPMClock`, to which `setHours` will be applied. (More intuitively, they are the clocks whose values are to be changed.) So, `myClock` and `yourClock` are actually additional arguments in the call to method `setHours`, but they appear just before the name of the method. In the contract for a method, the *distinguished parameter* is implicit and always goes by the name `this`. In a client call, the *distinguished argument* appears explicitly (e.g., `myClock` or `yourClock` above). The distinguished argument is also called the *receiver* of the method call.

We're all set to examine the obligation of the `setHours` method:

Ensures:

```
this.hours = newHours
```

The assertion `this.hours = newHours` just says that the outgoing value of `this.hours` will be the (outgoing) value of `newHours`. (Remember that `this` refers to a variable described by `AMPMClock` and that values of variables described by `AMPMClock` are 4-tuples.) But, you might ask, what about the outgoing values of `this.minutes`, `this.seconds`, and `this.am`? Clearly `setHours` is not supposed to change the minutes, seconds, and am/pm indicator of the given clock and our contract would be incomplete if we did not explicitly state that they cannot change. Here is where a convention comes into play. *Any arguments that might be changed by a method must be listed in the contract under a heading that describes how they might be changed*. Observe these lines in the contract:

Updates:

```
   this.hours
```

Updates is known as a *parameter mode*. It simply says that the parameters listed after it may be modified by the method and usually we'll look at the ensures clause to see how they might be modified. In this case, the only argument listed is `this.hours` and this indicates that `setHours` is only allowed to modify the value of `this.hours`. In other words, the other arguments (or *parts* of the other arguments) must have the same outgoing value as their incoming value: `this.minutes`, `this.seconds`, `this.am`, and `newHours` all have the same value after the call to `setHours` that they had before the call. Again, this is a convention that we will follow in all our descriptions of components. (Later on we will see that there are a few other parameter modes, e.g., *Replaces* and *Clears*, describing other kinds of modifications that can occur to parameters.)

There is one more important aspect of the contract for `setHours` that we should discuss. Whenever a parameter is mentioned in the requires clause, the name clearly refers to the incoming value of the parameter. For example, in the assertion `1 <= newHours <= 12`, we are imposing a constraint on the value provided by the client, i.e., the *incoming* value of `newHours`. The requires clause never talks about the outgoing value of any of the parameters. However, in the ensures clause where we usually want to describe the outgoing values of those parameters that can be modified by the method, sometimes we may need to refer to the incoming value of some of the parameters as well. How can we distinguish an incoming value from an outgoing value? We simply add a '`#`' symbol at the front of the name of the parameter to refer to the *incoming* value, and assume that when no '`#`' is present, we are referring to the *outgoing* value. As an example of this notation, if we had wanted to explicitly state in the ensures clause of `setHours` that the value of `this.minutes` was not going to change, we could have written it as

```
this.minutes = #this.minutes
```

simply saying that the outgoing value of `this.minutes` is equal to the incoming value of the minutes of `this`, written as `#this.minutes`.

Still Awake?

4-2. How do we know that variables described by `AMPMClock` have values that are 4-tuples?

4-3. Do you think that requires clauses are concerned with incoming values or with outgoing values of parameters? Explain. If your answer is incoming values, explain why the requires clause for `setHours` is not specified as `1 <= #new_hours <= 12`.

4-4. Suppose that `myClock = (11,25,48,`**`true`**`)` and that `newHours = 3`. What will be the value of `myClock` and `newHours` after the method call `myClock.setHours(newHours)`?

Three More "Set" Methods

An am-pm-clock component will need also methods to set the minutes, seconds, and am parts of a clock variable. Here are the specifications of two of these methods. The third method is left for you to specify.

Client Description of setMinutes and setAM Methods

// CONTINUATION OF AMPMClock

void `setMinutes(`**int** `newMinutes)`

Sets `this.minutes` to `newMinutes`.

Parameters:

`newMinutes` – the new minutes for `this`

Updates:

`this.minutes`

Requires:

`0 <= newMinutes <= 59`

Ensures:

`this.minutes = newMinutes`

// --

void `setAM(`**boolean** `am)`

Sets `this.am` to `am`.

Parameters:

`am` – the new am for `this`

Updates:

`this.am`

Ensures:

`this.am = am`

// AMPMClock TO BE CONTINUED...

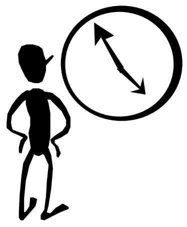
Still Awake?

4-5. Give the complete client description of `setSeconds`.

4-6. Suppose that `myClock = (11,25,48,`**`true`**`)` and that `newMinutes = 31`. What will be the value of `myClock` and `newMinutes` after the method call `myClock.setMinutes(newMinutes)`? How about after the method call `myClock.setMinutes(52)`?

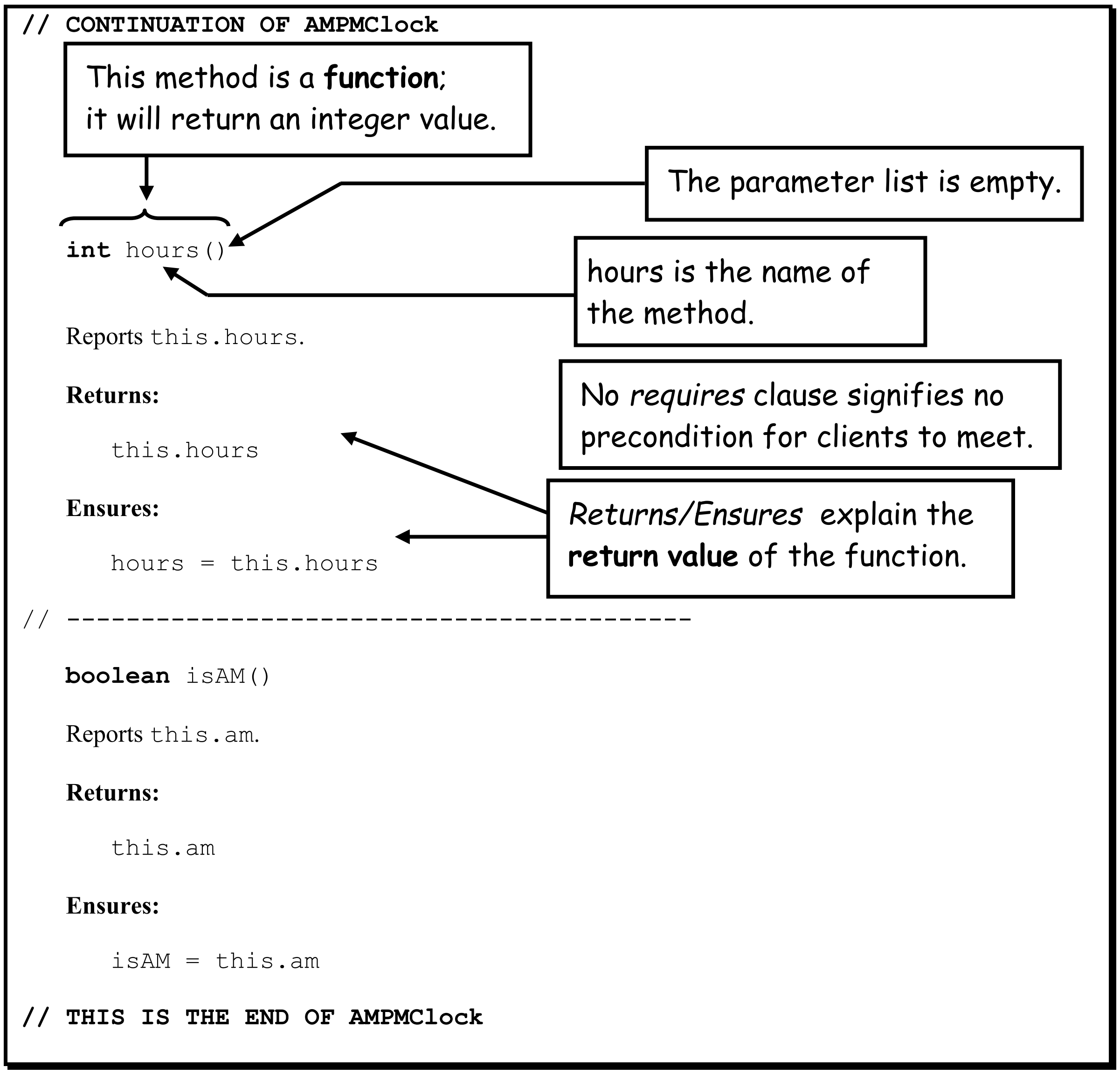
4-7. Suppose that `myClock = (11,25,48,`**`true`**`)` and `am =` **`true`**. What will be the value of `myClock` and `am` after the method call `myClock.setAM(am)`?

What Time Is It?



We're almost finished with the first complete example of a client description of a component. So far, the clock methods include `setHours`, `setMinutes`, `setSeconds`, and `setAM`. With this selection of methods, clock variables can be assigned any legal clock value, as specified in `AMPMClock`. But suppose you would like to know the time according to `yourClock`, where `yourClock` is an `AMPMClock` variable. How can you find out? None of the current methods allow us to inspect or observe the value of an `AMPMClock` variable. For this purpose, we need four additional methods: `hours`, `minutes`, `seconds`, and `isAM`. Following are the client descriptions of `hours` and `isAM`. We'll let you take care of `minutes` and `seconds`.

Client Description of hours and isAM Methods



```
// CONTINUATION OF AMPMClock

   int hours()

   Reports this.hours.

   Returns:

      this.hours

   Ensures:

      hours = this.hours

// ----------------------------------------

   boolean isAM()

   Reports this.am.

   Returns:

      this.am

   Ensures:

      isAM = this.am

// THIS IS THE END OF AMPMClock
```

This method is a **function**; it will return an integer value.

The parameter list is empty.

hours is the name of the method.

No *requires* clause signifies no precondition for clients to meet.

Returns/Ensures explain the **return value** of the function.

As usual, the description of these methods involves some new stuff. We'll work through the `hours` method; `isAM` is similar.

The function header is

```
int hours()
```

Each of the "set" methods such as `setHours` was specified as a ***procedure***. On the other hand, the `hours` method is specified as a ***function***. The idea of a function is borrowed from mathematics. Here's an example: $f(x, y) = x^y + y^x$. Clients of this little math function supply values for the parameters x and y, and the function returns the single value $x^y + y^x$, computed from the supplied values for x and y. Notice that f returns a single value without changing the values of parameters x and y.

The essential idea of a *functio*n is similar to that of a procedure in that:

- A function has a fixed number of formal parameters.
- Clients supply a function with incoming values (often in the form of variables), which correspond to the formal parameters.
- The function may change the values of the variables resulting in *"outgoing"* values for the client's variables (this is inconsistent with the behavior of mathematical functions, but it is unavoidable at times in Java functions).

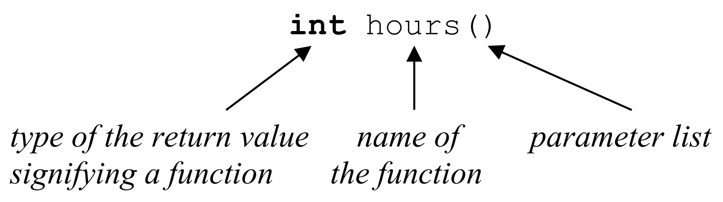
However, one thing that differentiates a function from a procedure is that a function *computes* and *explicitly returns* a single value to the client.

So, to describe a function to a client, we'll describe:

- the number and types of the formal parameters
- the allowable incoming values of the client-supplied variables
- the type of the resulting single value the function is computing for the client
- the resulting single value the function is computing for the client
- the resulting outgoing values of the client-supplied variables.

We can now understand the structure of the function header for `hours`:

Anatomy of the Function Header



Still Awake?

4-8. What is the type of the return value for the function `isAM`? Explain why this is the type.

The function contract is

Returns:

```
  this.hours
```

Ensures:

```
  hours = this.hours
```

Notice anything unusual? To start with, there is no requires clause! With a little explanation, this is understandable. Since the parameter list is empty, the only formal parameter for `hours` is the distinguished argument `this`, a variable with a value described by `AMPMClock`. Whatever clock value a client might supply for `this`, `hours` will be able to return the `hours` portion of the value (`this.hours`). So, there is no need to impose any restrictions on incoming values.

There is also no **Updates** section, indicating that this function will not modify the value of any of its arguments. The description of outgoing values is quite simple since `this` is the only parameter and `hours` is not modifying any of its parameters. So we know that `this = #this`.

Last, the name of the function appears in the ensures clause:

Ensures:

```
  hours = this.hours
```

This will always be the case for all functions. We simply let the name of the function, in the ensures clause, stand for the single value to be returned by the function. A good way to read the assertion `hours = this.hours` is “the value to be returned by `hours` is `this.hours`”.

We should also point out that there is some redundancy in this contract. As you can see, the **Returns** and **Ensures** entries say pretty much the same thing. This is because of how simple this function is. For more complex functions, the **Returns** entry will provide a short, informal description of the value returned by the function while the **Ensures** clause usually will provide a precise mathematical description of the entire behavior of the function.

Still Awake?

4-9. Explain how to read the ensures clause of `isAM`.

4-10. In the function `isAM`, what is the outgoing value of `this`?

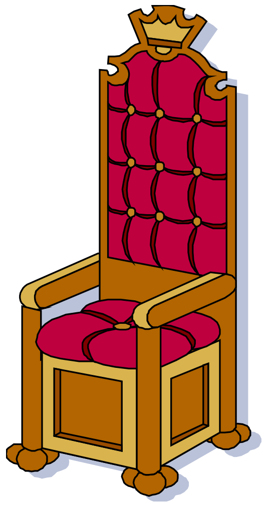
4-11. Provide a complete client description of the `minutes` method.

4-12. Provide a complete client description of the `seconds` method.

Client View Components

Now is an excellent time to step back and review what's happened so far. `AMPMClock` is a Java interface that provides us with a client-view description of the behavior of an am-pm clock. In this sense, it is an abstraction of the behavior of a clock component. Just like all interfaces it provides two things: a *type* (designating a set of values) and *methods* that can be applied to variables whose values come from the type (i.e., from the set of values).

The Emperor Has No Clothes

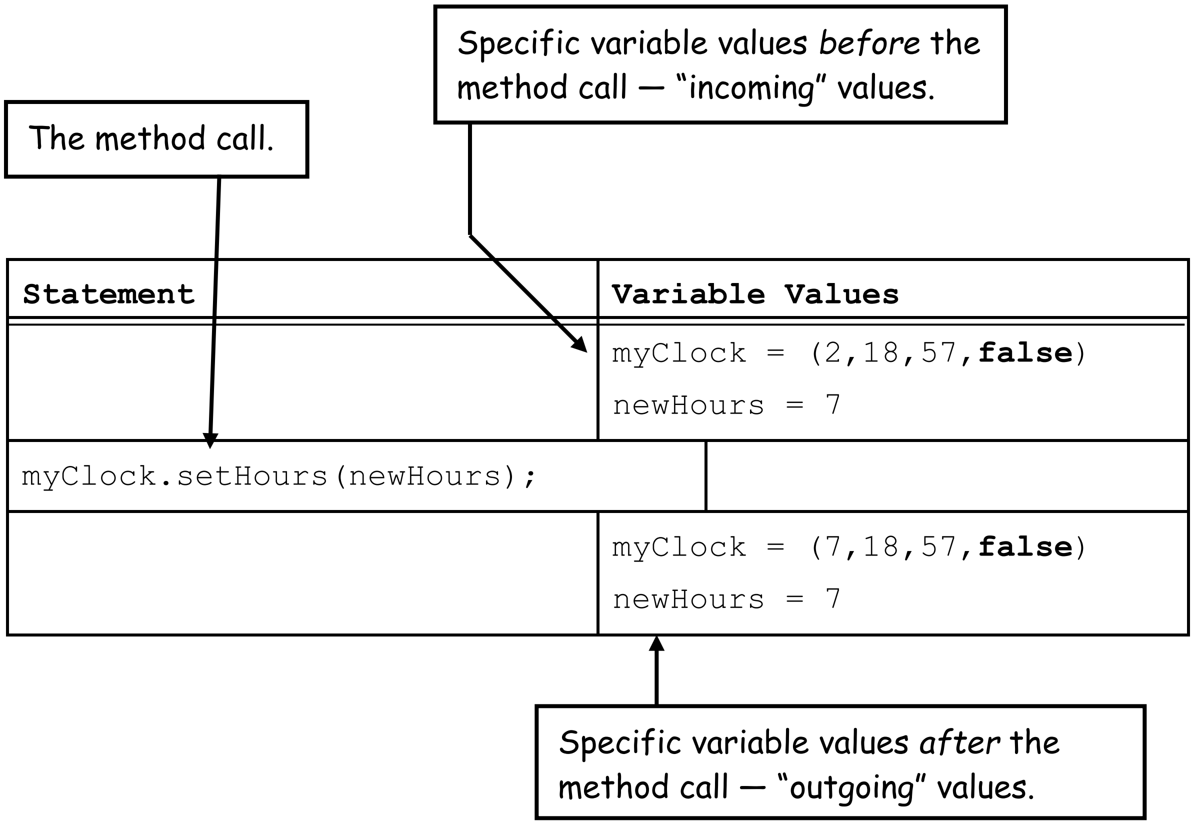


And now for a big surprise: It will not be sufficient for clients to order only `AMPMClock` from a library! `AMPMClock` provides descriptions that clients use to understand *what* am-pm clocks are and can do. `AMPMClock` does not provide a single piece of executable code specifying *how* the methods do their job. But in a program using `AMPMClock` variables, Java will need such executable code! As a result, clients will also order, from a library, a ***class component*** delivering the necessary executable code. For `AMPMClock`, this will be a component with a name that reads something like `AMPMClock1`; that is, `AMPMClock1`is a component providing executable code that implements the behavior described by `AMPMClock`. Think of it this way: `AMPMClock` describes, like an instruction manual, a part that a client would like to use, and `AMPMClock1` is the part number. In a Java program, then, clients would declare and initialize an am-pm-clock variable as

```
AMPMClock myClock = new AMPMClock1();
```

For Peace of Mind Try Tracing

The general specification of what methods do is provided abstractly in a method’s contract through the requires and ensures clauses. To aid our understanding of these specifications of methods, it is often helpful to see examples of the effects of methods when applied to specific variable values. To do this, we use tracing tables. Here’s an example:


Specific variable values *before* the method call — "incoming" values.
The method call.
Specific variable values *after* the method call — "outgoing" values.

Statement	**Variable Values**

	`myClock = (2,18,57,`**`false`**`)`
	`newHours = 7`

`myClock.setHours(newHours);`	

	`myClock = (7,18,57,`**`false`**`)`
	`newHours = 7`

In this example, `myClock` is a variable of type `AMPMClock` and `newHours` is a variable of type **`int`**. Declarations of these variables may look like this:

```
AMPMClock myClock;
int newHours;
```

The tracing table shows the effect of a call to the `setHours` operation on specific values for the variables `myClock` and `newHours`. The "incoming" values of these variables are `myClock = (2,18,57,false)` and `newHours = 7`; the "outgoing" values are `myClock = (7,18,57,false)` and `newHours = 7`.

Tracing tables can trace more than a single method call, as the next example shows. In this example, `isMorning` is a variable of type **`boolean`**.

Statement	**Variable Values**

	myClock = (2,18,57,**false**)
	newHours = 7
	isMorning = **true**

myClock.setHours(newHours);	

	myClock = (7,18,57,**false**)
	newHours = 7
	isMorning = **true**

isMorning = myClock.isAM();	

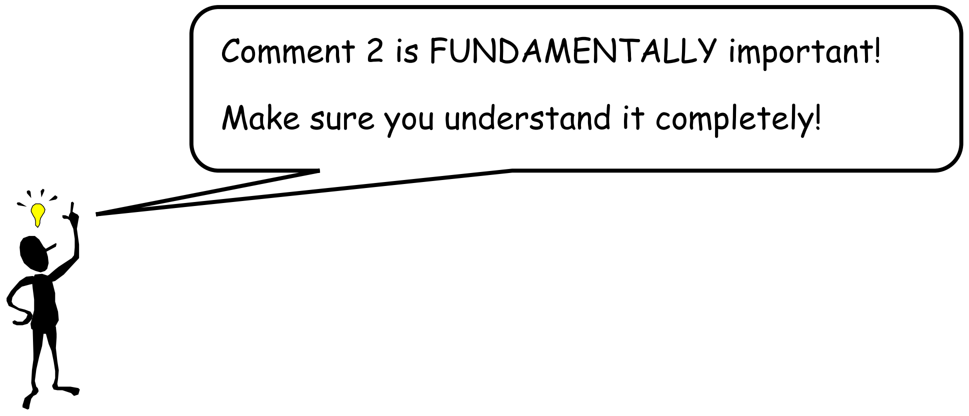
	myClock = (7,18,57,**false**)
	newHours = 7
	isMorning = **false**

Unit Wrap-up

Comment 1: In the second tracing table notice the difference between the two statements being traced. The statement `myClock.setHours(newHours)` is a call to the ***procedure*** `setHours`. The effects of the procedure call are made known to the caller, the client, through changes in the values of the client-supplied actual parameters, `myClock` and `newHours`. (Of course, only the value of `myClock` is changed.) On the other hand, the statement `isMorning = myClock.isAM()` is an assignment statement, one part of which is the call to the ***function*** `isAM` and the other part being the assignment of the value returned by `isAM` to the variable `isMorning`. We could have just called the function `isAM` using the statement `myClock.isAM()` and left out the assignment of the return value to `isMorning`. But this would not make much sense in this case! `isAM` does not change the value of any of its parameters. So why would a client bother to call `isAM` if the client does nothing with the return value? This would be like sending someone to the ticket office to buy movie tickets, and telling them to just leave the tickets at the ticket office! REMEMBER, usually you'll want to make use of the return value of a function call!

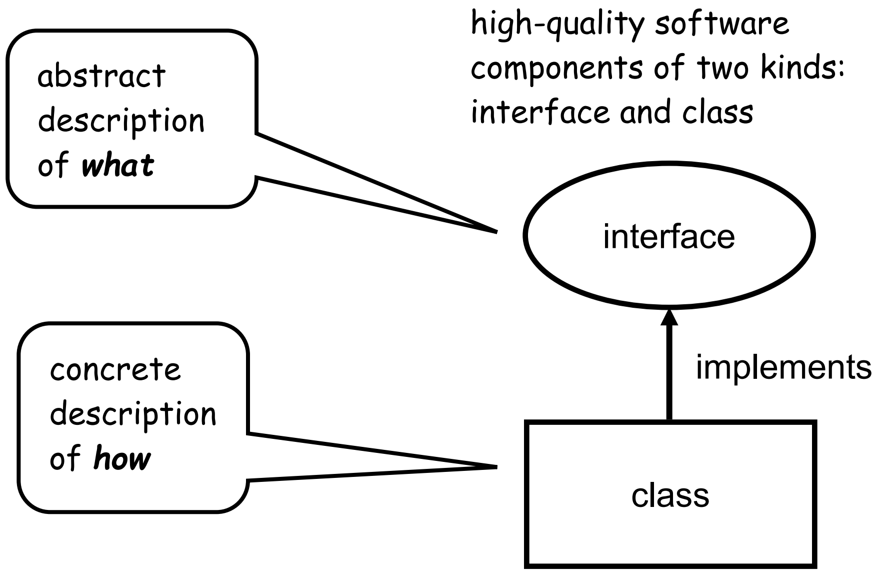
Comment 2: Comment 2 appears in the next figure. IT IS FUNDAMENTALLY IMPORTANT. Make sure you understand it completely. Put it into an accessible, long-term part of your memory!

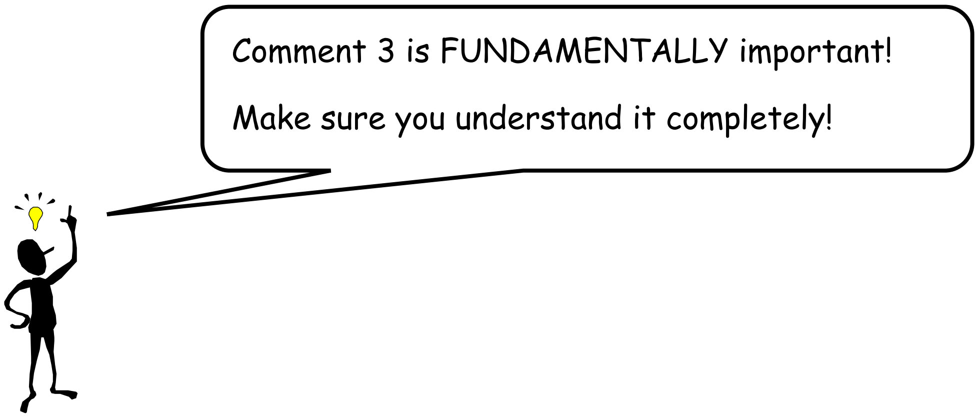
Client descriptions of methods must describe all assumptions about the incoming values of all parameters, and must describe the corresponding outgoing values of all parameters. Such descriptions can be presented in different ways. Good descriptions will be precise (no margin for error) and understandable (by human readers). The ***mechanism*** that is used in CSE 2221/2231 to make such precise and understandable descriptions of methods is ***contracts***, where each method contract consists of two sections, a requires clause and an ensures clause.


Comment 2 is FUNDAMENTALLY important!
Make sure you understand it completely!

Comment 3: Comment 3 appears in the next figure. IT IS FUNDAMENTALLY IMPORTANT. Make sure you understand it completely. Put it into an accessible, long-term part of your memory!

High quality software components should be organized into two distinct parts. One part describes ***what*** the component can do. We refer to this as the *abstract* part or ***interface***. The second part provides code (the ***how***) that implements the behavior described in the abstract part. We refer to this as the *concrete* part or ***class***.


high-quality software components of two kinds: interface and class
abstract description of ***what***
interface
implements
concrete description of ***how***
class


Comment 3 is FUNDAMENTALLY important!
Make sure you understand it completely!

Still Awake?

4-13. Complete the following tracing table.

Statement	**Variable Values**

	`myClock = (8,2,43,`**`true`**`)`
	`yourClock = (11,18,6,`**`false`**`)`
	`transferMinutes = 7`

`transferMinutes = myClock.minutes();`	

	`myClock =`
	`yourClock =`
	`transferMinutes =`

`yourClock.setMinutes(transferMinutes);`	

	`myClock =`
	`yourClock =`
	`transferMinutes =`