Chapter 3: Basic "Get-Compute-Report" Programming

In this chapter you will set out on a long journey, a tour of all the different kinds of boxes that are used as tools in programming. This tour will include details on how all these kinds of boxes behave when they execute, and how to use them to accomplish certain programming tasks.

The tour begins with the kinds of boxes needed to create the most basic kind of program, namely one that gets some input from the user at the keyboard, computes with it, and reports some results in the console box.

The tour begins with an overview of the overall arrangment of the jBoxes system.

Setting the Stage

Before detailing the executable kinds of boxes for this chapter, it will be helpful to examine the structure of a jBoxes universe and see exactly where the material of this chapter fits into it.

Every kind of box can be classified as either a **container** box or an **executable** box. Every executable box is either an **action** box or a **value** box. Container boxes simply serve to hold other boxes. Action boxes perform some action when they are told to execute. Value boxes produce a value when they are told to execute.

The **universe** box is the outermost box. It contains everything in a given box diagram, and that is its whole purpose.

When you create a new universe, it comes straight from the factory with five inner boxes pre-built. These boxes are the **stack** box, the **heap** box, the **static data** box, the **console** box, and the **ports** box. The console box is used to display information, while the stack box, heap box, static data box, and ports box are container boxes. Of these, only the stack box and console box will be used in this chapter and the next. It is a good idea to change unused boxes to show their abstract aspects so they don't take up so much space.

All your programming work for the next few chapters will take place in a single **class** box. A class box is created by hitting the space key when the focus is on the mark for the universe box, or on a class box that has already been created.

A class box has four inner boxes, namely a **class data** box, a **class methods** box, an **instance data** box, and an **instance methods** box. These are all container boxes. New inner boxes in each of these are made by putting the focus on their marks or inner boxes and pressing the space key.

A class box is like a factory for building objects. For the next few chapters, you will only be using the **class methods** box. This box holds **method** boxes. A method box is a box that contains a sub-program that can execute and perform some task. For this chapter and the next, you will only be using one class method box.

A method box has four inner boxes, namely an **inputs** box, an **output** box, a **locals** box, and an **instructions** box. In this chapter and the next, the inputs box and the output box

will not be used, because they are used to facilitate communication between two methods, and only one method will be used. The locals box is a container box that holds **data** boxes that are used to hold information while the program is executing. The instructions box is actually just a **sequence** box. It holds a sequence of action boxes that make up the directions for the program.

The Basic Kinds of Boxes for Input, Computation, and Output

In this section you will learn about the very most basic kinds of boxes that are needed in almost any program. These boxes provide the basic building blocks for getting input and producing output, and for doing computation, including storage and retrieval of data.

These boxes are so primitive that, with the exception of a few, you will never directly create them. Instead, you will write instructions in a **java box** that will translate to diagrams containing these kinds of boxes, as detailed later. Still, it is important to be somewhat familiar with these very basic boxes—you just don't need to know how to create them.

The Data Box

The locals box in a method, as well as several other container boxes to be studied later, typically holds some **data** boxes. A data box is used to hold information. During the execution of a program, other kinds of boxes can retrieve that information from the data box and store changed information in the data box. A data box has a name—displayed in the upper left corner of its full aspect—by which these other boxes can refer to it. It has a type—displayed in the upper right corner of its full aspect—which specifies exactly what kind of information can be stored in it.

A data box is created by pressing the space key when the focus is inside the locals box. The focus is then put on the mark for the name part of the data box, ready for you to enter the desired name.

The name should be picked to help you, or anyone else looking at your work, to remember the purpose of the box. You can only use lowercase and uppercase letters, digits, and the "underscore" symbol in the name of a data box. And, you shouldn't use the underscore symbol. jBoxes is *case sensitive*, which means that, for example, an uppercase "A" is a completely different symbol from a lowercase "a".

By Java convention, all names should start with a lowercase letter, except names of class boxes, which should start with an uppercase letter.

Once the name has been entered, hitting the space key moves the focus to the mark for the type part.

The type can be one of the **primitive data types** listed in the chart below. It can also be the name of a class box or a type name followed by one or two left bracket symbols, but those possibilities will be left until later.

Primitive Data Types

Type name	Full name	Sample Values
int	integer	-37, 28, 0, 123456789
float	floating point number	3.14, -12.73e-4
char	character (symbol)	a, ?, B, %
boolean	boolean	false, true
string	string	hello, 3.14, B, -37

In order to decide which type of value a data box needs to hold, you must be aware of its purpose in the program. For example, if a data box is intended to hold the number of people belonging to some organization, the `int` type would be used, while a data box used to hold the name of the organization would be of type `string`.

During the execution process, if an attempt is made to store a value of the wrong type in a data box, execution halts with an error message.

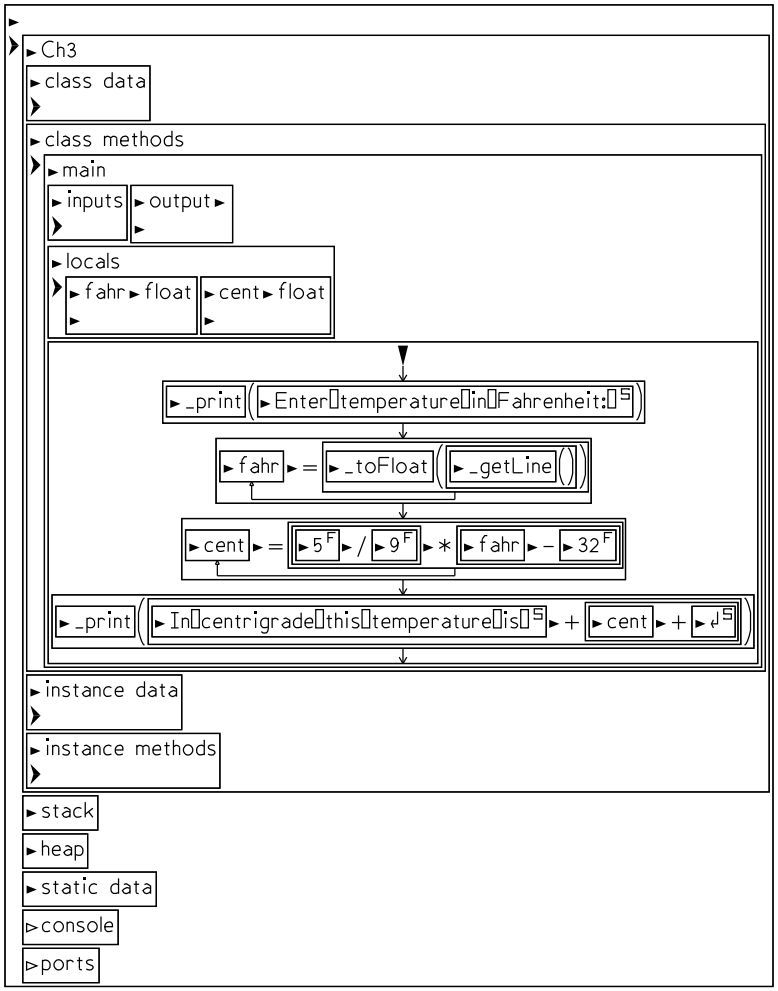
The only exception to this rule is that a value of type `int` may be stored in a data box of type `float`.

Once the type has been entered, pressing the space key yet again moves the focus to the mark for the value part.

The value part of a data box could be interactively filled, but it should typically be left empty, to be filled during execution. Any symbols may be inserted into the value part of the data box, except that a space symbol will cause the creation of a new data box, rather than putting that symbol in the value string, unless the data box has type `string` or `char`, in which case the space symbol will be inserted.

Exercise:

Download the universe named `ch3.box` and start jBoxes on it. It will look like this:



Note how this universe has a single class box, with a single class method box in that class box. Note further that the method box has two data boxes in its locals box.

This method box contains a program to convert a temperature expressed in Fahrenheit degrees to the equivalent temperature in centigrade. Since the input temperature can be any number, including one with a decimal part, the data type used for the first data box is `float`. The name `fahr` is used for this data box to remind the reader that it holds a temperature in Fahrenheit. Similar ideas apply to the other data box.

This universe is written entirely without using javaboxes so you can see the various basic kinds of boxes that are discussed in this section.

The Identifier Box

Of course, the data box is of no use without boxes that can retrieve and store information from and to it during execution. The **identifier** box is used for that purpose.

The identifier box only has one part, its name. The name is the same as the name of some data box. Wherever the identifier box appears, it executes by making a connection to the data box of the same name. Thus, an identifier box executes by "identifying" the location of the box it names.

The identifier box belongs to the category of *value boxes*, because when it executes, it either produces the value stored in the data box with its name, or produces the location of that data box, depending on where it is used.

Exercise:

In the `ch3` universe, note the various uses of identifier boxes with the names `fahr` and `cent`. But, also note other identifier boxes—what names are used in them?

Step through execution of this program using F1. Notice how each identifier box behaves when it executes. In particular, look closely at the upper right corner of the value aspect of each identifier box when it arrives. Sometimes this string is `system` or `local`, indicating that the box has produced a value which is a location, and sometimes it is `float`, indicating that the box has produced a data value.

The Empty Box

One of the most common and important kinds of boxes is the **empty** box. In Chapter 2 you saw how this box is sometimes created as a sort of placeholder, for example, in a branch box.

If you forget to (or choose not to, in the process of developing a program) convert an empty box to a specific kind of box, it executes by either doing nothing, if it is used in a context where an action box is expected, or by causing an error if it is used in a context where a value box is expected.

The Assignment Box

The assignment box is the workhorse of jBoxes. It is an action box, used whenever you need to perform the action of doing some computing and storing the result somewhere.

The assignment box has a string part positioned between two inner boxes. It has a decorative arrow drawn from the second inner box to the first.

The string part in the middle can be any of the strings `=`, `+=`, `-=`, `*=`, `/=`, or `%=`. The `=` that is provided when you press a is by far the most common choice.

The assignment box executes as follows. The first inner box executes, obtaining a value which is a location. Then the second inner box executes, obtaining a value. Then, if the string part is just `=`, the value of the second box is stored in the data box specified by the location given by the first box. If the string part is `+=`, then the current value at the location has the value of the second box added to it, and similarly for the other arithmetic operators.

Exercise:

In the `ch3` universe, execute the `main` method using F1 and notice carefully how the assignment boxes execute.

The Call Box

The call box is used, in this chapter, when you want to ask the system to perform some basic task, such as clearing the console box, displaying some information in the console box, getting input from the keyboard, or converting data from one type to another. Later you will learn how to use a call box to request the execution of a method box that you have written, but in this chapter you will just use **system methods**.

A call box has a first inner box that is an identifier box containing the name of one of the system methods. After that box it just has decorative parentheses. In between these decorations are zero or more other inner boxes, depending on the particular system method that you are calling. For example, the `_getLine` method, which is used to request that the user type some input, doesn't have any boxes between the parentheses, while the `_print` method, which is used to send information to the console box, has one.

These inner boxes between the parentheses are known as *arguments* or *parameters* for the method call.

A call box executes by first executing its first inner box. For now, that box will be an identifier box specifying which system method you want to call. Then, each of the argument boxes, if any, are executed. Finally, the appropriate system method performs its work, using the argument box values, sometimes, to specify exactly how it should work.

Exercise:

Step through execution of the temperature conversion program yet again, this time noticing carefully how the various system method calls behave. Be sure to notice how the method name is first identified, then how the arguments, if any, are evaluated, and finally how the call box receives a value which is then used somehow. Pay special attention to exactly how the `_toFloat` method behaves, in terms of the type of its argument and the type of its value after it executes.

The Summary Sheets list all the available system methods in jBoxes.

The Literal Boxes

Quite often a specific value needs to be used in a program, so jBoxes includes five related kinds of boxes that are collectively known as "literal boxes." For each of the primitive data types discussed earlier, there is a corresponding kind of box.

Each kind of literal box has a single string part. They also each have a decorative uppercase letter, namely `B`, `C`, `F`, `I`, or `S`, indicating which particular type of literal value the box can hold.

Literal boxes execute in the simplest possible way—the value of a literal box after execution is simply the same as its string part.

Exercise:

Find all the literal boxes occurring in the temperature conversion program. Step through execution of the method and note how these boxes execute.

The Operation Boxes

The main goal of any computer, is, after all, to compute! The literal boxes let you put specific values of the various primitive types directly in your program as needed, the identifier box lets you refer to a value already stored in a data box, and the assignment box lets you copy a value from one data box to another, but, other than the optional "update" feature of an assignment box (using `+=` and the others), you have not yet seen any way to ask for any *computation* to be performed. The binary and unary operation boxes perform computations when they execute.

A binary operation box has a string part positioned between two inner boxes. The string specifies the particular binary operation that is to be done. The inner boxes provide the values with which the operation is performed.

The following table details all the binary operations provided in jBoxes.

Binary Operations

Operation string	Name	Description
+	addition	adds two numbers or concatenates two strings
-	subtraction	subtracts second number from first number
*	multiplication	multiplies two numbers
/	division	divides two `float` values, does whole number division of two `int` values, second value cannot be 0
%	remainder	values must be `int`, produces remainder when divide first by second
<	less than	compares two numbers, two `char` values, or two `string` values, produces `boolean`
>	greater than	(similar to < but using >)
<=	less than or equal to	(similar to < but using $\leq$)
>=	greater than or equal to	(similar to < but using $\geq$)
==	equal to	(similar to < but using =)
!=	not equal	(similar to < but using $\neq$)
&	logical and	produces `true` if both are `true`, `false` otherwise
|	logical or	produces `false` if both are `false`, `true` otherwise

An operation box executes by first executing its two inner boxes, so they have values. Then, the operation specified by its string part is performed on those two values, and the operation box itself gets the resulting value as its value.

Sometimes the types of the inner boxes determine the meaning of the operation string. The main example of this is that `+` with inner boxes that evaluate to produce numbers—values of type `int` or `float`—add the two numbers in the usual way, but if either one of the inner boxes evaluates to produce a value of type `string`, then the other value is converted to a `string` value and the two strings are *concatenated* together.

Similarly, if the two inner values are both of type `int`, then the arithmetic operation that is performed is whole number addition, subtraction, and so on. This is important in two ways, both related to division. First, whole number division is different from `float` division. For example, `9/5` is `1`, and `5/9` is `0`, but `9.0/5.0` is `1.8` and `5.0/9.0` is `0.55555555`.

The unary operation box is similar to the binary operation box, except it only has one inner box, and the desired unary operation string is placed before that inner box.

The following table details all the unary operations in jBoxes.

Unary Operations

Operation string	Name	Description
-	opposite	changes the sign of its argument, which must be a number
!	not	flips the truth value of its argument, which must be `boolean`

Exercise:

Run the temperature conversion program and note how the various binary operation boxes in this method behave. This method doesn't use any unary operations.

The Javabox

The previous lengthy discussion of all the basic kinds of boxes that are needed to create programs of the "get, compute, report" type never mentioned how to make or otherwise edit these boxes—it was all about how they behave when they execute. The reason for this is simple—you will never directly make any of these kinds of boxes! Instead, you will use javaboxes.

A **javabox** is a very special kind of box. The basic idea of this new kind of box is simple—instead of laboriously creating box diagrams interactively, you can simply create a special box, known as a javabox, which contains a string that is then translated into the desired box diagram. This string is stored as the "name" part of the box, so when the javabox is showing its abstract aspect, you see this string, with a "J" in a box as the mark. This string represent one or more Java-like instructions. It can contain embedded enter symbols

that cause it to be displayed on multiple lines, thus appearing like a little page of Java-like instructions.

The string you put in the javabox must follow the grammar, or syntax, rules of the Java language. The drawback to using the javabox is that if what you type for the name string doesn't follow the rules, then you will get an error message when you try to execute the method containing the javabox.

To create a java box, press j. This can be done in any context where a value box or an action box is expected.

When a new javabox is created, its abstract aspect is shown, and the focus is put inside, waiting for you to simply type symbols giving the desired instructions in the Java-like language.

The javabox executes by translating the name string into the corresponding box diagram, if the string follows the rules and thus can be successfully translated. If the string breaks a rule, the translation fails, and an error box comes up, trying to tell you what rule you broke. Unfortunately, these *syntax error messages* are often incomprehensible.

But, after you hit enter to dismiss the error message, the focus will be put on the java box that failed to be translateable. You should then move in, either by pressing ↓ or ctrl i, which will move the focus in to the single symbol where the translation failed. Usually this will let you figure out what mistake you have made.

The system automatically translates java boxes to the corresponding box diagrams when a method box is executed. But, you can manually do translation when the focus is on or in a java box by pressing ctrl t. If the translation fails, you can fix it before going on. It is often easier to check your java boxes for syntax errors individually and right after you have made them, than to wait for errors that show up when they are automatically translated during execution.

The translated box diagram produced from the abstract part of a javabox is stored as a single inner box in the full aspect of the javabox. Be careful not to directly edit the translated diagrams for a java box, because any changes you make will be ignored, since the automatic translation always takes place and will thus override any changes that you think you have made.

Be warned that this process of writing instructions in a javabox, trying to run the program, finding out that something is wrong, looking at where it went wrong, and trying to fix it, can be very frustrating. That is the tradeoff—the process of building a desired program diagram manaully is cumbersome, but writing the instructions in a javabox to be translated leads inevitably to this syntax error problem.

In order to express textually, in the Java-like language, concepts that are completely clear in a box diagram, a number of somewhat arcane and confusing rules and extra symbols are required. The rules for using these symbols have to be understood, which will take some effort.

Basic Rules for Java Boxes For This Chapter

Now you need to learn what symbols to type in a javabox in order to produce, upon translation, the box diagram that you desire. Along with the basic ideas of what symbols translate to what boxes, you will need to learn some special rules for the Java-like code that are necessitated by its textual nature.

How to Create an Equivalent Java Box for Each Kind of Box in This Chapter

Every kind of action or value box (not container boxes, note—java boxes only appear inside the action box for a method box) covered earlier in this chapter can be created by writing the appropriate symbols in a java box. The list below shows how to generate all these possibilities.

Sequence box

In the earlier part of this chapter, the sequence box was mentioned in passing, since the instructions for a method box form a sequence box. The sequence box will be discussed more carefully in the next chapter, but it also turns out that a javabox itself allows for a sequence of actions inside it, but the catch is that the translator has to be able to separate the individual actions, which are known as *statements*. For the material of this chapter, you have to put a ; symbol at the end of each statement so that the translator can handle it.

One general rule is that the translator ignores "whitespace," which is spaces, end of line symbols, and tabs. You should use whitespace to make your javabox instructions more readable. For example, it is a good idea to hit enter after each statement, so that each statement in a javabox occurs on its own line.

Identifier box

To create an identifier box in Java code, you simply use the name of the desired data box.

Assignment box

A so-called assignment statement in Java translates to an assignment box. You simply type the name of the data box that is to receive the value, followed by a = symbol, followed by some symbols that represent some value to be stored, followed by a ; punctuation mark. The material between the = and the ; is known as an *expression*. You can also put `+=`, `-=`, and so on, in between the name of the target storage location and the expression that produces the value to be stored.

Literal boxes (Boolean, Char, Float, Int, and String)

In a java box, the translator has to be able to recognize symbols that represent a value of some type. This is done through use of some rules and some special symbols.

A `boolean` literal value is represented by the strings `true` or `false`. One consequence of this is that in Java, data boxes can't be named `true` or `false`, so you should avoid those names.

A `char` literal value is represented by the desired symbol between single quote marks. One special case is the enter symbol. Since enter behaves specially in a java box, it is impossible to type it in between quotes. So, a so-called *escape sequence* is used. Escape sequences consist of the backslash character followed by a symbol. The enter symbol is represented by `'\n'`. Another special symbol is the single quote symbol—it is represented by the escape sequence `'\''`.

`int` literal values and `float` literal values are represented exactly as they are in normal mathematical usage. Because of this, a data box can't have a name that starts with a digit.

Finally, `string` literal values are represented by the desired sequence of symbols between double quote symbols. Escape sequences are used, as with `char` literal values. An additional escape sequence is the `"\""` symbol.

Operation and Unary operation boxes

Expressions in a javabox behave just like they do in usual mathematical notation. These include binary and unary operations, which translate naturally to operation and unary operation boxes. Also, expressions can use literal values, names of data boxes, and parentheses, following all the normal mathematical rules of order and precedence.

Call box

A call box is produced by a notation that is the same as usual mathematical notation for a function evaluation. Call boxes can be actions, appearing by themselves with a ; at the end, or can be combined in expressions.

This concludes the overview discussion of how to do in a javabox the equivalent of each kind of box in this chapter. Now you need to gain some hands-on experience with these ideas.

Exercise:

Start up jBoxes on a new universe, say named `ch3play`, and make a new class box, with a class method box named `main`, In the instructions box for that method, experiment with some javabox issues as detailed below. For each activity, note that you are only supposed to observe how the given Java-like instructions translate, not how they execute. So, after making each javabox, hit ctrl t to translate the box and then change aspects to see the full aspect, which is the translated version of the Java instructions you typed.

Make a javabox and type `x=y` in it, intentionally leaving off the semi-colon at the end of this statement. When you translate it, note the error message.

Add the semi-colon at the end and translate again. This time, note that an assignment box was created, with the variables `x` and `y` translating in the obvious way to identifier boxes.

Insert `3+` so the statement now reads

```
x=3+y;
```

and translate. Note how the `3` translates to an `int` literal, and observe how the expression between the `=` and the `;` translates to a binary operation box.

To explore how Java code works, enter each of the statements below into a javabox, translate, and thoughtfully observe the result. `x=a+b+c;`

```
x=(a+b)+c;
x=a+(b+c);
x=a+b*c;
x=a*b+c;
x=(a+b)*c;
x=f(x);
g(x);
h(x,y,z);
```

Try putting several statements into one javabox and note how the translation includes a sequence box:

```
x=a+b;
y=c+d;
```

Check out how literals translate by putting these statements into javaboxes:

```
x="hello";
a='z';
```

Exercise 3 Officially due by the start of the next class period (but won't be penalized if turned in up to one week later). Turn in your answer by emailing the instructor at `shultzj@mscd.edu`.

Start jBoxes and enter `ex3` as the name of the universe to work on.

Create your own program, which will use many of the ideas of this chapter, that will ask the user to enter three numbers and will then compute and report their product (all three multiplied together).

Be sure to allow the inputs to be of type `float`. Test your program before turning it in!

When you are satisfied with your program, email me with your file `ex3.box` as an *attachment*.