

## jBoxes Reference Guide

This reference guide describes the jBoxes system in detail. Although it can be read from beginning to end, it is intended mostly as a summary and at some times enhancement of the information given in The jBoxes Booklet.

### Purpose of jBoxes

The purpose of the jBoxes system is to provide a comfortable environment in which to learn about programming. This is accomplished by making everything that happens in the system *visible*. This visibility lets you learn how the system works by observing it in action. Then, once you understand how the system works, when you create your own programs, you can observe how they behave, helping you to correct your errors.

### Fundamental Constructs in jBoxes

In addition to being a visible system, jBoxes is intended to be useful for teaching beginning computer science because it uses relatively few *constructs*.

This section describes, in fairly high level terms, the basic concepts and terms in the jBoxes system. This section is very terse, intending to provide a high level perspective on the system and trying to provide insight into the overall scheme, since you can observe the details by using the system.

### The Diagram Metaphor

The jBoxes system is built on the metaphor of a *diagram*. To create programs, you add your own pieces to the original diagram. When a program is executing, the system reads parts of the diagram as instructions and modifies the diagram as it works to perform the instructions.

The term *universe* is used interchangeably with the term “diagram.”

### Everything is a Box or a String or a Decoration

The diagram is built entirely from boxes, strings, and decorations.

A box is a rectangular region that contains, depending on its *kind*, various *parts*, together with some decoration. A part is either a sequence of boxes or a string.

A fundamental construct for boxes is *nesting*, in which a box contains a list of inner boxes, each of which may have its own inner list, and so on.

Each box has a specific kind. Its kind determines what string parts it has, what inner boxes it has, how these inner boxes arrange themselves, what decorations it has, how it can be edited, and how it behaves during execution.

The inner boxes are always in sequential order, from first to last, with each box having a previous box and a next box in that order (except for the first and last boxes). The inner boxes are always arranged from left to right and top to bottom in this order.

A *string* is a sequence of 0 or more symbols. A *symbol* is any of the usual symbols with Unicodes from 32 to 126, together with some additional symbols that are used by the

system, including several *marks*. These marks are used solely to mark the beginning point of a string, so that it shows up even if there are no other symbols in the string.

The symbols in a string are organized in a sequential order, similar to the ordering of inner boxes in a box. The symbols in a string display either left to right on a line, or in a standard left to right, top to bottom style as in usual text.

Strings only occur as parts of boxes. The possible parts are the *name* of a box, the *documentation* for a box, the *type* of a box, and the *value* of a box. The kind of a box determines exactly which string parts it has.

*Decorations* are additional features in the visual display of a box that help to indicate which kind of box it is and how it behaves. Decorations serve no other purpose and could be omitted entirely without affecting the behavior of the system in any way. Decorations include arrows indicating flow of data or control and little drawings specific to a kind of box. Technically, the rectangular border of a box is part of its decoration.

## Viewing

When jBoxes is running, one or more *viewers* are operating. A viewer is a camera, aimed at some rectangular region of the diagram, showing that portion of the diagram in some rectangular area of the screen.

Which combination of viewers is operating may be selected interactively by the user, using **F8** and **F7** as detailed in the *Summary Sheets*.

A viewer has a border with a color that indicates the state of the border, as follows.

Red: fully interactive

Yellow: interactive while paused during execution

Green: execution is proceeding with no interaction possible

Gray: execution is waiting for input from the keyboard

Blue: display only, no interaction possible

When doing regular viewing, just one viewer uses the whole window. When doing split viewing, the left viewer provides regular viewing while the upper right viewer remains fixed over the console box, showing its contents whether or not they are visible in the main viewer, and the lower right viewer similarly remains fixed over the ports box.

The region in the universe that a viewer is displaying may be changed in various ways, as described in the Summary Sheets. The “camera” may be moved left, right, up, or down, and the size of the region may be increased or decreased.

The rectangle in the window that a viewer is using to display its image is fixed, depending only on the size of the window.

## Focus

When an interactive viewer is displaying, it has a single box or symbol that is referred to as the *focus*. The focus is the box or symbol that responds to interactive commands. It is like the “cursor” in a typical word processing program.

The focus may be changed from one box or symbol to another in various ways, as described in the Summary Sheets.

## Aspects

A box may have, depending on its kind, a number of inner boxes and string parts, so the system allows the user to control how much of the detail, and which details, are shown for a given box at a given time. The particular combination of details being shown is known as the *aspect* the box is using or showing. Depending on the kind of box, some aspects may not be possible, simply because they make no sense.

Aspects are useful for avoiding display of details that the user does not want to see, and for suppressing details during execution.

The user may interactively change the aspect of the focus box in various ways, as detailed in the Summary Sheets. When the aspect of a box is changed, it changes to the next aspect that box has in a standard order. So, to get to a desired aspect, the user may need to change aspects several times.

Here are the possible aspects for a box:

**Full:** The inner boxes are shown, together with all string parts except for the documentation string, if any. String marks are filled triangles. Displays using black. Used when detailed view of the box is desired.

**Abstract:** Just the name part of the box is shown. The mark for the name string is a hollow triangle. Displays using blue. Used when just the existence of the box, with no further information, is desired. No execution detail is shown.

**Documentation:** Just the documentation string is shown. The mark for this string is a little page image. Displays using green. Used when a description of the box is desired. No execution detail is shown.

**Value:** The value string and the type string of the box are shown. Displays using red. Generated automatically for some boxes during execution, not typically changed to by the user.

## Navigation

*Navigation* refers to moving the focus from one box or symbol to another. This may be done either through use of the keyboard or through the mouse.

Mouse navigation is quite simple. If you left-click on a box or symbol, it becomes the focus. Note that the concept of “on” a box means that the mouse cursor is inside the border of the box but not inside any of its inner boxes or string parts. If you right-click anywhere inside a box that is not on one of its inner boxes, then the aspect of the box is changed.

Keystroke navigation is also simple, but unlike navigation via the mouse, the keys move the focus according to the nesting structure of the universe. Keys can be used, as detailed in the Summary Sheets, to move the focus to an inner box, to an outer box, to the next box in a list of inner boxes, and to the previous box in a list of inner boxes. They can also be used to move part to part within a box, where the parts can include the inner list of boxes and various string parts.

## Destruction

The focus box or symbol may be destroyed in various ways, as detailed in the Summary Sheets. Note that it may not be possible to destroy a given box or symbol, depending on its kind or context. For example, a mark can never be destroyed, unless the entire box it belongs to is destroyed.

When a box or symbol is destroyed, the focus moves automatically to the next box or symbol after it, or somewhere else if there is no box or symbol after it. For example, when the last actual symbol in a string is destroyed, the focus moves to the symbol preceding it, which may be the mark for the string.

## Creation

A new box of a desired kind or symbol may be created by hitting the appropriate key, if the context of the focus box is correct.

The details of the context in which a certain kind of box can be created are covered in the next section. For now, it is enough to observe that the `[space]` key will create a box in a certain context if any kind of box is allowed. If only one kind of box is allowed, it will be created by pressing the `[space]` key, and if several kinds are possible, an empty box will be created.

Once an empty box has been created, either interactively by the user or as part of a larger kind of box, it can later be converted, when it is the focus, by pressing an appropriate key.

Symbols can usually be created and inserted into a string, except that names of boxes can only contain letters, digits, and the underscore symbol.

## The Categories of Boxes

The whole purpose of the jBoxes system is to provide an environment in which programs can be created and executed. Every box and string in the universe is involved with this creation and execution of programs in some way.

Many kinds of boxes fall into the category of *container* boxes. Their purpose is to organize other kinds of boxes, providing the overall structure within which programs are created and executed.

All other kinds of boxes are *executable*, in the sense that during the execution process, they can receive a request to execute themselves and then do so, either performing some action, in which case they are categorized as *action* boxes, or *evaluating* themselves to produce a string, in which case they are categorized as *value* boxes.

## Controlling and Observing Execution

Execution begins when the user puts the focus on an appropriate kind of box and hits one of the keys, as detailed in the Summary Sheets, that can send an execution request to the focus box. After that, execution proceeds on its own, with various boxes being sent an execution request, which causes them in turn to send execution requests to other boxes as they perform their execution behavior. When asked to execute itself, a typical box will send out some execution requests to other boxes, wait for them to finish, finish its own execution behavior, and send a final execution request to some box that needs to execute next.

The amount of visual detail that is provided depends on how the user runs the execution process. By using different viewers, by changing aspects, by using breakboxes, and so on, as detailed in the Summary Sheets, the user can control how often the execution process pauses and waits for another keystroke from the user before continuing the execution process.

## The Kinds of Boxes

Apart from the fundamental constructs just described, and the details of interaction provided in the Summary Sheets, most of the rest of the information about the jBoxes system consists of descriptions of the properties of each kind of box.

The following lengthy list shows, for each kind of box, its category, its purpose, its appearance (including decoration, arrangement of inner boxes, and pictures), its string parts, the kinds of boxes in its inner list, its execution behavior, and any special behaviors.

This tour of all the kinds of boxes is organized in a high level to low level way, with all the boxes at a certain level described before describing their inners in detail. In an outline sense, this is like handling I, II, III, and so on, before going back and covering I.A, I.B, and so on. This is known as a *breadth first* order. One inevitable consequence of this approach is that higher level boxes will be described in terms of their inner boxes that haven't been described yet. For example, when the universe box is described, it will be described as holding certain kinds of boxes, which will only later be described themselves.

**Universe box****category:** container**Sample image:****Appearance:** Is outermost box, is surrounded by dark grey nothingness, has light gray background color.

---

**Creation:** Made by the system when start a new universe.**Purpose:** Contains everything in the universe.

---

**Parts:** Holds a horizontal list of 0 or more class boxes, followed by a fixed vertical list consisting of the stack box, the heap box, the static data box, the console box, and the ports box, followed by a horizontal list of 0 or more demo boxes.**Execution behavior:** Doesn't execute

---

---

**Class box****category:** container**Sample image:****Appearance:** Has four distinctively named inner boxes arranged vertically, has bluish background color.**Creation:** Hit `[space]` when focus is on universe box's mark or on a class box.**Purpose:** Provides a blueprint for building *instances* of the class, which are also loosely known as objects. The *type* of the object is the name of the class. In addition to specifying how to build objects of its type, a class box also provides related support data and programs for these objects.**Parts:** Has a name and fixed inner list consisting of a class data box, a class methods box, an instance data box, and an instance methods box.**Execution behavior:** When execution begins, a copy of this box's class data box is made, with its name the same as this box's, and is put in the static data box.**Stack box****category:** container**Sample image:****Appearance:** Has name "stack" in upper left corner, 0 or more inner boxes arranged horizontally, has reddish background color.**Creation:** Fixed inner box for universe box.**Purpose:** Holds copies of method boxes that are in process of executing.**Parts:** Has name and inner list.**Execution behavior:** Doesn't execute.**Heap box****category:** container**Sample image:****Appearance:** Has name "heap" in upper left corner, has 0 or more inner boxes arranged horizontally, has greenish background color.**Creation:** Made by system.**Purpose:** Holds copies of instance data boxes for objects that have been created during execution.**Parts:** Has name and inner list.**Execution behavior:** Doesn't execute.

**Static Data box****category:** container**Sample image:**

**Appearance:** Has name “static data” in upper left corner, has 0 or more inner boxes arranged horizontally, has yellowish background color.

**Creation:** Made by system.

**Purpose:** Holds copies of class data boxes for class boxes in the universe.

**Parts:** Has name and inner list.

**Execution behavior:** Doesn't execute.

**Console box****category:** container**Sample image:**

**Appearance:** Displays just symbols and has a green cursor.

**Creation:** Made by system.

**Purpose:** Displays output information produced by the program during execution.

**Parts:** Has just a single string part holding the displayed output information. This string displays using the multiple line style, with 25 rows and 40 columns allowed.

**Execution behavior:** When various system methods execute, they change the displayed information or move the cursor or toggle display of the cursor on/off.

Note: with split viewing, full aspect of this box is shown in the upper right viewer, even if it is using its abstract aspect.



**Ports box****category:** container**Sample image:****Appearance:** Has name “ports” in upper left corner, shows four inner port boxes.**Creation:** Made by system.**Purpose:** Holds the four port boxes.**Parts:** Has a name and fixed inner list of four port boxes.**Execution behavior:** Doesn’t execute.

With split viewing, full aspect of this box is shown in the lower right viewer, even if it is using its abstract aspect.

**Demo box****category:** its own special “demo” category**Sample image:****Appearance:** Appears below the ports box, name always starts with “Demo,” always shows abstract aspect.**Creation:** Can only be made by authorized user.**Purpose:** Contains an intermingled sequence of notes and keys to play back for a demonstration to the user.**Parts:** Has a name and inner list. Only authorized user will ever see the inner list.**Execution behavior:** Doesn’t execute (but does “run”).

Special kind of “meta level” box. Has inner boxes of kinds that a regular, unauthorized user need not be concerned with.

**Class data box****category:** container**Sample image:****Appearance:** Has name “class data” in upper left corner.**Creation:** Fixed part of its class box.**Purpose:** Holds 0 or more data boxes that hold information used by the entire class, rather than just one particular instance of the class.**Parts:** Has a name and inner list.**Execution behavior:** When execution begins, a copy of this box is put in the static data box, with the name changed to be the name of the class box it belongs to.

**Class methods box****category:** container**Sample image:****Appearance:** Has name “class methods” in upper left corner.**Creation:** Fixed part of its class box.**Purpose:** Holds 0 or more method boxes that provide operations performed by the class rather than an instance.**Parts:** Has a name and inner list.**Execution behavior:** Doesn't execute.**Instance data box****category:** container**Sample image:****Appearance:** Has name “instance data” in upper left corner.**Creation:** Fixed part of its class box.**Purpose:** Holds 0 or more data boxes that hold information used by a single instance.**Parts:** Has a name and inner list.**Execution behavior:** During execution, when an instance of this box's class is created, a copy of this box is put in the heap box.**Instance methods box****category:** container**Sample image:****Appearance:** Has name “instance methods” in upper left corner.**Creation:** Fixed part of its class box.**Purpose:** Holds 0 or more method boxes that provide operations performed by an instance.**Parts:** Has a name and inner list.**Execution behavior:** Doesn't execute.

**Data box****category:** container**Sample image:****Appearance:** Has just three string parts, no inner boxes.

**Creation:** Hit `space` when focus is on the mark or a data box inside a class data box, an instance data box, an inputs box, or a locals box. Shortcut allows hitting `space` when focus is on the value string of many data boxes (depends on whether the type allows a space symbol to be in the value string).

**Purpose:** Provides for storage and retrieval of data during program execution by referring to the data box's name. Protects against storing an incorrect type of data.

**Parts:** Has a name string in the upper left corner, a type string in the upper right corner, and a value string on the bottom row.

**Execution behavior:** Is the target and source, respectively, for information storage and retrieval during execution, but does not itself execute. If an execution step attempts to store a value string of a different type than the data box has in its upper right corner, an execution error occurs.

Note: data boxes can occur in several different kinds of container boxes, including in copies of instance data boxes in the heap box and in copies of class data boxes in the static data box, both of which are created during execution.

The type string should be either the name of a class box or one of the primitive data types listed in the Summary Sheets.

**Method box****category:** container**Sample image:****Appearance:** Has four fixed inner boxes.

---

**Creation:** Hit `space` when focus is on the mark or a method box inside a class methods box or an instance methods box.

**Purpose:** Provides the basic unit of execution, in the sense that all execution proceeds through calls to method boxes. It receives inputs, performs operations, and possibly returns a value.

**Parts:** Has a name string and four fixed inner boxes. The first three are named “inputs,” “output,” and “locals,” and the fourth, referred to as the “instructions box,” is actually a sequence box.

**Execution behavior:** When a method box is called by a call box, a copy of it is made. If the method box belongs to a class methods box, then the name of the class is inserted in a special extra string part in the upper right corner. If the method box belongs to an instance methods box, then the reference number of the object that was called upon to perform the method is put in the upper right corner. Then, the copy method box is inserted in the stack box at the far left. The call box provides zero or more arguments strings which are inserted, in order, in the data boxes inside the method box’s inputs box. The types of the arguments must match the types of the data boxes, except that either `int` or `float` types strings may be stored in a `float` data box.

Then, the instructions box of the method box copy is executed. When this execution is complete, a value string may be returned through the output box of the method box, and execution continues after the call box that called this method box.

---

Note: the instructions in a method box copy can refer to data boxes in various places, depending on whether the method box is a class method box or an instance method box, as follows. If it is a method box inside a class methods box, then it can use data boxes in its own inputs box, its own locals box, and the class data box copy in the static data box. If it is a method box inside an instance methods box, it can in addition use data boxes in the copy of the instance data box in the heap box.

---

**Sequence box****category:** action**Sample image:****Appearance:** Has a downward-pointing triangular mark at its top edge. Arranges its inners vertically with decoration arrows pointing from one to the next.**Creation:** Press **Ⓢ** in a context where an action box is expected.**Purpose:** Holds a sequence of action boxes.**Parts:** Has zero or more inner boxes and no string parts. In particular, it has no name, hence has no abstract aspect.**Execution behavior:** Executes by having each of its inners execute, from top to bottom.

---

Note: the “instructions box” of every method box is actually a sequence box.

---

**Java box****category:** action, value**Sample image:****Appearance:** Its abstract aspect has a “J” in a box as its mark and arranges the string in a multi-line style. Its full aspect has a fixed inner list consisting of one box.**Creation:** Press **⓵** in any context where an action box or a value box is expected.**Purpose:** Allows use of Java-like code, in the name string, which is then automatically translated into the corresponding box which becomes the single inner box of the java box.**Parts:** The name string is used to hold the Java-like code. The inner list consists of just the single box that starts out as an empty box and is then converted to the desired box when the name string is translated.**Execution behavior:** When it executes, if it has not already translated its name string, it does so. The single inner box is then executed.

---

Note: the rules for the “java-like code” are quite elaborate and are not detailed in this reference guide because they can be found in any text about Java. The differences between true Java code and the code in a java box *are* detailed later in this reference guide.

The category of a java box can be either action or value, depending on the context in which it appears.

---

**Empty box****category:** action, value**Sample image:****Appearance:** Has a 3D perspective drawing of an empty box.**Creation:** Press `space` in a context where an action or a value box can be created.**Purpose:** Is used as a placeholder. When certain kinds of boxes are created, they contain, in their inner list, empty boxes which need to be converted to specific kinds of boxes.**Parts:** Has no string parts and no inner boxes.**Execution behavior:** When an empty box is executed in a context where an action is expected, it executes by doing nothing. In a context where a value is expected, it is an error for an empty box to be executed.**Branch box****category:** action**Sample image:****Appearance:** Has a downward-pointing triangular mark like the sequence box, but arranges its inner boxes in vertical pairs (except for the last one), with the pairs arranged horizontally. Decorative arrows leave each box in the top row both from the bottom downward and from the right edge rightward.**Creation:** Press `b` in a context where an action box is expected.**Purpose:** Allows for conditional execution, where exactly one of the action boxes in the lower row is executed, depending on the values produced by the value boxes in the upper row.**Parts:** Has inner list of one or more boxes.**Execution behavior:** Each of the value boxes in the upper row is evaluated, in left-to-right order. If a value of `true` is obtained, execution flows to the action box below. If a value of `false` is obtained, execution flows to the value box to the right. When one lower row action box is executed, the branch box execution is done. If none of the value boxes produce a value of `true`, then the last action box on the lower row is executed.

Note: when the focus is inside a branch box, on the mark or on an inner box other than the last one, press `space` to make a vertical pair of empty boxes, which can then be converted to the value box with a question and the corresponding action box. When either of a vertical pair is destroyed, the other one is, too.

**While loop box****category:** action**Sample image:**

**Appearance:** Has two inner boxes arranged vertically with decorative arrows indicating how execution flows—the top inner box has the “exit arrow” leaving to its right, while the bottom inner box has the “go back up and do it again” arrow leaving from its bottom edge.

**Creation:** Press **[w]** in a context where an action box is expected.

**Purpose:** Allows for repeated execution of its second inner box, with the question of whether to execute it again *before* it has been done the first time.

**Parts:** Has two fixed inner boxes.

**Execution behavior:** The first inner box executes to produce a **boolean** value. If the value is false, execution flows to the right and execution of the while loop box is complete. If the value is true, execution flows down, the second inner box is executed, and then execution flows back up to the first inner, to repeat the entire process.

**Do loop box****category:** action**Sample image:**

**Appearance:** Has two inner boxes arranged vertically with decorative arrows indicating how execution flows—the bottom inner box has the “exit arrow” leaving to its right, and also has the “go back up and do it again” arrow leaving from its bottom edge.

**Creation:** Press **[d]** in a context where an action box is expected.

**Purpose:** Allows for repeated execution of its first inner box, with the question of whether to execute it again *after* it has been done at least once.

**Parts:** Has two fixed inner boxes.

**Execution behavior:** The first inner box executes and then execution flows down to the second inner box. The second inner executes to produce a **boolean** value. If the value is false, execution flows to the right and execution of the do loop box is complete. If the value is true, execution flows back up to the first inner and the entire process is repeated.

**For loop box****category:** action**Sample image:**

**Appearance:** Has four inner boxes arranged vertically with decorative arrows indicating how execution flows—the second inner box has the “exit arrow” leaving to its right, while the bottom inner box has the “go back up and do it again” arrow leaving from its bottom edge.

**Creation:** Press **f** in a context where an action box is expected.

**Purpose:** Allows for repeated execution of its third inner box, with the question of whether to execute it again *before* it has been done the first time. Is just a while loop box with two extra boxes included for convenience (the first and fourth).

**Parts:** Has four fixed inner boxes.

**Execution behavior:** The first inner box executes one time only. Then the second inner box executes to produce a **boolean** value. If the value is false, execution flows to the right and execution of the for loop box is complete. If the value is true, execution flows down, and the third inner box is executed. Then the fourth inner box is executed, and execution flows back up to the second inner, to repeat the entire process (except for execution of the first inner box).

**Identifier box****category:** action**Sample image:**

**Appearance:** Has just one string with right-pointing triangular mark.

**Creation:** Press **i** in any context where a value is expected.



**Purpose:** Refers to a data box by name.

**Parts:** Has just a name string.

**Execution behavior:** Searches all the data boxes that are accessible from the context where it is executed until one with the matching name is found. Then it makes a copy of the value string in that data box to be its value, or uses the data box as the target for storage of a value. Is an execution error if no data box with the same name is found.

Note: the “accessible” data boxes depend on the method box in which the identifier box occurs. See the listing for the method box for details.



**Assignment box****category:** action**Sample image:****Appearance:** Has two inner boxes with a string part between them, and a decorative arrow from the second back to the first indicating data flow.**Creation:** Press  in a context where an action is expected.**Purpose:** Allows for storage of a value into a data box.**Parts:** Has two fixed inner boxes and a string part.**Execution behavior:** The first inner box executes to determine a data box in which to store a value. Then the second inner box executes to produce a value. Finally, if the string part is just =, the value is simply stored in the data box. If the = symbol is preceded by one of the five arithmetic operators, then the current value of the data box is combined with the value of the second inner, using the specified operator, and the resulting value is stored in the data box.**(Binary) operation box****category:** value**Sample image:****Appearance:** Has a string part between two inner boxes, with no decorations.**Creation:** Press  (for “operation”) in a context where a value is expected.**Purpose:** Performs a specified binary operation.**Parts:** Has two fixed inner boxes and a string part.**Execution behavior:** The first inner box executes to produce a value. Then the second inner box executes to produce a value. Finally these two values are combined, using the binary operation given in the string part, and the value of this box becomes the resulting value.

Note: the possible binary operations include arithmetic operations such as +, relational operations such as <, and logical operations such as & (for “and”). They are all listed in the Summary Sheets.

**Unary operation box****category:** value**Sample image:****Appearance:** Has a string part followed by an inner box.**Creation:** Press `⌘` in a context where a value is expected.**Purpose:** Performs a specified unary operation.**Parts:** Has one fixed inner box and a string part.**Execution behavior:** The inner box executes to produce a value. Then the value of the unary operation box is obtained by applying the unary operation specified by the string part to that value.

---

Note: the two possible unary operations are listed in the Summary Sheets.

---


**Call box****category:** value, action**Sample image:****Appearance:** Has a first inner box followed by decorative parentheses in between which can appear additional inner boxes.**Creation:** Press `⌘` in any context.**Purpose:** Initiates the process of calling a specified method box.**Parts:** Has one or more inner boxes.**Execution behavior:** The first inner box executes to determine what method box is being called. A copy of that method box is made and inserted into the stack box at the beginning. Then the inner boxes of the call box are executed, producing values for each which are copied into the data boxes inside the inputs box of the copy of the method box. Then the instructions box of that method box copy is executed, while this call box waits. When the method is finished executing, it may or may not return a value to become the value of this call box, at which time execution of the call box is finally complete.

---


Note: whether the call box is an action or a value depends on whether the method box it is calling returns a value.

---

**Grow box****category:** action


<b>Sample image:</b>	<b>Appearance:</b> Has a string part followed by an inner box with a decorative arrow indicating data flow.
<b>Creation:</b> Press  in a context where an action is expected.	<b>Purpose:</b> Increases or decreases a specified integer value by 1.
<b>Parts:</b> Has a string part and a fixed inner box.	<b>Execution behavior:</b> The inner box is executed to determine a data box, and then the value in that data box is increased by one or decreased by one, respectively, if the string part is ++ or --.


**Return box****category:** action

<b>Sample image:</b>	<b>Appearance:</b> Has an inner box with a decorative arrow pointing to the right and then upward.
<b>Creation:</b> Press  in a context where an action is expected.	<b>Purpose:</b> Causes execution of the method box in which it appears to complete and sends a value to the output box of the method.
<b>Parts:</b> One fixed inner box.	<b>Execution behavior:</b> The inner box executes, producing a value unless it is an empty box. The value, if there is one, is put in the output box for the method this box occurs inside. Then the method execution is complete.

Note: an empty box is allowed here, even though a value is expected, to allow for the return box to simply terminate execution of its method.

**Int, float, char, boolean, and string box****category:** value


<b>Sample image:</b>	<b>Appearance:</b> Each of these five has a string part followed by a decoration consisting of the first letter of the type, in uppercase, such as “I” for the integer literal box, in a context where a value is expected.
<b>Creation:</b> Press the uppercase letter for the desired type, such as  for the int literal box.	<b>Purpose:</b> Allows a literal value of any of the five primitive types to be specified in a value context.
<b>Parts:</b> One string part.	<b>Execution behavior:</b> Each of these five very similar kinds of boxes executes by simply producing a copy of its string part as its value.



**New operation box****category:** value**Sample image:****Appearance:** Has a decorative image of two offset rectangles indicating copying of a box, followed by a string part and two inner boxes.**Creation:** Press  in a context where a value is expected.**Purpose:** Creates a new instance of a class or one or two dimensional array object in the heap.**Parts:** Has a string part and two fixed inner boxes.**Execution behavior:** If both inner boxes are empty boxes, then the string part must be the name of a class and the new operation box executes by making a copy of the instance data box for that class, assigning it a reference number as its name in its upper left corner, putting the name of the class in the upper right corner, and inserting the resulting box in the heap box.

If the first inner box is not empty and the second is, then a one-dimensional array is created and inserted into the heap box, with a reference number in its upper left corner. The string part of the new operation box is put in the upper right corner of the array box in the heap, indicating that each component box in the array is a data box that can hold that type of data.

If both inner boxes are not empty, then a two-dimensional array is created in similar fashion.

Whether the new operation box creates a single instance of a class, a one-dimensional array, or a two-dimensional array, its value string becomes the reference number of the box that was created and inserted in the heap box.

**Member box****category:** value**Sample image:****Appearance:** Has an inner box followed by a decorative dot followed by a string part.**Creation:** Press  in a context where a value is expected.**Purpose:** Provides access to an individual data box or method box in a class box.**Parts:** Has a fixed inner box and a string part.**Execution behavior:** The inner box executes to determine either the name of a class box or the reference number of an instance of some class. Then, the string part is used to determine the data box or method box within the instance or class that is desired.

**Array box****category:** value**Sample image:****Appearance:** Has a fixed inner box followed by a second inner box enclosed in decorative left and right brackets.**Creation:** Press  in a context where a value is expected.**Purpose:** Provides access to a component of a one-dimensional array.**Parts:** Has two fixed inner boxes.**Execution behavior:** The first inner box executes to determine the reference number of the array in the heap box. Then the second inner box executes to determine an `int` value specifying which component of the array is referred to.**Array 2D box****category:** value**Sample image:****Appearance:** Has a fixed inner box followed by second and third inner boxes enclosed in decorative left and right brackets and separated by a decorative comma.**Creation:** Press  in a context where a value is expected.**Purpose:** Provides access to a component of a two-dimensional array.**Parts:** Has three fixed inner boxes.**Execution behavior:** The first inner box executes to determine the reference number of the array in the heap box. Then the second and third inner boxes execute to determine `int` values specifying, respectively, the row and column of the component of the array that is referred to.

**Port box****category:** container**Sample image:****Appearance:** Looks just like a data box, but occurs inside the ports box (all four port boxes shown to the left).**Creation:** Fixed inner box for ports box.**Purpose:** Holds incoming and outgoing symbols for getting and sending from and to disk files.**Parts:** Has three string parts.**Execution behavior:** Doesn't execute, but shows results of various system method calls as follows. The upper right corner string shows --> to indicate that the port has been opened for output, and shows <-- to indicate that the port has been opened for input. The lower string part shows the current buffer of symbols that are either being prepared to be sent to a disk file or have been obtained in a bunch from a disk file.

## Overview of Java-Like Code

The Java-like instructions that can be written in a java box are very similar to the syntax and semantics of Java, as summarized below, but differ from Java in some ways, as detailed below.

First, the java-like code that can be written in a java box is restricted to the expression subset of Java (actually, other kinds of statements are provided, but the jBoxes philosophy says that the branch box and loop boxes should be used instead).

More specifically, a java box that is in a context where an action is expected can contain a sequence of one or more *statements*, each terminated by a semi-colon, where each statement is either an *assignment statement*, a *method call*, an *incremental statement*, or a *return statement*.

A java box that is in a context where a value is expected can contain just a single *expression*.

An assignment statement has an expression that specifies a data box, followed by an assignment operator which is = or one of the arithmetic operators preceding =, followed by an expression. It translates in the obvious way to an assignment box.

A method call is a special kind of expression, consisting of a method name followed by zero or more argument expressions separated by commas and enclosed in parentheses.

An incremental statement is either ++ or -- followed by an expression that specifies a data box. It translates in the obvious way to a grow box.

A return statement consists of the word `return` followed by an expression. It translates in the obvious way to a return box.

An expression consists of anything that can be formed by combining binary and unary operators, parentheses, method calls, names of data boxes, names of one-dimensional arrays followed by expressions enclosed in brackets, names of two-dimensional arrays followed by two expressions, each enclosed in brackets, names of classes or data box names followed by a period followed by the name of a data box or method box in some class, and literal values. The rules for all of this are the same as for Java.

Note in particular that the five types of literal values in jBoxes correspond to the same literal values in Java.

In addition to the fact that not all parts of the Java language can or should be used in a java box, the main difference is that jBoxes uses system method calls, whereas Java uses a very elaborate system of predefined classes.

Also, the `string` type in jBoxes is a primitive data type, whereas in Java it is written as `String` and is simply a special, commonly used class.

## Translation of Java-Like Code

When a java box is about to be executed for the first time, or when translation is requested manually by pressing `ctrl+t`, jBoxes translates the Java-like code into a corresponding box diagram, which is put into the java box as its single inner box.

If there is a syntax error in the code written in the java box, then an error message will be displayed. After hitting `enter` to dismiss that error message, you can hit the down arrow to move the focus into the java box and onto the symbol where that syntax error was encountered.

You can see, by changing aspects, exactly what box diagram is created from the java code you have written. It is also useful to change aspects during execution so that instead of executing as a single box, you can see the step-by-step execution of the simpler boxes that the java box translated into.