PART I :

SCRIPT SYNTAX

Chapter 1: **Getting your tools ready**

Chapter 2: **Variables, operators and Functions**

Chapter 3: **Loops and conditions**

Chapter 4: **Strings, Number, Date and Math**

Chapter 5: **Arrays and Objects**

Chapter 6: **Error handling**

CHAPTER 1 :

GETTING YOUR TOOLS READY

INTRODUCTION

It's just JavaScript!

Scripting in SimPhy is based on *Mozilla Rhino* which is an ECMA- 5 compliant implementation of the language. While SimPhy runs ECMA-compliant JavaScript, it is important to note that objects normally accessible in a web browser are not available, for example, `Console`, `window`, and so on. There are equivalents for these which we will see later on.

We have tried making it very close to common JavaScript to avoid learning a new language.

Note: *let, arrow functions and classes are not implemented, since these are new features introduced in ECMA-6.*

Where is my DOM ?

Although SimPhy doesn't support the HTML5 document model, it has a very similar way to access and create almost all standard GUI elements like button, text-field, slider, window etc and listen to their events. All GUI related actions are done through global object GUI.

SimPhy has a WYSWYG-type editor to create and edit widgets. For details refer *Part 2*.

What can I do with scripting?

All elements used in SimPhy (bodies, joints, shapes, widgets, graphics, sound, etc.) are accessible through scripting which lets user:

- create real world physics simulation with minimal code,
- create Interactive simulations with widgets,
- make games full of graphics and sound, and
- have full control over behavior of objects.

SIMPHY SCRIPTING ENVIRONMENT

Programming in SimPhy is inspired from my childhood code editor VISUAL BASIC 6.0, therefore it lets you create rapid application development with WYSWYG editor.

Toolbar

Toolbar contains few self explanatory buttons, just hover mouse over them you will get what they do.

The most important one is the **Save** button (button with floppy icon). You must save your script before closing script editor or running simulation/script, because edits are not automatically saved.

Once you write a significant amount of code, it is recommended to save the simulation as well, since bad code practices may result in the app becoming non-responsive or may require to force quit without giving you the opportunity to save it.

Event Source

From this combobox, you can filter/choose events for which the script is to be written.

For example, on choosing `'world'` event, cursor moves to world related functions and Function Selector is filled with all possible `world` events on which scripting can be done. All non-event functions are grouped in `'General'` category and the Function Selector combobox is filled with non-event `general` functions.

Function Selector

Whenever you select a function from the Function Selector, cursor moves to a declaration of that function in the editor, if function already exists, or creates a new function for the corresponding selection.

Outline View

The tree view contains outline of code, the variables and the function declaration. It automatically populates on editing script. Just click on any of the node to move to its declaration in the editor.

Editor

This is where you spend most of the time. The Editor may catch a few syntax errors while writing the code whereas runtime errors are caught whenever the script actually runs. For example, if the script for the click-event of a button contains some runtime error, it will be caught only when the button is clicked and the script is executed.

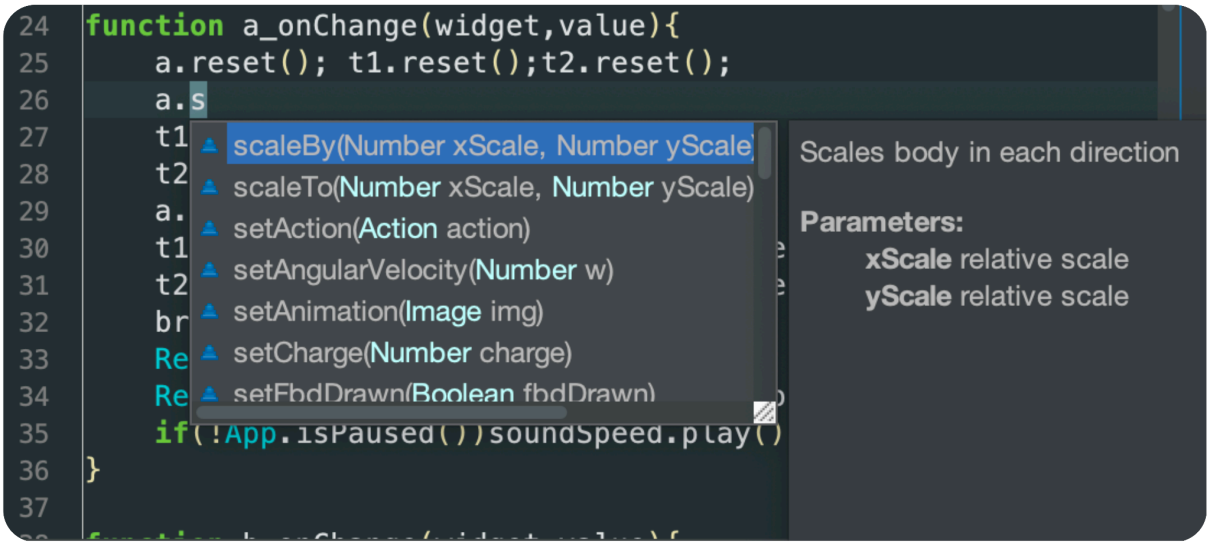
The most amazing part of the editor is that it is **IntelliSense enabled**, i.e. when you press '**.**' after a variable, available members and methods are shown as a popup, from where you can select them to auto-complete the code.

Error Strip

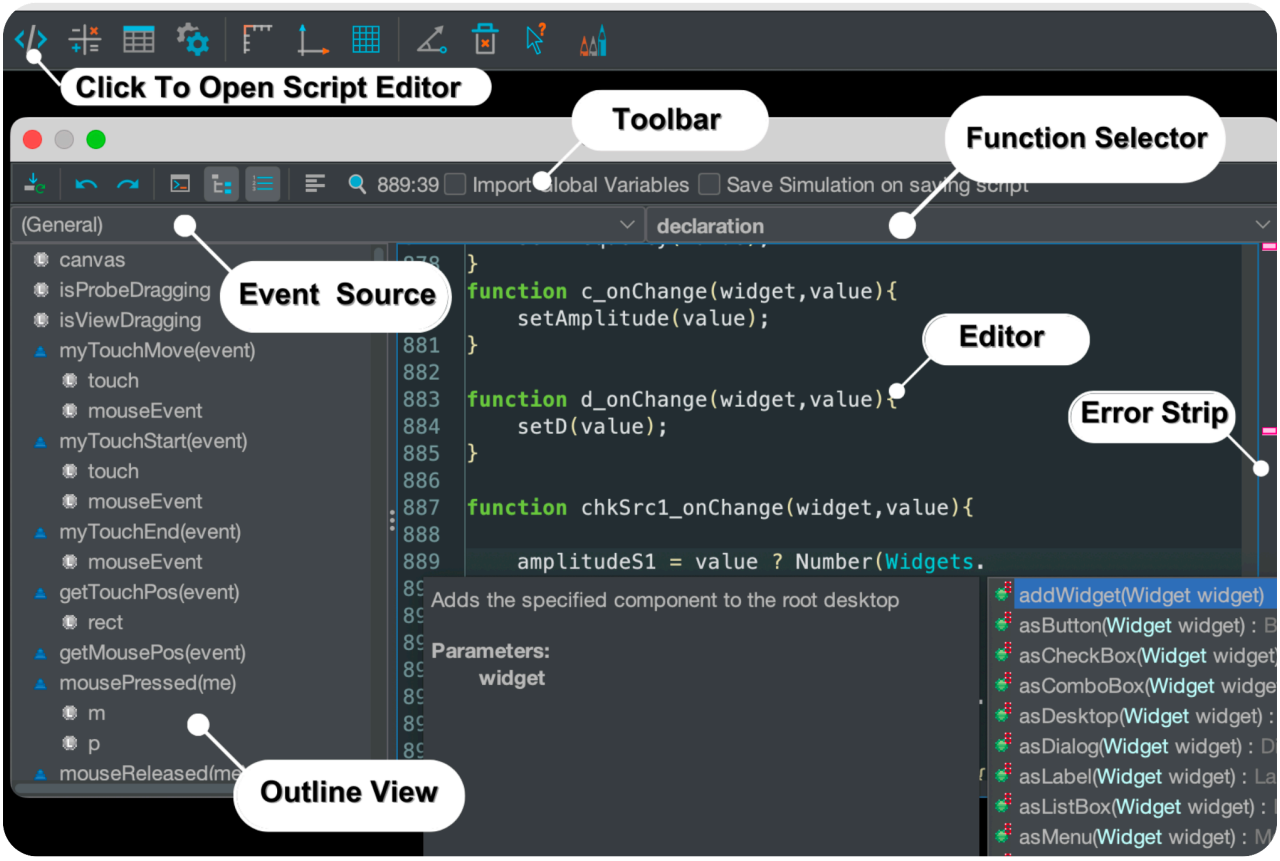
Whenever a script catches some error (syntax or runtime), it is indicated by a squirrel red line under the error-containing line. There also exists a dedicated vertical strip as shown which contains some red lines indicating the error. On clicking these red lines, the cursor moves to the corresponding location in the editor.

Autocomplete

Use **Ctrl + Space** for suggestion and autocompletion, for example, pressing Ctrl + Space after writing 's' will show following popup, it automatically enters placeholder/variable name on selecting one of these items.



The screen shot of **Script Editor** as in *version 3.2* of **SimPhy** is shown in the next page. Click 'Script Editor' button in toolbar to open script editor, obviously it will not contain code as shown in the screen shot, it will just be a blank window.



Shortcuts

- Ctrl +S
- Ctrl F (doesn’t work)
- Ctrl Space
- Ctrl J
- Ctrl+Shift+O (Cmd+Shift+O on OS X) opens a "Go to Member" popup, a la Eclipse.

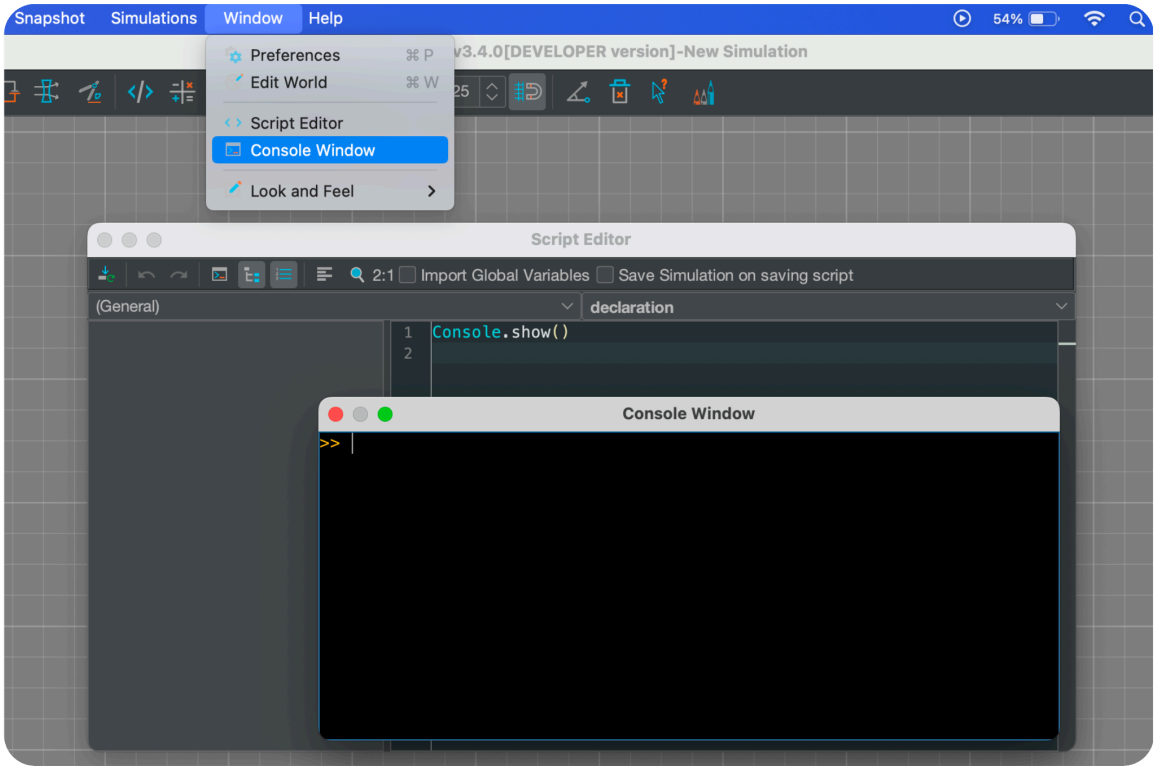
Shortcuts not working!

CONSOLE WINDOW

The basic output window you will initially need for debugging/displaying output is the **Console window**, which can be accessed using the keyword `Console`.

To open Console:

- either select **Console Window** from the "Windows" menu in SimPhy,
- or use command `Console.show()` in script to make it visible.



Console Methods:

- **`show()`** — shows Console window
- **`hide()`** — hides Console window

- `print(String)` — prints text in the Console window in same line
- `println(String)` — prints text in the Console window in new line
- `printf(String,Object...)` — prints formatted text in the Console window
- `format(String,Object...)` — returns formatted text without displaying on it
- `log(String)` — prints in new line with **green color**
- `info(String)` — prints in new line with **blue color**
- `error(String)` — prints in new line with **red color**

For example:

```
Console.show();
Console.println("Hi Dear !!!");
Console.printf("My answer is %.8f", 47.65734);
Console.log("Hi I am Log");
Console.info("Hi I am Info !");
Console.error("Hi I am Error !");
```

CHAPTER 2 :

VARIABLES, OPERATORS & FUNCTIONS

VARIABLES

Var

Old-style JavaScript had one way of *defining a variable* using the keyword **var**. So, we would write:

```
var a = 'hello'; // a string
var b = 5; // a number
var e = [ 1, 2, 3, 4, 'f', 'g', 'h', 'i' ]; // an array
containing numbers and strings
```

ES6 introduced **const** and **let**, which are basically designed to replace *var*. You can still use *var* to keep things backward compatible, but you don't need to anymore, and we'll never use *var* in this book.

Const

const is used to assign data to a variable that cannot be changed later.

```
const x = 5;
// x must always equal 5 for the lifetime of your program
```

Attempting to change it later will cause an error:

```
x = 6; // Error! Can't update variable
```

let

When using *let* to define a variable you are saying that the value will get changed sometime in the future:

```
let y = 5;
```

Unlike the *const* example, doing this is fine:

```
y = 6; // no error, y now holds the number 6
```

The reason for this split, aside from making it easier for us humans to read the code, is so that the computer can optimize the code by knowing which things may need to be updated.

You should always define a variable using *const* unless you know it will need to be changed at some point.

Case Sensitive

Variable names in JavaScript are case sensitive:

```
const apples = 4;

const Apples = 5;
```

Here, apples and Apples are two different variables.

CamelCase

We'll always use CamelCase for variable names that consist of more than one word:

```
const myName = 'Lewy';

let modelHasLoaded = false;
```

For example:

```
var a;                          // variable
var b = "init";                 // string
var c = "Hi" + " " + "Joe";     // = "Hi Joe"
var d = 1 + 2 + "3";            // = "33"
var e = [2,3,5,8];              // array
var f = false;                  // Boolean
var g = /()/;                   // RegEx
var h = function(){};           // function object
const PI = 3.14;                // constant
var a = 1, b = 2, c = a + b;    // one line
let z = 'zzz';                  // Will display Error that let
is not defined
```

DATATYPES

- **`String :`** A sequence of text/characters is known as a *string*. To signify that the value is a string, you must enclose it in **quotation marks**.

  For example:

```
var myVariable = 'Bob';
```

- **`Number :`** A *number* doesn't have quotes around them.

  For example:

```
var myVariable = 10;
```

- **`Boolean :`** A True/False value. The words true and false are special keywords in JS and don't need quotes.

  For example:

```
var myVariable = true;
```

- **`Array :`** A structure that allows you to store multiple values in one single reference.

  For example:

```
var myVariable = [1,'Bob','Steve',10];
/* Refer to each member of the array like this:
myVariable[0], myVariable[1], etc. */
```

- **`Object :`** Basically, anything! Everything in JavaScript is an *object* and can be stored in a variable. Keep this in mind as you learn!

  For example:

```
var name = {firstName:"John", lastName:"Doe"};
```

Values

- String : "flower", 'John'
- Boolean : 18, 3.14, 0b10011, 0xF6, NaN
- Boolean : false, true
- Boolean : undefined, null , Infinity

For example:

```
var age = 18;                                // number
var name = "Jane";                           // string
var name = {first:" Jane", last:" Doe"};     // object
var truth = false;                           // boolean
var sheets = ["HTML","CSS","JS"];            // array
var a; typeof a;                             // undefined
var a = null;                                // value null
```

TypeOf

typeof operator is used to return the data type of an operand.

```
var a;
Console.println(typeof a);          //outputs undefined
Console.println(typeof "SyntaxPage"); //outputs string
Console.println(typeof 7);            //outputs number
Console.println(typeof true);         //outputs boolean
a={}
Console.println(typeof true);         //outputs object
a=[]
Console.println(typeof a);            //outputs array
Console.println(typeof Console.println); //outputs boolean
```

Dynamic Typing

JavaScript is “ loosely” or “ dynamically” typed language. In short, this means that we don’ t care what data type a variable holds. This means that we can do this:

```
let x = 5; // x holds a number
// sometime later
x = 'lemon'; // x now holds a string.
```

This makes the language very flexible, but it can also lead to confusion as it puts the responsibility on you, the programmer, to remember what kind of data a variable holds. One of the benefits of const is that you know a variable will always hold the initial data that you assigned to it.

OPERATORS

Arithmetic Operators

```
a = b + c - d;              // addition, substraction
a = b * (c / d);            // multiplication, division
x = 100 % 48;               // modulo; 100 / 48 remainder = 4
a++; b--;                   // postfix increment and decrement
```

Bitwise Operators

```
&             AND                 5 & 1 (0101 & 0001) 1 (1)
|             OR                  5 | 1 (0101 | 0001) 5 (101)
~             NOT                 ~5 (~0101) 10 (1010)
^             XOR                 5 ^ 1 (0101 ^ 0001) 4 (100)
<<          left shift            5 << 1 (0101 << 1) 10 (1010)
>>         right shift            5 >> 1 (0101 >> 1) 2 (10)
>>>     zero fill right shift     5 >>> 1 (0101 >>> 1) 2 (10)
```

Logical Operators

```
a * (b + c)             // grouping
person.age              // member
Person[age]             // member
!(a == b)               // logical not
a != b                  // not equal
typeof a                // type (number, object, function...)
x << 2 x >> 3           // minary shifting
a = b                   // assignment
a == b                  // equals
a != b                  // unequal
a === b                 // strict equal
a !== b                 // strict unequal
a < b a > b             // less and greater than
a <= b a >= b           // less or equal, greater or eq
a += b                  // a = a + b (works with - * %...)
a && b                  // logical and
a || b                  // logical or
```

Truthy & Falsy

Nearly everything is interpreted as **true** (*truthy*) in a Boolean context, while things like **false**, **0**, **-0**, **null**, **undefined**, **''** and **""** (empty strings) and **NaN** (not a number) are interpreted as **false** (*falsy*).

FUNCTIONS

Function Declarations By Name

```
function square(number) {
     return number * number;
}
```

Primitive parameters (such as a number) are passed to functions **by value**, while an object (i.e. a nonprimitive value, such as an *Array* or a user-defined object) is passed **by reference**.

Function Expressions

Such a function can be anonymous; it does not have to have a name. For example, the function square could have been defined as:

```
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) { return n < 2 ? 1 : n *
fac(n -1); };
Console.log(factorial(3));
```

Function Return

Every function in JavaScript returns **undefined** unless otherwise specified.

Functions are *function objects*. In JavaScript, anything that is not a primitive type (**undefined**, **null**, **boolean**, **number**, or **string**) is an object. Objects in JavaScript are extremely versatile. Because of this, we can even pass a function as a parameter into another function.

Function Hoisting

Functions must be in scope when they are called, but the function declaration can be hoisted (appear below the call in the code), as in this example:

```
Console.log(square(5));
/* ... */
function square(n) { return n * n; }
```

The scope of a function is the function in which it is declared, or the entire program if it is declared at the top level.

Note: *Function hoisting only works with function declaration and not with function expression.*

Function Scopes

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined. In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

Nested Functions & Closures

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a closure. Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

Using The Arguments Object

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

```
arguments[i]
```

where i is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be arguments[0]. The total number of arguments is indicated by *arguments.length*.

Function Parameters

If a parameter is not provided, its value becomes undefined.

- ### Default Parameters

  Not Implemented yet

```
function show Message(from, text = "no text given") {
alert( from + ": " + text );
}
showMessage("Ann"); // Ann: no text given
```

- **Rest Parameters**

  Not Implemented yet

```
function multiply(multiplier, ...theArgus) {
   return theArgus.map(x => multiplier * x);
}
```

- **Arrow Functions**

  Not Implemented yet

Global Functions

The following are globally defined system functions, which are available by default:

```
eval();      // executes a string as if it was a script code
Boolean('true');       // return true
String(23);            // return string from number
(23).toString();       // return string from number
Number("23");          // return number from string
isFinite();            // variable is a finite, legal number
isNaN();               // variable is an illegal number
parseFloat();     // returns floating point number of string
parseInt();       // parses a string and returns an integer
parseBoolean();   // parses a string and returns an integer
```

Note: *Global functions can be accessed by using **this** in the global scope.*

CHAPTER 3 :

LOOPS & CONDITIONS

IF ELSE

If Else If

```
if ((age >= 14) && (age < 19)) {     // logical condition
     status = "Eligible."; // executed if condition is true
} else if ((age < 14) && (age >10)) { // optional logical
condition
status = "Eligible with parents consent";// executed if
condition is true
} else {                    // else block is optional
status = "Not eligible."; // executed if condition is false
}
```

Switch

```
switch (new Date().getDay()) {      // input is current day
   case 6:                          // if (day == 6)
       text = "Saturday";
       break;
   case 0:                          // if (day == 0)
       text = "Sunday";
       break;
   default:                         // else...
       text = "Whatever";
}
```

LOOPING

For

The most common way to create a loop in JavaScript.

```
for (var i = 0; i < 10; i++) {
   Console.println(i + ": " + i*3 + "<br />");
}
```

While

It sets up the conditions under which a loop executes.

```
var i = 1;          // initialize
while (i < 100) {   // enters the cycle if the statement is
true
   i *= 2;          // increment to avoid infinite loop
   Console.println (i + ", ");    // output
}
```

Do While

Similar to the while loop, however, it executes at least once and performs a check at the end to see if the condition is met to execute again.

```
var i = 1;              // initialize
do {                    // enters cycle at least once
   i *= 2;              // increment to avoid infinite loop
   Console.println (i + ", "); // output
} while (i < 100)       // repeats cycle if statement is
true at the end
```

Break

It is used to stop and exit the cycle at certain conditions.

```
for (var i = 0; i < 10; i++) {
   if (i == 5) { break; }      // stops and exits the cycle
   Console.println (i + ", "); // last output number is 4
}
```

Continue

It skips parts of the cycle if certain conditions are met.

```
for (var i = 0; i < 10; i++) {
   if (i == 5) { continue; } // skips the rest of the cycle
   Console.println (i + ", ");   // skips 5
}
```

CHAPTER 4 :

STRINGS, NUMBERS, DATE & MATH

STRINGS

Declarations

```
var person = "John Doe";
var m=new String("hello");
```

Escape Characters

- `\'` — Single quote
- `\"` — Double quote
- `\\` — Backslash
- `\b` — Backspace
- `\f` — Form feed
- `\n` — New line
- `\r` — Carriage return
- `\t` — Horizontal tabulator

String Methods

- `charAt()` — Returns a character at a specified position inside a string
- `charCodeAt()` — Gives you the Unicode of character at that position
- `concat()` — Concatenates (joins) two or more strings into one
- `romCharCode()` — Returns a string created from the specified sequence of UTF-16 code units
- `indexOf()` — Provides the position of the first occurrence of a specified text within a string
- `lastIndexOf()` — Same as *indexOf()* but with the last occurrence, searching backwards
- `match()` — Retrieves the matches of a string against a search pattern
- `replace()` — Find and replace specified text in a string
- `search()` — Executes a search for a matching text and returns its position
- `slice()` — Extracts a section of a string and returns it as a new string
- `split()` — Splits a string object into an array of strings at a specified position
- `substr()` — Similar to slice() but extracts a substring depended on a specified number of characters
- `substring()` — Also similar to slice() but can't accept negative indices

- `toLowerCase()` — Convert strings to lowercase
- `toUpperCase()` — Convert strings to uppercase
- `valueOf()` — Returns the primitive value (that has no properties or methods) of a string object

```
var abc = "abcdef,ghijklmnopqrstuvwxyz";
var str2 = "mdbjjd"
var esc = 'I don't \n know'; // \n new line
var len = abc.length;          // string length
abc.indexOf("lmno");           // find substring, -1 if
doesn't contain
abc.lastIndexOf("lmno");       // last occurrence
abc.slice(3, 6);               // cuts out "def", negative
values count from behind
abc.replace("abc","123");      // find and replace, takes
regular expressions
abc.toUpperCase();             // convert to upper case
abc.toLowerCase();             // convert to lower case
abc.concat(" ", str2);         // abc + " " + str2
abc.charAt(2);          // character at index: "c"
abc[2];                 // unsafe, abc[2] = "C" doesn't work
abc.charCodeAt(2);      // character code at index: "c" -> 99
abc.split(",");         // splitting a string on commas gives
an array
abc.split("");          // splitting on characters
(128).toString(16);     // number to hex(16), octal (8)
```

NUMBERS

Number Properties

- `MAX_VALUE` — The maximum numeric value representable in JavaScript
- `MIN_VALUE` — Smallest positive numeric value representable in JavaScript
- `NaN` — The “Not-a-Number” value
- `NEGATIVE_INFINITY` — The negative Infinity value
- `POSITIVE_INFINITY` — Positive Infinity value

Number Methods

- `toExponential()` — Returns a string with a rounded number written as exponential notation
- `toFixed()` — Returns the string of a number with a specified number of decimals
- `toPrecision()` — String of a number written with a specified length
- `toString()` — Returns a number as a string
- `valueOf()` — Returns a number as a number

```
var pi = 3.141;
pi.toFixed(0);                  // returns 3
pi.toFixed(2);                  // returns 3.14
pi.toPrecision(2);              // returns 3.1
pi.valueOf();                   // returns number
Number(true);                   // converts to number
Number(new Date());             // number of milliseconds since
1970
parseInt("3 months");           // returns the first number: 3
parseFloat("3.5 days");         // returns 3.5
Number.MAX_VALUE;               // largest possible JS number
Number.MIN_VALUE;               // smallest possible JS number
Number.NEGATIVE_INFINITY;       // -Infinity
Number.POSITIVE_INFINITY;       // InfinityMAX_VALUE
```

Returns For A+B, A-B & A*B Type

```
var pi = 3.14;
var a = “mscnnd”;
var b = 2;

/*Treated as strings if one of them is string*/

Console.log(pi+a);          // returns 3.14mscnnd

/*Treated as numbers, if both are parsable to numbers
else, returns undefined*/

Console.log(pi-a);          // returns NaN
Console.log(pi*a);          // returns NaN
Console.log(pi-b);          // returns 1.14
Console.log(pi*b);          // returns 6.28
```

String To Numbers

There are many ways to convert a String to a Number.

1. Using `parseFloat()` and `parseInt()`

The **`parseFloat()`** method converts a string into a point number (a number with decimal points) and you can even pass strings with random text in them.

```
var text = '3.14someRandomStuff';
var pointNum = parseFloat(text);     // returns 3.14
```

The **`parseInt()`** method converts a string into an integer (a whole number) and accepts two arguments. The first argument is the string to convert. The second argument is *optional*, called as radix. This is the base number used in mathematical systems.

```
var n1 = true;
var n2 = "1str";
var n3 = "123";
var n4 = "str123str1";

n1 = Number(n1);     //typeof n1 =number & n1 =1
n2 = Number(n2);     //typeof n2 =number & n2 =NaN
n3 = Number(n3);     //typeof n3 =number & n3 =123
n4 = parseInt(n4);   //n4=123

var integer = parseInt('101', 2);    // returns 5
```

2. Using `Number()`

The `Number()` method converts a string to a number.

Sometimes it's an integer, other times it's a point number. And if you pass in a string with random text in it, you'll get `NaN`, an acronym for "Not a Number."

As a result of this inconsistency, it's a less safe choice than `parseInt()` and `parseFloat()`. If you know the format of the number you'd like, use those instead. If you want the string to fail with `NaN`, if it has other characters in it, `Number()` may actually be a better choice.

```
Number('123');                  // returns 123
Number('12.3');                 // returns 12.3
Number('3.14someRandomStuff');  // returns NaN
Number('42px');                 // returns NaN
Number(023)
// returns 19, treats 023 as an octal number
```

3. Using Arithmetic operators

- **Using operator "+" before String**

```
+'10.20' //10.2
```

- **Using `Math.floor()` to convert to integers**

```
Math.floor('10,000')        //NaN
Math.floor('10.000')        //10
```

- **Using *1 or /1 to convert to numbers**

```
'10,000' * 1          //NaN
'10.000' * 1          //10
'20' / 1              // 20
'1.2'-0               // 1.2
'2.1ds'-0             // NaN
```

Note: *It will return undefined if num is non-numeric*

MATH

Math Properties

- **`E`** — Euler's number
- **`LN2`** — The natural logarithm of 2
- **`LN10`** — Natural logarithm of 10
- **`LOG2E`** — Base 2 logarithm of E
- **`LOG10E`** — Base 10 logarithm of E
- **`PI`** — The number PI
- **`SQRT1_2`** — Square root of 1/2
- **`SQRT2`** — The square root of 2

Math Methods

- **`abs(x)`** — Returns the absolute (positive) value of x
- **`acos(x)`** — The arccosine of x, in radians
- **`asin(x)`** — Arcsine of x, in radians

- `atan(x)` — The arctangent of x as a numeric value
- `atan2(y,x)` — Computes the angle from the X-axis to a point (x,y)
- `ceil(x)` — Value of x rounded up to its nearest integer
- `cos(x)` — The cosine of x (x is in radians)
- `exp(x)` — Value of Ex
- `floor(x)` — The value of x rounded down to its nearest integer
- `log(x)` — The natural logarithm (base E) of x
- `max(x,y,z,...,n)` — Returns the number with the highest value
- `min(x,y,z,...,n)` — Same for the number with the lowest value
- `pow(x,y)` — X to the power of y
- `random()` — Returns a random number between 0 and 1
- `round(x)` — The value of x rounded to its nearest integer
- `sin(x)` — The sine of x (x is in radians)
- `sqrt(x)` — Square root of x
- `tan(x)` — The tangent of an angle

For example:

```
var pi = Math.PI;       // 3.141592653589793
Math.round(4.4);        // = 4 - rounded
Math.round(4.5);        // = 5
Math.pow(2,8);          // = 256 - 2 to the power of 8
Math.sqrt(49);          // = 7 - square root
Math.abs(-3.14);        // = 3.14 - absolute, positive value
Math.ceil(3.14);        // = 4 - rounded up
Math.floor(3.99);       // = 3 - rounded down
Math.sin(0);            // = 0 - sine
Math.cos(Math.PI);      // = -1 - cos(π)
```

```
Math.min(0, 3, -2, 2); // = -2 - the lowest value
Math.max(0, 3, -2, 2); // = 3 - the highest value
Math.log(1);           // = 0 natural logarithm
Math.exp(1);           // = 2.7182pow(E,x)
Math.random();         // random number between 0 and 1
Math.floor(Math.random() * 5) + 1;
// random integer, from 1 to 5
```

Evaluating Expressions

To Evaluate/Execute block of code/expressions we can use **eval(string)**.

- **Parameters (`String`)**

  A string represents a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

- **Return Value**

  Returns the completion value of evaluating the given code. If the completion value is empty, *undefined* is returned.

For example:

```
var x = 10;
var y = 20;
var a = eval("x * y") ;      //200
var b = eval("2 + 2") ;      //4
var c = eval("x + 17");      //27
var res = a + b + c;
Console.log(eval('2 + 2'));  // expected output: 4
```

DATE

Setting Dates

- `Date()` — Creates a new date object with the current date and time
- `Date(2017, 5, 21, 3, 23, 10, 0)` — Creates a custom date object. The numbers represent year, month, day, hour, minutes, seconds, milliseconds. You can omit anything you want except for year and month.
- `Date("2017-06-23")` — Date declaration as a string
- `parse` — Parses a string representation of a date, and returns the number of milliseconds since January 1, 1970

Pulling Date and Time Values

- `getDate()` — Gets the day of the month as a number (1-31)
- `getDay()` — The weekday as a number (0-6)
- `getFullYear()` — Year as a four digit number (yyyy)
- `getHours()` — Gets the hour (0-23)
- `getMilliseconds()` — The millisecond (0-999)
- `getMinutes()` — Gets the minute (0-59)
- `getMonth()` — Month as a number (0-11)
- `getSeconds()` — Gets the seconds (0-59)
- `getTime()` — Gets the milliseconds since January 1, 1970

Set Part of a Date

- `setDate()` — Set the day as a number (1-31)

- `setFullYear()` — Sets the year (optionally month and day)
- `setHours()` — Sets the hour (0-23)
- `setMilliseconds()` — Sets the milliseconds (0-999)
- `setMinutes()` — Sets the minutes (0-59)
- `setMonth()` — Set the month (0-11)
- `setSeconds()` — Sets the seconds (0-59)
- `setTime()` — Sets the time (milliseconds since January 1, 1970)

For example:

```
Mon May 06 2019 15:32:18 GMT+0530 (India Standard Time)

var d = new Date();
Number(d);                  //1557136938064 milliseconds
passed since 1970
new Date("2017-06-23");     // date declaration
new Date("2017");           // is set to Jan 01
new Date("2017-06-23 12:00:00-09:45"); // date - time
YYYY-MM-DDTHH:MM:SSZ
new Date("June 23 2017");   // long date format
new Date("Jun 23 2017 07:45:00 GMT+0100 (Tokyo Time)");
// time

var d = new Date();
d.getDate();                // day as a number (1-31)
d.getDay();                 // weekday as a number (0-6)
d.getFullYear();            // four digit year (yyyy)
d.getHours();               // hour (0-23)
d.getMilliseconds();        // milliseconds (0-999)
d.getMinutes();             // minutes (0-59)
d.getMonth();               // month (0-11)
d.getSeconds();             // seconds (0-59)
d.getTime();                // milliseconds since 1970
```

CHAPTER 5 :

ARRAYS & OBJECTS

ARRAYS

Array Declaration

```
var dogs=[]      //empty array
var dogs = ["Bulldog", "Beagle", "Labrador"]
var dogs = new Array("Bulldog", "Beagle", "Labrador");
// declaration
```

JavaScript arrays are objects, that is, you can add string properties to your array, not just numbers.

Array Length

JavaScript array's **length** property and numerical properties are connected. Several of the built-in array methods (e.g., `join()`, `slice()`, `indexOf()`, etc.) take into account the value of an array's length property when they're called. Other methods (e.g., `push()`, `splice()`, etc.) also result in updates to an array's length property.

You can set the length property to truncate an array at any time. When you extend an array by changing its length property, the number of actual elements increases; for example, if you set length to 3 when it is currently 2, the array now contains 3 elements, which causes the third element to be a non-iterable empty slot.

Accessing Array Elements

```
• for (var i = 0; i < arr.length; ++i)
• for (var i in arr)
• for each (var v in arr)
• arr.forEach(function(v, i) { /* ... */ })
```

Note: *The first two looping constructs give you access to the index in the array, not the actual element, while the other two constructs give you access to the array element itself.*

The for…of is not implemented yet.

Takeaway: for/in and forEach() skip empty elements, also known as "holes", in the array.

For example:

```
var array = ['first item', 20, 30];
array['test'] = 40;
array.test2 = 50;
array.push('last item');

Console.info("arr length=" + array.length); //prints 4
(only numerical entries treated as elements of array

//iterates over numeric properties only
for (var i = 0; i < array.length; i++) {
     Console.info(array[i]);
     // prints first item, 20, 30, last item
}
```

Continued in the next page…

```
//iterates over properties considering array as an object
for (var x in array) {
     Console.log(x); //prints 0, 1, 2, 3, 'test', 'test2'
}

//iterates over values considering array as an object
for each(var x in array) {
     Console.info(x);
     // prints first item, 20, 30, last item, 40, 50
}

//iterates over numeric properties only considering array
as an object
array.forEach(function(value, index) {
     Console.log(index + ":" + value);
     //prints 0:first item, 1:20, 2:30 3:last item
});
```

Array Methods

```
var dogs = ["Bulldog", "Beagle", "Labrador"];
dogs = new Array("Bulldog", "Beagle", "Labrador");
// declaration

dogs.toString();
// convert to string: results "Bulldog,Beagle,Labrador"

dogs.join(" * ");
// join: "Bulldog * Beagle * Labrador"

dogs.pop();
// remove last element
dogs.push("Chihuahua");
// add new element to the end
```

Continued in the next page…

```
dogs[dogs.length] = "Chihuahua";
// the same as push
dogs.shift();
// remove first element
dogs.unshift("Chihuahua");
// add new element to the beginning

delete dogs[0];
// change element to undefined (not recommended)

dogs.splice(2, 0, "Pug", "Boxer");
//add elements (where, how many to remove, element list)

var animals = dogs.concat(cats,birds);
//join 2 arrays(dogs followed by cats and birds)

dogs.slice(1,4);          // elements from [1] to [4-1]
dogs.sort();              // sort string alphabetically
dogs.reverse();           // sort string in descending order
x.sort(function(a, b){return a - b});    // numeric sort
x.sort(function(a, b){return b - a});
// numeric descending sort

highest = x[0];
// first item in sorted array is the lowest (or highest)
value

x.sort(function(a, b){return 0.5 - Math.random()});
// random sorting
```

OBJECTS

Objects in JavaScript are defined using curly braces: {}, and hold collections of data in [key, value] pairs. Everything that is not one of the above primitive datatypes is an object, including *functions and arrays*.

Creation of Objects

1. Using the Object Literal syntax :

Object literal syntax uses the {…} notation to initialize an object and its methods/properties directly.

For example:

```
var obj = {
    member1 : value1,
    member2 : value2,
};
```

These members can be anything – strings, numbers, functions, arrays or even other objects. An object like this is referred to as an **object literal**. This is different from other methods of object creation which involve constructors and classes or prototypes, which we have discussed below.

2. Object Constructor :

Another way to create objects in JavaScript involves the “Object” constructor. The Object Constructor creates an object wrapper for the given value, which used in conjunction with the “new” keyword allows us to initialize new objects.

For example:

```
const school = new Object();
school.name = 'Vivekanada school';
school.location = 'Delhi';
school.established = 1971;

school.displayInfo = function(){
    Console.log(`${school.name} was established in
${school.established} at ${school.location}`);
}

school.displayInfo();
```

3. Constructors :

Alternatively, you can create an object with these two steps:

- Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter.
- Create an instance of the object with new.

For example:

```
function Vehicle(name, maker) {
  this.name = name;
  this.maker = maker;
}

let car1 = new Vehicle('Fiesta', 'Ford');
let car2 = new Vehicle('Santa Fe', 'Hyundai')

Console.log(car1.name);     // Output: Fiesta
Console.log(car2.name);     // Output: Santa Fe
```

Notice the usage of the “new” keyword before the function Vehicle. Using the “new” keyword in this manner before any function turns it into a constructor.

Notice the usage of the “ new” keyword before the function Vehicle. Using the “new” keyword in this manner before any function turns it into a constructor. What the “new Vehicle()” actually does is :

- It creates a new object and sets the constructor property of the object to Vehicle (it is important to note that this property is a special default property that is not enumerable and cannot be changed by setting a “constructor: someFunction” property manually).
- Then, it sets up the object to work with the *Vehicle* function’s prototype object (each function in JavaScript gets a prototype object, which is initially just an empty object but can be modified.

The object, when instantiated inherits all properties from its constructor's prototype object).

- Then calls Vehicle() in the context of the new object, which means that when the "this" keyword is encountered in the constructor(vehicle()), it refers to the new object that was created in the first step.

- Once this is finished, the newly created object is returned to car1 and car2 (in the above example).

Inside classes, there can be special methods named *constructor()*.

4. Prototypes :

Another way to create objects involves using prototypes. Every JavaScript function has a prototype object property by default (which is empty by default). Methods or properties may be attached to this property. A detailed description of prototypes is beyond the scope of this introduction to objects.

However you may familiarize yourself with the basic syntax used as below:

```
var obj = Object.create(prototype_object,
propertiesObject)
// the second argument (propertiesObject) is optional
```

For example:

```
var footballers = {
   position: "Striker"
}

var footballer1 = Object.create(footballers);
Console.log(footballer1.position); // Output:Striker
```

Accessing Object Members

Object members (properties or methods) can be accessed using :

- **Dot Notation** : `(objectName.memberName)`
- **Bracket Notation** : `objectName["memberName"]`
- **Index**

Unlike the Dot Notation, the bracket keyword works with any string combination, including, but not limited to multi-word strings.

For example:

```
somePerson.first name    // invalid
somePerson["first name"] // valid
```

Unlike the dot notation, the bracket notation can also contain names which are results of any expressions variables whose values are computed at run-time.

For example:

```
var key = "first name" somePerson[key] = "Name Surname"
```

Similar operations are not possible while using the dot notation.

Iterating Over All Keys of an Object

- **`For…in` loops :**

  This method traverses all enumerable properties of an object and its prototype chain

- **`Object.keys(o)` :**

  This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object o.

- **`Object.getOwnPropertyNames(o)` :**

    This method returns an array containing all own properties' names (enumerable or not) of an *object o*.

To iterate over all existing enumerable keys of an object, we may use the **for...in** construct. It is worth noting that this allows us to access only those properties of an object which are enumerable (recall that enumerable is one of the four attributes of data properties).

For instance, properties inherited from the `Object.prototype` are not enumerable. But, enumerable properties inherited from somewhere can also be accessed using the **for...in** construct.

```
var person = {
   gender : "male"
}
var person1 = Object.create(person);
person1.name = "Adam";
person1.age = 45;
person1.nationality = "Australian";

for (var key in person1) {
// Output : name, age, nationality and gender
   Console.log(key);
}
```

Deleting Properties

To delete a property of an object we can make use of the **delete operator**.

For example:

```
var obj1 = {
   propfirst : "Name"
}

Console.log(obj1.propfirst);     // Output : Name
delete obj1.propfirst

Console.log(obj1.propfirst);     // Output : undefined
```

It is important to **note** that we can not delete inherited properties or non-configurable properties in this manner.

Defining methods

A method is a function associated with an object, or simply put, a method is a property of an object that is a function. Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object. See also *method definitions* for more details.

For example:

```
objectName.methodname = functionName;

var myObj = {
   myMethod: function(params) {
       // ...do something
   }
   // OR THIS WORKS TOO
   myOtherMethod(params) {
       // ...do something else
   }
};
```

where *objectName* is an existing object, *methodname* is the name you are assigning to the method, and *functionName* is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

Using this for Object References

JavaScript has a special keyword, **this**, that you can use within a method to refer to the current object.

For example:

```
const Manager = {
   name: "John",
   age: 27,
   job: "Software Engineer"
}
const Intern= {
   name: "Ben",
   age: 21,
   job: "Software Engineer Intern"
}
function sayHi() {
   Console.log('Hello, my name is', this.name)
}

// add sayHi function to both objects
Manager.sayHi = sayHi;
Intern.sayHi = sayHi;

Manager.sayHi()      // Hello, my name is John'
Intern.sayHi()       // Hello, my name is Ben'
```

In the above example, suppose you have 2 objects, Manager and Intern. Each object have their own name, age and job. In the function *sayHi()*, notice there is *this.name*. When added to the 2 objects they can be called and returns the 'Hello, My name is' then adds the name value from that specific object.

Defining Getters and Setters

A **getter** is a method that gets the value of a specific property. A **setter** is a method that sets the value of a specific property. You can define getters and setters on any predefined core object or user-defined object that supports the addition of new properties.

In principle, getters and setters can be either :

- defined using object initializers, or
- added later to any object at any time using a getter or setter adding method.

When defining getters and setters using **object initializers,** all you need to do is to prefix a getter method with get and a setter method with set. Of course, the getter method must not expect a parameter, while the setter method expects exactly one parameter (the new value to set).

For example:

```
var o = {
 a: 7,
 get b() { return this.a + 1; },
 set c(x) { this.a = x / 2; }
};
```

Getters and setters can also be added to an object at any time after creation using the *Object.defineProperties* method. This method's first parameter is the object on which you want to define the getter or setter. The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions.

Here's an example that defines the same getter and setter used in the previous example:

```
var o = { a: 0 };

Object.defineProperties(o, {
   'b': { get: function() { return this.a + 1; } },
   'c': { set: function(x) { this.a = x / 2; } }
});

// Runs the setter, which assigns 10 / 2 (5) to the 'a'
property
o.c = 10;

// Runs the getter, which yields a + 1 or 6
Console.log(o.b);
```

Which of the two forms to choose depends on your programming style and task at hand. If you already go for the object initializer when defining a prototype you will probably most of the time choose the first form. This form is more compact and natural.

However, if you need to add getters and setters later — because you did not write the prototype or particular object — then the second form is the only possible form. The second form probably best represents the dynamic nature of JavaScript — but it can make the code hard to read and understand.

Inheritance

All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.

- **Object created with Syntax Constructs :**

```
var o = {
   a: 1
}

// The newly created object o has Object.prototype as
its [[Prototype]]
// o has no own property named 'hasOwnProperty'
// hasOwnProperty is an own property of
Object.prototype.
// So o inherits hasOwnProperty from Object.prototype
// Object.prototype has null as its prototype.
// o ---> Object.prototype ---> null

var b = ['yo', 'whadup', '?'];

// Arrays inherit from Array.prototype
// (which has methods indexOf, forEach, etc.)
// The prototype chain looks like:
// b ---> Array.prototype ---> Object.prototype --->
null

function f() {
   return 2;
}

// Functions inherit from Function.prototype
// (which has methods call, bind, etc.)
// f ---> Function.prototype ---> Object.prototype
---> null
```

- **Object created with Constructors :**

  A "constructor" in JavaScript is "just" a function that happens to be called with the new operator.

```
function Graph() {
    this.vertices = [];
    this.edges = [];
}

Graph.prototype = {
    addVertex: function(v) {
        this.vertices.push(v);
    }
}

var g = new Graph();
// g is an object with its own properties 'vertices'
and 'edges'.
// g.[[Prototype]] is the value of Graph.prototype
when new Graph() is executed.
```

- **Object created with `Object.create()` :**

  ECMAScript 5 introduced a new method: *Object.create()*. Calling this method creates a new object and the prototype of this object is the first argument of the function:

```
var a = {
    a: 1          // a ---> Object.prototype ---> null
}

var b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
Console.log(b.a);    // 1 (inherited)
```

Continued on the next page…

```
var c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

var d = Object.create(null);  // d ---> null
Console.log(d.hasOwnProperty);
// undefined, because d doesn't inherit from
Object.prototype
```

Classes

Concept of class is not yet implemented, rather we use functions to create class, and 'new' to create objects.

In the previous chapter, we introduced the **new** keyword and showed how it is related to constructor functions.

```
function Person ( name, age ) {
 this.name = name;
 this.age = age;
}
const elle = new Person( 'Eloise', 96 );
```

CHAPTER 6 :

ERROR HANDLING

TRY/CATCH/THROW

```
var x = document.getElementById("mynum").value;

// get input value
try {
   if(x == "")   throw "empty";          // error cases
   if(isNaN(x)) throw "not a number";
   x = Number(x);
   if(x > 10)    throw "too high";
}catch(err) {                            // if there's an
error
     Console.error("Input is " + err); // output error
}

finally {
     Console.info("</br />Done");
     // executed regardless of the try/catch result
}
```

JSON

Parse JSON String

```
var str = '{"names":[' +
       '{"first":"Hakuna","lastN":"Matata" },' +
      '{"first":"Jane","lastN":"Doe" },' +
      '{"first":"Air","last":"Jordan" }]}';
// creates JSON string
var obj = JSON.parse(str);                  // parse
Console.println(obj.names[1].first);        // access
```

Convert to JSON String

```
var myObj = { "name":"Jane", "age":18, "city":"Chicago" };
var myJSON = JSON.stringify(myObj);         // storing data
// you can parse above json string to create copy of myObj

var obj = JSON.parse(myJSON);               // retrieving
data
Console.println(obj.name);
```

PART II :

GUI PROGRAMMING

Chapter 1: **Introduction**

Chapter 2: **Creating GUI**

Chapter 3: **Widgets in SimPhy**

Chapter 4: **Event Handling**

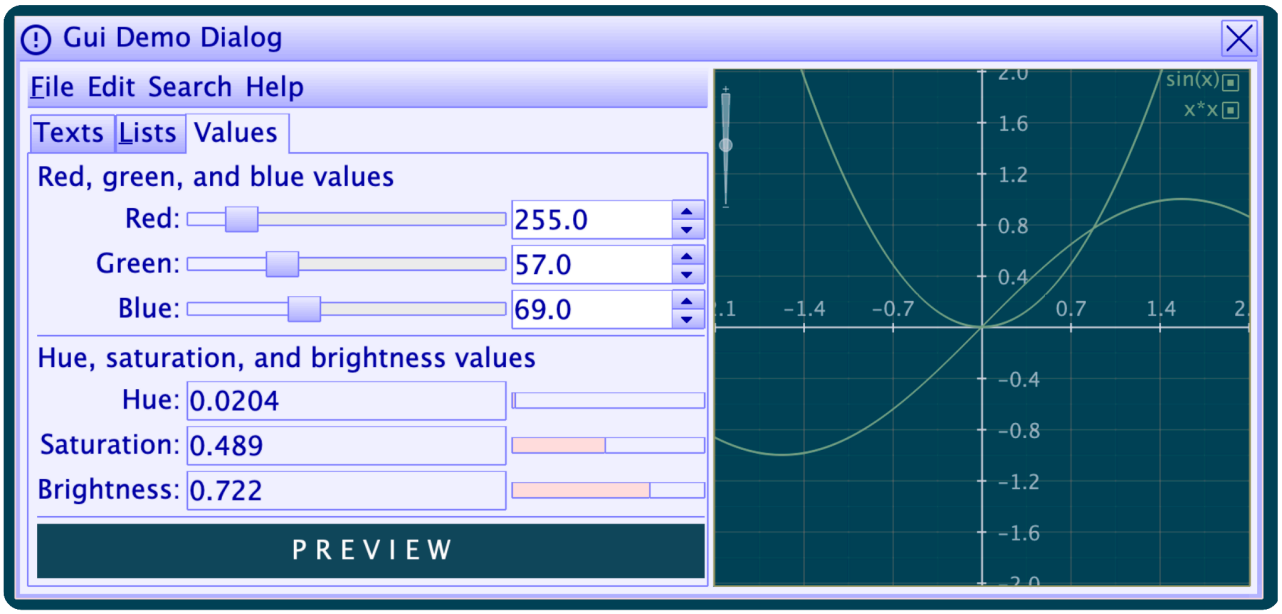
Chapter 5: **Getting your hands Dirty**

Chapter 6: **Widgets Appendix**

CHAPTER 1 :

INTRODUCTION OF WIDGETS

SimPhy has its own set of GUI widgets to make simulations interactive. It has almost all types of widgets needed to create full fledged GUI application as below.



It offers you a wide selection of GUI controls:

Container Widgets

- **Panel**
- **Dialog**
- **SplitPane**
- **TabbedPane**
- **Desktop**

Action Widgets

- **TextField**
- **TextArea**
- **Button**
- **Label**
- **CheckBox**
- **RadioButton**
- **DropDown**
- **Menu**
- **Slider**
- **Spinner**
- **ProgressBar**

Data Display Widgets

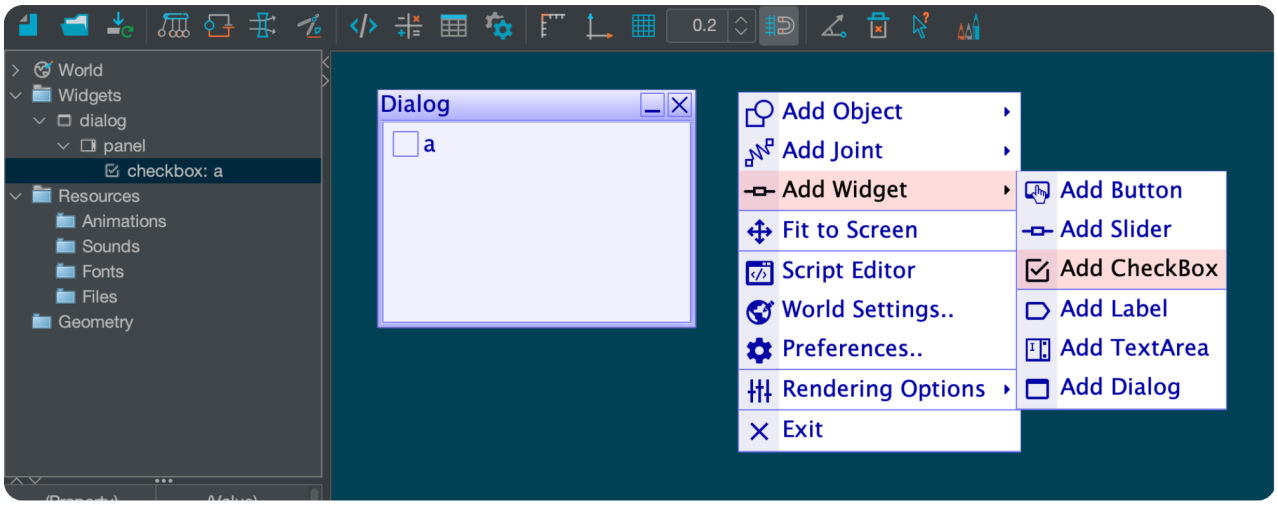
- **List**
- **Table**
- **TreeView**

CHAPTER 2 :

CREATING GUI

Beginner's Way

You can create GUI in WYSWYG way as in **NetBeans** or other *GUI* creation tools using frontend editor.



Properties can be edited by clicking over widget (or selecting widget in tree view) and then changing them on the fly in property table on the left side of your screen.

Programmer's Way

You can create GUI using script. All GUI related tasks can be done using keyword `Widgets`.

- ### Creating Widgets

  Widgets can be created by calling `Widgets.createXXX()` function.

- **Adding Widgets**

  Widgets so created are actually added to GUI after calling `Widgets.addWidget()`.

- **Setting Widget Properties**

  Properties of widgets can be retrieved by calling `Widget.getXXX()` or `Widget.isXXXX()` and changed by calling `Widget.setXXX()`.

- **Removing Widgets**

  Widgets of a particular type can be removed by calling `Widgets.removeWidget(widget Name)` function. Moreover, you can also remove all existing widgets using `Widgets.removeAll()` function.

For example:

```
var d=Widgets.createDialog("Hello World");
//create dialog with title

d.setResizable(true);
//allow user to resize it

d.setGaps(4);
//set 4 pixel gap between its components

d.setPreferredSize(140, 100);
//set size in pixels

d.add(Widgets.createButton("OK"));
//add button with text "OK" to it

d.add(Widgets.createButton("Cancel"));
//add button with text "cancel" to it

Widgets.addWidget(d);
//add dialog to gui
```

The code creates following output :

Designer's Way

You can define the GUI in an XML configuration which is very much similar to that in html, to keep view separate from controllers.

GUI is made in HTML. It's rather simple, and it's XML syntax.

`A small example is this:`

```
<dialog>
<label text="Enter your name here:" />
<textfield name = "yourname" />
<button text="Send" action = "send(yourname.text)" />
</dialog>
```

A prompt, an input field, and a button.

In SimPhy you'd make a configuration file like this:

```
<panel>
<label text="Enter your name here:" />
<textfield name = "yourname" />
<button text="Send" action = "send (yourname.text)" />
</panel>
```

Note: You have to call `Widgets.parse(xmlString)` in script to parse GUI.

For example:

```
var xml="<panel> + <label text='Enter your name here:'
foreground='White'/> + <textfield name='yourname' /> +
<button text='Send' action='send(yourname.text)' /> +
</panel>";

var pnl=Widgets.parse(xml);
Widgets.addWidget(pnl);
```

Output:

From my point of view this is the most preferred way of creating GUI, which keeps code clutter free and makes it readable.

CHAPTER 3 :

WIDGETS IN SIMPHY

Widget	XML Element	Data is retrieved by...
TextField	`<textfield text="TextField" columns="10" />`	getText()
PasswordField	`<passwordfield text="secret value" />`	getText()
TextArea	`<textarea text="TextArea" wrap="true" columns="40" rows="2" />`	getText()
Button	`<button text="Button" icon="image" alignment="left" tooltip="ToolTip" />` `<button text="www.SimPhy.com" type="link" action="goHome" />`	isSelected()
Label	`<label text="Label" icon="image" alignment = "center" />`	getText()
CheckBox	`<checkbox text="CheckBox" icon="image" selected="true"/>`	getText()
RadioButton	`<checkbox text="RadioButton-on" group="group" selected="true"/>` `<checkbox text="RadioButton" group="group"/>`	isSelected()

Drop Down with Input Field	`<combobox text="ComboBox">`
	`<choice text="Choice" icon="image" />`
	`<choice text="Disabled" enabled="false" />`
	`</combobox>`
		isSelected()

List	`<list selection="multiple">`
	`<item text="List" selected="true" />`
	`<item text="Item"/>`
	`<item text="Disabled" enabled="false" />`
		getSelectedIndex()

	`</list>`	

Table	`<table selection="multiple">`
	`<header>`
	`<column text="A" width="24" />`
	`<column text="B" />`
	`</header>`
	`<row selected="true">`
	`<cell text="a1" />`
	`<cell text="b1" enabled="false" />`
	`</row>`
	`<row>`
	`<cell text="a2" />`
	`<cell text="b2" />`
	`</row>`
	`</table>`
		`getValueAt(row,col)`

Tree	`<tree selection="multiple">`
	`<node text="Node A" icon="image">`
	`<node text="Node B" icon="image" selected = "true" />`
	`<node text="Node C" icon="image" />`
	`</node>`
	`<node text="Node D" expanded="false">`
	`<node text="Node E" icon="image" />`
	`</node>`
	`</tree>`
		`getSelectedIndex()`
		`or`
		`getSelectedNode()`

TabbedPane	`<tabbedpane placement="left" selected="1" action="tabchanged"> <tab text="One" icon = "image.gif">`
	`<textarea text="One" />`
	`</tab>`
	`<tab text="Two" alignment="right">`
	`<textarea text="Two" />`
	`</tab>`
	`<tab text="Three" enabled="false">`
	`<textarea text="Three" />`
	`</tab>`
	`</tabbedpane>`
		`getSelectedIndex()`

SplitPane	`<splitpane orientation="vertical" divider="24">`
	`<textarea text="Top"/>`
	`<textarea text="Bottom" />`
	`</splitpane>`	

Panel	`<panel columns="3" gap="4" top="4" left="4" bottom="4" right="4"> <textfield text="North" colspan="3" />` `<label text="East" />`	

	<textarea text="Center" weightx="1" weighty="1" /> <label text="West" /> <textfield text="South" colspan="3" /> </panel>	
Dialog	<dialog columns="3" gap="4" top="4" left="4" bottom="4" right="4"> <textfield text="North" colspan="3" /> <label text="East" /> <textarea text="Center" weightx="1" weighty="1" /> <label text="West" /> <textfield text="South" colspan="3" /> </dialog>	
SpinBox	<spinbox minimum="25" maximum="75" value="50" />	getValue()
ProgressBar	<progressbar minimum="25" maximum="75" value = "50" orientation="vertical" />	getValue()
Slider	<slider minimum="25" maximum="75" value="50" orientation="vertical" />	getValue()

Basic Properties of Widgets

Name	Type	Default	Description
name	string		Identifies the component.
enabled	boolean	true	Enables or disables this component. A disabled component is painted gray, and can't respond to user mouse or keyboard input, gain the keyboard focus, and generate events.
visible	boolean	true	An invisible component doesn't take place in parent's layout.
tooltip	string		The text pops up when the mouse lingers inside the component.

font	font		Custom font for texts.
foreground	color		Custom foreground (text) color to use, instead of the default text color.
background	color		Custom background color to use, instead of the default background color. Note: for components with gradients (buttons, menubars, etc...) this means that the background will be filled with solid color, and not a custom gradient
width	integer	0	Fixed preferred width of the component irrespectively of its content. If it is 0, the component will be asked for the preferred width.
height	integer	0	Preferred height of the component, or 0.
colspan	integer	1	Specifies the number of cells in a row in the component's display area.
rowspan	integer	1	Specifies the number of cells in a column occupied by the component.
weightx	integer	0	Used to determine how to distribute horizontal space among grid cells when more space is available for its parent component than required.
weighty	integer	0	The extra vertical space is distributed to each cell height in proportion to its weight. Default value is 0. The row contains 0 weighted components remains the calculated preferred size.
halign	choice	fill	Horizontal alignment of the component when more space available in the cell. Possible values are: fill, center, left, and right.
valign	choice	fill	Vertical alignment in the cell. Possible values are: fill, center, top, and bottom.
property	property		Binds an arbitrary key/value client property (or properties).

Properties Common to Widgets Containing Text and Icon

For the widgets containing text and icon such as button, label, dialog, list-item, menu-item, etc. there are few properties to set icon, text and their alignment.

Name	Type	Default	Description
`text`	`string`		The text string that the button displays.
`icon`	`icon`		The icon image that the button displays.
`alignment`	`choice`	`center`	The alignment of the text and image similar to label. Possible values are: center, left, and right. The default value is center. Icons are always displayed to the left of text (if any).
`mnemonic`	`integer`	`-1`	Specifies the index of underlined char and a key combination (Alt + the char) which invokes the (not necessarily focused, but enabled and visible) button's action listener.
`type`	`choice`	`normal`	Possible values are: normal, default, cancel, and link. The default value is normal. Default, and cancel values are for dialog control. Link changes the appearance of button so that it resembles HTML link.

Properties Types and Allowed Values :

- `integer`

  A signed decimal integer.

```
<dialog width="200" height="100"/>
```

- `color`

  Color value is defined by a string that represents a CSS color string in any of following format.

  1. Common color name `"red"`, `"green"`.

2. Hexadecimal representation of color such as `#FF0096 or 0xFF0096.`

3. Color functions `"red", "0xFF0096", "#FF0096", "rgb(255,0,0)", "hsl(180, 50%, 50%)", "rgba(255,255,120,1)".`

```
<label text="I am red" foreground ="red" background = "rgb(255,255,255)"/>
```

- **font**

  Font property is defined as font name already created in SimPhy. There are 4 already loaded fonts in SimPhy `default-normal, default-bold, default-large and default-small`. New fonts can be loaded/created from treeview font node under resources.

```
<label text="I am red" font ="default-bold" />
```

  If font with the name does not exist it sets font as `default-normal.`

- **image**

  The image specified by name corresponding to images/animations already loaded in SimPhy. New image can be imported from treeview animations node under resources. Predefined icons (available in property table's icon row) can also be used as image.

```
<button text="Apply" icon ="body_box" />
```

- **choice**

  Unlike string, the choice parameter's possible values are limited, e.g. only the left, center, and right values are legal for content alignment in a label.

Accessing Widgets in Scripts

Most often you pass either the widgets or their properties in the method call defined in the configuration file. A SimPhy program may however, also locate any widget if its name is known.

Widgets can be accessed by calling `Widgets.getWidget(widget name)` method in script, the resulting object has several getters and setters methods to change properties;

In the example shown below, we pass the textField widget as the second parameter, but we might leave it out and let the action and perform methods locate the widget directly.

```
function perform(item) {
   var s = item.getText();
   // find the textField widget
   var tf = widgets.getTextField("textField");
   tf.setText("perform, item text = " + s);
}
```

Normally you should specify the widgets as parameters to avoid making the widget name-binding between the program and the configuration file. When many widgets are involved in an action however, it's nice to avoid long parameter lists, and let the code look up the widgets needed.

Changing Properties of widget

1. Using Key Value Pairs

To get and set Properties of widget there are 2 dedicated methods on widgets :

- **`getProperty(key)`**
- **`setProperty(key,value)`**

where key is property name (String) and value is property value (can be text, Color, int, double etc based on property)

Note: Key and value must be valid for the widget, else an error will be thrown.

For example:

```
var p=Widgets.getButton("buttonOK");
p.setProperty("foreground", new Color(1.0,0.0,0.0,0.0));
p.setProperty("action", "onClick(this)");
```

2. Using Dedicated Getters and Setters

There are specific getters and setters for each allowed property of widget to avoid remembering allowed properties and to allow code completion, resulting faster and error free code.

For example:

List widget contains `getSelectedIndex()`, `addItem()`, `removeItem(index)`, etc. methods to get currently selected item location, adding new item to list and remove item at specific position.

Note: SimPHY scripting language is basically JavaScript which is not a strict type language so it is not always possible for editor to find all methods for the widget (especially those which are passed as parameters). To deal with this problem widgets has a few methods to cast widgets to their own type.

For example, if you already know that variable x is of type textfield then you can re-declare x as

```
var x=Widgets.asTextField(x);
```

Now on pressing "." after x, you can access all properties/methods of textfield widget.

Adding, removing and disabling widgets

- **Creating Widgets**

  Widgets can be created by calling `Widgets.createXXX()` function.

- **Adding Widgets**

  Widgets so created are actually added to GUI after calling `Widgets.addWidget()`.

- **Setting Widget Properties**

  Properties of widgets can be retrieved by calling `Widget.getXXX()` or `Widget.isXXXX()` and changed by calling `Widget.setXXX()`.

- **Assigning Methods to Widgets**

  Common methods can be assigned to widgets by calling `setonAction(methodtext)` ,`setOnPerform(methodText)`, etc. on widget.

- **Removing Widgets**

  You may also remove a widget programmatically. In order to do so you need to have a reference to the widget. If you know its position in the GUI you may code like this: `widgets.removeWidget(widget)`

For example:

```
var d=Widgets.createDialog("Hello World");
//create dialog with title
d.setName('dlgTest');
d.setResizable(true);
//allow user to resize it
d.setGaps(4);
//set 4 pixel gap between its components
d.setPreferredSize(140, 100);
//set size in pixels
d.add(Widgets.createButton("OK"));
//add button with text "OK" to it

var btnCalcel= Widgets.createButton("Cancel");
btnCalcel.setOnAction("removeDialog(dlgTest)");
//pass dialog as parameter

d.add(btnCalcel);
//add button with text "cancel" to it

Widgets.addWidget(d);
//add dialog to gui

function removeDialog (dlg){
   Widgets.remove(dlg);
}
```

The code creates following output, on clicking cancel it removes dialog from GUI:

CHAPTER 4 :

EVENT HANDLING

As mentioned previously, you may link events defined in the configuration file to global methods/functions in script.

Here's a list of the events available for all the common widgets:

event								
widget	focuslost	focusgained	action	insert	remove	caret	perform	expand collapse
label	x	x						
button	x	x	x					
checkbox	x	x	x					
togglebutton	x	x	x					
combobox	x	x	x					
textfield	x	x	x	x	x	x	x	
textarea	x	x	x	x	x	x		
spinbox	x	x	x					
slider	x	x	x					
list	x	x	x				x	
table	x	x	x				x	
tree	x	x	x				x	x
menuitem	x	x	x					

Here's a short description of the events.

Event	Description
`init`	A method to invoke only once when the loading of the XML resource (including this component) is finished.
`focuslost`	Invoked when a component loses the keyboard focus, thus it is no longer the focus owner.
`focusgained`	Invoked when a component gains the keyboard focus, thus it is now the focus owner.
`action`	Depends on the widget. Typically invoked after a mouse click (buttons, boxes etc.)
`insert`	Gives notification that there was an insert into the text (and possibly a portion has been removed too).
`remove`	Gives notification that a portion of the text has been removed.
`caret`	To track whenever the caret position has been changed.
`perform`	Textfield: Invokes the given method if enter was pressed in an editable and enabled textfield. Others: Calls the method whenever a double-click event occurs.
`expand` `collapse`	Expand: Called when the tree expands a node. Collapse: Called whenever a node in the tree has been collapsed.

Defining method

Method is defined as a String property containing method name and parameters in format `methodName(param1, param2,..)`.

The event description may contain parameters (in brackets, separated by comma or whitespace characters) as follows:

- **this** = The source widget (the object on which the event occurred)
- **widget name** = Another widget in GUI identified by the given name

- **item** = The component part on which the event occurred, valid for list item, tree node, table row, combobox choice, and tabbedpane tab.
- **this/name/item.attribute** = component's or item's attribute value, defined by the this, widget name, or item string, dot, and the attribute key. Ex. onSelectionChange(this, item.text, this.value).
- **constant string** = The source widget (the object on which the event occurred).
- **constant value** = The source widget (the object on which the event occurred).
- **constant number** = Long numbers end with 'L' character (it ignors case), floats width 'F', doubles have to include a dot character, otherwise it is expected as integer. *

Method in XML Configuration

Method is specified as one of the attribute of widget with name same as method name as in table under events, and the value of attribute (inside quotes) is the method text in format described above.

Look at some examples to get an idea...

```
<button action="one" />
<button action="two(this, label1)" />
<button action="three(this.text, this.icon, label1.visible,
label1.colspan)" />
<list action="four(item, item.text, item.icon)">
<item text="File" icon="file.gif" />
</list>
<button text="Button" mnemonic="0" action="five('string',
12, 1234567890L, 12.34, 45.6F)" />
```

Now there must exist these associated functions defined in script at global level :

```
function one() {}
function two( button, label) {}
function three(buttontext, buttonicon, labelvisible,
labelcols) {}
function four(item, itemtext, itemicon) {}
function five(s, i, l, d, f) {}
```

Setting Event Method in Script

Methods for the events can also be described in script by using corresponding setter function of widget object.

For example to set method for 'action' of button named 'buttonOK' we will be using following code:

```
var btnOk=Widgets.getButton("buttonOK");
btnOK.setOnAction("onOkClicked()");

function onOKClicked(){
   Console.info("OK Button is Clicked");
}
```

CHAPTER 5 :

GETTING YOUR HANDS DIRTY

CREATING FIRST GUI APPLICATION IN SIMPHY

So it's a calculator, at least a beginning, but for simplicity I've only added the digits, a period character, a clear button, and the display area. To make the correct configuration file you should realize these facts:

- Always start the configuration file with a "dialog" element.
- Define a table-like area in your panel consisting of 3 columns and 5 rows to hold the widgets.
- Define all widgets inside the dialog element.
- If a widget (like the display area) occupy more than one cell in the table then specify the number of columns (or rows) using the rowspan and colspan property.

A proper configuration file looks like this :

```
<dialog text="Calc" columns="3" gap="4" top="4" left="4"
right="4" bottom="4">
 <button text = "7" name="7" action="display(this, result)" />
 <button text = "8" name="8" action="display(this, result)" />
 <button text = "9" name="9" action="display(this, result)" />
 <button text = "4" name="4" action="display(this, result)" />
 <button text = "5" name="5" action="display(this, result)" />
 <button text = "6" name="6" action="display(this, result)" />
 <button text = "1" name="1" action="display(this, result)" />
 <button text = "2" name="2" action="display(this, result)" />
 <button text = "3" name="3" action="display(this, result)" />
 <button text = "C" name="C" action="clear(result)" />
 <button text = "0" name="0" action="display(this, result)" />
 <button text = "." name="." action="display(this, result)" />
 <textfield name="result" editable="false" colspan="3" />
</dialog>
```

The Panel Element

Since the panel element specifies `columns = "3"` we'll get a layout table with 3 columns and as many rows as we need. The widgets are now laid out one by one starting a new row every time a row is filled. The last widget--the textfield--uses `colspan = "3"` to fill the whole row.

The properties gap, top, left, right, bottom sets the space around each cell. If you're familiar with HTML tables, then most of the table layout will be familiar to you.

The Button Elements

The text property gives the text on each button. The spaces around the characters are only used to make the button somewhat wider. The 'name' property can be given to any component, but is often not necessary. We need it to identify which button was pressed. The action property is the name and signature of a Java method in our program. We'll return to that shortly.

The TextField Element

A `textfield` is normally an input field, but since we only use it for display, we have specified `editable="false"`. The `colspan` property has been mentioned above.

Linking to Java Event Methods

Many widgets react on "events", an example is clicking a button. Since events are defined in the configuration file, but should be handled by a Java program, SimPhy needs to link these together. It's simply done by allowing you to enter the method call in the configuration file. This means, that

```
action="display(this, result)"
```

activates a **display** method with two parameters. The first is the widget object itself, the second is the display area widget.

You may read more about events on Events.

To complete the Calculator, we therefore need two global methods in script: `display` and `clear`:

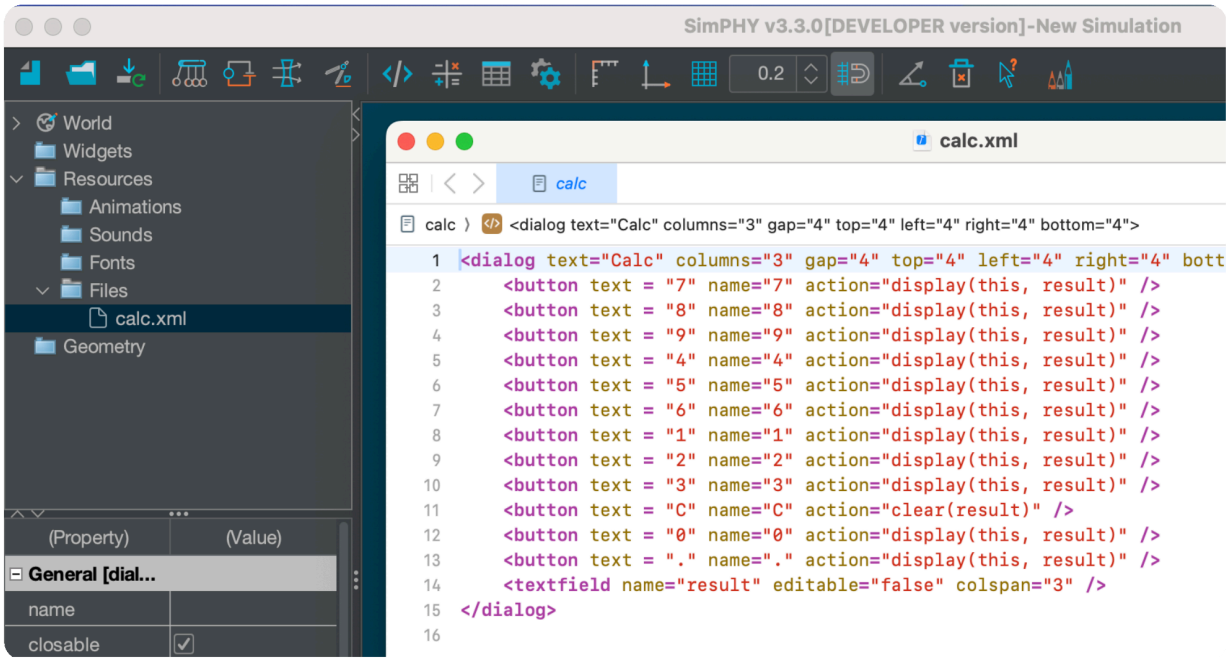
```
var i=0;
function display(button, result) {
   i++;
   var s = result.getText();
   var t = button.getName();
   if(i==1)    result.setText(t);
   else   result.setText(s+t);
}
function clear(result) {
   result.setText("");
}
```

Let's create the above, so called, calculator step by step!

Step 1 : Create XML File

Writing large strings as variable in code is tedius and also doesn't separate view from controller. So we will create a file with this content in simPHY resources.

- Right click on Resources folder in Tree View of SimPhy.
- Add or create File in simPHY named 'calc.xml'.
- Enter the above content in the file and update it.



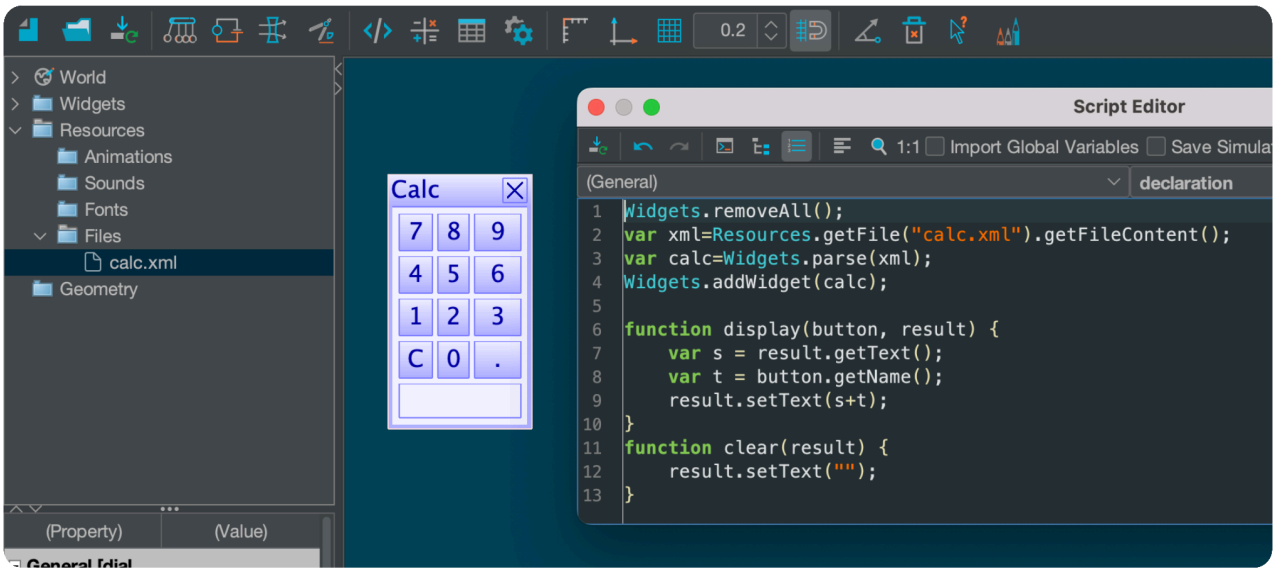
Step 2 : Create Methods Used in XML

- Open script editor
- create method display(button, result)
- create method clear(result)

Step 3 : Read XML, Parse and Add to GUI

- read content of the file 'calc.xml' by calling `resources.getFile("calc.xml").getFileContent()`
- parse XML to get root widget by calling `widgets.parse(xml)`
- add widget to GUI by calling `widgets.addWidget(root);`

The complete code should be as shown



Save script and run simulation to see calculator (which doesn't calculate anything !) in action. When you press a button result textfield is updated.

THE SECOND EXAMPLE

I always prefer to introduce my readers to some new technology by presenting many simple examples. Most often, you can deduce a lot from just a few examples.

Layout and Component Properties

Edit an XML file (called plusCalc.xml) which describes the hierarchy and the static attributes of the components. The root panel component contains 3 textfields, a label (+) and a button (=). For event handling, identify the textfields with names, and add method name (action listener) for the button. Import the XML file to simulation :

```
<panel gap="4" top="4" left="4">
     <textfield name="number1" columns="4" />
     <label text="+" />
     <textfield name="number2" columns="4" />
     <button text="=" action="calculate(number1.text,
     number2.text, result)" />
     <textfield name="result" editable="false" />
</panel>
```

Load XML to Simulation

Open Script Editor, and write following line of code.

```
var xml =
Resources.getFile("plusCalc.xml").getFileContent();
var root = Widgets.parse(xml);
Widgets.addWidget(root);
```

Save script and run simulation, you should get GUI similar to as below.

Event Handling

We want to display the sum of values entered in each textfield in the result textfield on clicking button. For this we will have to create a function *calculate* having 3 parameters corresponding to XML definition.

```
function calculate(number1, number2, result) {
   var x=parseInt(number1);
   var y=parseInt(number2);
   result.setText(x+y);
}
```

The calculate method has three parameters: the first two are of the type **attribute**, which gives us the text entered in the two textfields directly. The third parameter is of the type **widget name**. When the button is pressed, the sum is calculated. If we however, use the **insert** events on the two textfields, the sum will be calculated immediately when a digit is entered in any field:

```
<panel gap="4" top="4" left="4">
     <textfield name="number1" columns="4" insert =
     "calculate(this.text, number2.text, result)"/>
     <label text="+" />
     <textfield name="number2" columns="4" insert =
     "calculate(number1.text, this.text, result)"/>
     <button text="=" />
     <textfield name="result" editable="false" />
</panel>
```

Since the insert event is connected to the *number1* and *number2* fields, it's possible to replace their names with **this**.

Try changing XML file content and rerun simulation, you will see that the result is automatically updated on changing content of textfields.
(Beware of valid entries!)

THE THIRD EXAMPLE

Here's another example to show the use of the "item" parameter. It's a simple list with three items:

The configuration file is this (name it as list.xml):

```
<dialog columns="1" gap="4" top="4" left="4" bottom="4"
right="4">
     <textfield name="textField" width="170" height="20"/>
     <list name="listBox" selection = "multiple" action =
     "action(item, textField)" perform = "perform(item,
     textField)">
     <item name="DK" text="Denmark" />
     <item name="S" text="Sweden"/>
     <item name="N" text="Norway"/>
     </list>
</dialog>
```

A single click on an item invokes the action-method, and a double-click invokes the perform-method. These methods shall write the text of the selected item in the textfield at the top of the panel, and to show you how to change the properties of the item we set a new background color.

```
Widgets.removeAll();
var xml = Resources.getFile("list.xml").getFileContent()
var root = Widgets.parse(xml)
Widgets.addWidget(root);

function action(item, textField) {
   var s = item.getText();
   item.setBackground(new Color(0, 255, 0,255));
   item.setText("action, item text = " + s);
}
function perform(item, textField) {
   var s = item.getText();
   item.setBackground(new Color(0, 255, 0,255));
   item.setText("perform, item text = " + s);
}
```

FAQ : *How do i know the properties of widgets which can be used in XML*

Ans : *Create widget from front end and look at its properties in the property table.*

THE FOURTH EXAMPLE

Here's another example to show the use of different widgets to change the value of a number:

The XML code is written as below (“num.xml”).

```
<dialog columns="8" gap="20" top="4" left="4" bottom="4"
right="4">
     <textfield background="white" foreground="black"
     weightx="1" text="0" name="number" colspan="4" perform
     = "changeNum(this,spin,slide)" />
     <spinbox weightx="1" text="0" minimum="-50"
     maximum="50" name="number"
     action="changeVal(number,this,slide)" />
     <slider unit="1.0" weightx="1" text="value"
     halign="center" name="slide" minimum="-50"
     maximum="50" value="0" orientation = "horizontal"
     action= "changeVal1(number,spin,this)" />
     <button weightx="1" name="but1" text="decrement"
     action= "decrement(number,spin,slide)" />
     <button weightx="1" name="but2" text="increment"
     action= "increment(number,spin,slide)" />
</dialog>
```

A single click on an item invokes the action-method, and an enter invokes the perform-method in the textfield. Now, the below functions will help to change the value of textfield, spinbox and slider.

```
Widgets.removeAll();
var xml = Resources.getFile("num.xml").getFileContent();
var root = Widgets.parse(xml);
Widgets.addWidget(root);
```

Continued on the next page…

```
function increment(number,spin,slide){
   var num = parseInt(number.getText());
   num++;
   number.setText(num);
   spin.setText(num);
   slide.setValue(num);
}
function decrement(number,spin,slide){
   var num = parseInt(number.getText());
   num--;
   number.setText(num);
   spin.setText(num);
   slide.setValue(num);
}
function changeNum(number,spin,slide){
   var num = parseInt(number.getText());
   number.setText(num);
   spin.setText(num);
   slide.setValue(num);
}
function changeVal(number,spin,slide){
   var num = parseInt(spin.getText());
   number.setText(num);
   spin.setText(num);
   slide.setValue(num);
}
function changeVal1(number,spin,slide){
   var num = parseInt(slide.getValue());
   number.setText(num);
   spin.setText(num);
   slide.setValue(num);
}
```

CHAPTER 6 :

WIDGETS APPENDIX

Label

The following label displays a short text string and an image. Its content is centered both vertically and horizontally.

```
<label text="Label" icon="image.gif" alignment="center" />
```

- **Properties**

Name	Type	Default	Description
text	string		The text string that the label displays.
icon	icon		The icon image that the label displays
alignment	choice	left	The alignment of the label's content along the X axis, vertically is centered. Possible values are: left, center, and right. The default value, if not set, is left. The image position horizontally left, above, and right relative to the text.
mnemonic	integer	-1	Specifies the underlined char in the label's text.
for	component		Set the component this in labelling. The label will focus the component specified by its name when the mnemonic is activated.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

Button

The following button displays a short text string and an image. Its content is on the left and centered vertically. The second button resembles an HTML link, and when pressed invokes the method goHome.

```
<button text="Button" icon="image.gif" alignment="left"
tooltip="ToolTip" />
<button text="www.simphy.com" type="link" action="goHome" />
```

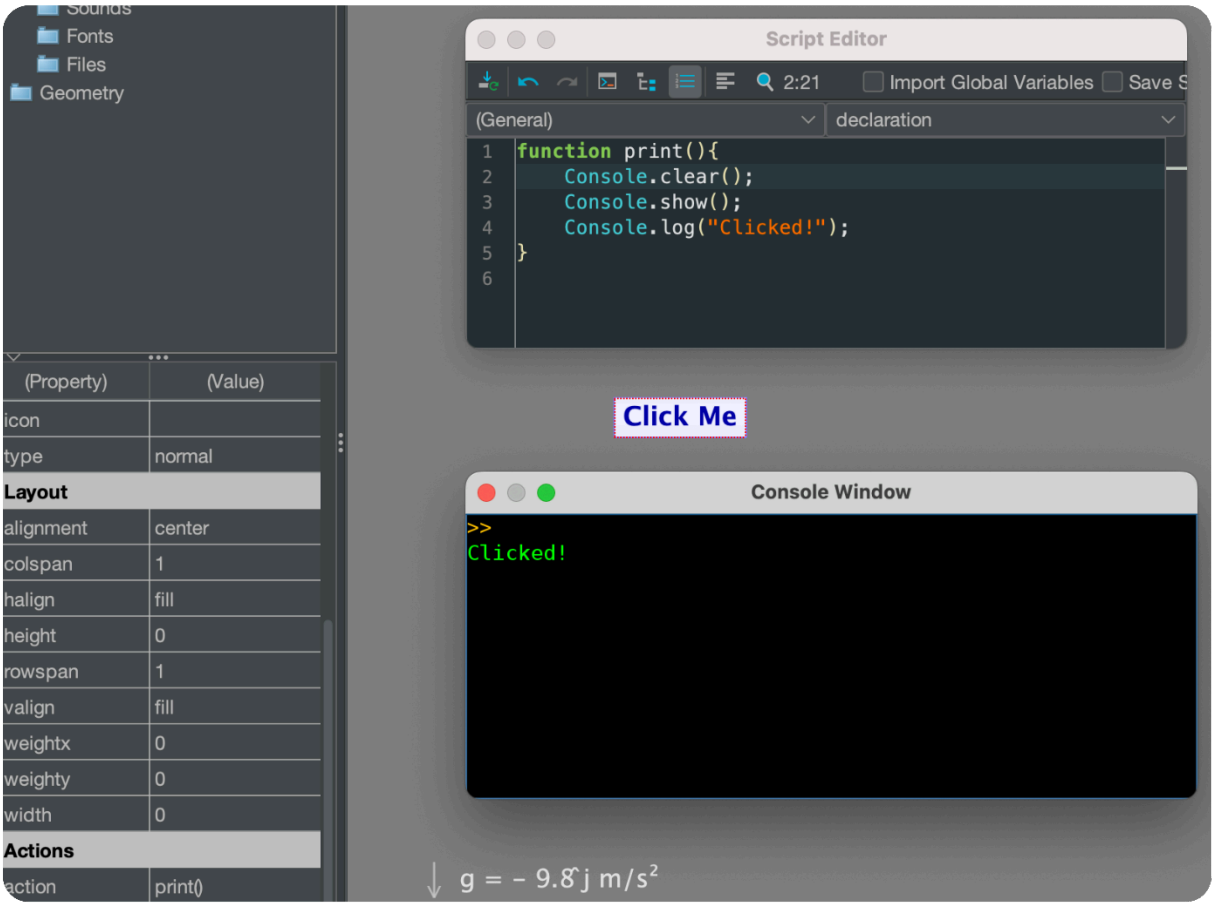
- **Properties**

Name	Type	Default	Description
text	string		The text string that the button displays.
icon	icon		The icon image that the button displays
alignment	choice	center	The alignment of the text and image similar to label. Possible values are: center, left, and right. The default value is center. Icons are always displayed to the left of text (if any).
mnemonic	integer	-1	Specifies the index of underlined char and key combination (Alt + the char) which invokes the (not necessarily focused, but enabled and visible) button's action listener.
type	choice	normal	Possible values are: normal, default, cancel, and link. The default value is normal. Default, and cancel values are for dialog control. Link changes the appearance of button so that it resembles HTML link.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to components.

- **Events**

Name	Description
`action`	Invokes the given method when the button is pressed by mouse or keyboard.



- **Keyboard Actions**

Operation	Action
Spacebar	Activates button
Tab, Shift-Tab	Navigate forward, backward

Check Box

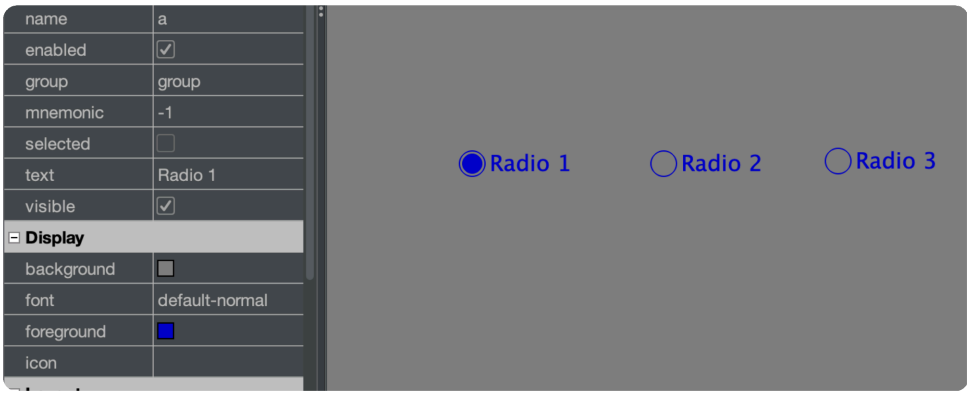
The first checkbox has a short text and an icon, its state is selected. The second one is a selected radio button, and the last is a deselected radio button. Both the radio buttons are identified by the *group* string id.

```
<checkbox text="CheckBox" icon="image.gif" selected="true"/>
<checkbox text="RadioButton-on" group="group" selected =
"true"/>
<checkbox text="RadioButton" group="group"/>
```

- **Properties**

Name	Type	Default	Description
`text`	`string`		The text string that the checkbox displays.
`icon`	`icon`		The icon image that the checkbox displays.
`alignment`	`choice`	`center`	The alignment of the text and image similar to label. Possible values are: left, center, and right.
`mnemonic`	`integer`	`1`	Specifies the index of underlined char and a key combination (Alt + char) which change the checkboxs's state and invokes the action listener.
`selected`	`boolean`	`false`	The state of the checkbox. True if the checkbox is selected, false if it's not.
`group`	`string`		Identifies the radio button group if not null. Only one radio button at a time can be selected. User can set on a radio button, the selected button of the group will be set off (the group members is searched only in the same parent).

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.



- **Listeners**

Name	Description
`action`	Invokes the given method when the checkbox state is changed by mouse or keyboard event.

- **Keyboard Actions**

Operation	Action
Spacebar	Selects or deselects
Tab, Shift-Tab	Navigate forward, backward

Toggle Button

An implementation of a two-state button, behaves as checkbox.

```
<togglebutton text="ToggleButton" icon="image.gif"
selected="true"/>
<togglebutton text="ToggleButton" group="group"/>
```

Text, icon, alignment, mnemonic, selected, button, and action parameters are similar to checkbox.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Keyboard Actions**

Operation	Action
Spacebar	Selects or deselects
Tab, Shift-Tab	Navigate forward, backward

Combo Box

ComboBox is a combination of a textfield and drop-down list. This example has a default value, two choices, and it is editable.

```
<combobox text="ComboBox">
<choice text="Choice" icon="image.gif" />
<choice text="Disabled" enabled="false" />
</combobox>
```

- **Properties**

Name	Type	Default	Description
`icon`	`icon`		The icon image that the combobox displays.
`selected`	`integer`		The index of the currently selected choice, value -1 indicates a custom edited value in an editable box.

Text, columns, editable, and alignment parameters are described at textfield. Name, enabled, visible, tooltip, property, icon, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Invokes the given method when there was change in the text or a new choice was selected.

Choice

The drop down list of a combobox contains choices. The popup combolist internally handles scrolling.

- **Properties**

Name	Type	Default	Description
`name`	`string`	`true`	Identifies the item.
`enabled`	`boolean`		Enables or disables the item. A disabled item is painted grey, and can't be selected by mouse or keyboard.
`text`	`string`		The text that the choice displays.
`icon`	`icon`	`left`	The icon image that the choice displays.
`alignment`	`choice`		The alignment of the text and image similar to label. Possible values are: left, center, and right.
`tooltip`	`string`		The text pops up when the mouse lingers inside this specified part of the component, otherwise the tooltip text of the component appears.
`font`	`font`		Custom font to use for text.
`foreground`	`color`		Custom foreground (text) color.
`background`	`color`		Custom background color.

Name	Type	Default	Description
`property`	`property`		Adds an arbitrary key/value client property (or properties) stored in a hashtable of the item.

- **Keyboard Actions**

Operation	Action
Spacebar, Down arrow	Post menu
Up arrow	Selects previous (or last) choice
Down arrow	Selects next (or first) choice
Home	Selects first choice
End	Selects last choice
Page Up	Selects choice one view up
Page Down	Selects choice one view down
Enter, Return, Spacebar	Activates selection, and retract menu

Text Field

TextField component allows the editing of a single line of text. This field has a simple text value and width of 10 characters.

```
<textfield text="TextField" columns="10" />
```

- **Properties**

Name	Type	Default	Description
`text`	`string`	`''`	The text contained in this field
`columns`	`integer`	`0`	The preferred width of the component is fixed (if 0) or calculated by the given value
`editable`	`boolean`	`true`	The specified boolean to indicate whether or not this textfield should be editable. A noneditable field is focusable, and selectable.
`alignment`	`choice`	`left`	The alignment of the text content along the X axis. Possible values are: left, center, and right. The default value, if not set, is left.
`start`	`integer`	`0`	Start index of the selection
`end`	`integer`	`0`	End index of the selection, same as the caret position.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to components.

- **Listeners**

Name	Description
`action`	Gives notification (invokes the given method) that there was an insert into the text or a portion of the text has been removed.
`insert`	Gives notification that there was an insert into the text (and possibly a portion has been removed too)
`remove`	Gives notification that a portion of the text has been removed.
`caret`	To track whenever the caret position has been changed.
`perform`	Invokes the given method if enter was pressed in an editable and enabled textfield.

- **Keyboard Actions**

Operation	Action
Right arrow	Moves insertion point one character to the right
Left arrow	Moves insertion point one character to the left
Ctrl-right arrow	Moves insertion point to beginning of next word
Ctrl-left arrow	Moves insertion point to beginning of previous word
Home	Moves insertion point to beginning of field
End	Moves insertion point to end of field
+Shift	Extends selection
Backspace	Deletes the previous character
Delete	Removes the following character
Ctrl-A, Ctrl-Slash	Selects all
Ctrl-Backslash	Deselects all
Ctrl-X	Cuts selected text into the clipboard
Ctrl-C	Copies selected text into the clipboard
Ctrl-V	Pastes the clipboard content
Tab, Shift-Tab	Navigate forward, backward

Password Field

PasswordField is similar to TextField component, but does not show the original characters.

```
<passwordfield text="secret value" />
```

- **Properties**

Text, columns, editable, alignment start, end, and action parameters are similar to textfield.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Keyboard Actions**

Operation	Action
Ctrl-X	Doesn't cut selected text into the clipboard.
Ctrl-C	Doesn't copy selected text into the clipboard.

Right arrow, Left arrow, Ctrl-right arrow, Ctrl-left arrow, Home, End, Backspace, Delete, Ctrl-A, Ctrl-Slash, Ctrl-Backslash, Ctrl-V, and Tab actions behave similar to textfield.

Text Area

TextArea is a multi-line area that displays plain text, optionally wrapped at word boundaries (whitespace), and internally handles scrolling.

```
<textarea text="TextArea" wrap="true" columns="40" rows="2" />
```

- **Properties**

Name	Type	Default	Description
`text`	`string`	`''`	The text contained in this textarea.
`columns`	`integer`	`0`	The number of visible letters in a column.

Name	Type	Default	Description
rows	integer	0	The number of visible rows for this textarea.
editable	boolean	true	The specified boolean to indicate whether or not this textarea should be editable. A noneditable area is focusable, and selectable.
wrap	boolean	false	If set to true the lines will be wrapped at word boundaries (whitespace) if they are too long to fit within the allocated width.
border	boolean	true	If set to false, no border will be drawn around the textarea, and its background will be set to the default background color, to visually "blend" with its container. You can still override this by setting a custom background color.
start	integer	0	Start index of the selection.
end	integer	0	End index of the selection, same as the caret position.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
action	Gives notification (invokes the given method) that there was an insert into the text or a portion of the text has been removed.
insert	Gives notification that there was an insert into the text (and possibly a portion has been removed too).
remove	Gives notification that a portion of the text has been removed.
caret	To track whenever the caret position has been changed.

- **Keyboard Actions**

Home	Moves to beginning of line
End	Moves to end of row or line
Ctrl-Home	Moves to beginning of data
Ctrl-End	Moves to end of data
Up arrow	Moves insertion point up one line
Down arrow	Moves insertion point down one line
Page Up	Moves up one view
Page Down	Moves down one view
Right arrow	Moves insertion point one character to the right
Left arrow	Moves insertion point one character to the left
Ctrl-right arrow	Moves insertion point to beginning of next word
Ctrl-left arrow	Moves insertion point to beginning of previous word
+Shift	Extends selection
Enter, Return	Inserts line-break
Backspace	Deletes the previous character
Delete	Removes the following character
Ctrl-A, Ctrl-Slash	Selects all
Ctrl-Backslash	Deselects all
Ctrl-X	Cuts selected text into the clipboard
Ctrl-X	Copies selected text into the clipboard
Ctrl-V	Pastes the clipboard content
Tab, Shift-Tab	Navigate forward, backward

Tabbed Pane

TabbedPane contains tabs and components. The first tab (has an index equal to 0) is associated with the first component. This example tabbed pane has 3 tab and 3 components (text areas), tabs are on the left, and the second component is visible.

```
<tabbedpane placement="left" selected="1"
action="tabchanged">
     <tab text="One" icon="image.gif">
     <textarea text="One" />
     </tab>
     <tab text="Two" alignment="right">
     <textarea text="Two" />
     </tab>
     <tab text="Three" enabled="false">
     <textarea text="Three" />
     </tab>
</tabbedpane>
```

- **Properties**

Name	Type	Default	Description
`placement`	`choice`	`top`	The placement for the tabs relative to the content. Possible values are: top, left, bottom, right, and stack. The default value, if not set, is top. The stack placement arranges tabs so that they resemble a sidebar.
`selected`	`integer`	`0`	The index of the currently selected tab, and the visible content (the first index is 0). Value -1 means there is no selected tab.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Invokes the method when the tabbedpane has changed its selected tab index.

Tab

Tabs are identified in the component's list as 'tab', and components with the 'component' key.

- **Properties**

Name	Type	Default	Description
`mnemonic`	`integer`	`-1`	Specifies the underlined char and the key combination which selects the tab.

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

- **Keyboard Actions**

Operation	Action
Left arrow, up arrow	Selects previous tab
Right arrow, down arrow	Selects next tab
Tab, Shift-Tab	Navigate forward, backward

Panel

Panel is a container with a layout manager similar to GridBagLayout. The following example is similar to BorderLayout. This panel has 5 components, the first row contains only the 'North' component, the second row has 3 components, and the last has only the 'South' one. The extra space is distributed to the 2nd row and the 2nd column. The gap between the components and on the border are 4pt.

```
<panel columns="3" gap="4" top="4" left="4" bottom="4"
right="4">
     <textfield text="North" colspan="3" />
     <label text="East" />
     <textarea text="Center" weightx="1" weighty="1" />
     <label text="West" />
     <textfield text="South" colspan="3" />
</panel>
```

- **Properties**

Name	Type	Default	Description
columns	integer	0	Specifies the number of available cells in a row. Default 0 value specifies 1 row and unlimited cell columns.
top	integer	0	The border gap from the top. It specifies the space that the panel must leave.
left	integer	0	The blank space from the left.
bottom	integer	0	The blank space from the bottom.
right	integer	0	The blank space from the right.
gap	integer	0	The horizontal and vertical gap between components.
text	string		The text string that the dialog title displays.

icon	icon		The icon image that the dialog title displays.
border	boolean	false	Border painted if true.
scrollable	boolean	false	If present and set to true, add scrollbars to the panel if its contents is bigger than the panel's bounds.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

Dialog

Dialog is similar to panel, but it has border and title, and you can drag it.

- **Properties**

Name	Type	Default	Description
text	string		The title is displayed in the title bar.
icon	icon		An image to be displayed in the titlebar of the dialog.
modal	boolean	false	A modal dialog grabs all the input to the components behind the dialog from the user.
resizable	boolean	false	Set whether the dialog can be resized.
closable	boolean	false	Set whether the dialog can be closed by a button.
maximizable	boolean	false	Set whether the dialog can be maximized by a button.
iconifiable	boolean	false	Set whether the dialog can be iconified by a button.

Columns, top, left, bottom, right, and gap parameters are described at panel.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Keyboard Actions**

Operation	Action
Tab, Shift-Tab	Navigate forward, backward inside the dialog
Ctrl-F6, Alt-F6, Ctrl-Shift-F6, Alt-Shift-F6	Navigate forward, backward, outside of the non-modal dialog too.

Spin Box

SpinBox is a textfield width 2 arrow buttons to change (increase, and decrease by 1) its integer value. The textfield input is filtered for digits, but the range isn't checked, you can also delete its content.

```
<spinbox text="100" />
```

- **Properties**

Name	Type	Default	Description
`minimum`	`integer`	`Integer.MIN_VALUE`	Minimum value. It's not possible to set the spinbox below this value by the arrow button.
`maximum`	`integer`	`Integer.MAX_VALUE`	Maximum value. It's not possible to set the spinbox above this value by the arrow button.

`step`	`integer`	`1`	Increment step when using spinbox arrows. 1 by default.

Text, columns, editable, and alignment parameters are described at textfield

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Invokes the given method when there was change in the text or a new value was spinned.

- **Keyboard Actions**

Operation	Action
Up arrow	Decrease the value
Down arrow	Increase the value
Tab, Shift-Tab	Navigate forward, backward

Progress Bar

ProgressBar displays an integer value within a bounded interval, the following between 25 and 75. Its value is currently in the middle.

```
<progressbar minimum="25" maximum="75" value="50"
orientation = "vertical" />
```

- **Properties**

Name	Type	Default	Description
`orientation`	`choice`	`horizontal`	Possible values are: horizontal, and vertical. The default value, if not set, is horizontal.
`minimum`	`integer`	`0`	The progressbar's minimum value. By default, this is 0.
`maximum`	`integer`	`100`	The progressbar's maximum value. By default, this is 100.
`value`	`integer`	`0`	The value is always between the progressbar's minimum and maximum values, inclusive. By default, the value equals the minimum.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

Slider

A slider lets the user graphically select a value by sliding a knob within a bounded interval, e.g. between 25 and 75.

```
<slider minimum="25" maximum="75" value="50"
orientation="vertical" />
```

- **Properties**

Name	Type	Default	Description
`orientation`	`choice`	`horizontal`	Possible values are: horizontal, and vertical. The default value, if not set, is horizontal.
`minimum`	`integer`	`0`	The slider's minimum value. By

			default, this is 0.
`maximum`	`integer`	`100`	The slider's maximum value. By default, this is 100
`value`	`integer`	`0`	The value is always between the slider's minimum and maximum values, inclusive. By default, the value equals the minimum.
`unit`	`integer`	`5`	The distance of the value change when using arrow buttons
`block`	`integer`	`25`	The distance of the value change when using page buttons.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- ## Listeners

Name	Description
`action`	Invokes the method when the slider has changed its value

- ## Keyboard Actions

Operation	Action
Left arrow, up arrow	Decreases the value (steps in left/top direction) by unit value
Right arrow, down arrow	Increases the value (steps in right/bottom direction) by unit value
Page Up	Decreases the value (jumps in left/top direction) by block value
Page Down	Increases the value (jumps in right/bottom

	direction) by block value
Home	Moves to minimum (left/top) value
End	Moves to minimum (left/top) value
Tab, Shift-Tab	Navigate forward, backward

Split Pane

SplitPane is used to divide two components. The following one has two text-areas aligned top to bottom.

```
<splitpane orientation="vertical" divider="24">
 <textarea text="Top"/>
 <textarea text="Bottom" />
</splitpane>
```

- **Properties**

Name	Type	Default	Description
`orientation`	`choice`	`horizontal`	The splitpane orientation is either horizontal or vertical. By default the splitter is divided horizontally.
`divider`	`integer`	`-1`	The location of the divider. A negative value implies the divider should be reset to a value that attempts to honor the preferred size of the two components.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Keyboard Actions**

Operation	Action
F8	Gives focus to splitter bar
Left arrow, up arrow	Steps location of splitter bar in left/top direction
Right arrow, down arrow	Steps location of splitter bar in right/bottom direction
Home	Moves splitter bar to left/top of splitter pane
End	Moves splitter bar to right/bottom of splitter pane
Tab, Shift-Tab	Navigate forward, backward

List

A list allows the user to select one or more objects from a list, and internally handles scrolling. The following example list has three items, the first is selected and the last is disabled, allows to select multiple items.

```
<list selection="multiple">
     <item text="List" selected="true" />
     <item text="Item"/>
     <item text="Disabled" enabled="false" />
</list>
```

- **Properties**

Name	Type	Default	Description
`selection`	`choice`	`single`	Possible values are: single, interval, and multiple. The default single value allows to select one list item at a time, the interval value to select one contiguous range of items, and the multiple value to select one or more contiguous ranges of items.

`line`	`boolean`	`true`	If present and set to false, don't draw lines separating the list items

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Calls the method whenever the selection changes.
`perform`	Calls the method whenever a double-click event occurs

Item

List contains items, items display a short text string and an image and can be selected.

- **Properties**

Name	Type	Default	Description
`selected`	`boolean`	`false`	True if the item currently selected.

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

- **Keyboard Actions**

Operation	Action
Up arrow	Selects previous item
Down arrow	Selects next item
Home	Selects first item

End	Selects last item
Page Up	Selects item one view up
Page Down	Selects item one view down
+Shift	Extends selection
+Ctrl	Set lead item
Space	Selects lead item
Ctrl-A, Ctrl-Slash	Selects all
Ctrl-Backslash	Deselects all
Tab, Shift-Tab	Navigate forward, backward

Table

A table presents data in a two-dimensional table format, allows to select rows, has header, and internally handles scrolling. The following example table includes two columns ('A', and 'B'), and two rows, the first row is selected, but allows to select multiple rows.

```
<table selection="multiple">
     <header>
     <column text="A" width="24" />
     <column text="B" />
     </header>
     <row selected="true">
     <cell text="a1" />
     <cell text="b1" enabled="false" />
     </row>
     <row>
     <cell text="a2" />
     <cell text="b2" />
     </row>
</table>
```

- **Properties**

Name	Type	Default	Description
`selected`	`choice`	`single`	Possible values are: single, interval, and multiple. The default single value allows to select one row at a time, the interval value to select one contiguous range of rows, and the multiple value to select one or more contiguous ranges of rows.

Line, and selection parameters are similar to list.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Calls the method whenever the selection changes.
`perform`	Calls the method whenever a double-click event occurs.

1. Header

Table header includes the columns. They define the column texts and widths.

2. Column

Table header includes the columns. They define the column texts and widths. If the total sum of column widths exceeds table bounds set by a parent container, horizontal scrollbar is drawn.

- ### Properties

Name	Type	Default	Description
`width`	`integer`	`80`	The width of the column. The last (rightmost) column always extends to fill in the remaining width of the table, as determined by the parent container.
`sort`	`choice`	`none`	If present, and set to either "ascent" or "descent", an arrow is drawn near right end of the column, as a visual indication of sorting order. NOTE: for now this is just a visual decoration, application still needs to do the sorting.

Name, text, icon, and alignment parameters are described at label.

3. Row

Table contains rows (and columns for the header), and row contains cells. The 'row' key identifies the list of rows. An entire row (not cell) is selectable in a table.

- ### Properties

Name	Type	Default	Description
`selected`	`boolean`	`false`	True if currently selected the entire row.

4. Cell

Cell displays a short text and icon, its height is equals with the row's height, and its width is defined in the header column.

○ Properties

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

○ Keyboard Actions

Operation	Action
Up arrow	Selects previous row
Down arrow	Selects next row
Home	Selects first row
End	Selects last row
Page Up	Selects row one view up
Page Down	Selects row one view down
+Shift	Extends selection
+Ctr	Set lead row
Space	Selects lead row
Ctrl-A, Ctrl-Slash	Selects all
Ctrl-Backslash	Deselects all
Tab, Shift-Tab	Navigate forward, backward

Tree

Tree displays a set of hierarchical data, contains nodes (not only one root node is allowed), and a node could have subnodes. It internally handles scrolling. The following example shows a tree with 2 root nodes ('Node A', and 'Node D'), the first has 2 subnodes (a selected 'Node B' and 'Node C'), the collapsed 'Node D' includes one node.

```
<tree selection="multiple">
<node text="Node A" icon="image.gif">
<node text="Node B" icon="image.gif" selected="true" />
<node text="Node C" icon="image.gif" />
</node>
<node text="Node D" expanded="false">
<node text="Node E" icon="image.gif" />
</node>
</tree>
```

- **Properties**

Name	Type	Default	Description
`selection`	`choice`	`single`	Possible values are: single, interval, and multiple. For the default single value the selection can only contain one path at a time, for interval the selection can only be contiguous (of the currently visible nodes), and for the multiple value the selection can contain any number of nodes that are not necessarily contiguous.
`angle`	`boolean`	`false`	To specify to draw lines detailing the relationships between nodes.

Line, and selection parameters are similar to list.

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

- **Listeners**

Name	Description
`action`	Calls the method whenever the selection changes.

`perform`	Calls the method whenever a double-click event occurs.
`expand`	The listener that's notified when the tree expands a node.
`collapse`	Called whenever a node in the tree has been collapsed.

1. Node

Tree node is similar to list item, but maybe contains subnodes, and has collapse control.

○ Properties

Name	Type	Default	Description
`selected`	`boolean`	`false`	True if currently selected.
`expanded`	`boolean`	`true`	Ensures that the node is expanded if true, otherwise collapsed.

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

○ Keyboard Actions

Operation	Action
Right arrow	Expands current collapsed node, or selects its first subnode
Left arrow	Collapses current expanded node, or selects its parent node
Up arrow	Selects previous node

Down arrow	Selects next node
Home	Selects first node
End	Selects last node
Page Up	Selects node one view up
Page Down	Selects node one view down
+Shift	Extends selection
+Ctr	Set lead node
Space	Selects lead node
Ctrl-A, Ctrl-Slash	Selects all
Ctrl-Backslash	Deselects all
Tab, Shift-Tab	Navigate forward, backward

Separator

Separator is a horizontal or a vertical line. The following panel has a button('Left'), a vertical separator line, and a second button ('Right') in the first row, a horizontal separator in the second row, and the final button ('Bottom') in the third row.

```
<panel columns="3" gap="4" top="4" left="4">
<button text="Left" />
<separator />
<button text="Right" />
<separator colspan="3" />
<button text="Bottom" colspan="3" />
</panel>
```

- **Properties**

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

Menu Bar

The following panel contains a menubar on the top and a large text-area. The menubar has only one menu ('Menu'), one menuitem ('MenuItem') and a menu ('Menu'). The later's popup contains a menuitem with a selected checkbox ('CheckBox'), a separator, and a radio button ('RadioButton').

```
<panel columns="1" gap="4">
     <menubar weightx="1">
     <menu text="Menu">
     <menuitem text="MenuItem" icon="image.gif" />
     <menu text="Menu">
     <checkboxmenuitem text="CheckBox" selected="true" />
     <separator />
     <checkboxmenuitem text="RadioButton" group="group"/>
     </menu>
     </menu>
     </menubar>
     <textarea weighty="1" />
</panel>
```

- **Properties**

Name	Type	Default	Description
placement	choice	top	Menubar may unfold the menus either downwards (default - for menubars placed at the top of the container), or upwards from the menubar position (bottom - for menubars placed at the bottom of the container).

Name, enabled, visible, tooltip, property, width, height, colspan, rowspan, weightx, weighty, halign, and valign parameters are similar to component.

1. Menu

Menubar includes menu, and menu could contain menuitems, check or radio button items, separators, and menus.

○ Properties

Name	Type	Default	Description
`mnemonic`	`integer`	`-1`	Specifies the index of the underlined char and the key combination (Alt + underlined character) which selects the menu.

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

2. Menu Item

A regular menu item can have either text or a graphic icon associated with it, or both.

○ Properties

Name	Type	Default	Description
`accelerator`	`keystroke`		The key combination which invokes the menuitem's action method without navigating the menu hierarchy.
`mnemonic`	`integer`	`-1`	Specifies the underlined char index (thus the key combination) which invokes the action listener.

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

- ### Listeners

Name	Description
`action`	Invokes the given method when the menuitem is pressed by mouse or keyboard.

3. CheckBox Menu Item

A menu item that can be selected or deselected. The selected menuitem appears with a checkmark next to it. A radiobutton menu-item is a menu item that is part of a group of menu items in which only one item in the group can be selected.

- ### Properties

Name	Type	Default	Description
`selected`	`boolean`	`false`	The state of the checkboxmenuitem.
`group`	`string`		Identifies a group if not null. Only one radio menuitem at a time can be selected in a group. User can set on a radio menuitem, the selected menuitem of the group will be set off (the group members is searched only in the same menu).

Name, enabled, text, icon, alignment, tooltip, and property parameters are similar to combobox choice.

Mnemonic, accelerator, and action parameters are described at menuitem.

- ## Keyboard Actions

Operation	Action
F10	Moves focus to menu bar and posts first menu
Up arrow	Selects previous item (or the last)
Down arrow	Selects next item (or the first)
Left arrow	Collapses last popupmenu, or selects previous (or the last) menu of menubar
Right arrow	Expands menu, or selects next (or the first) menu of menubar
Enter, Return, Spacebar	Activates a menu item, dismisses menu, and goes to last window item with focus
Escape	Dismisses menu without taking action and returns focus to last component that had focus

PART III :

INTERACT WITH SIMULATIONS

CHAPTER 1 :

INTRODUCTION

There are some dedicated global objects used to interact with bodies, joints and world, they are:

- `World :` used to add, edit, and remove bodies and joints
- `Resources :` used to access resources like graphics , sound and files
- `Actions :` used to perform some action on body in certain time
- `App :` used to create animation Timers and modal dialogs to display messages or to get values as input.

These objects can be accessed from anywhere in the script, to know the allowed properties/methods just write. after these keywords in script, a window containing all properties/methods of object opens, from where you can select one of them to enter in script.

Objects Type to interact with:

The above objects have methods which return or accept one of following object types:

- `Vector` : represents a vector or point in 2D space.
- `Color` : represents color holding the r, g, b and alpha components
- `Body` : physical object added to world
- `Joint` : joint connecting bodies
- `Tracer, field and controllers` : tools to define body motion

- **`Brush/Animation`** : loaded image or animation
- **`Sound`** : loaded sound
- **`Font`** : loaded font
- **`File`** : loaded file
- **`Action`** : when attached to body, performs some task, often over time

For example:

```
var d = World.addDisc(0.2);
// create a new vector
var pos = new Vector2(1, 0);
// place body at 1,0
d.setPosition(pos);
// set color of body to red
d.setFillColor(new Color("red"));
// find animation from resources folder with name "image"
var brush = Resources.getAnimation("image");
// create copy of animation/brush and apply to body
d.setBrush(brush.createCopy());
//finally insert body in the world
World.addBody(d);
```

CHAPTER 2 :

VECTOR & COLOR

The most common object used in script related to physics is Vector and that related to graphics is Color, thats why we are beginning with them.

Vector

This class represents a vector or point in 2D space. It is used almost everywhere in SimPhy, for body position, velocity, forces, joints anchor points, shapes locations etc.

The operations :

`Vector2.setMagnitude(double)`,
`Vector2.getNormalized()`,
`Vector2.project(Vector2)`, and
`Vector2.normalize()` require the `Vector2` to be non-zero in length.

Some methods also return the vector to facilitate chaining.

For example:

```
var a = new Vector2();
a.zero().add(1, 2).multiply(2);
```

Note: *All angles used in `Vectors` are in radians and anticlockwise is taken as positive.*

Constructor and Description

- `Vector2 ()` : Default Constructor
- `Vector2 (double direction)` : Creates a unit length vector in the given direction (in radians).
- `Vector2 (double x, double y)` : Optional Constructor
- `Vector2 (double x1, double y1, double x2, double y2)` : Creates a `Vector2` from the first point to the second point
- `Vector2 (Vector2 vector)` : Copy Constructor
- `Vector2 (Vector2 p1, Vector2 p2)` : Creates a `Vector2` from the first point to the second point.

Method and Description	Modifier Type
add`(double x, double y)` Adds the given `Vector2` to this `Vector2`.	`Vector2`
add`(Vector2 vector)` Adds the given `Vector2` to this `Vector2`.	`Vector2`
approxEqual`(Vector2 v)` Returns true if Vectors are very Close to each other	`boolean`
copy`()` Returns a copy of this `Vector2`.	`Vector2`
cross`(double z)` Returns the cross product of this `Vector2` and the z value of the right `Vector2`.	`Vector2`

cross(double x, double y) Returns the cross product of the this Vector2 and the given Vector2.	**double**
cross(Vector2 vector) Returns the cross product of the this Vector2 and the given Vector2.	**double**
difference(double x, double y) Subtracts the given Vector2 from this Vector2 returning a new Vector2 containing the result.	**Vector2**
difference(Vector2 vector) Subtracts the given Vector2 from this Vector2 returning a new Vector2 containing the result.	**Vector2**
distance(double x, double y) Returns the distance from this point to the given point.	**double**
distance(Vector2 point) Returns the distance from this point to the given point.	**double**
distanceSquared(double x, double y) Returns the distance from this point to the given point squared.	**double**
distanceSquared(Vector2 point) Returns the distance from this point to the given point squared.	**double**
dot(double x, double y) Returns the dot product of the given Vector2 and this Vector2.	**double**

dot(Vector2 vector) Returns the dot product of the given Vector2 and this Vector2.	**double**
equals(double x, double y) Returns true if the x and y components of this Vector2 are the same as the given x and y components.	**boolean**
equals(Object obj)	**boolean**
equals(Vector2 vector) Returns true if the x and y components of this Vector2 are the same as the given Vector2.	**boolean**
getAngleBetween(Vector2 vector) Returns the smallest angle between the given Vector2 in radians.	**double**
getAngleWithPositiveXAxis() Returns the angle of this Vector2 with +ve x axis as an angle in radians.	**double**
getDirection() Returns the direction of this Vector2 as an angle in radians.	**double**
getLeftHandOrthogonalVector() Returns the left-handed normal of this vector.	**Vector2**
getMagnitude() Returns the magnitude of this Vector2.	**double**
getMagnitudeSquared() Returns the magnitude of this Vector2 squared.	**double**

getNegative() Returns a Vector2 which is the negative of this Vector2.	**Vector2**
getNormalized() Returns a unit Vector2 of this Vector2.	**Vector2**
getRightHandOrthogonalVector() Returns the right-handed normal of this vector.	**Vector2**
getRotated(double theta) Returns new vector by Rotating this vector about the origin	**Vector2**
getRotated(double theta, Vector2 point) Returns new vector by Rotating this vector about the origin about the given point	**Vector2**
getXComponent() Returns the x component of this Vector2.	**Vector2**
getYComponent() Returns the y component of this Vector2.	**Vector2**
isOrthogonal(double x, double y) Returns true if the given Vector2 is orthogonal (perpendicular) to this Vector2.	**boolean**
isOrthogonal(Vector2 vector) Returns true if the given Vector2 is orthogonal (perpendicular) to this Vector2.	**boolean**
isZero()	**boolean**

Returns true if this `Vector2` is the zero `Vector2`.	
`left``()` Sets this vector to the left-handed normal of this vector.	**`Vector2`**
`multiply``(double scalar)` Multiplies this `Vector2` by the given scalar.	**`Vector2`**
`negate``()` Negates this `Vector2`.	**`Vector2`**
`normalize``()` Converts this `Vector2` into a unit `Vector2` and returns the magnitude before normalization.	**`double`**
`product``(double scalar)` Multiplies this `Vector2` by the given scalar returning a new `Vector2` containing the result.	**`Vector2`**
`project``(Vector2 vector)` Projects this `Vector2` onto the given `Vector2`.	**`Vector2`**
`right``()` Sets this vector to the right-handed normal of this vector.	**`Vector2`**
`rotate``(double theta)` Rotates about the origin with angle theta in radians ACW as positive.	**`Vector2`**
`rotate``(double theta, double x, double y)` Rotates the `Vector2` about the given coordinates.	**`Vector2`**

rotate(double theta, Vector2 point) Rotates the Vector2 about the given point.	**Vector2**
set(double x, double y) Sets this Vector2 to the given Vector2.	**Vector2**
set(Vector2 vector) Sets this Vector2 to the given Vector2.	**Vector2**
setDirection(double angle) Sets the direction of this Vector2.	**Vector2**
setMagnitude(double magnitude) Sets the magnitude of the Vector2.	**Vector2**
subtract(double x, double y) Subtracts the given Vector2 from this Vector2.	**Vector2**
subtract(Vector2 vector) Subtracts the given Vector2 from this Vector2.	**Vector2**
sum(double x, double y) Adds this Vector2 and the given Vector2 returning a new Vector2 containing the result.	**Vector2**
sum(Vector2 vector) Adds this Vector2 and the given Vector2 returning a new Vector2 containing the result.	**Vector2**
to(double x, double y) Creates a Vector2 from this Vector2 to the given Vector2.	**Vector2**

to(Vector2 vector) Creates a Vector2 from this Vector2 to the given Vector2.	Vector2
zero() Sets the Vector2 to the zero Vector2	Vector2

For example:

```
var a=new Vector2(1,1);
var b=new Vector2(2,-1);
Console.show();
Console.println("|a| = "+a.getMagnitude());
//displays |a| = 1.4142135623730951
Console.println("a+b = "+a.sum(b));
//displays a+b = (3.0, 0.0)
Console.println("a.b = "+a.dot(b));
//displays a.b = 1
```

Color

A Color object, holds the r, g, b and alpha component as floats in the range [0,1]. All methods perform clamping on the internal values after execution. Used for body , joint and shapes colors as well as graphics related operations (in 2d and 3d canvas)

Note: *Red green, blue and alpha components are stored as public properties r, g, b and a respectively.*

Constructor and Description

- Color(): Constructs a new Color with all components set to 0.
- Color(float r, float g, float b, float a): Creates an sRGB color with the specified red, green, blue, and alpha values in the range (0.0 - 1.0).

- **`Color(int rgba8888)`** : Creates an sRGB color with the specified combined RGBA value consisting of the alpha component in bits 24-31, the red component in bits 16-23, the green component in bits 8-15, and the blue component in bits 0-7.
- **`Color(String cssColor)`** : Creates sRGB color from CSS color, and probably the most dynamic way to create color (see below)

  CssColor in this constructor can be any string representing css color, it can be any of following :

  1. **color name :** opaque color as java default color names

     Ex: `red, green, purple`

  2. **hex codes :** opaque color with HEX notation for the combination of Red, Green, and Blue color values (RGB)

     Ex: `#FF0000, #FFFFFF,#6A5ACD </small>`

  3. **rgb function :** opaque color with RGB values from 0 to 255

     Ex: `rgb(255,0,0), rgb(255,255,255), rgb(0,0,250)`

  4. **rgba function :** transparent color with RGB values from 0 to 255 and alpha from 0 to 1

     Ex: `rgba(255,100,120,0.5), rgba(255,12,32,.5) </small>`

  5. **hsl function :** opaque color with Hue from 0 to 360, saturation and lightness each from 0 to 100

     Ex: `hsl(180, 50%, 50%)`

  6. **hsla function :** opaque color with Hue from 0 to 360, saturation and lightness each from 0 to 100 and alpha from 0 to 1

     Ex: `hsla(120, 100%, 50%, 0.3) </small>`

Method Summary

add(**Color** color) Adds the given color to this color.	**Color**
brighter() Creates a new Color that is a brighter version of this Color.	**Color**
cpy()	**Color**
darker() Creates a new Color that is a darker version of this Color.	**Color**
foreground() Returns a foreground color (for text) given a background color by examining the brightness of the background color.	**Color**
highlight() Returns highlight color for this color	**Color**
lerp(**Color** target, float t) Linearly interpolates between this color and the target color by t which is in the range [0,1].	**Color**
lerp(float r, float g, float b, float a, float t) Linearly interpolates between this color and the target color by t (in order of t:1) which is in the range [0,1].	**Color**
mul(**Color** color) Multiplies the this color and the given color	**Color**
mul(float value) Multiplies all components of this Color with the given value.	**Color**
set(**Color** color) Sets this color to the given color.	**Color**
set(float r, float g, float b, float a) Sets this Color's component values.	**Color**

shadow() returns shadow of a color	Color
sub(**Color** color) Subtracts the given color from this color	Color
toRGBA8888() returns color as 32bit in RGBA8888 format	int

For example:

```
// All variables have same color as 'red'
var a =new Color("red");
var b= new Color(1,0,0,1);
var c =new Color("#FF0000");
var d= new Color("rgb(255, 0, 0,1)");
var e= new Color("rgba(255, 0, 0,1)");
var f= new Color("hsl(0, 100, 50)");

Console.show();
Console.println(a);     //prints #ff0000ff
Console.println(b);     //prints #ff0000ff
Console.println(c);     //prints #ff0000ff
Console.println(d);     //prints #ff0000ff
Console.println(e);     //prints #ff0000ff

//below will print 255,0,0
Console.println(Math.round(f.r*255)+" ,
"+Math.round(f.g*255)+" , "+Math.round(f.b*255));
```

Every simulation is associated with a *world* which manages bounds, gravity, electromagnetic fields and air drag which act on all objects in the world (or simulation).

Whatever you see on screen is a simulated physics world (except widgets and math curves of course!)

CHAPTER 3 :

WORLD : ITS OBJECTS & EVENTS

Accessing World in Script

World is accessed in script by global keyword 'world' itself (which must not be reassigned in code anywhere)

Properties related to the world can be retrieved or set by calling getters and setters functions in world object.

For example:

```
World.setBounds(boundX,boundY);
//to set bounds of world

World.setGravity(gX,gY);
//to set gravity of world
```

Method and Description	Modifier Type
addDisc(double radius) Adds disc object to the world	**Body**
addDistanceJoint(Body body1, Body body2, double frequency, double damping) Adds Distance Joint between bodies by connecting them at centers	**Joint**

addDistanceJoint(Body body1, Body body2, Vector2 p1, Vector2 p2, double frequency, double damping) Adds Distance Joint between bodies by connecting them at centers	**Joint**
addPolygon(Vector2[] vertices) Creates polygonal body	**Body**
addRectangle(double width, double height) Adds rectangle object to the world	**Body**
addRevoluteJoint(Body body1, Body body2, Vector2 v, double lowerLimit, double upperLimit) Adds revolute joint(hinge)	**Joint**
addSpringJoint(Body body1, Body body2, double SpringConstant) adds Spring joint between bodies by connected to respective centers	**Joint**
addTracer(Body body, Vector2 localPt) Adds trajectory tracer to a body	**Tracer**
clear() Clears all Dynamically generated objects by script	**void**
createCopy(Body body) Creates copy of object and adds it to world	**Body**

createGroup() Creates an empty group of bodies,bodies can be added to the group using COMBody.addBody(Body)	**COMBody**
getBody(String name) returns body recognized by its names	**Body**
getBodyAt(Vector2 pt) returns Body at a point	**Body**
getCameraBody() Returns body if exists, to which the camera is following, else returns null	**Body**
getController(String name) returns body controller by name	**BodyController**
getElectricFieldAt(Vector2 pt) returns electric field at a point	**Vector2**
getElectricPotentialAt(Vector2 pt) Returns Electric potential due to charges at a point	**double**
getField(String name) returns field from name	**Field**
getGravity() returns gravity vector for simulation (in m/s2)	**Vector2**
getJoint(String name) returns joint recognized by its names	**Joint**

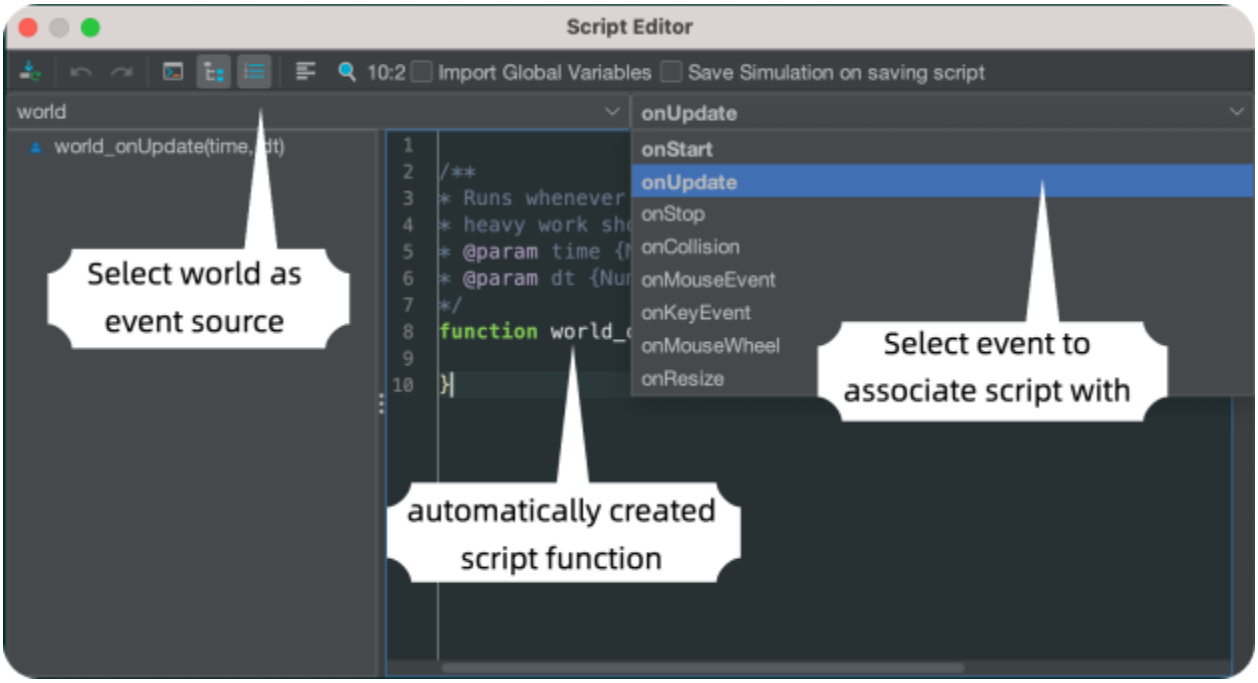
getMagneticFieldAt(Vector2 pt) returns magnetic field at a point in Tesla	**double**
getPointInCameraFrame(Vector2 v) returns point/position in camera reference frame	**Vector2**
getPointInGroundFrame(Vector2 v) returns point/position in in ground frame	**Vector2**
getSimulationTime() returns time elapsed in seconds (instantaneous time) since the beginning of simulation	**double**
getVectorInCameraFrame(Vector2 v) returns vector (velocity, force etc) as seen in camera frame	**Vector2**
getVectorInGroundFrame(Vector2 v) returns vector (velocity, force etc) in ground frame	**Vector2**
loadNextSimulation() load next simulation if there is next simulation linked, else does nothing	**void**
loadPrevSimulation() load previous simulation if there is next simulation linked, else does nothing	**void**
loadSimulation(String simName) load simulation specified by name if current simulation is multi simulation	**void**

remove(Body body) removes a body	**void**
remove(Joint joint) removes joint from world	**void**
removeAllTracers() removes all tracers attached to bodies	**void**
removeController(AbstractBodyController c) removes body controller from simulation	**void**
removeField(Field field) removes field from simulation	**void**
removeGroup(COMBody group) removes the group from simulation, bodies in the group are not removed	**void**
removeTracer(Tracer tracer) removes tracer from body	**void**
removeTracers(Body body) removes tracer from body	**void**
resetAllTracers() resets all tracers attached to bodies	**void**
resetTracers(Body body) Resets tracers attached to body	**void**

setCameraBody(Body body, boolean allowRotation) Attaches camera to the body, now the body will behave as world origin and camera will follow it	void
setGravity(double x, double y) Sets gravity for simulation (in m/s2)	void

World Events

SimPhy allows you to write JavaScript code corresponding to events associated with world. Open script editor Select the event type to add /edit script associated with this event.



1. The "OnStart" and "OnStop" Events

The "OnStart" and "OnStop" event are called by the physics world whenever the simulation starts and stops respectively. Both those properties are meant to hold some javascript code that it is going to be evaluated when the corresponding events occur.

For example:

An object named "Disc" already added to world, which changes its text to "Start!" when the simulation starts and changes again to "Stop!" when the simulation stops:

```
var disc=World.getBody("Disc");

function world_onStart() {
   disc.setText("Start!");
}
function world_onStop() {
   disc.setText("Stop!");
}
```

The "OnStart" event can be particularly useful to initialize a scene with objects kept at predefined locations.

For example, to add 10 objects on x-axis at separation 1m each:

```
   for (var i = 0; i < 10; i++) {
       var d = World.addDisc(0.2);
       d.translate(i, 0)
   }
```

2. The "OnUpdate" Event

The **`onUpdate()`** event is fired after every simulation step. If this property is not empty (i.e. it contains some JavaScript code) it will be executed by the program's script engine.

The argument passed to this event are *instantaneous time* of simulation (since simulation started) and **time elapsed** since last updates respectively.

For example:

Say that you want to add discs to the simulation after every 100 steps

Step 1: Declare global variable t in script t=0; to onStart Script in world

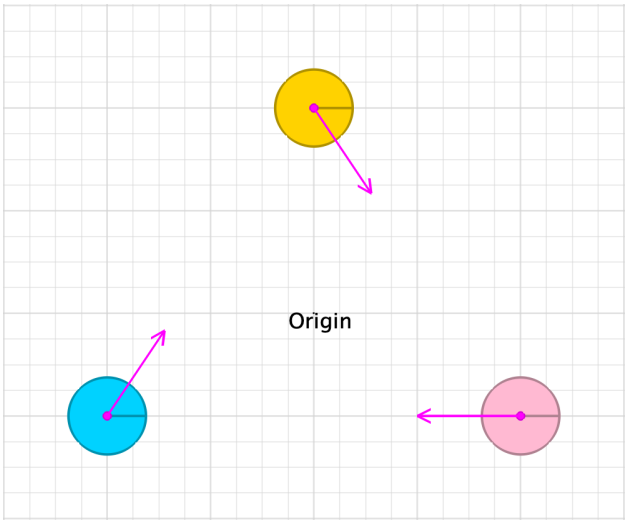
Step 2: Create disc after every 100 steps

```
var t = 0;
function world_onUpdate(time, dt) {
    t++;
    if (t == 100) {
        World.addDisc(0.2);
        t = 0;
    }
}
```

The "`OnUpdate`" property can also be useful to put constraint over object's motion.

For example:

To create simulation where three objects disc, disc1, disc2 follow each other. Create new simulation and add 3 discs (with default names) and arrange at vertices of the triangle as shown below.



Now add following code to `onUpdate()` property of world script and save script (hopefully with no compilation error).

```
//get bodies as variables
var disc=World.getBody("Disc");
var disc1=World.getBody("Disc1");
var disc2=World.getBody("Disc2");
//get body centers
var r1 = disc.getPosition();
var r2 = disc1.getPosition();
var r3 = disc2.getPosition();
//get unit vectors joining centers of pair of discs
var r12 = (r1.to(r2)).getNormalized();
var r23 = (r2.to(r3)).getNormalized();
var r31 = (r3.to(r1)).getNormalized();
//set velocity of each disc parallel to line joining it
with next disc
disc.setVelocity(r12.x, r12.y);
disc1.setVelocity(r23.x, r23.y);
disc2.setVelocity(r31.x, r31.y);
```

On running the simulation, you will observe that each disc follows its next disc (also try dragging any disc with mouse other discs will follow as its trial, seems pretty cool.. huh!)

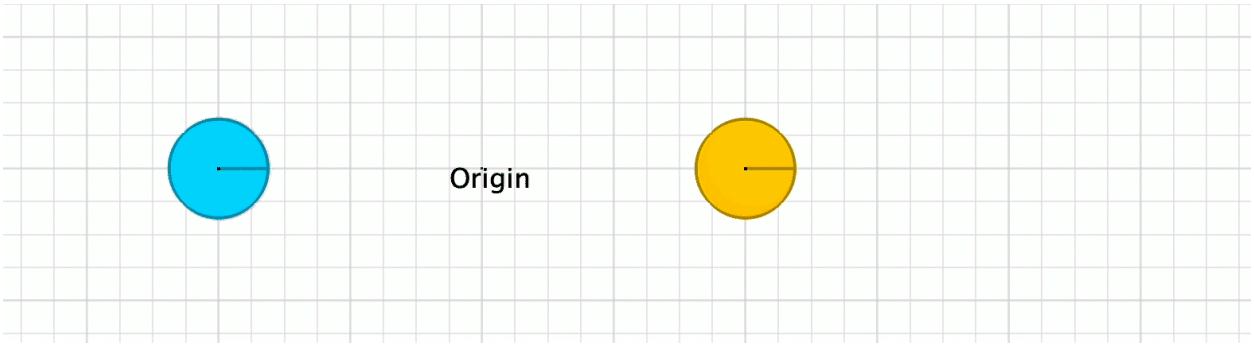
3. The "OnCollision" Event

This property is meant to hold javascript code which is going to be evaluated whenever the body collides with another body. The argument passed to the "`OnCollision`" method are body1, body2, time and pt, where body1 and body2 are bodies colliding together, time is time when collision occurred and Pt is point of impact in world coordinates.

For example:

Say you want to exchange colors of two bodies on collision. Create some objects of different colors in simulation and add following code to script, you will see bodies exchanging colors on impact.

```
function world_onCollision(body1, body2, time, pt) {
   var color = body2.getFillColor();
   body2.setFillColor(body1.getFillColor());
   body1.setFillColor(color);
}
```



4. The KeyEvents

The key events functions are invoked whenever the world registers key events, there are 2 mouse events with which script can be associated:

- keyPressed
- keyReleased

Each function is called with some argument containing event information.

For example:

If you an object to "jump" whenever the 'A' key is pressed, you'll have to set its "`onKeyEvent`" method to something similar to the snippet shown below:

```
var d = World.addDisc(0.3);

function world_onKeyEvent(keyChar,keyCode,key,id){
   if (keyCode == 65 & id == 401) {     // keycode =65= 'A'
       d.setVelocity(0, 3);
   }
}
```

For example:

If you want body to be accelerated right when 'd' is pressed and left when 'a' is pressed, and no acceleration otherwise in X direction, Create new simulation, copy paste following code in script editor, save script and run.

```
World.clear();
var d=World.addRectangle(0.4, 0.2);
d.setMass(1);
d.setFriction(0.1);
var accRight=false;
var accLeft=false;
var force=2;    //force of 2 Newton

function world_onKeyEvent(keyChar,keyCode,key,id){
  if(keyCode==65 & id == 401){
      d.applyForce(-force, 0);
  }
  if(keyCode==68 & id == 401){
      d.applyForce(force, 0);
  }
}
```

After running simulation you will see a box falling on ground, press keys 'a' and 'd' to move it left and right.

5. The MouseEvents

The mouse events functions are invoked whenever the canvas registers mouse events, there are 4 mouse events with which script can be associated

- mouseClicked
- mousePressed
- mouseReleased
- mouseMoved

Each function is called with some argument containing event information.

For example,

Complete mouse clicked event is as shown, where all arguments are explained.

```
/**
* Runs when the world recieves mouse event (events are
similar to java.awt events).
* @param x {Number} : x coordinate of mouse in widget space
(top left as origin)
* @param y {Number} : y coordinate of mouse in widget space
(top left as origin)
* @param worldPt {Vector2} :point coresponding to mouse
location in world space
* @param clickCount {Number} : Number of clicks
* @param id {Number}: EventType:
500(clicked),501(pressed),502(released),503(moved),504(ente
red),505(exited)
* @param button {Number} : Button involved in event
{1=Left, 2=Middle and 3=Right mouse button\n* @return if
true is returned event is consumed and not further handled
by simphy world
*/
function
world_onMouseEvent(x,y,worldPt,clickCount,id,button){
   var body=World.getBodyAt(worldPt);
   if(body!=null && body.getMass() > 0){
       body.setVelocity(0, 2);
   }
}
```

The above code will make body jump up with speed 2 m/s, whenever mouse is clicked over it.

CHAPTER 4 :

BODIES & JOINTS

Accessing Body /Joint in Script

Body or joint already existing in simulation can be accessed by calling getBody() and getJoint() functions of `'world'` object.

For example:

```
var d=World.getBody("Disc");
//stores body of name 'Disc' in variable d
var jt=World.getJoint("joint1");
//stores joint of name 'joint1' in variable jt
```

Creating Body / Joints

1. By Defining New Objects

New Body or joint can be created by calling `world.addXXX()` and `world.addXXXJoint()` functions.

The following code creates a rectangle and a disc connected by a spring of force constant 40 N/m.

```
var d = World.addDisc(0.5);
var r = World.addRectangle(1, 1);
r.setPosition(2, 0);
var j = World.addSpringJoint(d, r, null, null, 30, 0);
```

2. From Existing Objects

Clone of body can be created and added to world by calling function `createCopy(body)` in world object.

```
var d=World.getBody("Disc");
var d1=World.createCopy(d);
//creates copy of Disc and adds to world
```

Clones of body can also be created by calling `copy()` from body object, but note that this function creates copy of object and returns it without adding it to world.

This returned copy can later be added to world by using `world.addBody();`

```
var d=World.getBody("Disc");
var d1=d.copy();
World.addBody(d1);
```

Changing Appearance

There exist few functions in body object to change body's color, image and text.

```
var d=World.getBody("Disc");
d.setFillColor(“white”);
d.setOutlineColor(new Color("red"))
d.setBrush("image1");
//make sure animation with name "image1" already exists
d.setText("custom text");
var j=World.getJoint("joint");
j.setColor(new Color("green"));
```

Setting Text on Body

Body can act like a text display box by setting its text property using setText(text) method on body object text can be simple text or may contain shortcodes for multiple styles defined by comma separated attribute value pairs inside square brackets.

- **Allowed attributes**

  font = Name of currently loaded fonts

  x = x coordinate of text position in body local coordinates

  y = y coordinate of text position in body local coordinates

  color = CSS color string like "red" or "rgb(255,0,0)" etc

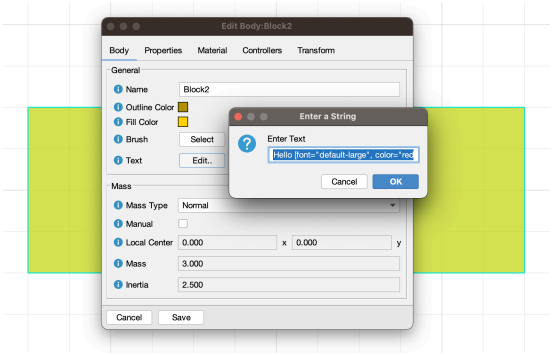
  xalign = 0, 1 or 2 for left, center and right respectively

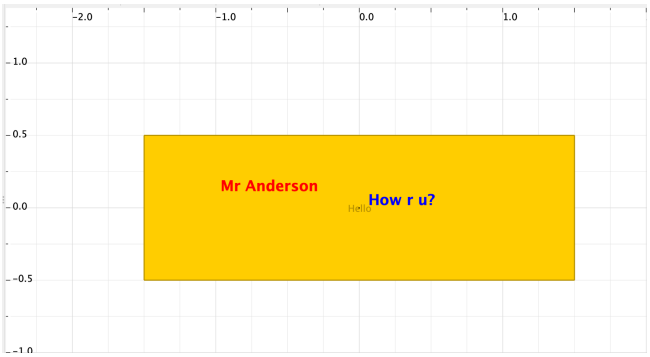
  yalign = 0, 1 or 2 for bottom, up and down respectively

  For example,

  Following value of text set on rectangular body will result as in screenshot below.

```
Hello [font="default-large", color="red" alignX="left"
alignY="top"  x="-1",y="0.2"] Mr Anderson
[color="blue" x="0.03" y="0.1"] How r u?
```





Dynamics

All kinematic properties like position, velocity, rotation as well as dynamic properties like mass, inertia, force and torque can be accessed and set using script. for example the following code instantly brings body to origin and set is speed 3m/s in y direction (by applying suitable impulse)

```
var d = World.getBody("Disc");
d.setPosition(0, 0);
d.setVelocity(0, 0);
d.applyImpulse(new Vector2(0, 3).product(d.getMass()));
```

Actions

Each body has an action which is updated each frame internally. Many types of actions are included with SimPhy. These can be instantiated, configured, and added to a body. When the action is complete, it will automatically be removed from it.

Actions are generally used to perform some task , often over time, giving nice effect using specific interpolation.

Actions are created using methods of `actions` object, for example following code moves body from position (0,0) to (3,0) in 3 seconds following `"bounce"` interpolation.

```
var d = World.getBody("Disc");
//don't let other forces interact with body
d.setMass(0);
//set initial position of body
d.setPosition(0, 0);
//create action object
var a = Actions.moveTo(3, 0, 4, "bounce");
//set action to body
d.setAction(a);
```

Try adding the above code in some button click event run simulation, then click the button to see bouncy animation of Disc (assuming it exists in simulation ! For details about action see chapter dedicated to actions.

For details about action see chapter dedicated to actions.

Method Summary for Bodies

Method and Description	Modifier and Type
applyForce(double fx, double fy)	**void**
applyImpulse(double jx, double jy) Applies Impulse(in Newton-sec) on center of body	**void**
applyImpulse(double jx, double jy, double px, double py) Applies Impulse(in Newton sec) at specific point of body	**void**
copy() Returns the Copy of this body, the copied body is same in look and size, but is placed at origin and has speed zero	**Body**
copy(boolean applyTransform) Returns the Copy of this body, the copied body is same in look and size	**Body**
getAction() Returns action associated to the body	**Action**
getAngularVelocity()	**double**

Returns angular velocity of the body in radians per second	
`getBrush()`	`Brush`
`getCharge()`	`double`
`getFillColor()` Returns the fill color.	`Color`
`getHeight()` Returns width of bounding box of non transformed body	`double`
`getInertia()` Get moment of inertia of body about its center of mass(return 0 of body is static)	`double`
`getMass()` Get mass of body (return 0 of body is static)	`double`
`getName()` Returns the name of the body.	`String`
`getOpacity()`	`int`
`getOutlineColor()` Returns the outline color.	`Color`
`getPosition()` Position of center of mass of body in world coordinates	`Vector2`
`getRotation()`	`double`

Returns rotation in radians CCW as positive	
getText() Returns text associated with body	**String**
getUserData() Returns custom user data associated with body	**Object**
getVelocity() Returns velocity of center of mass in world coordinates	**Vector2**
getWidth() Returns height of bounding box of non transformed body	**double**
getzOrder() Returns z-order of the body	**int**
isFbdDrawn() True, if free body diagram of body is to be drawn	**boolean**
isRenderable() Returns Sets visibility of body	**boolean**
isSensor() Returns true if body doesn't sense and process collision	**boolean**
isTouchable()	**boolean**
rotate(double th) Sets rotation about its center of mass.	**void**

scaleBy(double xScale, double yScale) Scales body in each direction	**void**
scaleTo(double xScale, double yScale) Scales body in each direction	**void**
setAction(Action action) Sets Action of body	**void**
setAngularVelocity(double w) Sets Angular Velocity of body	**void**
setAnimation(Brush brush)	**void**
setBrush(Brush brush)	**void**
setCharge(double charge)	**void**
setFDrawn(boolean fbdDrawn) Sets free body diagram of the body enabled	**void**
setFillColor(Color color) Sets Fill Color of the body (to disable filling shape pass null ar arguement)	**void**
setFillColor(float r, float g, float b, float a) Sets outline color of body	**void**
setFriction(double mu) Sets friction for body	**void**
setInertia(double I)	**void**

Set Moment of inertia of body about center of mass	
setMass(double m) Set mass of body	**void**
setName(String name) Sets the name of the body.	**void**
setOpacity(int opacity) Sets opacity of body	**void**
setOutlineColor(Color color) Sets Outline Color of the body (to disable rendering outline pass null ar arguement)	**void**
setOutlineColor(float r, float g, float b, float a) Sets outline color of body	**void**
setPosition(double x, double y) Sets position of body in world coordinates	**void**
setPosition(Vector2 v) Set position of body	**void**
setRenderable(boolean renderable) Sets visibility of body	**void**
setRestitution(double e) Set coefficient of restitution	**void**
setRotation(double th) Sets rotation about its center of mass.	**void**

setSensor(boolean sensor) Sets if body can sense and process collision	**void**
setSize(double width, double height) Sets size of body such that width and height in parameter become size of bounding box	**boolean**
setText(String s) Sets text associated with body	**void**
setTouchable(boolean touchable)	**void**
setUserData(Object userData) Sets custom user data for this body	**void**
setVelocity(double vx, double vy) Sets linear velocity of body in world coordinates	**void**
setzOrder(int zOrder) Sets z-order of the body	**void**
translate(double x, double y) Translates body in world	**void**
translateToOrigin() Moves body such that its center lies at world origin	**void**

Method Summary for Joints

Method and Description	Modifier and Type
getAnchor1 ()	abstract **Vector2**

Returns the anchor point on the first Body in world coordinates.	
getAnchor2 () Returns the anchor point on the second Body in world coordinates.	abstract **Vector2**
getColor ()	**Color**
getName () Returns the name of the joint.	**String**
getOpacity ()	**int**
getReactionForce (double invdt) Returns the force applied to the Body in order to satisfy the constraint in Newtons.	abstract **Vector2**
getReactionTorque (double invdt) Returns the torque applied to the Body in order to satisfy the constraint in newton-meters.	abstract **double**
getSize () Size of the joint	**float**
getUserData () Returns custom user data associated with joint	**Object**
isCollisionAllowed () Returns true if collision between the joined Body is allowed.	**boolean**
isRenderable ()	**boolean**
setCollisionAllowed (boolean flag)	**void**

Sets whether collision is allowed between the joined Bodys.	
setColor (Object color) Sets color with which joint should be rendered	**void**
setName (String name) Sets the name of the joint.	**void**
setOpacity (int opacity) Sets opacity of joint	**void**
setRenderable (boolean renderable)	**void**
setSize (float size) Size of the joint (size=1 means default size) size >1 scales up size while size<1 scales down the size (max size=5)	**void**
setUserData (Object userData) Sets custom user data for this joint	**void**

CHAPTER 5 :

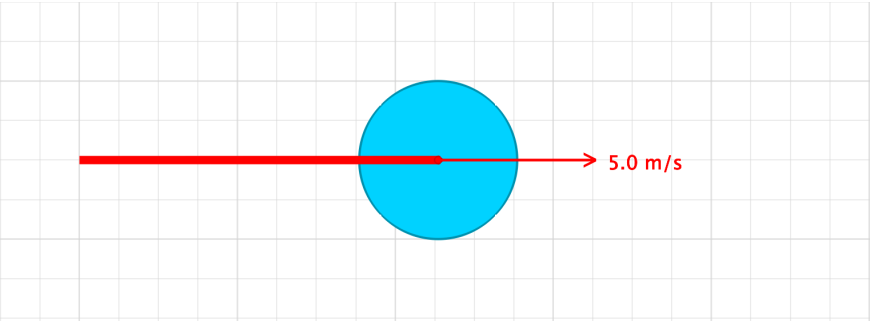
TRACERS, FIELDS & CONTROLLERS

TRACERS

Tracers are used to draw the trajectory of a point on body, sometimes we need to create tracers programmatically.

Creating Tracers

```
var d=World.getBody("Disc");
//attach tracer to center of disc
var tr = World.addTracer(d, null);
// adjust properties of tracer
tr.setColor(new Color("red"));
tr.setShowAngularVelocity(false);
```



Removing Tracers

- To remove tracer (with reference variable 'tr') use :

```
World.removeTracer(tr);
```

- To remove tracer for specific body (with reference variable 'b')

```
World.removeTracers(b)
```

- To remove all tracers

```
World.removeAllTracers()
```

Each body has an action which is updated each frame internally. Many types of actions are included

Method Summary

Method and Description	Modifier and Type
`getAcceleration()` Returns acceleration of tracer point in ground frame	`Vector2`
`getAngAcceleration()` Returns angular acceleration of body in radian/s2 w.r.t. ground	`double`
`getBody()` Returns body to which tracer is associated	`Body`
`getBodyPt()` Returns point in body space whose trajectory is to be traced	`Vector2`
`getColor()`	`Color`
`getMode()`	`int`

Returns mode used to draw Tracer, can be Solid [0] or dotted [1]	
getPtsCount() Returns maximum number of points in tracer array	**int**
getPtSize() Returns pixel size of tracer	**int**
getVelocity() Returns velocity in world space of point whose trajectory is to be traced	**Vector2**
getWorldPoint() Returns point in world space whose trajectory is to be traced	**Vector2**
isShowAcceleration() Returns true if linear acceleration is shown in tracers	**boolean**
isShowAngularVelocity() Returns true if angular velocity is to be dispalyed	**boolean**
isShowVelocity() True, if linear velocity is to be shown in tracer	**boolean**
reset() Reinitializes the tracer	**void**
setBody(Body body) Sets the body at which tracer point lies (it also sets tracer point as body centre)	**void**

setBodyPt(Vector2 bodyPt) Sets trace point on the body associated with tracer	**void**
setColor(Color color) Sets Fill Color of the body (to disable fill shape pass null arguement)	**void**
setMode(int mode) Sets mode used to draw Tracer can be Solid [0] or dotted [1]	**void**
setPtsCount(int count) Sets maximum number of points in tracer array	**void**
setPtSize(int ptSize) Sets pixel size of tracer	**void**
setShowAcceleration(boolean showAcceleration) Sets if linear acceleration is to be displayed in tracer	**void**
setShowAngularVelocity(boolean showAnglurVel) Sets if angular velocity is to be dispalyed	**void**
setShowVelocity(boolean showVelocity) Sets if linear veclocity of tracer point is displayed	**void**

FIELDS

SimPhy supports setting up of various fields other than gravity, which can only be added manually from front end.

- **Gravity:**

  The gravitational force on an object is calculated as :
  Force = mass x gravity x gravityFactor

- **Electrostatic field:**

  Electrostatic field (in N/C) added to the `world`, apply force on object iff it has non zero charge and it is inside bounds of field.

- **Magnetic field:**

  The magnetic lorentz force on an object is calculated asForce = charge x speed x magnetic field and is perpendicular to the velocity.

- **Buoyancy and drag:**

  Buoyancy can be added to a region representing hypothetical liquid in that region. The density and viscosity of liquid can be changed while adding/editing field.

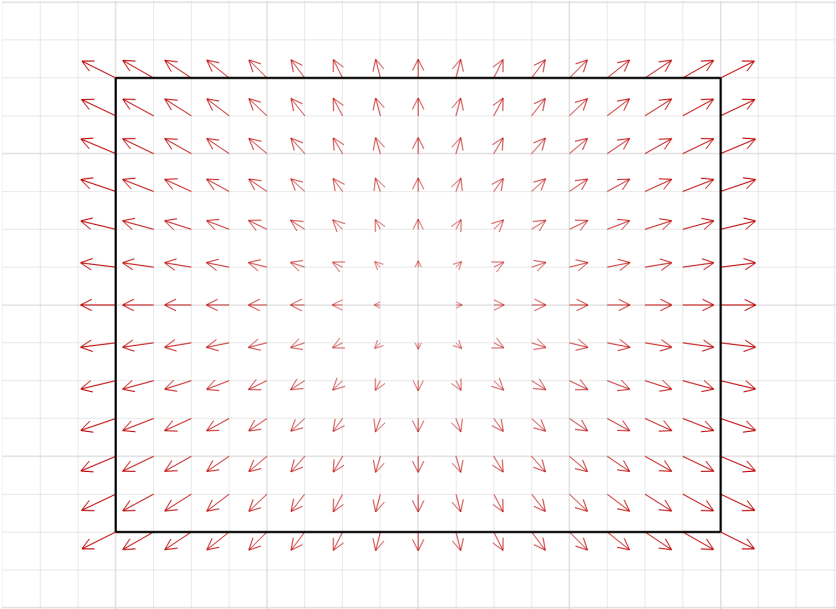
Editing Field

SimPhy **doesn't support creation or removal of field** programmatically (does it need so?), although you can access already created field and enable or disable them through script.

For example,

Let there be an electric field already created in SimPhy with name 'El', it can be accessed and modified through script as below:

```
var f = World.getField("El");
f.setEnabled(false);
f.setExpressions("x", "sin(y)");
f.setEnabled(true);
```



Method Summary

Method and Description	Modifier and Type
calculateForce(Body body) Calculates the instantaneous force based on current variables values and return force vector. Returns null if field/force at this point is not defined.	Vector2
getBounds() Returns bounds of this field	Object

getColor() Returns color used to render field	**Color**
getForce(Body body, double timestep) Returns force on body applied by this field. Returns null if field is not active or body is static or body lies outside the bounds of the field.	**Vector2**
getIntensityAtPoint(Vector2 v) Returns intensity of field at a point	**Vector2**
getName() Returns name/identifier of the field	**String**
getSymbol() Returns default symbol used to represent field	**String**
getUnit() Returns SI unit in which field is represented	**String**
getxExpr() Returns expression for x component of field	**String**
getyExpr() Returns expression for y component of field	**String**
isDefined() Returns true if expression of field is defined	**boolean**
isEnabled() Returns true if field is defined and active , therefore can apply force on bodies	**boolean**

setBounds(Object bounds) Sets bound of the field	**void**
setCircularBounds(double cx, double cy, double r)	**void**
setColor(Color color) Sets color used to render field	**void**
setEnabled(boolean enabled) If disabled, neither field is rendered nor applies any force.	**void**
setExpressions(String xExpr, String yExpr) Sets expression for the field, the expression can be a function of position **x**, **y** or polar coordinates **r** and **th**(theta)	**void**
setIntensity(Vector2 intensity) Resets expression of intensity and parser and sets intensity to constant value force	**void**
setName(String name) Sets name of the field, ame is used as identifier for field in script	**void**
setRectangularBounds(double x, double y, double w, double h) Sets bounds of field as rectangular	**void**

CONTROLLERS

Controllers control the motion of body and hence are very handy in creating controlled motion of body. For example if we want a body to move in circle we will have to control its position and we will be using force controller to simulate variable drag force on body.

SimPhy supports following controllers :

- Position Controller
- Velocity Controller
- Angular velocity Controller
- Acceleration controller
- Force Controller
- Torque Controllers

Editing controller

SimPhy **doesn't support creation or removal of controller** programmatically (does it need so?), although you can access already created controller and enable or disable them through script.

For example,

Let there be a force controller already created in SimPhy with name 'f', it can be accessed and modified through script as below :

```
var f = World.getController("f");
f.setEnabled(false);
f.setExpressions("-4x", "0");
f.setEnabled(true);
```

Method Summary

Method and Description	Modifier and Type
`getBody()` Returns body to which force is associated.	`Body`
`getExpressionString()` Returns current expression string.	`String`
`getInstantValue()` Returns instantaneous value.	`Vector2`
`getName()` Returns name/identifier of the controller.	`String`
`getParseErrorMessage()` Returns error in expression, if any.	`String`
`getUnit()` Returns SI unit string for the controller.	`String`
`getxExpr()` Returns expression for x component of controller.	`String`
`getyExpr()` Returns expression for x component of controller.	`String`
`isEnabled()` Returns true, if the controller is enabled.	`boolean`
`isValidExpr()` Returns true, if expression set is parsable.	`boolean`

isVisible() Returns true, if the controller is rendered.	**boolean**
setBody(Body body) Sets the body to which force is applied.	**void**
setEnabled(boolean enabled) Sets controller enabled or disabled. A disable controlller does't affect body	**void**
setExpressions(String xExpr, String yExpr) Set expressions for each component.	**void**
setName(String name) Sets name of the controller, name is used as identifier for controller in script.	**void**
setVisible(boolean visible) Sets controller visible.	**void**

CHAPTER 6 :

USING RESOURCES

The screenshot shows the resources folder containing one resource for each type. All resources in script are accessed by resources object.

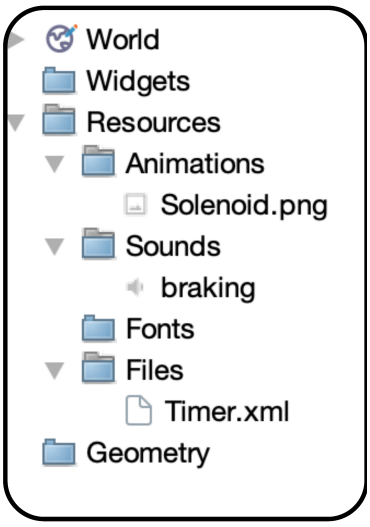
Accessing Resources

```
var anim=Resources.getAnimation("fruit");
body.setAnimation(anim);

var sound=Resources.getSound("bump");
sound.play();

var font=Resources.getFont("Serif24");
var w=font.stringWidth("Measure my
width");

var file=Resources.getFile("level.xml");
Console.println(file.getFileContent());
```



Methods Available for Resources Object

Method and Description	Modifier and Type
`getAllAnimations()` Returns array containing all loaded animations.	`Brush[]`

getAllFonts() Returns array containing names of all loaded fonts.	**Font[]**
getAllSounds() Returns array containing all loaded sounds.	**Audio[]**
getAnimation(String name)	**Brush**
getFile(String name) Return File with the name.	**File**
getFont(String name) Return Font with the name.	**Font**
getSound(String name) Return sound source with the name.	**Audio**
playAnimation(String animName, double x, double y) Plays animation until it stops, centered at point (x,y) in world coordinates.	**void**
playAnimation(String animName, Vector2 v) Plays animation until it stops, centered at position v in world coordinates.	**void**
playSound(String name) Plays sound source once with the name.	**void**
playSound(String name, boolean looping) Plays sound source with the name.	**void**
stopAllSounds() Stops all sounds.	**void**

Note: *There are no methods to create or delete any resource. Resources can be created or deleted from front end only.*

ANIMATION/BRUSH

Method Summary

`createCopy()` Returns copy of brush, ready to be used with other bodies.	`Brush`
`getHeight()` Returns height of brush (for current frame, if animating) in pixels after taking brush scale in account.	`int`
`getName()` Returns name of the brush.	`String`
`getWidth()` Returns width of brush (for current frame, if animating) in pixels after taking brush scale in account.	`int`

SOUNDS

Method Summary

`dispose()` Needs to be called when the audio is no longer needed.	`void`
`getDuration()` Returns the duration in seconds.	`float`

getName() Returns name of the audio.	**String**
getPosition()	**float**
getSampleRate() Returns Sample rate/frequency in Hz.	**int**
getSize() Returns size in KB.	**float**
getVolume()	**float**
isLooping()	**boolean**
isPlaying()	**boolean**
pause() Pauses the play back.	**void**
play() Plays sound with full volume and returns.	**long**
play(float volume) Plays sound with specified volume.	**long**
resume() Resumes sound if previously playing.	**void**
setLooping(boolean isLooping) Sets whether the music stream is looping.	**void**
setPan(float pan, float volume) Sets the panning and volume of this music stream.	**void**

setPosition(float position) Set the playback position in seconds.	**void**
setVolume(float volume) Sets the volume of this music stream.	**void**
stop() Stops a playing or paused music instance.	**void**

FONTS

Method Summary

charHeight(char ch) Returns height of a character.	**int**
charsWidth(char[] data, int off, int len) Returns the total advance width for showing the specified array of characters in this font.	**int**
charWidth(char ch) Returns the advance width of the specified character in this font.	**int**
getAscent() Determines the *font ascent* of the font described by this FontMetrics object.	**int**
getDescent() Determines the *font descent* of the font described by this FontMetrics object.	**int**
getHeight()	**int**

Gets the standard height of a line of text in this font.	
getLeading() Determines the *standard leading* of the font described by this FontMetrics object.	**int**
getMaxAscent() Determines the maximum ascent of the font described by this FontMetrics object.	**int**
getName()	**String**
stringWidth(String str) Returns the total advance width for showing the specified String in this font.	**int**

FILES

Method Summary

getFileContent() Returns text content of the file.	**String**
getName() Returns identifier name of the file.	**String**
getPath() Returns location of file on disc.	**String**
setFileContent(String text) Sets content of file.	**void**
toString()	**String**

CHAPTER 7 :

ACTIONS

There are 3 available actions in SimPhy ready for you :

- Animation Actions
- Composite Actions
- Other Actions

Animation Actions

Animation actions modify various properties of your actor, such as location, rotation, scale and alpha, etc. Here they are:

- **`FadeIn`** : changes alpha of your actor from actor's current alpha to 1
- **`FadeOut`** : changes alpha of your actor from actor's current alpha to 0
- **`FadeTo`** : changes alpha of your actor from actor's current alpha to specific value
- **`MoveBy`** : moves your actor by specific amount
- **`MoveTo`** : moves your actor *to* specific location
- **`RotateBy`** : rotates your actor *by* specific angle
- **`RotateTo`** : rotates your actor *to* specific angle
- **`ScaleTo`** : scales your actor to specific scale factor

Complex Actions

More complex actions can be built by running actions at the same time or in sequence. Use of the *Actions* class makes defining complex actions very easy:

- **Parallel :** execute given actions in parallel - all actions at once
- **Sequence :** execute given actions in sequence - one after another

Other actions

These should be fairly self explanatory :

- **Repeat :** repeats given action n-times
- **Forever :** repeats given action forever
- **Delay :** delays execution of given action for specific amount of time
- **Remove :** removes given Actor from stage
- **RunFunction :** runs specified function (already defined in script) with the actor body passed as an argument

Every action can be created by using object *Actions*.

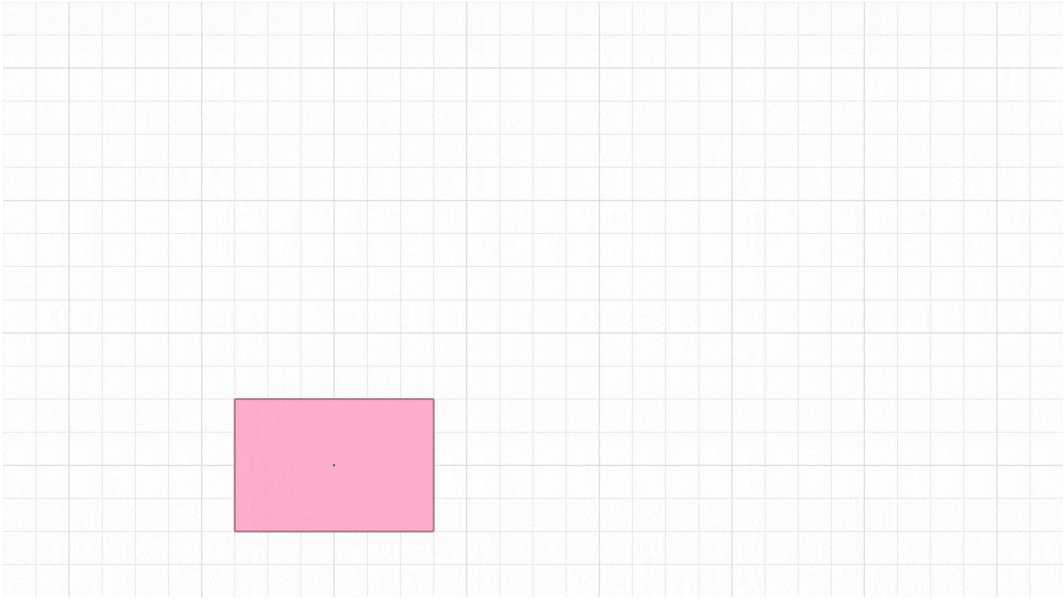
Example of creating animation actions:

```
var move = Actions.moveTo(2, 2, 0.5);
//move Actor to location (2,2) in 0.5 s
var rotate = Actions.rotateTo(6, 0.5);
//rotate Actor to angle 6 radians in 0.5 s
```

Example of creating more complex action sequence:

```
var sequence = Actions.sequence(
          Actions.moveTo(2, 2, 0.5),
           //move actor to 2,2
          Actions.rotateTo(9, 0.5),
           //rotate actor to 9 radians
          Actions.fadeOut(0.5),
           //fade out actor (change alpha to 0)
          Actions.removeActor()
           //remove actor from stage
);

body.setAction(sequence);
```



Interpolator

Animation actions also lets you specify Interpolator commonly known as *tweening*. Interpolation is useful for generating values between two discrete end points using various curve functions and are stored as constant field in actions.

Interpolations can be applied in one of 2 ways:

1. **Specifying interpolation at the time of creation of an action :**

```
var action=Actions.moveBy(4, 0, 2, Actions.bounceIn);
body.setAction(action);
```

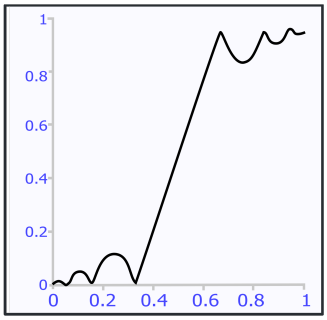
2. **Specifying interpolation at time of creation of action :**

```
var action=Actions.moveBy(4,0,1.5);
action.setInterpolation(Actions.bounceIn);
body.setAction(action);
```

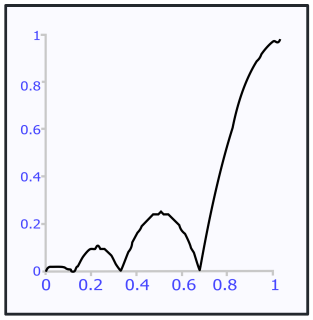
Not working

Interpolation modes:

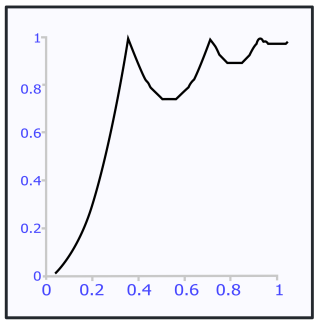
1. `in()`
    - Static in (easing)
    - Runs an easing functions forwards.
2. `out()`
    - Static out (easing)
    - Runs an easing functions backwards.
3. `inOut()`
    - Static inOut (easing)
    - Makes any easing function symmetrical. The easing function will run forwards for half of the duration, then backwards for the other half.



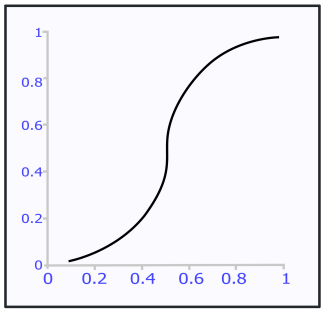
bounce



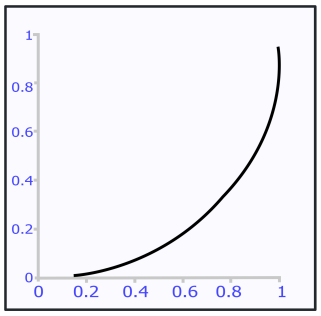
bouncein



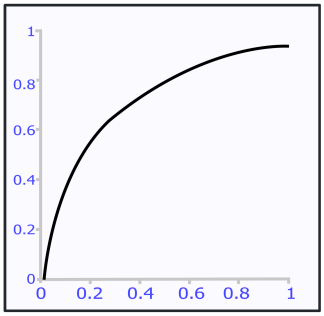
bounceout



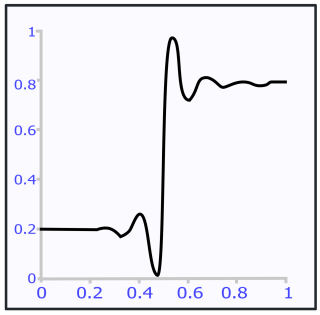
circle



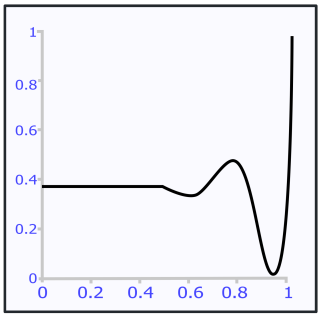
circlein



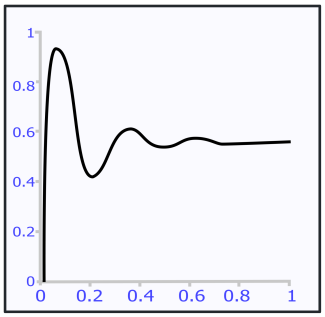
circleout



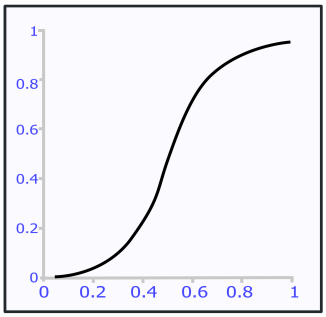
elastic



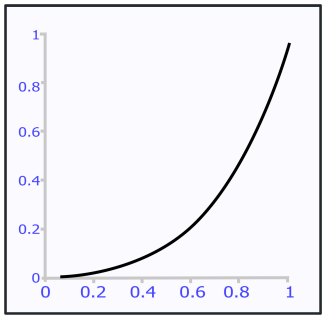
elasticin



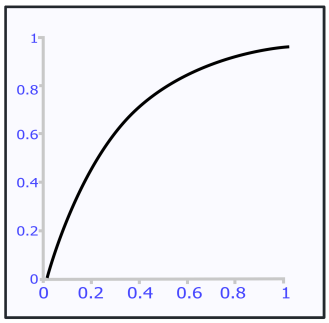
elasticout



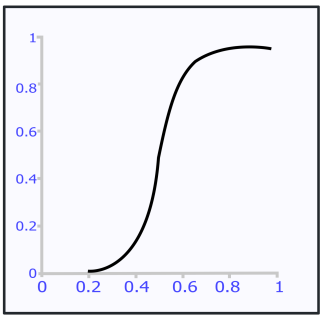
exp5



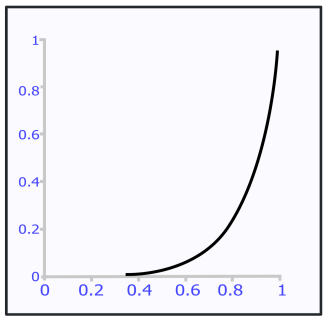
exp5in



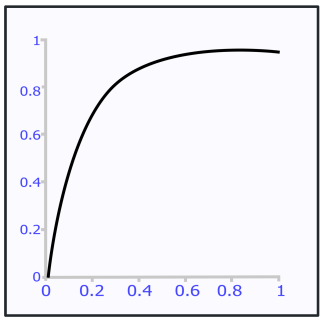
exp5out



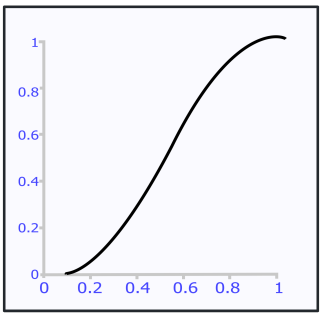
exp10



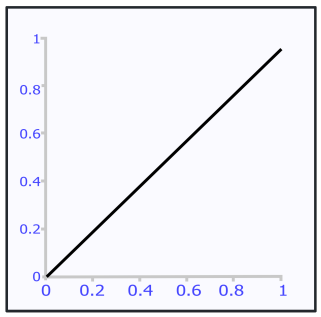
exp10in



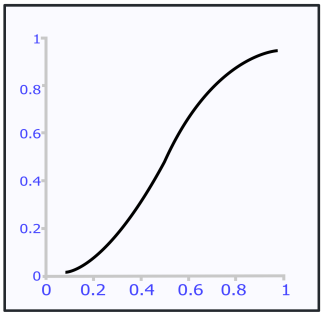
exp10out



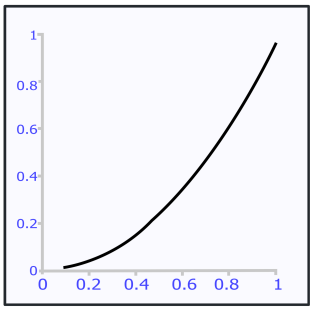
fade



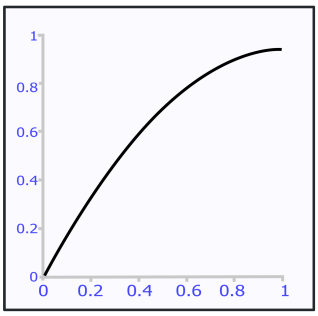
linear



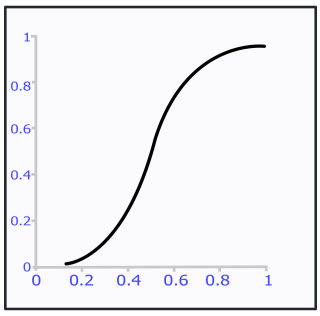
pow2



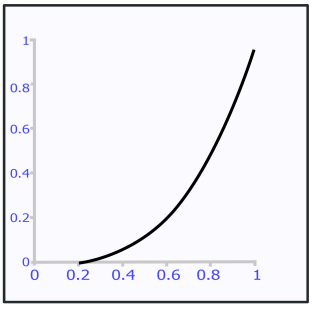
pow2in



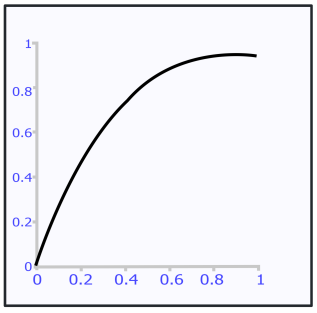
pow2out



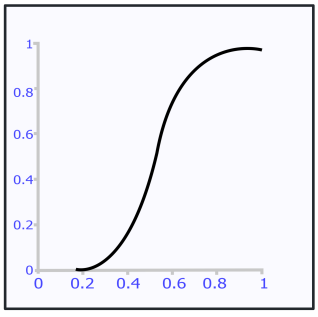
pow3



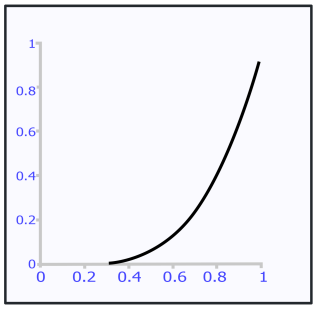
pow3in



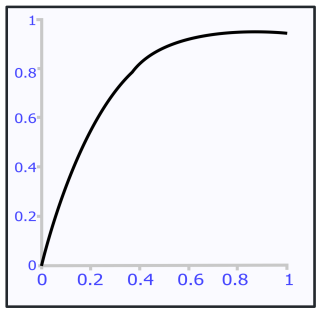
pow3out



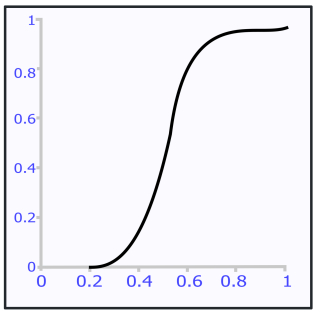
pow4



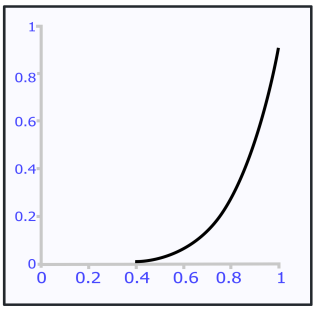
pow4in



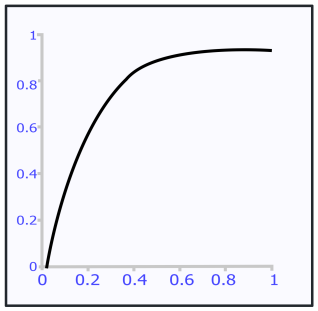
pow4out



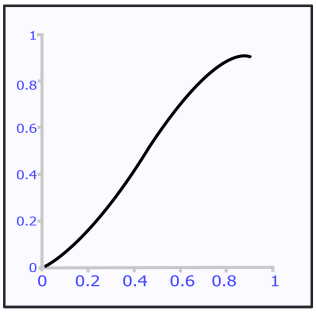
pow5



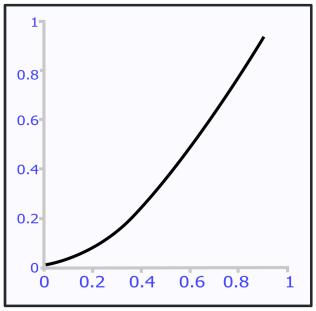
pow5in



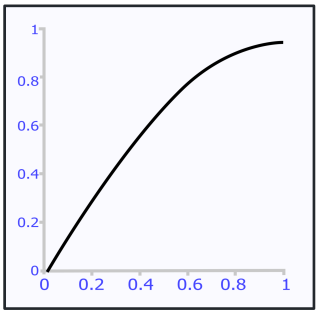
pow5out



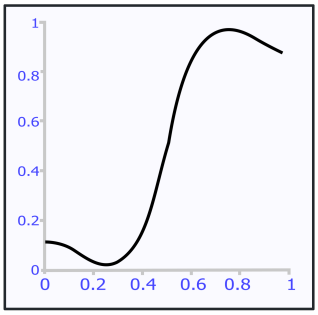
sine



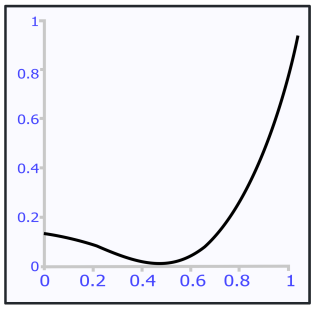
sinein



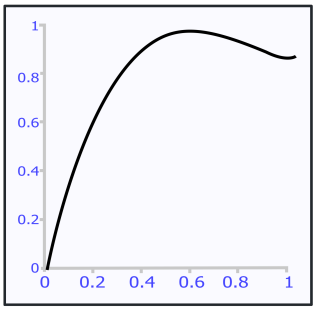
sineout



swing



swingin



swingout

CHAPTER 8 :

PREFERENCES OBJECT

There exist one more special object called **Preferences**, which can be used to retrieve or modify SimPhy settings.

Preferences Object has several methods to get/set *World* or *App* related settings.

For example, to set SimPhy canvas background color we can use :

```
Preferences.setBackGroundColor(new Color("red")
```

Methods Available for Preferences Object

`getBackGroundColor()`	`Color`
`getForceColor(int index)` Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	`Color`
`getForceName(int index)` Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	`String`
`getForceScale()`	`double`
`getForeGroundColor()`	`Color`
`getMaxSignificantFigures()` Returns number of significant figures used in calculations.	`int`

getMinForceDisplayed() Returns min force that can be shown in free body diagrams.	**double**
getMinVelocityDisplayed() Returns min speed that can be shown in tracer.	**double**
getVelocityScale() Returns velocity scale m/(m/s).	**double**
isBodyCenterEnabled() Returns true if the centre of mass should be rendered for bodies.	**boolean**
isBodyChargeDisplayed() Returns true if the body charge should be rendered.	**boolean**
isBodyLabeled() Returns true if body labels should be shown.	**boolean**
isBodyVelocityEnabled() Returns true if body velocities should be rendered.	**boolean**
isElectricFieldDrawn() Returns true if electric field lines are drawn else false.	**boolean**
isForceEnabled(int index) Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	**boolean**
isGravityRendered() Returns true if gravity should be rendered on screen.	**boolean**
isGridEnabled() Returns true if the Grids should be rendered.	**boolean**

isMagneticFieldDrawn() Returns true if magnetic field lines are drawn.	**boolean**
isOriginLabeled() Returns true if the origin and origin label should be shown.	**boolean**
isPositionInfoEnabled() Returns true if position of body is displayed.	**boolean**
isScaleEnabled() Returns true if the scale should be rendered.	**boolean**
isSimulationLocked() Returns false if simulation can be edited by user.	**boolean**
setBackGroundColor(Color backColor)	**void**
setBackGroundColor(Object color) Sets background color of the world.	**void**
setBodyCenterEnabled(boolean flag) Enables or disables the rendering of the centre of mass for bodies.	**void**
setBodyChargeDisplayed(boolean flag) Enables or disables the rendering of body charge.	**void**
setBodyLabeled(boolean flag) Sets the body labels flag.	**void**
setBodyVelocityEnabled(boolean flag) Enables or disables the rendering of body velocity vectors.	**void**
setElectricFieldDrawn(boolean electricFieldDrawn)	**void**

Enables rendering of electric field lines for existing electric fields.	
setElectricFieldForChargeDrawn(boolean electricFieldForChargeDrawn) Returns true if electric field due to charges is rendered else false.	**void**
setForceColor(int index, Color color) Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	**void**
setForceEnabled(int index, boolean enabled) Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	**void**
setForceName(int index, String name) Order "Mg","f","N","T","R","kx","E", "F","Fps","Fcf","Fcori"	**void**
setForceScale(double forceScale)	**void**
setForeGroundColor(Color foreColor) Sets foreground Color of world.	**void**
setForeGroundColor(String cssColor)	**void**
setGravityRendered(boolean flag) Sets the gravity rendered flag.	**void**
setGridEnabled(boolean flag) Enables or disables the rendering of the grid.	**void**
setMagneticFieldDrawn(boolean magneticFieldDrawn) Enables rendering of magnetic field lines for existing magnetic fields.	**void**
setMaxSignificantFigures(int n)	**void**

Sets number of significant figures used in calculations.	
setMinForceDisplayed(double minForceDisplayed) Sets min force that can be shown in free body diagrams.	**void**
setMinVelocityDisplayed(double minvelocity) Sets min speed that can be shown in tracer.	**void**
setOriginLabeled(boolean flag) Sets the origin label flag.	**void**
setPositionInfoEnabled(boolean posInfoEnabled) If enabled, displays the position of body.	**void**
setScaleEnabled(boolean flag) Enables or disables the rendering of the scale.	**void**
setSimulationLocked(boolean simulationLocked) Simulation can be edited by a user, if not locked.	**void**
setVelocityScale(double velocityScale) Sets force scale in m/(m/s).	**void**

CHAPTER 9 :

APP OBJECTS

App Object is a global object present in the script used mainly for any of the following :

- Playing/pausing simulation
- Showing popupBoxes
- Timing events

Playing / Pausing a Simulation

To pause a simulation use :

```
App.setPaused(true)
```

To run a simulation use :

```
App.setPaused(false)
```

To find paused state of a simulation use :

```
App.isPaused()
//returns false if simulation is running else true
```

Showing Popup Boxes

SimPhy has three kind of popup boxes: **Alert box, Confirm box, and Prompt box**.

1. Alert Box

An alert box is often used if you want to make sure information comes through the user. When an alert box pops up, the user will have to click "OK" to proceed. Code execution is blocked until user clicks a button or close this dialog.

There are three methods in *App* Object to create Alert boxes based on alert type :

- **showMessageBox**(String title, String message)

  Shows message box with information icon/style to the user.

- **showWarningMessageBox**(String title, String message)

  Shows message box with warning icon/style to the user.

- **showErrorMessageBox**(String title, String message)

  Shows message box with error icon/style to the user.

Message can be in html format (may contain html3 tags) or just plain text.

For Example, the following code displays an alert box shown below :

```
App.showWarningMessageBox( "Legal Warning", "<html><h2
color='red'>No Hunting !</h2>Wildlife population have
plummeted by <b>60%</b> since 1970.<br> Those found guilty
are liable to <span color='red'>imprisonment of 2 years
</span> along with penalty of <span
color='red'>$5000</span>.")
```

2. Confirm Box

A confirm box is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "Yes", "OK" or "Cancel" to proceed.

The return value of the function is integer and its value depends on the option chosen by the user.

- **`showOkCancelBox`**`(String title, String message)`

  Shows modal dialog to the user to choose Ok or Cancel Option. Returns 0 if *OK* is pressed, 1 if *Cancel* is pressed and -1 if dialog is cancelled.

- **`showYesNoBox(`**`String title, String message)`

  Shows modal dialog to the user to choose Yes or No action. Returns 0 if *Yes* is pressed, 1 if *No* is pressed and -1 if dialog is cancelled.

- **`showYesNoCancelBox`**`(String title, String message)`

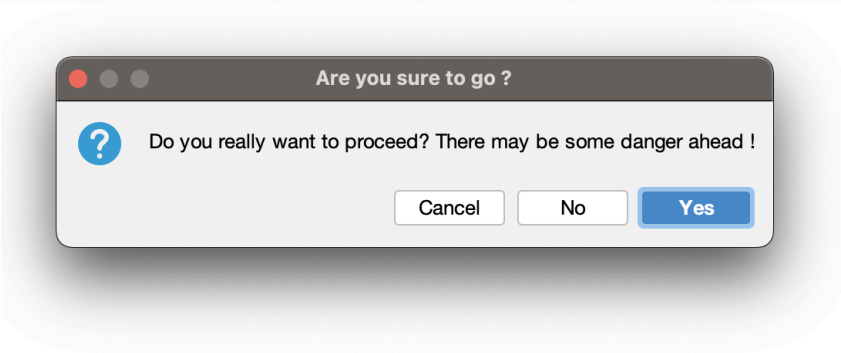
  Shows modal dialog to the user to choose Yes, No or cancel action.

Returns 0 if *Yes* is pressed, 1 if *No* is pressed, if *Cancel* is pressed and -1 if dialog is cancelled.

Once again, message can be in html format (may contain html3 tags) or just plain text.

For Example, the following code displays an alert box shown below :

```
var response=App.showYesNoCancelBox("Are you sure to go ?",
"Do you really want to proceed? There may be some danger
ahead !")
```



3. Prompt Box

A prompt box is often used if you want the user to input a value before entering a page. When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.

If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

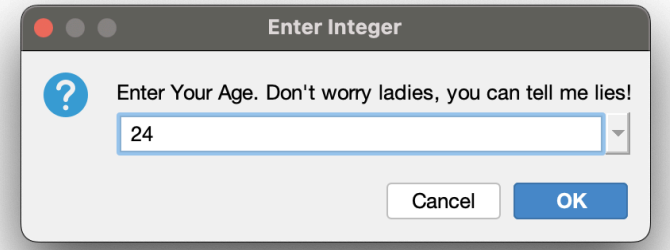
The following are available methods which are self explanatory, each method displays dialog with suitable GUI element to choose from.

Method	Return Value
readChoice(String message, String[] choices) Shows a box containing choices (displayed as radio button) from which user can choose exactly ne choice	**int**
readChoices(String message, String[] choices) Shows a box containing choices (displayed as checkbox) from which user can choose none, one or more choice	**int[]**
readDouble(String message) Shows input box where user can input an integer	**double**
readExpression(String message) Shows input box where user can input an expression	**String**
readFromList(String message, String[] choices) Shows a box from which user can select an option from predefined options(shown as buttons)	**String**
readInteger(String message) Shows input box where user can input an integer	**double**
readString(String message) Shows input box where user can input a string	**String**

Once again, message can be in html format (may contain html3 tags) or just plain text.

For Example, the following code displays an alert box shown below :

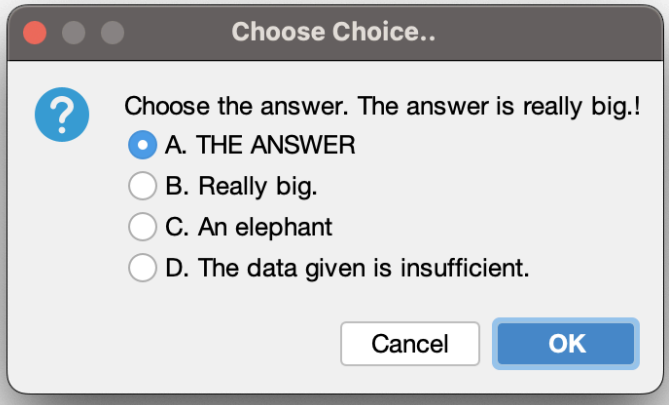
```
var age= App.readInteger("Enter Your  Age. Don't worry
ladies, you can tell me lies! ")
```



Take another Example, the following code displays a confirm box shown below :

```
var choices = [ "A. THE ANSWER", "B. Really big.",
"C. An elephant","D. The data given is insufficient." ];

App.readChoice("Choose the answer. The answer is really
big.! ", choices)
```



Scheduling : `setTimeOut` and `setInterval`

We may decide to execute a function not right now, but at a certain time later. That's called "scheduling a call".

There are two methods for it:

- **`App.setTimeout`**

  Allows a function to run once after the interval of time.

1. Syntax

```
var timer = App.setTimeout(func, delay, args)
```

func : Function name to execute which should be globally declared in script.

delay : The delay before run, in milliseconds (1000 ms = 1 second), default value is 0.

args : Array of arguments for the function (optional, can be null)

*For instance, this code open message dialog with text **Hello Reader** after one second:*

```
function sayHi(to) {
     App.showMessageBox("Introduction", "Hello " + to)
}
App.setTimeout(sayHi("Reader"), 1000);
```

2. Cancelling with `clearTimeOut()`

A call to `setTimeout` returns a "timer identifier" timer that we can use to cancel the execution.

The syntax to cancel:

```
var timer = App.setTimeout(...);
App.clearTimeout(timer);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = App.setTimeout(sayHi, 1000);
App.clearTimeout(timerId);
```

- **`App.setInterval`**

Allows a function to run regularly with the interval between the runs.

1. **Syntax**

The setInterval method has the same syntax as setTimeout:

```
var timerId = App.setInterval(func, delay, args)
```

All arguments have the same meaning.

2. **Cancelling with `clearTimeOut()`**

To stop further calls, we should call `App.clearInterval(timerId)`.

For example, To create a digital clock using `Button` and `app.setInterval()`:

- Create a button and rename it to 'Button'.
- Add the following code to script and save.
- Run simulation and press button to start timer (button will start acting as digital clock).

```
var timer = null;

// add action to button to start/reset timer
Widgets.getButton("Button").setAction(startTimer())
App.clearAllTimers();

function startTimer() {
   // stop timer if already running
   if (timer != null)App.clearInterval(timer);
   timer = App.setInterval(tick, 500);
}

function tick() {
   var today = new Date();
   //get current date
   //retrieve hour, minute and second from date object
   var h = today.getHours();
   var m = today.getMinutes();
   var s = today.getSeconds();
   m = checkTime(m);
   s = checkTime(s);
   Widgets.getButton("Button").setText(h + ":" + m + ":" +
s);
}

/** add zero in front of numbers if < 10 */
function checkTime(i) {
   if (i < 10) {
       i = "0" + i
   }
   return i;
}
```

Works when startTimer() called in Properties table

- ## Recursive `setTimeOut()`

  Recursive `setTimeout()` is another way of running something regularly.

For example, if we want to call tick function regularly after every 2000 ms :

```
/** instead of:
var timerId = setInterval(tick, 2000);
*/

var timerId = App.setTimeout(tick,2000);

function tick(){
Console.println("ticked");
 timerId = App.setTimeout(tick, 2000);
}
```

The setTimeout above schedules the next call right at the end of the current one (*).

The recursive setTimeout is a **more flexible** method than setInterval. This way the next call may be scheduled differently, depending on the results of the current one.

Recursive setTimeout **guarantees a delay** between the executions whereas setInterval does not.

And if the functions that we're scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.

For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds...

Here's the pseudocode:

```
let delay = 5000;

let timerId = setTimeout(function request() {
   ...send request...
   if (request failed due to server overload) {
       // increase the interval to the next run
       delay *= 2;
   }
   timerId = setTimeout(request, delay);
}, delay);
```

PART IV :

CANVAS 2D API

CHAPTER 1 :

BASIC SETUP

GUI Widget element **canvas** gives you an easy and powerful way to draw graphics using scripting. It can be used to draw graphs, make photo compositions, or do simple (and not so simple) animations.

Basic Setup

- Create a dialog box on the *world* and then add a canvas element to it (see figure-1).

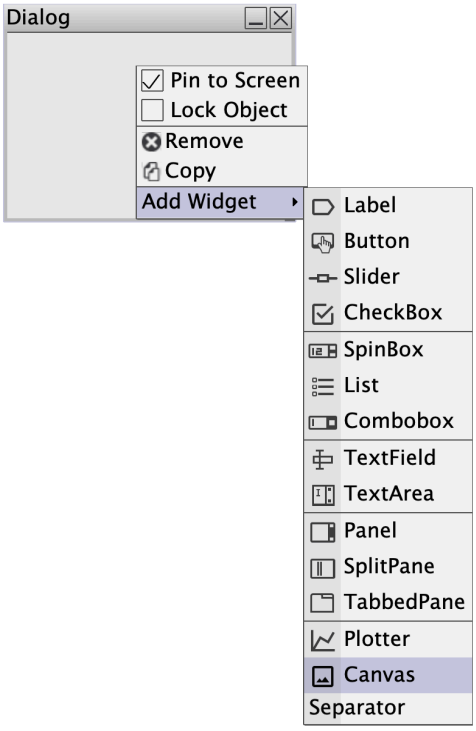


Figure - 1

(Property)	(Value)
General [be...	
name	canvas
enabled	☑
text	Canvas Demo
visible	☑
Display	
background	☐
bgimage	
border	☑
font	default-normal
foreground	■
Layout	
colspan	1
halign	fill
height	60
rowspan	1
valign	fill
weightx	1
weighty	1
width	100
Actions	
bean	org.shikhar.si...
property	

Figure - 2

- Set width and height plus all the core attributes like name, weights background etc. through the property editor (see figure-2). Remember the name attribute (which is “canvas” by default).

- Open Script Editor and insert the following code to create a variable referencing the above canvas widget:

```
var canvas = Widgets.getCanvas("canvas");
```

- The canvas is initially blank, and to display something, a script first needs to access the rendering context and draw on it. The canvas element has a method called getContext(), used to obtain the rendering context and its drawing function.

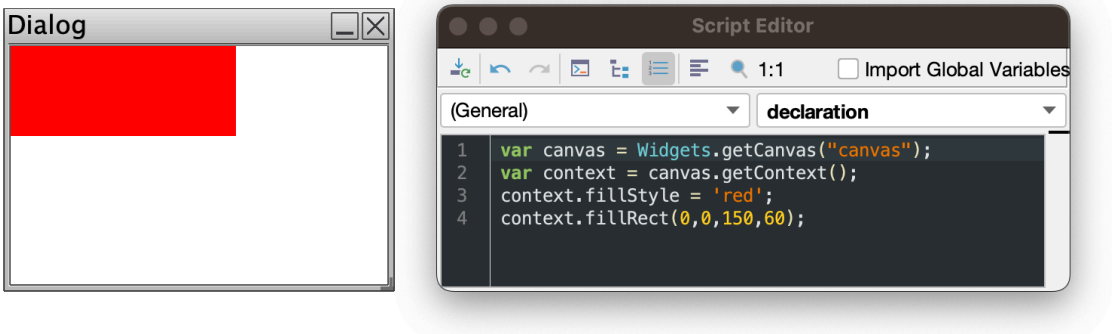
```
var context = canvas.getContext();
```

Each canvas element can only have one context. If we use the getContext() method multiple times, it will return a reference to the same context object.

Now we have everything prepared to do the real drawings by changing and adding stuff to the 2D context named *context*. That means either getting or setting context properties or calling context methods can be done in this manner :

```
context.fillStyle = 'red';
context.fillRect(0,0,150,60);
```

Now save the script, you will see a red rectangle drawn on your canvas.



CHAPTER 2 :

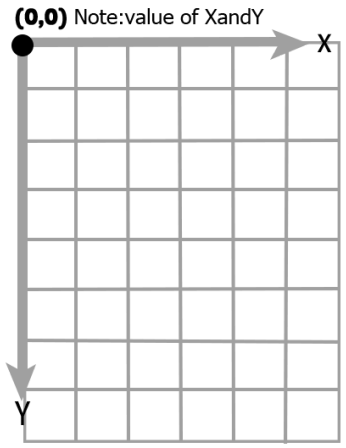
BASIC DRAWING

Canvas Coordinate System

The coordinate system is one of the most important pieces of knowledge you need to know before drawing on the canvas. If you have used any other 2D graphics programming language before, then you will be familiar with the standard Cartesian coordinate system with the (0,0) at the top left corner and everything counts from the top left.

For example,

If you want to move your rectangle to the right, you will have to increase your x-axis value; if you want to move down, you will have to increase your y-axis value.



To get an idea of coordinates on canvas (generally used for debugging) let's develop utility functions that render a grid on canvas which automatically updates when canvas resizes. It also renders mouse coordinates whenever the cursor moves.

Simple Graphics

The canvas is initially blank and to display something, you will need the help of a script to draw it for you. You can draw basic elements like ovals, rectangles, and text to create simple graphics.

- ### Drawing Rectangles

  There are three methods that draw rectangles on the canvas :

  1. `fillRect(x, y, width, height)`

     This method draws a filled rectangle.

  2. `strokeRect(x, y, width, height)`

     This method draws a rectangular outline.

  3. `clearRect(x, y, width, height)`

     This method clears the specified area and makes it fully transparent.

  Here, x and y specify the position on the canvas (relative to the origin) of the top-left corner of the rectangle and width and height are width and height of the rectangle.

- ### Drawing Ovals / Circles

  There are two methods that draw ovals/circles on the canvas :

  1. `fillOval(x, y, width, height)`

     This method draws a filled oval.

  2. `strokeOval(x, y, width, height)`

     This method draws an oval outline.

Here x and y specify the position on the canvas (relative to the origin) of the center of oval, and width and height are width and height of the oval.

To render a circle centered at (x, y) and radius r use.

```
context.fillOval(x, y, 2*r, 2*r)
```

- ## Drawing Texts

There are two methods to render texts on the canvas :

1. `fillText(text, x, y, [maxWidth])`

   Writes the given text string at point (x,y) in “filled letters”.

2. `strokeText(text, x, y, [maxWidth])`

   Writes the given text string at point (x,y) in “stroke letters.

The actual rendering of the text is also dependent on the current values of the following properties:

- *font*
- *textAlign* for the horizontal and
- *textBaseline* for the vertical text alignment

By default, text is rendered from leftmost char at x and its baseline at y. To render center aligned text at (x, y) use :

```
//set text style
context.textAlign="center";
context.textBaseLine="middle";
context.font="default-large";

//perform actual rendering
context.fillText("Hello World",x,y);
```

For detailed text styling, refer chapter 5.

- ## Drawing Images

  First of all, what you need to do is to create a variable called ‘img’ and declare it as an image object.

```
var img=new Image("imageName");
```

  where 'imageName" is the name of image already imported (just by dragging and dropping the image) in SimPhy.

  There are two methods to render image on the canvas :

  1. `drawImage(image, dx, dy)`

     Position the image on the canvas.

  2. `drawImage(image, dx, dy, dWidth, dHeight)`

     Position the image on the canvas and specify width and height of the image.

  3. `drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)`

     Clip the image and position the clipped part on the canvas.

  `dx` : The x coordinate for positioning the image's leftmost side on canvas.

  `dy` : The y coordinate for positioning the image's topmost side on canvas.

  `dWidth` : The original image scales itself to fit into the specified *width*.

  `dHeight` : The original image scales itself to fit into the specified *height*.

  `sx` : The x coordinate from where the clipping begins.

  `sy` : The y coordinate from where the clipping begins.

  `sWidth` : The width of the clipped image.

  `sHeight` : The height of the clipped image.

- ## Changing Fill and Stroke Colors

  If you want to apply colors to a shape, there are two important properties you can use:

  1. `Fillstyle = color`

     Sets the style used when filling shape.

  2. `strokeStyle = color`

     Sets the style for shapes' outlines.

  Here, color is a string representing a CSS, a gradient object, or a pattern object. We'll look at gradient and pattern objects later. By default, the stroke and fill color are set to black (CSS color value #000000)

  Note: *When you set the* `strokeStyle` *and/or* `fillStyle` *property, the new value becomes the default for all shapes being drawn from then on. For every shape you want in a different color, you will need to reassign the* `fillStyle` *or* `strokeStyle` *property.*

- ## Clearing Canvas

  1. ### Using dedicated method `context.clear()`

     This is the preferred way to remove all content graphics from canvas, keeping current styles (strokeStyle, fillStyle, font, linewidth etc) and transforms (scale, rotation etc) unchanged.

  2. ### Using method `context.reset()`

     This method clears content of canvas as well as resets all setting, styles and transforms to default.

3. **Using method `context.clearRect(x, y, w, h)`**

   This method clears content of any rectangular section of the canvas without affecting style and transform.

   Note : The rectangular region described depends on current transformation (see chapter 5 for information).

For example, to clear whole canvas with default transformations use:

```
// Clear the entire canvas
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

Another example:

```
var context=Widgets.getCanvas("canvas").getContext();
context.clear();
context.drawImage(new Image("train.jpg"),0,0);
context.fillRect(355,0,100,40);
context.strokeRect(465,0,100,40);
context.fillOval(405,70,100,40);
context.strokeOval(515,70,100,40);
context.clearRect(50, 50, 50, 50);
```

The output will be :

CSS Colors

- Hexadecimal colors, specified as `#RRGGBB`, where RR denotes the red, GG is the green and BB the blue component of the color and each of these six letters stands for a hexadecimal value, i.e. one of 0, ..., 9, A, ..., F.

  Ex : `#0000FF` has no (= 00) red, no green, and full (= FF) blue components, so, this is pure blue.

- RGB colors work the same way as hexadecimal colors do, but their syntax is different, namely `rgb(R, G, B)` where R, G and B values for the red, green and blue components, respectively. But here, the values are either decimal integers between 0 and 255 (including), or percentage values between 0% and 100%.

  Ex : `rgb(0, 0, 255)` is the color with no red, no green and full blue components. So this is pure blue. `rgb(0%, 0%, 100%)` is also a pure blue.

- RGBA colors are specified by the form `rgba(R, G, B, A)`, where the R, G, B part is the same as in RGB colors. The alpha parameter A specifies the opacity and is a value between 0.0 (fully transparent) and 1.0 (fully opaque).

  Ex : `rgba(100%, 0%, 0%, 0.5)` is a purely red color, which is half transparent.

- HSL colors has the form `hsl(H, S, L)`, specifying the hue, saturation and lightness for a cylindrical-coordinate representation of colors. Hue H is a degree on the color wheel, from 0 to 360, where 0 (or 360) is red, 120 is green, and 240 is blue. Saturation S is a percentage value from 0% to 100%, where 0% means a shade of gray and 100% is the full color. Lightness L is also a percentage, 0% is black and 100% is white.

  Ex : `hsl(120, 65%, 75%)`

- HSLA colors has the form `hsla(H,S,L,A)`, where H, S and L are the same as in HSL colors and the alpha parameter A defines the opacity, from 0.0 (for fully transparent) to 1.0 (fully opaque).

  Ex : `hsl(120, 65%, 75%, 0.5)`

- Predefined or cross-browser color names are colors in HTML and CSS specified by their name, such as *Violet* or *Blue*.

  There are 17 standard colors: *Aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver , slate, white and yellow.*

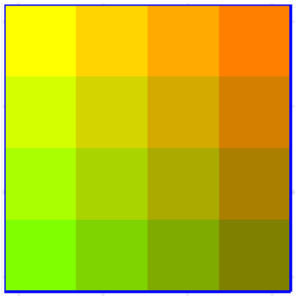
For example:

```
var canvas =Widgets.getCanvas("canvas");
var context=canvas.getContext();

//draw a grid of 16 blocks each in each in a different
color
for (var i = 0; i < 4; i++){
   for (var j = 0; j < 4; j++) {
     context.fillStyle = 'rgb(' + Math.floor(255 - 42.5 *
i) + ', ' + Math.floor(255 - 42.5 * j) + ', 0)';
     context.fillRect(j * 25, i * 25, 25, 25);
   }
}

//draw border with blue color
context.strokeStyle="blue";
context.strokeRect(0, 0, 100, 100);
```

The outcome will be as shown :



CHAPTER 3 :

COMPLEX DRAWING

Complex drawing (other than rectangles and ovals) is done using path. A path is a sequence of lines and curves. The 2D canvas interface takes a peculiar approach to describing such a path. It is done entirely through a sequence of method calls to describe its shape.

Each context has its own path object which is directly modified by using path methods of context. But we can create our own Path2D object which can be used to cache drawing commands. Let's see each in detail.

Using Context Path Methods

With `strokeRect()` and `fillRect()` we can draw (stroke or filled) rectangles. But in most cases, we would also like to draw other and more complex shapes. This can be done in a stepwise fashion by building paths. The standard procedure of creating any complex shape is thus

- The first step to create a path is to call the **`beginPath()`**. Internally, paths are stored as a list of sub-paths (lines, arcs, etc) which together form a shape. Every time this method is called, the list is reset and we can start drawing new shapes.

- Create a sequence of primitive steps, in each one moving from the current point in the path to a new one. Possible primitive steps are: moving to a new point (with `moveTo()`), create a straight line to a new point (with `lineTo()`), a rectangular box (with `rect()`) or a somehow curved line (with `quadraticCurveTo()`, `bezierCurveTo()`, `arc()` or `arcTo()`.

- Optionally, close the path with **`closePath()`**, i.e. draw a line from the current terminal point to the initial point again.

Once the path has been created, you can stroke or fill the path to render it with either `stroke()` or `fill()`.

1. moveTo() / lineTo()

```
context.moveTo(x, y), context.lineTo(x, y)
```

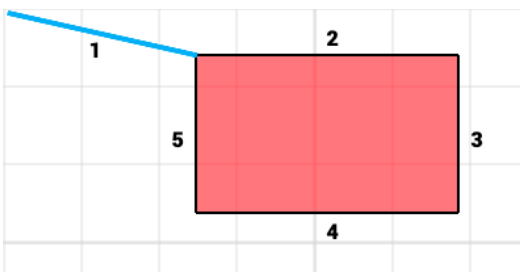
For example, redefine `strokeRect()` and `fillRect()` by the path method.

```
var canvas=World.getCanvas();
var context=canvas.getContext();

context.fillStyle = 'rgba(255,0,0,0.5)';
// set the style for the fill() call below
context.strokeStyle = 'rgba(0,0,0,1)';

context.beginPath();
// 0. start a new path
context.moveTo(80, 20);
// 1. moves the current point to (80,20)
context.lineTo(180, 20);
// 2. horizontal line from (80,20) to (180,20)
context.lineTo(180, 80);
// 3. vertical line from (180,20) to (180,80)
context.lineTo(80, 80);
// 4. horizontal line from (180,80) to (80,80)
context.lineTo(80, 20);
// 5. vertical line from (80,80) back to the start at (80,20)
context.fill();
// 6. fill the solid figure with the shape of the path lines
context.stroke();
// 7. stroke the solid figure outline
```

Generates this picture:



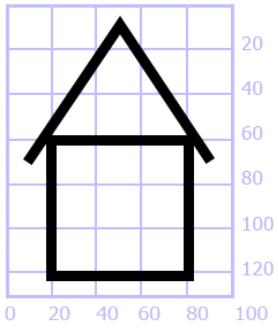
Note: With each `moveTo()` step (here, step 1 is indicated by the cyan line), the current point of the path is only moved to the new indicated position. Whereas each `lineTo()` step (lines in black) actually draws a line from the previous position to the new indicated point. After the path is outlined, a call of `fill()` creates a filled object in `fillStyle` inside the given lines. In our case, this results in the filled light red rectangle.

2.rect()

```
context.rect(x, y, w, h)
```

Above code creates a rectangle with the left upper corner at (x,y), which is w pixels wide and h pixels high.

Now, suppose we want to draw a little house :



We could either construct a path of six separate lines, or we can built it by creating one square (with `rect()`) and a roof (two `lineTo`() calls) like so :

```
context.beginPath();     // start a new path
context.rect(20, 60, 60, 60);
// create the square body of the house
context.moveTo(10, 70); // create the roof: 1. move to the
left
context.lineTo(50, 10); // 2. create the left line
context.lineTo(90, 70); // 3. create the right line
context.strokeStyle = "black";
// draw the house: 1. set the strokeStyle to 'black'
context.lineWidth = 5.0;
// 2. increase the default lineWidth to 5.0
context.stroke();          // 3. do the actual drawing
```

3. quadraticCurveTo()

```
context.quadraticCurveTo(cpx,cpy,p2x,p2y);
```

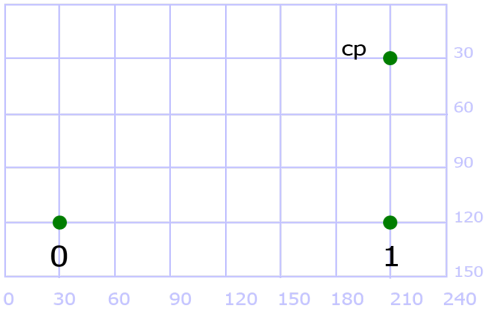
Adds a *Quadratic Bézier curve* to the current sub-path. It requires two points: the first one is a **control point (say cp)** and the second one is the **end point (say 1)**. The starting point is the latest point in the current path (say 0), which can be changed using `moveTo()` before creating the Quadratic Bézier curve.

Consider this code snippet :

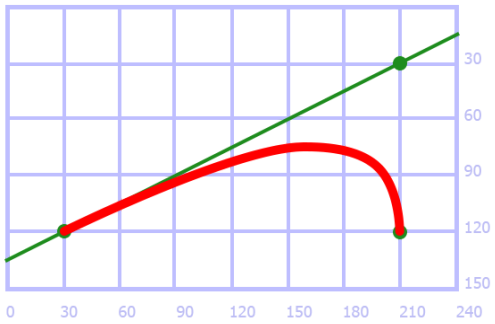
```
context.lineWidth = 5.0;
context.strokeStyle = 'red';
context.beginPath();
context.moveTo (30,120);
context.quadraticCurveTo (210, 30, 210, 120);
context.stroke();
```

Generates this picture:

With `moveTo(30,120)` the current point 0 is located at (30,120). By calling `quadraticCurveTo(210,30,210,120)`, the control point cp is (210,30) and the new point 1 is (210,120).



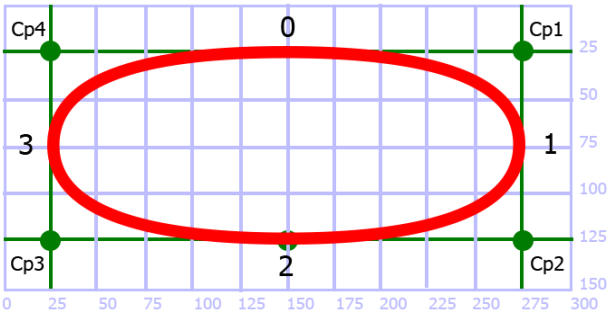
The red curve starts in **0** and moves towards **cp** (geometrically speaking: the line from **0** to **cp** is a tangent to the curve at point **0**), and then bends towards **1** (again: the line from **cp** to **1** is a tangent to the curve at point **1**).



For example, let's draw an ***oval shape:***

```
context.strokeStyle = 'red';
// set the color to 'red' for the stroke() call below
context.beginPath();
// start the new path
context.moveTo (150,25);
// move to point 0 at (150,25)
context.quadraticCurveTo (275,25,275,75);
// curve from point 0 to point 1 at (275,75)
context.quadraticCurveTo (275,125,150,125);
// curve from point 1 to point 2 at (150,125)
context.quadraticCurveTo (25,125,25,75);
// curve from point 2 to point 3 at (25,75)
context.quadraticCurveTo (25,25,150,25);
// curve from point 3 back to point 0
context.stroke();
```

Generates this output :



4.bezierCurveTo()

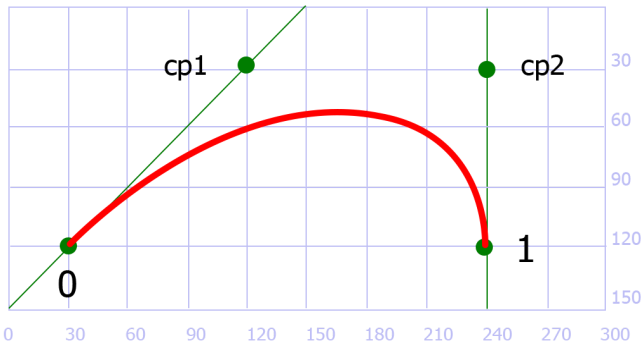
```
context.bezierCurveTo(cp1x, cp1y, cp1x, cp1y, x, y);
```

Creates a curved line from the current point **0** in the path to the new point **1** at (x,y). The two control points **cp1** at (cp1x,cp1y) and **cp2** at (cp2x,cp2y) determine the actual shape of the line: Leaving from **0** the curve first heads into the direction of **cp1**, and finally entering into **1**, the curve comes out of the direction of **cp2**.

Let us consider the following snippet :

```
context.beginPath();
context.moveTo(30, 120);
// go to point 0 at (30,120)
context.bezierCurveTo(120, 30, 240, 30, 240, 120);
// curve from point 0 to
// point 1 at (240,120)
context.stroke();
```

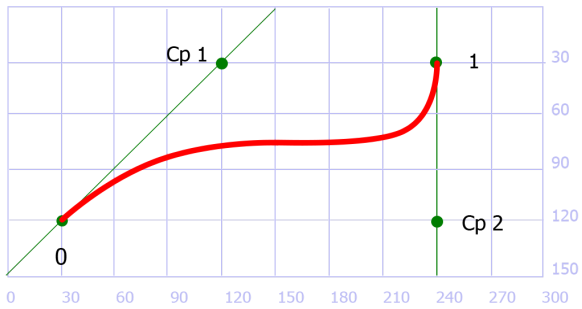
Generates this output :

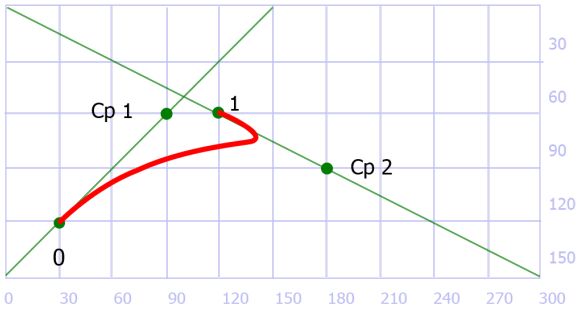


The curve starts in point **0** towards **cp1** and terminates in point **1** coming from **cp2.**

The same principle holds when we change the position of the involved points.

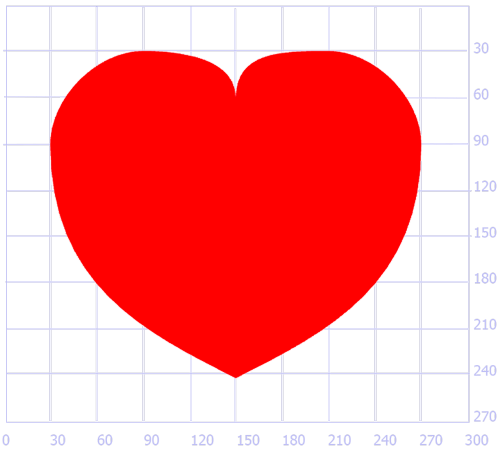
Such as :



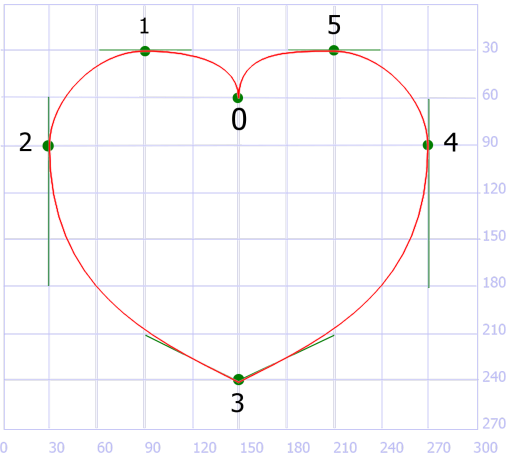
,


*For example: Let's draw a **heart!***

The whole theory on Beziér curves was developed in car factories for the practical purpose of creating smooth auto bodies. But finding the code for a given shape by hand can be a daunting task. Let us exercise a straight forward method on how this can be done. Suppose we try to find the code for say the shape of a heart.



We start by identifying a couple of significant points **0, 1, 2, 3, 4, 5,** and the tangents (displayed in green), i.e. the lines that touch the red shape at that particular point:



We finally just gather the pieces and end up with the code for the original canvas picture :

```
context.fillStyle = 'red';
context.beginPath();
context.moveTo (150,60);
// start at point 0
context.bezierCurveTo (150,30, 100,30, 90,30);
// from point 0 to point 1
context.bezierCurveTo (60,30,30,60,30,90);
// from point 1 to point 2
context.bezierCurveTo (30,180,90,210,150,240);
// from point 2 to point 3
context.bezierCurveTo (210,210,270,180,270,90);
// from point 3 to point 4
context.bezierCurveTo (270,60,240,30,210,30);
// from point 4 to point 5
context.bezierCurveTo (180,30,150,30,150,60);
// from point 5 to point 0
context.fill();
```

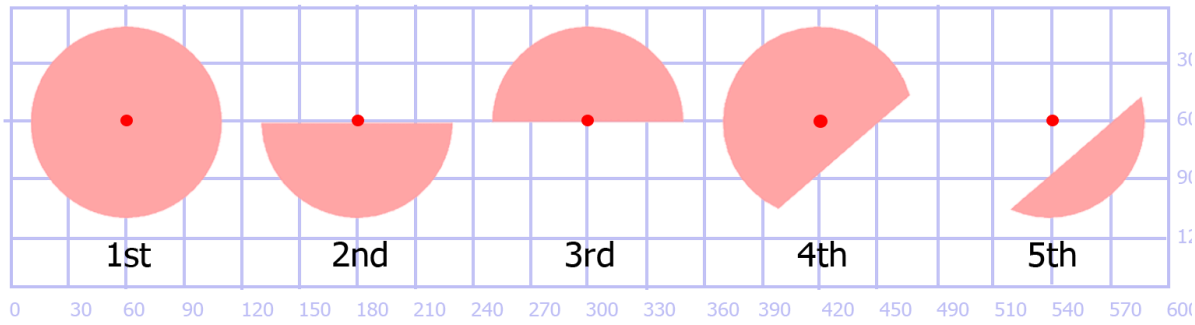
5.arc()

```
context.arc (x, y, r, start, end, anticlockwise)
```

Defines a piece of a circle. The center of this circle is **(x,y)**, the radius is r. The **start and end point** of the arc are given as **angles** in radians. The optional **boolean anticlockwise** parameter defines if the arcs are measured anticlockwise (value true) or clockwise (value false, which is the default). A call of `arc()` is part of a path declaration, the actual drawing is done with a call of `stroke()`, drawing a piece of the circle line, or `fill()`, which draws a section of the circle disk.

For example:

Let's draw sections of five circle disks, each one with a radius of 50 pixels.



which is generated by this code snippet :

```
context.fillStyle = "rgba(255,0,0,0.33)";
// red color with 1/3 transparency

// now draw five filled circle pieces:
context.beginPath();
context.arc ( 60, 60, 50, 0, 2 * Math.PI, false);
context.fill();
// 1st circle
context.beginPath();
context.arc (180, 60, 50, 0, Math.PI, false);
context.fill();
// 2nd circle
context.beginPath();
context.arc (300, 60, 50, 0, Math.PI, true );
context.fill();
// 3rd circle
context.beginPath();
context.arc (420, 60, 50, 2, 6, false);
context.fill();
// 4th circle
context.beginPath();
context.arc (540, 60, 50, 2, 6, true );
context.fill();
// 5th circle
```

And if we replace `fillStyle()` by `strokeStyle()` and each occurrence of `fill()` by `stroke()`, we obtain this picture of five circle arcs:

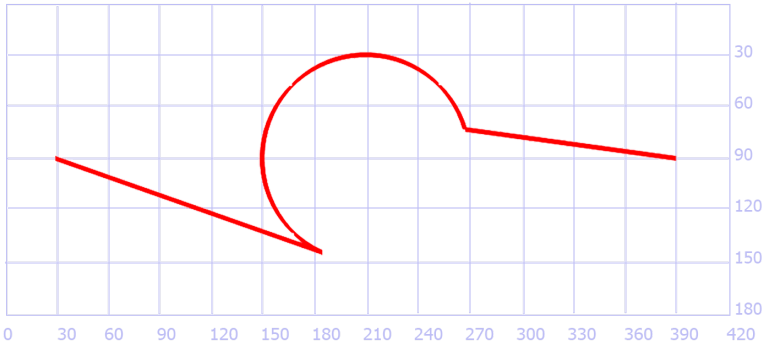
For example: **`arc()` as a piece in the current path.**

A call of `arc()` is part of the current path such that it depends on the current point before it was called, and the end point of the arc becomes the new current point afterwards.

For example, this code :

```
context.lineWidth = 3.0;
context.strokeStyle = "red";
context.beginPath ();
context.moveTo (30, 90);
context.arc (210, 90, 60, 2, 6);
context.lineTo (390, 90);
context.stroke();
```

Generates this output :



6.arcTo()

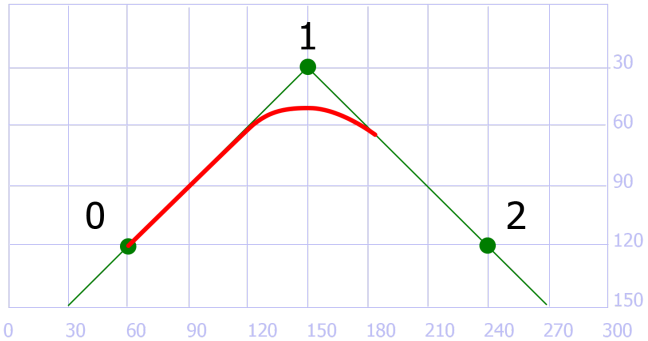
```
context.arcTo (x1, y1, x2, y2, radius)
```

Draws an arc with the given radius, depending on the current point in the path and the two specified points **(x1, y1)** and **(x2, y2)**. For more details of this method, see the following examples and description.

```
context.beginPath();
context.lineWidth=5;
context.strokeStyle='red';
context.moveTo (60, 120);
context.arcTo (150, 30, 240, 120, 50);
context.stroke();
```

Generates this output :

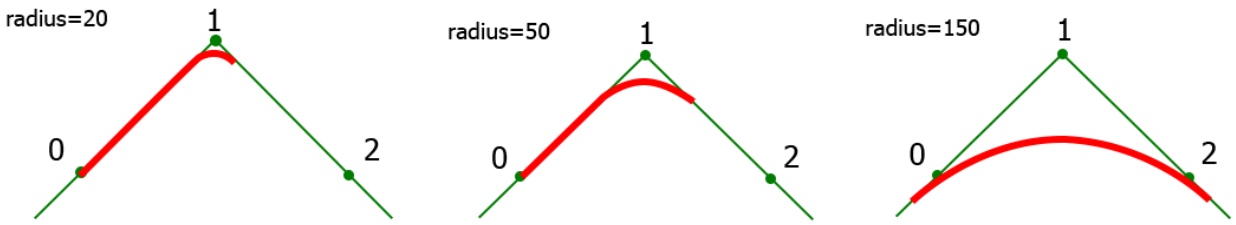
Here, the `arcTo (x1, y1, x2, y2, radius)` draws a line that has its origin in point 0, runs towards point 1, makes a turn towards point 2, so that radius is the actual radius of the arc. The ending point of the `arcTo()` line is the point, where the arc meets the tangent line (from point 1 towards point 2).



The whole idea probably becomes more obvious, if we use the same code snippet and just vary the radius :

```
context.beginPath();
context.moveTo (60, 120);
context.arcTo (150, 30, 240, 120, radius);
// different values for radius
context.stroke();
```

The resulting pictures are these:



7.isPointInPath()

```
context.isPointInPath (x, y)
```

Returns true, if the specified point (x,y) is in the current path, and false otherwise.

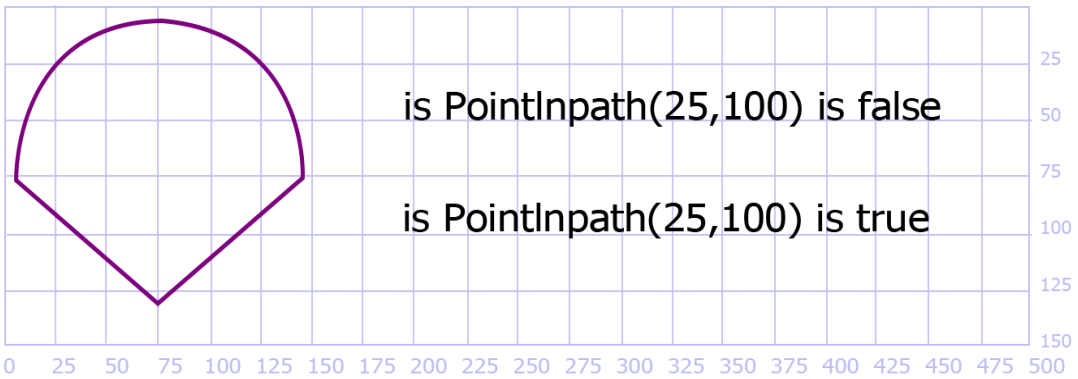
For example:

```
// add a new path
context.beginPath();
context.moveTo (75,130);
// make (75,130) the current point
context.lineTo (145,75);
// line from (75,130) to (145,75)
```

Continued on the next page…

```
context.arc (75,75,70,0,Math.PI,true);
//draw half circle disk with center (75,75), radius 70 and
counterclockwise
context.lineTo (75,130);
// line from (5,70) to (75,130)
context.lineWidth = 3.0;
// set the line width for the stroke drawing
context.strokeStyle = 'purple';
// set the line color for the stroke drawing
context.stroke();
// draw the shape // determine the position of two points
var answer1 = context.isPointInPath (25,100);
// answer1 is now either true or false
var answer2 = context.isPointInPath (100,25);
// answer2 is also either true or false
context.font = "14pt sans-serif";
context.fillText ( "isPointInPath(25,100) is " + answer1,
200, 50);
context.fillText ( "isPointInPath(100,25) is " + answer2,
200, 100);
// print out the result on the canvas
```

Generates this canvas picture :



Using Path2D Object

The Path2D object is used to cache drawing commands and replay them or test them for hits as required.
You can either create new **Path2D** objects and then use the API to specify the path (eg. `path.moveTo(), path.lineTo()` etc.) or you can feed the constructor with another *Path2D* object and the resultant object will be a combination of the two.

The *Path2D* interface of the Canvas API is used to declare a path that can then be used on a Context object. The path methods of the Context interface are also present on this interface, which gives you the convenience of being able to retain and replay your path whenever desired.

- ### Creating Path2D Objects

  `Path2D():` Path2D constructor

  Creates a new Path2D object.

  `Path2D(Path2d path):` Path2D constructor

  Creates a new Path2D object from a path object. All of the initial geometry and the winding rule for this path are taken from the specified Shape object.

- ### Path2D Methods

  `Path2D.beginPath()`

  The `beginPath()` function can be used to "reset" the canvas path - throwing away whatever the current path is.

Path2D.addPath(Path path)

Adds a path to the current path.

Path2D.closePath()

Causes the point of the pen to move back to the start of the current sub-path. It tries to draw a straight line from the current point to the start. If the shape has already been closed or has only one point, this function does nothing.

Path2D.moveTo()

Moves the starting point of a new sub-path to the (x, y) coordinates.

Path2D.lineTo()

Connects the last point in the subpath to the (x, y) coordinates with a straight line.

Path2D.bezierCurveTo()

Adds a Cubic Bézier curve to the path. It requires three points. The first two points are control points and the third one is the end point. The starting point is the last point in the current path, which can be changed using `moveTo()` before creating the Bézier curve.

Path2D.quadraticCurveTo()

Adds a Quadratic Bézier curve to the current path.

Path2D.arc()

Adds an arc to the path which is centered at (x, y) position with radius r starting at `startAngle` and ending at `endAngle` going in the given direction by anticlockwise (defaulting to clockwise).

`Path2D.arcTo()`

Adds a circular arc to the path with the given control points and radius, connected to the previous point by a straight line.

`Path2D.rect()`

Creates a path for a rectangle at position (x, y) with a size that is determined by width and height.

`Path2D.contains(x, y)`

Returns true if a certain point (x,y) in canvas space is (top left as origin) in the current path.

`Path2D.addText(text,x,y,font)`

Appends shape/glyph from text to this path at (x,y) using given font.

- ## Rendering Path2D objects

Path2D objects can be drawn using methods `fill()` and `stroke()` of context object.

`context.stroke(path)`

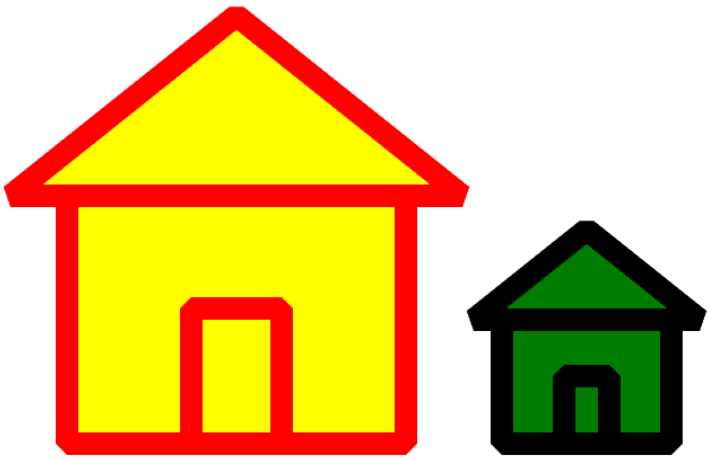
Draws shape outline using current stroke style.

`context.fill(path)`

Fills shape using current fill style.

For example:

Lets draw pair of houses shown below using same path object, and transformations.



```
var canvas = Widgets.getCanvas("canvas");
var context=canvas.getContext();
// Set line width
context.lineWidth = 10;
var path=new Path2D();
// Wall
path.rect(75, 140, 150, 110);
// Door
path.rect(130, 190, 40, 60);
// Roof
path.moveTo(50, 140);
path.lineTo(150, 60);
path.lineTo(250, 140);
path.closePath();
//draw Big Yellow house
context.fillStyle="yellow";
context.fill(path);
context.strokeStyle="red";
context.stroke(path);
//draw small Green house
context.scale(0.5, 0.5);
context.translate(460, 250);
context.fillStyle="green";
```

CHAPTER 4 :

COLOR, LINE AND TEXT STYLES

Change Color Styles

So far everything that we created has been filled in black color. To customize it, *context* provides the following two important properties to apply colors to a shape:

S.No.	Method and Description
1	`fillStyle()` This attribute represents the color or style to use inside the shapes.
2	`strokeStyle()` This attribute represents the color or style to use for the lines around shapes.

By default, the *stroke* and *fillColor* are set to black which in CSS color value is #000000. There are three kinds of possible style values :

1. CSS colors

A **CSS color** is either a color name (such as 'green' or 'aqua') or a hexadecimal color (e.g. `'#00FF00'`). If we want the filled objects to be green, we set the canvas property in this manner :

```
context.fillStyle = '#008000';
//or alternatively and with the same effect, using a color
name instead of the hexadecimal color value
context.fillStyle = 'green';
//or alternatively and with the same effect, using a rgb or
hsl color function
context.fillStyle = 'rgb(0,255,0)';
```

2. Gradients

A gradient can be created to define colors that are not constant, but gradually change on the area. If we want to set the style this way, we first need to create an object that implements the opaque Canvas Gradient interface, which allows us to call the `addColorStop()` method.

- **Linear Gradient**

  For example:

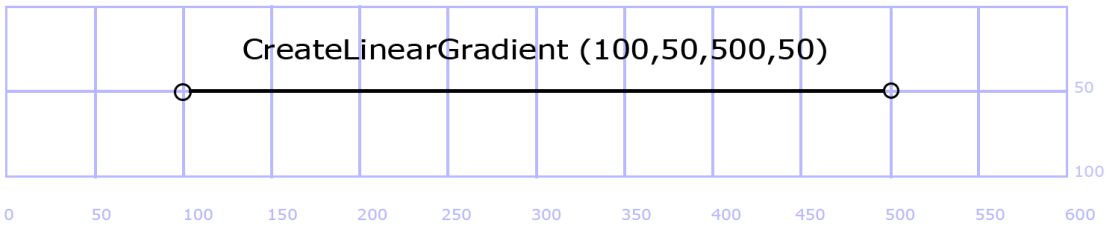
```
var canvas = Widgets.getCanvas("canvas");
var context = canvas.getContext();
// 1. create a linear gradient
var rainbowGradient = context.createLinearGradient(100, 50,
500, 50);
// 2. add colors to the gradient
rainbowGradient.addColorStop(0, 'red');
rainbowGradient.addColorStop(0.25, 'yellow');
rainbowGradient.addColorStop(0.5, 'green');
rainbowGradient.addColorStop(0.75, 'blue');
rainbowGradient.addColorStop(1, 'violet');
// 3. set the fillStyle to this new gradient
context.fillStyle = rainbowGradient;
// 4. now draw some filled objects; in this case just a
rectangle
context.fillRect(10, 70, 600, 100);
context.font = "Arial 44px bold";
context.fillText("Text drawn in Gradient Style", 10, 50);
```

Explanation of the above example:

The whole process is more complicated than just assigning a color (say 'green') to `fillStyle`. Instead, we take the following steps:

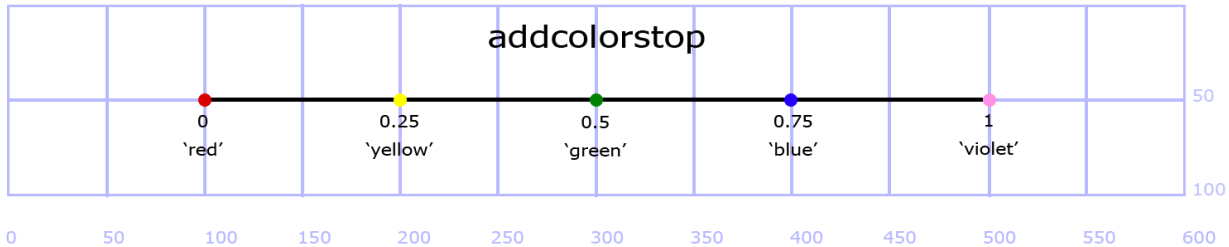
- By calling `createLinearGradient (100, 50, 500, 50)` function, we create a linear gradient (named `rainbowGradient`) from starting point **(100,50)** to end point **(500,50)**.

  That is indicated by the line in the middle of the canvas.

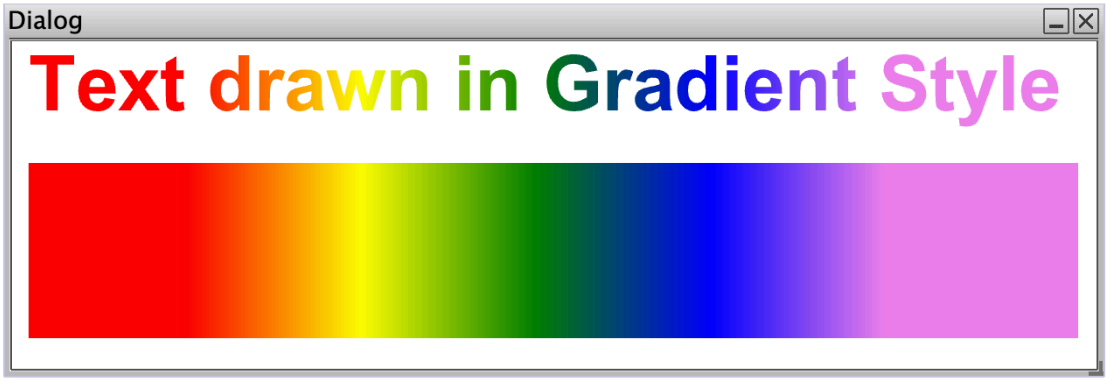


- In the next step we add five colors to the gradient by calling `rainbowGradient.addColorStop(position, color)` five times. Each time, position is a value between 0 and 1, which is the relative distance between the start and end point.

  Instead of color names such as 'red', 'yellow' etc. we could have taken any CSS color string.



- In the third step, we assign this newly created linear gradient (`rainbowGradient`) as a value to `fillStyle`, and when we draw any filled objects, these objects are colored according to the `rainbowGradient` specification.



- **Radial Gradient**

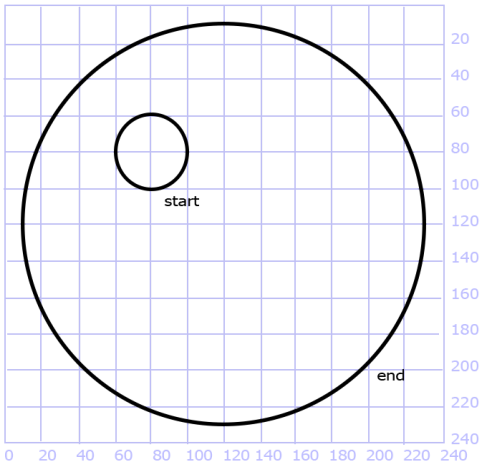
  For example:

```
var canvas=Widgets.getCanvas("canvas");
var context = canvas.getContext();
// 1. create a radial gradient
var rg = context.createRadialGradient(80, 80, 20, 120, 120,
110);
// 2. add colors
rg.addColorStop(0, 'yellow');
rg.addColorStop(1, 'red');
// 3. set the fill style to the new gradient
context.fillStyle = rg;
// 4. now draw some filled objects; in this case just a
circle
context.beginPath();
context.arc(120, 120, 110, 0, 2 * Math.PI, false);
context.closePath();
context.fill();
```

The steps are as follows:

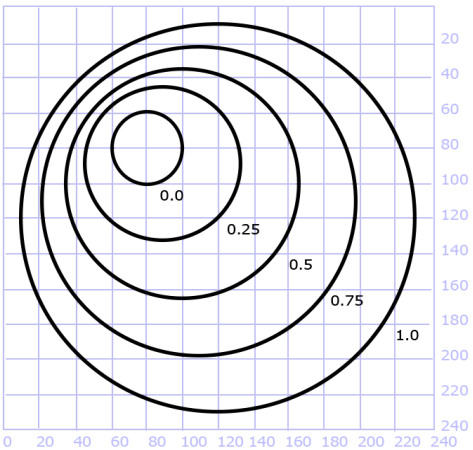
- We start with the creation of a radial gradient, by calling `createRadialGradient(80, 80, 20, 120, 120, 110)` function.

  The arguments of this call are the parameters for two circles : The start circle (x_0,y_0,r_0) = (80,80,20) with center point (80,80) and radius 20; and the end circle (x_1,y_1,r_1) = (120,120,110) and radius 110.

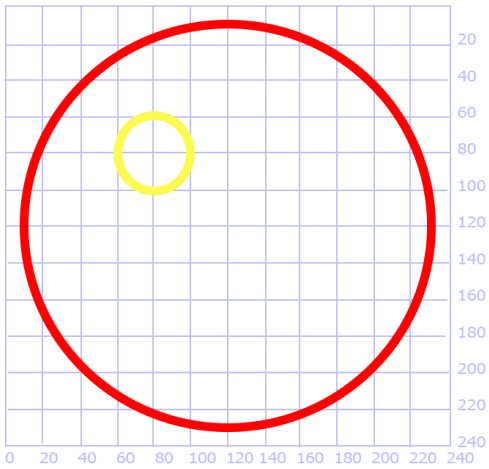


  The process of gradually changing from start circle to end circle can again be numbered into a continuous process from position 0.0 to position 1.0.

  For example, the five positions **0.0, 0.25, 0.5, 0.75** and **1.0** are

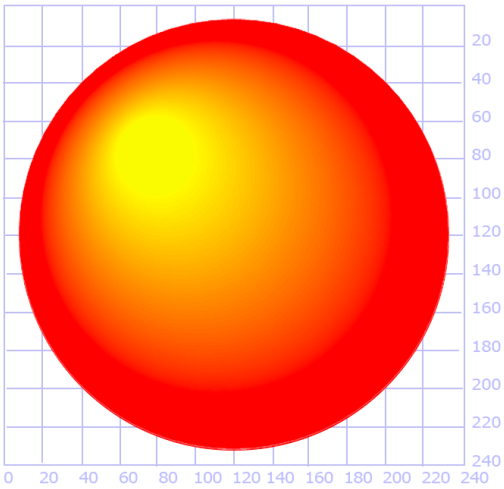


- We now add color stops to the gradient. In our case, just **'yellow'** at start position **0.0** and **'red'** at end position **1.0**.



- The gradient **rg** is now assigned as a style value to `fillStyle()`, so that all subsequently drawn filled objects take this "color" rg.

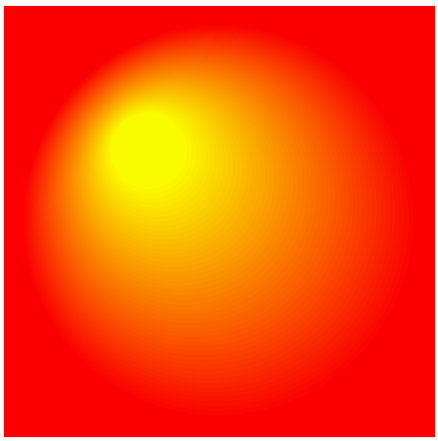
- We finally draw filled objects, in this case just one filled circle, which covers exactly the end circle of the radial gradient.



If we want to draw a filled rectangle over the size of the whole canvas instead of the circle, then replace the code in step 4 with this line :

```
context.fillRect (0, 0, 240, 240);
```

And the result would be this :



3. Patterns

The *patternFill* is created with `createPattern (image, repetition)` function which returns a **CanvasPattern** object and can be used as a value for a style property.

The `image` argument must be an **Image** object already loaded in SimPhy.

The `repetition` argument must be one of the following values:

- `'repeat-x'` (for horizontal repetition of the image),
- `'repeat-y'` (for vertical repetition of the image),
- `'repeat'` (for both horizontal and vertical repetition), or
- `'no-repeat'` (for neither repetition).

The default value is `'repeat'`.

For example, let us use this JPG file called “swimmer.jpg” as a pattern image for a *CanvasPattern* object and use this object as a value for the *fillStyle* property. Subsequently, we draw some *filled* objects: with `fillRect()`, a `fill()` path, and some `fillText()`.

Here's the full code snippet :

```
var canvas=Widgets.getCanvas("canvas");
var context = canvas.getContext();
// create an Image object from the swimmer.jpg file
var swimmer = new Image (swimmer.jpg');
// create a CanvasPattern object with this image and make
that the new fillStyle value
var swimmerPattern = context.createPattern (swimmer,
'repeat');
context.fillStyle = swimmerPattern;
// Now draw some objects: first a rectangle
context.fillRect (0, 0, 200, 210);
// then a triangle
context.beginPath();
context.moveTo (220, 0);
context.lineTo (220, 100);
context.lineTo (550, 50);
context.lineTo (220, 0);
context.fill();
context.closePath();
// and finally write "Hello!"
context.font = '120px sans-serif';
context.fillText ('Hello!', 210, 200);
```

You will see the following output :

Change Line Styles

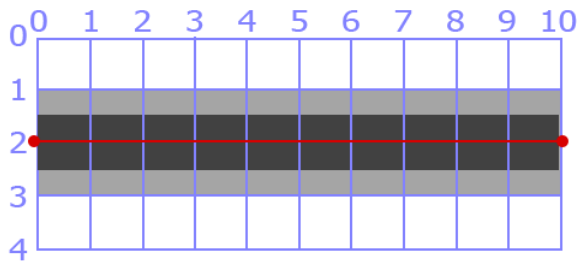
Next to the general style settings, there are special style properties defining the rendering of lines, their caps and joins; namely `lineWidth, lineCap` and `lineJoin`. And to limit the tips of pointed angles, there is also `miterLimit`.

1. lineWidth

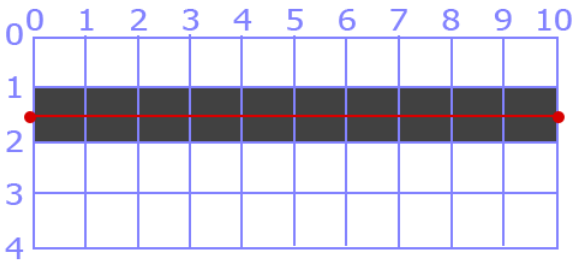
This method holds a float number defining the current line thickness (in coordinate space units). The default value is **1.0.**

```
context.lineWidth=2;
```

In the following enlarged canvas example, a line from (0, 2) to (10, 2) covers 20 half pixels. These affected pixels are therefore rendered gray. The line from (0, 1.5) to (10, 1.5) on the other hand fully fits into the affected ten pixels, the line is therefore rendered crisp and black.



black line of width 1.0 from (0,2) to (10,2)



black line of width 1.0 from (0,1.5) to (10,1.5)

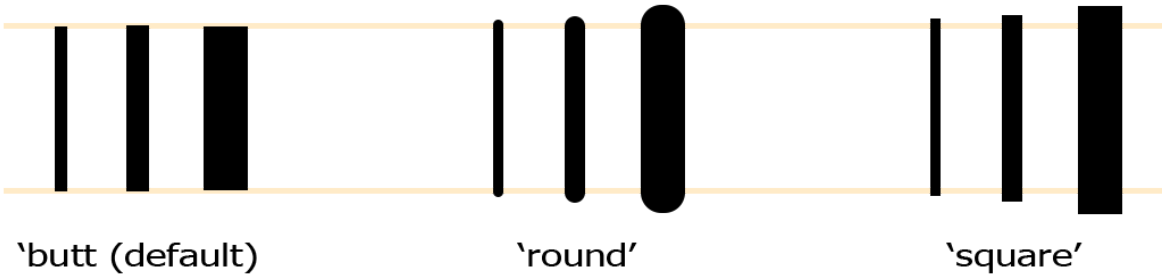
2. lineCap

This method defines the style of line endings. Possible values are `'butt'`, `'round'` or `'square'`. The default is `'butt'`.

```
context.lineCap="round";
```

In the following example, we draw three lines three times from the top to the bottom orange line. The `lineWidth` is set to 5.0, 10.0, and 20.0, respectively. The `lineCap` is set to `'butt'`, `'round'` and `'square'`, as explained in the picture.

Note : The lines with the round and square line caps exceed their ending points by half of the line width.



3.lineJoin

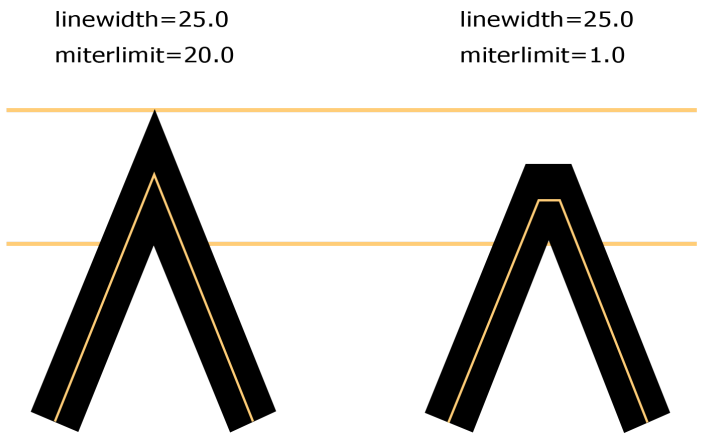
This method defines the style of lines at their meeting point. Possible values are `'round'`, `'bevel'` or `'miter'`. The default is `'miter'`.

```
context.lineJoin = "bevel";
```

4.miterLimit

The miter length is the distance from the point, where the join of two lines occurs to the intersection of the line edges on the outside of the join. The miter limit ratio, set by `miterLimit` with default value **10.0**, is the maximum allowed ratio of the miter length to half the line width. If the miter length would cause the miter limit ratio to be exceeded, the corner will be displayed as if `lineJoin` is set to `bevel`.

```
context.miterLimit=5
```



For example:

```
var context = Widgets.getCanvas("canvas").getContext();
context.reset();
context.clear();

drawLineJoins();
drawLineEnds()

function drawLineJoins() {
   var lineJoin = [ 'round', 'bevel', 'miter' ];
   context.lineWidth = 12;
   for (var i = 0; i < lineJoin.length; i++) {
       context.lineJoin = lineJoin[i];
       context.beginPath();
       context.moveTo(-5, 15 + i * 40);
       context.lineTo(35, 55 + i * 40);
       context.lineTo(75, 15 + i * 40);
       context.lineTo(115, 55 + i * 40);
       context.lineTo(155, 15 + i * 40);
       context.stroke();
   }
}
```

Continued on the next page…

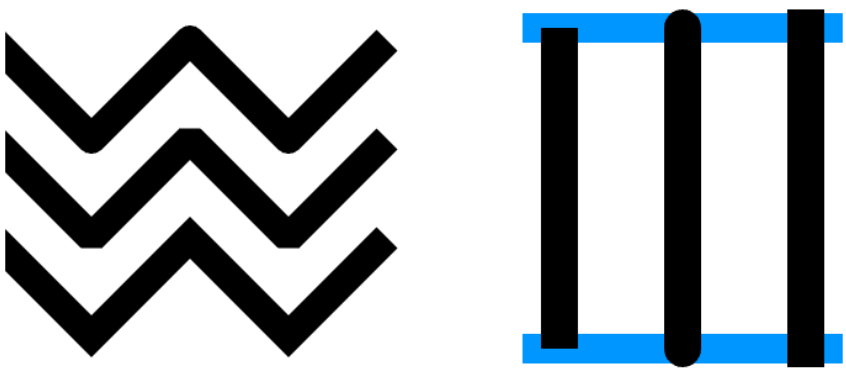
```
function drawLineEnds() {
   var lineCap = [ 'butt', 'round', 'square' ];

   // Draw guides
   context.strokeStyle = '#09f';
   context.beginPath();
   context.moveTo(210, 10);
   context.lineTo(340, 10);
   context.moveTo(210, 140);
   context.lineTo(340, 140);
   context.stroke();

   // Draw lines
   context.strokeStyle = 'black';
   for (var i = 0; i < lineCap.length; i++) {
       context.lineWidth = 15;
       context.lineCap = lineCap[i];
       context.beginPath();
       context.moveTo(225 + i * 50, 10);
       context.lineTo(225 + i * 50, 140);
       context.stroke();

   }
}
```

The output will be :



Change Text Styles

1. textAlign

This method sets the horizontal alignment of text (similar to the CSS text-align property). Its values is one of the following: `'start'`, `'end'`, `'left'`, `'right'` or `'center'`. The default value is `'start'`.

For example:

```
context.textAlign = "center";
context.textAlign = 'left';
context.fillText ("Hello world!", 300, 30);
```

This is the output :

2. textBaseLine

This method sets the vertical alignment of text and has one of the following values: `'top'`, `'hanging'`, `'middle'`, `'alphabetic'`, `'ideographic'` or `'bottom'`. The default value is `'alphabetic'`.

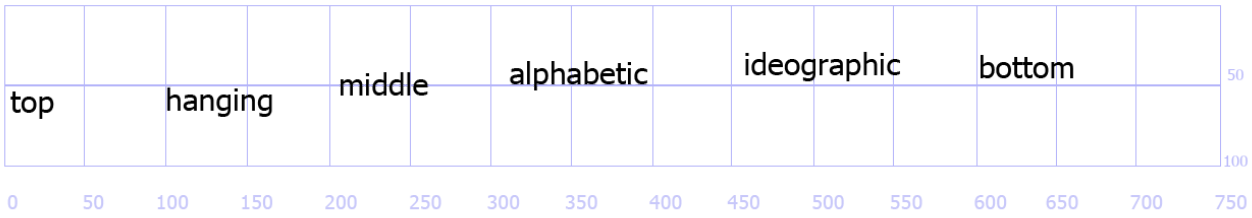
For example:

```
context.font = '20px monospace';
context.textBaseline = 'top';
context.fillText ( "top", 0, 50);
context.textBaseline = 'hanging';
context.fillText ( "hanging", 100, 50);
```

Continued on the next page…

```
context.textBaseline = 'middle';
context.fillText("middle", 200, 50);
context.textBaseline = 'alphabetic';
context.fillText("alphabetic", 300, 50);
context.textBaseline = 'ideographic';
context.fillText("ideographic", 450, 50);
context.textBaseline = 'bottom';
context.fillText("bottom", 600, 50);
```

Generates following outcome :



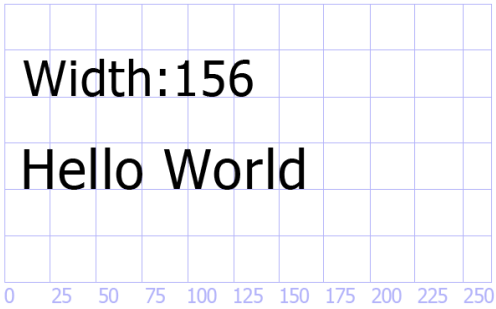
3.measureText

This method returns an object that contains the width and height of the specified text, in pixels.

For example:

```
context.font = "30px Arial";
var txt = "Hello World"
context.fillText("width:" + context.measureText(txt).width,
10, 50)
context.fillText(txt, 10, 100);
```

Generates outcome :



Note: You can also use `measureText()` method to measure text of font object.

```
font = Resources.getFont("default-large");
Console.show();
var tm=font.measureText("Hello World");
Console.println("text Width="+tm.width);
Console.println("text Height="+tm.height);
```

CHAPTER 5 :

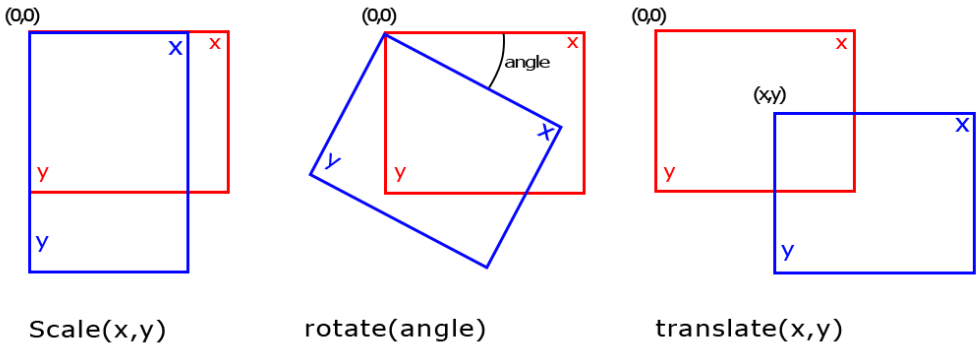
TRANSFORMATIONS

getAllAnimations() Returns array containing all loaded animations.	**Brush[]**
getAllFonts() Returns array containing names of all loaded fonts.	**Font[]**
getAllSounds() Returns array containing all loaded sounds.	**Audio[]**
getAnimation(String name)	**Brush**
getFile(String name) Return file with the name.	**File**
getFont(String name) Return Font with the name.	**Font**
getSound(String name) Return sound source with the name.	**Audio**
playAnimation(String animName, double x, double y) Plays animation until it stops, centered at point (x,y) in world coordinates.	**void**
playAnimation(String animName, Vector2 v)	**void**

Plays animation until it stops, centered at position v in world coordinates.	
playSound(String name) Plays sound source once with the name.	**void**
playSound(String name, boolean looping) Plays sound source with the name.	**void**
stopAllSounds() Stops all sounds.	**void**

In general geometry, a **transformation** turns a given object into another object, but preserves its structure. Things may appear deformed, but they are not ruptured or destroyed. The absolute position of points may change, but the relative positions remain, neighboring points are still neighbors after the transformation.

Three special transformations can be called by special methods on the canvas context:



`Scale()` resizes the canvas, `rotate()` turns it around the origin, and `translate()` shifts it to a new position. But actually, and different to what the three previous images suggest, the canvas as an object in the brower window is **not** moved by any of these operations, but rather its coordinate system is altered. This will be clarified in the later part of this chapter with many examples.

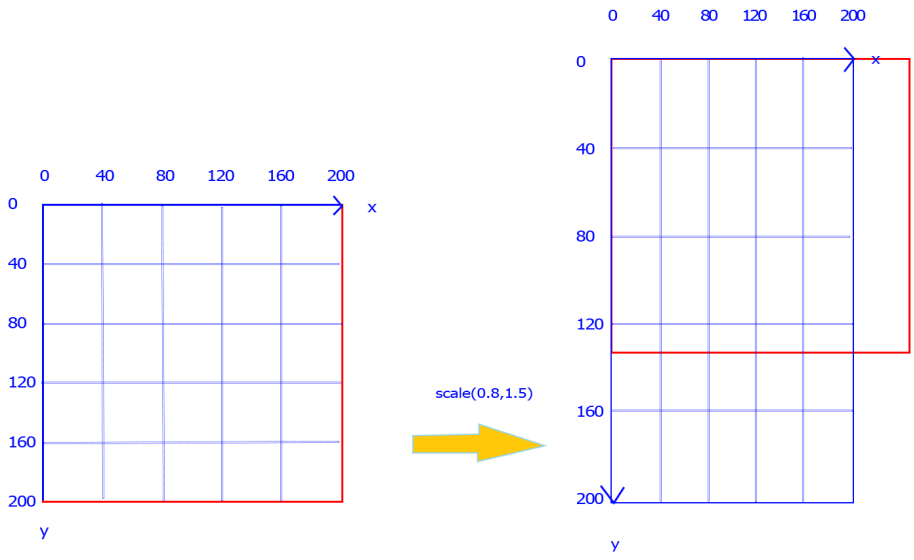
Next to these **three** transformations, there are also the general `transform()` and `setTransform()` methods, which are more powerful, but also more difficult to comprehend and apply. We also show that any combination of transformations (e.g. first scaling, then a rotation, and finally scaling again) is itself a transformation and can be performed with one `transform()` call. The other way round also implies, that we can decompose a complex transformation into simpler steps.

- **`Scale()`**

```
context.fillRect (0, 0, 240, 240);
```

The **`scale()`** method of the *Canvas* adds a scaling transformation to the canvas units horizontally and/or vertically.

By default, one unit on the canvas is exactly one pixel. A scaling transformation modifies this behavior. For instance, a scaling factor of 0.5 results in a unit size of 0.5 pixels; shapes are thus drawn at half the normal size. Similarly, a scaling factor of 2.0 increases the unit size so that one unit becomes two pixels; shapes are thus drawn at twice the normal size.



For example: given a 200x200 pixel canvas shown with a red border and with a blue coordinate system, on top of the red border, in the left image above.

If we now call, say, `scale(0.8,1.5)`, the horizontal x-axis shrinks by the factor of **0.8** and the vertical y-axis extends by the factor of **1.5**. The same blue coordinate system on the red border canvas is shown on the right side. This means that all drawn picture elements after the call of `scale(0.8,1.5)` are now smaller in width and bigger in height, accordingly.

Note : The size of the canvas has not changed. The picture in the browser is still everything that lies within the red border.

Example 1, scaling a shape.

```
context.fillStyle = 'red';
context.strokeStyle = 'black';
context.font = '40px Arial';
// 1. draw a rectangle with text
context.fillRect (50, 50, 150, 40);
context.strokeText ('1. Hello', 55, 85);
// 2. scale and draw the same rectangle with text again
context.scale (0.8, 2);
context.fillRect (50, 50, 150, 40);
context.strokeText ('2. Hello', 55, 85);
// 3. scale once more, and make the same drawings
context.scale (0.8, 2);
context.fillRect (50, 50, 150, 40);
context.strokeText ('3. Hello', 55, 85);
```

Generates this picture:

Example 1, flipping things horizontally or vertically.

You can use `scale(-1, 1)` to flip the context horizontally and `scale(1, -1)` to flip it vertically. In this example, the words "Hello world!" are flipped horizontally.

```
var canvas = Widgets.getCanvas("canvas");
var context=canvas.getContext();
context.scale(-1, 1);
context.font = '48px serif';
context.fillText('Hello world!', -280, 90);
context.setTransform(1, 0, 0, 1, 0, 0);
```

Generates this picture:

Note : The call to `fillText()` specifies a negative x coordinate. This is to adjust for the negative scaling factor, where -280 *-1 becomes 280, and text is drawn leftwards from that point.

- **`Rotate()`**

```
context.rotate(angle)
```

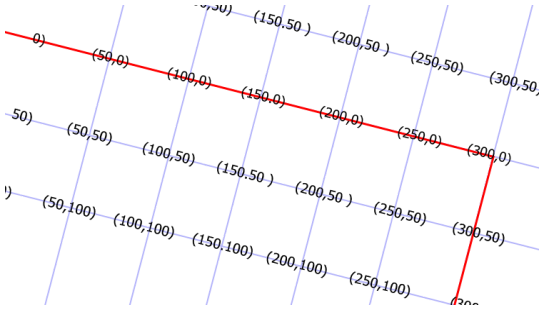
If we call `rotate(angle)`, all subsequently drawn objects are rotated by the given **angle**, where *angle* is a (floating point) number, according to the new blue coordination system.

Note : The rendered canvas in the browser is still the red one, not the rotated blue one. Only the points that lie inside the red canvas are visible.

More precisely, let us consider the default size (300x150 pixel) canvas with a purple grid and the coordinates such as (50,50).

This is a picture of the canvas before the rotation :

After the rotation by **15** degrees by calling `rotate(15*Math.PI/180)`, the same grid looks like this :



Here, the point (250, 100) has disappeared because it now falls outside the canvas, while (250, -50) now does exist.

For example:

```
context.fillStyle = 'red';
context.strokeStyle = 'black';
context.font = '40px Arial';
// 1. draw a rectangle with text
context.fillRect (100, 5, 150, 40);
context.strokeText ('1. Hello', 105, 40);
// 2. declare a rotation and draw the same rectangle with
text again
context.rotate (30 * Math.PI / 180);
// first rotation of 30 degree
context.fillRect (100, 5, 150, 40);
context.strokeText ('2. Hello', 105, 40);
// 3. declare the same rotation, once more, and make the
same drawings
context.rotate (30 * Math.PI / 180);
// second rotation of 30 degree
context.fillRect (100, 5, 150, 40);
context.strokeText ('3. Hello', 105, 40);
```

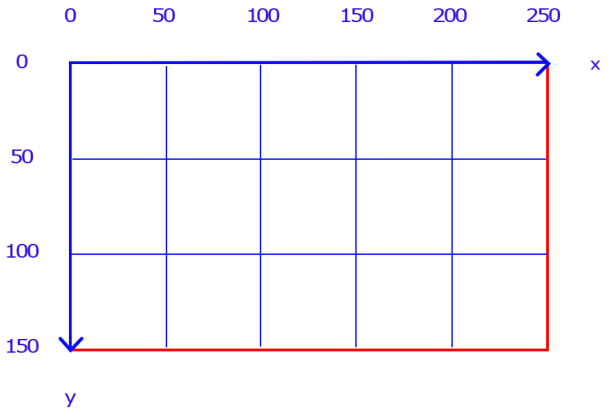
Generates this picture:

- **`Translate()`**

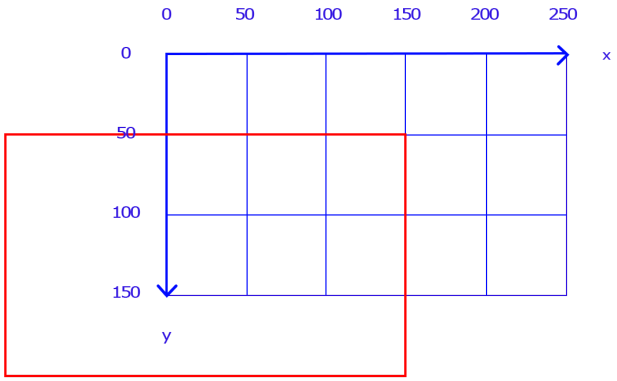
```
context.translate(x, y)
```

moves the origin of the canvas **x** pixels to the right and **y** pixels down. Every new item after the call of `translate(x,y)` is added according to these new coordinates.

Suppose we have a canvas ofsay **250x150** pixel displayed by the red frame and its blue coordinate system on top of it.



After a call of translate (100,-50), this is the new coordinate system



For example:

```
context.fillStyle = 'red';
context.strokeStyle = 'black';
context.font = '40px Arial';
// 1. draw the first red rectangle with text
context.fillRect (0,5,150,40);
context.strokeText ('1. Hello', 5, 40);
// 2. declare a translation and draw the same rectangle,
again
context.translate(125,50);
// first call of translate(125,50)
context.fillRect (0,5,150,40);
context.strokeText ('2. Hello', 5, 40);
// 3. declare the same translation, once more, and make the
same drawings
context.translate(125,50);
// second call of translate(125,50)
context.fillRect (0,5,150,40);
context.strokeText ('3.Hello', 5, 40);
```

Generates this picture:

- **`Transform()`**

```
context.transform(a, b, c, d, e, f)
```

This is the general method to perform an affine transformation. Each point **(x,y)** is transformed into the new point **(x',y')**, where

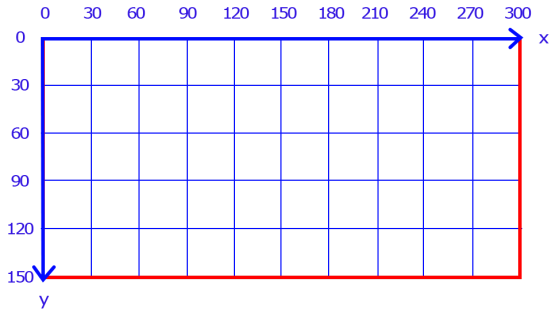
```
x' = a*x + c*y + e
y' = b*x + d*y + f
```

The effect of an each parameter on the outcome is as follows:

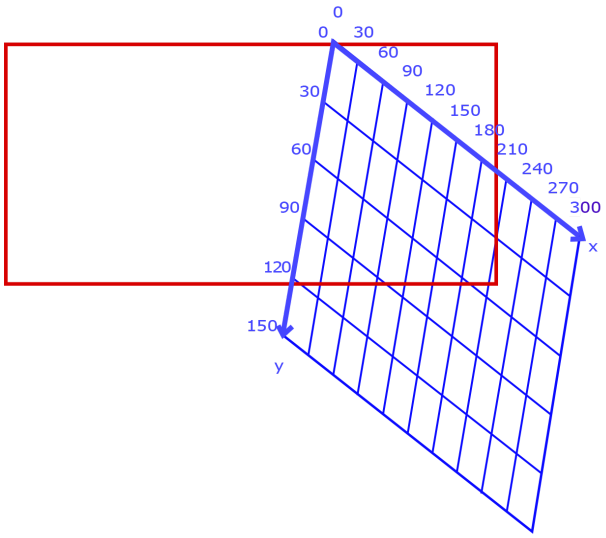
Parameter	Effect
`a`	scales horizontally (and mirrors at the y-axis for negative a)
`b`	skews horizontally
`c`	skews vertically
`d`	scales vertically (and mirrors at the x-axis for negative a)
`e`	moves horizontally
`f`	moves vertically

For example:

Consider a default size (i.e. 300 x 150 pixel) canvas, given by a red frame, with a blue coordinate grid on top:



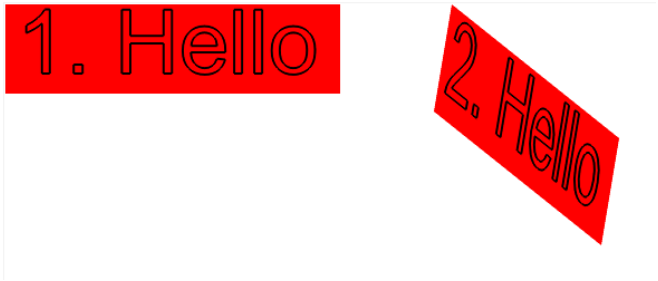
After a transformation of say, `transform(0.5, 0.4, -0.2, 1.2, 200, 0),` the new coordinate system is this :



And if we populate this canvas **context** before and after the same transformation by running this code :

```
// prepare the settings for color and font styles
context.fillStyle = 'red';
context.strokeStyle = 'black';
context.font = '40px Arial';
// 1. draw the first red rectangle with text
context.fillRect (0, 5, 150, 40);
context.strokeText ('1. Hello', 5, 40);
// 2. declare the translation and draw the same rectangle,
again
context.transform (0.5, 0.4, -0.2, 1.2, 200, 0);
context.fillRect (0, 5, 150, 40);
context.strokeText ('2. Hello', 5, 40);
```

Then this is the resulting picture:



- ## `Scale(), rotate() & translate in terms of transform()`

`Scale()`, `rotate()` and `translate()` are just special versions of `transform()` with less parameters. The following table shows the exact implementations:

`scale(x,y)`	**`transform`** `(x, 0, 0, y, 0, 0)`
`rotate(angle)`	**`transform`** `(c, s, -s, c, 0, 0)`, where c = `Math.cos(angle)` and s = `Math.sin(angle)`
`translate(x,y)`	**`transform`** `(1, 0, 0, 1, x, y)`

- **`SetTransform()`**

```
context.setTransform(a, b, c, d, e, f)
```

Initially, every canvas context has the initial coordinate system, as described in the introduction. After every transformation, this coordinate system is changed to a new current system. With each `tranform()` call, this current system is transformed, again. A call of `setTransform()` however does not transform the current system, but starts from the initial one.

Note: In standard jargon, we would rather talk about "transformation" instead of "coordinate system".

- **`ResetTransform()`**

```
context.resetTransform(a, b, c, d, e, f)
```

This method resets the current transformation to the identity matrix. Therefore, whenever you're done drawing transformed shapes, you should call `resetTransform()` before rendering anything else.

`For example:` the first two shapes are drawn with a skew transformation, and the last two are drawn with the identity (regular) transformation.

```
var canvas = Widgets.getCanvas("canvas");
var context=canvas.getContext();
// Skewed rectangles
context.transform(1, 0, 1.7, 1, 0, 0);
context.fillStyle = 'gray';
context.fillRect(40, 40, 50, 20);
context.fillRect(40, 90, 50, 20);
// Non-skewed rectangles
context.resetTransform();
context.fillStyle = 'red';
context.fillRect(40, 40, 50, 20);
context.fillRect(40, 90, 50, 20);
```

The skewed rectangles are gray, and the non-skewed rectangles are red :

Note : You can also reset transformations using

```
context.setTransform (1,0,0,1,0,0);
```

CHAPTER 6 :

DRAWING STATE

A drawing state is a set of the current settings of a `CanvasRenderingContext2D` object. It includes the current style values (`strokeStyle` and `fillStyle`), `globalAlpha` and `globalCompositeOperation`, the line (`lineWidth`, `lineCap`, `lineJoin`, `miterLimit`), the text styles (`font`, `textAlign`, `textBaseline`), the current clipping region and the current transformation matrix.

Each **`CanvasRenderingContext2D` object maintains a stack of drawing states**. And with `save()` and `restore()`, drawing states can be pushed onto this stack and recovered at a later point.

- ## Global Alpha

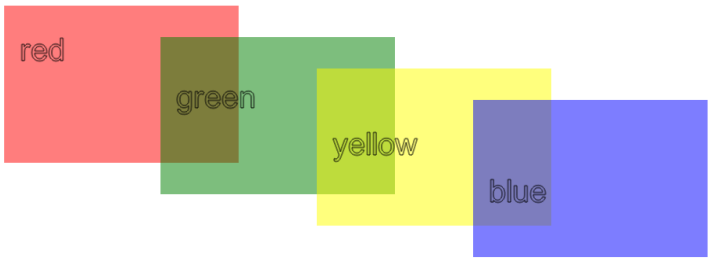
```
context.globalAlpha =value;
```

This method determines the current α (alpha or transparency) value for the drawings, where α value is a float number between **0.0** (for fully transparent) and **1.1** (for fully opaque). The default value is **1.0,** which means that all drawn items are not transparent at all.

For example:

```
// 1. setting alpha
context.globalAlpha = "0.5";  // THIS IS AN IMPORTANT LINE!
// 2. set the text style
context.font = '20px Arial';
context.strokeStyle = 'black';
// 3. draw the four rectangles
context.fillStyle = 'red';
context.fillRect( 10,10,150,100);
context.strokeText('red', 20, 50);
context.fillStyle = 'green';
context.fillRect(110,30,150,100);
context.strokeText('green', 120, 80);
context.fillStyle = 'yellow';
context.fillRect(210,50,150,100);
context.strokeText('yellow',220,110);
context.fillStyle = 'blue';
context.fillRect(310,70,150,100);
context.strokeText('blue', 320,140)
```

Generates this product:



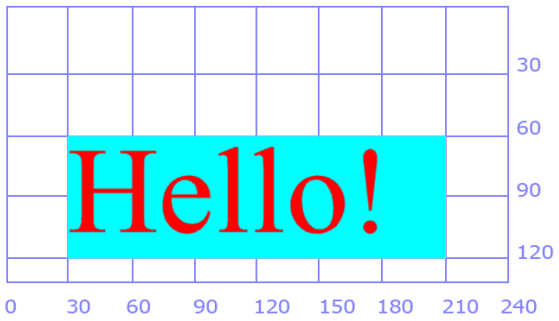
- **Clipping Graphics**

```
context.clip();
```

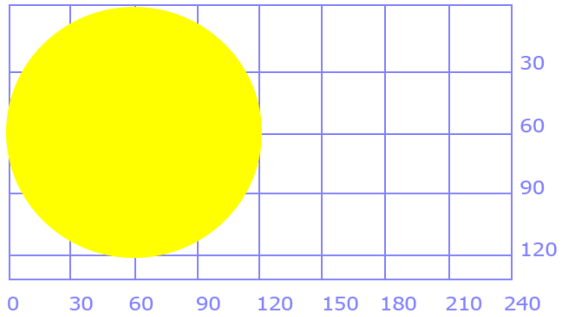
This method clips a region of any shape and size from the canvas.

For example:

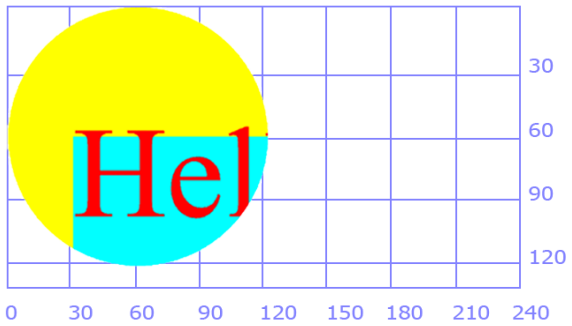
Suppose we have any drawing on a canvas, say a rectangle with text :



And suppose, we want to clip from this canvas, the part that is defined by this circle :



So that the overall result looks like this :



The standard method to achieve this is in three steps is:

- draw the shape of the area to be clipped,
- call the `clip()` method and
- draw the canvas content

Note: *The clip technique works for all canvas drawings, including images.*

The source code of the previous example is :

```
// 1. clipped circle area
context.fillStyle = 'yellow';
context.beginPath();
context.arc(60, 60, 60, 0, 2 * Math.PI);
context.fill();
// 2. clip
context.clip();
// 3. draw a rectangle with text
context.fillStyle = 'cyan';
context.fillRect(30, 60, 180, 60);
context.fillStyle = 'red';
context.font = '60px sans-serif';
context.fillText('Hello!', 30, 110);
```

- ## Creating Dashed Lines

Dashed Lines are created using `setLineDash()` method and `lineDashOffset` property of context.

1. `setLineDash()` method

```
context.setLineDash(array);
```

This function is how you specify the **linedash** setting. It takes an array of numbers, which are specifying the sequence of dash / space / dash / space, etc.. When finished, the sequence repeats itself to the end of the line.

Some examples:

```
context.setLineDash([5])
```

A basic setting that will result in a dashed line where both the dashes and spaces are 5 pixels in size.

```
context.setLineDash([2,3])
```

A setting that will result in a dotted line where the dashes are 2 pixels in size and the spaces are 3.

```
context.setLineDash([2,3])
```

Another dashed line but with small dashes.

```
context.setLineDash([5,5,2,2])
```

A setting where the first dash is five pixels, then a space of five pixels, then a dash of two pixels, then a space of two pixels, then the sequence starts again.

2. getLineDash() method

```
context.getLineDash();
```

This method returns the current linedash setting.

3. lineDashOffset Property

```
context.lineDashOffset = 2;
```

This setting can be used to stipulate how far into the line dash sequence drawing commences. So using the **[5,5,2,2]** setting from above, if you set the `lineDashOffset` to **2,** the whole sequence of dashes and spaces will shift 2 pixels towards the left from the original position.

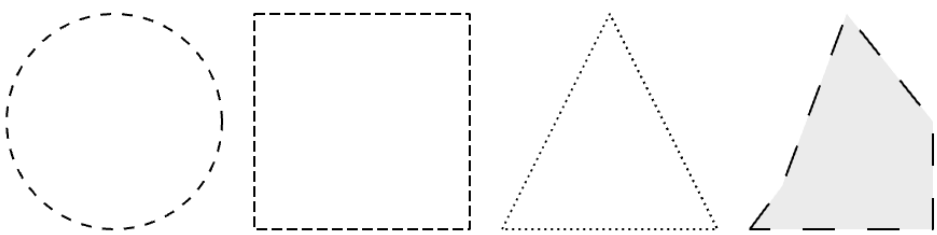
For example:

```
// Draw the circle
context.beginPath();
context.setLineDash([ 5 ]);
context.arc(65, 65, 50, 0, 2 * Math.PI, false);
context.stroke();

// Draw the square
context.beginPath();
context.setLineDash([ 5, 2 ]);
context.rect(130, 15, 100, 100);
context.stroke();

// Draw the triangle
context.beginPath();
context.setLineDash([ 1, 2 ]);
context.moveTo(245, 115);
context.lineTo(295, 15);
context.lineTo(345, 115);
context.closePath();
context.stroke();

// Draw the irregular shape
context.beginPath();
context.fillStyle = '#eee';
context.setLineDash([ 15 ]);
context.moveTo(360, 115);
context.lineTo(375, 95);
context.lineTo(405, 15);
context.lineTo(445, 65);
context.lineTo(445, 115);
context.closePath();
context.fill();
context.stroke();
```



Saving and Restoring State

The drawing state that gets saved onto a stack consists of:

A. The current transformation matrix.
B. The current clipping region.
C. The current dash list
D. The current values of the following attributes: `strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, lineDashOffset, globalCompositeOperation, font, textAlign, textBaseline.`

The following functions are used to save and restore the state of context :

1. **`context.save()` method**

   pushes the current state on the state stack.

2. **`context.restore()` method**

   removes the top state from the state stack and thus restore the previous state that was pushed on the state stack.

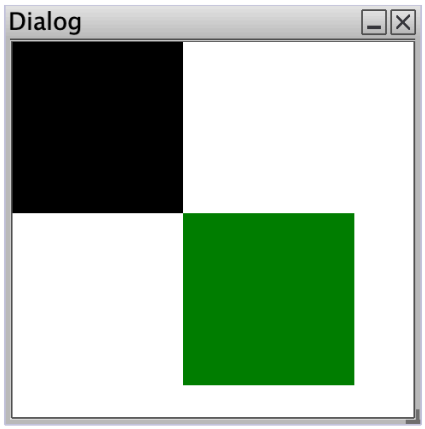
   For example:

   This example uses the `save()` method to save the default state and `restore()` to restore it later, so that you are able to draw a rect with the default state later.

```
// Save the default state
context.save();
context.translate(100, 100)
context.fillStyle = 'green';
context.fillRect(0, 0, 100, 100);
```

```
// Restore the default state
context.restore();
context.fillRect(0, 0, 100, 100);
```

The output is shown as below :



Note : After calling `restore()`, state is changed back to default state. That's why rectangle is black and is untranslated.

- ## Offscreen Canvas

Offscreen canvas is used to store the complete drawing and data of a canvas. You can make as many offscreen canvas as you want and make different drawings in them as per your choice. And make the drawing on onscreen canvas by using `context.drawImage()` method of the onscreen canvas' context.

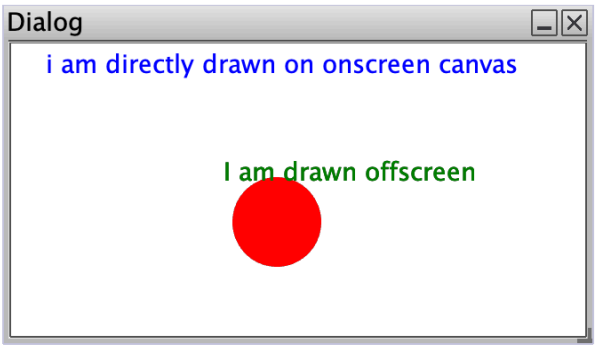
For example:

```
var canvas=Widgets.getCanvas("canvas");
var ctx=canvas.getContext();
//create canvas named "mycanvas" of size 250x250 for
offscreen rendering
//but first check if it is already created (needed if
script runs multiple times, will create many canvases)
```

Continued on the next page…

```
var offCanvas=canvas.getOffscreenCanvas("mycanvas");
if(offCanvas==null)canvas.createOffscreenCanvas("mycanvas",
250, 250);
//Let's draw a circle and some text on this canvas
var octx=offCanvas.getContext();
octx.fillStyle="red";
octx.fillOval(50,50,50,50);
octx.strokeStyle="green";
octx.strokeText("I am drawn offscreen", 20,30);
//Lets copy back all content on our real canvas
//for this simply treat offscreencanvas as simple image
ctx.drawImage(offCanvas, 100,50)
ctx.strokeStyle="blue";
ctx.strokeText("i am directly drawn on onscreen canvas",
20,20);
```

This will give the following result :



CHAPTER 7 :

EVENT HANDLING

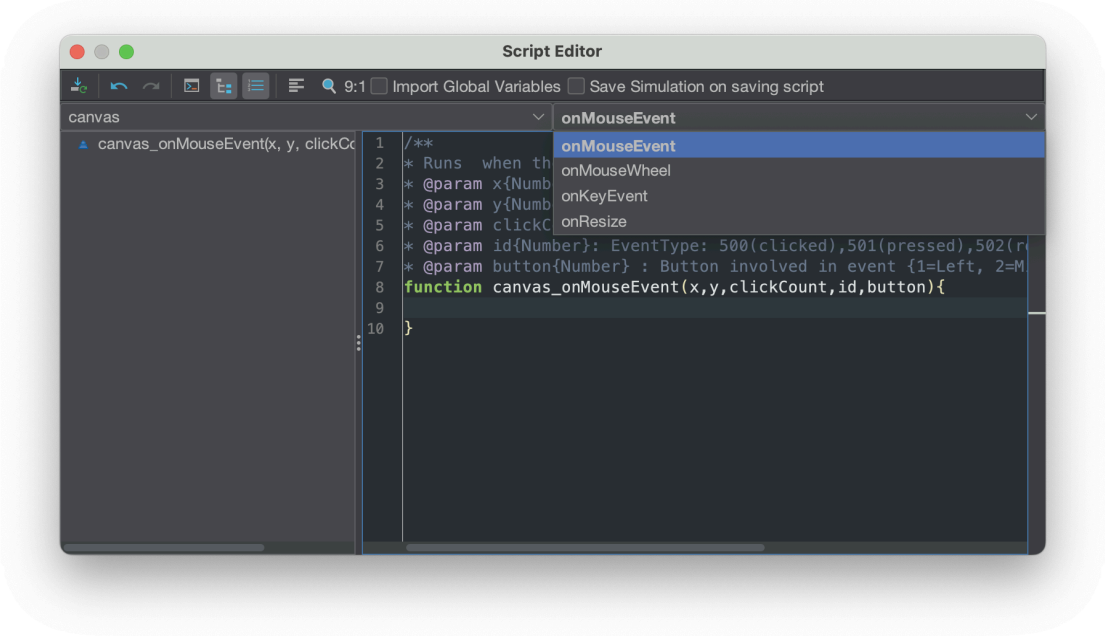
To make simulations interactive you must respond to events generated by input devices like mouse and keyboards. In canvas, you will encounter 3 of the following type of events :

- **Resize Event:** occurs whenever canvas is resized (manually setting its size or because of layout change in parent of canvas).
- **Keyboard Events:** occurs whenever canvas has focus and some key is pressed or released.
- **Mouse or Touch Events:** occurs whenever mouse/finger is moved, dragged or clicked on canvas.

Creating an Event Handler in Editor (Preferred Method)

To create an event handler:

- Right click on canvas to open popup menu.
- Click on 'Script Editor', the script editor window will pop up.
- Select the Canvas from the left drop-down list (should be auto selected) and the desired event from the right drop-down list as shown below.

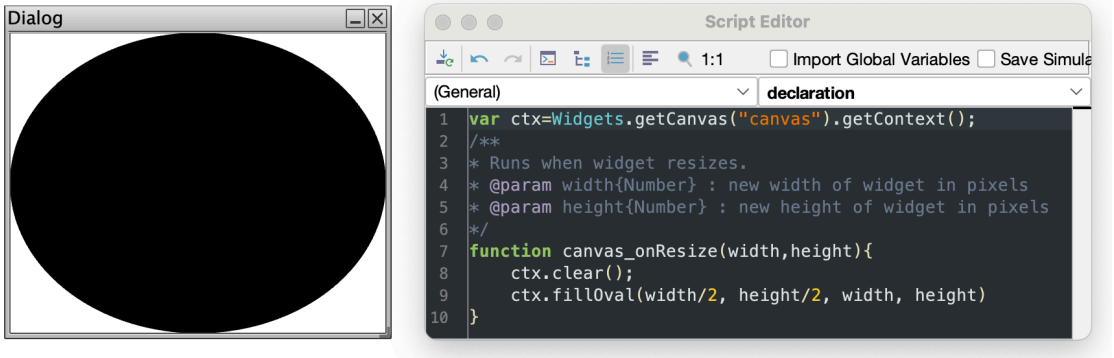


Editor will create the event handler method, with documentation. Now you can write code inside this method which will automatically be invoked (once script is saved) with event parameters as mentioned in documentation whenever corresponding event occurs.

Note : There is a single method for all Mouse Events, individual events (like click, move, drag, etc.) can be filtered with the help of parameter **id** which describes the actual event that occurred. Same holds true for all Key Events.

An Example of Resize Event of Canvas

The screen shot shows an oval which automatically fits in canvas even when canvas is resized.



An Example of Mouse Event of Canvas

You must be familiar with our vintage drawing app 'M.S. Paint'. Lets create a little paint demo on canvas, which draws lines on canvas on dragging mouse.
Just create a canvas with the name 'canvas' and set its background color as green through the property panel (to give it a cool green board effect!).

Now add following snippet to the script editor and save it :

```
var ctx=Widgets.getCanvas("canvas").getContext();
var pt=new Vector2(0,0);
ctx.strokeStyle="white";
ctx.lineWidth=2;
/**
* Runs when the widget receives mouse event (events are
similar to java.awt events).
* @param x{Number} : x coordinate of mouse in widget space
(top left as origin)
* @param y{Number} : y coordinate of mouse in widget space
(top left as origin)
* @param clickCount{Number} : Number of clicks
* @param id{Number}: Event Type:
500(clicked),501(pressed),502(released),503(moved),504(entere
d),505(exited),506(dragged)
* @param button{Number} : Button involved in event {1=Left,
2=Middle and 3=Right mouse
button*/

function canvas_onMouseEvent(x,y,clickCount,id,button){
   //clear canvas on double clicking
   if(clickCount>1) ctx.clear();
   //draw lines on dragging like a pen
   if(id==501){
       pt.set(x,y);
   }
   else if(id==506){
       ctx.beginPath();
       ctx.moveTo(pt.x,pt.y);
       ctx.lineTo(x,y);
        pt.set(x,y);
       ctx.stroke();
   }
}
```

This will give the following result :

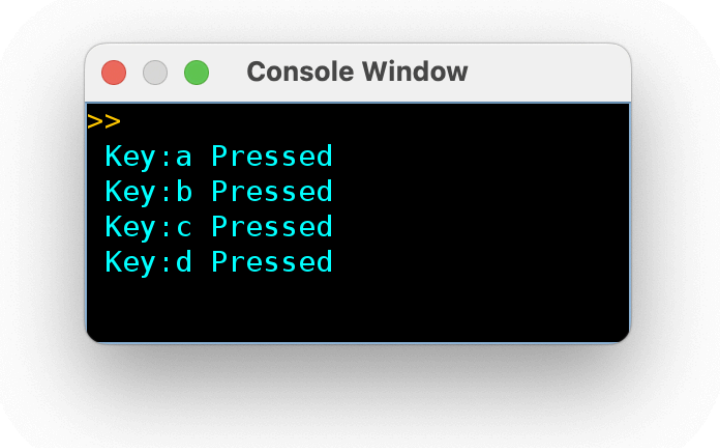
Now try writing something on canvas with mouse drag, double click to erase all and relive your childhood again!

An Example of Key Event of Canvas

Lets see how to respond to Key Events in canvas. Create a canvas named "canvas" and add following code to script.

```
/**
* Runs when widget receives key event (events are similar to
java.awt events).
* @param keyChar{char} : the character(if any) associated
with keyevent
* @param keyCode{Number} : key code {@see
com.jogamp.newt.event.KeyEvent} for keycodes
* @param key{String} : the identifier of the key (like
'a','Enter','F11) that was pressed when a
* key event occured
* @param id{Number} : EventType 401(keyPressed),
402(KeyReleased)
* Note that key events are received only when widget is
active (click on widget to activate)*/
function canvas_onKeyEvent(keyChar,keyCode,key,id){
   Console.show();
   if(id==401){
       Console.info(" Key:"+key+" Pressed");
   }
   else{
        Console.log(" Key:"+key+" Released");
   }
}
```

Now save script and then bring focus to canvas(by clicking over it). Now press some keys and you will observe console logging key info as below.



Creating an Event Handler in Editor (Alternate Method)

- **Mouse or Touch Events:**

Detecting mouse events in a canvas is simple enough, you add an event listener to the canvas and the browser invokes that listener when the event occurs.

For example, you can listen to mouse down events like this:

```
canvas.onmousedown = function (e) {
    // React to the mouse down event
};
```

In addition to `onmousedown`, you can also assign functions to `onmousemove`, `onmouseup`, `onmouseover`, and `onmouseout`.

Alternatively, you can use the more generic `addEventListener()` method:

```
canvas.addEventListener('mousedown', function (e) {
    // React to the mouse down event
});
```

Assigning a function to `onmousedown`, `onmousemove`, etc., is a little simpler than using `addEventListener()`. However, `addEventListener()` is necessary when you need to attach multiple listeners to a single mouse event.

- **Key Events:**

When you press a key in a browser window, the browser generates key events. Those events are targeted at the HTML element that currently has focus. If no element has focus, key events bubble up to the window and document objects. The canvas element is not a focusable element, and therefore in light of the preceding paragraph, adding key listeners to a canvas is an exercise in futility. Instead, you will add key listeners to either the document or window objects to detect key events.

```
document.addEventListener(event, function, useCapture)
```

Here,

1. **`event`** requires a string that specifies the name of the event.

   Note: Do not use the "on" prefix. For example, use "click" instead of "onclick". Here's a list of all HTML DOM events for reference.

2. **`function`** specifies the function to run when the event occurs and an event object is passed to the function as the first parameter.

   The type of the event object passed depends on the specified event. For example, the "click" event belongs to the MouseEvent object.

3. **`useCapture`** (optional) is a boolean value that specifies whether the event should be executed in the capturing (if the value is true) or in the bubbling phase (if false).

There are three types of key events:

- `Keydown`
- `Keypress`
- `Keyup`

The `keydown` and `keyup` events are low-level events that the browser fires for nearly every keystroke.

Note : Some keystrokes, such as command sequences, may be swallowed by the browser or the operating system. However, most keystrokes make it through to your `keydown` and `keyup` event handlers, including keys such as Alt, Esc, and so on.

When a `keydown` event generates a printable character, the browser fires a `keypress` event before the inevitable `keyup` event. If you hold a key that generates a printable character down for an extended period of time, the browser will fire a sequence of `keypress` events between the `keydown` and `keyup` events. Implementing key listeners is similar to implementing mouse listeners.

You can assign a function to the document or window object's `onkeydown`, `onkeyup`, or `onkeypress` variables, or you can call `addEventListener`(), with `keydown`, `keyup`, or `keypress` for the first argument, and a reference to a function for the second argument.

Determining which key was pressed can be complicated, for two reasons. First, there is a huge variety of characters among all the languages of the world. When you take into consideration the Latin alphabet, Asian ideographic characters, and the many languages of India, just to mention a few, supporting them all is mind boggling.

Second, although browsers and keyboards have been around for a long time, key codes have never been standardized until DOM Level 3, which few browsers currently support.

In a word, detecting exactly what key or combination of keys has been pressed is a mess. However, under most circumstances you can get by with the following two simple strategies :

1. For `keydown` and `keyup` events, look at the **`keyCode`** property of the event object that the browser passes to your event listener. In general, for printable characters, those values will be ASCII codes. Notice the general caveat, however. Here is a good website that you can consult for interpreting key codes among different browsers.

   Event objects for key events also contain the following boolean properties:

   - altKey
   - ctrlKey
   - metaKey
   - shiftKey

2. For `keypress` events, which browsers fire only for printable characters, you can reliably get that character like this:

```
var key = String.fromCharCode(event.which);
```

In general, unless you are implementing a text control in a canvas, you will handle mouse events much more often than you handle key events.

Tip: Use the **`document.removeEventListener()`** method to remove an event handler that has been attached with the `addEventListener()` method.