Programming paradigm: A paradigm refers to the set of practices that are commonly used.

Imperative language: Imperative languages are based on instructions to change the computer memory.

JS is an example.

Assembly language were the first imperative programming language. Each type of processor had its own set of instructions, so these languages were different for each processor.

Procedural languages: Imperative + Procedures

Related instructions are grouped together in procedures (or functions, or methods).

All procedural languages are also imperative. Example is C.

Object oriented programming: Procedural + objects

Object-oriented paradigm lets programmers group methods into something called a class. The memory that these groups of methods can change are called objects,

A class can describe something like the ghosts in a game of Pac-man. Then each individual ghost corresponds to the object of the class.

C++ is an example.

Then later Java was developed, eliminating some stuff from C and solving some issues with C++.

Web Applications:

Java could run in a browser as so-called applet.

But this required plug-ins and there was no straightforward way for applets to interact with the elements of a browser.

HTML is a document formatting language and it’s an abbreviation of Hyper Text Markup Language.

Netscape invented LiveScript which was later named JavaScript.

It was submitted to ECMA standardization organization, which changed its name to ECMAScript.

ECMAScript 5.1 was released in 2011. ECMAScript 6 (ES6) in 2015.

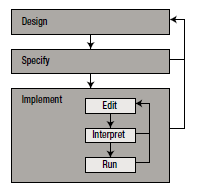
Although it doesn’t have all the features of modern programming languages, it’s still very capable language. Currently, JavaScript is the only language integrated with HTML that works across different platforms. Together with HTML5, it has become a powerful framework for web development.

Game development:

Engines provide widely used features to help code games faster.

In JS, instead of using an engine, programmers write the game directly using HTML5 elements such as the canvas.

Game development Approaches:



On large Scale: Design-Specify-Implement

On small Scale: Edit-Interpret-Run

HTML, CSS and JS:

HTML for defining structure of page.

CSS for making it look good.

JS for making it dynamic.

HTML5 Canvas:

Canvas element allows you to draw graphics in a HTML document.

<div id=”gameArea”>

<canvas id=”mycanvas” width=”800” height=”480”></canvas>

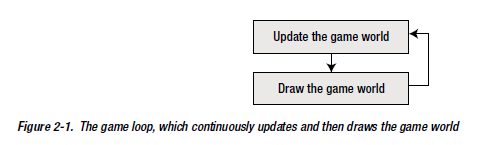
</div>

Game Programming Basics:

Game world: this

Game loop:

* Tasks related to updating and maintaining the game world
* Tasks related to displaying the game world to the player



Building game using JS:

Instructions are grouped into a function.

Setting up a simple game loop in JS.

var canvas = undefined;

var canvasContext = undefined;

function start () {

canvas = document.getElementById("myCanvas");

canvasContext = canvas.getContext("2d");

mainLoop();

}

document.addEventListener('DOMContentLoaded', start);

function update () {

}

function draw () {

}

function mainLoop () {

canvasContext.fillStyle = "blue";

canvasContext.fillRect(0, 0, canvas.width, canvas.height);

update();

draw();

window.setTimeout(mainLoop, 1000 / 60);

}

Structure of a Program:

Console applications: text-based applications (no graphics)

Types of Applications:

Windows application: screen containing windows, buttons and other parts of GUI. Often event-driven.

App: run on a mobile phone or a tablet PC. Limited screen space, new interaction possibilities (touch, GPS, sensors).

Web based applications can run on different types of devices.

Functions:

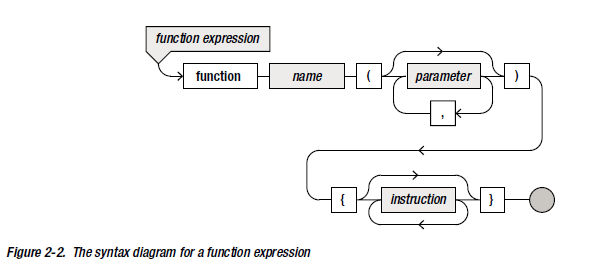
function function\_name (parameters) {

//function body

}

Syntax diagram.:

Syntax diagrams explain how language is structured. Syntax=formal rules. Semantics=actual meaning. Interpreter checks for syntax errors but not semantic.



Calling a function:

function\_name(necessary\_arguments);

OR object. function\_name(necessary\_arguments);

Program layout:

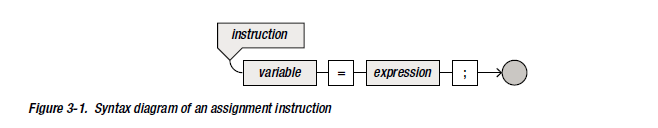
* Comments /\*Multi line comment\*/ and //single line comment.
* Multiple instructions can be written in a single line. But it is better to distribute to improve readability of code.
* Whitespace and Indentation can be used as required. Except, you can’t add spaces between a word.
  + It is good to keep spaces behind every comma and semicolon, before and after operator (except unary), before lines for indentation.

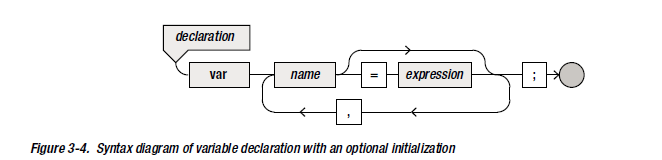
Creating a Game World.

Basic Types and Variables.

Variables are names given to memory location.

Type declaration is not required in JS and JS converts where required if possible.





Strict mode:

“use strict”; //You can add this instruction to tell interpreter to use strict mode.

In strict mode, variable declaration is compulsory.

Instructions and Expressions:

Instructions changes the memory in some way, whereas an expression has a value.

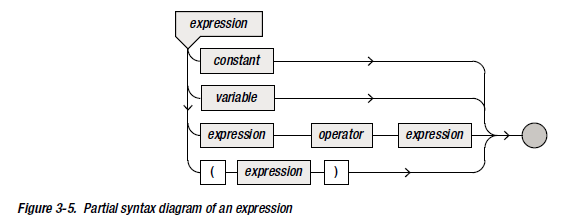
a=5 is an instruction.

a+5 is an expression.

Operators:

+,-,\*,/,%

Priority of Operators: Regular arithmetic rules apply. But it is better to use parenthesis if you aren’t sure.



Assigning function to a variable:

In JS, it’s possible to assign a function to a variable.

let someFunction = function () {

//do something

}

You can call the function using someFunction();

Difference between normal function and function like above:

someFuntion();

function someFunction() {

//do something

}

//That doesn’t generate an error but the following does:

someFunction();

var someFunction() = function () {  
 //do something

}

someFunction();

Composite variables:

var gameCharacter = {

name : “Merlin”,

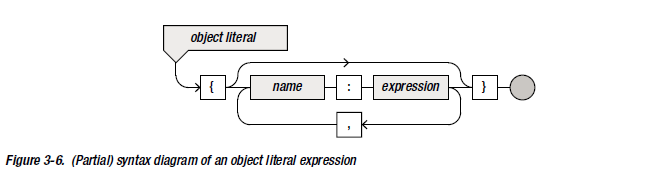
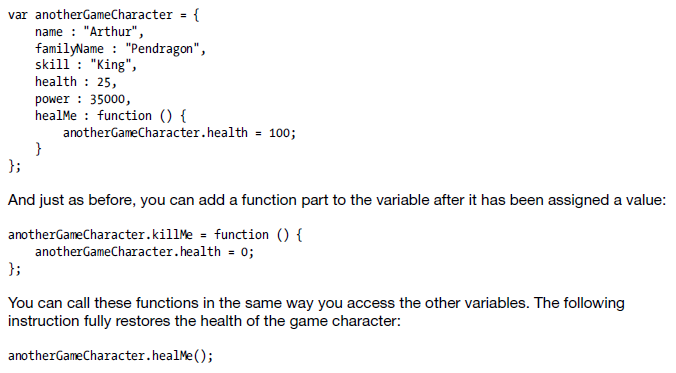
skill : “Magician”,

health : 100,

power : 230

};

The expression consisting of the names and values enclosed by the braces is called an object literal.

The MovingSquare Game: In folder Fourth Application – The MovingSquareGame

Game Assets:

Game.balloonSprite = {

src: “spr\_balloon.jpg”

width : 35,

height: 63,

…

}

This becomes problematic when you want to load hundreds of sprites for your game. So, use types.

A type is basically a definition of what an object of that type should look like; i.e. a blueprint for an object.

To create object of Image type:

Game.balloonSprite = new Image();

Then Game.balloonSprite.src = “spr\_balloon.png”;

Height and width is automatically filled because of type.

Window.setTimeout(Game.mainLoop, 500);

Drawing Sprites:

Game.canvasContext.drawImage(sprite, 0,0, sprite.width, sprite.height, 0,0,sprite.width, sprite.height);

Drawing state is a set of parameters and transformations that will be applied to all the things drawn within that state.

Creating a new drawing state:

Game.canvasContext.save();

Translation:

Game.canvasContext.translate(100,100);

Removing drawing state:

Game.canvasContext.restore();

You could create a function like this:

Game.drawImage = function (sprite, position) {

Game.canvasContext.save();

Game.canvasContext.translate(position.x,position.y);

Game.canvasContext.drawImage(sprite,0,0,sprite.width, sprite.height, 0,0, sprite.width, sprite.height);

Game.canvasContext.restore();

};

Now, you can call the function to draw a image at 100,100 as follows:

Game.drawImage(Game.balloonSprite, {x:100, y: 100});

Music:

Game.backgroundMusic = new Audio();

Game.backgroundMusic.src = “snd\_music.mp3”;

Game.backgroundMusic.play();

Game.backgroundMusic.volume = 0.4; //0 = no sound, 1 = max volume

Knowing what the player is Doing:

A Sprite following the mouse pointer

Retrieving the Mouse position

Event handlers in JS allow you to execute instructions when specified event occurs.

An event-handler function has a specific header. It contains a single parameter that, when the function is called, contains an object providing information about the event.

Example:

Document.onmousemove = handleMouseMove;

Function handleMouseMove(evt) {

Game.balloonPosition = {x: evt.pageX, y: evt.pageY};

}

//You can use clientX and clientY instead but this doesn’t take scrolling into account. It might be useful if you are trying to create annoying ads that keeps appearing in the middle of the browser view even if the user tries to scroll past it.

Changing origin of sprite:

Var origin = {x:someSprite.width/2, y:someSprite.height/2};

Var pos = { x: somePosition.x -origin.x, y: somePosition.y -origin.y};

Game.drawImage(someSprite, pos);

// drawImage method from the canvas context also has a way to specify the origin offset.

Game.canvasContext.drawImage(sprite, 0,0, sprite.width, sprite.height, -origin.x, -origin.y, sprite.width, sprite.height);

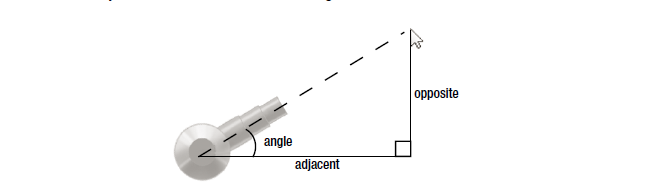
Using mouse position to rotate the cannon barrel:

Add a rotation variable in game object.

Now before using drawImage(), rotate the canvasContext

i.e. Game.canvasContext.rotate(rotation);

To calculate rotation to point at a stuff:

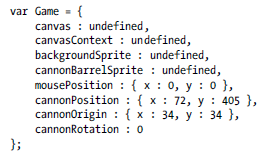


Var opposite = Game.mousePosition.y – Game.cannonPosition.y;

Var adjacent = Game.mousePosition.x – Game.cannonPosition.x;

Rotation = Math.atan2(opposite, adjacent);

Reacting to Player Input:



Now, that’s lot of varibles just for a balloon, background and cannon.

Problem: Everthing in single, big object called Game.

Solution: Separate stuff.

Var Canvas2D = {

Canvas: undefined,

canvasContent: undefined

};

Var Game = {

backgroundSprite: undefined };

var cannon = {

cannonBarrelSprite: undefined,

position: {x: 72, y: 405 };

origin: {x: 34, y: 34},

rotation: 0

};

Var Mouse = { position: { x:0, y: 0} };

Now you can add methods specific to those objects.

// use small letters for objects that can be used in the particular game only and capital letters for objects useful in other games.

Loading sprites:

Load all sprites at once OR add methods to each object to load sprite.

Second option makes code more readable but if you need to use same image for different game

Objects, you have to load sprite multiple times. (not good)

Better option. Separate sprites from the rest of the program.

Var sprites = {};

For each sprite you want to load, create an Image object and set its source to the sprite location.

Var spriiteFolder = “../sprites”

Sprites.background = new Image();

Sprites.background.src = spriteFolder + “spr\_background.jpg”; //+ operator to concatenate

//similarly for others

Handling a key down event:

i.e. Reacting to the event that the player holds down a key on the keyboard.

Store the key, so that you can access it later and do something with that information. Easiest way to store which key way pressed is to use key codes.(standarized).

Var keyboard = { keyDown: -1}; //-1 when no key is pressed.

function handleKeyDown(evt) {

Keyboard.keyDown = evt.keyCode; }

Function gets an event as a parameter. That event object has a variable called keyCode.

Document.onkeydown = handleKeyDown;

For when the key is released:

Function handleKeyUp(evt) {

Keyboard.keyDown = -1;

}

Document.onkeyup = handleKeyUp;

Later you learn Keyboard object to take simultaneous keypresses into account.

Lets add three color balloons : red, green and blue (yellow one exists)

We want to load different colored balloon when key is pressed.

Add a currentColor variable to balloon element.

sprites.balloon\_red = Game.loadSprite(spriteFolder + "spr\_balloon\_red.png");

sprites.balloon\_green = Game.loadSprite(spriteFolder + "spr\_balloon\_green.png");

sprites.balloon\_blue = Game.loadSprite(spriteFolder + "spr\_balloon\_blue.png");

Create a variable keys to store key codes of 4 colors:

Var Keys = {

R: 82, G:71, B:66, Y:89

};

If (Keyboard.keyDown === Keys.R)

balloon.currentColor = sprites.cannon\_red;

Game.draw = function () {

Canvas2D.clear();

Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,

{ x : 0, y : 0 });

cannon.draw();

};

cannon.draw = function () {

Canvas2D.drawImage(sprites.cannon\_barrel, cannon.position, cannon.rotation,

cannon.origin);

Canvas2D.drawImage(cannon.currentColor, cannon.colorPosition, 0,

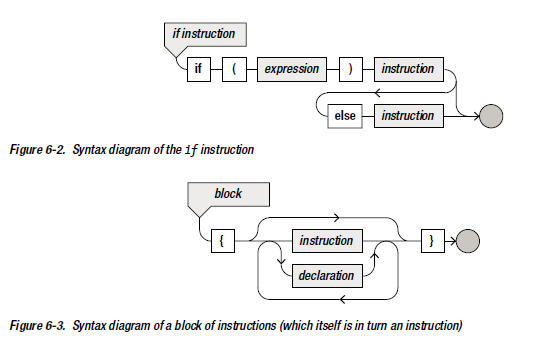
{ x : 0, y : 0 });

};

Comparision Operators:

Others are normal.

Except: === is strictly equals to, !=== is strictly not equals to.



The Boolean Type:

You can do this:

Var test = x>3; // test will either be true or false depending on value of x

And you can use it for conditional statement.

if(test)

console.log(“X is greater than 3”);

Aiming the Barrel at the Mouse Pointer:

When a mouse button is pressed or released, the which variable of the event object tells you which button it was ( 1 is the left button, 2 is the middle button, and 3 is the right button).

Var Mouse = {

position: {x: 0, y: 0},

leftDown: false

}

You can now add handler functions:

Function handleMouseDown(evt) {

If(evt.which === 1)

Mouse.leftDown = true;

}

function handleMouseUp(evt) {

if(evt.which === 1)

Mouse.leftDown = false;

}

Now you assign these handlers to appropriate variables:

Document.onmousedown = handleMouseDown;

Document.onmouseup = handleMouseUp;

So that you can use Mouse.leftDown as condition to do certain thing.

For example: make balloon follow the mouse only if you click left button.

Basic Game Objects:

Using separate JS file makes code manageable. Good way is to split the different JS objects over separate files. You can even keep them in separate folders to indicate that they belong together. For example keyboard and mouse cann be kept in input folder.

Loading these files is an issue. You can use different script tags to load them but they might load at different time and dependencies can cause issue. You need to make sure they are loaded in order that respects the existing dependencies.

Dynamic script-loading tool : LABjs (<https://labjs.com/>)

This lets you load other JS files dynamically in a predefined order.

<script src=”../LAB.min.js”></script>

<script>

$LAB.script(‘input/Keyboard.js’).wait()

.script(‘input/Mouse.js’).wait()

.script(‘Canvas2D.js’).wait()

.script(‘Cannon.js’).wait(function() {

Game.start(‘myCanvas’);

});

</script>

But using such method for final release might not be good. It is preferable to use another program to combine all JS files into a single big file. It’s common to perform some optimization on the structure of the code such that the script file size is as small as possible. (minification)

Loading Game Assets the wrong way:

var sprite = new Image();

sprite.src = "someImageFile.png";

var anotherSprite = new Image();

anotherSprite.src = "anotherImageFile.png";

// and so on

And at last:

window.setTimeout(Game.mainLoop, 500);

This might not be good enough when internet connection is really bad or really good.

Solution: Execute main loop when all the images have been loaded before executing the main loop.

Methods and Functions:

Some have parameters, some don’t.

Some have a result value and some don’t.

Loading Game Assets, the Right Way:

In order to make sprite loading a little bit easier, let’s add a method loadScript to the Game object:

Game.loadSprite = function(imageName) {

Var image = new Image();

Image.src = imageName;

Return image;

}

Code for loading the different sprites now becomes much shorter:

var sprFolder = "../../assets/Painter/sprites/";

sprites.background = Game.loadSprite(sprFolder + "spr\_background.jpg");

sprites.cannon\_barrel = Game.loadSprite(sprFolder + "spr\_cannon\_barrel.png");

sprites.cannon\_red = Game.loadSprite(sprFolder + "spr\_cannon\_red.png");

sprites.cannon\_green = Game.loadSprite(sprFolder + "spr\_cannon\_green.png");

sprites.cannon\_blue = Game.loadSprite(sprFolder + "spr\_cannon\_blue.png");

Add a variable to the game object:

Var Game = {

spritesStillLoading :0

};

Now every time you load a sprite. You increment the variable by 1. (in LoadSprite method)

Game.loadSprite = function(imageName) {

Var image = new Image();

Image.src = imageName;

Game.spritesStillLoading += 1;

Image.onload = function() {

Game.spritesStillLoading -= 1;

};

Return image;

}

Now everytime you load a sprite, spritesStillloading variable is incremented. Now decrement the variable every time a sprite is finished loading. Using event handler function as above.

spritesStillLoading variable represents how many sprites are still loading. When it reaches 0 you can start initialize() method and mainLoop() method.

Game.assetLoadingLoop = function() {

If(Game.spritesStillLoading > 0)

Window.setTimeout(Game.assetLoadingLoop, 1000/60);

Else {

Game.initialize();

Game.mainLoop();

}

};

Writing a More Efficient Game Loop:

Instead of using window.setTimeout() method, we can use window.requestAnimationFrame(callBackFunction) method.

You can extend definition of the function as follows:

window.requestAnimationFrame = window.requestAnimationFrame ||

window.webkitRequestAnimationFrame || window.mozRequestAnimationFrame ||

window.oRequestAnimationFrame || window.msRequestAnimationFrame ||

function (callback) {

window.setTimeout(callback, 1000 / 60);

};

Not writing the window is also fine. i.e. requestAnimationFrame(callbackfunction) is similar

Now you can do this:

Game.assetLoadingLoop = function () {

if (Game.spritesStillLoading > 0)

window.requestAnimationFrame(Game.assetLoadingLoop);

else {

Game.initialize();

Game.mainLoop();

}

};

Game.mainLoop = function () {

Game.handleInput();

Game.update();

Game.draw();

Mouse.reset();

window.requestAnimationFrame(Game.mainLoop);

};

Separating Generic code from Game-specific code:

By splitting generic code from code that is specific to Painter, it will be easier to reuse that generic code later.

Generic code includes: sprite-loading code, key-board mouse functions etc.

i.e. make files: Game.js and gamename.js.

Adding a ball to the Game world:

var ball = {

};

ball.initialize = function() {

ball.position = { x : 65, y : 390 };

ball.velocity = { x : 0, y : 0 };

ball.origin = { x : 0, y : 0 };

ball.currentColor = sprites.ball\_red;

ball.shooting = false;

};

You now load ball sprites.

Creating the ball:

ball.initialize = function() {

ball.position = { x : 0, y : 0 };

ball.velocity = { x : 0, y : 0 };

ball.origin = { x : 0, y : 0 };

ball.currentColor = sprites.ball\_red;

ball.shooting = false;

}; //initial color as red // Initial velocity to 0

ball.reset = function () {

ball.position = { x : 0, y : 0 };

ball.shooting = false;

}; //Reset ball position and it’s shooting status.

When ball goes outside screen after being shot, you can reset it by calling this method.

Now add draw method:

ball.draw = function () {

if (!ball.shooting)

return;

Canvas2D.drawImage(ball.currentColor, ball.position, ball.rotation,

ball.origin);

};

And in the painterGameWorld you can draw the ball.

painterGameWorld.draw = function () {

Canvas2D.drawImage(sprites.background, { x : 0, y : 0 }, 0,

{ x : 0, y : 0 });

ball.draw();

cannon.draw();

};

Shooting the ball:

The player can click the left mouse button in the game screen to shoot a ball of paint.

Speed and direction are determined by the position where the player clicks.

Add handleInput method to the ball object. Inside that, when left button is pressed

i.e.

if (Mouse.leftPressed)

//do something…

You can extend this with extra condition that the ball isn’t currently in the air:

If(Mouse.leftPressed && !ball.shooting)

//do something…

ball.shooting = true;

ball.velocity.x = (Mouse.position.x - ball.position.x) \* 1.2;

ball.velocity.y = (Mouse.position.y - ball.position.y) \* 1.2;

//that 1,2 value needs to be adjusted as needed to make game suitable to play.

Now add the handler function to handleInput method of painterGameworld object:

painterGameWorld.handleInput = function () {

ball.handleInput();

cannon.handleInput();

};

Updating the Ball:

Now you can add update and draw method to ball so that you can call them in game loop method of painterGameWorld.

ball.update = function (delta) {

if (ball.shooting) {

ball.velocity.x = ball.velocity.x \* 0.99;

ball.velocity.y = ball.velocity.y + 6;

ball.position.x = ball.position.x + ball.velocity.x \* delta;

ball.position.y = ball.position.y + ball.velocity.y \* delta;

}

else {

if (cannon.currentColor === sprites.cannon\_red)

ball.currentColor = sprites.ball\_red;

else if (cannon.currentColor === sprites.cannon\_green)

ball.currentColor = sprites.ball\_green;

else

ball.currentColor = sprites.ball\_blue;

ball.position = cannon.ballPosition();

ball.position.x = ball.position.x - ball.currentColor.width / 2;

ball.position.y = ball.position.y - ball.currentColor.height / 2;

}

if (painterGameWorld.isOutsideWorld(ball.position))

ball.reset();

};

It has one parameter delta: which is time passed. You can set it to 1/60 so you are assuming 1/60s has passed since last time. Fixed timestep method.

You can get system time too:

Var d = new Date();

Var n = d.getTime();

Variable n contains number of ms since January 1, 1970. You can use this now (different between previous and new time) . Variable timestep() method. But when the user goes to do something else or pauses, issue.

Game object types:

Creating multiple objects of the same type

Till now, there is one cannon and one ball.

Suppose we want to be able to shoot three balls at the same time. We could create three ball variables: ball1, ball2 and ball3. This is problematic as if you want to change behavior of balls, you need to modify behavior of three individual balls.

Solution: JS programming construct called prototypes. Prototypes allow us to create blueprint for an object, including variables and methods it contains. Once defined, you can create objects using that prototype with a single line of code.

Remember this: var image = new Image();

Example: Creating a ball prototype

function Ball() {

}

Ball.prototype.initialize = function() {

This.position = { x: 0 , y: 0};

This.velocity = {x: 0, y: 0};

// and other stuff;

};

Now you can create as many balls as you want and they will contain the initialize function:

Var ball1 = new Ball();

Var ball2 = new Ball();// and so on

You can do: ball2.initialize(); and that would work

You could shorten the code since Ball itself is a function:

Function Ball() {

This.position = {x: 0, y: 0};

This.velocity = {x:0, y:0};

//and other stuff;

}

Now balls are initialized on creation.

You can also pass parameters:

Function Ball(pos,color) {

//stuff here

}

Now you can pass color and position.

Because the ball function is responsible for initializing or constructing the object, this function is also called constructor. The constructor together with the methods defined in the prototype is called a class.

Thus, a class is a blueprint for an object, and such it consists of data that is contained within an object and methods that manipulate the data.

You must use this.sth instead of ball.sth in prototype.

So you can convert normal methods to prototype methods by replacing ball with this.

Why take the constructor approach:

1. It is hard to find global variable declaration. Using this approach, you could instantiate the object in one place.
2. Using different global variables means you are throwing away any structure or relationship that exists between variables. For e.g. Your code becomes easier to understand the relation if you express that by letting the game objects be part of game-world object. (Like cannon and ball could be part of another object called game-objects)

Writing a class with Multiple instances:

Function PainterGameWorld() {

This.cannon = new Cannon();

This.ball = new Ball();

This.can1 = new PaintCan(450);

This.can2 = new PaintCan(575);

This.can3 = new PaintCan(700);

}

PaintCan can be prototyped as follows:

Function PaintCan(xPosition) {

This.currentColor = sprites.can\_red;

This.velocity = new Vector2();

This.position = new Vector2(xPosition, -200);

This.origin = new Vector2();

This.reset();

}

Here, we are creating three PaintCan in PainterGameWorld constructor.

Dealing with Randomness in Games:

We can use random method which is part of Math object. It generates pseudorandom number between 0 and 1. Every number in the range has an equal chance of being generated. i.e. uniform distribution.

Var probability = Math.random();

You can generate a random number between a and b like this:

Math.random()\*(b-a) gives random value between 0 and (b-a) but we don’t want that.

We can shift both end by a towards right to get value between a and b:

Math.random()\*(b-a) + a // This is noice

If you want integer only you can do this:

Math.floor(Math.random()\*(b-a)+a+1) //Think of what the 1 does.

Representing positions and velocities as vectors:

Function Vector2(x,y) {

This.x = x;

This.y = y;

}

Var position = new Vector2(0,0);

It would be better if we were able to use some default value:

Function Vector2(x,y) {

This.x = typeof x!==’undefined’ ? x : 0; //i.e. 0 if undefined and given value if defined

This.y = typeof x!==’undefined’?y:0;

}

Now you can do this: Var position = new Vector2();

You can also add some methods to Vector2 class to make it easier to perform calculations:

Vector2.prototype.copy = function() {

return new Vector2(this.x, this.y);

};

You can do this: position2 = position1.copy();

Lets define a method for adding a vector to an existing vector:

Vector2.prototype.addTo = function(v) {

This.x = this.x + v.x;

This.y = this.y + v.y;

Return this;

};

Var position = new Vector2(10,10);

Var anotherPosition = new Vector2(20,20);

Position.addTo(anotherPosition).addTo(anotherPosition);

//position now represents vector (50,50)

You could do this:

Vector2.prototype.addTo = function (v) {

if (typeof v === 'Vector2') {

this.x = this.x + v.x;

this.y = this.y + v.y;

}

else if (typeof v === 'Number') {

this.x = this.x + v;

this.y = this.y + v;

}

return this;

};

Instead of typeof you could use constructor variable which is part of each object in JS:

Vector2.prototype.addTo = function(v) {

If(v.constructor === Vector2) {

This.x = this.x + v.x;

This.y = this.y + v.y;

}

Else if (v.constructor === Number) {

This.x = this.x + v;

This.y = this.y + v;

}

Return this;

};

You can define and add method which returns the sum as a new vector instead of changing the existing one:

Vector2.prototype.add = function(v) {

Var result = this.copy();

Return result.addTo(v);

}

You can do: var sum = position.add(anotherPosition);

Color and Collisions:

Colors can be represented using #rrggbb in hexadecimal

Controlled data access for objects.

Cannon, Ball and PaintCan each consist of certain color.

Consider a color variable:

var Color = {

aliceBlue: "#F0F8FF",

antiqueWhite: "#FAEBD7",

aqua: "#00FFFF",

aquamarine: "#7FFFD4",

azure: "#F0FFFF",

beige: "#F5F5DC",

bisque: "#FFE4C4",

black: "#000000",

blanchedAlmond: "#FFEBCD",

blue: "#0000FF",

blueViolet: "#8A2BE2",

brown: "#A52A2A",

// and so on

}

We could use this to represent color of cannon, ball or paint:

Function Cannon() {

This.position = new Vector2(72, 405);

This.colorPosition = new Vector2(55, 388);

This.origin = new Vector2(34, 34);

This.currentColor = sprites.cannon\_red;

This.color = Color.red;

This.rotation = 0;

}

Here, the color information is represented by two variables.

You could not store a reference to current sprite.

Function cannon() {

This.position…

colorPosition…

origin…

this.color =Color.red;

this.rotation = 0;

}

You could use two methods to read and write a color value.

Cannon.prototype.getColor = function() {

If(this.currentColor === sprites.cannon\_red)

Return Color.red;

Else if (this.currentColor === sprites.cannon\_green)

Return Color.green;

Else

Return Color.blue;

};

Cannon.prototype.setColor = function(value) {

If(value === Color.red)

This.currentColor = sprites.cannon\_red;

Else if (value === Color. Green)

This.currentColor = sprites.cannon\_green;

Else if (value === Color.blue)

This.currentColor = sprites.cannon\_blue;

};

Now the user of the Cannon doesn’t need to know that internally, you use a sprite to determine the current color of the cannon. The user can simply pass along a color definition to read or write the color of the cannon:

myCannon.setColor(Color.blue);

var cannonColor = myCannon.getColor();

These kinds of methods are called getters and setters. In many OOPL, methods are only ways to access data inside an object. JavaScript provides a feature called properties. A property defines what happens when you retrieve data from an object and what happens when you assign a value to data inside an object.

Read-only Properties.

Object named Object has a method called defineProperty which lets you add properties to objects.

It takes three parameters:

Prototype to which property should be added.

Name of property

Object containing at most two variables: get and set.

The get and set variable should point to a function that should be executed when the property is read or written.

Example:

Object.defineProperty(Cannon.prototype, “center”,

{

get: function() {

return new Vector2(this.currentColor.width/2, this.currentColor.height/2);

}

});

Now you can do this: var cannonCenter = cannon.center;

Similarly, you can use set which sets value. (doesn’t return)

Handling collisions between the Ball and the Cans.

When they collide, you have to handle it in update method of one of the classes.

Collision checking can be done in several way. One is to check the distance between their centers and if it is smaller than certain value, you can say there is a collision.

Limited Lives:

Maintaining number of lives:

Just add a variable to PainterGameWorld: this.lives = 5;

You can decrement it when required.

But you need to add targetColor to each PaintCan when they reach the bottom, check that and then decrement if required. Also you would need to reset the ball and the paintCan position.

Indicating number of lives to the player:

1. While statement:

While(condition)

{

//Block of code

}

So you could do this:

var i = 0;

while (i < numberOfLives) {

Canvas2D.drawImage(sprites.lives,

new Vector2(i \* sprites.lives.width + 15, 60));

i = i + 1;

}

1. For loop:

Var I;

For(i= begin value; i<end value; i++){

//do something

}

So:

for (var i = 0; i < this.lives; i++) {

Canvas2D.drawImage(sprites.lives,

new Vector2(i \* sprites.lives.width + 15, 60));

}

Restarting the game:

You can load an extra sprite on the screen:

Sprites.gameover = loadSprite(“spr\_gameover\_click.png”);

Draw this if game is over i.e. no lives left.

Now in the handleInput method in the PainterWorld

If (this.lives > 0) {

This.ball.handleInput(delta);

This.cannon.handleInput(delta);

}  
else {

If (Mouse.leftPressed)

This.reset();

}

Now you can add reset method to Painter .world class so you can reset the game to its initial state.

PainterGameWorld.prototype.reset = function () {

this.lives = 5;

this.cannon.reset();

this.ball.reset();

this.can1.reset();

this.can2.reset();

this.can3.reset();

};

You also need to change the update method so that update occurs only when game isn’t over.

if (this.lives <= 0)

return;

this.ball.update(delta);

this.cannon.update(delta);

this.can1.update(delta);

this.can2.update(delta);

this.can3.update(delta);

Organizing Game Objects:

You can represent similarities between game objects in JS.

For eg: ball and paintCan class have width and height properties.

Draw method of Ball and PaintCan classes look similar.

In case of the game: balls, paint cans and cannons are all game objects. All of them have position, can be drawn, have velocity(0 for cannon) , and they all have color. Also, most of them handle input of some kind and are updated.

Inheritance:

Using prototypes in JS makes it possible to group this similarities together in a sort of generic class and then define other classes that are a special version of this generic class. In OOP, this is called inheritance.

Consider this:

Function Vehicle() {

This.numberOfWheels = 4;

This.brand = “”;

}

Vehicle.prototype.what = function() {  
 return ‘nrOfWheels = ‘ + this.numberOfWheels + ‘, brand = ‘ + this.brand;

};

Now you can create a vehicle object:

Var v = new Vehicle();

v.brand = “Volkswagen”;

console.log(v.what()); // outputs “nrOfWheels = 4, brand = Volkswagen”

You can now create a car class with the same features of Vehicles + some more.

Function Car(brand) {

Vehicle.call(this); //Vehicle constructor is called, using the same object that is created when the car constructor is invoked. You are essentially telling the interpreter that the Car object(this) is also an Vehicle object.

This.brand = brand;

This.convertible = false;

}

Car.prototype = Object.create(Vehicle.protoype); //This makes a copy of prototype of Vehicle object and store that copy in prototype of object Car.

Now,

Var c = new Car(“Mercedes”);

Console.log(c.what()); //outputs “nrOfWheels = 4, brand = mercedes”

Game objects and Inheritance:

In case of game you can create a class ThreeColorGameObject with color,velocity,position,origin,rotation, and visible.

You can take three colors as parameters. You can create a prototype of the draw method, update method and add get and set properties.

So, you have this:

function ThreeColorGameObject(sprColorRed, sprColorGreen, sprColorBlue) {

this.colorRed = sprColorRed;

this.colorGreen = sprColorGreen;

this.colorBlue = sprColorBlue;

this.currentColor = this.colorRed;

this.velocity = Vector2.zero;

this.position = Vector2.zero;

this.origin = Vector2.zero;

this.rotation = 0;

this.visible = true;

}

ThreeColorGameObject.prototype.draw = function () {

if (!this.visible)

return;

Canvas2D.drawImage(this.currentColor, this.position, this.rotation, 1,

this.origin);

};

ThreeColorGameObject.prototype.update = function (delta) {

this.position.addTo(this.velocity.multiply(delta));

};

Object.defineProperty(ThreeColorGameObject.prototype, "color",

{

get: function () {

if (this.currentColor === this.colorRed)

return Color.red;

else if (this.currentColor === this.colorGreen)

return Color.green;

else

return Color.blue;

},

set: function (value) {

if (value === Color.red)

this.currentColor = this.colorRed;

else if (value === Color.green)

this.currentColor = this.colorGreen;

else if (value === Color.blue)

this.currentColor = this.colorBlue;

}

});

Cannon as a subclass of ThreeColorGameObject:

function Cannon() {

ThreeColorGameObject.call(this, sprites.cannon\_red, sprites.cannon\_green,

sprites.cannon\_blue);

this.position = new Vector2(72, 405);

this.origin = new Vector2(34, 34);

}

Cannon.prototype = Object.create(ThreeColorGameObject.prototype);

Cannon.prototype.handleInput = function (delta) {

if (Keyboard.down(Keys.R))

this.currentColor = this.colorRed;

else if (Keyboard.down(Keys.G))

this.currentColor = this.colorGreen;

else if (Keyboard.down(Keys.B))

this.currentColor = this.colorBlue;

var opposite = Mouse.position.y - this.position.y;

var adjacent = Mouse.position.x - this.position.x;

this.rotation = Math.atan2(opposite, adjacent);

};

In addition to adding new methods and properties, you can also choose to replace a method in the cannon class.

Cannon.prototype.draw = function () {

if (!this.visible)

return;

var colorPosition = this.position.subtract(this.size.divideBy(2));

Canvas2D.drawImage(sprites.cannon\_barrel, this.position, this.rotation, 1,

this.origin);

Canvas2D.drawImage(this.currentColor, colorPosition);

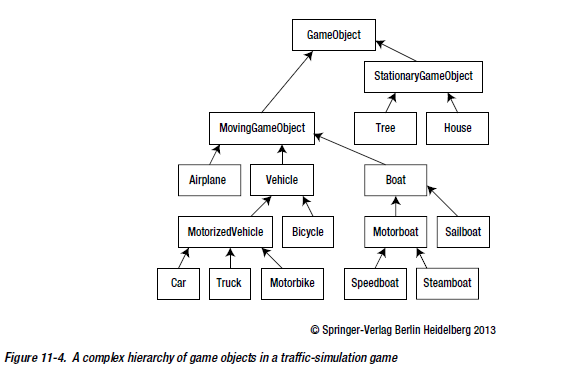
};

This overrides the original method in the ThreeColorGameObject class.

Also you could do this: Cannon.prototype.draw = undefined to remove the method.

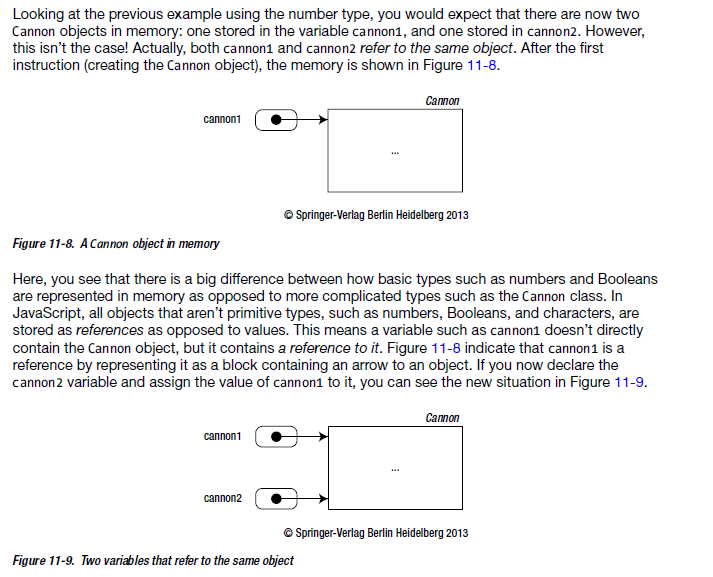
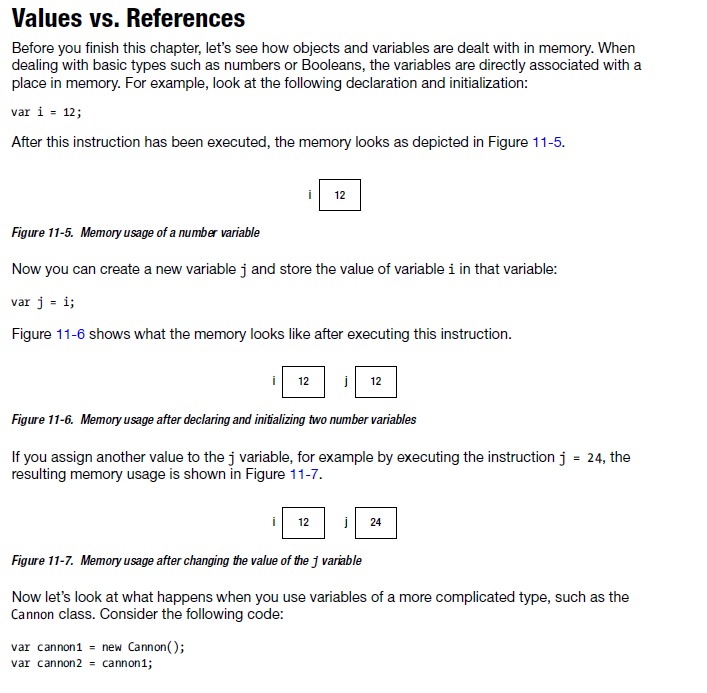
Polymorphism: different versions of methods can exist and the interpreter keeps track of which one to call.

Hierarchies of Classes:



Just an example.

Values vs References:

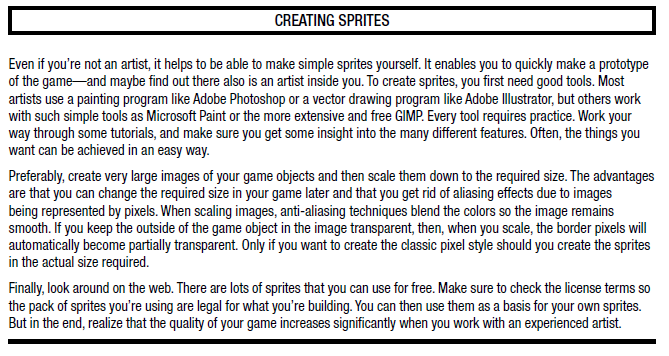


Finishing the Painter Game:

Adding motion effect:

To PaintCan.update you can add:

This.rotation = Math.sin(this.position.y/50) \* 0.05;

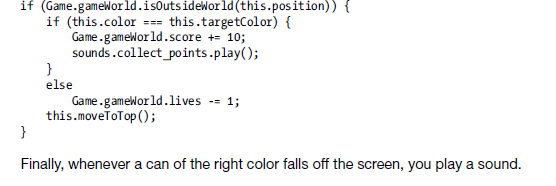


Adding Sounds and Music:

//lets read this at last

Maintaining a score:

Add a variable score to the PainterGameWorld in which to store the current score:



A More complete canvas2D class:

Canvas2D.drawText("Hello, how are you doing?", Vector2.zero, Color.green); can be used to draw

Character and strings:

Once string is created, it cannot be changed. Although it is possible to replace with another.

Everthing enclosed in quotes (single or double) is treated as string.

\n is for end of line. \t is for tabulation. \ can be sued as escape character. i.e. \\ works as backspace character.

Converting number to string happens automatically. i.e. if you use “string” + 5 . 5 is converted to string and concatenated.

Converting string to number can be done using parseInt() or parseFloat() function.

Adapting to Different Devices:

In order to let the canvas element scale automatically, you need to set its width and height to 100% of the browser window width and height.

html, body {

margin: 0;

}

#gameArea { //this is for div containing canvas element

position: absolute;

}

#mycanvas {

Position: absolute;

Width: 100%;

Height: 100%;

}

Game.start can be passed native game size.

Game.size can be set using the native size.

You can add a event listener:

Window.onresize = Canvas2D.resize; and Canvas2D.resize() in Canvas2D.initialize() method.

In Canvas2D.resize method:

Calculate aspectRatio using gameSize.

Calculate new aspectRatio.

If new aspectRatio is higher, calculate newWidth based on newHeight.

Else calculate newHeight based on newWidth.

Now use: Canvas2D.div.style.width = newWidth + ‘px’; and same for height to change size of div containing canvas.

Set margin of the div similarly based on new height and width and window.innerHeight and window.innerHeight to center the div.

Now set width and height of canvas to new value.

Create a method getScale() that returns Vector2d(canvas.width/Game.size.x, canvas.height/Game.size.y);

You can use getScale to get scale and change scale of CanvasContext in drawImage method:

Var canvasScale = this.scale;

Canvas2D.scale(canvasScale.x, canvasScale.y);

You can do the same in drawText method too.

Redisigning Mouse-input Handling:

Why?:

We have assumed canvas drawn at top left on the screen. So, you need to take position of canvas into account.

You also need to take scale into account.

Function handleMouseMove(evt) {

Var canvasScale = Canvas2D.scale;

Var canvasOffset = Canvas2D.offset;

Var mx = (evt.pageX – canvasOffset.x) / canvasScale.x;

Var my = (evt.pageY – canvasOffset.y) / canvasScale.y;

Mouse.position = new Vector2d(mx,my);

}