Modern Web Browsers and TypeScript

CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

PREV NEXT

OUTLINE

Search...

Modern Web Browsers

In order to build a **Modern Web Application**, it's very important to deeply understand how **Modern Web Browsers** work and what they do with the code we write.

At the core of the computer are the Central Processing Unit **CPU** and the Graphics Processing Unit **GPU**.

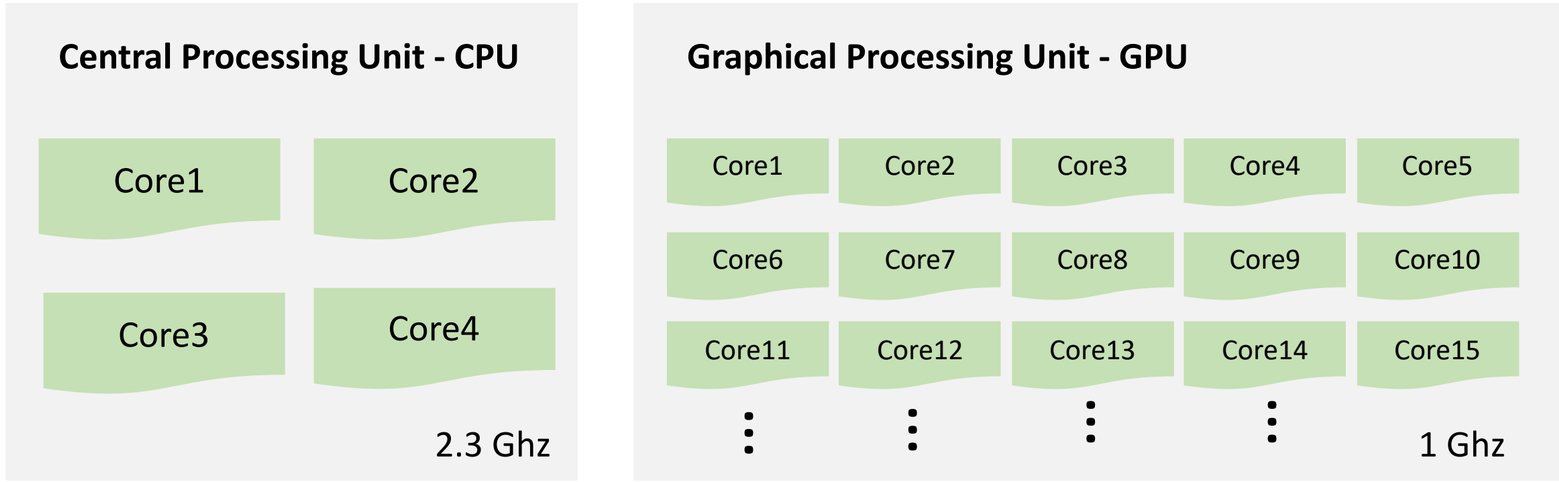
Applications run on the CPU and GPU using mechanisms provided by the Operating System.

google.com

PREV NEXT

OUTLINE

CPU vs GPU



- CPU performs **simple** central works, while GPU performs **complex** calculations (render animations, images, and videos).
- CPU commonly has 4–8 fast and flexible cores clocked at 2.3Ghz whereas GPU has thousands of relatively simple cores clocked at about 1Ghz.
- CPU is good for processing **sequential** works whereas GPU is very good at processing **parallel** tasks.

PREV NEXT

Process vs Thread

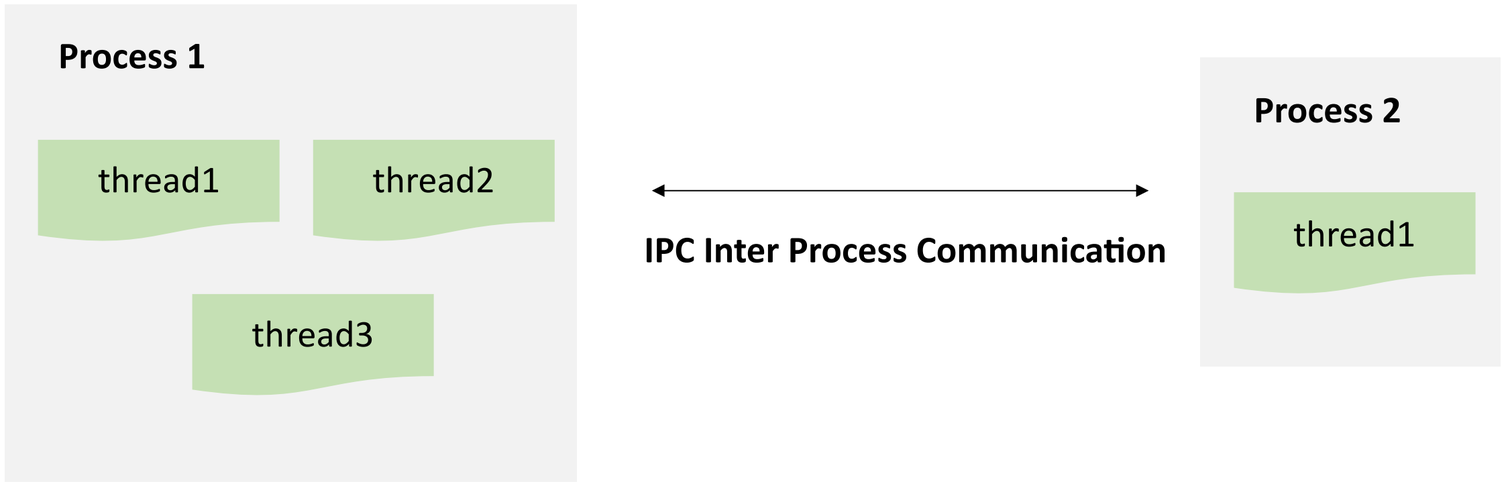
A **Process** is an application's executing program lives in memory.

A **Thread** lives inside of process and executes any part of its process's program.

When you start an application, a process is created. The program might create thread(s) to help it do work. The OS gives the process a space of memory to work with and all application state is kept in that private memory space. When you close the application, the process goes away and the OS frees up the memory.

A process can ask the OS to fork another process to run different tasks. When this happens, different parts of the memory are allocated for the new process. If two processes need to talk, they can do so by using **Inter Process Communication (IPC).**

Application in Memory



Browser Architecture

A web browser can be built using one process with many different threads, or with many different processes with a few threads communicating over IPC.

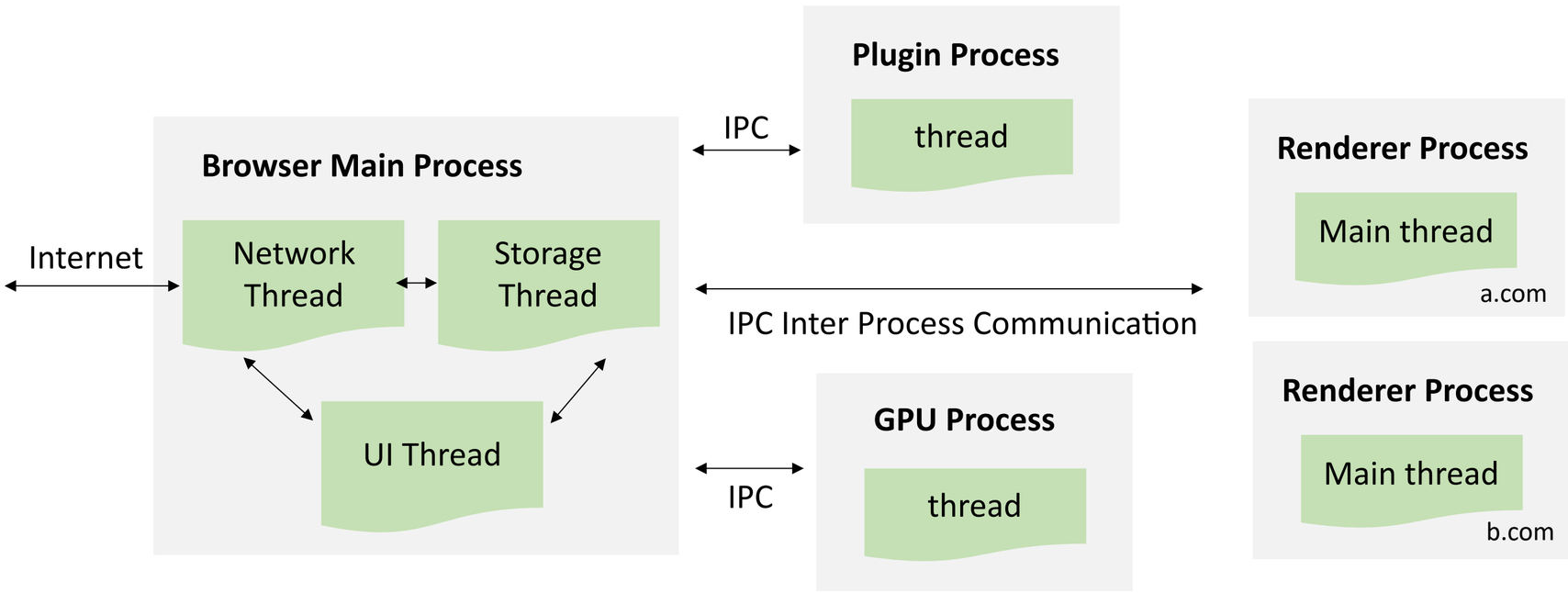
Chrome Architecture: The main browser process coordinates with other processes that take care of different parts of the application (plugin processes, GPU process, renderer processes, utility process). For the renderer process, multiple processes are created and assigned to each tab (per domain). Chrome gives each site its own process (See Task Manager).

Chrome Processes

- **Browser Main Process** Controls the address bar, bookmarks, back and forward buttons. Also handles other parts of a web browser such as network requests and file access.
- **Renderer Process** Controls anything inside of the tab where a website is displayed (Represents the DOM for each tab).
- **Plugin Process** Controls plugins used by the website.
- **GPU Process** Handles GPU requests from multiple apps.

PREV NEXT

Chrome Architecture



Multi-core processes in Chrome

Responsiveness In most cases, you can imagine each tab has its own renderer process. Let’s say you have 3 tabs and each tab is run by an independent renderer process. If one tab becomes unresponsive, then you can close it while keeping other tabs alive.

Security Because processes have their own private memory space, they often contain copies of common infrastructure (like V8 which is a Chrome's JavaScript engine, each tab has its own scope).

Site Isolation Running a separate renderer process for each cross-site iframe, making sure one site cannot access data from other sites without consent (Same Origin Policy).

Navigation Phase

You type a URL into a browser, the browser fetches data from the internet. Everything outside of a tab is handled by the **Browser process**: **UI thread** (navigation buttons and input address bar of the browser), **Storage thread** (access files, cache), **Network thread**.

UI thread needs to parse and decide whether to send you to a search engine, or to the site you requested.

When a user hits enter, the **UI thread** initiates a network call to get site content. The **network thread** goes through appropriate protocols like DNS lookup and establishing TLS Connection for the request.

Receiving the Response

Once the response starts to come in, the **network thread** looks at the first few bytes of the stream and starts parsing the header and body. The response **Content-Type header** should say what type of data it is.

If the response is an HTML file, then the **network thread** will need to find and pass the data to a **renderer process**, but if it is a zip file or some other file then that means it is a download request so they need to pass the data to download manager.

Cross Origin Read Blocking (CORB) check happens in order to make sure sensitive cross-site data does not make it to the renderer process.

PREV NEXT

OUTLINE

Search...

Finding a Renderer Process

The **Network thread** tells **UI thread** that the data is ready (Address bar is updated, the security indicator and site settings UI reflects the site information of the new page). **UI thread** then finds/starts a **Renderer process** to carry on rendering of the web page.

An IPC is sent from the **Browser process** to the **Renderer process (main thread)** to commit the navigation. It also passes on the data stream so the Renderer process can keep receiving HTML data.

Once the Browser process hears confirmation that the commit has happened in the Renderer process, the navigation is complete and the Document Loading Phase begins.

PREV NEXT

OUTLINE

Search...

Finishing Navigation Phase

Once the navigation is committed, the **Renderer process** carries on loading resources and renders the page. Once the renderer process finishes rendering, it sends an IPC back to the Browser process. At this point, the UI thread stops the loading spinner on the tab.

Note: Client side JavaScript could still load additional resources and render new views after this point.

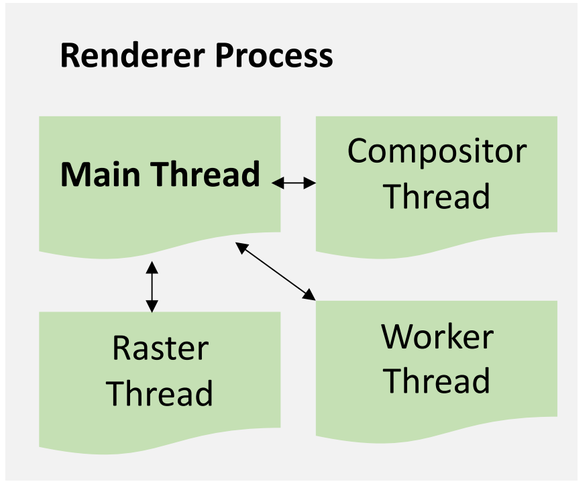
Renderer Process

The **Renderer process** is responsible for everything that happens inside of a tab.

The **Main thread** handles most of the code you send to the user.

Compositor and Raster threads are also run inside of a Renderer processes to render a page efficiently and smoothly.

The Renderer process core job is to turn HTML, CSS, and JavaScript into a web page that the user can interact with.



Parsing

Construction of a DOM

When the Renderer process receives a commit message for a navigation and starts to receive HTML data, the Main thread begins to parse the text string (HTML) and turn it into a Document Object Model (DOM).

Sub-Resource Loading

All external resources (images, CSS, and JavaScript) need to be loaded from network or cache. The Main thread could request them and sends requests to the Network thread in the Browser process.

JavaScript can block the parsing

When the HTML parser finds a `<script>` tag, it **Pauses** the parsing of the HTML document and has to load, parse, and execute the JavaScript code.

PREV NEXT

Style Calculation

Having a DOM is not enough to know what the page would look like because we can style page elements in CSS. The Main thread parses CSS and determines the **computed style** for each DOM node.

The browser has a default style sheet.

Layout

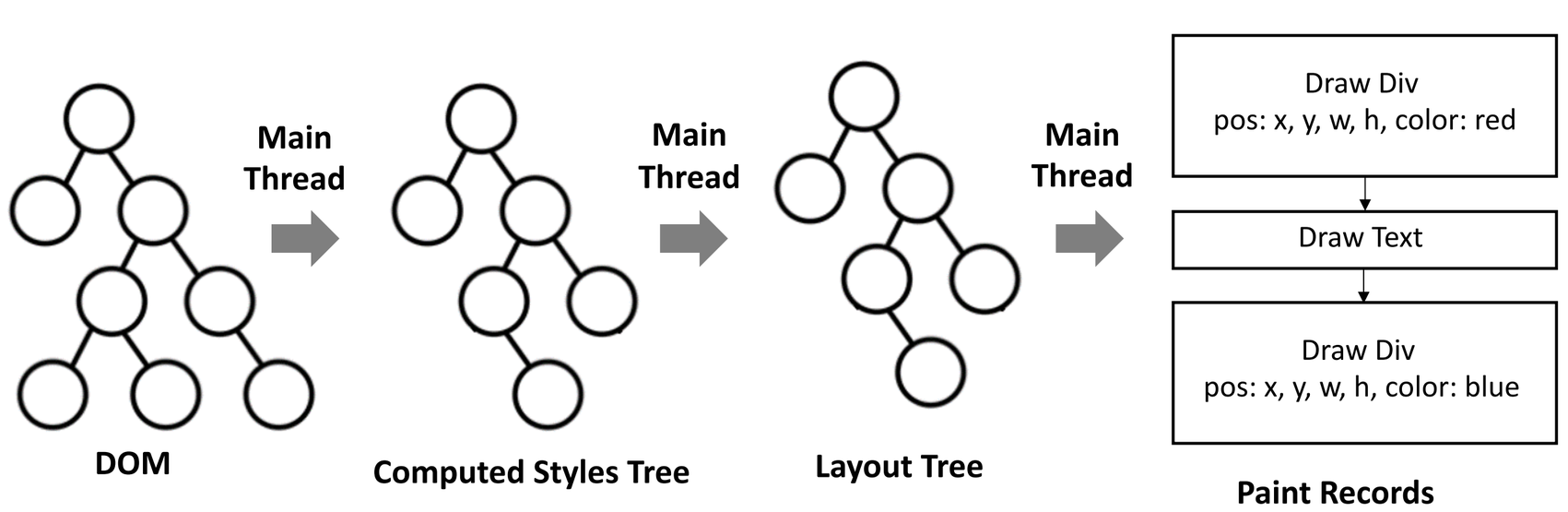
The layout is a process to find the geometry of elements. The Main thread walks through the DOM and computed styles and creates the **Layout Tree** which has information like x y coordinates and bounding box sizes. Layout tree may have similar structure to the DOM tree, but it only contains information related to **what's visible on the page**.

- If `display:none` is applied, that element is not part of the layout tree.
- An element with `visibility:hidden` is in the layout tree.
- If a pseudo class with content like `p::before{content:"CS572"}` is applied, it is included in the layout tree even though that is not in the DOM.

PREV NEXT

Paint

At this paint step, the main thread walks the layout tree to create **Paint Records**. Paint record is a note of painting process like "background first, then add text, then draw a rectangle".



Compositing

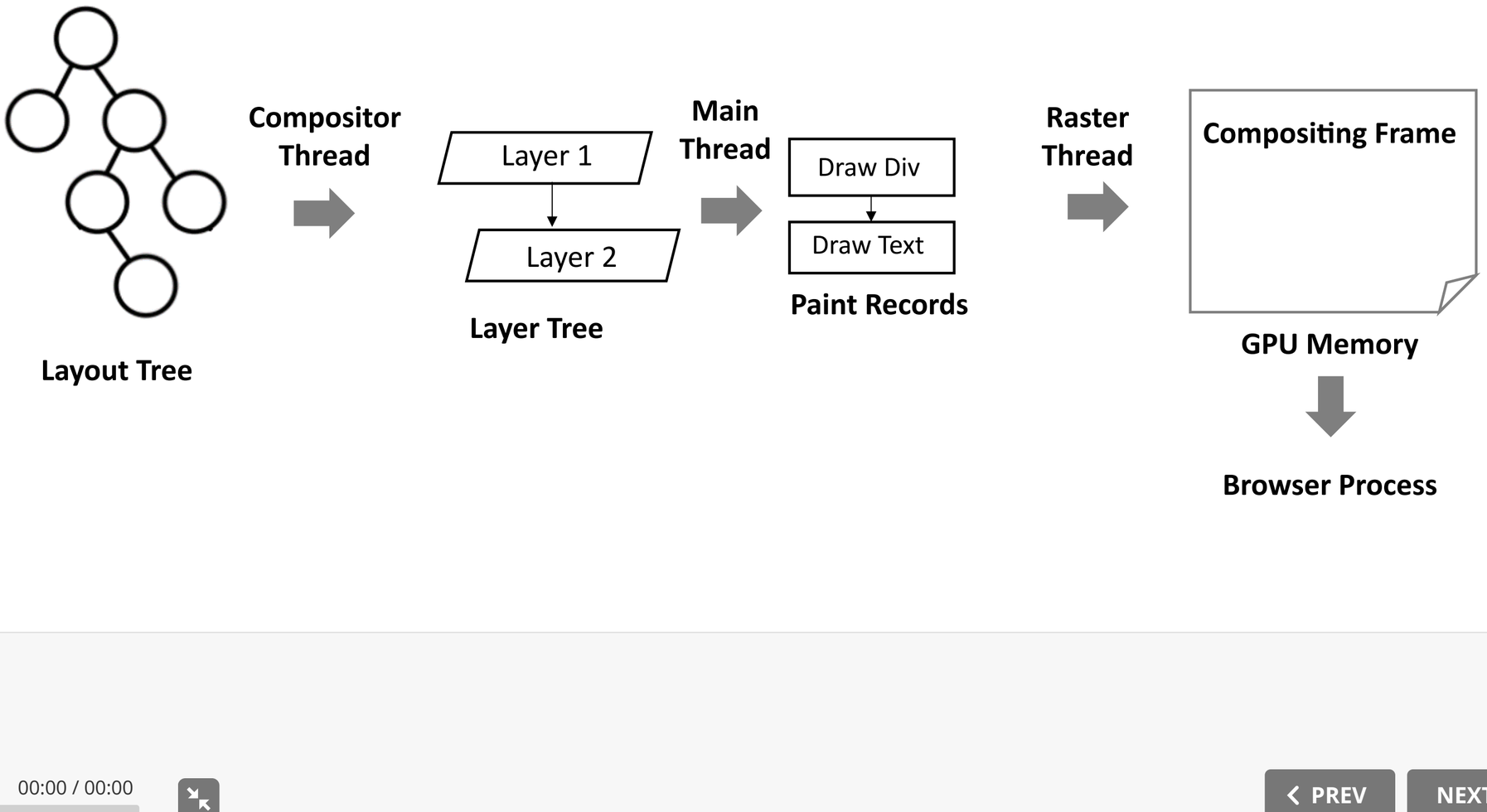
After generating paint records, we will turn it into pixels on the screen, this is called **Rasterizing**.

We can raster only the parts inside of the viewport and when a user scrolls the page, fill in the missing parts by rastering more.

Compositing is a technique to separate parts of a page into layers, rasterize them separately. Composite for a page happens in a separate thread called **Compositor thread**. In order to find out which elements need to be in which layers, the main thread walks through the **Layout Tree** to create the **Layer Tree.**

PREV NEXT

Compositing and Rastering



Raster thread

Once the **Layer Tree** is created and **Paint records** are determined, the **Main thread** commits that information to the **Compositor thread**. The Compositor thread then **Rasterizes** each layer.

A layer could be large like the entire length of a page, so the Compositor thread divides them into tiles and sends each tile off to Raster threads. **Raster threads** rasterize each tile and store them in **GPU memory**. Once tiles are rastered, Compositor thread creates a **Frame**.

The **Frame** is then submitted to the **Browser process** via IPC. These compositor frames are sent to the **GPU** to display it on a screen. If a scroll event comes in, Compositor thread creates another compositor frame to be sent to the GPU.

Input Events

Input Events mean any gesture from the user. Mouse wheel scroll is an input event and touch or mouse over is also an input event.

When user gesture like touch on a screen occurs, the **Browser process** is the one that receives the gesture at first. Then it sends the event type (like touchstart) and its coordinates to the **Renderer process** which handles the event appropriately by finding the event target and running event listeners that are attached.

Updating the DOM is costly

The most important thing to understand is that every time you make a change to the DOM, the rendering pipeline will be executed.

If we are animating elements or playing a video, the browser has to run the DOM rendering pipeline in between every frame. Even if our rendering operations are keeping up with screen refresh, these calculations are running on the **Main thread**, which means it could be blocked when your application is running JavaScript.

Frameworks like Angular and React, do not allow developers to access the DOM directly. **Instead, when changes are made to the App state, the framework patches the DOM efficiently for us.**

Be Kind to the Browsers

Remember that JavaScript is the Main thread's job and you do not want to block the main thread.

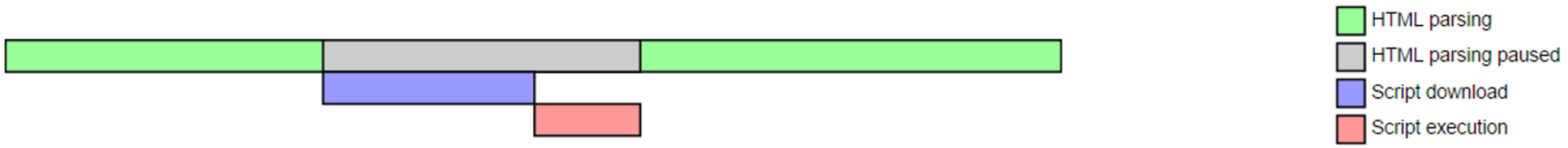
PREV NEXT

Script

```
<script src="myscript.js"></script>
```

This is the default behavior of the <script> element. Parsing of the HTML code pauses while the script is executing. The browser will run the script immediately after it arrives, before rendering the elements that's below your script tag.

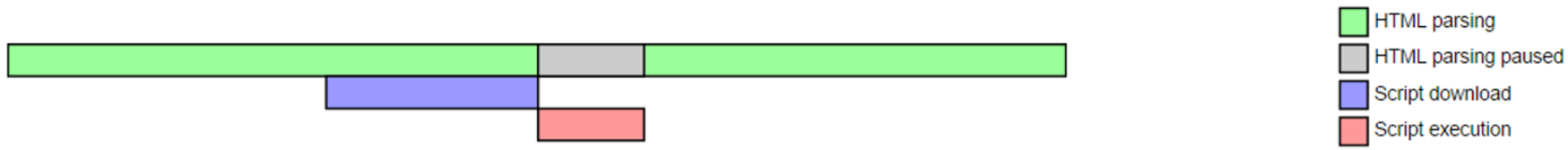
For slow servers and heavy scripts this means that displaying the webpage will be delayed.



Async

```
<script async src="myscript.js"></script>
```

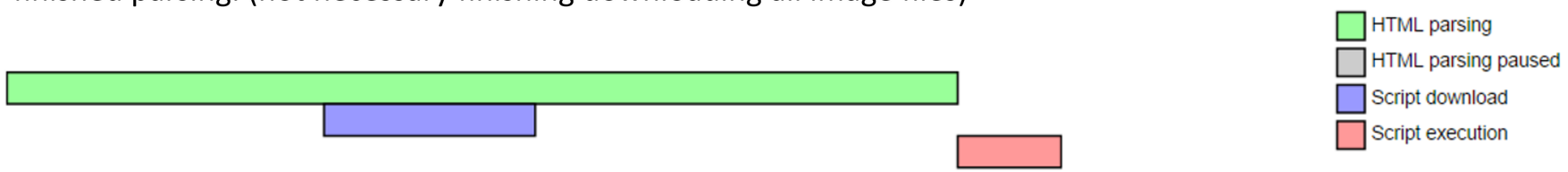
The browser will continue to load the HTML page and render it while the browser load and execute the script at the same time. Use it when you don’t care when the script will be available.



Defer

```
<script defer src="myscript.js"></script>
```

Delaying script execution until the HTML parser has finished. The browser will run your script when the page finished parsing. (not necessary finishing downloading all image files)



Resource Prioritization

Not every byte that is sent to the browser has the same degree of importance. Browsers have heuristics that attempt to make a best-guess at the most important resources to load first — such as CSS before scripts and images. (priority would change to **`Low`** if it has the **`async`** attribute)

Name	Status	Type	Initiator	Size	Time	Priority	Waterfall 30.00 s
www.youtube.com	200	document	Other	474 KB	1.80 s	Highest	
desktop_polymer_selective_initializa…	200	document	(index)	101 KB	133 ms	High	
scheduler.js	200	script	(index)	2.4 KB	124 ms	High	
www-tampering.js	200	script	(index)	3.4 KB	124 ms	High	
www-prepopulator.js	200	script	(index)	458 B	124 ms	High	
spf.js	200	script	(index)	11.9 KB	125 ms	High	
network.js	200	script	(index)	4.6 KB	124 ms	High	
web-animations-next-lite.min.js	200	script	(index)	14.3 KB	124 ms	High	
www-onepick-2x-webp-vflsYL2Tr.css	200	stylesheet	(index)	522 B	124 ms	Highest	
desktop_polymer.js	200	script	(index)	366 KB	173 ms	High	
photo.jpg	200	jpeg	(index)	3.0 KB	44 ms	Low	
photo.jpg	200	jpeg	(index)	4.7 KB	52 ms	Low	

https://developers.google.com

Preload, Preconnect

Preload informs the browser that a resource is needed as part of the current navigation, and that it should start getting fetched as soon as possible. Use it with web fonts and critical path CSS and JavaScript.

```
<link rel="preload" as="script" href="super-important.js">
<link rel="preload" as="style" href="critical.css">
```

Preconnect informs the browser that your page intends to establish a connection to another origin, and that you'd like the process to start as soon as possible. Use it when you know where from, but not what you are fetching, also good for establishing a connection for media streams.

```
<link rel="preconnect" href="https://example.com">
```

Prefetch

Prefetch is somewhat different from Preload and Preconnect, it doesn't try to make something critical happen faster, instead, it tries to make something non-critical happen earlier, if there's a chance.

It does this by informing the browser of a resource might be needed later, if the user takes the action we're expecting. These resources are fetched at the Lowest priority in Chrome, when the current page is done loading and there's bandwidth available.

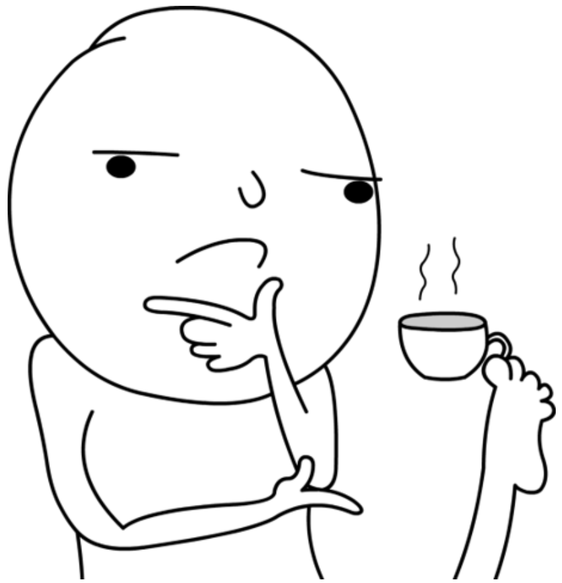
```
<link rel="prefetch" href="page-2.html">
```

Quiz!

Is it ever possible that

`(a==1 && a==2 && a==3)`

could evaluate to **`true`** in JavaScript?



What is TypeScript?

TypeScript is a free and open-source programming language developed and maintained by **Microsoft**. It is a strict **superset of JavaScript**, and adds optional static typing and class-based object-oriented programming to the language.

TypeScript is JavaScript with **static typing**. Basically the compiler knows what the variable's type is during compilation.

It's the official language adopted by the **Google Angular Team** to write Angular projects.

What do we get with TypeScript?

There are five big improvements that TypeScript bring over ES5:

- Types
- Classes
- Decorators (Annotations)
- Imports
- Language utilities (e.g. destructuring)

PREV NEXT

OUTLINE

Search...

Why Types?

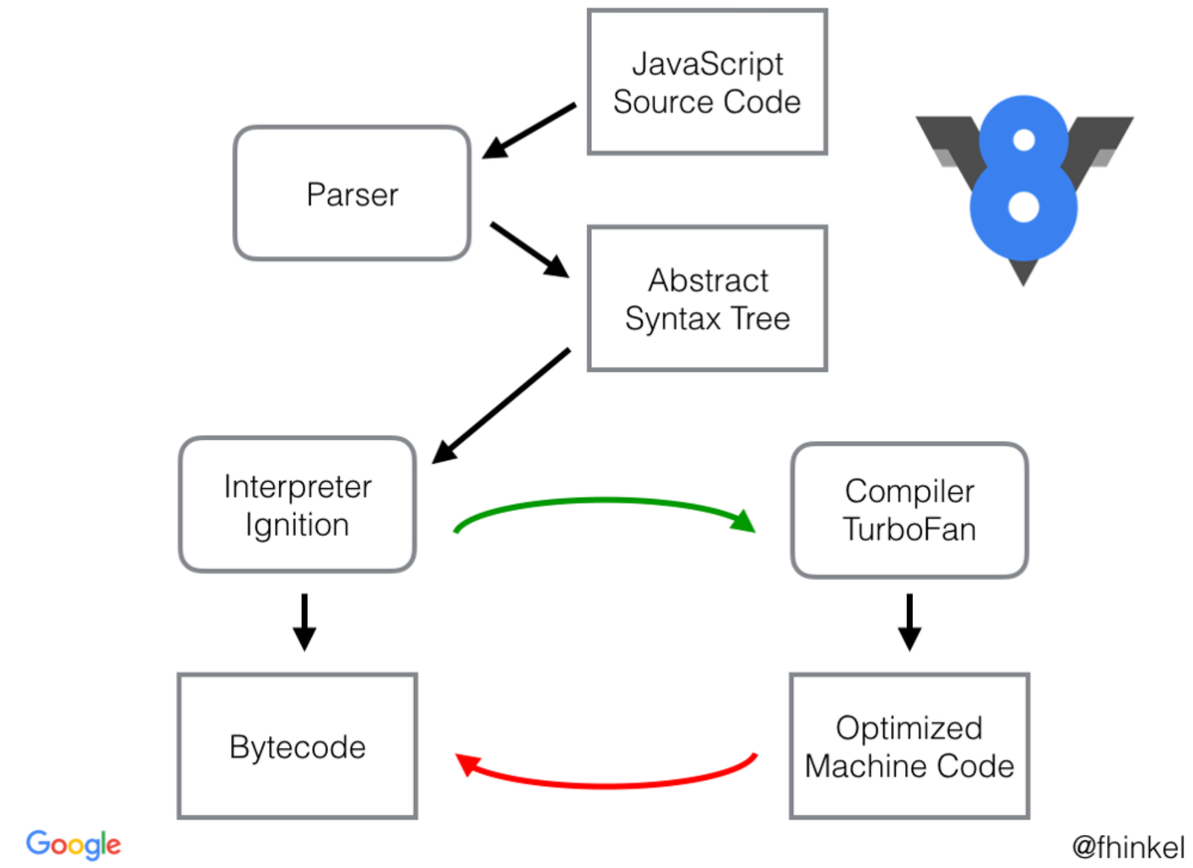
One of the great things about type checking is that:

1. It helps writing safe code because it can prevent bugs at compile time.
2. Compilers can improve and run the code faster.

It’s worth noting that types are **optional** in TypeScript.

PREV NEXT

V8



TypeScript Examples

TypeScript allows us to write really clean code that will compile down to ES5 (or any other version of JS) and be supported by all browsers. Even if you didn't want to use all the TypeScript features, you could use it mainly as a transpiler and stick to vanilla JS.

```
// Typescript
class Greeter {}
```

```
// JavaScript ES5
var Greeter = (function() {
        function Greeter() {}
        return Greeter;
}());
```

Types

We define types by using a **colon**

```
let courseName: string = "CS572"; // explicit type "string"
let courseName = "CS572"; // implicit type "string"
```

Typescript also gives us inference, which means when there is no type the compiler will guess what type we want:

```
let courseName = "CS572";
courseName = ["CS472", "CS572"]; // error
```

The compiler can also infer function return types:

```
let userId = (a: string, b:number): string => a + b; // explicit return type
let userId = (a: string, b:number) => a + b; // implicit return type
```


PREV NEXT

Basic Types

```
let courseName; // implicit type "any"

let hobbies: any[] = ["Cooking", "Sports"];
hobbies = [100]; // okay
hobbies = 100; // error
```

If we don't add a type to a variable and don't give it a value then the compiler will give it a value **undefined** and because it can't infer what the type is, it gives it type of **any**

```
let hobbies : any[];
let hobbies : Array<any>;
```

Tuple types allow you to express an array where the type of a fixed number of elements is known

```
let address: [string, number] = ["Superstreet", 99]; // tuple

enum Color {
        Gray, // 0
        Green = 100, // 100
        Blue }
let myColor: Color = Color.Blue;
console.log(myColor); // 101
```

44

Function Type and Object Type

```
// function types
let Multiply: (a: number, b: number) => number;
```

Param name does not matter, only type matters!

Key names matter, their value must match the type, key order does not matter!

```
// complex object
let complex: { data: number[], output: (all: boolean) => number[] } =
    {
        data: [1, 2, 3],
        output: function (all: boolean): number[] { return this.data; }
    };
```

Union Types and Type Aliases

```
let name: string | string[];

let func1 = (p: string | number | string[] | boolean) => p;
let func2 = (p: string | number | string[] | boolean) => p;

type myCustomType = string | number | string[] | boolean;
let func1 = (p: myCustomType) => p;
let func2 = (p: myCustomType) => p;
```

46

PREV NEXT

OUTLINE

Search...

String Literal Type

```
type courseType = "CS472" | "CS572";

function study(studentID: number, course: courseType) {
        console.log(`Student ${studentID} is studying ${course}`);
};

study(9812345, "CS123"); // Error
study(9812345, "CS572"); // Okay
```

Using Interfaces to Describe Types

- Interfaces describe the shape of a value but don't contain definitions.
- They can be set inline or as a type that can be referenced. Both options are useful.
- Interfaces help with typos and errors. They can have optional properties. They can be used to type other interface properties or extend it.

PREV NEXT

Using Interfaces

```
let course1: { name: string; code: number; project: boolean };
let course2: { name: string; code: number };
```



```
interface CourseInterface {
        name: string;
        code: number;
        project?: boolean;
}
let course1: CourseInterface = {
        name: "Web Applications Programming",
        code: 472
}
let course2: CourseInterface = {
        name: "Modern Web Applications",
        code: 572,
        project: true
}
```


PREV NEXT

Creating a Class in TypeScript

A typescript **`class`** is a function constructor. Functions are objects, so they can have properties. Classes can also have methods. You don't use the word function or a colon when defining a class method, just the name of the method, parens, and curly brackets.

In TypeScript you can have only one **`constructor`** per class.

PREV NEXT

Class Example

```
interface Book {
        bookName: string;
        isbn: number;
}
class Course {
        name: string;
        code: number;

        constructor(name: string, code: number) {
                this.name = name;
                this.code = code;
        }

        useBook(book: Book) {
                console.log(`Course ${this.name} is using the textbook:
                                ${book.bookName} who's ISBN = ${book.isbn}`);
        }
}
```

Constructor methods must be named **constructor**. They can optionally take parameters but they can't return any values. When a class has no constructor defined explicitly, one will be created automatically.

Class Example - Shortcut

```
interface Book {
        bookName: string;
        isbn: number;
}
class Course {
        constructor(public name: string, public code: number) {}

        useBook(book: Book) {
                console.log(`Course ${this.name} is using the textbook:
                        ${book.bookName} who's ISBN = ${book.isbn}`);
        }
}
```

Adding **access modifiers** to the constructor arguments lets the class know that they're properties of a class. If the arguments don't have access modifiers, they'll be treated as an argument for the constructor function and not properties of the class.

PREV NEXT

Inheritance

```
class Person {
        name: string;
        private type: string;
        protected age: number = 27;
        constructor(name: string, public username: string) {
                this.name = name;
        }
        printAge() {
                console.log(this.age);
                this.setType("Old Guy");
        }
        private setType(type: string) {
                this.type = type;
                console.log(this.type);
        }
}
class Asaad extends Person {
        constructor(username: string) {
                super("Asaad", username);
                this.age = 36;
        }
}
```

```
const person = new Person("Asaad", "asaad");
console.log(person.name, person.username);
person.printAge();
person.setType("Cool guy"); // error
const asaad = new Asaad("asaad");
```

Method name is setType, this is NOT a setter!

If you don't define a constructor, the derived class will use the base class's constructor. If you do define the constructor in a derived class, **super() must be called before anything else can happen**.

Getters and Setters

```
class Plant {
        private _species: string = "Default";
        get species() {
                return this._species;
        }
        set species(value: string) {
                if (value.length > 3) {
                        this._species = value;
                } else {
                        this._species = "Default";
                }
        }
}
let plant = new Plant();
console.log(plant.species);
plant.species = "Green Plants";
```

"species" is NOT a function! It's a property

To make a property read only:
- Make it private
- Provide only getter, no setter

Another way is:
- Make it public
- Add `readonly` to it:
  `public readonly name: string;`

Static Properties & Methods

```
class Helpers {
        static PI: number = 3.14;
        static calcCircumference(diameter: number): number {
                return this.PI * diameter;
        }
}

console.log(2 * Helpers.PI);
console.log(Helpers.calcCircumference(8));
```

if we try to call the static method on the instance, it's unavailable. The static method is only available on the class.

Abstract Classes

```
abstract class Project {
        projectName: string = "Default Project";
        budget: number = 0;

        abstract changeName(name: string): void;

        calcTwoPercentTax() {
                return this.budget * 2 / 100;
        }
}

class FinalProject extends Project {
        changeName(name: string): void {
                this.projectName = name;
        }
}
let newProject = new FinalProject();
newProject.changeName("CS572 Project");
```

- Cannot be initialized
- No full implementation (setup class)
- Any class extends from it must implement all abstract methods

Implement Abstract method

Decorators

- Decorators are functions called **when the class is declared** (compile time)—not when an object is instantiated (runtime).
- Decorators will change or add functionality to its destination.
- Multiple decorators can be applied on the same Class/Property/Method/Parameter.
- Decorators are not allowed on constructors.

In order for TS to understand decorators we should add this to the `tsconfig.json` *(proposed for ES7)*

```
"compilerOptions":{"experimentalDecorators": true}
```

Decorator Pattern

```
const course = { name: 'CS572' };

function addLevel(obj){
   return {
        level: 500,
        name: obj.name
    }
}

const decoratedObj = addLevel(course);

console.log(decoratedObj); // object {level: 500, name: 'CS572'}
```


PREV NEXT

Factory Decorator Pattern

```
const course = { name: 'CS572' };

function addLevel(level){
   return function(obj){
       return {
          level: level,
          name: obj.name
       }
   }
}

const decoratedObj = addLevel(500)(course);

console.log(decoratedObj); // object {level: 500, name: 'CS572'}
```


PREV NEXT

Simple Decorator in TS

```
@addLevel
class Course { name = "CS572" }

function addLevel(targetClass){
   return class {
        level = 500;
        name = new targetClass().name;
    }
}

console.log(new Course()); // object {level: 500, name: 'CS572'}
```

Factory Decorator in TS

```
@addLevel(500)
class Course { name = "CS572" }

function addLevel(val){
   return function(targetClass){
        return class {
            level = val;
            name = new targetClass().name;
        }
   }
}

console.log(new Course()); // object {level: 500, name: 'CS572'}
```


PREV NEXT

Decorator Composition

```
function printable(targetClass) {
        targetClass.prototype.print = function () {
                                        console.log(this);
                                    }
}

@logging(false)
@printable
class Plant {
        name = "Green Plant";
}

const plant = new Plant();
plant.print();
```

We can use multiple decorators

When we have two decorators @A and @B they are similar to A(B(class))

- A evaluated
- B evaluated
- B called
- A called

Install TypeScript Transpiler Globally

```
npm install -g typescript
tsc –v
```

```
tsc app.ts // will transpile app.ts to app.js
```

(It will convert the class defined to what a class would be defined as in ES5)

Using TypeScript config (tsconfig.json)

To create a TypeScript configuration file:

```
tsc -init //it creates a tsconfig.json
```

Since all configurations live in this file we could simply type **`tsc`** it's going to automatically find all **`*.ts`** and compile them to JavaScript.

```
{
   "compilerOptions": {
       "module": "commonjs",
       "target": "es5",
       "noImplicitAny": false,
       "noEmitOnError": true,
       "sourceMap": true,
       "outDir": "./js",
   },
   "exclude": [ "node_modules" ]
}
```

tsconfig.json

Very useful when debugging in browser!

Click here for full list of options

71

TypeScript and modules

Depending on the module target specified during compilation, the TypeScript compiler generates appropriate code and supports many kinds of **module-loading systems**:

- Node.js (CommonJS)
- require.js (AMD)
- isomorphic (UMD)
- SystemJS
- ECMAScript 2015 native modules (ES6)

PREV NEXT

ES2015/ES6 Modules

Synchronous and asynchronous loading supported.

```
//------ lib.js ------
const x = 5;
export function multiplyByFive(val) {
        return val * x;
}

//------ main.js ------
import { multiplyByFive } from 'lib';
console.log(multiplyByFive(2)); // 10
```

Named Exports

	CommonJS	ES6
Export	`module.exports.foo = 'foo'` `module.exports.baz = 'baz'`	`export var foo = 'foo'` `export var baz = 'baz'`

Import	`var f = require('myModule').foo;`
	`var b = require('myModule');`
	`b.baz;`
		`import {baz} from 'myModule'`
		`import {foo, baz} from 'myModule'`
		`import {foo as f, baz} from 'myModule'`
		`import * as m from 'myModule'`

In the ES6 module system, strict mode is turned on by default.

The need for a Bundler!

Unfortunately none of the major JavaScript runtimes support ES2015 modules in their current stable branches. This means no support in Firefox, Chrome or Node.js.

Many transpilers do support modules and a polyfill is also available.

Solutions to use modules in browsers today are: Gulp, Browserify, **Webpack**

PREV NEXT

Webpack – Module Bundler

It bundles our code into one JS file. This file will still need compilation in order to make our app to work (to be done later in the browser JiT).