الجزء الأول => ليش أصلاً في Event Loop؟

الفكرة الأساسية:
جافاسكربت **لغة single-threaded**. يعني تشتغل في **خط واحد فقط**:

- تقدر تنفّذ مهمة واحدة فقط بنفس اللحظة.
- ما تقدر تعمل حاجتين بنفس الوقت (مو زي Java أو C++ اللي عندها multi-threads).

طيب... لو جافاسكربت ما بدها تتوقف عن تنفيذ الكود، كيف رح تتعامل مع:

- events مثل click, mousemove
- مؤقّتات مثل setTimeout
- عمليات شبكة مثل fetch
- تحميل صور
- تنفيذ وعود Promise

هنا يجي دور **Event Loop**.

الجزء الثاني => فكرة الـ Event Loop ببساطة

تخيّلي عندك **عمّال في مطبخ**:

- عامل واحد فقط هو اللي "يطبخ فعلاً" ⇒ هذا هو **JavaScript Engine**
- وبرا المطبخ في **طابور مهام** ⇒ هذا هو **Task Queue**
- وفي "مشرف ترتيب" يقول:
  "في أي مهمة جاهزة؟ دخّلها للمطبخ الآن" ⇒ هذا هو **Event Loop**

الـ Event Loop يشتغل كالتالي:

1. لو في مهمة جاهزة ⇒ يعطيها للمحرّك ينفّذها
2. ينتظر لينتهي المحرك
3. يشوف إذا في Microtasks ⇒ ينفّذها
4. يسمح للمتصفح يعمل Rendering
5. يرجع لنقطة البداية

وهكذا... إلى ما لا نهاية.

الجزء الثالث => يعني إيش Macrotask؟

هي مهمّة كبيرة يتحرك لها المتصفح بعد ما يخلص التنفيذ الحالي.

أمثلة:

- setTimeout
- setInterval
- load events
- click, input, mousemove events
- script كامل
- DOMContentLoaded

هذه كلها توضع في **Macrotask Queue**.

المتصفح ينفّذ **Macrotask واحدة فقط** في كل دورة Event Loop.

الجزء الرابع => يعني إيش Microtask؟

هي مهمة أصغر وأسرع وأهم من الماكرو.
أمثلة:

- Promise.then
- Promise.catch
- async/await
- queueMicrotask

القانون الذهبي:

بعد كل Macrotask، المتصفح ينفّذ كل Microtasks الموجودة بالكامل قبل ما يعمل أي شيء ثاني.

وهذا سبب أنّ:

```
console.log("code");
setTimeout(() => console.log("timeout"));
Promise.resolve().then(() => console.log("promise"));
```

الترتيب هو:

1. "code" ⇒ synchronous عادي
2. "promise" ⇒ Microtask
3. "timeout" ⇒ Macrotask

الجزء الخامس => ليش Microtasks أهم من Macrotasks؟

لأن المتصفح **لا يقاطع Microtasks**.

يعني إذا بدأت سلسلة Microtasks، المتصفح:

- ما يعمل Render
- ما يستقبل Events
- ما يشغل أي Macrotask

إلا بعد ما يكمل كل الميكرو تسكات.

هذا يجعلها مثالية للعمليات السريعة اللي لازم تتنفذ فورًا بعد السطر الحالي، بدون انتظار.

الجزء السادس => ليش لازم نفهم فرقهم؟

لأن هذا الفرق يأثر في:

- سرعة الكود
- ترتيب التنفيذ
- الـ UI responsiveness
- فهم async/await
- بناء architecture صح

الجزء السابع => المشكلة الكبرى: المهام الثقيلة

افرضي عندك function تعمل حسابات ضخمة:

```
for (let j = 0; j < 1e9; j++) {}
```

إيش يصير؟

❌ المتصفح يتجمّد
❌ الأحداث ما تشتغل
❌ الواجهة تتوقف
❌ ممكن يظهر error: Page Unresponsive

لأن **جافاسكربت شغالة في خط واحد فقط**.

الحل؟

نقّسم المهمة الثقيلة إلى أجزاء صغيرة.
مثلاً:

- احسب 1 مليون
- وقف
- رجع للمتصفح
- وبعدين كمّل

نستخدم هنا **setTimeout**.

الجزء الثامن => مثال تقسيم المهمة الثقيلة

```
let i = 0;

function count() {
  do {
    i++;
  } while (i % 1e6 !== 0);

  if (i < 1e9) {
    setTimeout(count);
  }
}
count();
```

ليش هذا ممتاز؟
لأن:

- كل جزء صغير جدًا
- المتصفح بين كل جزء وجزء يرجع يستقبل الأحداث
- الواجهة تبقى سلسة

الجزء التاسع => ملاحظة مهمة

لماذا ننقل `setTimeout(count)` إلى بداية الدالة يجعل الأداء أسرع؟

لأن:

- setTimeout دائمًا لديه **حد أدنى للتأخير = 4ms** حسب مواصفات المتصفح
- بالتقديم، نضمن أنّ تأخير 4ms يبدأ مبكرًا
- فتقليل مجموع التأخيرات يجعل التنفيذ أسرع

الجزء العاشر => مثال تقدّم progress bar

لو كتبنا:

```
for (let i = 0; i < 1e6; i++) {
  progress.innerHTML = i;
}
```

المتصفح لن يحدّث الـ DOM إلا بعد ما تنتهي المهمة.

لكن لو قسّمناها:

```
function count() {
  do {
    i++;
    progress.innerHTML = i;
  } while(i % 1000 !== 0);

  if (i < 1e7) setTimeout(count);
}
```

هنا تتحدث الواجهة **بين كل جزء وجزء**.

الجزء الحادي عشر => Microtask vs setTimeout في التصيير Rendering

لما تستخدم:

setTimeout ⇒ المتصفح يقدر يعمل Render بين الأجزاء

queueMicrotask ⇒ لا يسمح بالـ Render إلا بعد انتهاء كل الميكروتاسكات

لذلك:

```
queueMicrotask(f);
```

يجعل التحديثات تظهر "دفعة واحدة" في النهاية.

بينما:

```
setTimeout(f);
```

يجعل التحديثات تظهر تدريجيًا.

الجزء الثاني عشر => Web Workers

لو المهمة ثقيلة جدًا جدًا ومقسّميها ما يفيد، نستخدم **Web Workers**:

- تعمل في Thread منفصل
- لا تملك access للـ DOM
- لكنها ممتازة للحسابات الثقيلة
- وتستغل كل CPU cores

تلخيص نهائي

1) جافاسكربت single-threaded

تعمل مهمة واحدة فقط بنفس الوقت.

2) Event Loop

هو المشرف الذي يقرر أي مهمة تدخل الآن.

3) Macrotasks

أشياء كبيرة مثل load، events، setTimeout…

4) Microtasks

وعود Promise و queueMicrotask، وهي تتنفّذ قبل أي حدث آخر.

5) ترتيب التنفيذ دائمًا:

1. تنفيذ الكود العادي
2. تشغيل كل Microtasks
3. عمل Render
4. تشغيل أول Macrotask
5. الرجوع للخطوة 2

6) تقسيم المهام الثقيلة

نقسمها باستخدام setTimeout حتى ما تتعلق الصفحة.

7) Web Workers

للعمليات الضخمة جدًا.