المثال 1 => الترتيب العادي (Synchronous code)

```
console.log("1");
console.log("2");
console.log("3");
```

✅ **النتيجة المتوقعة:**

1
2
3

مافي أي شيء غير متزامن هون، كله بينفذ سطر بسطر مباشرة.

المثال 2 => setTimeout

```
console.log("Start");
setTimeout(() => {
  console.log("Timeout done!");
}, 0);
console.log("End");
```

✅ **النتيجة المتوقعة:**

Start
End
!Timeout done

مع إنك حطيت `setTimeout(..., 0)` يعني صفر ملي ثانية،
بس الـ **callback** ما بينفذ فوراً — لأنه بيروح على **Callback Queue**
والـ **Event Loop** بينتظره لين يخلص الـ Call Stack (العمليات الحالية).

المثال 3 => Promise مقابل Timeout

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

✅ **النتيجة المتوقعة:**

Start
End
Promise
Timeout

⚡ الـ **Promises** تشتغل ضمن **Microtasks Queue**
وهي تُنفذ قبل الـ **Callback Queue** (اللي فيها الـ setTimeout).
يعني أي `then()` بيتنفذ قبل أي `setTimeout` حتى لو بنفس الوقت.

المثال 4 => Promise داخل Timeout

```
console.log("Start");
setTimeout(() => {
  console.log("Timeout started");
  Promise.resolve().then(() => console.log("Promise inside Timeout"));
  console.log("Timeout ended");
}, 0);
console.log("End");
```

✅ **النتيجة المتوقعة:**

Start
End
Timeout started
Timeout ended
Promise inside Timeout

لاحظ كيف promise داخل الـ timeout بينتظر ينتهي كود الـ timeout نفسه
وبعدين يدخل دور الـ **microtask queue** ويشتغل بعده مباشرة.

المثال 5 => خلط كبير شوي

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
Promise.resolve().then(() => {
  console.log("5");
  setTimeout(() => console.log("6"), 0);
});
console.log("7");
```

✅ **النتيجة:**

1
4
7
3
5
2
6

✅ الكود العادي (synchronous): 1 → 4 → 7
✅ بعدها الـ Promises (microtasks): 3 → 5
✅ وأخيراً الـ Timeouts (macrotasks): 2 → 6