תכנות אסינכרוני

הצורך בתכנות אסינכרוני נובע מכך ש JavaScript היא single thread – כלומר כל הפעולות בסקריפט מתבצעות אחת אחרי השנייה ורק כאשר מסתיימת הפעולה הבאה מבוצעת. הבעייתיות בעובדה זאת היא כאשר מגיעים לקטע קוד בתוכנית שעלול לגרום למעבד להמתין לסיום פעולה שלוקחת המון זמן ותוקעת את משך הסקריפט, כי כאמור עד שלא נגמרת פעולה מסוימת, המעבד לא מריץ את הפעולה הבאה. פעולות שביצועם דורש זמן "רב" במושגים של מהירות של מחשב: תקשורת עם api ברשת, ביצוע פעולות במסד הנתונים, קריאת נתונים מהדיסק הקשיח ועוד...

בתכנות סינכרוני: הפעולות מבוצעות זו אחר זו, כאשר פעולה מסתיימת המעבד עובר לפעולה הבאה.

בתכנות אסינכרוני: מאפשר למספר דברים לקרות במקביל. כאשר נתחיל בפעולה שדורשת זמן רב (למשל ביצוע בקשת GET ל api ברשת) התוכנית תמשיך להתקדם וכאשר תסתיים הפעולה התוכנית מבצעת קטע קוד שהוגדר מראש שיקרא כאשר תסתיים הפעולה.

דוגמה לפעולה אסינכרונית: **setTimeout** .

Callback

אחת הדרכים להשתמש בפעולות אסינכרוניות היא callback. Callback היא פונקציה אשר מועברת כפרמטר לפעולה אסינכרונית, כך שהפונקציה תתבצע כאשר תסתיים הפעולה האסינכרונית. דוגמה:

```
setTimeout(()=>console.log(2),2);
```

הפרמטר הראשון הוא פונקציית ה callback, הפרמטר השני הוא כמה שניות נרצה להמתין עד שתתבצע הקריאה לפונקציית ה callback שלנו. בדוגמה פה לאחר 2 שניות תודפס הספרה 2.

דוגמה יותר מורכבת:

```
const request = require('request');

const url = 'https://api.agify.io/?name=bella';
request.get(url,(err,res)=>{

    console.log(JSON.parse(res.body));
})
```

בדוגמה הנ"ל אני משתמש ב request ניתן לקרוא עליו כאן: https://www.npmjs.com/package/request בגדול הוא משמש על מנת לבצע קריאות http.

request.get היא פעולה אסינכרונית אשר מבצעת בקשת http מסוג GET לפרמטר הראשון. הפרמטר השני הוא בעצם פונקציית ה callback שתקרא על ידי request.get רק כאשר התקבלה כל התגובה עבור אותה קריאת http.

לסיכום: הרעיון ב callback הוא להעביר פונקציה שתתבצע רק כאשר תסתיים הפעולה האסינכרונית ודבר זה מבטיח לנו שהקוד שנמצא בתוך ה callback יתבצע כאשר הפונקציה האסינכרונית תסתיים.

Event loop

על מנת לעבור לנושא הבא של ניהול שגיאות עם callback, יש להבין היטב את הנושא Event loop . Event loop מאפשר ל JavaScript להריץ פעולות אסינכרוניות כלומר מבלי לחכות לסיום כל פעולה על מנת להמשיך לפעולה הבאה. ראשית נגדיר את מושגי הבסיס:

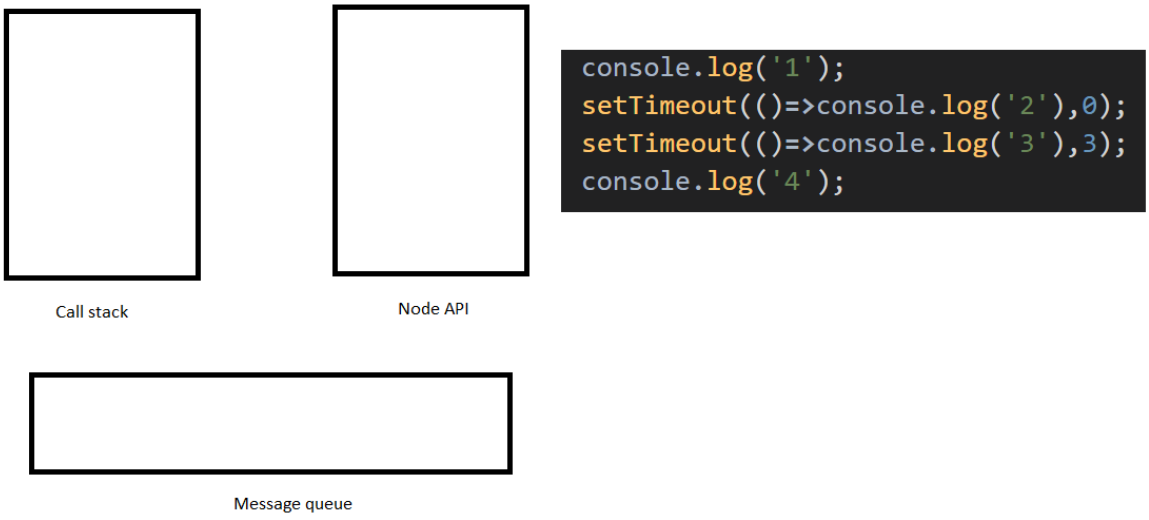
- **Call stack**- מחסנית הקריאות שמייצגת את סדר הקריאות של הפונקציות. הפונקציה שנוספה ראשונה תסתיים בסוף.
- **Node api**- הוא לא חלק מה core של JavaScript, מספק פונקציות שונות אשר ניתן להשתמש בהם עם JavaScript כמו setTimeout .

- **Message queue**- רשימה של פונקציות שמחכות להתבצע, יתבצעו רק כאשר ה call stack ריק. פונקציות נוספות לתוך ה Message queue מ הNode Api.
- **Event loop**- הוא תהליך שבודק האם ה call stack ריק או לא. הוא מתווך בין ה Message queue ל call stack: אם ה call stack ריק הוא דוחף אליו פונקציה מתוך ה message queue.

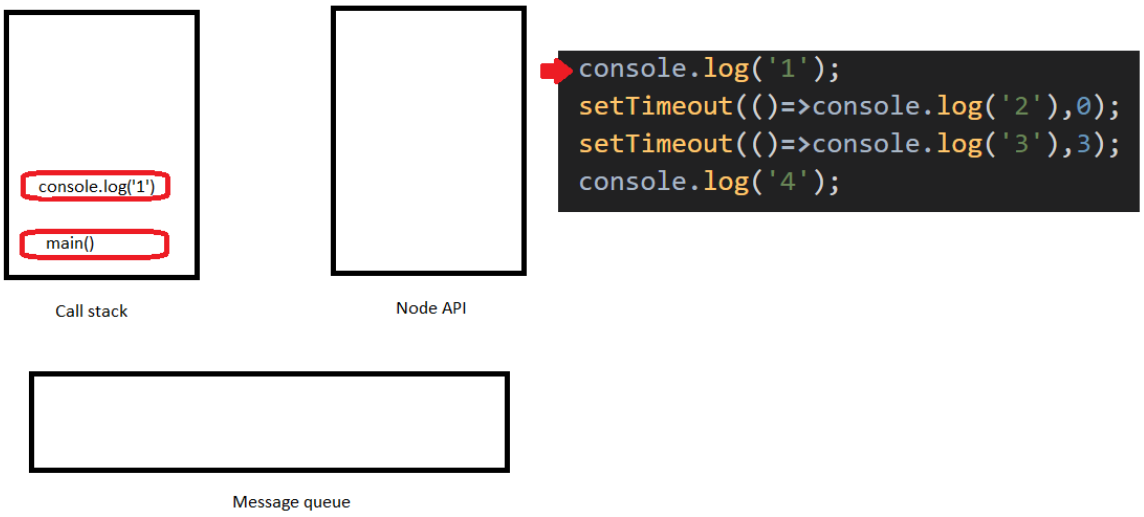
כעת אציג דוגמה שתמחיש איך הדברים פועלים מאחורי הקלעים:

```
console.log('1');
setTimeout(()=>console.log('2'),0);
setTimeout(()=>console.log('3'),3);
console.log('4');
```

תחילה כל התורים ריקים:

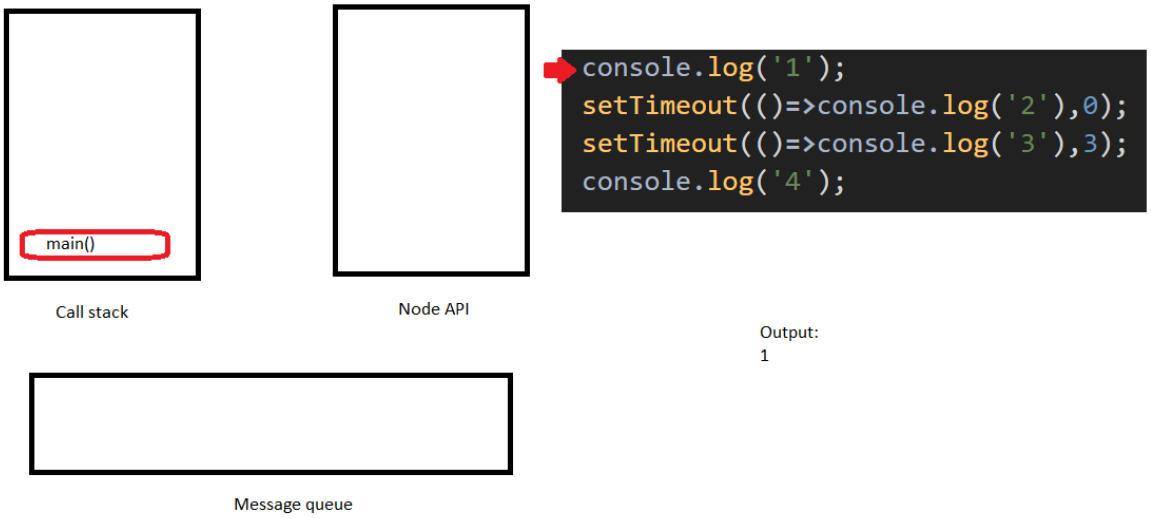


תחילה נכנסת הפוקנציה main, ולאחריה יכנסו הפונקציות לפי הסדר שנמצאות ב main:

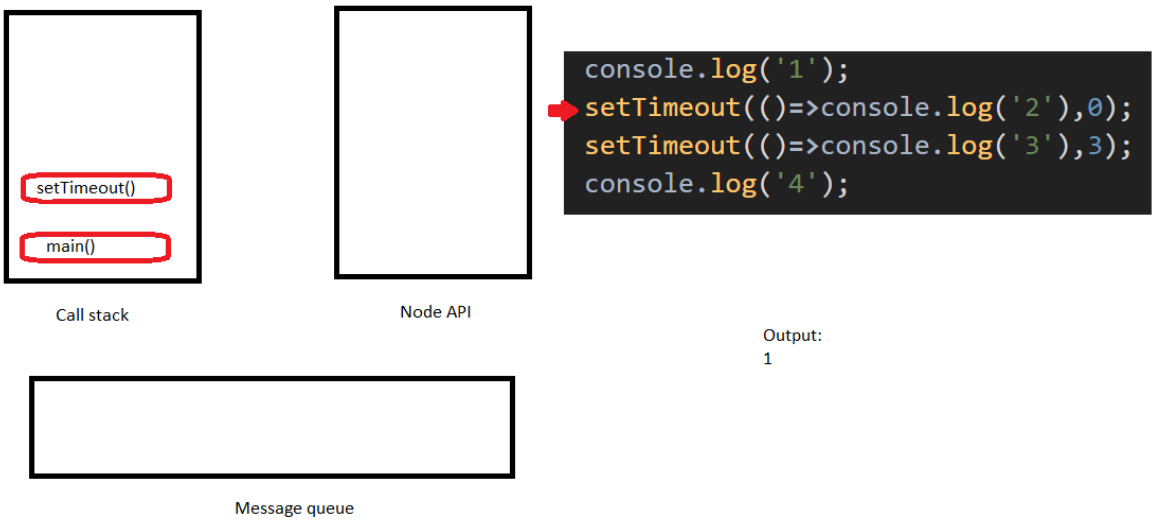


כעת, יבדוק ה event loop האם ה call stack ריק. כמובן שה call stack לא ריק ולכן לא יתאפשר ל event loop להכניס פונקציות מה Message queueלתוך ה call stack. (כרגע אין צורך גם להכניס כי ה message queue ריק).

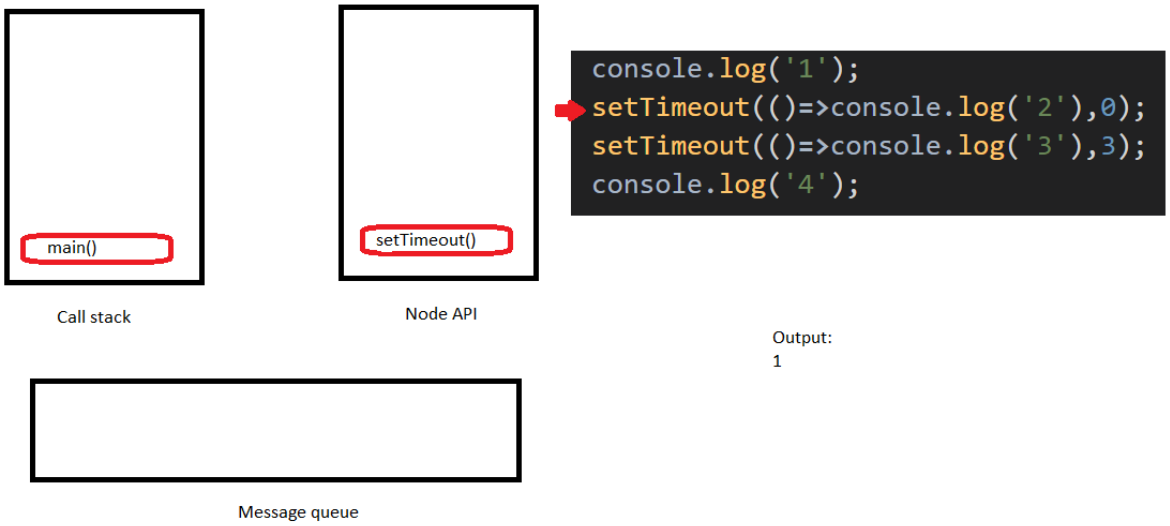
לאחר מכן בגלל ש console.log היא חלק מה JavaScript engine הפעולה מתבצעת מיד ונקבל על המסך את הפלט והפעולה תמחק מ ה callStack כי היא כבר הסתיימה:



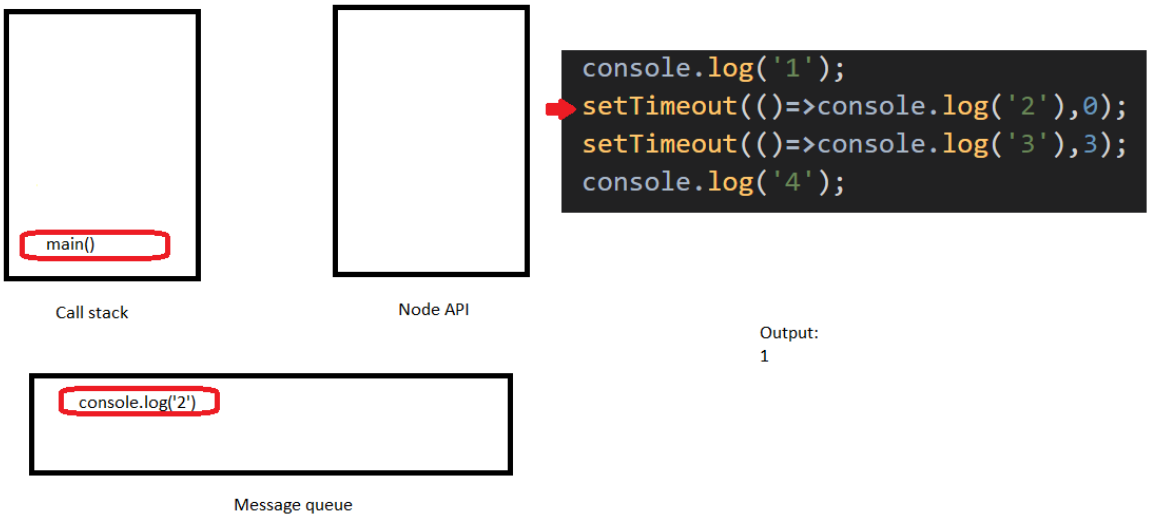
לאחר מכן, תכנס הפעולה הבאה ל call stack : setTimeout:



בגלל ש setTimeout היא פונקציה, היא תתבצע. אך בשונה מ console.log הפונקציה הזאת היא לא חלק מ JavaScript, לא קיים לה מימוש ב JavaScript אלה ב NodeJS. לכן היא מועברת ל NodeAPI. **ה NodeAPI ממש את setTimeout בכך שהוא מקצה 0 שניות ולאחר מכן פונקציית ה callback מועברת ל Message queue ולא ל callback לביצוע מיידי.** כלומר, כל מה שמגיע מ NodeAPI עובר תהליך של מימוש שהגדירו מפתחי ה Node לאותה הפונקציה והתוצאה מועברת ל Message queue. כלומר:

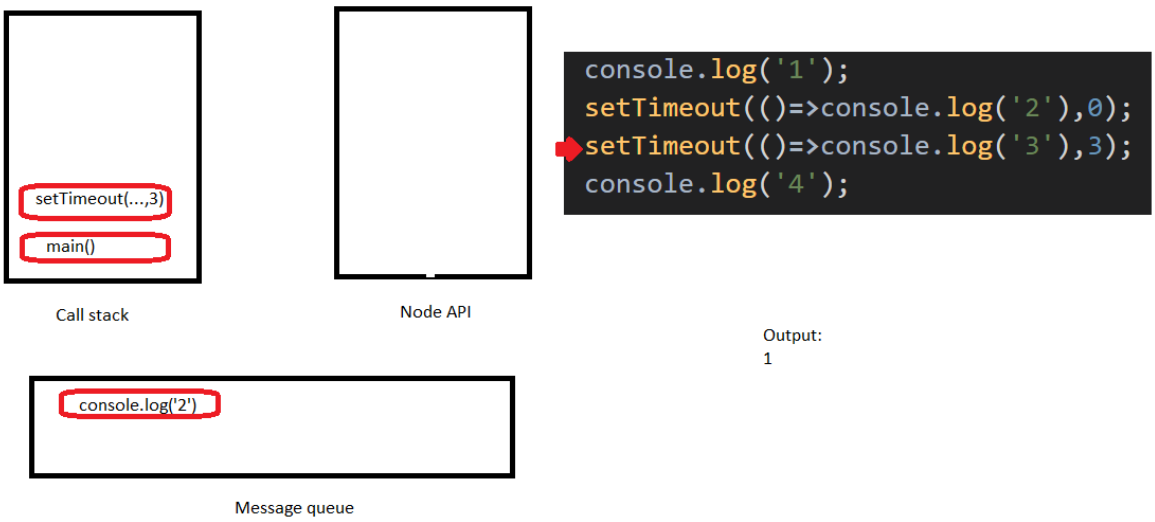


ולאחר מכן אחרי טיימר של 0 שניות נקבל את התוצאה הבאה:

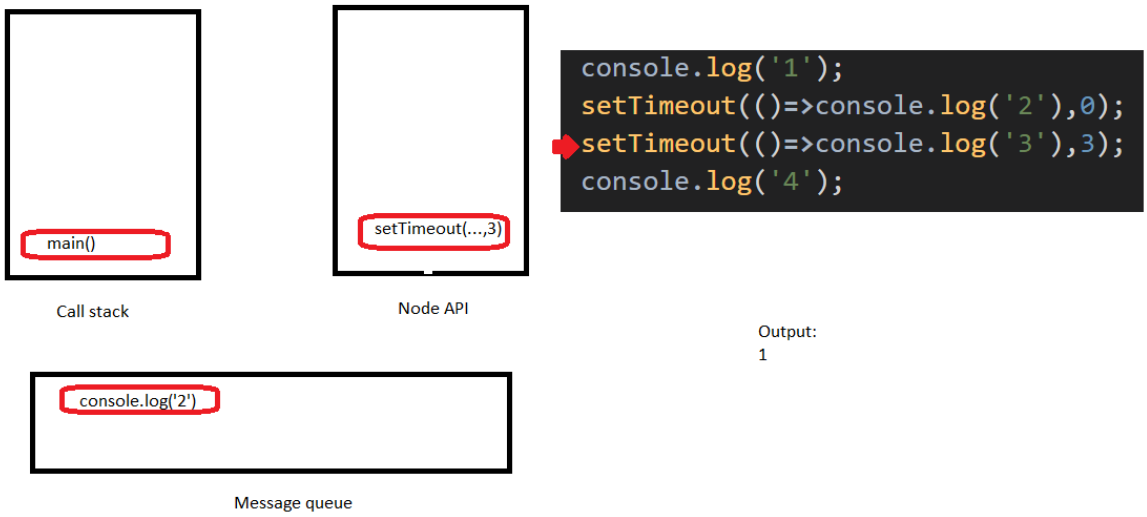


יש לשים לב שהדפסת '2' עדין לא מתבצעת כל עוד היא לא בראש ה call stack. כעת ה Event loop בודק האם ה call stack ריק על מנת שהוא יוכל להעביר לו פונקציות מתוך ה Message queue ל call stack, אך הוא מגלה שלא ולכן console.log('2') נשאר בתוך ה **Message queue** ולא מתבצע כעת.

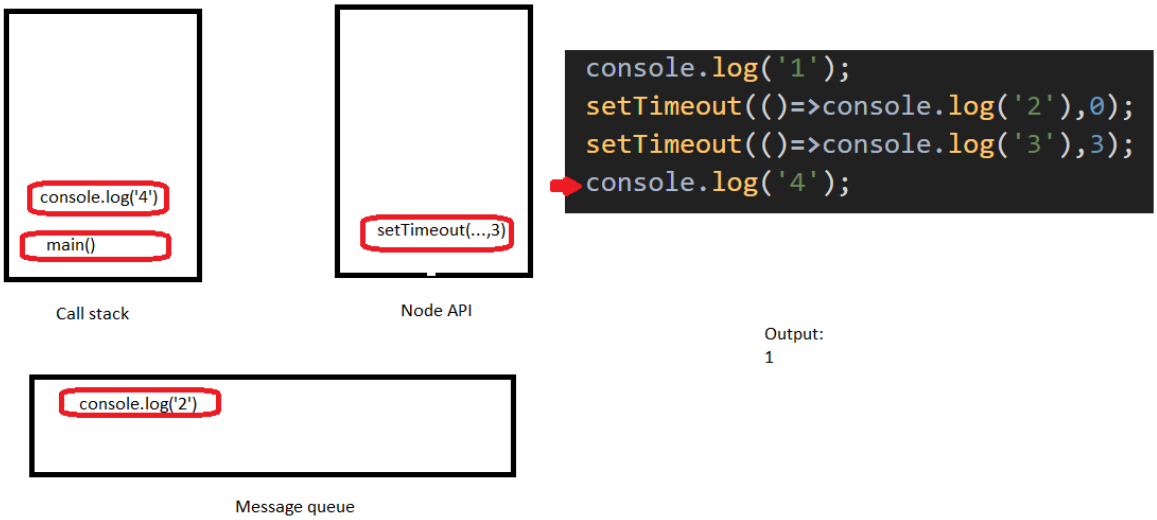
כעת, נעבור לשורה הבאה שגם היא זימון של setTimeout. הפעם יופעל טיימר של 3 שניות מהזמן שהפונקציה נמצאת ב Node API ורק בסוף הטיימר הפונקציה תועבר ל Message queue:



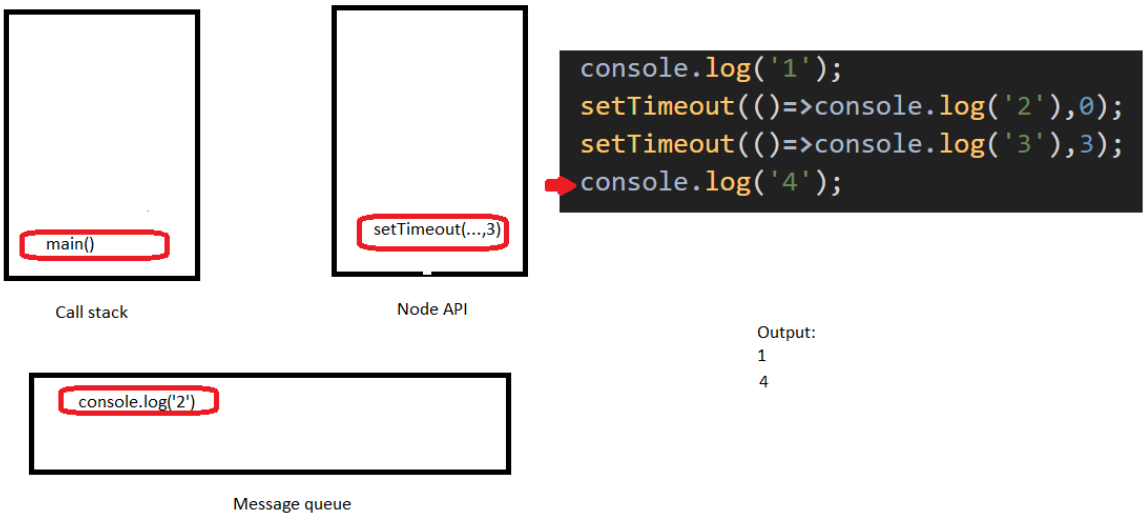
ומיד הפונקציה מועברת ל NodeAPI בו היא תמתין 3 שניות: כעת ה Event loop עדין רואה שה call stack לא ריקה, ולכן לא יכול לדחוף לה פונקציות.



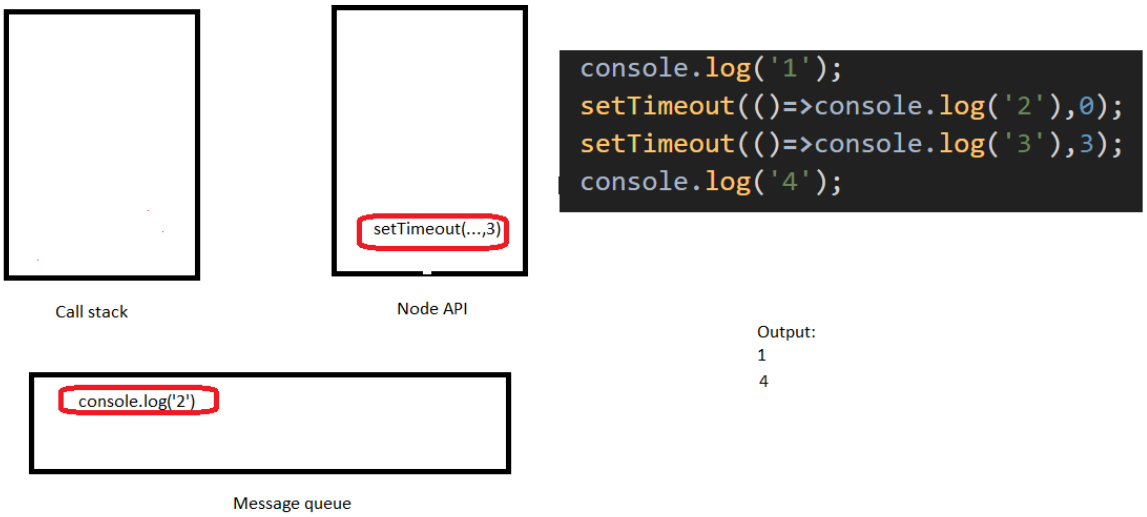
בזכות מנגנון זה אין צורך לחכות לסיום הsetTimeout אלה אפשר לעבור לפעולה הבאה!



ואז יתבצע ה console.log, ויודפס 4 על המסך:



לאחר מכן, בודק ה Event loop האם ה call stack ריקה, הוא רואה שלא. כעת הסתיימה פונקציית ה main והיא יוצאת מה call stack.



לאחר מכן, ה Event loop רואה שה call stack ריקה, וכעת כן יכול לדחוף את console.log('2') ל call stack שם היא מתבצעת ומודפס 2 למסך. כעת נגמר גם הטיימר של ה 3 שניות, ומועבר console.log('3') ל Message queue ובזכות זה שה call stack ריקה, מודפס 3 למסך.

לסיכום: בזכות מנגנון זה JavaScript מריצה פונקציות אסינכרוניות ובכך לא תוקעת את התוכנית למרות שהיא single thread!

יצירת פונקציה אסינכרונית

נניח שיש לנו פונקציה שאנו מעוניינים שתהיה אסינכרונית משום שהיא לוקחת המון זמן ושמה foo. איך נהפוך אותה לאסינכרונית ? פשוט מאוד נעטוף אותה ב setTimeout עם 0 , ובכך היא תעבור ל Node API ומשם ל Message queue ותתבצע בלי לתקוע את משך התוכנית עד אשר היא תסתיים.

```
const foo = ()=>{
    //...
}

const asyncFoo = ()=>{
    setTimeout(foo,0);
}
```

ניהול שגיאות עם callback:

כעת אנחנו יכולים לדבר על ניהול שגיאות עם callback.

נתון הקוד הנ"ל אשר מחשב סכום של שתי מספרים בצורה אסינכרונית אך במידה ואחד המספרים שלילי הוא זורק שגיאה:

```
const sum = (a,b)=>{
    setTimeout(()=>{
        if(a>=0 && b>=0)
            console.log(a+b);
        else
            throw new Error('arguments must be positive');
    },1)
}

try{
    sum(1,2);
}
catch(e){
    console.log(e.message);
}
```

באופן מפתיע למרות שזימון הפונקציה sum עטוף ב try ו catch, השגיאה לא נתפסת ב catch! הסיבה לכך היא שכאשר נזרקת השגיאה בהכרח ה call stack ריקה (כי השגיאה הגיע מה message queue), ולכן בהכרח הסתיים הקוד שאמור לתפוס את השגיאה ולכן השגיאה לא נתפסת. לכן הדרך הזו היא לא טובה עבור ניהול שגיאות עם callback. הדרך הנכונה היא כזאת:

```
const sum = (a,b,callback)=>{
    setTimeout(()=>{
        if(a>=0 && b>=0)
```

```
            callback(undefined,a+b);
        else
            callback(new Error('arguments must be positive')
);
    },1)
}


sum(1,2,(error,result)=>{
    if(error)
        console.log(error.message);
    else
        console.log(result);
});
```

הוספת פרמטר שהוא פונקציית ה callback שתקרא כאשר נגמר ה setTimeout. ה callbackמקבל שתי ארגומנטים: error, result. כל פעם רק אחד מהם יהיה עם ערך, השני יהיה undefined. אם יש שגיאה נכניס ל error את השגיאה ולפרמטר השני undefined, במידה ואין שגיאה, נשים ב error את הערך undefined וב result את תוצאת החיבור. על ה callback לבדוק האם error מכיל ערך או שהוא מכיל תוצאה.

לסיכום: על ה callback לקבל 2 פרמטרים: error,result ועל הפונקציה האסינכרונית לקרוא ל callback כאשר אחד המשתנים הוא בהכרח undefined והשני בהכרח מכיל ערך.

סנכרון בין שתי callback:

נניח שנתונות לנו 2 פונקציות שהן **אסינכרוניות** ותלויות אחת בשניה. למשל האחת מגרילה מספר ואם הוא בין 10 ל 100 אזי היא מזמנת פונקציה אחרת שמעלה אותו בריבוע, אחרת מודפסת שגיאה שהמספר לא בין 10 ל 100. איך נבצע את התלות הזאת בין הפונקציות?

```
const randNumber = (callback)=>{
    const num = Math.floor(Math.random() * 100);
    setTimeout(()=> num > 10 && num < 100 ? callback(undefin
ed,num)
    : callback('number must be (10,100)'),0);
}
const square = (error,num)=> setTimeout(()=> error ? console
.log(error) : console.log(num),0);


randNumber(square);
```

Promises

מאפשרים לנהל פעולות אסינכרוניות בצורה קלה יותר מאשר callback. למעשה Promise בנוי על ידי callback. Promise הוא אובייקט. על מנת לאפשר תמיכה של הפונקציה האסינכרונית שלנו ב Promise נדאג שהיא תחזיר Promise בתחביר הבא:

```
const setTimeoutPromise = amount =>{
  return new Promise((resolve,reject)=>{
    // executor: here write your async function
    setTimeout(x=>resolve('Hi'),amount);
  })
}
```

נתחיל לחקור את האובייקט Promise. בבנאי שלו אנו רואים שהוא מקבל פונקציה ששמה מכונה executor. ברגע שיוצרים Promise חדש, **הבנאי של Promise קורא אוטומטי ל executor**. תפקיד ה executor הוא להיות הקוד האסינכרוני שלנו, כלומר כל הלוגיקה של הפונקציה תהיה בתוך ה executor. בסופו של דבר ה executor יוצר **result** שהוא התוצר הסופי של הפעולה האסינכרונית. הפונקציות resolve ו reject מסופקים על ידי JavaScript וכל הקוד שלנו יהיה בתוך ה executor. Resolveו reject הם בעצם callback.

- Resolve- זו בעצם פונקציית ה callback שנרצה לזמן במידה ואין שגיאות.
- Reject- זו בעצם פונקציית callback אותה נרצה לזמן כאשר יש שגיאות.

כשנזמן את resolve נזמן אותה עם הערך שהפיק ה executor כלומר:

```
    setTimeout(x=>resolve('Hi'),amount);
```

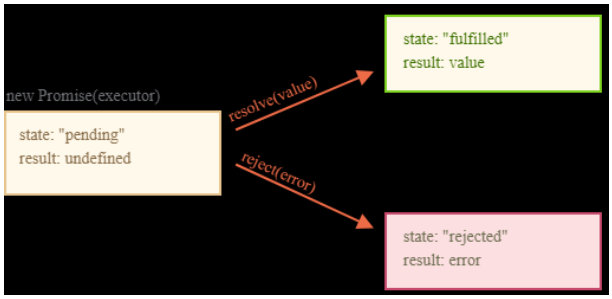
ואם יש שגיאה נזמן את reject עם ה error : (למשל זו אותה הפונקציה רק עם בדיקה ש amount > 20):

```
const setTimeoutPromise = amount =>{
  return new Promise((resolve,reject)=>{
    // executor: here write your async function
    amount > 20 ?
      setTimeout(x=>resolve('Hi'),amount)
      :reject('amount must be greater then 20');
  })
}
```

אובייקט מסוג Promise מכיל את המאפיינים הבאים:

- State- מאותחל כ Pending לאחר מכן אם הצליח משתנה ל Fulfilled (על ידי קריאה ל resolve) ואם לא הצליח משתנה ל Rejected על ידי קריאה ל reject.
- Result- הערך שהופק על ידי ה executor במידה ואין שגיאה (ומועבר על ידי פרמטר ל resolve), ואם יש שגיאה זה בעצם השגיאה, ומועבר על ידי פרמטר ל reject.

דוגמה שממחישה זאת:



ההבדל בין ה callback pattern ל promise הוא שפה אנחנו מפרידים בין הפונקציה אשר תתבצע כאשר אין שגיאות לבין הפונקציה שתתבצע כשיש שגיאות בניגוד ל callback pattern שם מועברת פונקציה אחת שאחראית גם על טיפול בשגיאות וגם על ביצוע הלוגיקה של הפונקציה.

לסיכום:

1. הבנאי של Promise קורא אוטומטי ל executor.
2. Resolve,reject אוטומטי מוגדרים על ידי המנוע של JavaScript וכל שנותר לנו זה לזמן את אחד מהפונקציות הללו עם הערך שייצר ה executor \ עם error .

כלל חשוב מאוד שאתם יוצרים Promise משלכם: ה executor צריך לקרוא או ל reject או ל resolve כתוצאה מכך ה state של ה promise משתנה והוא סופי! לא ניתן לזמן את resolve ואז את reject. לדוגמה הקוד הבא

```
const setTimeoutPromise = amount =>{
  return new Promise((resolve,reject)=>{
    // executor: here write your async function
    amount > 20 ?
      setTimeout(x=>resolve('Hi'),1)
      :setTimeout(x=>reject('amount must be greater then 20'
),1);
      setTimeout(x=>reject('another error'),3);
  })
}
```

עבור amount = 21 יפעיל אחרי שניה את resolve ויתעלם מהפעולה הבאה שהיא להפעיל את reject. כלומר ברגע שהstate ישתנה הוא סופי וכל הפעלה של שינוי state לאחר מכן לא תקרא. לכן יש להפעיל פעם אחת או את resolve או את reject.

בנוסף כלל נוסף: resolve, reject מקבלים פרמטר אחד או 0 פרמטרים, reject , resolve יתעלמו מפרמטרים נוספים.

בנוסף ניתן לקרוא אוטומטית ל reject/resolve ללא המתנה של זמן כלומר באופן סינכרוני. דבר זה טוב למשל כאשר מגלים שגיאה שאין טעם להפעיל את התהליך האסינכרוני כי הקלט לא תקין. כמו בדוגמה:

```
const setTimeoutPromise = amount =>{
  return new Promise((resolve,reject)=>{
    // executor: here write your async function
    amount > 20 ?
      setTimeout(x=>resolve('Hi'),amount)
      :reject('amount must be greater then 20');
  })
```

```
}
```

Then

משמש כמתווך בין ה executor לבין הcallback שנרצה שתקרא כאשר נגמר התהליך האסינכרוני. Then היא פונקציה שנמצאת באובייקט מסוג Promise, והיא מקבלת 2 פרמטרים:

- פרמטר ראשון: פונקצית callback שמקבלת 0 או 1 ארגומנטים, היא תופעל כאשר יסתיים התהליך האסינכרוני בהצלחה (fulfilled). הפרמטר שהיא מקבלת הוא בעצם הפרמטר ששלחנו לפונקציית ה resolve ב executor. שהוא בעצם ה result של ה Promise.
- פרמטר שני- אופציונאלי: פונקציית callback שמקבלת 0 או 1 פרמטרים, שהיא תופעל כאשר התהליך יסתיים עם שגיאה כלומר reject הופעלה. הפרמטר של הפונקציה יהיה אותו ה error ששלחנו ל reject.

בדרך כלל נזמן את then עם הפרמטר הראשון ועבור טיפול בשגיאות נשתמש ב catch. דוגמה:

```
const setTimeoutPromise = amount =>{
  return new Promise((resolve,reject)=>{
    // executor: here write your async function
    amount > 20 ?
      setTimeout(x=>resolve('Hi'),1)
      :setTimeout(x=>reject('amount must be greater then 20'
),1);
  })
}

setTimeoutPromise(21).then(x=>console.log(x),x=>console.log(
x));
```

אותה הפונקציה שהראנו ממקודם, עכשיו בזכות then ניתן לזמן את setTimeoutPromise . במידה ונזמן את הפונקציה עם amount < 20 תופעל ה callback שבפרמטר השני, אחרת הפרמטר הראשון.

Catch

שימוש ב Catch שקול לשימוש ב Then עם 2 פרמטרים, הייתרון ב Catch שהקוד יותר קריא. דוגמה לזימון עם Catch:

```
setTimeoutPromise(21).then(x=>console.log(x)).catch(x=>conso
le.log('Error detected'))
```

התהליך הסתיים בהצלחה? תופעל הפונקציה שהועברה ל Then, אחרת תופעל הפונקציה שהועברה ל Catch.

Finally

זהה ל Finally של Try & Catch. תקרא תמיד בין אם יש שגיאה ובין אם לא, שקולה ל then(f,f) כך שלא משנה האם התהליך הצליח f תקרא.

יש לשים לב:

- Finally לא מקבלת פרמטרים
- Finally מעבירה את result ל handler הבא כלומר:

```
setTimeoutPromise(21).finally(console.log('pass Hi')).then(x
=>console.log(x));
```

אמנם Finally לא מקבלת כפרמטר את Hi, אך היא מעבירה אותו ל Then שמתבצע אחריה.

Promise Chaining

הייתרון הגדול ב Promise לעומת Callback הוא בסיטואציה הבאה:

דמיינו שיש לנו 4 תהליכים אסינכרונים שתלויים אחד בשני כלומר צריך שיתבצעו בסדר הבא:

Task 1 -> Task 2 -> Task 3 -> Task 4 ברגע שנגמר תהליך רק אז ניתן יהיה להפעיל את התהליך הבא. דוגמה לקוד שממחיש זאת:

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

לינק למקור: http://callbackhell.com/

ברגע שקריאת כל קבצי התקייה מסתיימת (יש פה קריאה מהדיסק שהיא פעולה איטית ולכן הפונקציה readdir היא אסינכרונית), מופעלת פונקציית ה callback -> לאחר מכן עוברים כל הקבצים בעזרת forEach שגם לה מעבירים callback -> לאחר מכן מפעילים על כל קובץ עוד פונקציה אסינכרונית שמקבלת callback שתופעל רק ש size תסתיים. בקיצור התוצאה היא שהקוד לא קריא, יש היררכיה בסוף של {{ מה שמקשה על קריאות הקוד והבנתו. עם Promise זה היה נראה כך:

```
fs.readdir('C:\Users')
  .then(x=> ...) Task 1
    .then(x=> ...) Task 2
      .then(x=> ...) Task 3
      .catch(x=> ...) Task 3
    .catch(x=> ...) Task 2
  .catch(x=> ...) Task 1
```

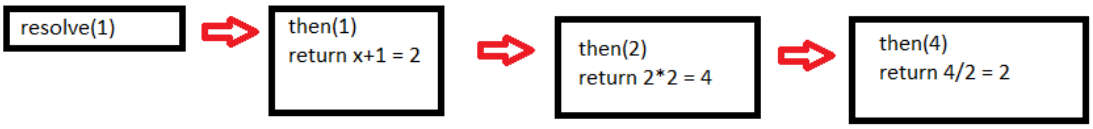
הרבה יותר קריא!

על מנת להעביר פרמטרים בין handlers פשוט נרשום return x כך ש x הוא הפרמטר אותו נרצה להעביר. דוגמה:

```
new Promise((resolve,reject)=> setTimeout(resolve(1), 10))
```

```
.then(x=>x+1)
.then(x=>x*x)
.then(x=>x/2);
```

התהליך מתבצע כך:



דבר זה מתאפשר בזכות זה ש then/finally/catch מחזירים אובייקט מסוג Promise , מה שמאפשר לשרשר בניהם.
דבר זה דומה להחזרת reference ב C++.

ניהול שגיאות עם Promise

על מנת לנהל שגיאות עם Promise נוכל להיעזר ב Catch:

```
const fs = require('fs').promises

fs.readdir('blabla').catch(x=>console.log(`Error ${x}`));
```

ובגלל שלא קיים תקייה בשם blabla תופעל ה callbackשקיבל catch.

ובמדיה ויש לנו שרשרת של Then, נוכל להגדיר Catch כללי שיתפוס כל שגיאה שתתרחש במהלך השרשרת:

```
fs.readdir('C:\Users')
  .then(x=> ...)
    .then(x=> ...)
      .then(x=> ...)
  .catch(x=> ...) // if any of the promises above rejects
```

- ברגע ששגיאה נזרקת ב executor אוטומטית מופעל reject(error)
- ברגע ששגיאה נזרקת ב then אוטומטי היא נתפסת על ידי ה catch הקרוב.

כאשר מופעלת reject אך אין catch שמנהל את השגיאה, התוכנית נעצרת ונזרקת שגיאה ומודפסת ב console ממש כמו שגיאה רגילה שנזרקת ולא מטופלת.

Promise API

Promise.all: נניח ואנחנו רוצים לבצע כמה פעולות אסינכרוניות במקביל וש resolve תופעל כאשר כולם הסתיימו. בשביל מצב כזה קיימת הפונקציה all. הפונקציה מקבלת מערך של אובייקטים מסוג Promise ומחזירה Promise חדש שמאחד את כולם, כלומר resolve מופעלת כאשר כל ה Promises במערך הסתיימו וכתוצאה מכך result הופכת להיות מערך של כל תוצאות כל ה Promises. לדוגמה:

```
Promise.all([
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Hi ')
,1)),
```

```
  new Promise((resolve,reject)=>setTimeout(x=>resolve('my ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('name
'),1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('is ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Ofek'
),1))
]).then(x=>x.forEach(word=>process.stdout.write(word)));
```

למשל ה Promise שמוחזר מ all מרכיב משפט: Hi my name is Ofek , וללא all שמאחד את כולם ל Promise אחד לא נוכל להרכיב בוודאות את המשפט עם ביצוע מקבילי של כל הפעולות האסינכרוניות כי לא נדע מתי הסתיימו כל ה Promises.

שים לב: אף על פי שבמצבים מסוימים פעולות לקוחות יותר זמן מאחרות, הסדר של ה Promises ב executor יהיה זהה לסדר של הערכים ב results.

```
Promise.all([
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Hi ')
,4000)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('my ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('name
'),1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('is ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Ofek'
),1))
]).then(x=>x.forEach(word=>process.stdout.write(word)));
```

בדוגמה הנ"ל עדין נשמר סדר המשפט אף על פי זה שה Promise הראשון לוקח הרבה יותר זמן מהאחרים.

ברגע שאחד מן ה Promises נדחה, כלומר מופעל אצלו פונקציית ה reject, אוטומטית יוחזר ע"י all אובייקט Promise עם state של rejected ו ה result יהיה השגיאה.

```
Promise.all([
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Hi ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>reject('my '),
1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('name
'),1)),
```

```
  new Promise((resolve,reject)=>setTimeout(x=>resolve('is ')
,1)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Ofek'
),1))
]).then(x=>x.forEach(word=>process.stdout.write(word))).catc
h(x=>console.log('Error'))
```

בדוגמה זו עבור המילה my מופעלת reject וכתוצאה מכך מוחזר Promise שה stateשלו הוא rejected ולכן מופעל ה catch.

Promises.allSettled

בדומה ל all פונקציה זו מקבלת מערך של Promises ומחזירה Promise חדש. בשונה מ all, המערך results הוא בהכרח מערך של אובייקטים מסוג:

{status: string, value: any} כלומר עבור כל Promise שהופעל, results יכלול את המידע הבא עליו:

status- rejected, fulfilled

value- הערך שזומנה איתו resolve או השגיאה שזומנה איתה reject.

דוגמה לפלט:

```
[
  { status: 'fulfilled', value: 'Hi ' },
  { status: 'fulfilled', value: 'my ' },
  { status: 'rejected', reason: 'name ' },
  { status: 'fulfilled', value: 'is ' },
  { status: 'fulfilled', value: 'Ofek' }
]
```

ניתן לסכם זאת כך:

- all – או כולם או כלום.
- allSettled – האחר לא משפיע על השאר.

Promise.race

אותו דבר כמו all, רק מחזיר אובייקט של Promise שמכיל את ה result וה state של הPromise שהסתיים ראשון.

```
Promise.race([
  new Promise((resolve,reject)=>setTimeout(x=>resolve('Hi ')
,1000)),
  new Promise((resolve,reject)=>setTimeout(x=>resolve('my ')
,30)),
]).then(x=>console.log(x)).catch(x=>console.log('Error'))
```

בדוגמה הזו נקבל כפלט את 'my'.

Promise.resolve/reject

פונקציות אלו מקבלות ארגומנט אחד. הם מחזירות אובייקט של Promise כך ש :

Result = ארגומנט אותו Promise.resolve\reject קיבלו

State = בהתאם להפעלת resolve\reject .

דוגמה:

```
const fulfilledPromise = Promise.resolve('Done');
fulfilledPromise.then(x=>console.log(x));
```

הפלט: Done משום שה state של ה Promise נקבע להיות fulfilled כי זימנו את resolve. בהתאמה כאשר נזמן את Promise.reject נקבל אובייקט חדש של Promise אשר הstate שלו rejected.

Async\Await

החל מ ES7 ניתן להשתמש ב Async/await על מנת להקל בתחביר השימוש ב Promise.

Async

המילה השמורה Async מצטרפת להגדרת פונקציה. המשמעות עבור הוספת מילה זו היא שהפונקציה תמיד מחזירה Promise. כלומר הערך היוחזר מהפונקציה הוא אובייקט של Promise. דוגמה:

```
const myFucnAsync = async () =>{
    return 'Hi';
}

const myFunc = () =>{
    return Promise.resolve('Hi');
}
```

ההגדרה של myFuncAsync מתרגמת ל myFunc. זה המשמעות של async. תמיד החזר Promise על ידי לקיחת הערך המוחזר, והפעלת Promise.resolve(return value) .

Await

מילה שמורה נוספת שחייבת להיות בתוך פונקציה שמוגדרת כ async. דוגמה:

```
const myFucnAsync = async () =>{
    const packets = await fetch('https://www.google.com/');
    console.log(packets);
}
```

כל ניסיון לרשום await בפונקציה שהיא לא async יגרור שגיאה:

```
SyntaxError: await is only valid in async function
```

Await מופעל על Promise, בזכות זה ש fetch מחזיר Promise ניתן להפעיל עליו await. כלומר התבנית היא:

```
const myFucnAsync = async () =>{
    const value = await Promise;
}
```

המשמעות של Await היא המתן עד אשר יסתיים ה Promise (עד אשר resolve יופעל \ עד אשר reject יופעל \ עד אשר ישתנה ה state שלו) והחזר את result של אותו ה Promise.

```
const myFucnAsync = async () =>{
    const value = await new Promise((resolve,reject)=> setTi
meout(()=>resolve('Hi'),2000));
    return value;
}

myFucnAsync().then((x)=>console.log(x));
```

תחילה משום שהשתמשנו ב await, value ימתין 2 שניות (עד אשר יסתיים ה Promise) ורק אחרי 2 שניות value יכיל את הערך Hi ונמשיך לשורה הבאה. בשורה הבאה מוחזר value כלומר Hi. משום ש async מחזירה Promise, נשתמש ב then על מנת לחכות לסיום ה Promise שמוחזר מ myFuncAsync ולבסוף נוכל להדפיס את Hi.

לסיכום:

- await יכול להיות בשימוש רק בתוך פונקציות async
- פונקציות async מחזירות Promise
- Await גורם ל JavaScript להמתין עד אשר יסתיים הפעולה האסינכרונית ורק לאחר מכן להמשיך הלאה לשורה הבאה, אך במקביל לא מבזבז את ה cpu משום שכמו שלמדנו בפעולה אסינכרונית ב JavaScript התוכנית ממשיכה לרוץ ולא להיתקע (להסבר נוסף יש לחזור על תחילת המאמר).

Await הוא תחביר הרבה יותר מובן ופשוט ששקול ל then.

Promise chaining with await

נחזור לדוגמה שהצגתי בפרק של Promise chaining :

```
fs.readdir('C:\Users')
  .then(x=> ...) Task 1
    .then(x=> ...) Task 2
      .then(x=> ...) Task 3
      .catch(x=> ...) Task 3
    .catch(x=> ...) Task 2
  .catch(x=> ...) Task 1
```

איך היינו פותרים אותה בעזרת async\await?

```
const myFucnAsync = async () =>{
    try{
        const files = await fs.readdir('C:\Users')
        const afterTask1 = await task1(files);
        const afterTask2 = await task2(afterTask1);
        const afterTask3 = await task2(afterTask2);
    }
    catch(e){
        console.log(e);
```

```
    }
}
```

הרבה יותר פשוט, והרבה יותר קריא!

בתחילת הדרך מתכנתים נוטים לשכוח ש await לא עובד מחוץ לפונקציה שהיא לא async למשל הם נוטים לרשום ב top level code :

```
const packets = await fetch('https://www.google.com/');
    console.log(packets);
```

אך כמובן שיקבלו שגיאה משום שהקוד משתמש ב await לא בתוך async function. פתרון:

```
(async ()=>{
    const packets = await fetch('https://www.google.com/');
    console.log(packets);
})();
```

Thenables

אילו אובייקטים אשר מכילים את הפונקציה then (כמו ב Promise) כך ש then מקבלת 2 פרמטרים: resolve,reject. (2 פונקציות). בתוך הגוף של then יהיה בעצם הקוד של ה executor (כלומר הקוד האסינכרוני) שבסופו תקרא או resolve או reject:

```
class MyThenable{
    constructor(a,b){
        this.a = a;
        this.b = b;
    }

    then(resolve,reject){
        (this.a >= 0 && this.b>= 0)?
        setTimeout(()=>resolve(this.a+this.b),1500):
        reject('arguments must contain positive integers');
    }
}

(async ()=>{
    const sum = await new MyThenable(7,3);
    console.log(sum);
})();
```

בדוגמה הנ"ל יצרתי thenable שמטרתו הוא חישוב סכום של 2 מספרים חיוביים בצורה אסינכרונית אחרי שניה וחצי. בדומה לסינטקס של יצירת Promise חדש then מקבלת את resolve,reject על ידי JavaScript באופן אוטומטי, ולאחר שניה וחצי מחושב הסכום. יש לשים לב שבדומה ל Promise חייבים להפעיל פה את resolve או את reject. Await יודע לעבוד עם אובייקטים שהם thenable כלומר מכילים את הפונקציה:

```
    then(resolve,reject){

    }
```

בנוסף ניתן להגדיר פונקציהasync בתוך אובייקט:

```
class WithAsyncFunc{
    constructor(name){
        this.name = name;
    }

    async getNameAsync(){
        return Promise.resolve(this.name);
    }
}
(async ()=>{
    const obj = new WithAsyncFunc('Ofek');
    const name = await obj.getNameAsync();
    console.log(name);
})();
```

ניהול שגיאות Async\Await

כמו למשל במקרה הזה:await שאנחנו מחכים לו עם Promise עבור rejectבמידה ומופעלת

```
class WithAsyncFunc{

    constructor(name){
        this.name = name;
    }

    async getNameAsync(){
        return Promise.reject(this.name);
    }
}
(async ()=>{
    const obj = new WithAsyncFunc('Ofek');
    const name = await obj.getNameAsync();
```

```
    console.log(name);
})();
```

נזרקת שגיאה, הכוונה שזה שקול ל(throw new Error(the argument of reject . לכן ניתן להשתמש ב try&catch:

```
(async ()=>{
    try{
        const obj = new WithAsyncFunc('Ofek');
        const name = await obj.getNameAsync();
        console.log(name);
    }
    catch(e){
        console.log('Error '+e);
    }
})();
```

במידה ולא נרצה להשתמש ב try&catch נזכור שפונקציה async מחזירה Promise ולכן נוכל לרשום:

```
(async ()=>{

    const obj = new WithAsyncFunc('Ofek');
    const name = await obj.getNameAsync();
    console.log(name);

})().catch(x=>console.log('Error: ' + x));
```