**שקף 3**

בואו שנייה נחשוב על execution של פונקציות ב-javascript . זה נראה לנו מאוד טבעי שברגע שפונקציה מתחילה היא תרוץ עד שהיא מסיימת. זה ברור לנו שלפני שהפונקציה מסיימת את הריצה שלה לא יוכל להתבצע קוד JavaScript אחר.

עכשיו, בואו נסתכל על קטע הקוד הזה . שימו לב יש כאן לולאה מטורפת שמדפיסה את כל המספרים מאחד עד מספר ממש אסטרונומי.

יקח ללואה הזו המון זמן לרוץ הרבה יותר מmillisecond אחד.

ומה אם היה אפשר בעצם להפריע לריצה של הלולאה ? אז כאן אנחנו מתחילים להיכנס לכל הבעיות שיש בשפות מסורתיות של multi threading , שיכולים לגרום לנו לבלאגן טוטאלי בתוכנה.

למזלינו, אנחנו ב-javaScript , וכאן אין multi threading – רק פונקציה אחת היא executed בכל זמן נתון.

**שקף 4 :**

אז generators הם פונקציות שיכולות לעצור באמצע, פעם אחת או יותר, ויכולות להמשים יותר מאוחר, תוך כדי כך שהן מאפשרות לקוד אחר לרוץ בזמן שהן עוצרות.

ה-generator יכול להעצר רק מתוך הגוף של הפונקציה. כדי לעצור את ה-generator התווספה ב-ES6 מילה שמורה חדשה ב-JavaScript , המילה yield , וכאשר yield נקרא מתוך הפונקציה הפונקציה נעצרת. היא יכולה להמשיך לפעול רק כאשר קטע קוד חיצוני יקרא לה עם האופרטור next .

**שקף 5:**

אז בואו נסתכל על דוגמה בסיסית של generators .

קודם כל – שימו לב ש-generator מוגדר עם כוכבית. שימו לב לכוכבית ליד השם של הפונקציה.

ההפעלה של הפונקציה בעצם תחזיר לנו אובייקט חדש של generator , שאפשר להתחיל לתפעל אותו.

אז generators הם פונקציות רגילות של javascript , אבל ההבדל המשמעותי הוא במילה Yield בתוך הפונקציה. בכל פעם שיש yield ה-execution של הפונקציה יעצור.

שימו לב איך "שולטים" בפונקציה. בכל פעם שבה קןראים ל-next ה-execution של הפונקציה ממשיך מהמקום שבו הוא נעצר.

אז הפונקציה שאם רואים כאן מחזירה מספר ערכים ולא ערך אחד , כאשר בכל פעם שקוראים ל-next יוחזר ערך אחר.

מה שעוד חשוב להגיד, זה שאנחנו בעצם מבצעים את השליטה ב-generator מבחוץ של ידי בנייה של generator iterator .

ה-generator תמיד יחזיר אובייקט עם הערך שמוחזר על ידי ה-generator ובנוסף Boolean שמסמן האם ה-generator סיים את ה-flow שלו או לא.

**שקף 6:**

אם קראתם קצת חומר תיאורטי על concurrency או על threads , יתכן ששמעתם את המונחים cooperative ו-preemptive . cooperative אומר שה-process (או הפונקציה במקרה שלנו)בעצמה תבחר מתי היא "מרשה" שיעשו לה Interruption , כדי שהיא בעצם תוכל לתקשר עם קוד נוסף.

הקונספט הזה הוא מנוגד ל-preemptive , שבעצם אומר שה-process או הפונקציה במקרה שלנו תוכל להיות interrupted נגד רצונה.

ה-generators הם cooperative מהבחינה הזאת.

**שקף 7:**

העובדה שאפשר לעצור ולהתחיל את הgenerator היא לא רק נועדה לשליטה על ה-execution של ה-generator function , אלא היא מאפשרת העברת messages לשני הכיוונים , לתוך הgenrator ומהgenrator החוצה. עם פונקציות רגילות , מעבירים פרמטרים רק בתחילת הפונקציה, ומחזירים Return value רק ב-Return בסוף. עם ה-generator , אתה שולח message החוצה לכל yield , ואתה יכול לשלוח message בחזרה פנימה עם כל Restart .

**שקף 8:**

אז בואו נסתכל על קוד של generators שגם מבצע העברת messages .

אז מה קורא פה בעצם ? בפעם הראשונה שקוראים ל-next בעצם ה-generator מחזיר את הערך “foo” . באותה נקודה, ה-generator גם יכול לקלוט ערך אחר. שימו לב, שכאשר נקרא ל-next נעביר לו ערך 5, והמשתנה x יקבל את הערך 5 . ההדפסה של x+1 בעצם תדפיס 6 . שימו לב שהעברנו ערך לgenerator על ידי קריאה ל-next עם הערך.

**שקף 9:**

אז שימו לב מה קורה כאן. באתחול אנחנו מאתחלים את ה-generator עם הערך 5, כמו שאנחנו מעבירים פרמטרים לפונקציה רגילה. כאן בעזרת ה-statement : foo(5) אנחנו מאתחלים את x להיות 5.

בקריאה הראשונה ל-next אנחנו לא מעבירים כלום. למה ? בגלל שאין Yield כדי לקבל את מה שאנחנו מעבירים.

ה-yield (x+1) הוא מה ששולח החוצה את הערך 6.

אחר כך, next(12) שולחת 12 ל-statement yield (x+1) שמחכה, כך ש-y נהיה 12\*2 =24 . אז ה-statement הבא yield (y/3) משתערך ל(24/3) ושולח החוצה את הערך 8. ה-next השלישי next(13) שולח את הערך 13 ל-waiting statement yield(y/3) ושם Z יקבל את הערך 13 .

לבסוף , return (x+y+z) יחזיר 5+24+13 וזה מה שיחזור כ-value האחרון.

**שקף 10:**

אז אני מתארת לי שבשלב זה אתם קצת במצב של mind blending , ואתם רוצים לראות דוגמאות שימוש קצת יותר מציאותיות לדבר הז שנקרא generators . אז בואו נסתכל קצת על קוד שיש בו promises ונראה איך אפשר לגרום לו להראות יותר טוב עם generators .

**שקף 11:**

אז יש כאן דוגמת קוד עם promise chaining , הזכרנו בשיעור הקודם ש-promise chaining בעצם נותן לנו אפשרות לבצע פעולות בseries כאשר אנחנו מבצעים then על הpromise שחוזר. בואו נעבור על קטע הקוד כאן.

אנחנו מבצעים 4 בקשות get ל-url : <http://httpbin.org> , כאשר כל get כזה בעצם יחזיר לנו את ה-html שחוזר מה-request .

קודם כל השתמשתי כאן ב-promise api של bluebird , ולשם הנוחות, יש כאן שימוש ב-API שנקרא promisifyall , שעוטף את כל ה-API של ספריות מוכרות ב-promise . GetAsync הוא ה-API ש-promisifyAll מספק עבור ביצוע get request ומחזיר promise באופן out of the bix .

אז כאן יש לנו דוגמא לpromise chaining , שבו כל Request מתבצע אחד אחרי השני.

**שקף 12:**

עכשיו, בואו נסתכל איך כותבים את הקוד הזה עם generators .

קודם כל , שימו לב לשימוש ב-API של bluebird של promise.coroutine , שמחזירה לנו promise שמשתמש ב-yield כדי להחזיר את השליטה ל-generator ולעשות Invoke ל-promises אחד אחרי השני.

שימו לב כמה הקוד נהיה שטוח ופשוט ככה. זה קןד א סינכרוני אבל הוא נראה כמו קוד סינכרוני .מאחורי הקלעים יש עדיין Callbacks שקורים.

**שקף 13:**

אז עכשיו בואו נראה עוד קצת פונקציונליות מגניבה של generators .

**שקף 14:**

אז מה שאני רוצה להראות לכם כאן זה איך אפשר לטפל בשגיאות עם generators .

אז אחד מהדברים הכי powerful בgenerators של ES6 design , הוא שכאשר נבחינה סמנטית הקוד בתוך הgenerator נראה סינכרוני, מאחורי הקלעים אנחנו עובדים באופן א-סינכרוני עם callbacks ובקוד שעוטף את ה- generator הכל א-סינברוני – אז בעצם, הקוד בתוך ה-generator הוא קוד א- סינכרוני שנראה כמו קוד סינכרוני.

אז זה בעצם משפט קצת פלצני כדי להגיד לכם שאתם יכולים להשתמש ב-try catch כדי לטפל בשגיאות !! אתם יכולים להשתמש בטכניקות שאתם מכירים והן פופולריות.

**שקף 15:**

בואו נסתכל על קטע הקוד הבא :

יש כאן generator , שה-next הראשון שלו יחזיר לנו 3, ואז הוא מצפה לקבל ערך שהוא string – כי ל-string תהיה את הפונקציה toUpperCase . במידה ולא יתקבל כאן string יזרק error . ב-next השני המשתנה x אמור לקבל את הערך שיועבר מה-next ולעשות לו toUpperCase . אז כאן מכיוון שהעברנו int תיזרק שגיאה. שימו לב לקטע המגניב כאן – וזה הטיפול בשגיאות שממש נראה כמו קוד סינכרוני עם try catch כמו שאנחנו רגילים ואוהבים.

**שקף 16 :**

בואו נחזור לדוגמא שלנו עם ה-generators וה-promises , ונתמקד כאן על ה-error handling . יצרתי כאן caller function שהוא גם generator , שקורא לfoo – שהוא ה-generator שמבצע את כל ה-requests . שתוך ה-calling function נמצא הטיפול בשגיאות שנראה כמו שאנחנו מכירים ואוהבים – עם try catch . מה שיקרה כאן זה שאם Request אחד יכשל foo תפסיק את הביצוע שלה וה-calling function תטפל ב-exception .

**שקף 17:**

אז מהם הדברים שהכי הכי כדאי לעשות עם generators ?

**שקף 18**

בואו נדבר על lazy evaluation . אחרי ש-yield נכנס ל-ES6 זה הרבה הרבה יותר קל לעשות lazy evaluation . על ידי זה שאנחנו עושים suspend ל-execution ו-resume בשלב יותר מאוחר , אנחנו יכולים לאסוף את הערכים רק כשנצטרך אותם.

שימו לב לדוגמה הבאה – fibGen שכאן עובדת כ-lazy וזה היתרון שלה – אנחנו יכולים למשוך את הערכים רק כשאנחנו רוצים.

**שקף19 :**

אז בואו נדבר על loops אינסופיים – נכיוון שאנחנו יכולים להיות lazy , אפשר לייצר sequences אינסופיים מאוד בקלות וגם בצורה יעילהץ יש כאן דוגמה ל-generator של ה-fibonachi sequence שהוא אינסופי לדוגמה. שימו לב לשימוש בפנים של while true , וניתן להשתמש ב-next באופן אינסופי.

**שקף 20:**

הpattern האחרון שאפשר לדבר עליו הוא control flow של פונקציות א-סינכרוניות. אז בעזרת generators אנחנו יכולים להגיע לקוד שממש נראה סינכרוני גם מבחינת syntax וגם מבחינת error handling, למרות שמאחורי הקלעים הכל נראה שינכרוני. ראינו דוגמה כזאת בעזרת ה-promises מוקדם יותר, אבל ישנן עוד טכניקות לעשות את זה.